
Version 2024 Q1

RASON Modeling Language

User Guide

Copyright

Software copyright 1991-2024 by Frontline Systems, Inc.

User Guide copyright 2024 by Frontline Systems, Inc.

Trademark

RASON® and Analytic Solver® are registered trademarks of Frontline Systems, Inc.

Patent Pending

Systems and Methods for Automated Risk Analysis of Machine Learning Models.

Contact Us

Contact Frontline Systems, Inc., P.O. Box 4288, Incline Village, NV 89450.

Tel (775) 831-0300 Email info@solver.com Web <https://RASON.com>

Table of Contents

Copyright	2
Trademark	2
Contact Us.....	2
Introduction	9
What's New: Frontline Solvers V2023 Q3	9
AI Agent: Ask for Help from ChatGPT “Trained on Analytic Solver”	9
Identify Inputs: Easily Set Up your Model for Data Updates and New Solves	10
What's New: Frontline Solvers V2023 Q1	11
Faster LP/Quadratic Solver and Large-Scale LP/QP Solver Engine.....	11
More Plug-in Solver Engine Improvements	11
Greatly Improved “Deploy Your Model to Teams” Capability	11
Risk Analysis of Machine Learning Models Created in Other Software.....	11
What's New: Frontline Solvers V2023.....	12
Model Management and Run Monitoring	12
Automated Risk Analysis of Machine Learning Models.....	12
What's New: Frontline Solvers V2022.....	13
Faster Interaction, Faster Solves.....	13
What's New: Frontline Solvers V2021.5	14
Automate Data Mining with Find Best Model.....	14
Better Simulation Models with Metalog Distributions and Fitting	14
Share Data Mining and Probability Models via RASON	14
And of Course, Optimization Enhancements.....	14
Analytic Solver Decision – Replaces Analytic Solver Upgrade	14
Analytic Solver Learning – Replaces Analytic Solver Basic.....	15
What's New: Frontline Solvers V2021	15
Lambda, Let and Box Functions.....	15
Let's Get Started!	16
What's New: Frontline Solvers V2020.5.....	16
Direct Use of Excel Models	16
An Example Decision Flow.....	16
Scheduled Model Runs	17
Key Benefits	17
Overview of Features.....	19
An Optimization Example Model.....	19
A Monte Carlo Simulation Example Model	20
An Example Data Science Model.....	21
An Example Decision Table.....	22
Web and Mobile Applications	23
Server –Based Applications	24
Binding to Data and Dimensional Flexibility	24
RASON Subscriptions	27
Introduction.....	27
Registering	27
My Account	29
RASON Licenses	29
AI Agent	29
My Account Overview Page	31
A Note on Throttling Limits.....	32

My Account: Data Connections and Creating API Tokens.....	33
Data Connections Tab.....	33
Creating API Tokens	39
Basic API Tokens	39
OAuth2 API Tokens	40
Runtime API Tokens	40
Creating OAuth2 API Tokens.....	41

RASON Services Web IDE 45

Introduction	45
RASON Models	45
RASON Editor Page.....	52
Running Example Models on the "Editor" Page	55
Viewing Results in ODATA	63
Results Formatting in Web Editor	63
Uploading Data from an External Data File	68
Using Query Parameters	73
Using the Chart Wizard	74
Example: Using the Chart Wizard to create a Line Chart	77
Example: Using the Chart Wizard to create a Bar Chart.....	81
Chart Options	83
Solving a Data Science Model	87
Solving a RASON Model using a Runtime Token	96

Creating and Running a Decision Flow 98

What is a Decision Flow?	98
RASON Decision Flow Editor	98
Getting Results	102
Adding Input and Query Parameters in the Editor.....	103
API Parameters.....	107
Model Data Parameters.....	109
Model Data Source Parameters	110
Model Input Parameters	112
Solving An "In-Line" Decision Flow and Display Results	113
Using the OData Endpoint	117
Solving a Decision Flow with Reusable Models.....	122
1 st Stage Reusable Model.....	122
2 nd Stage Reusable Model	124
Creating the Decision Flow.....	125
Solving the Decision Flow	130
Data Science Decision Flow Example.....	131
Reusable Partitioning Model.....	132
Scheduling.....	135
Reusable Rescaling Model.....	137
mlrModel Reusable Model.....	144
Invoking an Excel Model in a Decision Flow	157
Decision Flow using SQL Server.....	160
More on Explicit and Automatic Scheduling	168
Explicit Scheduling.....	168
Automatic Scheduling.....	168
More on the Model Type Property	169

Defining Your Optimization Model 170

Introduction	170
--------------------	-----

Defining an Optimization Model.....	170
Setting Up a Model.....	170
The Basic Model.....	171
The Improved Model	173
Using A Simple For() with Index Sets and a Table Assignment	173
Using the Binding Property.....	175
Running an Optimization	176
Defining Your Simulation Model	177
Introduction.....	177
Defining a Simulation Model.....	177
Setting Up a Model.....	177
Running a Simulation	181
An Airline Management Revenue Model.....	181
Changing Parameter Values	184
Multiple Parameterized Simulations	185
Using a Probability Model within RASON.	190
Sensitivity Analysis and Model Parameters	194
Introduction.....	194
Parameter Identification.....	194
Using Sensitivity Parameters	196
How Parameters are Varied.....	196
Varying All Parameters Simultaneously	197
Varying Two Parameters Independently	198
Defining Your Stochastic Optimization Model	201
Introduction.....	201
Defining a Stochastic Optimization Model.....	201
A Project Selection Model	201
Solving with Simulation Optimization.....	204
Stochastic Transformation using Deterministic Equivalent.....	205
Defining Your Data Science Model	206
Introduction.....	206
Supported Features	206
Defining a Data Science Model.....	208
Essential RASON Data Science Model Sections	210
"datasources"	210
"datasets"	211
"estimator"/"transformer"	211
"actions"	213
POSTing/Exporting Fitted Models.....	217
Optional RASON Sections.....	219
"data"	219
"fittedModel"	220
"preProcessor"	221
"weakLearner"	222
Example RASON Models.....	222
Sampling Example.....	222
Partitioning Example	225
Example Performing Feature Selection.....	228
Find Best Model for Classification and Prediction	232

Synthetic Data Generation	248
Linear Regression Simulation Example	264
Data Science Risk Analysis Decision Flow	269
Scoring Examples	274
Perform Risk Analysis Given Fitted Model Stored in PMML/JSON Format	278

Custom Type Definitions 281

Introduction	282
Data-source Binding	282
Custom Types in RASON	283
Custom Type Specifications	284
Using Type Definitions with Box Functions	285
A Simple Box Function Example	286
Posting and Solving the Model	286
Box Function Example with Type Definitions	288
Using Type Definitions with Decision Tables	291
Decision Table with Type Definition with Allowed Values	292
Advanced Features: Type Definitions with Collections	294
Computing Specific Results	296
Negative Indexing	296
Clause Selection	297

Defining Custom Functions 300

Introduction	300
Using the Lambda Function	300
Lambda Function Defined	300
Optimization Example	300
Posting and Solving the Model	303
Simulation Example Using a Lambda Function	304
Posting and Solving the Model	305
Elements of a Box Function	306
Invoking the Function in the RASON Model	309
Full RASON Optimization Model	310
Posting and Solving the Model	312
Simulation Example Using a Box Function	313
Posting and Solving the Model	315
Creating independent DMN/Feel models	317

Defining Decision Tables in RASON 319

Introduction	319
Decision Table Structure	320
Optional Arguments	326
Decision Table Hit Policies	326
Decision Tables with Dates and Times	334
Using an External Data Source	339
Loan Strategy Example	342
Using a PMML	350
Parametric Selection Feature	354
Using PMML Model to Score New Data	358
Merging Decision Table Results	360
Decision Table Using DMN Conformance Level 3	363
Formatting results of box objects as custom types	363

Defining Contexts in RASON	366
Introduction: Relation to Custom Types	366
The Context Definition	366
FEEL and DMN RASON Examples	367
Using the REST API	369
Introduction	369
RASON REST API	369
Authorization Headers	369
Responses: Solution Status and Results	373
CPU Time Limits and Charges	374
Throttling	374
Synchronous REST API Endpoints (Quick Solve)	374
Asynchronous RASON REST API Endpoints	405
RASON Decision Flow Endpoints	563
RASON OData Endpoints	584
OData Service for RASON Decision flows	601
Introduction	601
Example	601
Example Results	606
Querying	608
Paging	610
Using OData in Power BI	610
Creating Your Own App	615
Introduction	615
Creating Your Own Web Application	615
Uploading your RASON Model to Power BI	617
Installing Power BI	617
Creating a Custom Visual from a RASON Model	618
Scoring New Data within Power BI	627
Advanced Settings	631
Uploading your RASON Model to Tableau	632
Solving RASON Models in Tableau	632
Viewing the Tableau Extension File	635
Starting up the Server	636
Opening Extensions within Tableau	636
Scoring New Data within Tableau	641
Using Array Formulas, For Loops and Tables in RASON	647
Introduction	647
Array Formulas	647
Parallel array formulas	647
Non-parallel array formulas	648
Indexed array formulas	648
Indexed array formulas (advanced)	649
Arrays and tables in indexed array formulas	650
Examples with Indexed Array Formulas	651
Using For Loops	654
Examples using Loops	654
Using Statements	655

Using Tables	656
Creating a Sparse Table	661

RASON Power Apps Tutorial 669

Calling RASON Services from Power Apps.....	669
Airline Crew Scheduling Excel Model.....	669
Deploying the Excel Model to Rason Cloud Services.....	670
Entering Data in SharePoint.....	671
Creating the Power App	677
Creating the Decision Flow	686

Glossary of RASON Property Names 702

RASON Property Names Glossary	702
-------------------------------------	-----

RASON Organization Accounts 723

Introduction	723
RASON Organization Accounts.....	723
RASON without Organization Accounts	723
RASON with Organization Accounts	723
Instructions for Administrators	724
Preparing Azure Storage for use with your RASON Organization Account.....	724
Instructions for End Users	729
Using a RASON Organization Account.....	729
Posting a Model.....	730
Working with Containers	731
Working with Decision Flows.....	732
Working Outside of RASON Editor	732
Introduction	734
Problem Definition	734
Building the RASON Model	734

Introduction

Welcome to Frontline Systems' RASON® Decision Services. RASON is an Azure-hosted cloud service that enables companies to easily embed 'intelligent decisions' in a custom application, manual or automated business process, applying the full range of analytics methods – from simple calculations and business rules to data science and machine learning, simulation and risk analysis, and conventional and risk-based optimization.

RASON Decision Services can be used from nearly any application, via a series of simple REST API requests to <https://rason.net>. To express the full range of analytic models, RASON includes a high-level, declarative modeling language, syntactically embedded in JSON (JavaScript Object Notation), the popular structured format widely used in Web and mobile applications. RASON results appear in JSON, or as more structured OData JSON endpoints.

You can use RASON to quickly and easily create and solve optimization, simulation/risk analysis, data science, decision table, and decision flow models. RASON works with Windows and Linux desktops and servers, but is especially useful if you are building Web or mobile applications. You can learn RASON, create models, supply data and solve them, and even manage model versions and cloud data connections, “point and click” using <https://RASON.com>, Frontline's “web portal” to the underlying REST API service.

If you've used another modeling language to build an analytic model, you'll find the RASON language to be simple but powerful and expressive – and integrating RASON models into a larger application, especially a web or mobile app, is much easier than with other modeling languages. If you've used Excel to build planning or analytic models, you'll find that RASON includes virtually the entire Excel formula language as a subset. If you've used tools based on the DMN (Decision Model and Notation) standard, you'll find that RASON (and Analytic Solver® for Excel) include full support for DMN and FEEL Level 2.

RASON Decision Services also includes comprehensive *data access* support for Excel, SQL Server on Azure, Power BI, Power Apps, Power Automate (aka Microsoft Flow) and Dynamics 365. And it includes powerful model management tools, such as tracking model versions including “champions and challengers”, monitoring model results, and automated scheduling of model runs.

Unlike existing “heavyweight” Business Rule Management Systems, with year-long implementation schedules, six-figure budgets and limited analytics power, RASON Decision Services enables you to get results in just weeks to months, from building and testing models, to deploying them across an organization. With RASON, you can build successful POCs (Proofs of Concept) without any IT or professional developer support – yet RASON is very “IT and developer friendly” when you're ready to deploy your POC across your company.

What's New: Frontline Solvers V2023 Q3

In Analytic Solver V2023 Q3, we've made it easier than ever to create analytic models, and easier to deploy them for ongoing use, through two major new features: our new conversational “AI Agent”, appearing as a tab on the RASON.com navigational bar and as a button next to “Help” on the Analytic Solver Ribbon, plus a new “Identify Inputs” feature that simplifies the task of updating key input parameters when it's time to re-run your model.

AI Agent: Ask for Help from ChatGPT “Trained on Analytic Solver”

By now, nearly everyone who's been following developments in software is aware of ChatGPT, the conversational agent using “Generative AI” methods, developed by the OpenAI nonprofit closely affiliated with

Microsoft. You can ask ChatGPT questions on almost any topic – including analytic methods – and get meaningful and interesting (though not always 100% correct!) answers.

That's great – but what if you could have a ChatGPT “technical support agent” that had studied all 2,300 pages of Frontline's User Guides (including this one), Reference Guides and QuickStart Guides – everything about optimization, Monte Carlo simulation, data science and machine learning, and business rules – and was instructed to use its knowledge to answer your questions? Well, that's what you have in Analytic Solver V2023 Q3!

Actually making this work involves a fair amount of software engineering: Generative AI tools need the right “context” to respond to your question – so we've built an online resource where all those 2,300 pages of Guides are represented via “vector embeddings” that enable searches by meaning, not just “keyword matches”. Our AI Agent automatically searches this online resource to create “context” for your queries to ChatGPT – and you can also search this online resource directly.

We're using the “real” online ChatGPT 3.5 Turbo version for its full conversational capabilities, which costs us money for every query – so you will find some limits placed on query length and number of queries per month, depending on your software license type. But we expect you'll be able to use our AI Agent to amplify your own efforts, ultimately building more capable and effective analytic models. We look forward to your feedback!

Identify Inputs: Easily Set Up your Model for Data Updates and New Solves

When you're first building and solving a model in Analytic Solver, usually your focus is on getting a solution for a specific instance of a business problem, with data you have today – gathered from one or several external sources into Excel. After some effort, you're now getting solutions from optimization, simulation, or data science and machine learning. That's a success by itself ... but it leads to a desire to “do it again (and again) in the future.”

But ... those data and parameter values that *should be updated* for a future run may be scattered around your Excel spreadsheet, which also includes cells with calculated formulas, cells with constant data that doesn't need updating, and cells with data that you want to be visible, but that doesn't actually affect or “participate in” the analytic model. How do you find *just* the parameter values that *should* be updated – and then, how do you *actually update* them?

Now you can choose **Tools – Identify Inputs** and get help doing exactly that. This tool uses our PSI Interpreter to scan your model formulas, identify and list only those cell ranges that are candidates for data updates that will affect the model when solved. You can then pick and choose from a list of cell ranges, which ones you actually need to update for a new run.

Then you can either (i) let Analytic Solver “highlight” those cells with colors and backgrounds, or even better, (ii) automatically add calls to our PsiInput() function that reference those cells. These cells will automatically appear in the Task Pane as part of your model (under the **Input Data** heading) – but when you deploy your model for use *beyond Excel*, via our RASON and Solver SDK tools, you'll have easy ways to supply new values for exactly those input cells.

If you've planned in advance for data updates and re-solves, Tools – Identify Inputs can help you check and validate your work. But if – like most of us – you were focused on getting a first-time solution and ‘put off’ the task of data updates and re-solves, Tools – Identify Inputs can be a huge time-saver!

What's New: Frontline Solvers V2023 Q1

We released Analytic Solver V2023 in September 2022, with the innovative enhancements described above – but we surprised even ourselves by how much more we could offer our customers by December 2022! So we’ve christened this new release “Analytic Solver V2023 Q1”.

Faster LP/Quadratic Solver and Large-Scale LP/QP Solver Engine

Our “headline feature” for this release is a new, higher performance version of the built-in LP/Quadratic Solver, as well as its “cousin” the Large-Scale LP/QP Solver Engine which has much higher problem size limits. We expect most users will see **faster – sometimes *much* faster** solutions with this new version, both for LP and QP (linear programming and quadratic programming) models, and for LP and QP models with integer constraints.

And there’s more: Now you can solve models with (convex) **quadratic constraints**, with or without a quadratic objective, using this Solver Engine.

(There are always tradeoffs: We’ve found that for about 10% of models we’ve tested, the new version is not faster, and can even be *slower* for models with integer constraints. If you encounter this, just select the **Classic Search** option to get exactly the same performance you were getting before.)

More Plug-in Solver Engine Improvements

V2023 Q1 also includes new versions of other large-scale Solver Engines. The **Gurobi Solver Engine** is upgraded (to their V10.0) with a range of enhancements, yielding solution times faster by 3-10% to 25% on a wide range of linear and quadratic mixed-integer models. The **Xpress Solver Engine** is upgraded (to their V9.0) with enhancements to strong branching, separation of cutting planes, and a new heuristic method, run at the branch & bound root. The **KNITRO Solver Engine** is upgraded (to their V13.2) with a range of enhancements for smooth nonlinear mixed-integer models, including new presolve, cut selection and heuristic methods.

Greatly Improved “Deploy Your Model to Teams” Capability

We learned from surveys that a large majority of our customers work in companies using **Microsoft Teams** – so we’ve significantly enhanced a feature introduced in Analytic Solver V2022, that makes it easy to **share** your Excel model results with colleagues in your company, using Teams. The V2022 feature was designed to share your (entire) model, but we realize that many users want or need to share just the **model results** – not the full model with all its optimization and/or simulation model elements.

In V2023 Q1, when you choose **Deploy Model** from the Ribbon and click **Teams – Teams Report**, Analytic Solver will automatically create a new workbook holding only **model results**, with external links to your **model** workbook. You can choose exactly which optimization and/or simulation results you want to include in this workbook. The new workbook will be saved online, and made available to the users you want, through a “Teams channel” that you select. Your colleagues, using **just Teams**, will be able to open the workbook and view, copy or work with the results you’re providing. And perhaps the best part: When you **re-run** your optimization or simulation model with new data, the workbook in Teams will be **automatically updated** (via those external links) with the latest model results! See the chapter “Deploying Your Model” in this Guide for full details.

Risk Analysis of Machine Learning Models Created in Other Software

Analytic Solver V2023 introduced an innovative (and patent pending) new capability for **automated risk analysis** of **machine learning** models – see “What’s New in Analytic Solver V2023” above for a complete summary, and see our Data Mining User Guide for full details. But we realize that many people create and test machine learning models using other software. Those folks just don’t have the ability to quantify how their ML models **may perform differently** on data they will encounter in the future ... until now. But **you can help them**, using Analytic Solver V2023 Q1 (or using RASON V2023 Q1, if they prefer to use a cloud platform).

In V2023 Q1, we've made it easy to perform **automated risk analysis** of models **created in other software** and saved in **PMML** (Predictive Modeling Markup Language) format. PMML is an open standard that is widely supported by software for machine learning. Analytic Solver and RASON will also save a trained machine learning model in PMML form. But now you can bring a PMML model into Analytic Solver, plus some of the data you used to train the model (just copy the PMML text and the data onto worksheets) – then use simple menu options to quickly get insights into the future performance of this ML model, from our **automated risk analysis** methods.

There are plenty of other small enhancements and fixes in V2023 Q1 – and we have more new features “in the works”. But you can see why we felt we were ready to deliver another major version to our customers – in time for Christmas!

What's New: Frontline Solvers V2023

Model Management and Run Monitoring

As you or your company make further use of RASON, you'll build up a set of RASON models for optimization, simulation, data mining and business rules, fitted machine learning models, probability models, and even Excel workbook models, that are maintained in the RASON Service and run periodically to compute new optimal solutions, new risk analyses, or make new predictions and new rule-based decisions. Managing these models as valuable intellectual property assets, and monitoring their performance, will become more important over time.

The new **Models tab** in the RASON IDE in V2023 gives you easy access to a great deal of information about each of your models, including properties such as model name, type, version and date last modified, and metrics such as total run time, number of runs, runs ending in an error, etc. The Models tab makes it easy to view your models filtered, sorted and grouped by type, data connection use, and other properties, and run outcomes and metrics such as solve time, over your choice of time intervals – and to export this data for further analysis.

Automated Risk Analysis of Machine Learning Models

RASON V2023, our latest release, features an innovative (and patent pending) new capability for **automated risk analysis** of **machine learning** models. A further benefit of this feature is a general-purpose, easy to use new tool for **synthetic data generation**, to augment the data you already have. RASON Data Mining and Comprehensive users are able to use these new features with size limits constrained only by memory, but *all* RASON users have access to these features with “Basic size limits” for your datasets.

Until RASON V2023, data science and machine learning (DSML) tools – including ours – had no facility for **risk analysis** of machine learning (ML) models, prior to their production use. Most tools (including ours) had facilities for ‘training’ the model on one set of data, ‘validating’ its performance on another set of data, and ‘testing’ it versus other ML models on a third set of data. But this is not *risk analysis*.

Synthetic data generation has come into use in recent years to augment available datasets, when the available data is limited, or is restricted by law or regulation, such as with personal health information (PHI). In RASON V2023, you have a powerful, general-purpose **transformer** named "syntheticDataGenerator". So far, we're keeping our tools current with the “state of the art”.

But with the new **simulation** option available for every **estimator** you define in RASON V2023, we've gone *beyond* the “state of the art”, to bring you a new way to assess your trained ML model's performance – not to determine how well it **has performed** on data you have, but to quantify how it **may perform differently** on data it will encounter in the future. We generate synthetic data “on the fly” and use it in a Monte Carlo simulation of your ML model's performance – and we highlight **differences** in model performance in training versus simulated production use.

The beauty of this approach is that **you don't have to do any work** to obtain a risk analysis of your model's performance, beyond including the **simulation** option when defining your estimator, and including your choice of new evaluations such as **simulationPrediction** as an action to be performed – the risk analysis is entirely

automated. You don't even have to be familiar with the Monte Carlo simulation features of RASON to use this capability for machine learning!

(You *can* specify your own choices for distribution fitting, correlation and copulas, Monte Carlo sample generation, etc. – but the default options work very well.) Typically, the analysis adds only **seconds** to a perhaps a **minute** to the time taken to train and validate your ML model. So you can make this a routine **part of the process** of training and assessing new ML models.

As noted earlier (opposite the title page) in this User Guide, we have a patent application pending titled “Automated Risk Analysis of Machine Learning Models”. But as a RASON (or Solver SDK or Analytic Solver) user, you gain first access to this new capability at no extra cost.

What's New: Frontline Solvers V2022

Faster Interaction, Faster Solves

In Analytic Solver V2022, Excel startup is **faster**, and your ‘regular interaction’ with Excel is **faster** when Analytic Solver Desktop is loaded. Dialog rendering is improved for users with very-high-resolution monitors, as well as monitors with limited vertical depth.

Most impactful, the process of interpreting the model – when the model type is diagnosed and when “Setting Up Problem...” appears in the Task Pane status bar – is **faster** for most models, and *significantly faster* for larger models, especially on models with very deep “chains” of formulas that depend on other formulas. We’re also including new, **faster** versions of the Gurobi Solver for linear mixed-integer models, and the Knitro Solver for nonlinear models.

V2022 re-introduces the **Freeze** and **Thaw** options (now located on the Task Pane Tools tab), which allow you to share Analytic Solver models containing PSI function calls with users who don't have Analytic Solver installed. (“Freeze” will save PSI function call formulas in cell comments, and “Thaw” will restore them later as formulas.)

Deploy Model functionality in V2022 is enhanced on our RASON cloud server, and **Microsoft Teams** messaging is improved for several account types.

In case you didn't know: Recent Analytic Solver releases, including V2022, allow you to **Test Run** models that *exceed* the limits of your current license. (You don't get full results for all variables and constraints, but you do get the final objective value and solution time.) You can even use optional large-scale Solver Engines, like the Gurobi and Knitro Solvers, in a “Test Run”. To make it clear what your current license does and doesn't include, in V2022 you'll see “(Test Run)” next to the names of optional Solver Engines in the Task Pane.

What's New: Frontline Solvers V2021.5

Automate Data Mining with Find Best Model

Analytic Solver, our SDKs and RASON include comprehensive, powerful support for data mining and machine learning. Using these tools, you can “train” or fit your data to a wide range of statistical and machine learning models: Classification and regression trees, neural networks, linear and logistic regression, discriminant analysis, naïve Bayes, k-nearest neighbors and more. But the task of choosing and comparing these models, and selecting parameters for each one was up to you.

With the new Find Best Model options in V2021.5, you can automate this work as well! Find Best Model uses methods similar to those in (expensive high-end) tools like DataRobot and RapidMiner, to automatically choose types of ML models and their parameters, validate and compare them according to criteria that you choose, and deliver the model that best fits your data.

Better Simulation Models with Metalog Distributions and Fitting

Analytic Solver, our SDKs and RASON support over 60 “classical” probability distributions for Monte Carlo simulation. Since mid-2017, they’ve also supported the increasingly popular Metalog family of distributions, created by Dr. Tom Keelin, and recently popularized by the nonprofit Probability Management group. Metalog distributions can closely approximate virtually any classical continuous distribution, and often they can better fit user data than classical distributions. In V2021.5 we’ve brought Metalog distributions to the fore, with a powerful new facility to automatically fit user data to the full range of possible (bounded and unbounded, multi-term) Metalog distributions. It’s never been easier to get an accurate probability distribution that fits a real-world phenomenon.

Share Data Mining and Probability Models via RASON

In recent years, our Azure-hosted RASON Decision Services platform has offered increasingly powerful facilities to deploy models to the cloud and share them with other users – culminating in Frontline Solvers V2020.5, when we enabled deployment, sharing, versioning and management of Excel models as well as native RASON models, plus support for multi-stage decision flows, encompassing and going beyond traditional data science workflows.

In V2021.5 we’ve gone further: You can now deploy and share data mining and machine learning models, trained in Analytic Solver or RASON, to the Azure cloud, and use them directly for classification and prediction (without needing auxiliary “code” in R or Python, RASON or Excel). You can also deploy and share probability models, following the open Probability Management 3.0 standard. Using these Shared Information Probability resources (SIPs, also known as “Stochastic Information Packets”), you can ensure that your group or organization uses consistent data about uncertain/risky variables across simulation or decision models, enabling model results to be meaningfully compared.

And of Course, Optimization Enhancements

Analytic Solver, our SDKs and RASON have always offered rich support for conventional and stochastic optimization, improved in every new release. V2021.5 is no exception: We’ve made PSI Interpreter enhancements to better utilize main memory in large optimization models (with 1 million or more

decision variables) in all three product lines. In our V2021.5 release, we’re also shipping the latest Gurobi Solver 9.1, Xpress Solver 37.1.3.0, and new KNITRO Solver V12.4 with each of our products.

Analytic Solver Decision – Replaces Analytic Solver Upgrade

Analytic Solver is our comprehensive tool for predictive, prescriptive and decision analytics. In recent years we’ve added deep support for business rules and decision tables, following the DMN standard, to Analytic

Solver, and similar facilities in RASON. Since we've matched some entire products offered by competitors in the "rules" or "decision management" world, we've brought together our decision trees, DMN business rules and decision tables, and DMN Box functions, and combined them with Analytic Solver Upgrade (for optimization models) to create a new product, Analytic Solver Decision – which offers "more for the same price" (\$995 first year) compared to Analytic Solver Upgrade.

In Excel, you'll see a new icon group on the Analytic Solver Ribbon Decision Model, next to the Optimization Model and Simulation Model groups. You'll also see the Tools group options moved to the Task Pane (already true in Analytic Solver Cloud version). Analytic Solver Decision will still be included in Analytic Solver Comprehensive and in a full RASON license, but its DMN-related and optimization features will have size limits in Analytic Solver Simulation and Analytic Solver Data Science.

Analytic Solver Learning – Replaces Analytic Solver Basic

We've found that many purchasers of Analytic Solver Basic, who lacked a previous analytics background and didn't also purchase learning aids such as our Solver.Academy courses, experience limited success and often don't renew their licenses. In V2021.5 we are replacing Analytic Solver Basic with Analytic Solver Learning – a bundle of Analytic Solver Basic and all four current Solver.Academy courses. This would be about \$1,100 at our best current prices, but we'll offer Analytic Solver Learning at \$500 per year or just \$49.95 per month.

What's New: Frontline Solvers V2021

The latest release of RASON Decision Services gives you new power to define your own **custom functions**, in a way that works in the RASON modeling language *and* in both Excel Desktop and Excel for the Web.

Lambda, Let and Box Functions

For Excel users: In years past, you probably used VBA (Visual Basic for Applications) to define custom functions. While this still works in Excel Desktop using COM (28-year-old Component Object Model), VBA functions are *not* supported in Excel for the Web – and according to Microsoft, VBA and COM will *never* move to the cloud. If you want your custom functions to work in *both* desktop and cloud, your options have been limited – until now:

- Microsoft has introduced new Excel functions LAMBDA and LET. These are very special because you can use them in Excel formulas to define your own custom functions. The Excel community has expressed much excitement over these new functions, since they effectively make Excel a "complete programming language". (In Q1 2021, these functions are being rolled out across the different Office update channels.)
- On another front, there's the open standard known as DMN (Decision Model and Notation) – a business user-friendly "formula language" used to define business rules and decision tables, supported in "decision management" platforms from various vendors, and in Analytic Solver and RASON since 2019. DMN – now in version 1.3 – offers a way to define your own custom functions, known as "Box functions".

RASON V2021 and Analytic Solver V2021 include support for *both* Excel's LAMBDA and LET functions, and for DMN-compatible Box functions. For a tutorial on how to use both approaches, please consult the *Frontline Solvers User Guide*. You'll find a new chapter in this guide, "Defining Custom Functions", that explains how to use both LAMBDA and LET, and DMN Box functions. Even better, both of these approaches enjoy full support from our PSI Interpreter – which means that our full range of Solver Engines, and our high-speed Monte Carlo simulation engine "understand" and take full advantage of custom functions that you define this way. This can yield better results than you've ever had with VBA-based functions that are embedded in an optimization or simulation model.

Also new in RASON V2021 is significantly enhanced support for **fitted models** – forecasting and machine learning models that have been created (in either RASON or Analytic Solver) by using past data to fit

parameters of an equation, classification or regression tree, or neural network. In V2021, fitted models are now “first class objects” that can be referenced by name, tracked through multiple versions, and used in “champion” versus “challenger” tests.

Let's Get Started!

See the chapter *Creating and Running a Decision Flow* to see how easy it is to create and solve a decision flow using RASON Decision Services.

What's New: Frontline Solvers V2020.5

RASON began as an Azure-hosted cloud service in 2015. At that time, the RASON Analytics API offered an easy way to create and solve advanced analytics models “in the cloud” and use the results in web and mobile apps – only. The ideal “RASON user” was a modern web developer working in JavaScript. RASON focused on solving “model instances” when models and datasets were submitted through its REST API; it assumed that the developer would manage models and datasets and monitor results outside the service.

Today, RASON Decision Services is much more – a complete platform for creating, deploying and managing analytic models. While RASON is still fully usable by web developers as before, the service now aims to support a modern business analyst with low-code or no-code options. RASON Decision Services offers facilities to manage both RASON models and “fitted” data science and machine learning models with versioning, and monitoring results via query-based reporting. RASON Decision Services offers rich facilities for using Microsoft cloud-based datasets, in OneDrive for Business and the Common Data Service that underlies Power Apps, Power Automate and Dynamics 365. It also features security enhancements such as OAuth2-based methods for request authentication, Azure vaults for data access credentials and 24x7 active monitoring of the RASON public server for suspicious activity.

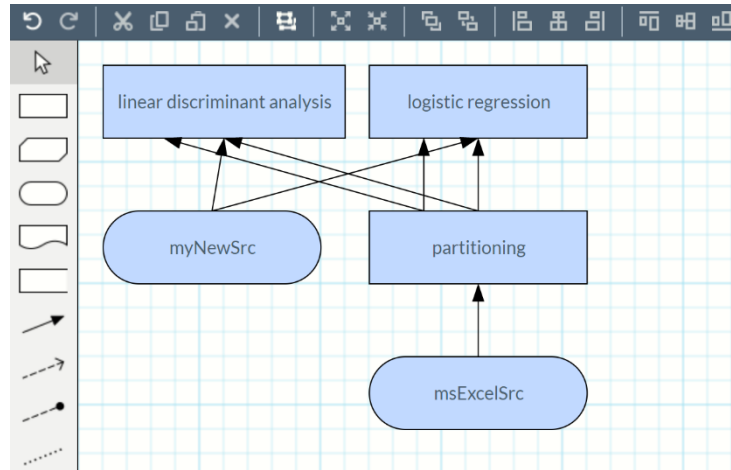
Direct Use of Excel Models

Perhaps the most exciting feature of RASON Decision Services for business analysts is its full support for Excel analytic models, created with Frontline’s Analytic Solver software for Excel, “on a par” with models written in the RASON modeling language. Although an Excel user can certainly learn and master RASON, and Analytic Solver software can automatically translate many Excel models into RASON, there are plenty of scenarios where it’s desirable to *maintain* an analytic model in Excel – so a business analyst who has domain expertise but isn’t a “coder” can continue to enhance the model.

With Frontline’s latest RASON release, an analyst can create and test models, deploy them “to the cloud” – point and click – as full-fledged RESTful decision services, and even get reports of recent runs of his/her decision services, all without leaving Excel. It’s easy enough to transition between Analytic Solver in Excel and the web portal at <https://RASON.com>. You can not only deploy your Excel workbook model as a self-contained cloud services – you can even embed your Excel workbook in a multi-stage “decision flow” – discussed below – that can combine SQL, RASON, Excel, and DMN models, easily passing rich results from stage to stage.

An Example Decision Flow

Perhaps the most advanced feature of RASON Decision Services is its decision flow engine, which enables you to define a multi-stage decision flow of SQL operations, data mining and machine learning, predictive model fitting and/or scoring, optimization and/or simulation model runs, business rules and decision tables, executing all or part of the decision flow with a single REST API call. Decision Flows may be created “by hand” or quickly assembled “point and click” using the new RASON Decision Flow Editor.



Further, the decision flow engine creates both JSON responses and dynamic OData endpoints for the results of each stage. This enables RASON *analytic model results to be queried as data*, and consumed by a wide range of other tools, such as Power BI and Power Apps.

Scheduled Model Runs

Training a machine learning model, or solving a complex optimization model can sometimes take hours of compute time, whereas scoring a new case for a machine learning model or retrieving selected results from an optimization model may take seconds. With RASON Decision Services, you can schedule a decision flow to **run at fixed intervals** or even simpler, specify **how recently updated** you want each stage to be – RASON will automatically determine when to run each stage.

Properties	
Name	partitioning
Type	Data Mining
Recency	PT12H
Comment	Partition data source every 12 hours

Key Benefits

Embracing Open Standards

Designed to be usable by business analysts and domain experts, not just data scientists and IT professionals, RASON Decision Services offers easier model creation and deployment by embracing two open standards: DMN (Decision Model and Notation) and its FEEL (Friendly Enough Expression Language) from the Object Management Group – now a widely used alternative to proprietary business rules languages and ODATA (Open Data Protocol), an ISO/IEC approved OASIS standard for RESTful data access – now a widely used alternative to older, on-premise methods such as ODBC (Open Data Base Connectivity). Besides enabling analytic models to easily consume OData sources, RASON produces OData endpoints for model results, enabling analytic model results as data – a key idea.

Integrating Analytics with Business Rules

RASON Decision Services provides everything needed both both advanced analytics and business rules: comprehensive data access, forecasting, text mining, machine learning, Monte Carlo simulation and mathematical optimization. Use of a predictive model as input to a set of business rules – considered an advanced use case in older systems – is simple in RASON Decision Services, where results from any analytics method can be used in decision tables and decision tables using FEEL, Excel formulas and high-level RASON notation can be used for business logic inside optimization or simulation models.

Multi-Stage Decision Flows of Analytic Models

RASON Decision Services enables users for the first time to define multiple "stages" in a single script, where a stage can perform a SQL operation, apply a data transformation, train a machine learning model, apply it to score new data, run a simulation, solve a mathematical optimization problem, or evaluate one or more linked decision tables. Results are passed between stages in a rich, standard "Indexed Data Frame" form.

OData Endpoints for Analytic Model Results

As a REST API, RASON accepts analytic models in JSON and returns results of solving these models in JSON; this is extended to multi-stage decision flows of analytic models. In addition, RASON Decisions Services offers an automatically created OData endpoint for the results from solving an analytic model. While it's easy for a newly-written Web or mobile application to consume RASON results in JSON, OData can be even better – since many popular Microsoft tools already consume OData feeds "out of the box". By surfacing analytic model results as data this way, RASON enables many integration tasks to be done with no code at all. For example, a POWER BI user can simply "point" to a RASON OData endpoint URL, then visualize, "slice and dice" analytic model results in a wide range of tables, charts and dashboards.

Rich Support for Power Platform, Office 365 and Azure

RASON aims to be the best decision management solution for companies with applications built on the "Microsoft technology stack," by greatly simplifying model creation, data access and model deployment. If you have used Analytic Solver for optimization, simulation, or data science – itself a modern, cloud-based "Office addin" - you'll find that it's easy to translate Excel analytic models into RASON notation with just a two mouse clicks! Your knowledge of Excel formulas and functions is immediately usable, but you'll find that RASON models can be more flexibly "bound" to data from a variety of sources.

Easily Convert Existing Analytic Solver, Solver SDK or XLMiner SDK Models

If you've ever programmed Solver SDK Platform or XLMiner SDK in a language such as .NET or C++, you'll quickly find that using the RASON tools is much faster/more productive than writing models entirely in code. This is true especially if you are using JavaScript and you are familiar with AJAX and REST API's. You'll find it's exceptionally easy to embed RASON models in your code – since RASON is JSON – and to solve them using Frontline's RASON server. This server, which exposes a simple REST API, is free for small models and experimentation, yet scalable to handle very large, compute-intensive analytic models.

Problems you can solve with the RASON server include linear programming and mixed-integer programming problems, quadratic programming and second-order cone problems, nonlinear and global optimization problems, problems requiring genetic algorithm and tabu search methods – from small to very large (LP/MIP models with millions of variables). You can also solve Monte Carlo simulation / risk analysis problems and create and solve models with uncertainty, using simulation optimization, robust optimization and stochastic programming methods. All facets of the data science process are supported, including data exploration and transformation, visualization, feature selection, text mining, time series forecasting, affinity analysis and unsupervised and supervised learning.

Free Trials, Learning and Coaching Resources

Business analysts and developers can sign up for free trial accounts to evaluate RASON Decision services at <http://RASON.com> which will provide access to tools to compose RASON models, exercise the REST API, try out dozens of example models illustrating use of decision tables, predictive models and machine

learning, optimization and simulation and download the RASON User Guide and Reference Guide in PDF form.

RASON.com makes it easy to get started with RASON Decision Services. But Frontline Systems can offer help, through the firm's consulting partners, to companies who need a solution, but may have limited experience with advance analytics, decision tables or REST APIs. For information, please contact sales@solver.com.

Overview of Features

The RASON REST API provides a set of simple, easy to use endpoints for defining models, solving them using conventional optimization, Monte Carlo simulation, stochastic optimization, decision table or a data science/prediction technique and getting results in JSON.

How do RASON models compare to alternative ways of building analytic applications?

- Compared to a modeling language such as AMPL or GAMS for optimization, or ARENA for simulation, RASON models have similar expressive power and are easy to understand. But as you'll see below, it is *much* easier to use a RASON model in a Web, mobile or server application.
- Compared to Excel as a modeling language, RASON models can use essentially all of Excel's operators and built-in functions, but it is *much* easier to build "dimensionally flexible" RASON models and to use a RASON model in a Web, mobile or server application.
- Compared to writing a model in programming language code, RASON models are "higher level" – *much* easier to create, modify and understand. But as you'll see below, RASON models are so easy to manipulate in code that you don't give up any flexibility.

An Optimization Example Model

Below is an example RASON optimization model. Its purpose is to find the optimal location for an airline hub that serves six cities. The cities are located at the (simplified x-y) coordinates given by the dx and dy parameters in the data section. Our goal is to find the x, y coordinates of the airline hub that will minimize the distance flown to any of the cities.

```
{
  modelName: "ExampleOptimization",
  modelType: "optimization",
  variables : {
    x : { value: 1.0, finalValue: [] },
    y : { value: 1.0, finalValue: [] },
    z : { value: 1.0, finalValue: [] }
  },
  data : {
    dx : { dimensions: [6], value: [1, 0.5, 2, 2, 2, 0.5] },
    dy : { dimensions: [6], value: [4, 3, 4, 2, 5, 6] }
  },
  constraints : {
    c : { dimensions: [6], upper: 0, formula: "sqrt((x - dx)^2 + (y - dy)^2) - z" }
  },
  "objective" : {
    "obj" : { formula: "z", "type": "minimize", "finalValue": [] }
  }
}
```

If you aren't familiar with modeling languages, you may not realize how much the RASON language is doing for you. This is a simple nonlinear optimization model. Note that the expression " $\sqrt{(x - dx)^2 + (y - dy)^2} - z$ " is an *array expression* operating over all six cities. The RASON Interpreter computes not just the values of this expression, but its partial derivatives with respect to x and y – used by the nonlinear optimizer to guide the search for a solution.

Web developers will recognize the overall syntax as that of JSON, JavaScript Object Notation – except that identifiers and keywords are not surrounded by double quotes, outside the "objective" section which shows an example of writing "strict JSON". The RASON Interpreter doesn't require the quotes in a model, but the result – the optimal solution to this nonlinear model - is always valid JSON.

```
{
  "status" : {
    "code" : 0,
    "id": "2590+ExampleOptimization+2020-01-18-16-53-45-215131",
    "codeText" : "Solver found a solution. All constraints and optimality
conditions are satisfied."
  },
  "variables" : {
    "x" : { "finalValue" : 1.250000 },
    "y" : { "finalValue" : 4.000000 },
    "z" : { "finalValue" : 2.136001 }
  },
  "objective" : {
    "obj":{ "finalValue" : 2.136 }
  }
}
```

A Monte Carlo Simulation Example Model

Below is a simple Monte Carlo simulation / risk analysis model that determines the optimal number of tickets to sell in an upcoming flight. The example takes into account "no-shows" as well as compensation fees if the flight sells out. The `uncertainVariables` section defines one uncertain (random) variable (`no_shows`) which uses a `PsiLogNormal` distribution to model the amount of customers who will not show up for a flight) and the `uncertainFunctions` section defines one uncertain function (revenue). Our goal is to find the expected mean (or average) revenue.

```
{
  modelName: "ExampleSimulation",
  modelType: "simulation"
  modelSettings : { numSimulations: 1, numTrials: 1000, randomSeed: 1 },
  data : {
    price: { value: 200 },
    capacity: { value: 100 },
    sold: { value: 110 },
    refund_no_shows: { value: 0.5 },
    refund_overbook: { value: 1.25 }
  },
  uncertainVariables : {
    no_shows: { formula: "PsiLogNormal(0.1*sold, 0.06*sold)", mean: [] }
  },
  formulas : {
    show_ups: { formula: "sold - Round(no_shows, 0)" },
    overbook: { formula: "Max(0, show_ups - capacity)" }
  },
  uncertainFunctions : {
    revenue: {
```



```

        formula: "price* (sold - refund_no_shows * ROUND(no_shows,0) -
        refund_overbook * overbook)", mean: [] }
    }
}

```

By default, the JSON result of evaluating this model contains a set of summary statistics for the output "revenue", across all Monte Carlo trials.

```

{
  "status": {
    "code": 0,
    "id": "2590+ExampleSimulateion+2020-01-02-07-30-10-477802",
    "codeText": "Solver has completed the simulation."
  },
  "uncertainFunctions": {
    "revenue": {
      "mean": 20286.4
    }
  },
  "uncertainVariables": {
    "no_shows": {
      "mean": 10.9922
    }
  }
}

```

An Example Data Science Model

The simple data mining example below, performs stratified random sampling on the hald-small-binary dataset. Beginning in *datasources*, the contents of hald-small-binary.txt are imported into the *mySrc* data source. Next, under *datasets*, the *mySrc* data source is bound to the *myData* dataset while also selecting the Y variable as the stratum variable. Moving on to *transformer*, the *mySampler* transformer is constructed using the stratified random sampling algorithm with a sample size equal to 10. Lastly, under *actions*, the stratified random sampler, *mySampler*, "transforms", or samples from, the *myData* dataset.

```

{
  modelName: "ExampleDM",
  modelType: "datamining"
  datasources: {
    mySrc: {
      type: 'csv',
      connection: 'hald-small-binary.txt',
      direction: "import"
    }
  },
  datasets: {
    myData: {
      binding: 'mySrc',
      strataCol: 'Y'
    }
  },
  transformer: {
    mySampler: {
      type: 'transformation',
      algorithm: 'stratifiedSampling',
      parameters: {
        sampleSize: 10
      }
    }
  }
}

```

```

    }
  },
  actions: {
    sampleData: {
      data: 'myData',
      action: 'transform',
      evaluations: [
        'transformation'
      ]
    }
  }
}

```

As in both the optimization and simulation examples above, the results are given in JSON which contain the ten sampled records: 1, 2, 3, 5, 6, 7, 8, 9, 11 and 13.

```

{
  "status": {
    "id": "2590+ExampleDM+2020-02-26-17-22-28-146373",
    "code": 0,
    "codeText": "Success"
  },
  "results": ["sampleData.transformation"],
  "sampleData": {
    "transformation": {
      "objectType": "dataFrame",
      "name": "Sample:myData",
      "order": "col",
      "rowNames": [
        "Record 11", "Record 8", "Record 2", "Record 9", "Record 1",
        "Record 7", "Record 13", "Record 6", "Record 3", "Record 5"
      ],
      "colNames": ["X1", "X2", "X3", "X4", "Weights"],
      "colTypes": ["double", "double", "double", "double", "double"],
      "indexCols": null,
      "data": [
        [1, 1, 1, 2, 7, 3, 10, 11, 11, 7],
        [40, 31, 29, 54, 26, 71, 68, 55, 56, 52],
        [23, 22, 15, 18, 6, 17, 8, 9, 8, 6],
        [34, 44, 52, 22, 60, 6, 12, 22, 20, 33],
        [3, 2, 3, 1, 1, 1, 1, 1, 2, 1]
      ]
    }
  }
}

```

An Example Decision Table

RASON users have the ability to create and evaluate Decision Tables. A decision table contains a set of rules which specify actions to perform based on specific conditions. Decision tables are a good tool to use when there is a consistent number of rules, or conditions, to be evaluated followed by a specific set of actions to be performed once a rule, or condition, is met. For example, the simple decision table below returns a patient's medical risk rating based on their age and medical history.

```

{
  modelName: "exampleDT",
  modelType: "calculation",

```

```

data: {
  age: { value: 54 }, medHistory: { value: 'good' }
},
decisionTables: {
  tblRisk: {
    inputs: ['age', 'medHistory'], outputs: ['riskRating', 'rule'],
    rules: [
      ['>60', 'bad', 'High', 'r2'],
      ['[25..60]', '-', 'Medium', 'r3'],
      ['<25', 'good', 'Low', 'r4'],
    ],
    hitPolicy: 'Unique'
  }
},
formulas: {
  res: { formula: "tblRisk(,age, medHistory)", finalValue: [] }
}
}

```

The results, as for optimization, simulation and data science models, are returned in JSON. A 54-year old patient with a good medical history rates as "medium" for his/her medical risk rating. See the chapter Defining Decision Tables in RASON for more information on how to define a decision table using the RASON modeling language.

```

{
  "status": {
    "code": 0,
    "id": "2590+DecisionTable+2020-01-02-05-21-19-476102",
    "codeText": "Solver has completed the calculation.",
    "observations": { "res": { "value": [ ["Medium", "r3"] ] } }
  }
}

```

Web and Mobile Applications

Suppose you wanted to define and solve a model in a mobile application written in JavaScript. Powerful optimizers that would run directly on a mobile device aren't available, nor would they be the best solution. But it's *simple* to send a model to the RASON REST Server, and get the optimal solution for an optimization model, the expected mean for a simulation model or a data science model (or forecasting model) fit to a training partition.

```

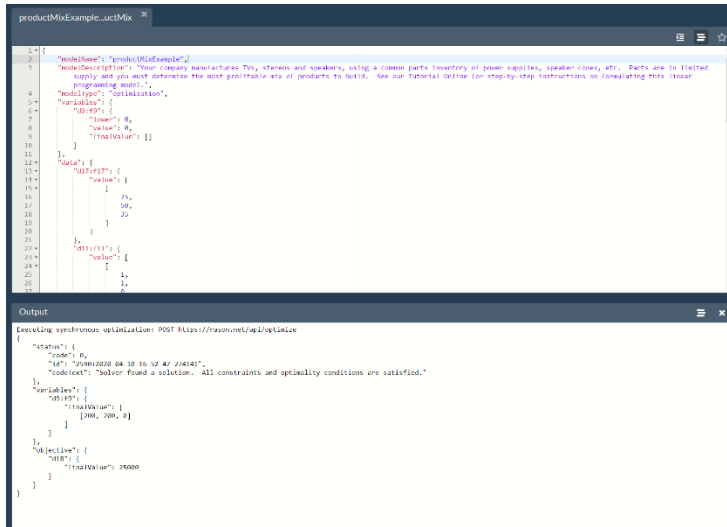
var request = { "variables" : { ...
"obj" : { formula:"z", "type": "minimize", "finalValue": [] }
} };
$.post("https://rason.net/api/optimize",
JSON.stringify(request))
.done(function(response) {
alert(response.objective.z.finalValue);
});

```

Since RASON models are **valid in JSON**, we can write the entire model as an object constant in JavaScript, assigned to the variable *request*. Then (using JQuery syntax) we make an AJAX request to the RASON Server's REST API endpoint *optimize*, which means "optimize this model and immediately return the result."

When the server returns a response, the "done" function is called, and it can **easily reference** the final value of the objective, since the response is **also JSON**.

Using the RASON Web IDE, we can perform the same actions as in the code segment above, by simply clicking the button labeled *POST rason.net/api/optimize*.



The screenshot shows a web browser window with the RASON Web IDE. The top part displays a JSON model for a product mix optimization problem. The bottom part shows the output of a POST request to the optimization API, which includes the status, code, and the optimized values for the variables.

```
1 [
2   {
3     "model": {
4       "name": "productMixExample",
5       "description": "Your company manufactures TVs, cameras and speakers, using a common parts inventory of power supplies, speaker cones, etc. Parts are in limited supply and you must determine the most profitable mix of products to build. See our Tutorial Online for step-by-step instructions on simulating this linear programming model.",
6       "variables": {
7         "x": {
8           "name": "x",
9           "value": 0,
10          "initialValue": 0
11        },
12        "y": {
13          "name": "y",
14          "value": 0,
15          "initialValue": 0
16        }
17      },
18      "data": {
19        "x": {
20          "value": 1,
21          "initialValue": 1
22        },
23        "y": {
24          "value": 1,
25          "initialValue": 1
26        }
27      }
28    },
29    "output": {
30      "status": "Success",
31      "code": 0,
32      "message": "Solving synchronous optimization: POST https://rason.net/api/optimize",
33      "variables": {
34        "x": {
35          "value": 1,
36          "initialValue": 1
37        },
38        "y": {
39          "value": 1,
40          "initialValue": 1
41        }
42      },
43      "objective": {
44        "value": 25000,
45        "initialValue": 25000
46      }
47    }
48  ]
49 ]
```

What about a mixed-integer or global optimization model, large simulation model or Big Data model that might take 30 minutes – or overnight - to run? That's easy: With *POST rason.net/api/model*, you can create a "resource ID," then start an optimization via *GET/POST rason.net/api/model/id/optimize*, check on its progress at any time with *GET rason.net/api/model/id/status* and obtain results when finished with *GET rason.net/api/model/id/result*.

Server –Based Applications

Next, suppose you wanted to define and solve the model in an application on a corporate server or Web server. RASON tools make that easy, too. Since the RASON Interpreter is embedded in Frontline's Solver SDK and XLMiner SDK products, an object-oriented library callable from a wide range of languages such as C++, C#, Java and PHP, you can simply load and run a RASON model from a text file on the server:

```
Problem prob = new Problem ();
prob.Load ("model.json");
prob.Solver.Optimize ();
MessageBox.Show (prob.FcnObjective.FinalValue [0].ToString ());
```

As this C# code suggests, once you've loaded a RASON model, you can easily access model elements from code, update data, optimize or simulate, even monitor the solver's progress.

Binding to Data and Dimensional Flexibility

The optimization example model above includes the data – the x,y coordinates of six cities – in the RASON request. It's easy to change the data in JavaScript by referencing `request.data.dx.value`, but the request itself will find a solution only for the included data. What if you want to solve this model for many different sets of data? RASON makes that easy too. For a small problem like this one, if you change data as shown below, you can pass new data directly in the REST API call, via standard HTTP GET query string parameters.

```
"data" : {
  {
    "dx" : { "dimensions": [6], "binding": "GET" },
    "dy" : { "dimensions": [6], "binding": "GET" }
  }
}
$.get (https://rason.net/api/optimize?dx=1,0.5,2,2,2,0.5&dy=4,3,4,2,5,6...
```

But many realistic models use large tables of data, often drawn from multiple data fields or databases. Further, our example model is "fixed" to six cities, but we might want to solve this problem many times, each time for a different number of cities – changing the dimensions of parameters *dx* and *dy*, or even changing the dimensions of arrays of decision variables, uncertain variables or constraints.

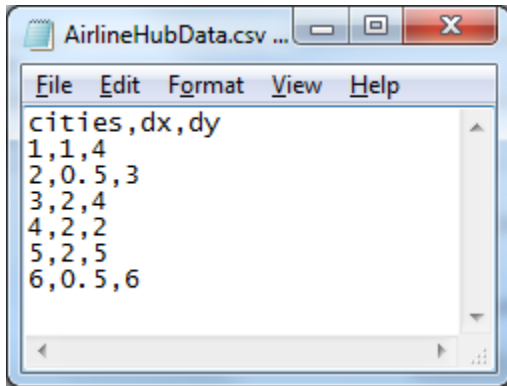
Below is the same RASON optimization model with an added section, *datasources*. Our list of city coordinates is now located in a CSV file, however, our data could also have been located in an ODBC database or an Excel file.

```
{
  modelName: "AirlineHubExample",
  modelType: "optimization",
  engineSettings : { engine : 'GRG Nonlinear' },
  datasources : {
    city_data: { type: 'csv', connection: 'AirlineHubData.csv',
      indexCols: ['cities'], valueCols: ['dx', 'dy'],
      direction: "import" }
  },
  variables : {
    x : { value: 1.0, finalValue: [] },
    y : { value: 1.0, finalValue: [] },
    z : { value: 1.0, finalValue: [] }
  },
  data : {
    dx : { binding: 'city_data', valueCol: 'dx' },
    dy : { binding: 'city_data', valueCol: 'dy' }
  },
  constraints : {
    c : { dimensions: ['cities', 1], upper: 0, formula:
      "sqrt((x - dx)^2
      (y - dy)^2) - z" }
  },
  objective : {
    obj : { type: 'minimize', formula: 'z', finalValue: [] }
  }
}
```

(We've used double and single quotes in this example, which are interchangeable.) The *dataSources* section defines a source from which the data values will be imported, formerly in the *data* section, in this case a comma-separated-value (CSV) file named *AirlineHubData.csv*. The data source defines an index column, *CITIES*, which is used to dimension the array of constraints, *c*.

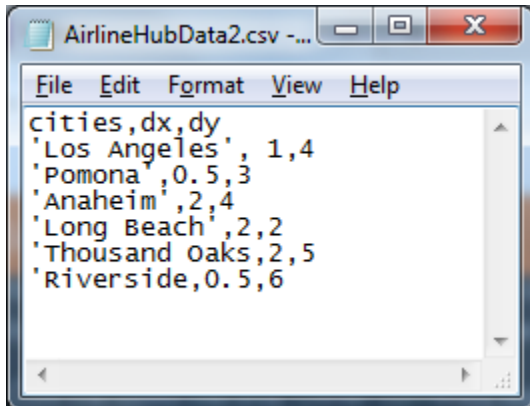
In the *data* section of the model, we *bind* the names *dx* and *dy* to 1-dimensional arrays from the corresponding *value columns* *dx* and *dy*. Our model can now be used to solve many different *instances* of this facility location problem, for a variable number of cities. (For more information on indexed sets, please see the *RASON Reference Guide*.)

The CSV file looks like this. Other currently supported data source types are 'excel', 'access' and 'odbc' for Oracle, SQL Server, OData and other databases.



Since the data source gave no URL or path for `AirlineHubData.csv`, this file must be in the 'current folder' when the model executes on the RASON back-end server; you can upload files to this folder along with your REST request. To try this yourself, create a text file `AirlineHubData.csv` from the values shown above (or below), upload `AirlineHubData.csv` using the Choose Files button on the [Editor](#) page, retrieve the model above from the RASON Examples drop down menu (`UGAirlineHubCSV.json`), create a resource ID using `POST rason.net/api/model`, then click `POST rason.net/api/model/id/optimize` and `GET rason.net/api/model/id/result`. (The "quick solve" `POST rason.net/api/optimize` endpoint doesn't accept uploaded files.)

Index columns and *index sets* are much more general than the 1 through 6 values in the example above. If our CSV file contained instead:



Our model would still solve. Further, we could refer to `dx['Pomona']` - "subscripting" the `dx` array with the index set element name, and get the value 0.5. More general references can return "slices" of multidimensional arrays, as we'll show in future examples.

Now that you have been introduced to the RASON modeling language and RASON REST API, continue reading for help registering to obtain a subscription to the RASON REST Server, defining a RASON model, and finally, calling the RASON REST API endpoints to solve your optimization, simulation, data science or decision table model.

RASON Subscriptions

Introduction

After you have registered on www.RASON.com, you will have access to our RASON REST API via rason.net. You can make calls to this API, either from the Editor page on www.RASON.com or from within your own application, to analyze a model, solve an optimization model, run a simulation, data science or decision table model, check the status of a previous run and to get the final results.

After you have subscribed to RASON.com, read the next section to learn how to setup a data connection in order to link to your data or upload your results. The RASON portal at <https://RASON.com> and the scalable REST API server at <https://rason.net> both run on Microsoft Azure. RASON Decision Services makes it exceptionally easy to work with data sources in the Microsoft ecosystem:

- OneDrive and OneDrive for Business
- Common Data Service for Dynamics 365, Power Apps and Power Automate
- OData and CDS support for Power BI
- [CData Cloud Hub](#) support for access to 100+ enterprise data sources

RASON handles authentication and data source authorization using Microsoft Identity Management and OAuth2 standards and maintains data access credentials in Azure "vaults" for information security.

Registering

To create a new account, visit www.RASON.com and click the orange Register for Free button on the Home screen.

The screenshot shows the RASON website's registration page. At the top, the RASON logo is displayed. Below it, a paragraph describes RASON as a modeling language embedded in JSON and a REST API for decision services. Three orange buttons are visible: 'Quick Overview >', 'Register for Free >', and 'Already Registered? Try it! >'. A line of text below the buttons states that RASON can be used as a cloud service or in SDK form. The bottom section of the page is divided into three columns for different user types: 'Analytics Professionals', 'Excel Solver Users', and 'Web App Developers'. Each column contains a brief description of how RASON integrates with their respective tools and a 'Learn more >' button.

You will be directed to the account registration page on the Website. After selecting your User Type and Industry Type by clicking the downward pointing arrows, complete the remaining fields, then click **Register**.

Register to use the RASONSM Service.

Have large or complex models? Need to run larger simulations? Want more computing power to solve your models?

Please contact sales:

- Call Us: 888-831-0333
- Email Us: sales@solver.com

To discuss your needs,

Create a new account here: (To download RASON[®] software for desktop use, [register on Solver.com](#).)

Use RASON online to define optimization and simulation models, solve them via our REST API, and create a basic web or mobile app.

User Type:

Industry Type:

Email:
A confirmation email will be sent to this address.

Password:
At least 6 characters containing UPPER and lower case, numbers, and special characters.

Confirm password:

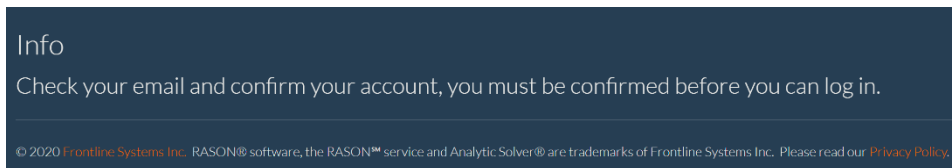
First & Last Name:

Company or University:

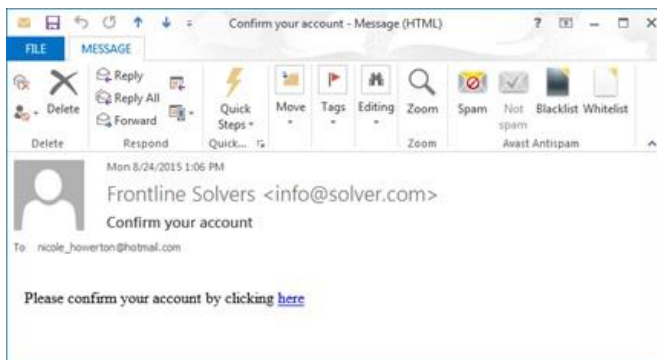
Country Code:

Telephone:

If your registration and password is accepted, you will see the following screen. If you are not directed to the following screen, correct any errors and again click Register.



An email will be sent to the address provided during account registration. (If you do not receive this email immediately, check your Junk folder to be sure it was not identified as spam.)



Click the hyperlink to confirm your account and log in to the RASON.com website. Enter your Email account and password, then click Log in, or click the Sign in with Microsoft button to login using your Microsoft credentials. Select the Remember me? checkbox if you'd like RASON.com to prefill your Email and Password information the next time you log in.

Log in to use the RASONSM Service

Use a local account to log in:

Email:

Password:

☐ Remember me?

Use another service to log in:

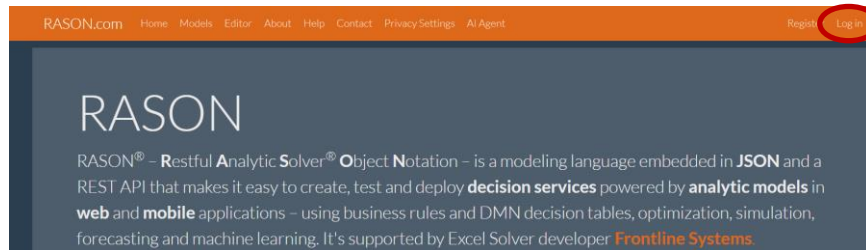
Integrated Microsoft Login

RASON.com includes Integrated Microsoft Login functionality which will automatically login visitors to www.RASON.com using their Microsoft credentials. If you already have a RASON.com account under a different email, a new RASON.com account will be created for you using the email utilized in your Microsoft credentials.

Once registered, a representative from Frontline will contact you with a trial license formulated to meet your needs.

My Account

Once logged in, you will be able to solve RASON models from within your own application or via the Editor page on RASON.com.



Click the down arrow on the upper right corner and click View Account to be directed to the My Account page.

- Click Overview to check the number of compute minutes used
- Click Data Connections to create and maintain your named data connections
- Click API Tokens to obtain your RASON API authorization token (or OAuth token), OAuth2 API Tokens or Runtime API tokens
- Click Personal Settings to change your personal profile or password

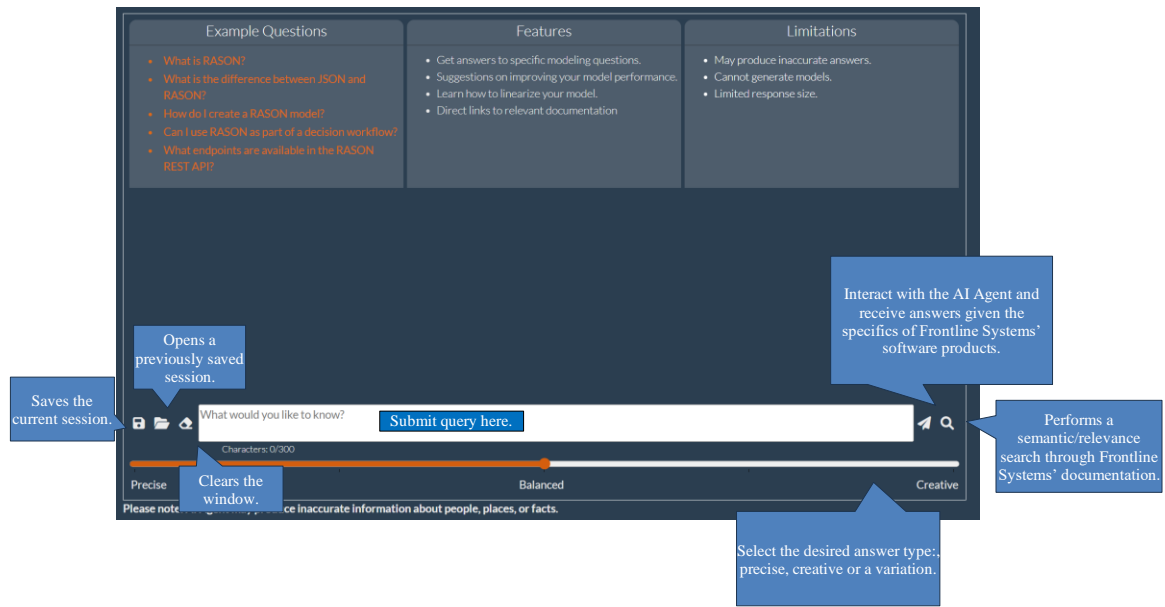
RASON Licenses

The RASON license in use determines the problem size limits that can be solved. Choose from various RASON licenses in order to select one that best meets your needs.

- RASON Comprehensive – Supports all functionality of the RASON Server. Solves optimization, simulation and data science models, including multi-stage decision flows, along with models containing DMN business rules, custom types and box functions.
- RASON Optimization -- Supports conventional optimization. (Stochastic or simulation optimization is not supported.) Simulation and Data Science functionality supported at basic problem limits.
- RASON Simulation -- Supports conventional, stochastic and simulation optimization along with Data Science functionality supported at basic problem limits.
- RASON Data Science -- Supports forecasting and data science techniques. Optimization and simulation functionality supported at basic problem limits.

AI Agent

If you are new to RASON Services and/or optimization, simulation, forecasting, data science, don't worry – Frontline's AI technical support assistant, AI Agent, is here to help. AI Agent is designed to provide assistance and support for users of Frontline Solvers' RASON Services software. The AI Agent is knowledgeable about the functionality and features of the software, as well as the concepts and processes involved in optimization, simulation, data science and forecasting. To get started, click the AI Agent tab on www.RASON.com and enter a topic or question such as "What classification algorithms are supported in RASON?", then click Submit Query.



Toggle between Precise, Balanced, and Creative to determine the type of answers returned. If Precise is selected, AI Agent will attempt to be as exact and deterministic as possible while Creative will usually result in more original, uncertain and non-repetitive answers. Use Balanced (the default) for the best of both worlds.

Click the **Save** file icon to save the current session. The file, transcript.json, will be downloaded. Click the **Open** file icon to open and restore a previously saved session. Click the **Erase** icon, to clear the current search window and start a new topic.

After typing a query,

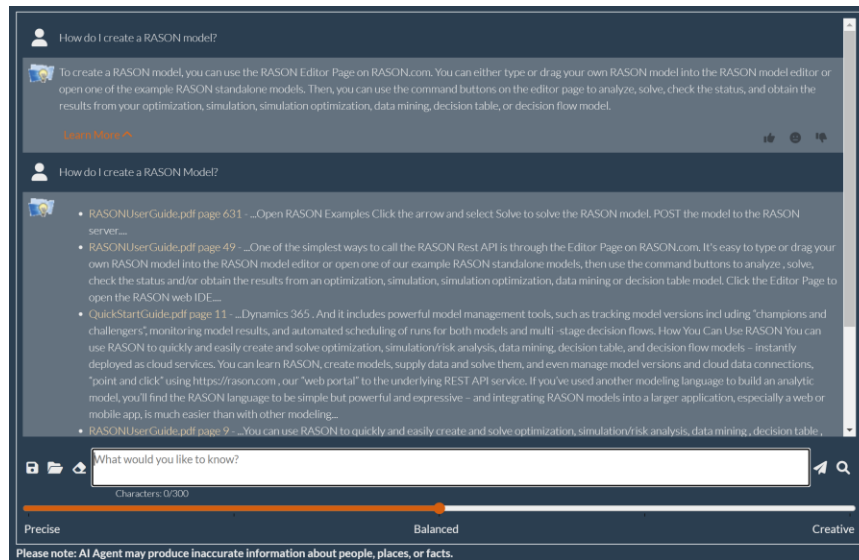
- Click **Submit Query** to interact with the AI Agent and receive answers given the specifics of Frontline Systems' software products. On each new question, the AI Agent will take into account the chat history that is present in the current session. Click *Learn More* to view all extracts of source documents that are most relevant to the query. Each extract contains a link to the appropriate page in one of the official documents.
- Click **Search Documentation** (for faster search results) to perform a semantic/relevance search through all Analytic Solver and RASON documentation. The resulting response will be similar to the "Learn More" section as described above.

The screenshot below illustrates the difference between the two different types of queries. The first query used AI technology by using Submit Query. The 2nd query was performed using Search Documentation.

Tips on getting the most from AI Agent

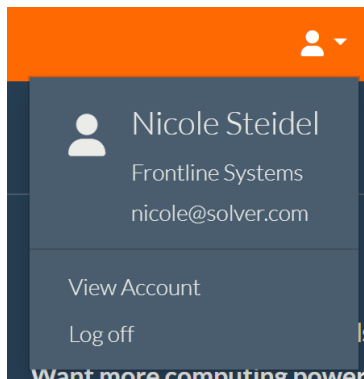
- Use "New Topic" frequently if your question does not require chat history. Results will be faster and more precise.
- Be as concise as possible in your question, i.e. "How to use RASON?" and "Give me step-by-step instructions on how to use RASON" will produce different results.

Note: The AI Agent in Frontline Solvers software uses "generative artificial intelligence" methods, which can at times produce erroneous information about people, places or facts, including incorrect information about Frontline's software products. Users are advised to take this into account, and use "common sense and human intelligence" when conversing with the AI Agent. Frontline Systems Inc. assumes no responsibility for inaccuracies in the AI Agent's responses.



My Account Overview Page

After logging in, click the down arrow next to the My Account icon and select View Account.



On the Overview tab of your My Account page, click Run Details to check the number of compute minutes used.

My Account	Overview			
Overview	RASON compute time used this month 00:01:00, 15:58:59 remaining.			
Data Connections	Period	Total Runs	Successful	Problem
API Tokens	24 Hours	4	4	0
Personal Settings	7 Days	134	115	19
Profile	30 Days	242	185	57
Security	Run Details			

Overview / Run Details

Download Run Details ▾

Solve Type	Timestamp (GMT)	Solve Time (Millis)	Result
Workflow	1/21/2020 10:29:08 PM	34	SUCCESS
Workflow	1/21/2020 10:00:56 PM	24	SUCCESS
Workflow	1/21/2020 9:59:40 PM	2201	SUCCESS
Workflow	1/21/2020 1:22:10 AM	35	ENGINE_EXCEPTION
Data Mine	1/21/2020 1:14:31 AM	1129	SUCCESS
Workflow	1/21/2020 12:36:58 AM	1438	SUCCESS
Decision Table	1/20/2020 3:29:00 AM	45	SUCCESS
Decision Table	1/20/2020 3:11:38 AM	13	SUCCESS
Decision Table	1/20/2020 2:49:27 AM	336	SUCCESS
Data Mine	1/20/2020 1:55:38 AM	438	SUCCESS
Data Mine	1/20/2020 1:35:52 AM	298	SUCCESS

Click the down arrow next to Download Run Details to download details of past runs from the last 24 hours, last 3 days, last 7 days, last 15 days or last 30 days.

Download Run Details ▾

☒ Last 24 hours
 ☐ Last 15 days
 ☐ Last 3 days
 ☐ Last 30 days
 ☐ Last 7 days
 ☐ Custom

Download

Data Mine 1/21/2020 1:14:31 AM

Click Custom to enter a specific date range.

☐ Last 24 hours
 ☐ Last 15 days
 ☐ Last 3 days
 ☐ Last 30 days
 ☐ Last 7 days
 ☒ Custom

Start Date
 2020-01-01

End Date
 2020-01-22

Download

See below for explanations on the remaining two tabs: Data Connections and API Tokens.

A Note on Throttling Limits

Throttling limits are imposed regardless of your license type. See the table below for current limits; limits can be adjusted if your application solves many small problems at a faster rate.

Time	Max Calls
per second	50

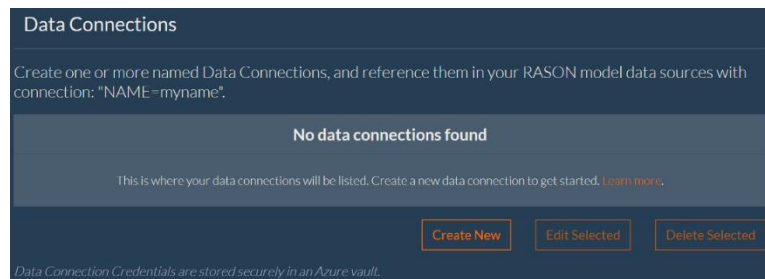
per minute	3,000
per hour	6,000
per day	6,000
per week	50,000

My Account: Data Connections and Creating API Tokens

See below for explanations on the remaining two My Account tabs: Data Connections and API Tokens

Data Connections Tab

Click Data Connections (on the left) to open the Data Connections page.



In previous versions of RASON, models that accessed external databases required actual credentials to be passed, such as database URLs, port numbers, usernames and passwords, in the text of the RASON model, in a datasource declaration, as shown below.

Previous versions of RASON

```
"parts_data": {
  "type": "odbc",
  "connection": "Driver={SQL Server Native Client
11.0};Server=tcp:solver.database.windows.net,1433;Database=RASON;Uid=RA
SONread;Pwd=RASON1234;Encrypt=yes;TrustServerCertificate=no;Connection
Timeout=30;",
  "selection": "SELECT Parts as parts, Products as prods, Qty as qty
FROM Parts ORDER BY ID",
  "indexCols": [ "parts", "prods" ],
  "valueCols": [ "qty" ],
  "direction": "import"
},
```

RASON offers an alternative to tackle this security risk by substituting

```
"connection": "Driver={SQL Server Native Client
11.0};Server=tcp:solver.database.windows.net,1433;Database=RASON;Uid=RA
SONread;Pwd=RASON1234;Encrypt=yes;TrustServerCertificate=no;Connection
Timeout=30;",
```

with three options: a file containing the contents of "connection" as in (1) below, a named Data Connection as shown in (2) or a URL pointing to Microsoft Common Data Service as shown in (3).

1. "connection": "File = filename",

RASON Decision Services will interpret this as (i) get the text contents of filename, which must be attached to the current model instance and (ii) substitute this text for the string "File=filename".

Therefore, if filename contains the text

"Driver={SQLServerNativeClient...Timeout=30;", the effect will be the same as in previous versions of RASON.

2. "connection": "Name=myname", where myname is the name given to the Data Connection. See below for instructions on how to create a named Data Connection.

3. "connection": "secret=uri", where uri is the Microsoft Common Data Service URL

"connection": "xxxx.crm.dynamics.com" where the actual Microsoft Common Data Service URL is passed directly to "connection".

If using a with "secret=url" in the dataSources section of your RASON model, enter a URI of the form <https://subdomain.crm.dynamics.com>, i.e.

https://www.cdatacloud.net/solver/api.rsc/GoogleSheets1_ODataSourceExample_Sheet1,

RASON Decision Services will interpret this as (i) get the text contents of the "secret" represented by the URL and (ii) substitute this text for the string "Secret=url". So if the "secret" contains the text "Driver={SQLNativeClient...Timeout=30;", the effect will be the same as in previous versions of RASON.

Similarly, if using CData Cloud Hub with "connection": "xxxx.crm.dynamics.com", enter a URI of the form <https://subdomain.crm.dynamics.com>.

RASON Decision Services will interpret this as (i) get the text contents of the connection represented by the URL and (ii) substitute this text for the string "connection".

Currently, RASON Decision Services supports "secrets" maintained, only, in an Azure Key Vault.

Enterprise customers can provision their own Key Vault and arrange to authenticate the RASON Server to this Key Vault if so desired.

Organization-Specific Vaults and Storage Accounts

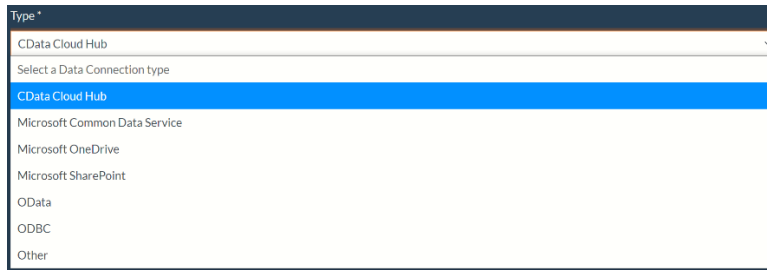
The RASON Server uses an Azure "storage account" to persist all data, including RASON models, fitted models (PMML or JSON) and attached files (CSV or XLSX), as well as temporary queues and tables. Via an API request, database credentials and similar "secrets" may be stored in a Frontline-provisioned Azure vault. (See the Data Connections topic.) All data is encrypted in motion and encrypted at rest, and requires an OAuth2-based token for access. However, RASON Decision Services includes a provision for the public RASON Server to use an organization's own Azure vault instead of Frontline -provisioned vaults, and to use an organization's own private Azure storage account to persist all models and data used under its RASON Organization Account. The RASON Server will authenticate itself through Azure AD to the organization's Azure resources.

Creating a Named Data Connection

When using data connections, you must first create a Named Data Connection on the My Account page on www.RASON.com. Begin by clicking **Create New** to open the Create New Data Connection dialog.



Type: Select CData Cloud Hub, Microsoft Common Data Service, Microsoft OneDrive, Microsoft Sharepoint OData, ODBC or Other for the data connection type.



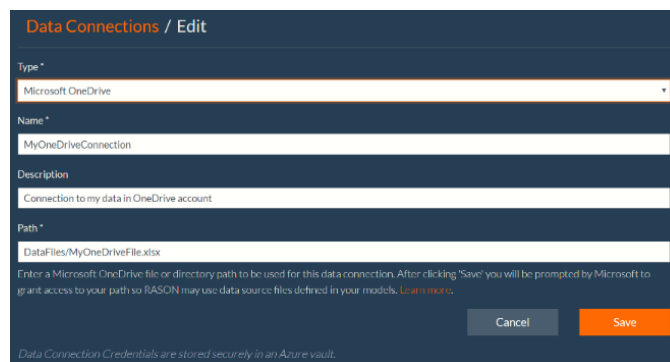
Name: Enter a Name for the new named data connection. This is "myname" that will be passed for "connection": "Name=myname".

Description: Enter a description (optional) for the data connection.

URI: Enter the appropriate connection depending on your connection type.

- **OneDrive (or OneDrive for Business)**

Path: Enter the path to your file (i.e. "/MyOneDriveFile.xlsx"), then click Create. The Microsoft Login screen will launch. Login to your Microsoft account to grant permission to rason.net for your selected file.



- **Sharepoint**

Path: Enter the path to your file (i.e. "/MySharepointFile.xlsx"), then click Create. The Microsoft Login screen will launch. Login to your Microsoft account to grant permission to rason.net for your selected file.

SharePoint Site Collection: Enter a Microsoft SharePOint site collection name such as "MyCompany.sharepoint.com" for the default collection or "MyCompany.sharepoint.com:/data" for a subsite collection.

Data Connections / Edit

Type *
Microsoft SharePoint

Name *
MySharepointConnection

Description
Connection to my data saved on my SharePoint account

Path *
DataFiles/MySharepointFile

Enter a Microsoft SharePoint file or directory path to be used for this data connection. After clicking 'Save' you will be prompted by Microsoft to grant access to your path so RASON may use data source files defined in your models. [Learn more](#)

SharePoint Site Collection *
MyCompany.Sharepoint.com

Enter a Microsoft SharePoint site collection name. For example, "companyname.sharepoint.com" for the default collection or "companyname.sharepoint.com/subsite-name" for a subsite collection.

Cancel Save

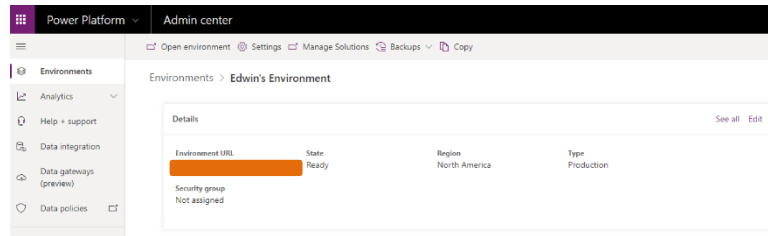
- **Common Data Service**

Click "Copy" to copy the Appid to the Clipboard. Within Power Apps, you'll need to create an Application User using this Appid, and assign a security role to this APP user with the appropriate permissions. Click [here](#) for more details.

The image illustrates the process of creating a security role in the Power Platform Admin center. It shows the 'Data Connections / Edit' form, the 'Power Platform Admin center' 'Settings' page, the 'Security Roles' list, and the 'Manage User Roles' dialog. Arrows indicate the flow from the Admin center to the Security Roles page, then to the 'RasonUser' role, and finally to the 'Manage User Roles' dialog where the role is assigned to the application user.

1. Select the desired Environment within Power APPs and click Settings.
2. Select Security Roles, enter a Role Name (in this case RASONUser) and assign privileges.
3. Click back to Users, enter the [www.RASON.com](https://www.rason.com) User Name and the RASON Server AppsID/ClientID (eb69c186-3bee-4146-9483-b0f4d7d55be1), click Manage Roles, select the appropriate Role Name (created in 2) from the Manage User Roles list and click OK.

URI: Enter the Environment URL as found within your Power Apps environment, click Create.



Data Connections / Edit

Type *
Microsoft Common Data Service

Name *
MyCommonDataServiceConnection

Description
Connection to my data using Common Data Service

URI *
<https://myenvironmentURL>

Enter a Microsoft Common Data Service URL using the following format: `https://unique_name.crm.dynamics.com/api/data/v9.1/`. Replace the highlighted component with your specific Environment Unique Name that has been configured to allow access by the RASON server. [Learn more.](#)

Cancel Save

- **CData Cloud Hub**

Using CData Cloud Hub, you'll first need to create a new user, who has been granted access to your data, and create an Authtoken, as shown in the screenshot below. Click [here](#) for more details.

Resource URL: Enter a CData Cloud Hub resource URL using the following format: https://www.cdatacloud.net/instance_name/api.rsc/my_entity where "instance_name" should be replaced with your specific OData Endpoint and "my_entity" with your specific table name.

CData Auth Token: Enter the Authtoken generated from the Users page on CData Cloud Hub (see screenshot below) for "URI", then click Create.



Data Connections / Create

Type *
CData Cloud Hub

Name *
MyCloudHubConnection

Description
Connection to my Data in CData Cloud Hub

Resource URL *
https://www.cdatacloud.net/instance_name/api.rsc/my_entity

Enter a CData Cloud Hub resource URL using the following format: `https://www.cdatacloud.net/instance_name/api.rsc/my_entity`. Replace the highlighted components with your specific OData Endpoint and Table names. [Learn more.](#)

CData Auth Token *
8x4Q.....

Cancel Create

- **ODATA**

URI: Enter the URL for your OData service using the following format: `https://host/service` or `https://host/service/my_entity` where `host`, `service` and `my_entity` are your specific OData Endpoint and entity names. For example, "`https://rason.net/DansOfficeScheduler`" where "`rason.net`" is the host and "`DansOfficeScheduler`" is the service, then click Create.

Authorization Header Name: Enter the name of your header if one is required.

Authorization Header Value: Enter the value for your header, if required.

The screenshot shows the 'Data Connections / Create' form. The 'Type' dropdown is set to 'OData'. The 'Name' field contains 'MyODataConnection'. The 'Description' field contains 'Connection to my data in OData'. The 'URI' field contains 'https://rason.net/DansOfficeScheduler'. Below the URI field, there is a note: 'Enter a URI for your OData service using the following format: https://host/service/ or https://host/service/my_entity. Replace the highlighted components with your specific OData Endpoint and entity names. Learn more.' The 'Authorization Header Name' field contains 'Authorization' and the 'Authorization Header Value' field contains 'Bearer <123456789...>'. Below these fields, there is a note: 'If required, enter the authorization header name and value for your data service. For example:'. A table shows two examples: 'Authorization' with 'Basic <base64 encoded username:password>' and 'Authorization' with 'Bearer <oauth token>'. At the bottom, there are 'Cancel' and 'Create' buttons.

- **ODBC**

Connection String: Enter the ODBC connection string as shown for Connection String, then click Create. Code snipped below taken from the RASON example, `ProductMixSQL11.json`.

```
"parts_data": {
  "type": "odbc",
  "connection": "Driver={SQL Server Native Client
11.0};Server=tcp:solver.database.windows.net,1433;Database=RASON;
Uid=RASONread;Pwd=RASON1234;Encrypt=yes;TrustServerCertificate=no
;Connection Timeout=30;",
  "selection": "SELECT Parts as parts, Products as prods, Qty as
qty FROM Parts ORDER BY ID",
  "indexCols": [ "parts", "prods" ],
  "valueCols": [ "qty" ],
  "direction": "import"
},
```

The screenshot shows the 'Data Connections / Create' form. The 'Type' dropdown is set to 'ODBC'. The 'Name' field contains 'MyODBC Connection'. The 'Description' field contains 'Connection to my Data using ODBC'. The 'Connection String' field contains 'Driver={SQL Server Native Client 11.0};Server=tcp:solver.database.windows.net,1433;Database=RASON;Uid=RASONread;Pwd=RASON1234;Encrypt=yes;TrustServerCertificate=no;Connection Timeout=30;'. Below the connection string field, there is a note: 'Enter an ODBC connection string to access your database. For example: Driver={SQL Server Native Client 11.0}; Server=tcp:my_company.database.windows.net,1433; Database=my_database; Uid=my_username; Pwd=my_password; Encrypt=yes; TrustServerCertificate=no; Connection Timeout=30; Learn more.' At the bottom, there are 'Cancel' and 'Create' buttons.

- **Other**

Select Other to enter any type of connection string. RASON will include the header name and value in the connection request.

After the data connection is created and access is granted to rason.net with your data service provider, you'll need to enter your "connection" into your RASON model using "connection": "name=myname", or specifically:

Data Connections			
Create one or more named Data Connections, and reference them in your RASON model data sources with connection: "NAME=myname".			
<input type="checkbox"/>	Name	Type	URI
<input type="checkbox"/>	MyOneDriveConnection	Microsoft OneDrive	DataFiles/MyOneDriveFiles.xlsx
<input type="checkbox"/>	MySharepointConnection	Microsoft SharePoint	DataFiles/MySharepointFile
<input type="checkbox"/>	MyCloudHubConnection	CData Cloud Hub	https://www.cdatacloud.net/instance_name/api/rsz/my_entity
<input type="checkbox"/>	MyCDSCConnection	Microsoft Common Data Service	https://myenvironmentURL
<input type="checkbox"/>	MyODDataConnection	OData	https://rason.net/OfficeScheduler
<input type="checkbox"/>	MyODBCConnection	ODBC	Driver={SQL Server Native Client 11.0}; Server=tcp:my_companydatabase.windows.net,1433; Database=my_database;Uid=my_username;Pwd=my_password; Encrypt=yes;TrustServerCertificate=no;Connection Timeout=30;

- Microsoft One Drive: "connection": "name=MyOneDriveConnection"
- Microsoft Sharepoint: "connection": "name=MySharepointConnection"
- CData Cloud Hub: "name=MyCloudHubConnection"
- Microsoft Common Data Service: "name=MyCDSCConnection"
- OData: "name=MyODDataConnection"
- ODBC: "name=MyODBCConnection"

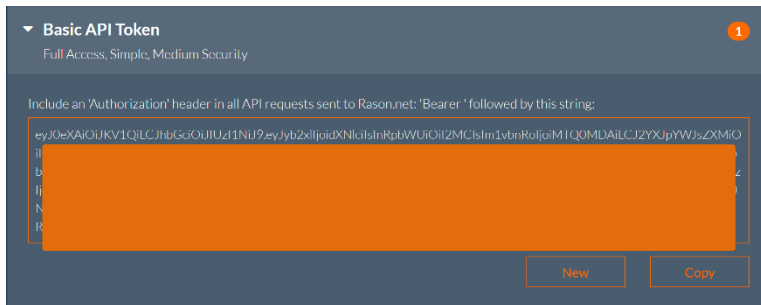
Creating API Tokens

Click API Tokens to create Basic, OAuth2 and Runtime API Tokens.

Each API request to RASON Decisions Services is authenticated and authorized by examining the Bearer API Token that accompanies each request. In the past, Basic and Runtime Tokens were both Base64-encoded JWT tokens encrypted "in transit" by TLS 1.2. *The latest version of RASON Decision Services includes even more advanced encryption for Basic and Runtime Tokens.* For more information on RASON Decision Services security features, please contact us at support@solver.com.

Basic API Tokens

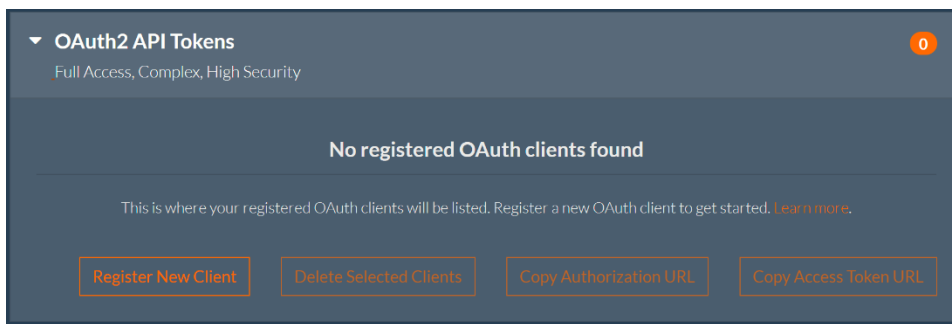
Click the arrow beside Basic API Token to view your Basic API Token. A Basic API Token allows Full Access to rason.net. When solving a RASON Model via the Editor page on www.RASON.com, your OAuth token is passed automatically to each RASON REST API call. However, if you are solving a RASON model from within your own application, you must include an 'Authorization' header (which includes a value 'Bearer' followed by your account OAuth token) in order to make a call to a RASON REST API endpoint. See the next chapter *Using the RASON REST API* for more details on how to pass an Authorization header from within your own application.



Click **New** to create a new Basic API Token or **Copy** to copy the token to the Clipboard. One Basic API Token is supported per account.

OAuth2 API Tokens

Click the arrow beside OAuth2 API Tokens to view your OAuth2 tokens.



OAuth 2 is a well-documented and widely used method for securing web accessible resources. Currently, the RASON service implements the OAuth 2 Authorization Code Grant flow with Proof Key Code Exchange (PKCE). This section provides documentation and guidance on how to use the OAuth 2 Authorization Code Grant Flow with PKCE to secure your RASON resources.

- Click Register New Client and follow the directions below to create an OAuth2 API Token.
- Click Delete Selected Clients to delete the desired clients
- Click Copy Authorization URL to copy the Authorization URL to the clipboard
- Click Copy Access Token URL to copy the Access Token URL to the clipboard

Runtime API Tokens

Click the arrow beside RunTime API tokens to view your RunTime tokens.

2. Click the 'My Account' link in the upper right corner or simply navigate to <https://RASON.com/Manage/Account>
3. Click API Tokens, then OAuth2 API Tokens.
 - a. Any previously registered clients will be listed here.
4. Click 'Register New Client' to launch the new client registration dialog
5. Fill out the client registration form
 - a. **Type:** (Required) Select either 'public' or 'confidential' depending on the type of client. See <https://tools.ietf.org/html/rfc6749#section-2.1> for an explanation of those types. However, please note that RASON.com does not issue client secrets, even in the case of a confidential client.
 - b. **Scope:** (Required) Select either 'full' or 'runtime'. This defines what the client can do with the resources. A scope of 'full' will allow the client to create, update, read and delete all resources. A scope of 'runtime' only allows model execution and access to model results. Clients with this scope will not be able to create, modify or delete resources. Nor will this allow the client to read the model itself.
 - c. **Name:** (Required) This is the name of the registered client.
 - d. **Redirect URI:** (Required) This is the URI that will be redirected to after submitting the authorization form.
 - e. **Description:** (Optional) Enter a description of the client.
 - f. **Access token duration in minutes:** This will define the "time to live" for all access tokens issued to this client. Depending on the needs of the client, the user may wish to have longer or shorter access token duration. The security trade off here is that longer-lived access token requires less human interaction (in the form of authorization), but a larger security exposure if the token is compromised. A shorter duration token has a smaller security exposure window, but may require more frequent authorization.
 - g. **Issue refresh tokens?:** Check the box if you want the server to issue refresh tokens with access tokens. Note that refresh tokens can be exchanged for access tokens and any given refresh token can be used only once. If checked, a new refresh token will be issued with each access token issued. If you do not want refresh tokens to be issued, leave the box unchecked.
 - h. **Refresh token duration in minutes:** If you checked the box to issue refresh tokens, enter the "time to live" for the refresh token. This defines the interval of time when a refresh token can be exchanged for an access token, beginning when the refresh token was first issued. Depending on your needs a good practice is to have relatively short access token durations and relatively long refresh token duration.
 - i. **Click the 'Register Client' button** to register the client.

Once registered, the client will be issued an Identifier by RASON.com. This identifier is required along with the client redirect URI and scope to complete the process.

Grant Authorization to Registered Client

In this step, users will authorize their clients at RASON.com using PKCE.

1. Generate a suitable PKCE challenge value. See <https://tools.ietf.org/html/rfc7636#page-8> for details on generating a PKCE challenge and value pair. For testing purposes, use a PKCE generator available on the web, such as <https://tonyxu-io.github.io/pkce-generator>. Note that the method used here must be S256.

2. Navigate to the authorization endpoint: <https://RASON.com/Auth/authorize> with the following query parameters:
 - a. **client_id**: (Required) Use the client ID issued to the client by RASON.com when the client was registered.
 - b. **redirect_uri**: (Required) Use the redirect URI for the registered client identified in 'a' above.
 - c. **response_type**: (Required) Must be 'code' (no quotes).
 - d. **code_challenge**: (Required) The PKCE code challenge generated in step 1 above.
 - e. **code_challenge_method**: (Required) Must be 'S256' (no quotes).
 - f. **scope**: (Optional) If omitted, will default to the scope of the registered client. If specified, and the scope of the registered client is more restrictive, then the scope of the registered client will be used. Otherwise, this scope will be used.
 - g. **state**: (Optional) This is additional information that will be passed through to the redirect URI.
3. Example authorization request
 - a. https://RASON.com/Auth/authorize?client_id=abcd1234&redirect_uri=https://mydomain.com/clientauthorization&response_type=code&code_challenge=IDqjKbnDNCEISSCYiTxREAYhvp1grPcWuQ0oupoV3_o&code_challenge_method=S256&scope=runtime&state=userdata
4. Authorize the client by:
 - a. Entering the user's RASON.com credentials.
 - b. Checking the 'Allow Client Access' checkbox.
 - c. Clicking the 'Authorize' button.
5. At this point, users will be redirected to the registered clients redirect URI with the following query parameters:
 - a. **code**: This is the authorization code, not the access token. This authorization code must be exchanged for the access token. *After ten minutes, this code will expire and can no longer be used.*
 - b. **state**: Any state information specified in step 2, part g, above.

Exchange Authorization Code for Access Token

Once the authorization code has been received, it must be exchanged for an access token. Recall that the authorization code is good for 10 minutes from the time it was issued, so the following steps must be performed before the code expires. Note that the authorization code will be URI encoded and will need to be decoded in order to be used further.

To exchange the authorization code for an access token:

1. POST the following values to <https://RASON.com/Auth/Token> using the "application/x-www-form-urlencoded" format:
 - a. **grant_type**: (Required) Value MUST be set to "authorization_code" (without quotes).
 - b. **code**: (Required) The authorization code received from the authorization request as documented above.
 - c. **redirect_uri**: (Required) Must be identical to the registered redirect_uri of the registered client.
 - d. **client_id**: (Required) The ID assigned when the client was registered, as documented in the 'client registration' section.
 - e. **code_verifier**: (Required) The value used to generate the code_challenge in the authorization request, as noted in step 1 under 'Grant Authorization to Registered Client'.

2. Upon successful processing of the request, RASON.com will issue a JSON object response with the following properties:
 - a. **access_token**: This is the access token allowing the bearer to access resources at rason.net, the resource server.
 - b. **token_type**: All tokens are of type “bearer”.
 - c. **expires_in**: The interval of time, in seconds from now, for which the token is valid. This number is simply the **Access token duration in minutes** specified at client registration expressed in seconds rather than minutes. Once the time interval has passed, the token is no longer valid.
 - d. **scope**: This is scope of the access token, either ‘full’ or ‘runtime’.
 - e. **refresh_token**: If the **Issue refresh tokens?** checkbox was selected when the client was registered, this will have the refresh token, which can be exchanged for an access token and another refresh token.

Exchange Refresh Token for Access Token

If the client was configured to issue refresh tokens, then you can exchange a refresh token for an access token as long as the refresh token has not expired. Perform the following steps to exchange a refresh token for an access token.

1. POST the following values to <https://RASON.com/Auth/Token> using the "application/x-www-form-urlencoded" format:
 - a. **grant_type**: (Required) Value MUST be set to "refresh_token" (without quotes)
 - b. **refresh_token**: (Required) the refresh token received from a prior access token request.
 - c. **client_id**: (Required) The ID assigned when the client was registered, as documented in the ‘client registration’ section.
 - d. **redirect_uri**: (Required) Must be identical to the redirect_uri of the registered client.
2. Upon successful processing of the request, RASON.com will issue a JSON object response with the following properties:
 - a. **access_token**: This is the access token allowing the bearer to access resources at rason.net, the resource server.
 - b. **token_type**: All tokens are of type “bearer”
 - c. **expires_in**: The interval of time, in seconds from now, for which the token is valid. This number is simply the **Access token duration in minutes** specified at client registration expressed in seconds rather than minutes. Once the time interval has passed, the token is no longer valid.
 - d. **scope**: This is scope of the access token, either ‘full’ or ‘runtime’
 - e. **refresh_token**: A new refresh token, which can be exchanged for an access token and another refresh token.

Use Access Token to Interact with Resource Server (rason.net)

Once you have the access token, include it as a bearer token in the Authorization header for all requests to the rason.net resource server.

RASON Services Web IDE

Introduction

The RASON API (rason.net) may be called in two different ways: from within your own application or from the Editor page on RASON.com. (You can also solve a RASON model through the RASON IDE which requires a license to either Frontline's Solver SDK Platform or XLMiner SDK.) To solve a RASON model through the Editor page on RASON.com or from within your own application, you must first register and/or purchase a subscription. This chapter walks you through the required steps to get you up and running with a RASON subscription.

See the later section, *Using the REST API*, for help on calling the RASON REST API endpoints directly in order to solve your optimization, simulation, simulation optimization, decision table or data science model from within your own application.

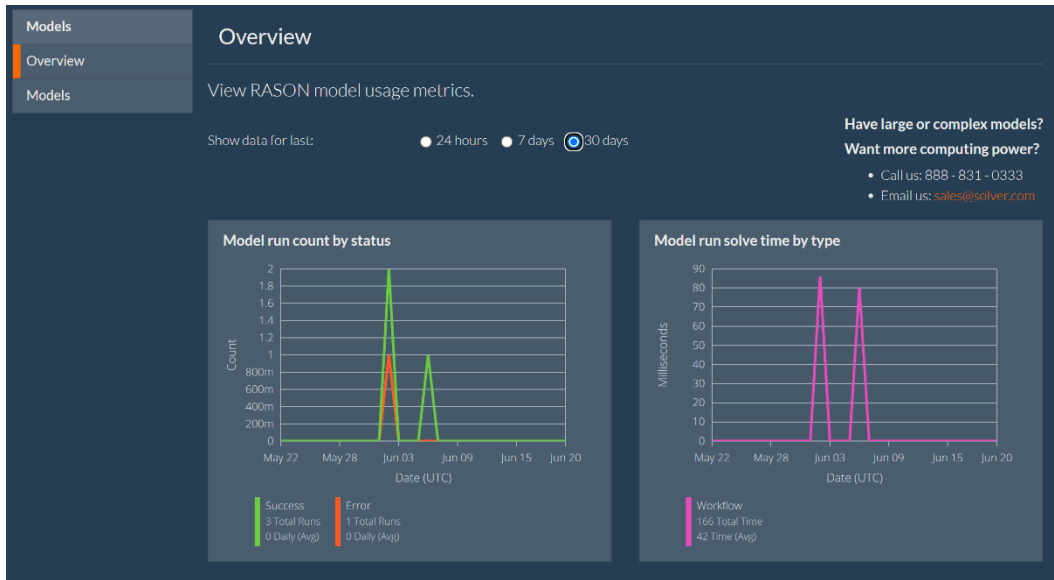
The Models tab at www.RASON.com allows users to easily manage their models and to obtain model information such as a list of all models posted to your RASON account, total run time for each model, number of successful runs, etc. In addition, users can quickly sort, group and/or filter through their model list in order to quickly identify a specific model.

RASON Models

As you or your company make further use of RASON, you'll build up a set of RASON models for optimization, simulation, data science and business rules, fitted machine learning models, probability models, and even Excel workbook models, that are maintained on the RASON Service and run periodically to compute new optimal solutions, new risk analyses, or make new predictions and new rule-based decisions. Managing these models as valuable intellectual property assets, and monitoring their performance, will become more important over time.

The **Models tab** in the RASON IDE gives you easy access to a great deal of information about each of your models, including properties such as model name, type, version and date last modified, and metrics such as total run time, number of runs, runs ending in an error, etc. The Models tab makes it easy to view your models filtered, sorted and grouped by type, data connection use, and other properties, and run outcomes and metrics such as solve time, over your choice of time intervals – and to export this data for further analysis.

The Overview tab shows your total RASON model usage metrics within the last 24 hours, the last 7 days or the last 30 days.



The Models tab gives you a comprehensive view of RASON model use, no matter how the model was created or run. It includes metrics for models you create, save and run interactively using the RASON Editor, but it also includes metrics for runs of models previously saved to your RASON account and run “on demand” from an external application, using an asynchronous POST endpoint (POST rason.net/api/model/id/<solvetype>). It includes results for models saved to your RASON account that originated in Analytic Solver for Excel (that were either translated to RASON, or “wrapped for RASON” and uploaded as Excel workbooks). It includes models run from a decision flow, or scheduled for automatic runs using RASON’s own scheduling facility. The Model tab is likely to prove most useful in “production use”, when you have a number of models (or model versions) being run from external applications, or run on a schedule on a regular basis.

Click Models (beneath Overview) to view the metrics for each model saved to your account.

A Note on RASON Organization Accounts

As discussed in the RASON Organization Account chapter that occurs later in this guide, with an Organization Account, your company’s models and data are maintained privately in the company’s own Azure Storage Account, isolated from all other RASON users. Using standard Azure Role-Based Access Control (RBAC), the company can assign “roles” to different users that give them selective access to storage containers where their models and data are saved.

In the RASON Model Editor, your company’s authorized users will be able to “see” the containers they can access (read-only or read/write), and easily create, update or delete models within those containers in accordance with their roles. Similarly, metrics appearing on the Models Overview page and models reported in the Models list, are determined by the user’s role and permissions. See the section below to see how permissions will affect the Editor page functionality.

Models Tab

- List Context Menu
 - “Open Model” is disabled if container does not have Read permission.
 - “Model Details” is disabled if container does not have Read permission.
- Detail Context Menu
 - “Open” is disabled if container does not have Read permission.

For more information on RASON Organization Accounts, see the RASON Organization Account chapter or contact Sales at sales@solver.com.

Change column order by dragging columns to the desired orientation. Note: A "grouped" column, such as the Name column below, may not be dragged to a new location. Grouped columns are always appear as the first columns in the table and are positioned according to the order in the Groups dialog, shown below.

Change column widths by hovering over the right edge of each and dragging to the left or right.

Click the column title to sort in ascending or descending order.

Click the triangle to close the group. See Note below on group aggregations.

"N/A" represents a Fitted or Probability model.

Hover over a status bar to display the number of models that solve successfully or stopped with an error.

Models

Overview

Models

Models

Customize list using filters, columns, and groups.

Filter

Columns

Groups

Export

Total Solve Time

42.482 s

Total Runs

18

15 Success

3 Error

0 Stopped

Name	Last Modified	Kind	Language	Solve Time (Total)	Runs Summary
✚ AirlineHub2Example (1)	8/23/2022, 1:10:24 PM	Optimization	RASON	50 ms	
✚ AirlineHubConic3Example (1)	8/23/2022, 1:11:52 PM	Optimization	RASON	35 ms	
✚ AssociationRules (1)	8/23/2022, 1:25:06 PM	Datamining	RASON	38 ms	
✚ CollegeFundGrowth2(Sim) (3)	8/23/2022, 1:15:39 PM	Simulation	RASON	153 ms	
	8/23/2022, 1:16:07 PM	Simulation	RASON	135 ms	
	8/23/2022, 1:16:22 PM	Simulation	RASON	106 ms	
✚ DecisionTreeClassification (3)	8/23/2022, 1:28:47 PM	Datamining	RASON		
	8/23/2022, 1:29:42 PM	Datamining	RASON	86 ms	
	8/23/2022, 1:29:45 PM	N/A	Fitted		
✚ DTDataSourceExample (1)	8/23/2022, 1:23:20 PM	Decision	RASON	15 ms	
✚ DTLoanRecommendExample (1)	8/23/2022, 1:23:59 PM	Decision	RASON	12 ms	
✚ GBMSimpleModel(Sim) (1)	8/23/2022, 1:22:33 PM	Simulation	RASON	397 ms	
✚ Inventory2Example (1)	8/23/2022, 1:19:20 PM	Optimization	RASON	8,484 ms	
✚ KMeansClustering (2)	8/23/2022, 1:30:10 PM	Datamining	RASON	53 ms	
	8/23/2022, 1:30:25 PM	N/A	Fitted		
✚ LinearRegression (3)	8/23/2022, 1:31:05 PM	Datamining	RASON		
	8/23/2022, 1:31:43 PM	Datamining	RASON	66 ms	
	8/23/2022, 1:31:47 PM	N/A	Fitted		
✚ optSimWorkflow (1)	8/23/2022, 1:24:43 PM	Flow	RASON	176 ms	
✚ PortfolioOptExample (1)	8/23/2022, 1:20:03 PM	Optimization	RASON	107 ms	

Right click any model in the list to open the model in the Editor tab (where you can solve or modify the existing model) or view details about the model and its results.

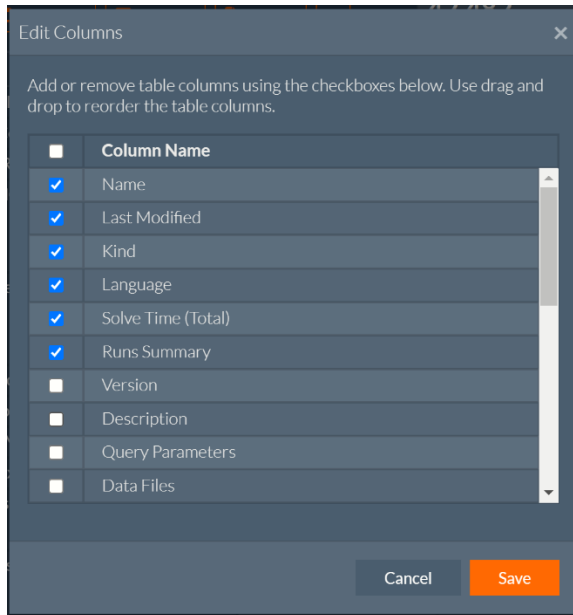
Name	Last Modified	Kind	Language	Solve Time (Total)	Runs Summary
	5/10/2021, 5:14:02 PM	Datamining	RASON	107 ms	
	5/10/2021, 5:16:41 PM	Datamining	RASON	159 ms	
	5/11/2021, 12:13:19 PM	Datamining	RASON	168 ms	
	9/9/2021, 10:47:40 AM	Datamining	RASON	221 ms	
✚ Arima1 (1)	1/14/2020, 12:23:19 PM	Unknown	Unknown	2,277 ms	
✚ Arima2 (1)	1/15/2020, 9:50:28 AM	Unknown	Unknown	835 ms	
✚ ArimaPostFM (1)	5/14/2021, 11:10:19 AM	Datamining	RASON	203 ms	
✚ AssociationRules (4)	1/3/2020, 12:55:14 PM	Unknown	Unknown	585 ms	
	3/23/2020, 3:03:05 PM	Unknown	Unknown	297 ms	
	4/8/2020, 6:30:36 AM	Unknown	Unknown	695 ms	
	5/18/2021, 12:56:40 PM	Datamining	RASON	51 ms	
✚ AutoInsuranceCalculationExample (2)	3/10/2022, 9:29:29 AM	Unknown	RASON		
	3/10/2022, 9:29:50 AM	Unknown	RASON		
✚ AutoInsurancePolicyExample (55)	3/10/2022, 9:44:05 AM	Unknown	RASON	0 ms	
	3/10/2022, 12:00:00 AM	Unknown	RASON	0 ms	
	3/10/2022, 12:40:22 PM	Unknown	RASON		
	3/10/2022, 12:40:35 PM	Unknown	RASON		
	3/10/2022, 12:40:48 PM	Unknown	RASON		



Add Filter Descriptions

Champion	If true, displays only models designated as “champion”. For a definition of “champion” see the next section.
Container	Filters models based on the storage container name.
Creator	Filters on name of model creator.
Data Connections	Filters on the name of the Data Connection.
Data Files	Filters on name of data files utilized.
Deleted	Filters on deleted models.
Description	Filters on contents of the Description field.
Kind	Filters on the ModelKind field: ModelKind = fitted (fitted data science model), Excel (posted Excel model) or RASON (RASON model)
Language	Filters models by Language type. Language = Excel (posted Excel models), Fitted (fitted data science models), Probability (probability distribution model) , RASON (RASON model) or unknown (error occurred).
Last Modified	Filters by the model’s last modified date.
Name	Filters by model name.
Query Parameters	Filters by query parameters.
Runs (Total)	Filters by total number of runs.
Schedule	Filters by scheduled time to run.
Solve Date	Filters by solve date.
Solve Status	Filters by solve status: Error, Invalid, Stopped, Success or Unknown.
Solve Time (ms)	Filters by time taken to solve the model.
Version	Filters by version number of model, i.e. 2020-10-09-19-32-24-345489. For example, to filter on all models created/modified in 2020, use “contains 2020”.

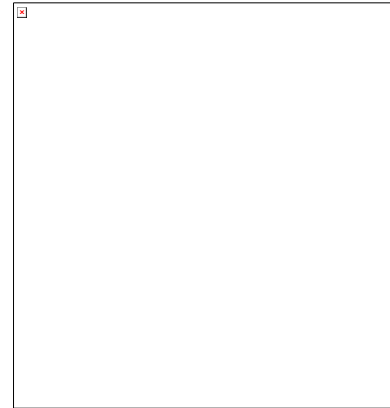
- Click **Columns** to add or delete columns from the display. Drag and drop the columns in the desired order.



Example: Add Version column/Remove Language column.

- Click **Columns**.
- Uncheck **Language**.
- Select **Version**.

The Version column will be added to the end of the existing display and Language will be removed.

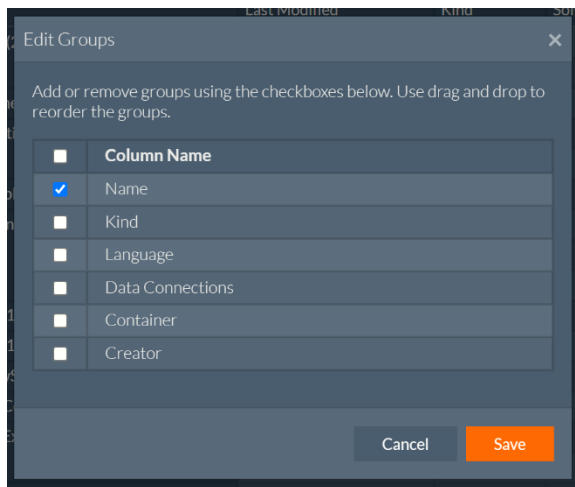


Edit Columns Descriptions

Name	Displays the name of the model as given in the modelName field in the RASON model.
Last Modified	Displays the date and time that the model was last modified.
Kind	Displays the model kind: ModelKind = fitted (fitted data science model), Excel (posted Excel model) or RASON (RASON model) Kind = Unknown if the RASON model has been POSTed but not solved.
Language	Displays the model Language Language = Excel (posted Excel models), Fitted (fitted data science models), Probability (probability distribution model) , RASON (RASON model) or unknown (error occurred).
Solve Time	Displays the total model solve time.
Runs Summary	Displays a bar graph indicating the number of successfully runs, runs ending in an error or runs where the model execution was stopped prematurely.
Version	Displays the model version number.
Description	Displays text from the modelDescription field in the RASON model.
Query Parameters	Displays query parameters used, if any.
Data Files	Displays names of data files utilized, if any.
Data Connections	Displays the name of the data connection used, if any.
Schedule	Displays the schedule for the model to be run, if a schedule exists.
Champion	Displays a star in the column if the model has been designated as a champion.

Container	Displays the storage container where the model is saved. (Used for Organizational Accounts only. For more information see Organization Accounts in the Appendix.)
Creator	Displays the user name of the model creator.
First Solved	Displays the date and time when the model was first solved.
Last Solved	Displays the date and time the model was last solved.
Solve Time (Min)	Displays the minimum time the model took to solve.
Solve Time (Max)	Displays the maximum time the model took to solve.
Solve Time (Avg)	Displays the average time the model took to solve.
Runs (Total)	Displays the total number of runs. Includes successful runs, runs that stopped with an error, runs that were stopped prematurely and invalid runs.
Runs (Success)	Displays the total number of successful runs for the model.
Runs (Error)	Displays the number of runs that ended in an error.
Runs (Stopped)	Displays the number of runs where the model was stopped prematurely.
Runs (Invalid)	Displays the number of invalid runs for the model.

- Use **Groups** to add a new category to the display. Drag and drop the groups in the desired order.



Example: Group models by Kind, then Name.

- Click **Groups**.
- Select Name and Kind.
- Drag Name below Kind.

Click OK to display models grouped first by Kind (Datamining, Optimization, Simulation, etc.) and then by Name.



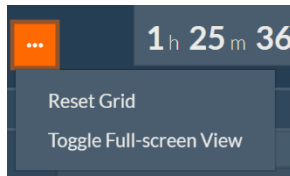
Edit Groups Descriptions

Name	Select Name to group models by the name passed to the modelName field in the RASON model.
Kind	Select Kind to group models by “kind”: fitted (fitted data science model), Excel (posted Excel model) or RASON (RASON model)
Language	Select Language to group models by “language”: Excel (posted Excel models), Fitted (fitted data science models), Probability (probability distribution model) , RASON (RASON model) or unknown (error occurred).
Data Connection	Select Data Connection to group models by the Data Connection name.

Container	Select container to group models by storage container. (Organizational Accounts)
Creator	Select Creator to group models by user name of creator.

Notes on Groups:

1. If a group is closed, i.e. not expanded, all column values are summarized for all models in the group. For example: total time will be aggregated and reported for all models, most recent date will be taken from the most recent model in the group, the most recent version will be the most recent version in the group, etc.
 2. A “grouped” column may not be dragged to a new location. Grouped columns are always appear as the first columns in the table and are positioned according to the order in the Groups dialog, shown above.
- Click **Export** to download the list of models to the Excel file, models-list.xls.
 - Click ... to reset the grid back to the default settings or to view the grid in full-screen format.



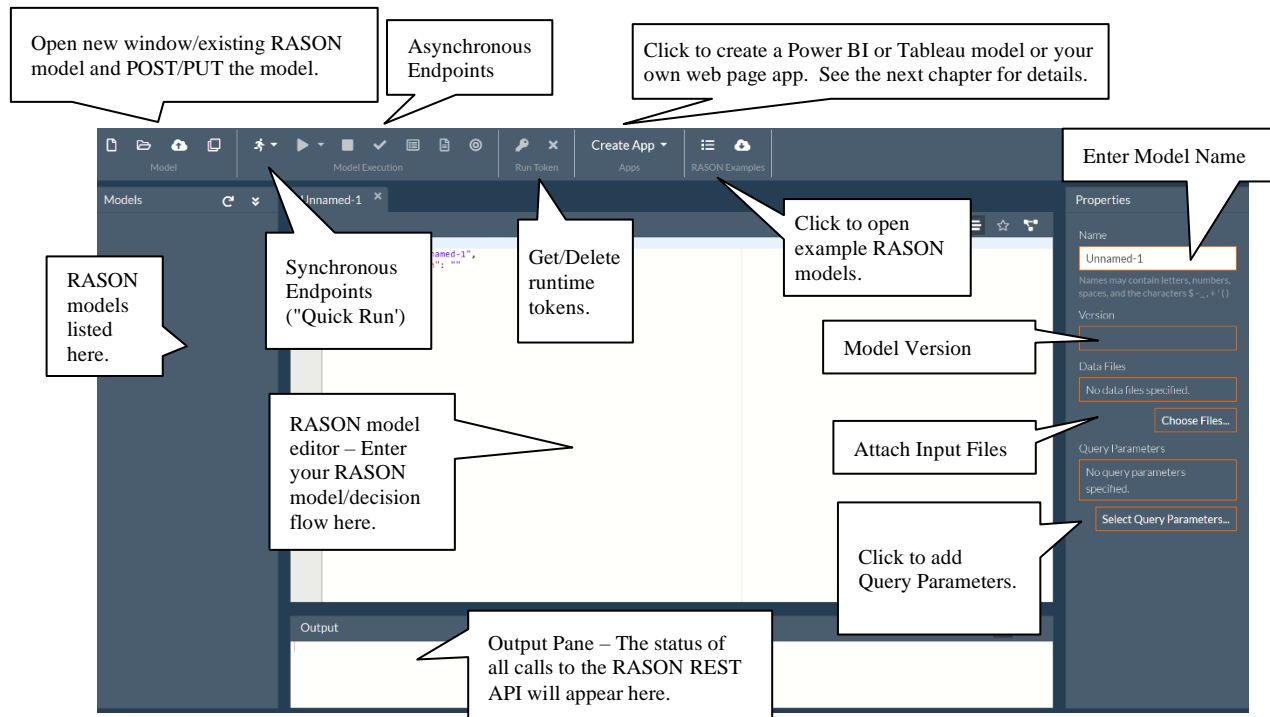
RASON Editor Page

One of the simplest ways to call the RASON Rest API is through the Editor Page on RASON.com. It's easy to type or drag your own RASON model into the RASON model editor or open one of our example RASON standalone models, then use the command buttons to analyze, solve, check the status and/or obtain the results from an optimization, simulation, simulation optimization, data science or decision table model.

Click the Editor Page to open the RASON web IDE.



The blank Web IDE appears.

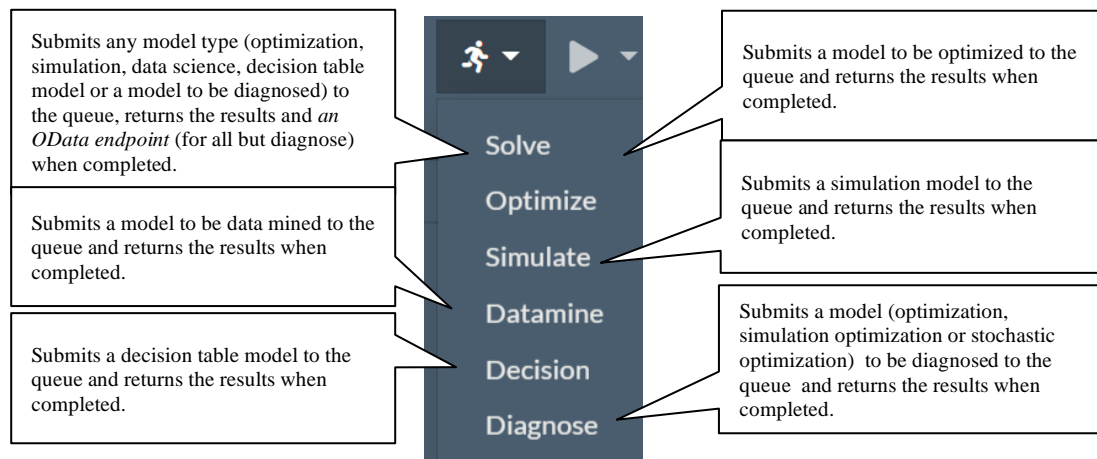


Models entered or dragged onto the RASON model editor in the Web IDE may be solved or analyzed using either a 'Quick Solve' REST API endpoint (Quick Run) or a standard REST API endpoint (Model Execution). The endpoints displayed in the command button intellisense are the actual REST API endpoints that you would call if solving a RASON model from within your own application. (See the next section within this chapter for a complete description of each RASON REST API endpoint.

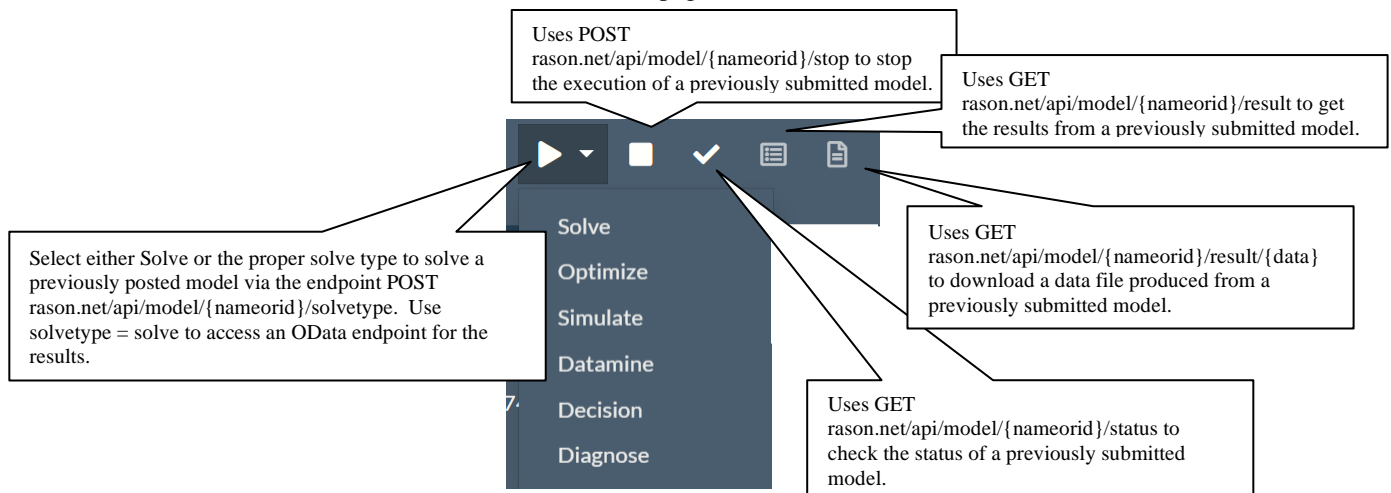
A model name may be entered into the Name field (on the Properties pane). Once the model has been posted (saved) to the RASON server, the version ID will appear under Version. Click Choose Files to upload any data files utilized in the mode. Click Select Query Parameters when calling the POST `rason.net/api/model/id/solvetype` endpoint to control the content of the final results and if and where intermediate decision flow results are stored. (See the next chapter Creating and Running a Decision Flow for more information.)

Note: When dragging a file into the Editor window, a copy of the model will be posted to the RASON server, *even if the original file is saved on a user's OneDrive account*. This new model will not overwrite or update the file saved on the OneDrive account.

Click the down arrow on the first icon, the Quick Run icon, to display the six "Quick Solve" REST API endpoints. 'Quick Solve' endpoints are synchronous and should only be used with models that solve very quickly, i.e. within a few seconds. There is a time limit of 30 seconds on all 'Quick Solve' calls which means that if your model takes longer than 30 seconds to solve, the RASON Server will return the result: `"status": { "code" : 6, "id": "285573+2020-02-04-12-52-02-911912", "codeText" : "Solver stopped at user's request." }`. While these endpoints do create a "resource ID", this ID will never be referenced by the RASON client, it will only appear in the JSON response and the RASON event log. This means that a model optimized, simulated, datamined, recalculated or diagnosed with a 'Quick Solve' endpoint cannot be saved, edited, retrieved or solved at a later time.



For all but the smallest / most quickly solved models, you'll need to use the asynchronous set of REST API calls as used in the Model Execution section of the Editor page.



When solving any type of problem, you'll first need to make a call to the REST API endpoint POST `rason.net/api/model`, within the Model section of the ribbon, which will create a "resource ID". This "model ID" or "resource ID" is used to identify versions of RASON models (where a model's text is changed) and also model instances (where a model is run with new data). These IDs also contain an embedded timestamp which can be used to identify version history and may be used for "auditability" purposes when identifying the model instance used to generate a RASON result.

In past versions of RASON, the model ID consisted of a series of digits (i.e. 2590+2015-05-15-20-33-31-034142); there was no way to "name" a model thereby creating an easy way to identify a model. RASON Decision Services supports both named and unnamed models. A model becomes "named" either by including `modelName: "name"` in the RASON model text and then calling POST `rason.net/api/model` or by simply calling POST `rason.net/api/model/<name>` or both. (It's possible to also enter the model name in the Name field on the Properties pane.) Either call returns a Location header with a new resource ID that identifies this unique model instance, for example: 2590+productMixExample+2020-01-17-22-10-22-683772

The "name" must be unique among models with a user's account. An unnamed model, or a model not containing `modelName`, has no name. For more information on naming a model, see Using the RASON REST API section below. Every model (named or unnamed) that is POSTed to the RASON Server will have an assigned resource ID, though for "quick solves" via POST `rason.net/api/solvetype` that resource ID will never be referenced by the RASON client.

After the model has been posted, then the solve can be started via the new REST API endpoint `POST rason.net/api/model/{nameorid}/solve`, where `{nameorid}` is either the model's name or the ID of the desired version of the model. The call to this endpoint will return a Location header with a new resource ID that identifies this unique model instance (model bound to data).

You can check on the model's progress at any time with the REST API endpoint `GET rason.net/api/model/{nameorid}/status` and obtain results when finished with the REST API endpoint `GET rason.net/api/model/{nameorid}/result`.

A Note on RASON Organization Accounts

As discussed in the RASON Organization Account chapter that occurs later in this guide, with an Organization Account, your company's models and data are maintained privately in the company's own Azure Storage Account, isolated from all other RASON users. Using standard Azure Role-Based Access Control (RBAC), the company can assign "roles" to different users that give them selective access to storage containers where their models and data are saved.

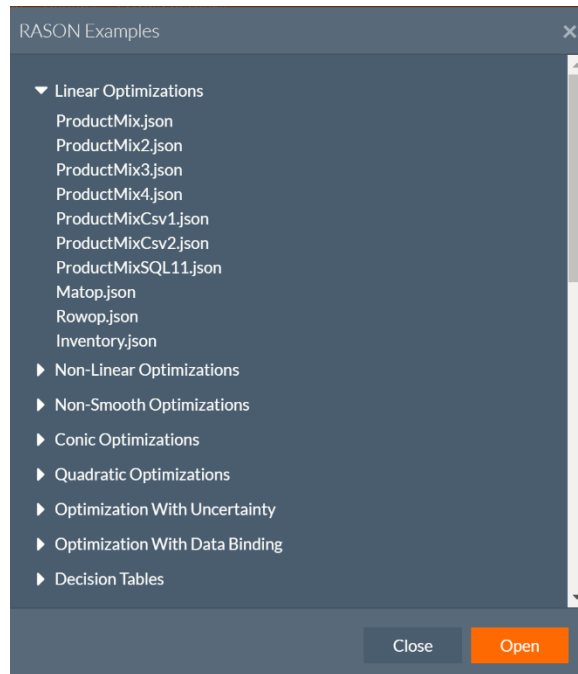
In the RASON Model Editor, your company's authorized users will be able to "see" the containers they can access (read-only or read/write), and easily create, update or delete models within those containers in accordance with their roles. See the list below to see how various user permissions will affect the Editor page functionality.

Editor Tab

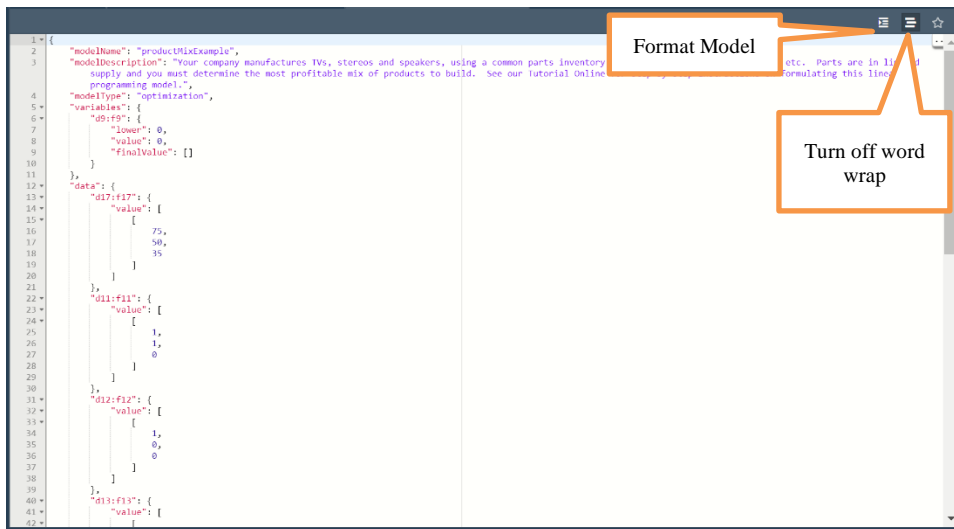
- Read Only User
 - Does not have access to the "my models" container
 - POST and PUT menu items are disabled
 - "Quick Solve" menu enabled
- Container without Write Permission
 - POST to {container name} and PUT to {container name} menu items disabled.
- Container without Delete Permission
 - Models list container "Delete All..." and "Delete" context menu items disabled.
- Current model in container without Write permission
 - Async solve menu items disabled.
 - Score button disabled.
 - Create and Delete run token buttons disabled.
 - Create App menu disable.

Running Example Models on the "Editor" Page

To open any example model discussed in this guide, click the down arrow in the RASON Examples drop down menu. (Note that RASON models may also be dragged onto the Editor window.)



If you select **ProductMix.json** beneath Linear Optimizations from the list of example models. The ProductMix.json example model is loaded into the blank RASON model editor. To format a model and make it easier to read, click *Format Model*. To turn off "word wrap", click *Turn word wrap off*.



To quickly solve the model, click the down arrow next to Quick Run (top left) and select either Optimize or Solve to run the Quick Solve endpoint POST rason.net/api/optimize or [\(/solve\)](https://rason.net/api/solve), respectively. Recall that this endpoint can be used to run an optimization that will complete in less than 30 seconds and retrieve the result without creating a resource ID. Since the model is linear, and we did not specify a specific engine, the LP/Quadratic engine was used to solve the model. For more information on how to select an engine, see "Engine Settings" within the *RASON Model Components* chapter in the *RASON Reference Guide*.

The result "Solver found a solution. All constraints and optimality conditions are satisfied" is displayed in the Output editor for codeText. This result means that no other solution exists that is better than the solution found. (For more information on this and all possible Solver Result messages, please see the *RASON Reference*

Guide.) For more information on this example model, see the earlier chapter, *Defining Your Model*. To clear this result from the editor click the *Clear Output* button (or the X) in the bottom right.

```
{
  "status": {
    "code": 0,
    "id": "2590+2020-01-17-22-05-08-311425",
    "codeText": "Solver found a solution. All constraints and
    Optimality conditions are satisfied."
  },
  "variables": {
    "d9:f9": {
      "finalvalue": [
        [200, 200, 0]
      ]
    }
  },
  "objective": {
    "d18": {
      "finalvalue": 25000 }
    }
  }
}
```

To solve more complex models, you'll need to use the asynchronous endpoints.

POSTing a RASON Model



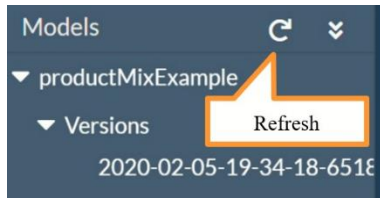
To post a model of any type and obtain a resource ID, click endpoint POST `rason.net/api/model`. In return you will receive a similar result. For more information on the returned fields see the description of this endpoint within the chapter, *Using the REST API*.

Executing ajax call...: POST `https://rason.net/api/model`

```
{
  "ModelId": "2590+productMixExample+2020-02-05-19-34-18-651834",
  "ModelName": "productMixExample",
  "ModelDescr": "Your company manufactures TVs, stereos and speakers,
  using a common parts inventory of power supplies, speaker cones, etc.
  Parts are in limited supply and you must determine the most profitable mix
  of products to build. See our Tutorial Online for step-by-step
  instructions on formulating this linear programming model.",
  "ModelFiles": [],
  "RuntimeToken": "",
  "ModelType": "Origin",
  "ModelKind": "RASON",
  "IsChampion": false,
  "ParentModelId": null,
  "QueryString": "",
  "ModelContainer": null
}
```

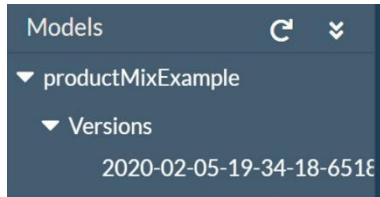
Calling this endpoint creates the Resource ID, 2590+productMixExample+2020-02-05-19-34-18-651834, where productMixExample matches the modelName component in the RASON model (screenshot above).

The model name and version appears on the left of the Editor page beneath Models. (Click Refresh if needed.)



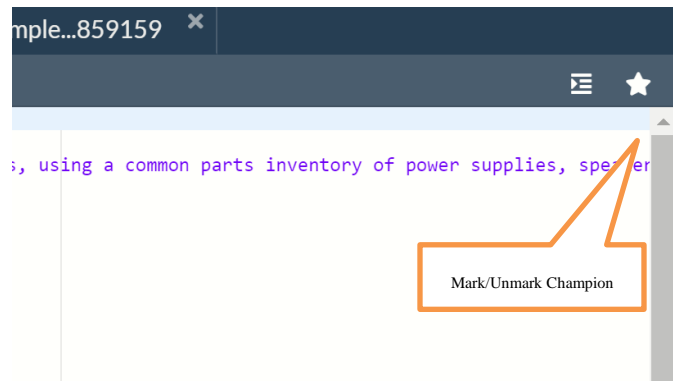
Note: Unnamed models will appear under "Unnamed" in the Models pane.

As mentioned earlier, every model (named or unnamed) that is posted to the RASON Server will have a resource ID. Every model that is updated on the RASON Server (PUT `rason.net/api/model/{nameorid}`) will have a new ID for the new version; the old ID will identify the old version.



What is a Model Champion?

An API call using a RASON model name always refers to the "champion" version of that model. If no champion is designated, then the most recent version of the model is used. The "champion" is designated in the Models pane on the Editor page with a ★. The user can designate a specific version (resource ID) as the current "champion" by clicking the desired version under Models to open, then clicking the *Mark Champion* icon on the right of the Model Editor thereby marking that resource ID as the current "champion" (i.e. the model to use when future direct API calls specify the model name) then the user can create newer versions as "challengers".

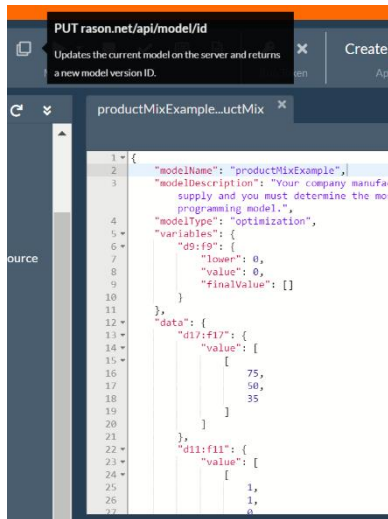


This button toggles between "Unmark Champion" and "Mark Champion" depending on if the version selected has been marked as a champion or not.

To mark a different version as the champion, simply select the desired version and click Mark Champion.

Editing a previously POSTed Model

Change the first element in the d17:f17 data component from 75 to 100, then use the REST API endpoint PUT `rason.net/api/model/{nameorid}` to update the model version.

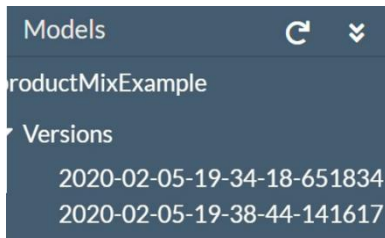


You'll receive a result similar to the following.

Executing ajax call...: PUT https://rason.net/api/model

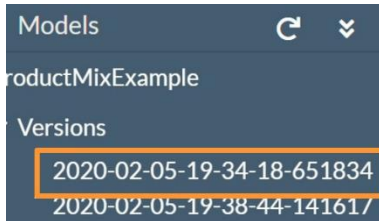
```
{
  "ModelId": "2590+productMixExample+2020-02-05-19-38-44-141617",
  "ModelName": "productMixExample",
  "ModelDescr": "Your company manufactures TVs, stereos and speakers,
  using a common parts inventory of power supplies, speaker cones, etc.
  Parts are in limited supply and you must determine the most profitable mix
  of products to build. See our Tutorial Online for step-by-step
  instructions on formulating this linear programming model.",
  "ModelFiles": [],
  "RuntimeToken": "",
  "ModelType": "Version",
  "ModelKind": "RASON",
  "IsChampion": false,
  "ParentModelId": "2590+productMixExample+2020-02-05-19-34-18-651834",
  "QueryString": "",
  "ModelContainer": null
}
```

Every model that is updated on the RASON server via PUT will have a new ID for the new version; the old ID will identify the old version. You'll notice a new version now appears in the list. An API call using the model name "ProductMixExample" will refer to this newly created version, 2020-02-05-19-38-44-141617. Each time a new version is created, that version becomes the "latest" and will be used in any API call to a named model, **unless** a specific model is designated as the "champion" using the "Mark Champion" button, or the complete Model ID is used, i.e. "ModelId": "2590+productMixExample+2020-02-05-19-38-44-141617".

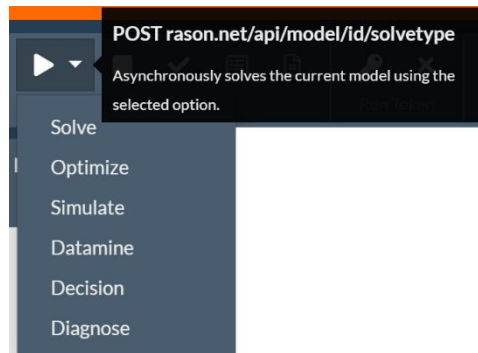


Solving a RASON Model

Click the previous version under Models (2020-02-05-19-34-18-651834) to open the previously posted model.



Click the down arrow next to the Play button and select Solve from the menu. This endpoint solves the model (of any modeltype) and automatically creates an OData endpoint. See the next chapter for more information on this new endpoint. Alternatively, the model type (optimize) could have been selected.



The following appears in the Result pane.

Executing asynchronous solve: POST

`https://rason.net/api/model/2590+productMixExample+2020-02-05-19-34-18-651834/solve`

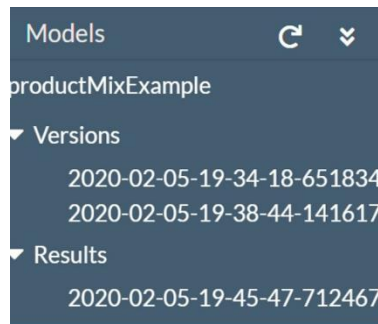
```
{
  "ModelId": "2590+productMixExample+2020-02-05-19-38-44-712467",
  "ModelName": "productMixExample",
  "ModelDescr": "Your company manufactures TVs, stereos and speakers,
using a common parts inventory of power supplies, speaker cones, etc.
Parts are in limited supply and you must determine the most profitable mix
of products to build. See our Tutorial Online for step-by-step
instructions on formulating this linear programming model.",
  "ModelFiles": null,
  "RuntimeToken": "",
  "ModelType": "Instance",
  "ModelKind": "RASON",
  "IsChampion": false,
  "ParentModelId": "2590+productMixExample+2020-02-05-19-34-18-651834",
  "QueryString": "",
  "ModelContainer": null
}
```



```
}
```

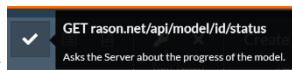
When a model is solved via POST `rason.net/api/model/{nameorid}/solvetype` (where `solvetype` = Solve, Optimize, Simulate, Datamine, Decision or Diagnose and ID = the ID of the desired version of the model) the call returns a Location header with a new resource ID that identifies this unique model instance (model bound to data), as shown above. This new ID will be used in subsequent calls to GET `rason.net/api/model/id/status` and GET `rason.net/api/model/id/result`.

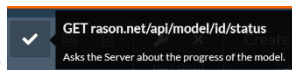
The Models pane has been updated with the new model instance appearing beneath Results. The timestamp embedded in this ID records the date/time when the model was run, and when any regularly-updated data sources (OneDrive, CDS, CDataCloudHub) were accessed by the model instance.



Now we will use this new model instance to check the status of the solve and obtain our results using GET `rason.net/api/model/{nameorid}/status` and GET `rason.net/api/model/{nameorid}/result`.

Checking the Status of a RASON Model



Click  to check the status of the previously submitted model. This endpoint returns:

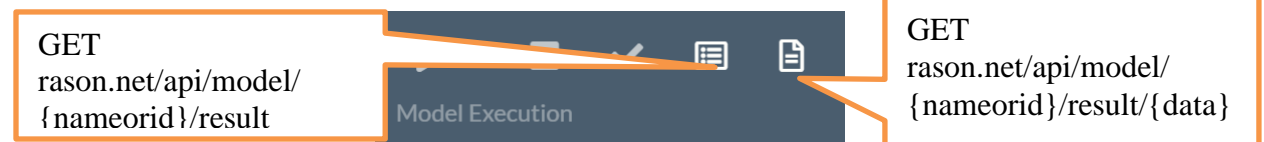
```
{"status": "Complete" }
```

Status: Complete means that the optimization has been completed. If the optimization was still in progress we would have received an incomplete status with an intermediate result, for example:

```
{
  "status": "Incomplete",
  "Milliseconds" : -2147483648,
  "Iterations" : 0,
  "Subproblems" : 287,
  "LocalSolutions" : 8,
  "Objective" : 3849
}
```

Obtaining the Results of a RASON Model

Click the entry under Results to enable the two result endpoints, GET `rason.net/api/model/{nameorid}/result` and GET `rason.net/api/model/{nameorid}/result/data`.



Click GET `rason.net/api/model/{nameorid}/result` to display the results in the Model Editor. Notice that the RASON response produced by the RASON Server includes the resource ID of the model instance that was run. (This is true for "quick solves" as well as "long solves".) This model resource ID is stored in the User Metrics table on the My Account page.

```
{
```

```

"status": {
  "id": "2590+productMixExample+2020-02-05-19-45-47-712467",
  "code": 0,
  "codeText": "Solver found a solution. All constraints and
    optimality conditions are satisfied.",
  "solveTime": 80
},
"results": {
  "d18": ["finalValue"],
  "d9:f9": ["finalValue"]
},
"d9:f9": {
  "objectType": "dataFrame",
  "name": "d9:f9",
  "order": "col",
  "colNames": ["finalvalue"],
  "colTypes": ["double"],
  "indexCols": null,
  "data": [
    [200, 200, 0]
  ]
},
"d18": {
  "objectType": "dataFrame",
  "name": "d18",
  "order": "col",
  "colNames": ["finalvalue"],
  "colTypes": ["double"],
  "indexCols": null,
  "data": [
    [25000]
  ]
}
}

```

Note that the endpoint `POST rason.net/api/model/{nameorid}/solve` returns the results as `"objectType": "dataframe"` and automatically creates an OData endpoint for easy retrieval, querying, etc.

If using the endpoint `POST rason.net/api/model/{nameorid}/optimize`, the results would be as follows. For more information on each endpoint, see the next section below.

```

{
  "status": {
    "code": 0,
    "id": "2590+productMixExample+2020-02-05-19-45-47-712467",
    "codeText": "Solver found a solution. All constraints and
      optimality conditions are satisfied."
  },
  "variables": {
    "d9:f9": {
      "finalValue": [
        [200, 200, 0]
      ]
    }
  },
  "objective": {
    "d18": {
      "finalValue": 25000
    }
  }
}

```

```
}  
}
```

If your RASON model contains an exported data source, you would use the endpoint GET `rason.net/api/model/{nameorid}/result/{data}` to download the exported file. See the next section for more details on this endpoint and all endpoints offered in RASON.

Viewing Results in ODATA

Results returned in the Web Editor (on the Editor tab of [www. RASON.com](http://www.RASON.com)) are always returned in the easy-to-read, ODATA results viewer. Click the Table field down arrow to select the results to be displayed. From here, you can explore and filter the displayed columns. You can also choose to view the results in JSON format by clicking the "Show JSON results" icon on the top right corner of the Editor window.

The screenshot displays the RASON Web Editor interface. At the top, there's a header bar with a 'Table' dropdown set to 'd9_f9' and a 'Columns' dropdown set to 'All columns'. Below this is a table with three rows and two columns: 'ID' and 'finalValue'. The rows contain values (0, 200), (1, 200), and (2, 0). A blue callout box points to the 'finalValue' column header with the text 'Click to reformat column values.' To the right of the table, another blue callout box points to a 'Show JSON Results' icon with the text 'Show JSON Results'. Below the table is a pagination bar showing 'Page 1 of 1' and 'Displaying 1 to 3 of 3 items.' At the bottom, there's an 'Output' window showing the JSON response from the API. The JSON includes status information, a results array with 'd10' and 'd9:f9' fields, and engine details for the 'Gurobi Solver'.

ID	finalValue
0	200
1	200
2	0

```
Getting model results: GET https://rason.net/api/model/2590+productMixExample+2023-09-07-20-07-45-235021/result  
{  
  "status": {  
    "id": "2590+productMixExample+2023-09-07-20-07-45-235021",  
    "code": 0,  
    "codeText": "Solver found a solution. All constraints and optimality conditions are satisfied.",  
    "solveTimestamp": "2023-09-07-20-08-07-093839",  
    "solveTime": 6996  
  },  
  "results": {  
    "d10": ["finalValue"],  
    "d9:f9": ["finalValue"]  
  },  
  "engine": "Gurobi Solver",  
  "d9:f9": {  
    "objectType": "dataFrame",  
    "name": "d9:f9",  
    "order": "col",  
    "colNames": ["finalValue"],  
    "colTypes": ["double"],  
    "indexCols": null,  
    "data": [  
      [200, 200, 0]  
    ]  
  }  
}
```

Results Formatting in Web Editor

Results in the OData viewer may be of type number, date or Boolean.

Number Formatting

Number formatting has been added as a column option in the OData results viewer. This new feature provides several advantages when sorting or charting results.

Click the vertical ellipsis on the right side of the column header to display the column format dialog.

The 'Format Column' dialog box has a 'Sample' field with the value '200'. The 'Type' dropdown menu is open, showing the following options: None (highlighted), Number, Date, and Boolean. The 'Format' field is empty. At the bottom are 'Cancel' and 'Ok' buttons.

Select a number format from the predefined list of formats or enter a custom format string into the Format field. Click Ok to set the format.

Format as Number

The 'Format Column' dialog box has 'Type' set to 'Number'. The 'Format' field displays a list of predefined number formats: 0, 0.00, #,##0, #,##0.00, 0%, 0.00%, #,##0%, #,##0.00%, \$0, and \$0.00. At the bottom are 'Cancel' and 'Ok' buttons.

Format as Date

The 'Format Column' dialog box has 'Type' set to 'Date'. The 'Format' field displays a list of predefined date formats: M/d/yyyy, EEEE MMMM d, yyyy, yyyy-MM-dd, M/d, M/d/yy, MM/dd/yy, d-MMM, d-MMM-yy, MMM-yy, and MMM-yyyy. A blue callout box with the text 'Enter custom formats here.' is overlaid on the list. At the bottom are 'Cancel' and 'Ok' buttons.

Format as Boolean

The 'Format Column' dialog box has 'Type' set to 'Boolean'. The 'Format' field displays a list of predefined boolean formats: true,false, yes,no, and 1,0. At the bottom are 'Cancel' and 'Ok' buttons.

Date Formatting

Date formatting is particularly important because it converts a date string into a date object that can be sorted properly and used in charts utilizing dates, such as a line chart demonstrating sales over time.

Unformatted Date and Number values

ID	Name	Fitted_Passengers
0	Jan-1949	100.65468953538863
1	Feb-1949	111.21427411778255
2	Mar-1949	130.6022865702808
3	Apr-1949	122.28014955550344
4	May-1949	122.5515743775264
5	Jun-1949	142.39758811492953
6	Jul-1949	157.76749063832563
7	Aug-1949	158.66515599020403

Format Column

Sample: Jan-49

Type: Date

Format: MMM-yy

M/d/yyyy
EEEE, MMMM d, yyyy
yyyy-MM-dd
M/d
M/d/yy
MM/dd/yy
d-MMM
d-MMM-yy
MMM-yy

Formatted Date and Number values

ID	Name	Fitted_Passengers
0	Jan-49	100.65
1	Feb-49	111.21
2	Mar-49	130.60
3	Apr-49	122.28
4	May-49	122.55
5	Jun-49	142.40
6	Jul-49	157.77
7	Aug-49	158.67

Format Column

Sample: 100.65

Type: Number

Format: 0.00

0
0.00
#,###
#,###.##

Note: The date format must match the date string to convert the string to a date object. The date string is converted to a date object after a date format is selected and set the first time by clicking Ok. After the first conversion any date format may be used.

For example, since the original date format was MMM-YYYY, this format must first be selected. (Select the MMM-YYYY format in the drop down menu, then click OK. Then reopen the Format Column dialog and select the desired format.)

The OData viewer supports custom date formats. See the table below for supported formats.

Standalone token	Format token	Description	Example
S		millisecond, no padding	54
SSS		millisecond, padded to 3	054
u		fractional seconds, (5 is a half second, 54 is slightly more)	54
uu		fractional seconds, (one or two digits)	05
uuu		fractional seconds, (only one digit)	5
s		second, no padding	4
ss		second, padded to 2 padding	04
m		minute, no padding	7
mm		minute, padded to 2	07
h		hour in 12-hour time, no padding	1
hh		hour in 12-hour time, padded to 2	01
H		hour in 24-hour time, no padding	13
HH		hour in 24-hour time, padded to 2	13
Z		narrow offset	+5
ZZ		short offset	+05:00
ZZZ		techie offset	+0500
z		IANA zone	America/New_York
a		meridiem	AM

Standalone token	Format token	Description	Example
d		day of the month, no padding	6
dd		day of the month, padded to 2	06
E	c	day of the week, as number from 1-7 (Monday is 1, Sunday is 7)	3
EEE	ccc	day of the week, as an abbreviate localized string	Wed
EEEE	cccc	day of the week, as an unabbreviated localized string	Wednesday
M	L	month as an unpadded number	8
MM	LL	month as an padded number	08
MMM	LLL	month as an abbreviated localized string	Aug
MMMM	LLLL	month as an unabbreviated localized string	August
y		year, 1-6 digits, very literally	2014
yy		two-digit year, interpreted as > 1960 by default (also accepts 4)	14
yyyy		four-digit year	2014
yyyyy		four- to six-digit years	10340
yyyyyy		six-digit years	010340
G		abbreviated localized era	AD
GG		unabbreviated localized era	Anno Domini
GGGGG		one-letter localized era	A
kk		ISO week year, unpadded	17
kkkk		ISO week year, padded to 4	2014
W		ISO week number, unpadded	32
WW		ISO week number, padded to 2	32
o		ordinal (day of year), unpadded	218
ooo		ordinal (day of year), padded to 3	218
q		quarter, no padding	3
D		localized numeric date	9/6/2014
DD		localized date with abbreviated month	Aug 6, 2014
DDD		localized date with full month	August 6, 2014
DDDD		localized date with full month and weekday	Wednesday, August 6, 2014

Standalone token	Format token	Description	Example
t		localized time	1:07 AM
tt		localized time with seconds	1:07:04 PM
T		localized 24-hour time	13:07
TT		localized 24-hour time with seconds	13:07:04
f		short localized date and time	8/6/2014, 1:07 PM
ff		less short localized date and time	Aug 6, 2014, 1:07 PM
F		short localized date and time with seconds	8/6/2014, 1:07:04 PM
FF		less short localized date and time with seconds	Aug 6, 2014, 1:07:04 PM
,		literal start/end, characters between are not tokenized	"T"

Boolean Formatting

Boolean formatting allows 0/1 numbers to be formatted with a more descriptive label such as true/false or yes/no. To create a custom boolean format, enter two labels separated by a comma in the Format field.

Unformatted Boolean Results

ID	Name	Prediction__CATMEDV
0	Record 1	1
1	Record 2	1
2	Record 3	0
3	Record 4	0
4	Record 5	0
5	Record 6	0
6	Record 7	0

Formatted Boolean Results

ID	Name	Prediction__CATMEDV
0	Record 1	yes
1	Record 2	yes
2	Record 3	no
3	Record 4	no
4	Record 5	no
5	Record 6	no
6	Record 7	no

None Formatting

Selecting None formatting will remove a previously set column format.

Viewing Results in ODATA outside of the Web Editor

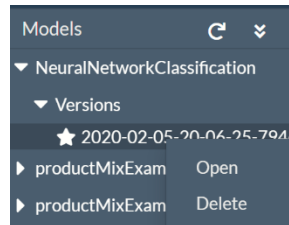
To view results in OData outside of the Web Editor, use the OData endpoint GET `rason.net/ODATA/result`. This endpoint requires two headers: the Authorization header and the resource ID. When both headers are passed to this endpoint, all results for the given model will be returned. Note: To authenticate and authorize access, you need to include the Authorization header in the request. An example of the Authorization header is: Authorization: bearer {your RASON token}

Once you make the request, you will receive a response that contains the requested evaluations represented as DataFrames. The response may be in JSON format if the data storage layer is set to JSON, or it may contain metadata for SQLite table names if the data storage layer is memory (DB_Memory) or file (DB_FILE).

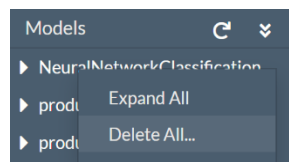
For more information on viewing OData results outside of the Web Editor, see the chapter, "Odata Service for RASON Decision Flows", that appears later in this guide.

Deleting a Version or Result

To delete a model version, decision flow or result from the Editor page, simply right click the version, decision flow or result in the Models pane and select Delete

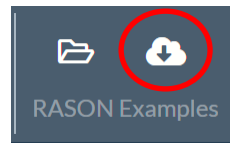


To delete all versions and results of a model or decision flow, right click the model name, then select Delete All.



Uploading Data from an External Data File

A few optimization, decision table and data science examples require input files such as CSV files or a Microsoft Excel™ workbook. If solving one of these data source example models, you must download the input files by clicking the **Download RASON Example Data** icon (cloud icon) on the Editor ribbon.



The compressed file, datafiles.zip, will be downloaded; unzip these files into a desired location. You can determine if the example uses an external data source by inspecting the dataSources section of the RASON model. As an illustration, open the example ProductMixCsv1.json under Linear Optimizations in the Example RASON model list.

This example imports 3 external dataSources, parts_data, invent_data and profit_data, exports 2 files containing the results of the solve, fcns_data and obj_data, and uses 1 file to import the starting values of the variables and then exports back the final variable values, vars_data.

```
{
  "modelName": "productMixCsv1Example",
  "modelType": "optimization",
  "modelDescription": "Example of using CSV table binding to read params
and save results",
  "datasources": {
    "parts_data": {
      "type": "csv",
      "connection": "ProductMixParts.txt",
      "indexCols": [
        "parts",
        "prods"
      ],
      "valueCols": [
        "qty"
      ]
    }
  }
}
```



```

    ],
    "direction": "import"
  },
  "invent_data": {
    "type": "csv",
    "connection": "ProductMixInventory.txt",
    "indexCols": [
      "parts"
    ],
    "valueCols": [
      "inventory"
    ],
    "direction": "import"
  },
  "profit_data": {
    "type": "csv",
    "connection": "ProductMixProfits.txt",
    "indexCols": [
      "prods"
    ],
    "valueCols": [
      "profits"
    ],
    "direction": "import"
  },
  "vars_data": {
    "type": "csv",
    "connection": "ResultVarsInit.txt",
    "indexCols": [
      "prods"
    ],
    "valueCols": [
      "initials"
    ],
    "direction": "import/export"
  },
  "fcns_data": {
    "type": "csv",
    "connection": "ResultFcns.txt",
    "direction": "export"
  },
  "obj_data": {
    "type": "csv",
    "connection": "ResultObj.txt",
    "direction": "export"
  }
},

```

All data sources containing the property "direction": "import" must first be uploaded. Data contained in an external source file may be uploaded by using *Choose Files* on the Properties pane on the right of the Editor page. External data can be used in two ways. 1. As constant values or data or 2. as initial starting points for decision variables in a nonlinear optimization model.

Currently the RASON modeling language supports nine different data types: "excel" (Microsoft Excel), "access" or "msaccess" (Microsoft Access), "odbc" (ODBC database), "OData" (OData endpoint), "mssql" (Microsoft SQL), "oracle" (Oracle database), CSV (Comma Separated Value), "json" (JSON file), or "xml" (XML file). For more information on using external data sources, please see Data Sources within the RASON Reference Guide.

Properties

Name

productMixExample

Version

Data Files

No data files specified.

Choose Files...

Query Parameters

name1=val1&name2=val2&...

To upload the files that are importing data onto the RASON server, click Choose Files on the right, browse to the location where you saved the files and click OK. Files that will contain exported data do not need to be uploaded.

Once the files are uploaded, solving the model proceeds as normal by calling:

- POST rason.net/api/model to post the model, along with the data files, to the RASON server and obtain a resource ID.
- POST rason.net/api/model/{nameorid}/solvetype where
 - solvetype = solve, optimize, simulate, datamine, decision or diagnose
 - nameorid = resource ID obtained from the call to POST rason.net/api/model
- GET rason.net/api/model/{nameorid}/status where
 - nameorid is the new model instance obtained from the call to POST rason.net/api/model/{nameorid}/solvetype
- GET rason.net/api/model/{nameorid}/result where
 - nameorid is the model instance obtained from the call to POST rason.net/api/model/{nameorid}/solvetype

The chart below displays which input files are required to successfully run the examples that appear when you select the Open RASON Example Model icon (the open folder).

Optimization Examples

Example Name	Input Files Required
RGProductMixCSV1.json	ProductMixParts.txt, ProductMixInventory.txt, ProductMixProfits.txt, ResultVarsInit.txt, ResultFens.txt and ResultObj.txt
RGProductMixExcel11.json	ProductMixExcel.xlsx

Data Science Examples

Example Name	Input Files Required
AssociationRules.json	Associations.txt
MSEExcelTable.json	hald.xlsx
Classification - DecisionTree.json, Classification - DiscriminantAnalysis.json, Classification - LogisticRegression.json, Classification - NaiveBayes.json, Classification - NearestNeighbors.json, Classification - NeuralNetwork.json,	hald-small-binary-train.txt and hald-small-binary-valid.txt

Classification - RandomTrees.json, kMeans.json, Univariate.json	
Classification - Bagging.json, Classification - Boosting.json, LogisticWrapping.json, Partitioning.json, PartitioningWithOversampling.json, Sampling.json, StratifiedSampling.json,	hald-small-binary.txt
DelimitedFile.json, Hierarchical.json, LinearWrapping.json, Regression - Bagging.json, Regression - Boosting.json, Factorization.json, OneHotEncoding-FileData.json,	hald-small.txt
JSONFile.json	hald-small-nested.json
LinearRegression.json, Regression - NearestNeighbors.json, Regression - NeuralNetwork.json, Rescaling.json, Regression-DecisionTree.json	hald-small-train.txt and hald-small-valid.txt
Arima.json, LagAnalysis.json,	close.txt
LatentSemanticAnalysis.json	tdmRomeo.txt
TfIdf-FileData.json	tm.txt
AddHoltWinters.json, DoubleExponential.json, Exponential.json, MovingAverage.json, MulHoltWinters.json, NoTrendHoltWinters.json	Airpass.txt
Binning.json	Binning.txt
CanonicalVariateAnalysis.json	Iris.txt
PrincipalComponentsAnalysis.json, JSONPCA.json	Irisfacto.txt
MSAccessDatabase.json	Ms-access-db.accdb

Twelve different examples illustrate how to score labeled and unlabeled data. All can be found under RASON Examples – Scoring. These examples all include at least one input data file and one fitted model. All fitted models are generated using a previous classification, regression or transformation RASON model. *You must first run the example that generates and POSTS the fitted model to the RASON server.* The following chart lists the scoring example name, the files required and the RASON example that generated the fitted model.

Scoring Example	Input Files Required	Example Generating Fitted Model
JSONClassifier.json	Data File: hald-small-binary.txt Data File: hald-small-binary-score.txt Fitted Model: classification-nearest-neighbors.json	Data Science- -- Classification – NearestNeighbors.json
JSONClusterizer.json	Data File: hald-small-binary.txt Fitted Model: clustering-kmeans.json	Data Science -- Clustering – KMeans.json
JSONForecaster.json	Data File: close.txt Fitted Model: arima.json	Data Science -- Time Series – Arima.json
JSONLinearRegression.json	Data File: hald-small-binary.txt Data File: hald-small-score.txt	Data Science -- Regression – LinearRegression.json

	Fitted Model: regression-linear-model.json	
JSONPCA.json	Data File: Irisfacto.txt Fitted Model: transformation-pca.json	Data Science -- Transformation – PrincipalComponentsAnlaysis.json
JSONRegressor.json	Data File: hald-small-score.txt Fitted Model: regression-nearest-neighbors.json	Data Science -- Regression – NearestNeighbors.json
JSONTransformer.json	Data File: hald-small.txt Fitted Model: transformation-rescaling.json	Data Science -- Transformation – Rescaling.json
PMMLClassifier.json	Data File: hald-small-binary-score.txt Fitted Model: classification-logistic-model.json	Data Science -- Classification – LogisticRegression.json
PMMLForecaster.json	Data File: close.txt Fitted Model: arima.json	Data Science -- Time Series – Arima.json
PMMLRegressor.json	Data File: hald-small-score.txt Fitted Model: regression-linear-neighbors.json	Data Science -- Regression – LinearRegression.json
PMMLTransformer.json	Data File: hald-small-valid.txt Fitted Model: transformation-rescaling.json	Data Science -- Transformation – Rescaling.json

Decision Table Examples

Decision Table Examples

Input Files Required

DT Loan Strategy Model and
DataSource.json

customers.txt and loans.txt

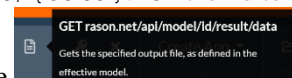
Note: Input files in a Data Science RASON model must not contain a path to a file location. Contents such as these have no meaning in the context of the RASON server.

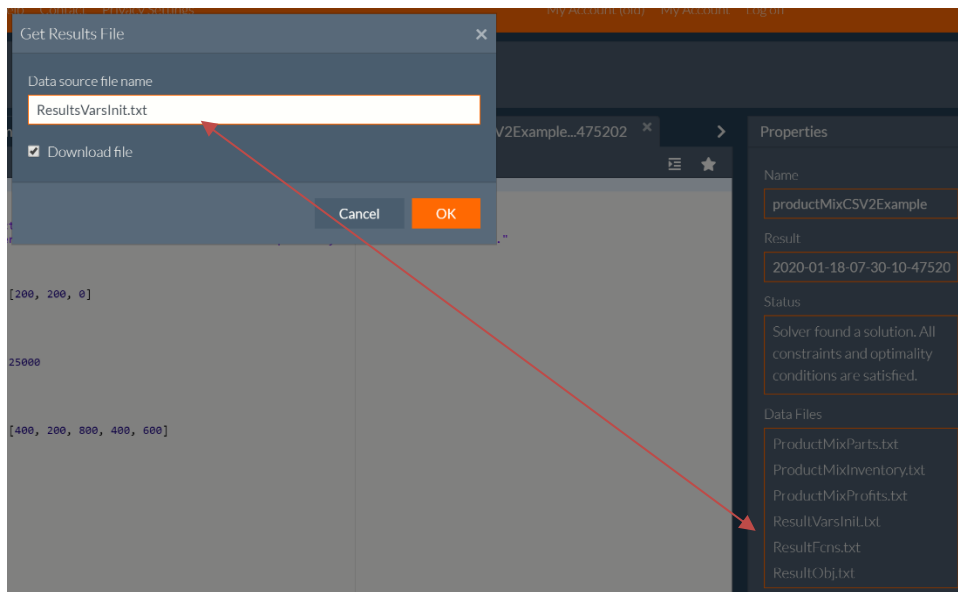
Downloading Final Results to an External Data File

Final results for the data science model, decision/uncertain variables, uncertain functions, constraints, and/or the objective function may also be *written* to an external source file. To retrieve this file(s), you must call the REST API endpoint, GET `rason.net/api/model/{nameorid}/result/{data}`. On the Editor

page click the desired entry beneath Results for the specified model, then click the icon on the ribbon. Enter the name of the output file into the *Get Results File* dialog, then click **OK**. The name of the output file entered into the Get Results File dialog must be identical to the datasource name in the RASON model.

For instance, the example RASON model below writes back the final variable values to ResultsVarsInit.txt. Note that the data source file name entered into the Get Results File dialog is the same as the file appearing in Data Files.





Note: If exporting to a file location such as 'Results/Export/influence-diagnostics.csv', you must enter the full path (Results/Export/influence-diagnostics.csv) into the edit box on the Get Results File dialog.

Select the *Download File* option to download the output file which can either be saved to your hard drive or opened.

Note: If your output file is a Microsoft Excel™ workbook (.xlsx) or Access Database™ file (.mdb) file, your results will only be available via the Download File option. For more information on writing data to an external file, see the *RASON Reference Guide*.

Using Query Parameters

RASON Web Services allows users to supply query parameters outside of the RASON model environment. To test these parameters, a Query Parameters field has been added to the Editor page on the right. For example, if using the RASON example, ProductMix4.json, you might want to temporarily change the values of the Profit array from 75, 50 and 35 to 100, 75 and 60?

First, we would need to add a binding property to the profits array within data, as shown in the example. The binding property within the RASON Modeling language serves two functions, one of which is to allow write access to a table or array outside of the RASON model environment. For example, passing binding: "get" to the profits array within the data section would allow us to change the values for this array directly, through a REST API call, or from within the Query Parameters section on the Editor page.

```
data: {
  profits: {
    dimensions: [3],
    value: [75, 50, 35],
    binding: "get",
    finalValue: []
  },
}
```

To change the array elements in profits to 100, 75, 50; you would pass the following to a REST API call, via standard HTTP GET parameters, for example:

```
$.get (https://rason.net/api/optimize?profits=100,75,50...
```

To change only one element, say the middle element from 50 to 60, use:

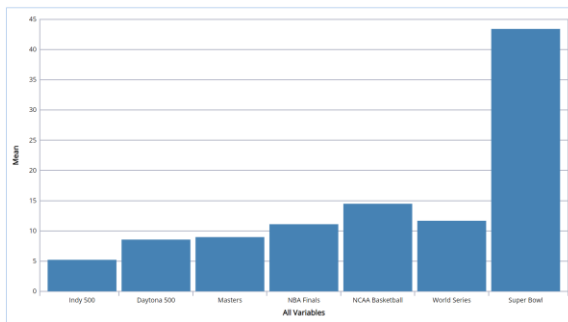
```
$.get (https://rason.net/api/optimize?profits[3]=60...
```

Using the Chart Wizard

The Chart Wizard, now offered in Frontline Systems' RASON Editor, allows users to easily create charts and visualize data. It offers a range of customization options for creating different types of charts, such as bar charts and variable plots. The Chart Wizard also provides features for saving and opening charts, as well as copying and printing chart images. With the Chart Wizard, users can quickly create and customize charts to analyze and present their data effectively.

• **Bar Chart**

The bar chart is one of the easiest and most effective plots to create and understand. The best application for this type of chart is comparing an individual statistic (i.e. mean, count, etc.) across a group of variables. The bar height represents the statistic while the bars represent the different variable groups. An example of a bar chart is shown below.



• **Box Whisker Plot**

A box plot graph summarizes a dataset and is often used in exploratory data analysis. This type of graph illustrates the shape of the distribution, its central value, and the range of the data. The plot consists of the most extreme values in the data set (maximum and minimum values), the lower and upper quartiles, and the median.

Box plots are very useful when large numbers of observations are involved or when two or more data sets are being compared. In addition, they are also helpful for indicating whether a distribution is skewed and whether there are any unusual observations (outliers) in the data set. The most important trait of the box plot is its failure to be strongly influenced by extreme values, or outliers.

A box plot includes the following statistical features.

Median: The median value in a dataset is the value that appears in the middle of a sorted dataset. If the dataset has an even number of values then the median is the average of the two middle values in the dataset.

Quartiles: Quartiles, by definition, separate a quarter of data points from the rest. This roughly means that the first quartile is the value **under** which 25% of the data lie and the third quartile is the value **over** which 25% of the data are found. (Note: This indicates that the second quartile is the median itself.)

First Quartile, Q1: Concluding from the definitions above, the first quartile is the median of the lower half of the data. If the number of data points is odd, the lower half includes the median.

Third Quartile, Q3: Third quartile is the median of the upper half of the data. If the number of data points is odd, the upper half of the data includes the median. See the following example.

Consider the following dataset --

52, 57, 60, 63, 71, 72, 73, 76, 98, 110, 120, 121, 124

The dataset has 13 values sorted in ascending order. The median is the middle value, (i.e. 7th value in this case.)

Median = 73

Q1 is the median of the first 7 values

25th Percentile = 63

Q3 is the median of the last 7 values.

75th Percentile = 110

The mean is the average of all the data values $((52 + 57 + 60 + 63 + 71 + 72 + 73 + 76 + 98 + 110 + 120 + 121 + 124) / 13)$.

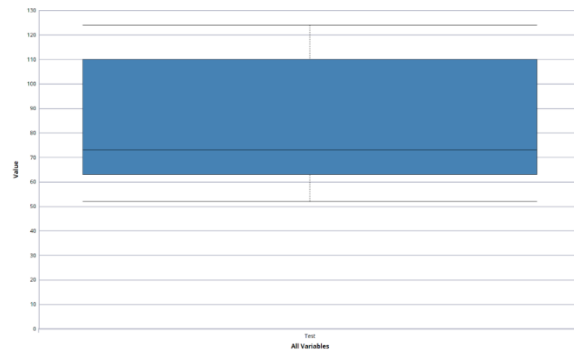
Mean = 84.38

Interquartile Range (IQR) = 47 The Interquartile range is a useful measure of the amount of variation in a set of data and is simply the 75th Percentile – 25th Percentile ($110 - 63 = 47$).

- The box extends from Q1 to Q3 and includes Q2.
- The median is denoted with a solid line through the box.
- The “whiskers” denote the extreme range of the data, not including any outliers.
 - Min range: 25th percentile - $1.5 * \text{IQR}$
 - Max range: 75th percentile + $1.5 * \text{IQR}$
- Outliers are denoted by a circle.

Min: 52

Max: 124



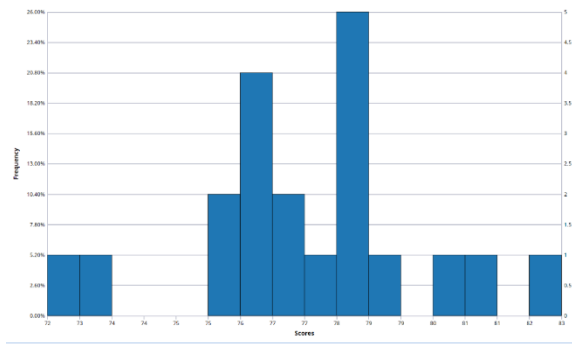
○ ***Histogram***

A Histogram, or a Frequency Histogram is a bar graph which depicts the range and scale of the observations on the x axis and the number of data points (or frequency) of the various intervals on the y axis. These types of graphs are popular among statisticians. Although these types of graphs do not show the exact values of the data points, they give a very good idea about the spread and shape of the data.

Consider the percentages below from a college final exam.

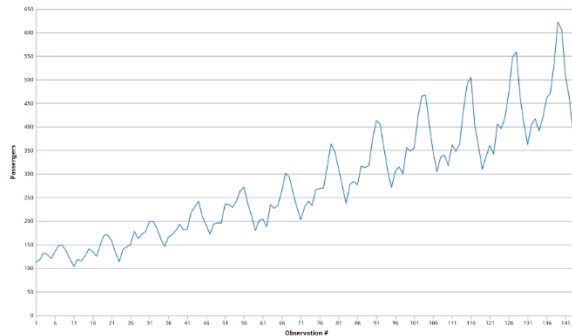
82.5, 78.3, 76.2, 81.2, 72.3, 73.2, 76.3, 77.3, 78.2, 78.5, 75.6, 79.2, 78.3, 80.2, 76.4, 77.9, 75.8, 76.5, 77.3, 78.2

One can immediately see the value of a histogram by taking a quick glance at the graph below. This plot quickly and efficiently illustrates the shape and size of the dataset above.



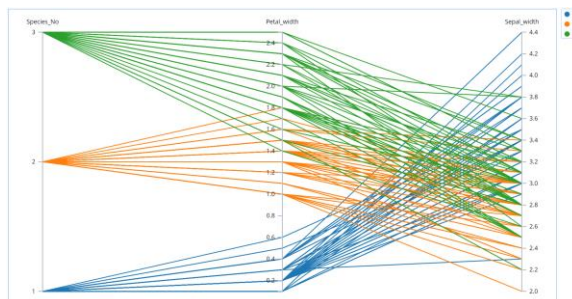
○ **Line Chart**

A line chart is best suited for time series datasets. In the example below, the line chart plots the number of airline passengers from January 1949 to December 1960. (The X – axis is the number of months starting with January 1949 as “1”.)



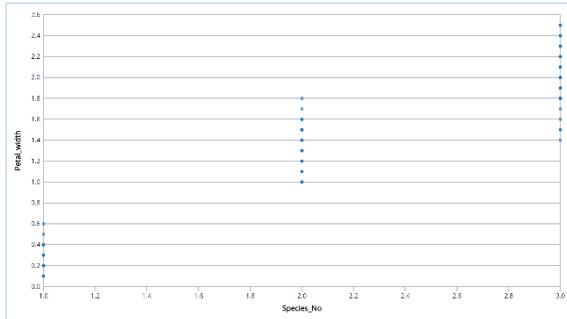
○ **Parallel Coordinates**

A Parallel Coordinates plot consists of N number of vertical axes where N is the number of variables selected to be included in the plot. A line is drawn connecting the observation’s values for each different variable (each different axis) creating a “multivariate profile”. These types of graphs can be useful for prediction and possible data binning. In addition, these graphs can expose clusters, outliers and variable “overlap”. An example of a Parallel Coordinates plot is shown below.



○ **Scatterplot**

One of the most common, effective and easy to create plots is the scatterplot. These graphs are used to compare the relationships between two variables and are useful in identifying clusters and variable “overlap”.



○ Scatterplot Matrix

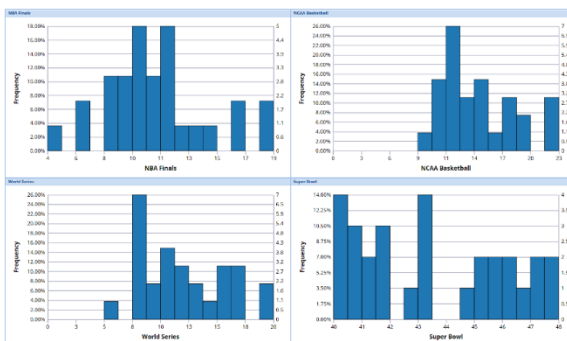
A Matrix plot combines several scatterplots into one panel enabling the user to see pairwise relationships between variables. Given a set of variables Var1, Var2, Var3, ..., Var N the matrix plot contains all the pairwise scatter plots of the variables on a single page in a matrix format. The names of the variables are on the diagonals. In other words, if there are k variables, there will be k rows and k columns in the matrix and the ith row and jth column will be the plot of Var_i versus Var_j .

The axes titles and the values of the variables appear at the edge of the respective row or column. The comparison of the variables and their interactions with one another can be studied easily and with a simple glance which is why matrix plots are becoming increasingly common in general purpose statistical software programs. An example is shown below.



○ Variable Plot

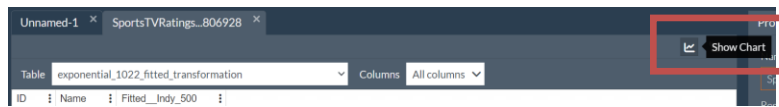
Analytic Solver Data Science's Variables graph simply plots each selected variable's distribution. See below for an example.



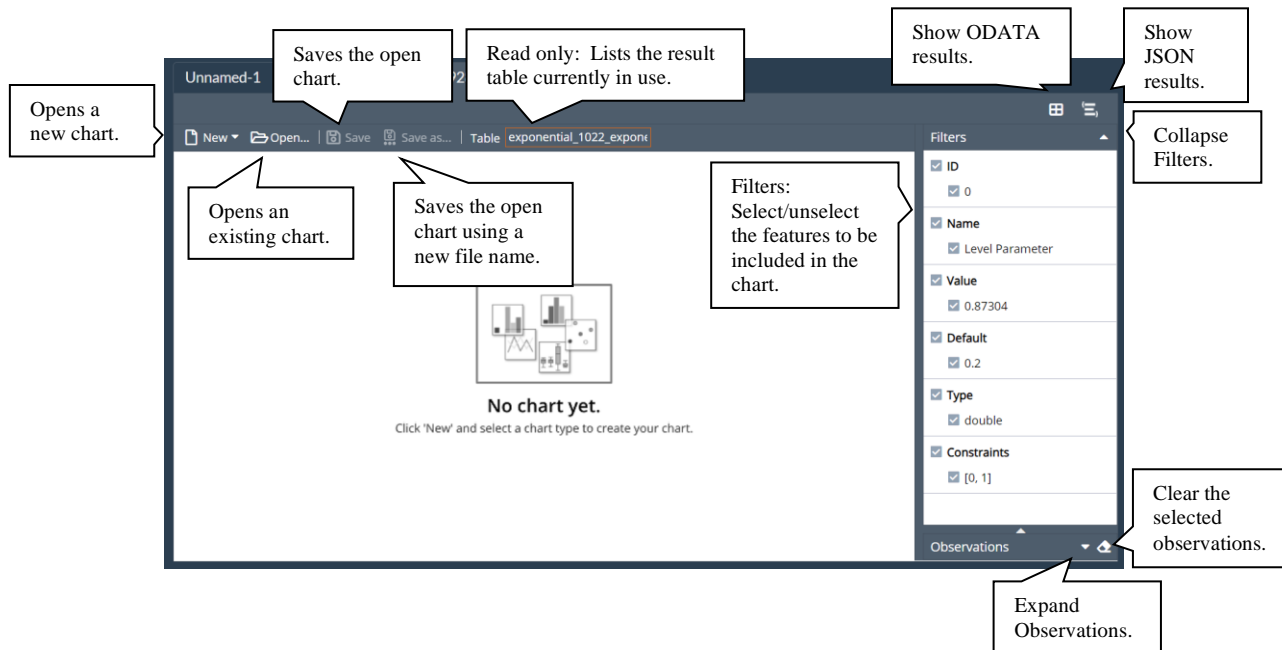
Example: Using the Chart Wizard to create a Line Chart

This example illustrates how easy it is to create a chart of your results using the chart wizard. Note: Currently, the Chart Wizard is only available for OData results. This example creates a line chart but any chart can be created using the same steps.

1. Once results are returned to the RASON Editor, select the drop down menu to view the desired result table in the chart.
2. Click the Show Chart icon on the upper right of the RASON Editor.

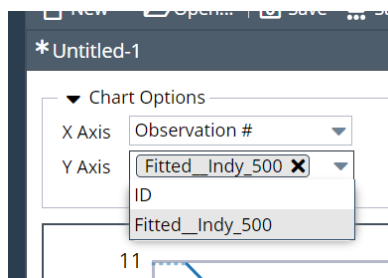


3. The Chart Wizard opens.

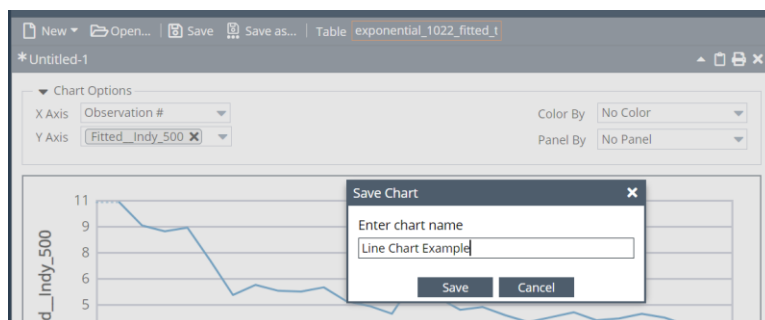


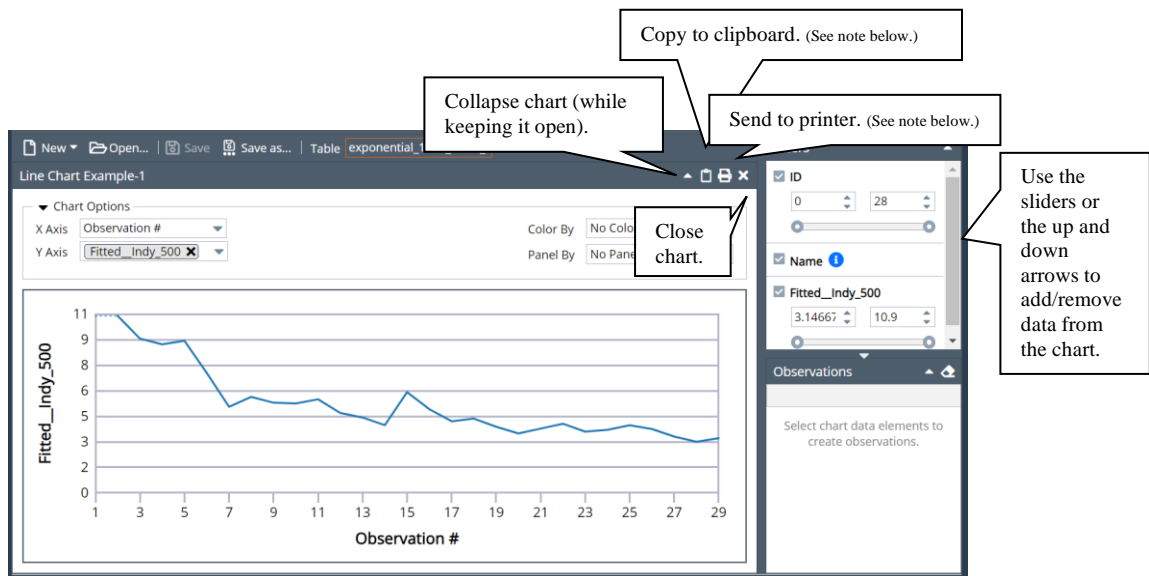
4. Select Fitted_Indy_500 for Y Axis.

Note: Since the variable ID is not a descriptive metric, click the X to remove from the chart.

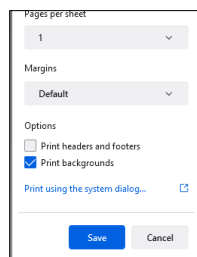


5. Click Save or Save as... to save the chart.



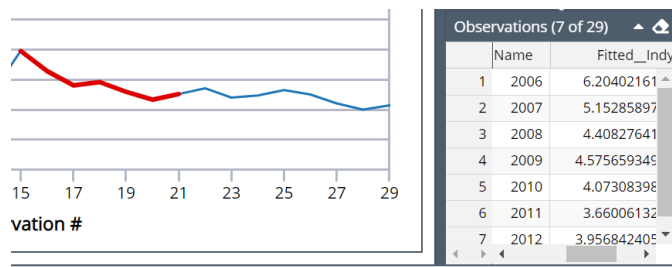


Note on printing: If using Mozilla's Firefox web browser, "Print backgrounds" must be selected in the Print Options pane, as shown in the screenshot below.



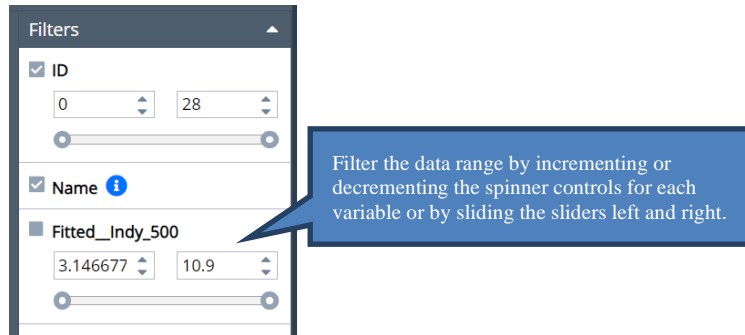
Note on copying to clipboard: Mozilla's Firefox web browser uses a default configuration that does not allow images to be copied to the clipboard. This configuration may be changed by navigating to *about:config*, within Firefox, and setting *dom.events.asyncClipboard.clipboardItem* to *True*.

- Highlight a portion of the chart to inspect those observations in the Observations panel.

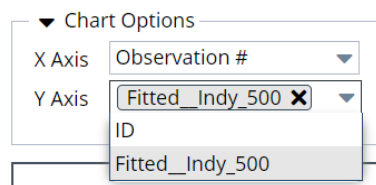


- To change the variables plotted on the X Axis, use the Filters pane. Uncheck the variable to remove from the chart. Select to include in the chart. The data range may be altered by clicking the up and down buttons on the spinner fields or by moving the sliders left (to decrement) or right (to increment).

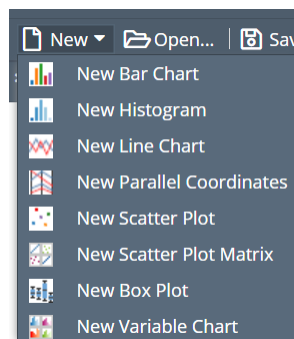
On charts that explicitly include the variable tag field, use these tags to add to or remove variables from the chart.



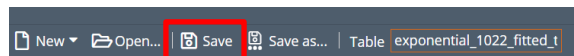
- To change the plotted metric on the Y Axis, click the down arrow next to Y Axis.



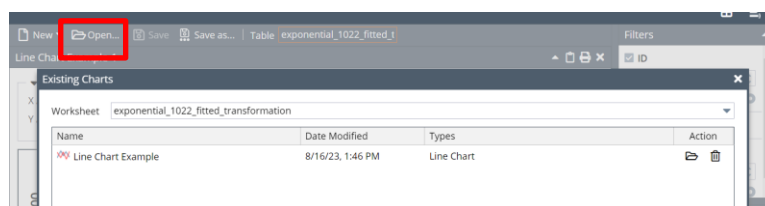
- To add a 2nd chart to the window, simply click the New Chart icon at the top, left. If multiple charts are open on the Chart window, use the Collapse panel arrow to collapse and expand charts as needed.





- To save the chart, click the Save icon on the top of the dialog.



- To open an existing chart, click the open file icon.





Click the  icon to open the chart and the  icon to delete the chart.

- Chart changes are tracked. Show confirm dialogs will be displayed when going back, closing or opening an existing chart when the current chart has been changed. The chart title will display an * when unsaved changes have been applied to the chart.

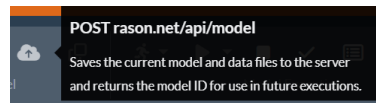
- Note: Categorical variables are variables with less than or equal to 12 unique values.

Example: Using the Chart Wizard to create a Bar Chart

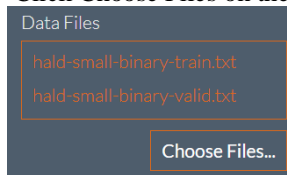
To open and run the following example, click  on the RASON Editor ribbon, then Data Science to open the full list of Data Science examples. Drill down to open the individual example files. All input data can be found by downloading and extracting datafiles.zip by clicking  on the RASON Editor ribbon.

To run,

1. Open the example. *DecisionTreeClassification* RASON example (Data Science – Classification – DecisionTree.json).
2. Post the example by clicking POST rason.net/api/model on the RASON Editor ribbon.

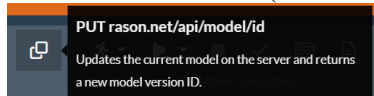


3. Click Choose Files on the properties pane to attach an input file(s) if required.

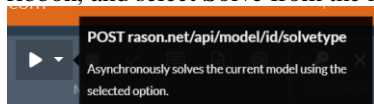


Input Files: hald-small-binary-train.txt and hald-small-binary-valid.txt

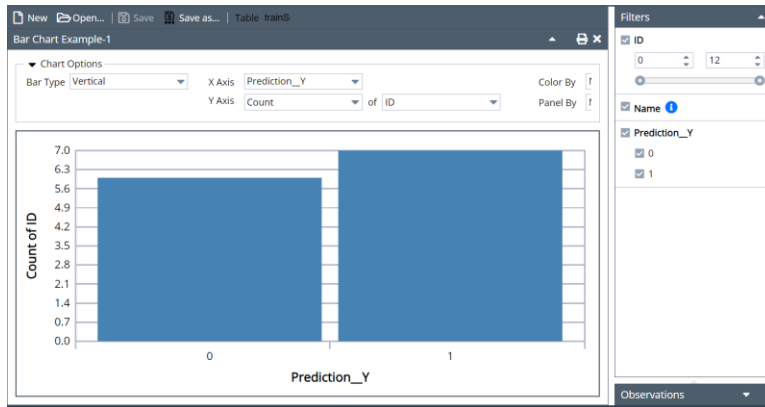
4. Attach the input files to the originally POSTed model by clicking PUT rason.net/api/model/id on the RASON Editor ribbon. (Recall that the PUT endpoint modifies the previously POSTed model.)



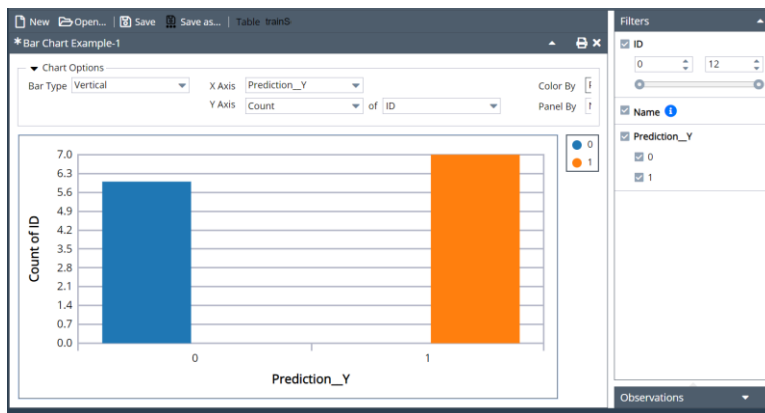
5. Run the data science example by clicking POST rason.net/api/model/id/solvetype on the RASON Editor ribbon, and select Solve from the menu.



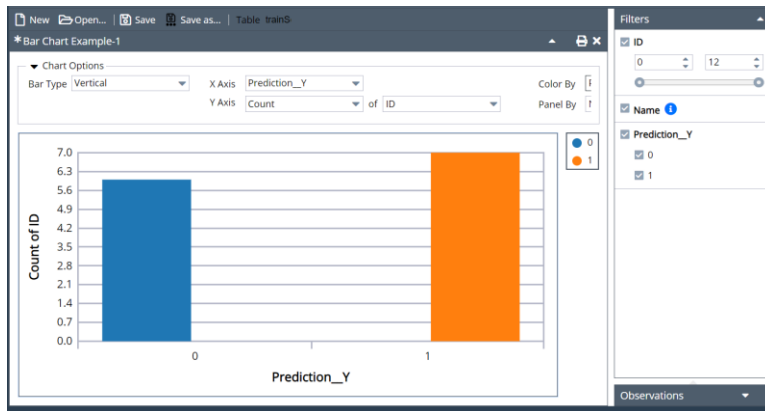
6. Select trainscore_prediction from the Table dropdown menu.
7. Click Show Chart.
8. Click New Bar Chart.
9. Select X Axis: Prediction_Y.
10. Select Y Axis: Count. This chart displays the number of records labeled as 0 and the number of records labeled as 1 for the training partition.



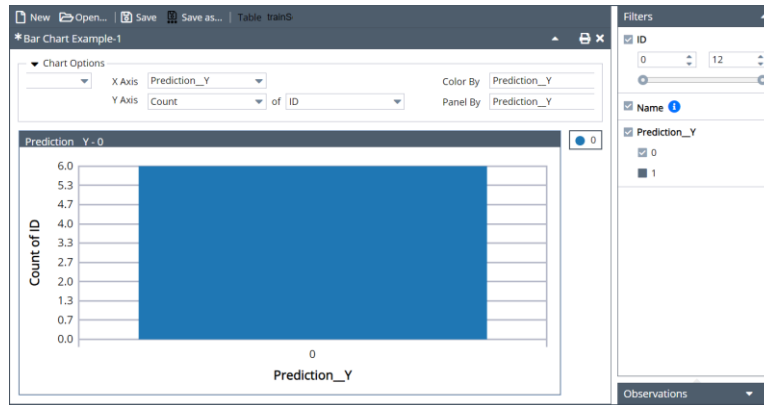
11. Select Color By Prediction_Y to change the bar color for the Predicted 1. Notice the * directly in front of the file name, "Bar Chart Example – 1". This * indicates that changes applied to the chart have not yet been saved.



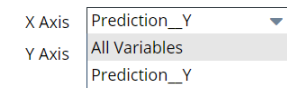
12. Select Panel By Prediction_Y to open two panels; each containing 1 bar.



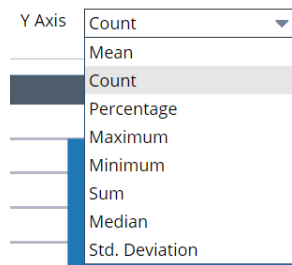
13. Uncheck "1" under Prediction_Y in the Filters pane to display the number of records labeled as 0 in the training partition.



- To select a different variable on the X-axis, click the down arrow next to X-Axis and select the desired variable from the menu.



- To change the statistic on the Y-axis, select the down arrow next to Y Axis and select the desired statistic.



- To change the orientation of the chart, click Bar Type and select Vertical or Horizontal.

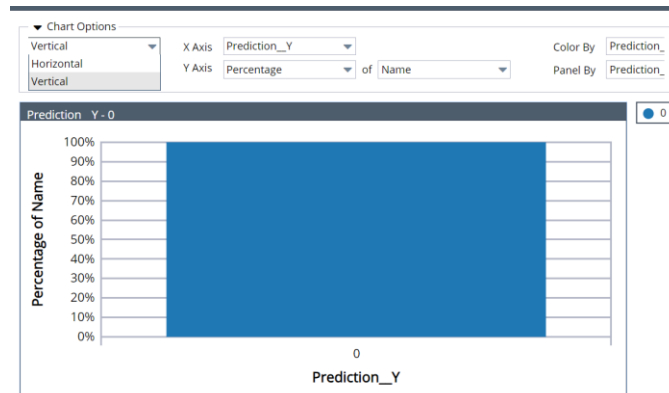
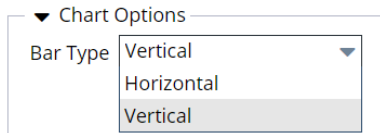


Chart Options

The following options may appear within the Chart Options pane in each chart.

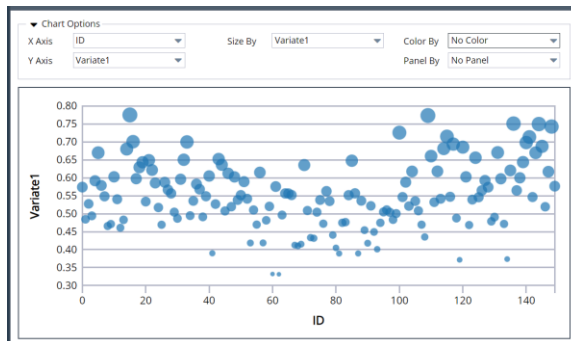
- The following option appears with bar charts.**

Bar Type: Select the Bar Type: Vertical or Horizontal. Select Vertical to draw bars from the Y axis. Select Horizontal to draw bars from the X axis.



- **The following option appears only with Scatterplot charts.**

Size By: Click the down arrow to select a variable to determine the size of the points on the chart. In the chart below, for example, the size of the dots are determined by the size of Variate1 for each record in the dataset. This option only appears in a scatterplot.



- **The following two options appear on Scatterplot Matrix charts.**

Layout: Choose Fit to Width to fit the chart to the width of the chart dialog area. Select Fit to Page to fit all scatter plots onto the chart dialog area.

Square Plots: Select this option to generate square scatter plots. Uncheck this option to generate rectangular scatter plots.



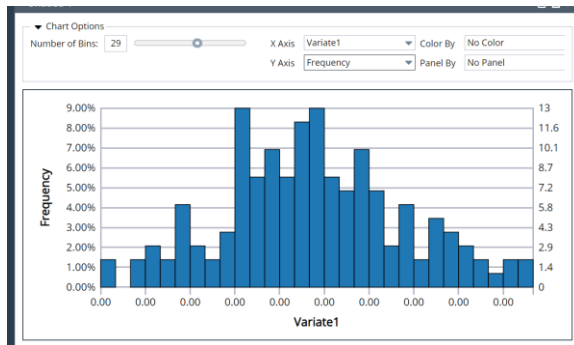
- **The following option appears on three chart types: parallel coordinates, scatterplot matrix and variable.**

Variable Tag Field: Click the down arrow to select a variable to include in the chart. Click the x in the upper right of the variable tab to remove the variable from the chart.



- The following option appears on histogram and variable charts.

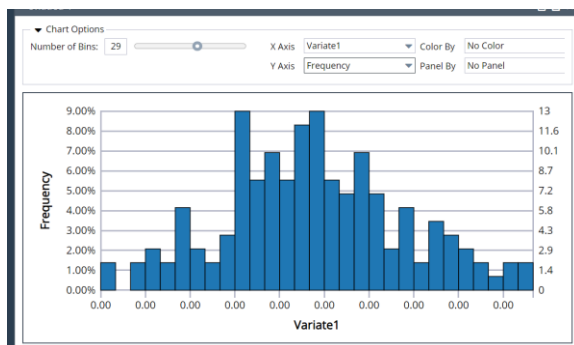
Number of Bins: Use the slider to increase or decrease the number of bins.



- The following options appear on the following chart types: bar, box plot, histogram, line, scatter plot and variable.

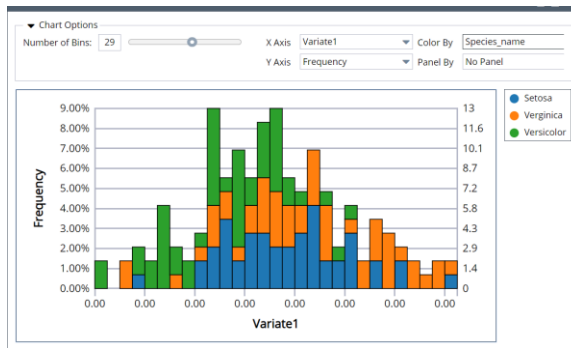
X Axis: Click the down arrow select the variable(s) to appear on the x-axis.

Y Axis: Click the down arrow to select the metric to appear on the y-axis.

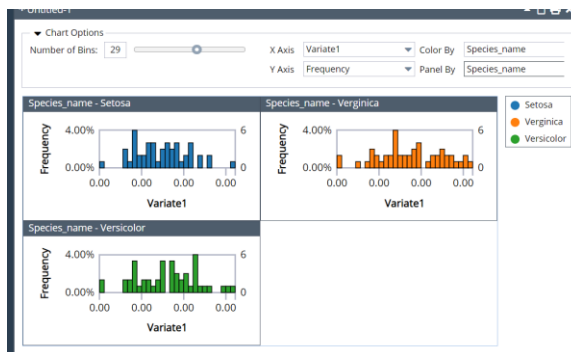


- The following options appear on all charts.

Color By: If the dataset contains categorical variables, click the down arrow next to Color By to display the data by category. A different color will be applied to each category, as shown below.



- **Panel By:** If the dataset contains categorical variables, click the down arrow next to Panel By to display the data by category in separate charts. Each category will be displayed in a separate chart, as shown below.



Filters Pane

Use the Filters pane to add and remove variables when a Variable Tag Field is not present. (For more information on the Variable Tag Field, see above.)

Uncheck the variable to remove from the chart. Select to include in the chart. The data range may be altered by clicking the up and down buttons on the spinner fields or by moving the sliders left (to decrement) or right (to increment).

Filters

☒ ID

0

149

☒ Name ⓘ

☒ Variate1

0.331404

0.775222

☒ Variate2

-0.930071

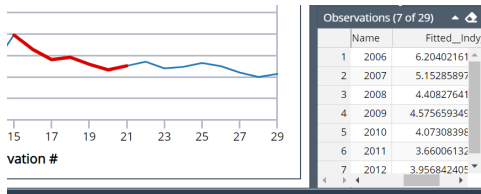
0.638775

☒ Species_name

☒ Setosa
☒ Verginica
☒ Versicolor

Observations Pane

Use the mouse to highlight a portion of the chart. Each observation included in the highlighted portion of the chart will appear on the Observations pane.



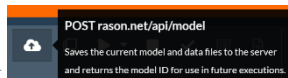
Solving a Data Science Model

Most data science models that perform a classification, transformation, forecast, etc. produce a "fitted model", which "fits" a model to the labeled or "known" data and then is used later to score new data. (For more information, see the Defining a Data Science Model chapter that appears later in this guide.) This section steps you through how to post and solve a data science model via the RASON Editor page and then how to use the fitted model to score new or already labeled data.

Open the data mining Classification model, NeuralNetwork.json by clicking RASON Examples on the Editor ribbon then clicking Data Science – Classification – NeuralNetwork.json. See the chapter Defining a Data Science Model for a discussion of this model and its results.

Since there are two files being imported in the datasources section, we must first click Choose Files on the Properties pane, browse to where the two files (hald-small-binary-train.txt and hald-small-binary-valid.txt) are located and click Open.

```
"datasources": {
  "myTrainSrc": {
    "type": "csv",
    "connection": "hald-small-binary-train.txt",
    "direction": "import"
  },
  "myValidSrc": {
    "type": "csv",
    "connection": "hald-small-binary-valid.txt",
    "direction": "import"
  },
}
```



After the files are uploaded, click [POST rason.net/api/model](https://rason.net/api/model) to call the REST API endpoint `POST rason.net/api/model` to post the model with both input files attached. In return you will receive a similar result.

Executing ajax call...: `POST https://rason.net/api/model`

```
{
  "ModelID": "2590+NeuralNetworkClassification+2020-01-21-00-25-31-005383",
  "ModelName": "NeuralNetworkClassification",
  "ModelDescr": "classification: neural network",
  "ModelFiles": ["hald-small-binary-train.txt", "hald-small-binary-valid.txt", "classification-nn-model.xml", "classification-nn-model.json"],
  "RuntimeToken": "",
  "ModelType": "Origin",
}
```

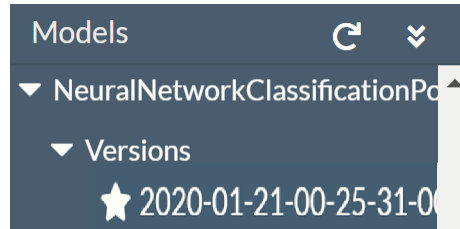
```

    "ModelKind": "RASON",
    "IsChampion": true,
    "ParentModelId": null,
    "QueryString": "",
    "ModelContainer": null
}

```

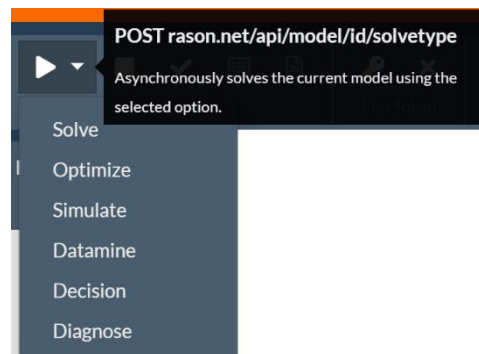
Calling this endpoint creates the Resource ID, 2590+NeuralNetworkClassification+2020-01-21-00-25-31-005383.

The newly posted version is displayed beneath Models, on the left. To mark this version as the "Champion", click "Mark Champion". Note the star that now appears next to the Version.



Solving the RASON DM Model

Click the down arrow next to the Play button and select either Solve or Datamine from the menu.



The following appears in the Result pane.

Executing asynchronous solve: POST https://rason.net/api/model

```

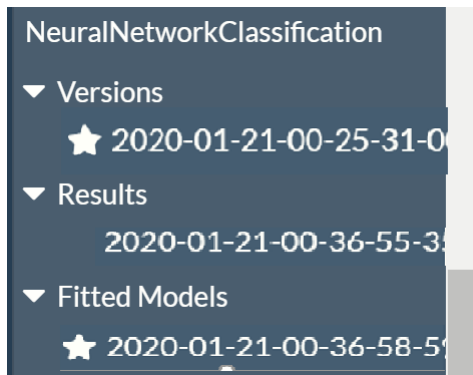
{
  "modelId": "2590+NeuralNetworkClassification+2020-01-21-00-36-55-353394",
  "modelName": "NeuralNetworkClassification",
  "modelDescr": "classification: neural network",
  "modelFiles": ["hald-small-binary-train.txt", "hald-small-binary-valid.txt", "classification-nn-model.xml", "classification-nn-model.json"],
  "runtimeToken": "",
  "modelType": "Instance",
  "modelKind": "RASON",
  "isChampion": true,
  "parentModelId": "2590+NeuralNetworkClassification+2020-01-21-00-25-31-005383",
  "queryString": "",
  "ModelContainer": null
}

```

Notice that the modelID is different from the parentModelID. This is because POST rason.net/api/model/{nameorid}/solve returned a new resource ID that identifies this unique model instance (model bound to data).

}

Notice that a new entry has appeared under Fitted Models under the NeuralNetworkClassification named model heading. A named fitted model (which is the only kind of fitted model) with the same modelName as the parent model is created when a RASON data science model containing action "fit" is run. (See line 69 in the example NeuralNetwork.json file.) The name given to the fitted model is the same as the setting given to modelName in the parent model. The fitted model receives a unique resource ID. If this RASON model or another one that fits the same {modelName} is run again, a new version of the fitted model is created, with a new unique resource ID. Mark this Fitted Model as the "champion" by highlighting the fitted model and selecting Mark Champion.



Note: The REST API endpoint `GET rason.net/api/model/nameorid/result/fitted-name` can be used to retrieve the text of the fittedModel. Continue reading to find out how to use this fitted model to score new data.

Now we will use this new model instance to check the status of the solve and obtain our results using `GET rason.net/api/model/{nameorid}/status` and `GET rason.net/api/model/{nameorid}/result`.

Checking the Status of a RASON Model



Click `GET rason.net/api/model/id/status` to check the status of the previously submitted model. This endpoint returns:

```
{"status": "Complete" }
```

Status: Complete means that the optimization has been completed.

Obtaining the Results of a RASON Model

Click the entry under NeuralNetworks - Results (+2020-01-21-00-36-55-353394) to enable the two result endpoints, `GET rason.net/api/model/{nameorid}/result` and `GET rason.net/api/model/{nameorid}/result/data`.

Click `GET rason.net/api/model/{nameorid}/result` to display the results in the Model Editor. Notice that the RASON response produced by the RASON Server includes the resource ID of the model instance that was run. This model resource ID is stored in the User Metrics table on the My Account page.

```
{
  "status": {
    "id": "2590+NeuralNetworkClassification+2020-01-21-00-36-55-353394",
    "code": 0,
    "codeText": "Success"
  },
  "results": {
    "nncModel.neuronWeights": [],
    "nncModel.numEpochsUsed": [],
    "nncModel.partitionCausedStopping": [],
    "nncModel.stoppingReason": [],
    "nncModel.trainingLog": [],
```

Status of solve

OData Results produced by
POST `rason.net/api/model/{nameorid}/solve`

```

"nncModel.trainingTime": [],
"trainScore.accuracy": [],
"trainScore.confusionMatrix": [],
"trainScore.f1": [],
"trainScore.posteriorProbability": [],
"trainScore.precision": [],
"trainScore.prediction": [],
"trainScore.recall": [],
"trainScore.sensitivity": [],
"trainScore.specificity": [],
"validScore.accuracy": [],
"validScore.confusionMatrix": [],
"validScore.f1": [],
"validScore.posteriorProbability": [],
"validScore.precision": [],
"validScore.prediction": [],
"validScore.recall": [],
"validScore.sensitivity": [],
"validScore.specificity": []
},
"nncModel": {
  "trainingLog": {
    "objectType": "dataFrame",
    "name": "Training Log",
    "order": "col",
    "rowNames": ["Epoch 1", "Epoch 2", "Epoch 3", "Epoch 4", "Epoch
5", "Epoch 6", "Epoch 7", "Epoch 8", "Epoch 9", "Epoch 10", "Epoch 11",
"Epoch 12", "Epoch 13", "Epoch 14", "Epoch 15", "Epoch 16", "Epoch 17",
"Epoch 18", "Epoch 19", "Epoch 20", "Epoch 21", "Epoch 22", "Epoch 23",
"Epoch 24", "Epoch 25", "Epoch 26", "Epoch 27", "Epoch 28", "Epoch 29",
"Epoch 30", "Epoch 31", "Epoch 32", "Epoch 33", "Epoch 34", "Epoch 35",
"Epoch 36", "Epoch 37", "Epoch 38", "Epoch 39", "Epoch 40", "Epoch 41",
"Epoch 42", "Epoch 43", "Epoch 44", "Epoch 45", "Epoch 46", "Epoch 47",
"Epoch 48", "Epoch 49", "Epoch 50", "Epoch 51", "Epoch 52", "Epoch 53",
"Epoch 54", "Epoch 55", "Epoch 56", "Epoch 57", "Epoch 58", "Epoch 59",
"Epoch 60", "Epoch 61", "Epoch 62", "Epoch 63", "Epoch 64", "Epoch 65",
"Epoch 66", "Epoch 67", "Epoch 68", "Epoch 69", "Epoch 70", "Epoch 71",
"Epoch 72", "Epoch 73", "Epoch 74", "Epoch 75", "Epoch 76", "Epoch 77",
"Epoch 78", "Epoch 79", "Epoch 80", "Epoch 81", "Epoch 82", "Epoch 83",
"Epoch 84", "Epoch 85", "Epoch 86", "Epoch 87", "Epoch 88", "Epoch 89",
"Epoch 90", "Epoch 91", "Epoch 92", "Epoch 93", "Epoch 94", "Epoch 95",
"Epoch 96", "Epoch 97", "Epoch 98", "Epoch 99", "Epoch 100"],
    "colNames": ["Training: Network Error (Cross Entropy)",
"Training: Data Error (Misclassification)", "Validation: Network Error
(Cross Entropy)", "Validation: Data Error (Misclassification)"],
    "colTypes": ["double", "double", "double", "double"],
    "indexCols": null,
    "data": [
      [0.66303106458569439, 0.61046150383331155, 0.56944006636282729,
0.52779162039300587, 0.48417504066980516, 0.44362260680789289,
0.40638475616914688, 0.37215396361664815, 0.3411592111360856,
0.31431372121567258, 0.29131909802834266, 0.27102389766093798,
0.25353556252847897, 0.23813822487069872, 0.22470097813635537,
0.21280936293863378, 0.20228703681028415, 0.19291095351064058,
0.1845022151930211, 0.1769364291783426, 0.17008566901396463,
0.16385202086310643, 0.15815736165219668, 0.1529302492035593,
0.14811227303762814, 0.14365570720572687, 0.13951863318194754,

```

Results from model fitting

0.13566514472547134, 0.13206511877311614, 0.12869251061694048,
 0.12552454521962747, 0.1225415301884923, 0.11972637230850892,
 0.11706403575912855, 0.11454124699820405, 0.11214629679558265,
 0.10986881018953637, 0.10769953657381337, 0.10563020106791242,
 0.1036533877314311, 0.10176242944089263, 0.099951309541604375,
 0.098214582156025687, 0.096547306563537702, 0.09494498943028741,
 0.093403533610519077, 0.091919194140355126, 0.090488540652579968,
 0.089108424598987587, 0.087775950263900768, 0.086488449223837263,
 0.085243457964565922, 0.084038698196981468, 0.082872059416664465,
 0.08174158339000176, 0.080645450347748282, 0.07958196668651682,
 0.078549553981234477, 0.077546739135733622, 0.076572145533259556,
 0.075624485073849626, 0.074702550998768227, 0.073805211411634764,
 0.072931403416608628, 0.072080127805153577, 0.071250444232266827,
 0.070441466830221316, 0.069652360213760117, 0.068882335836058628,
 0.068130648659690554, 0.067396594111116953, 0.066679505290812818,
 0.065978750414207185, 0.065293730461298868, 0.064623877015205694,
 0.063968650272027958, 0.063327537206275833, 0.062700049877748992,
 0.062085723867207059, 0.061484116829451943, 0.06089480715359024,
 0.060317392721253721, 0.059751489754459056, 0.059196731745587997,
 0.058652768462681568, 0.058119265023883551, 0.057595901035437588,
 0.057082369788157505, 0.056578377507750292, 0.056083642654785618,
 0.055597895270477579, 0.055120876364782037, 0.054652337343614694,
 0.05419203947226759, 0.053739753372352525, 0.053295258549820014,
 0.052858342951806907, 0.052428802550251424, 0.052006440950377902,
 0.051591069022308984],
 [0.15384615384615385, 0.076923076923076927, 0.076923076923076927,
 0.076923076923076927, 0.076923076923076927, 0.076923076923076927,
 0.076923076923076927, 0.076923076923076927, 0.076923076923076927,
 0.076923076923076927, 0.076923076923076927, 0.076923076923076927,
 0.076923076923076927, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0,
 0,
 0, 0],
 [0.66303106458569439, 0.61046150383331155, 0.56944006636282729,
 0.52779162039300587, 0.48417504066980516, 0.44362260680789289,
 0.40638475616914688, 0.37215396361664815, 0.3411592111360856,
 0.31431372121567258, 0.29131909802834266, 0.27102389766093798,
 0.25353556252847897, 0.23813822487069872, 0.22470097813635537,
 0.21280936293863378, 0.20228703681028415, 0.19291095351064058,
 0.1845022151930211, 0.1769364291783426, 0.17008566901396463,
 0.16385202086310643, 0.15815736165219668, 0.1529302492035593,
 0.14811227303762814, 0.14365570720572687, 0.13951863318194754,
 0.13566514472547134, 0.13206511877311614, 0.12869251061694048,
 0.12552454521962747, 0.1225415301884923, 0.11972637230850892,
 0.11706403575912855, 0.11454124699820405, 0.11214629679558265,
 0.10986881018953637, 0.10769953657381337, 0.10563020106791242,
 0.1036533877314311, 0.10176242944089263, 0.099951309541604375,
 0.098214582156025687, 0.096547306563537702, 0.09494498943028741,
 0.093403533610519077, 0.091919194140355126, 0.090488540652579968,
 0.089108424598987587, 0.087775950263900768, 0.086488449223837263,
 0.085243457964565922, 0.084038698196981468, 0.082872059416664465,
 0.08174158339000176, 0.080645450347748282, 0.07958196668651682,
 0.078549553981234477, 0.077546739135733622, 0.076572145533259556,
 0.075624485073849626, 0.074702550998768227, 0.073805211411634764,
 0.072931403416608628, 0.072080127805153577, 0.071250444232266827,
 0.070441466830221316, 0.069652360213760117, 0.068882335836058628,
 0.068130648659690554, 0.067396594111116953, 0.066679505290812818,

```

0.065978750414207185, 0.065293730461298868, 0.064623877015205694,
0.063968650272027958, 0.063327537206275833, 0.062700049877748992,
0.062085723867207059, 0.061484116829451943, 0.06089480715359024,
0.060317392721253721, 0.059751489754459056, 0.059196731745587997,
0.058652768462681568, 0.058119265023883551, 0.057595901035437588,
0.057082369788157505, 0.056578377507750292, 0.056083642654785618,
0.055597895270477579, 0.055120876364782037, 0.054652337343614694,
0.05419203947226759, 0.053739753372352525, 0.053295258549820014,
0.052858342951806907, 0.052428802550251424, 0.052006440950377902,
0.051591069022308984],
    [0.15384615384615385, 0.076923076923076927,
0.076923076923076927, 0.076923076923076927, 0.076923076923076927,
0.076923076923076927, 0.076923076923076927, 0.076923076923076927,
0.076923076923076927, 0.076923076923076927, 0.076923076923076927,
0.076923076923076927, 0.076923076923076927, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0],
    ],
    },
    "neuronWeights": {
        "objectType": "dataFrameVector",
        "name": "Neuron Weights",
        "data": {
            "Neuron Weights: Input Layer - Hidden Layer 1": {
                "objectType": "dataFrame",
                "name": "Neuron Weights: Input Layer - Hidden Layer 1",
                "order": "col",
                "rowNames": ["Neuron 1", "Neuron 2", "Neuron 3", "Neuron 4"],
                "colNames": ["X1", "X2", "X3", "X4", "Weights", "Bias"],
                "colTypes": ["double", "double", "double", "double", "double", "double"],
                "indexCols": null,
                "data": [
                    [-0.75069145845261487, -0.27887307054565263, -
0.79378896450938241, 0.093324509265532637],
                    [-0.43066792161670547, -0.5410528516228591, -
0.1673473671984495, 0.36763405948607503],
                    [-0.66269427879746901, 0.44816597640320865,
0.022985461449100159, -1.1276745743539944],
                    [0.01091317921324568, 0.5809695393525508,
0.023724515541765721, -0.23574636017890654],
                    [-0.11534712105085403, 0.51441944511523519, -
0.19651109483154025, -0.49304953437134286],
                    [-0.0056227144792817983, 0.00024935739304918469, -
0.0009730192415791769, -0.00075515790996579716]
                ]
            },
            "Neuron Weights: Hidden Layer 1 - Output Layer": {
                "objectType": "dataFrame",
                "name": "Neuron Weights: Hidden Layer 1 - Output Layer",
                "order": "col",
                "rowNames": ["0", "1"],
                "colNames": ["Neuron 1", "Neuron 2", "Neuron 3", "Neuron 4", "Bias"],
            }
        }
    }
}

```



```

    "colTypes": ["double", "double", "double", "double", "double"],
    "indexCols": null,
    "data": [
      [-0.77478453633089717, -0.15371632782460409],
      [1.1850172724323604, -1.8065785521236266],
      [-0.65417688123264595, -0.64439128694926806],
      [-1.7117601902255974, 2.5736303856604104],
      [0.48784434452201758, -0.48784434452201725]
    ]
  }
},
"numEpochsUsed": 100,
"trainingTime": 0.5813059999999999,
"stoppingReason": "Maximum number of epochs reached",
"partitionCausedStopping": null
},
"trainScore": {
  "prediction": {
    "objectType": "dataFrame",
    "name": "Prediction",
    "order": "col",
    "rowNames": ["Record 1", "Record 2", "Record 3", "Record 4",
"Record 5", "Record 6", "Record 7", "Record 8", "Record 9", "Record 10",
"Record 11", "Record 12", "Record 13"],
    "colNames": ["Prediction: Y"],
    "colTypes": ["wstring"],
    "indexCols": null,
    "data": [
      ["0", "0", "1", "0", "1", "1", "1", "0", "0", "1", "0", "1", "1"]
    ]
  }
},
"posteriorProbability": {
  "objectType": "dataFrame",
  "name": "Posterior Probability",
  "order": "col",
  "rowNames": ["Record 1", "Record 2", "Record 3", "Record 4",
"Record 5", "Record 6", "Record 7", "Record 8", "Record 9", "Record 10",
"Record 11", "Record 12", "Record 13"],
  "colNames": ["0", "1"],
  "colTypes": ["double", "double"],
  "indexCols": null,
  "data": [
    [0.98142563041514053, 0.98142506620239767, 0.035388177536841307,
0.98141295352694591, 0.036583192233183064, 0.035889568070065693,
0.035968315202947687, 0.98142532181709696, 0.72414124228947319,
0.035283978729137452, 0.98142313253302671, 0.035240926303000457,
0.035240035931403735],
    [0.018574369584859543, 0.018574933797602357, 0.9646118224631588,
0.018587046473054213, 0.96341680776681693, 0.96411043192993429,
0.96403168479705237, 0.018574678182903086, 0.27585875771052676,
0.96471602127086253, 0.018576867466973224, 0.96475907369699965,
0.96475996406859632]
  ]
},
"confusionMatrix": {
  "objectType": "dataFrame",

```

Results for training dataset

Posterior Probabilities

Confusion matrix

```

    "name": "Confusion Matrix",
    "order": "col",
    "rowNames": ["0", "1"],
    "colNames": ["0", "1"],
    "colTypes": ["double", "double"],
    "indexCols": null,
    "data": [
      [6, 0],
      [0, 7]
    ]
  },
  "accuracy": 100,
  "specificity": 1,
  "sensitivity": 1,
  "recall": 1,
  "precision": 1,
  "f1": 1
},
"validScore": {
  "prediction": {
    "objectType": "dataFrame",
    "name": "Prediction",
    "order": "col",
    "rowNames": ["Record 1", "Record 2", "Record 3", "Record 4",
"Record 5", "Record 6", "Record 7", "Record 8", "Record 9", "Record 10",
"Record 11", "Record 12", "Record 13"],
    "colNames": ["Prediction: Y"],
    "colTypes": ["wstring"],
    "indexCols": null,
    "data": [
      ["0", "0", "1", "0", "1", "1", "1", "0", "0", "1", "0", "1",
"1"]
    ]
  }
},
  "posteriorProbability": {
    "objectType": "dataFrame",
    "name": "Posterior Probability",
    "order": "col",
    "rowNames": ["Record 1", "Record 2", "Record 3", "Record 4",
"Record 5", "Record 6", "Record 7", "Record 8", "Record 9", "Record 10",
"Record 11", "Record 12", "Record 13"],
    "colNames": ["0", "1"],
    "colTypes": ["double", "double"],
    "indexCols": null,
    "data": [
      [0.98142563041514053, 0.98142506620239767,
0.035388177536841307, 0.98141295352694591, 0.036583192233183064,
0.035889568070065693, 0.035968315202947687, 0.98142532181709696,
0.72414124228947319, 0.035283978729137452, 0.98142313253302671,
0.035240926303000457, 0.035240035931403735],
      [0.018574369584859543, 0.018574933797602357,
0.9646118224631588, 0.018587046473054213, 0.96341680776681693,
0.96411043192993429, 0.96403168479705237, 0.018574678182903086,
0.27585875771052676, 0.96471602127086253, 0.018576867466973224,
0.96475907369699965, 0.96475996406859632]
    ]
  }
},

```

Results for validation
dataset

Posterior Probabilities

```

"confusionMatrix": {
  "objectType": "dataFrame",
  "name": "Confusion Matrix",
  "order": "col",
  "rowNames": ["0", "1"],
  "colNames": ["0", "1"],
  "colTypes": ["double", "double"],
  "indexCols": null,
  "data": [
    [6, 0],
    [0, 7]
  ]
},
"accuracy": 100,
"specificity": 1,
"sensitivity": 1,
"recall": 1,
"precision": 1,
"f1": 1
}
}

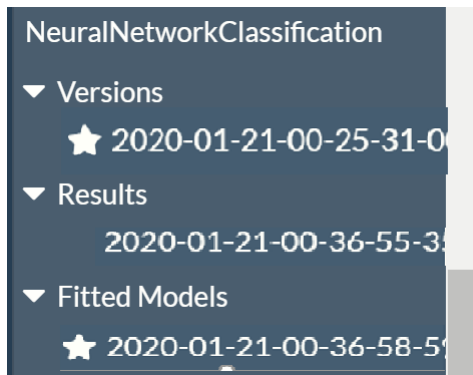
```

Confusion Matrix

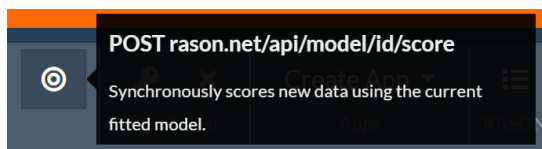
Note that the endpoint `POST rason.net/api/model/{nameorid}/solve` automatically creates the OData endpoints under Results for easy retrieval, querying, etc.

Scoring New Data

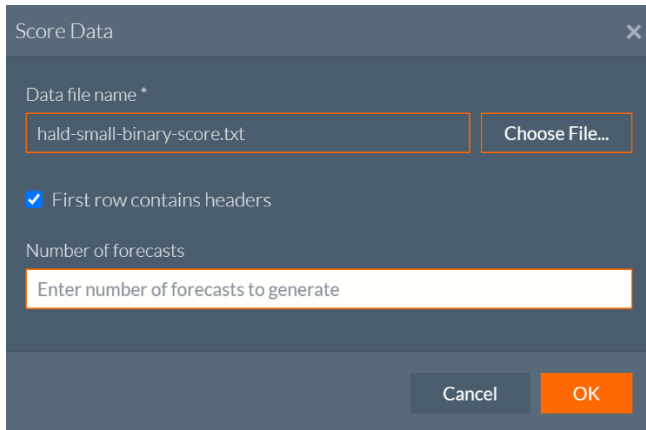
Now that the model has been solved and the fitted model created, new data may be scored using this fitted model. Double click the fitted model in the Model task pane to open in the Editor window.



Then click the score icon to open the Score Data dialog.



Enter the name of the data file to be scored, in this example `hald-small-binary-score.txt`. Then click OK to score the new data. The Number of Forecasts field is used when scoring using a regression model.



The image shows a 'Score Data' dialog box with a dark blue header and a light blue body. The header has a close button (X) in the top right corner. The body contains the following elements: a label 'Data file name *' above a text input field containing 'hald-small-binary-score.txt' and a 'Choose File...' button; a checked checkbox labeled 'First row contains headers'; a label 'Number of forecasts' above a text input field containing the placeholder text 'Enter number of forecasts to generate'; and at the bottom, two buttons: 'Cancel' and 'OK'.

Scoring results may be viewed in the Output window.

Score result using 'NeuralNetworkClassification' fitted model with 'hald-small-binary-score.txt' data file.

```
[ "0", "0", "1", "0", "1", "1", "1", "0", "0", "1", "0", "1", "1"]
```

Solving a RASON Model using a Runtime Token

As discussed in the previous chapter, RASON Subscriptions, runtime tokens can be used to execute (only) a model that has already been posted to the server. When using a runtime token, you will not be able to edit or delete the model. See the "Runtime Token" section within the "RASON Subscription" chapter for more information on how to create and manage runtime tokens.

Given that a resource ID has already been posted to the RASON Server, the process for executing a model with a runtime token continues as:

- Use GET `rason.net/api/model/{id}/optimize` (or `simulate`, `datamine`, `decision` or `solve`) to solve the model.
- Use GET `rason.net/api/model/{id}/status` to check the progress.
- Use GET `rason.net/api/model/{id}/result` or GET `rason.net/api/model/{id}/result/data` to obtain the model results.

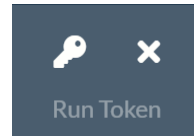
Note that you must pass the **resource ID**, rather than the model name, when using a runtime token.



The bearer of a runtime token may also call POST `rason.net/api/model/{id}/stop` to stop the execution of a model in progress. The complete list of API's allowed by a runtime token are listed below. Calls to any other API will result in a 401 Unauthorized error.

- `/api/model/{id}/solve`
- `/api/model/{id}/optimize`
- `/api/model/{id}/decision`
- `/api/model/{id}/simulate`
- `/api/model/{id}/datamine`
- `/api/model/{id}/status`
- `/api/model/{id}/result`
- `/api/model/{id}/result/data`
- `/api/model/{id}/stop`

Note that a runtime token is valid *only* for the model for which it was generated and may *not* be used with Quick Solve REST API endpoints. The token does not allow access, even execution, to other models.

Click My Account to manage your Runtime tokens. (See the RASON Subscription chapter for more information on this page.) You can also create and obtain Runtime tokens on the Editor page by clicking the two icons found within the Run Token section of the ribbon.



Click  to call the REST API endpoint, GET `rason.net/api/model/{nameorid}/runtoken`, to create and obtain a runtime token. Click  to call the REST API endpoint DELETE `rason.net/api/model/{nameorid}/runtoken` to delete the runtime token. See the following section for more information on both of these endpoints.

Helpful Editor Window Shortcuts

See below for a list of helpful shortcuts that can be used in the Editor window when entering a RASON model.

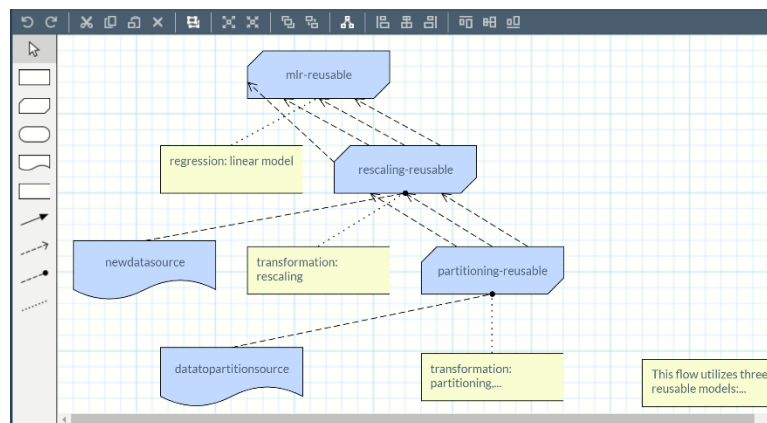
showSettingsMenu : Ctrl-, goToNextError : Alt-E goToPreviousError : Alt-Shift-E selectall : Ctrl-A gotoline : Ctrl-L fold : Alt-L Ctrl-F1 unfold : Alt-Shift-L Ctrl-Shift-F1 toggleFoldWidget : F2 toggleParentFoldWidget : Alt-F2 foldOther : Alt-0 unfoldall : Alt-Shift-0 findnext : Ctrl-K iSearchAndGo : Ctrl-K findprevious : Ctrl-Shift-K iSearchBackwardsAndGo : Ctrl-Shift-K selectOrFindNext : Alt-K selectOrFindPrevious : Alt-Shift-K find : Ctrl-F iSearch : Ctrl-F overwrite : Insert selecttostart : Ctrl-Shift-Home gotostart : Ctrl-Home selectup : Shift-Up golineup : Up selecttoend : Ctrl-Shift-End gotoend : Ctrl-End selectdown : Shift-Down golinedown : Down selectwordleft : Ctrl-Shift-Left gotowordleft : Ctrl-Left selecttolinestart : Alt-Shift-Left gotolinestart : Alt-Left Home selectleft : Shift-Left gotoleft : Left selectwordright : Ctrl-Shift-Right gotowordright : Ctrl-Right selecttolineend : Alt-Shift-Right gotolineend : Alt-Right End	selectright : Shift-Right gotoright : Right selectpagedown : Shift-Pagedown gotopagedown : Pagedown selectpageup : Shift-Pageup gotopageup : Pageup scrollup : Ctrl-Up scrolldown : Ctrl-Down selectlinestart : Shift-Home selectlineend : Shift-End toggleRecording : Ctrl-Alt-E replaymacro : Ctrl-Shift-E jumpmatching : Ctrl- Ctrl-P selecttomatching : Ctrl-Shift- Ctrl-Shift-P expandToMatching : Ctrl-Shift-M removeline : Ctrl-D duplicateSelection : Ctrl-Shift-D sortlines : Ctrl-Alt-S togglecomment : Ctrl-/ toggleBlockComment : Ctrl-Shift-/ modifyNumberUp : Ctrl-Shift-Up modifyNumberDown : Ctrl-Shift-Down replace : Ctrl-H undo : Ctrl-Z redo : Ctrl-Shift-Z Ctrl-Y copylinesup : Alt-Shift-Up movelinesup : Alt-Up copylinesdown : Alt-Shift-Down movelinesdown : Alt-Down del : Delete backspace : Shift-Backspace Backspace cut_or_delete : Shift-Delete removetolinestart : Alt-Backspace removetolineend : Alt-Delete removetolinestarthard : Ctrl-Shift-Backspace removetolineendhard : Ctrl-Shift-Delete removewordleft : Ctrl-Backspace removewordright : Ctrl-Delete	outdent : Shift-Tab indent : Tab expandSnippet : Tab blockoutdent : Ctrl-[blockindent : Ctrl-] transposeletters : Alt-Shift-X toupper : Ctrl-U tolower : Ctrl-Shift-U expandtoline : Ctrl-Shift-L openCommandPalette : F1 openInlineEditor : F3 addCursorAbove : Ctrl-Alt-Up addCursorBelow : Ctrl-Alt-Down addCursorAboveSkipCurrent : Ctrl-Alt-Shift-Up addCursorBelowSkipCurrent : Ctrl-Alt-Shift-Down selectMoreBefore : Ctrl-Alt-Left selectMoreAfter : Ctrl-Alt-Right selectNextBefore : Ctrl-Alt-Shift-Left selectNextAfter : Ctrl-Alt-Shift-Right toggleSplitSelectionIntoLines : Ctrl-Alt-L alignCursors : Ctrl-Alt-A findAll : Ctrl-Alt-K snippet : Alt-C focusCommandLine : Shift-Esc Ctrl-` nextFile : Ctrl-Tab previousFile : Ctrl-Shift-Tab execute : Ctrl-Return increaseFontSize : Ctrl-= Ctrl+= decreaseFontSize : Ctrl-- Ctrl-_ resetFontSize : Ctrl-0 Ctrl-Numpad0 save : Ctrl-S load : Ctrl-O startAutocomplete : Ctrl-Space Ctrl-Shift-Space Alt-Space beautify : Ctrl-Shift-B showKeyboardShortcuts : Ctrl-Alt-H
---	--	---

Creating and Running a Decision Flow

Users can easily create **multi-stage decision flows**, with business rules, optimization, simulation and machine learning models using the drag and drop editor on www.RASON.com. Decision flow stages can execute the **full range of predictive and prescriptive analytics**: DMN-compatible decision tables, Excel calculations, SQL operations, Monte Carlo simulations, mathematical optimizations, or machine learning training or prediction steps. Multi-dimensional data is automatically passed between stages in a standard, general "indexed data frame" form that maintains a dimensional information across statistic and machine learning transformations. Users can choose to run only the **stages that need updating**, easily determine the outcome of each stage and obtain results in JSON or OData form, at each stage or at the final stage. Each stage can be **written inline or can invoke a reusable model** via its interface. Models can be written in **RASON or Excel** with Analytic Solver and may be used in multiple decision flows. Using **automatic scheduling**, users can schedule a decision flow to run at fixed intervals or specify how recently updated they want each stage to be. RASON will automatically determine when to run each stage.

What is a Decision Flow?

RASON Services enables users to define multiple "stages" or nodes in a **single RASON file**, where a stage can perform an SQL operation, apply a data transformation, train a machine learning model, apply it to score new data, run a simulation, solve a mathematical optimization problem or evaluate one or more linked decision tables. You can think of these stages as events. In the example decision flow below three events or stages are linked together: first, in the partitioning node, an Excel data source is partitioned into multiple data sets. In the middle step, the three data partitions are rescaled along with a new data source. In the final step, multiple linear regression is applied to all three partitions and the new data source. The result of several stages linked together within a single RASON file is referred to as a *RASON Decision Flow*.



RASON Decision Flow Editor

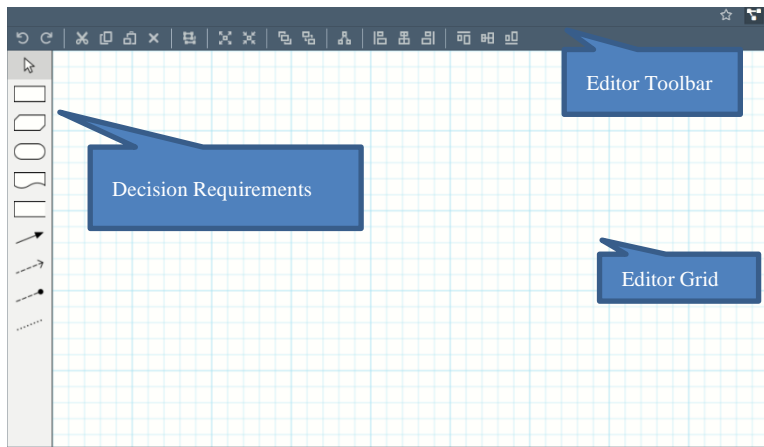
Along with decision flows, RASON Decision Services includes an exciting feature, the RASON Decision Flow Editor which is a graphical editor allowing users to quickly and easily define and structure "decision workflows" which may depend on multiple RASON models. This GUI editor resembles editors in other products implementing the DMN 1.2 standard, but is more general in nature in that it encompasses DRDs

(Decision Requirements Diagrams), decision tables editable as "boxed expressions" with FEEL syntax and Business Knowledge Models (BKMs) that directly define the logic of datamining / ML models, optimization models and simulation models.

To toggle between the RASON model editor and the decision flow editor, click the right most icon above the RASON model editor.

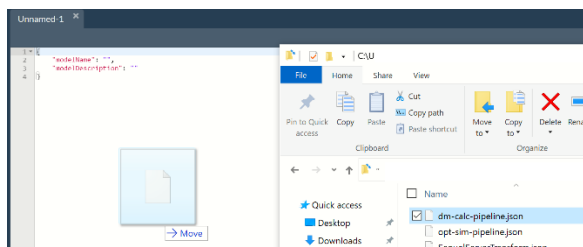


RASON Model Editor



To create a Decision Flow, drag Decision Requirements to the grid and connect the elements using an appropriate connection.

You can also open an existing decision flow by dragging the file to the grid.



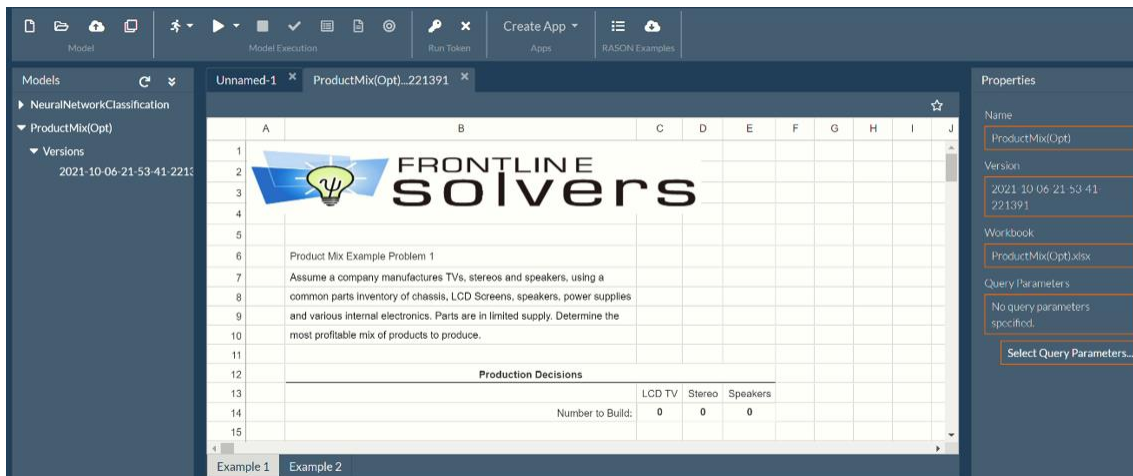
Note: When dragging a file into the Editor window, a copy of the model will be posted to the RASON server, *even if the original file is saved on a user's OneDrive account*. This new model will not overwrite or update the file saved on the OneDrive account.

The Decision Requirements icons are found down the left of the Editor. A decision flow may either be made up of "inline" models, where the model is written within the decision flow stage script, or a stage may invoke a "reusable model", which is a slightly modified standalone model. (A standalone RASON model is one that can be successfully solved "on its own", meaning that it contains all required elements of a RASON model: data, decision variables, constraints, uncertain variables, uncertain functions, etc.) A decision flow may contain both types of stages.

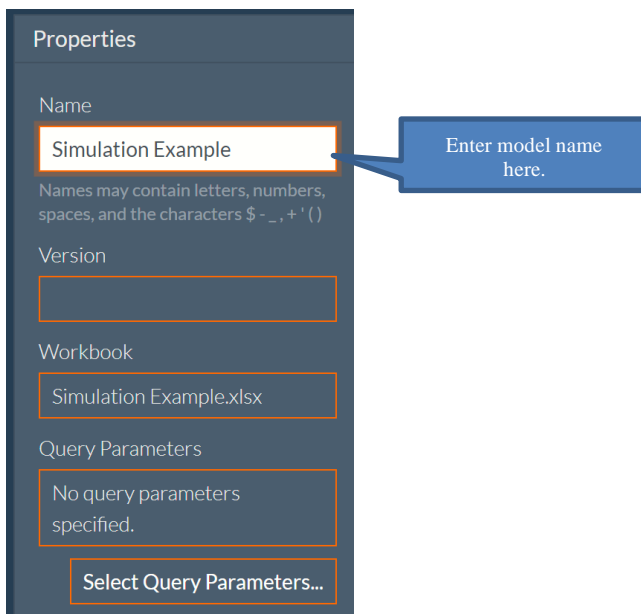
In the RASON Decision Flow Editor, each node represents a stage in a RASON Decision Flow. Arrows represent dependencies between stages and invocations represent "interface files" that define binding of data from stage to stage. Opening a node displays the RASON text editor or if the node invokes an Excel workbook,

a *non-editable* representative image of the Excel workbook. (Users can edit the Excel workbook through Excel.)

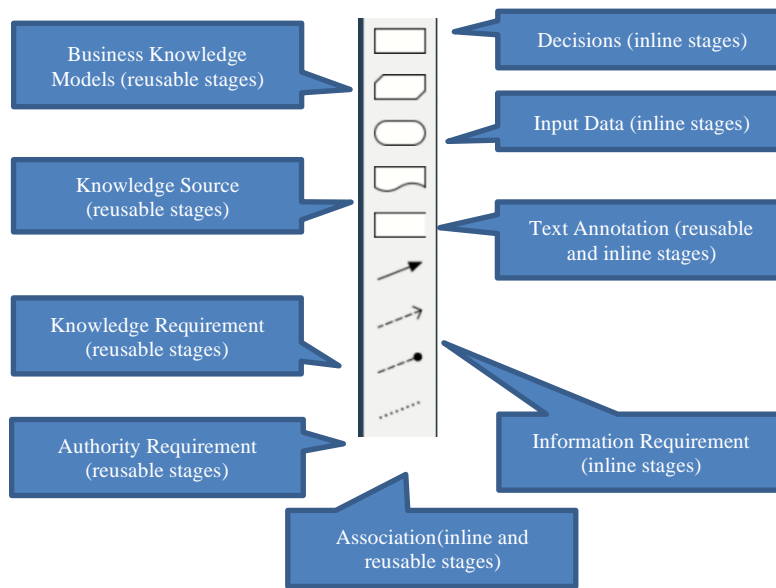
Example of Node invoking an Excel workbook



When posting an Excel model to the RASON server, use the Name field on the Properties pane to enter a model name.


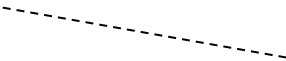


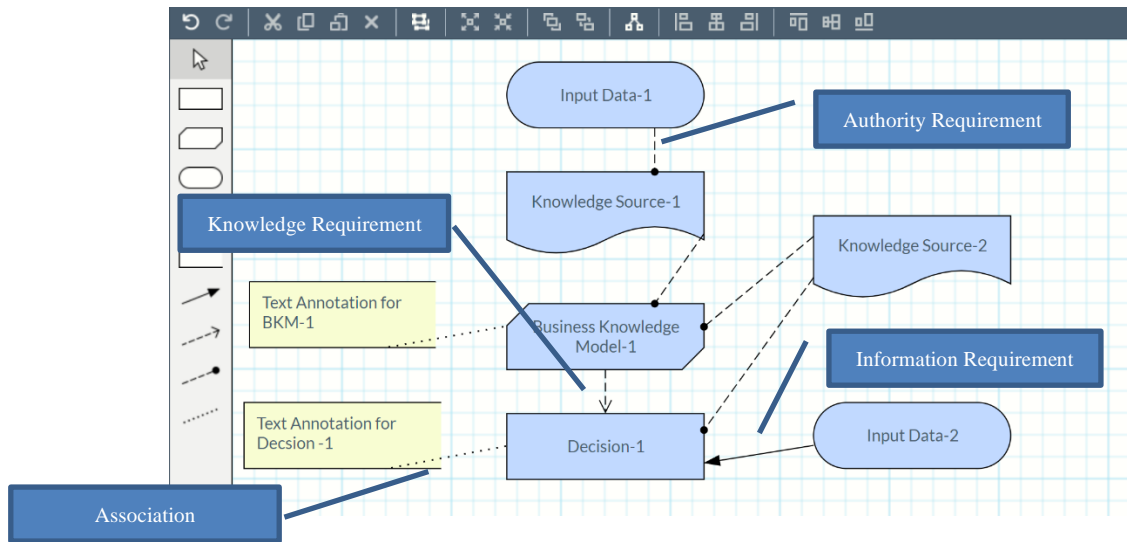
Other operations in RASON Data Science, such as SQL queries, rescaling or binning, missing value imputation and partitioning of datasets, can also be represented as Business Knowledge Models if a reusable model or Decisions if the decision flow is "inline".



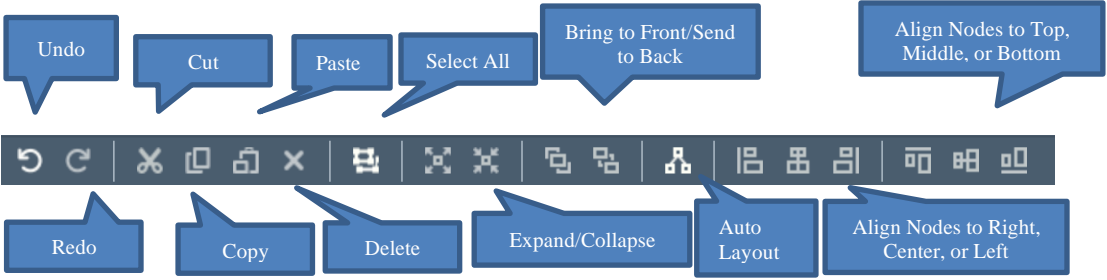
The `DMN 1.2 standard` defines the following elements of a Decision Requirements Graph (Section 6.1). Frontline's Graphical Editor implements these elements in the following ways.

A Decision element represents the act of determining an output from a number of inputs, using decision logic. In Frontline's Editor, inline stages are represented as Decisions.	
A Business Knowledge Model element represents a reusable model.	
An Input Data element denotes information used as an input by one or more Decisions	
A Knowledge Source element denotes input for a Business Knowledge Model or Decision.	
An Information Requirement denotes Input Data or Decision output being used as input to a Decision, i.e. use an Information Requirement to link a Decision with a subsequent Decision or Input Data to a Decision.	
A Knowledge Requirement represents the invocation of a Business Knowledge Model. Use this connection to link a Business Knowledge Model with a subsequent Business Knowledge Model or a Business Knowledge Model with a subsequent Decision.	
An Authority Requirement denotes the dependence of a Decision Requirements Graph element on another Decision Requirements Graph element that acts as a source of guidance or knowledge. Use an Authority Requirement to link a Knowledge Source with a subsequent Business Knowledge Model or Decision, Input Data to a subsequent Knowledge Source or a Decision to a subsequent Knowledge Source.	

A Text Annotation is user-entered text for comment or explanation.	
An Association is a dotted connector used to link a Text Annotation to a Decision Requirements Graph Element.	

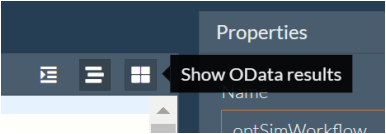


Use the toolbar at the top of the editor to undo/redo changes, copy, paste, delete and align nodes.



Getting Results

In order to solve a decision flow, a user must call the RASON API endpoint, POST `rason.net/api/model/{nameorid}/solve`. This endpoint creates a new model instance, enqueues the model to be solved, returns a Location header with a new instance ID and creates an OData endpoint to allow for easy querying of the results. (For a complete description of this endpoint, see the chapter, Using the REST API.) Because of this endpoint, decision flow results can be easily displayed on the Editor tab by clicking the "Show OData results" icon in the top right corner of the Editor.



Click the vertical dots too format the column data.

Click the down arrow to select the desired result to display in the window.

Click the down arrow to select the columns displayed.

Click to open the Chart Wizard.

Click to view the results in JSON.

ID	Name	Component_1	Component_2	Component_3	Component_4	Component_5	Component_6
0	Eigenvalue	0.0000036730071738696753	0.010435918746027308	0.07701574213304058	0.2718202237800657	0.5262865042390427	5.114437938094649
1	Condition Number	1180.0164678487108	22.13775679417665	8.149091686552866	4.337685629418541	3.1173663258103557	1
2	Intercept	0.9999794556131322	0.00001724062001145557	0.000003057856075520253	2.2986685440246014e-9	4.966835986631746e-8	1.9394375241198394e-7
3	X1	0.990267656034644	0.008825799739191066	0.0001793519732054093	0.00046250118143391085	0.000252242229389297	0.000012448842136433821
4	X2	0.9991944090799727	0.0007867408719436721	6.572104360633705e-7	0.000008039446824656248	0.000009403635744160249	7.497550787411175e-7
5	X3	0.9922489492615014	0.00731719974698259	0.00011897926922988472	0.000300234112295231	0.000007557943800078451	0.000007079666190848475
6	X4	0.999758767919465	0.00006564016295910087	0.00012275090771918826	0.00003223260548425524	0.000019453368308914693	0.0000011550360633061476
7	Weights	0.0008422495872710054	0.1283418794355096	0.8313006103830848	0.00003765627985364275	0.03617747936598411	0.0033001249482969112

Notes about the RASON OData viewer

- Results generated via the endpoint `POST rason.net/api/model/{nameid}/solve` are displayed in the OData viewer by default.
- If no OData results are available for an instance either because it was not solved via `POST rason.net/api/model/{nameid}/solve` or because the solve failed, the results JSON text editor will open.
- When only one OData table exists in the solution, all rows are displayed by default.
- Where multiple OData tables are available for the solution, a table must be selected in the Table drop down in order to view the results.
- Column headers allow rows to be sorted.
- Columns may be filtered using the Columns dropdown menu.
- The pager control provides standard paging functionality.
- Viewer state (page, cell selection and scroll location) is saved and restored when switching between text and OData view or between tabs.
- Grid supports the copy and paste of selected cells.

Adding Input and Query Parameters in the Editor

To add a Query or Input Parameter to a model or decision flow, click **Select Query Parameters**, on the bottom of the Properties pane, to open the Query Parameters dialog. The query parameters are sent to the REST server when a model is POSTed or PUT. Those query parameters are saved in the model info as a query string. When the model is subsequently opened the query parameters are set in the properties automatically.

- Model Data Parameters**

Any data parameters that include the "binding": "get" parameter, can be called programmatically outside of the RASON model. In the Editor tab of www.RASON.com, users can click **Select Query Parameters** to edit each parameter value.

Example: Notice the binding parameter for both "profits" and "inventory" in the "data" section below and also within the Model Data Parameters section in the screenshot below.

```
...
"data": {

    "profits": {
        "dimensions": ["prod"],
        "value": [75, 50, 35],
        "binding": "get",
        "finalValue": []
    },
```

```

"inventory": {
  "dimensions": ["part"],
  "value": [450, 250, 800, 450, 600],
  "binding": "get",
  "finalValue": []
}

```

...

- Model Data Source Parameters

Any datasources parameters that include the "binding": "get" parameter, can be called programmatically outside of the RASON model. In the Editor tab of www.RASON.com, users can click Select Query Parameters to edit each parameter value.

Example: Notice the binding parameter for "cust_data" and "loan" in the "datasources" section below (and also for monthIncome, monthExpenses and loanAmt under "data") and also within the Model Data Parameters section in the screenshot below.

...

```

"datasources": {
  "cust_data": {
    "type": "csv",
    "connection": "customers.txt",
    "selection": "custID = ?",
    "parameters": {
      "cuID": {
        "binding": "get",
        "value": "c1"
      }
    }
  },
  "indexCols": ["custID"],
  "valueCols": ["age", "maritalStatus", "employmentStatus",
    "creditScore", "bankrupt"],
  "direction": "import"
},
"loan_data": {
  "type": "csv",

```

```

        "connection": "loans.txt",
        "selection": "loanID = ?",
        "parameters": {
            "loID": {
                "binding": "get",
                "value": "l1"
            }
        },
        "indexCols": ["loanID"],
        "valueCols": ["type", "rate", "term"],
        "direction": "import"
    }
},
"data": {
    "comment": "use binding to feed dif. values",
    "custExist": {
        "value": false
    },
    "custAge": {
        "value": 40,
        "binding": "cust_data",
        "valueCol": "age"
    },
    "maritalStatus": {
        "value": "s",
        "binding": "cust_data",
        "valueCol": "maritalStatus"
    },
    "employmentStatus": {
        "value": "selfEmployed",
        "binding": "cust_data",
        "valueCol": "employmentStatus"
    },
    "creditScore": {
        "value": 610,
        "binding": "cust_data",
        "valueCol": "creditScore"
    },
    "bankrupt": {
        "value": false,
        "binding": "cust_data",
        "valueCol": "bankrupt"
    },
    "monthIncome": {
        "value": 2500,
        "binding": "get"
    },
    "monthExpenses": {
        "value": 1000,
        "binding": "get"
    },
    "loanType": {
        "value": "standard",
        "binding": "loan_data",
        "valueCol": "type"
    },
    "loanRate": {

```

```

        "value": 5.0,
        "binding": "loan_data",
        "valueCol": "rate"
    },
    "loanTerm": {
        "value": 30,
        "binding": "loan_data",
        "valueCol": "term"
    },
    "loanAmnt": {
        "value": 100000.0,
        "binding": "get"
    }
},

```

Select REST query parameters and set values to use when solving the DTLoanStrategyExampleDatasource model.

API Parameters	
stage	Single stage name to solve
keep intermediate results	false
simplify final results	false
data-storage	Database
response-format	Workflow
schedule	Schedule for solving model
Model Data Parameters	
monthIncome	2500
monthExpenses	1000
loanAmnt	100000
Model Data Source Parameters	
curl	c1
id	11
Model Input Parameters	
No input parameter properties in the model.	

- **Model Input Parameters**

Any "inputparameters" can be called programmatically outside of the RASON model, *without* the need for the binding parameter. In the Editor tab of www.RASON.com, users can click Select Query Parameters to edit each parameter value.

Example: Notice the twelve input parameters ("E14:E19" and "F14:F19") in the "inputParameters" section below and also within the Model Input Parameters section in the screenshot below.

```

...
"inputParameters": {
    "E14:E19": {
        "type": "array",
        "value": [6, 5, 4, 3, 3, 1]
    },
    "F14:F19": {
        "type": "array",
        "value": [2, 5, 8, 4, 3, 2]
    }
},

```

Select Query Parameters to open the Query Parameters dialog to access these parameters outside of the RASON model. See the POST rason.net/API/model/id/solvetype endpoint, within the chapter Using the RASON API, to discover how to use a query parameter programmatically.

API Parameters

The POST rason.net/api/solve (Quick Solve) endpoint and the POST rason.net/api/model/{nameorid}/solvetype endpoint offer optional parameters controlling the content of the final results and where intermediate results are stored: stage, keep-intermediate-results, simplify-final-results, data-storage, response-format and schedule. (Scheduling not supported by Quick Solve endpoint).

Data-storage = DATABASE/JSON

This parameter controls whether the SQLite DATABASE or JSON file storage is used to store the results of *intermediate* stages. Database mode does not affect terminal stages. When database mode is selected, the in-memory SQLite database is used to store the results of intermediate stages and will spill on disk in out-of-memory conditions. The default setting is Database.

DATABASE mode can be orders of magnitude faster than JSON mode. In JSON mode, costs are incurred when reading/writing to/from disk. However, in DATABASE mode all in-memory database read/write operations are much more efficient. Intermediate results can still be stored and examined in DATABASE mode. These results may be queried with REST API or OData but they will *not* appear in the JSON response as serialized data frames.

Note: If *data-storage* =DATABASE, the results of intermediate stages (if kept) would be stored (and initially processed) only in SQLite DB. Therefore, even with *keep-intermediate-results=true*, you still will not see the actual evaluations for intermediate stages in the JSON response. You can only query these results using REST or OData APIs. (See below.) If results must be in JSON response, set *data-storage* to JSON. However, in any case with *keep-intermediate-results=true* the complete *status* object for each intermediate stage will be returned.

If data-storage = database and

- *keep-intermediate-results* =False, the database stays in memory and will be discarded after terminal stages are processed.

- `keep-intermediate-results = True`, an in-memory database is saved to disk to enable later querying.

Keep-intermediate-results = True/False

This parameter determines whether the RASON server stores the results of the *intermediate* stages in the decision flow, returns them with JSON response and makes them available for REST or OData querying later. Currently, this parameter not only affects the "pipeline solve" (i.e. when a single terminal stage is specified) but also the entire decision flow solve, which might have multiple terminal stages. The default setting is False.

Response-format = STANDALONE/WORKFLOW

Use this parameter to switch between the default dataframe-based decision flow reporting format (for both single and multi-stage decision flows) and the default reporting format (for single-stage decision flows only). When `response-format= WORKFLOW` and `simplify-final-results = false`, the formula outputs are reported as dataframes. The default setting is WORKFLOW.

Note: If a formula refers to a decision table, the dataframe will be complete with column headers-as defined in the "outputs" section of the decision table definition, and property types – that will be available in both the JSON response and later in REST/OData queries. If a generic formula, RASON will use default column names and the best possible type for each result column.

Schedule=interval/automatic (Not supported by Quick Solve)

"interval" (ISO 8601 time or repeating time interval)- If time is specified, the RASON Server will run the model once on that day and time. If a repeating time interval is specified, the server will run the model at the starttime. After the model finishes and duration has elapsed, the server will run the model again for the specified number of repetitions. Applications will typically use a model name (rather than an ID) in API Calls to `GET rason.net/api/model/{name}/status` to refer to the most recent run.

"automatic" - This parameter behaves differently on standalone models versus decision flows. RASON Decision Services utilizes a new property, "modelRecency": "time-since-last-run", at the same level as the existing `modelName`, `modelDescription` and `modelType` properties.

This new property informs the RASON server of how recently the model was run in order to determine if the results (in JSON or OData form) can be considered "current". The object "time-since-last run" must be in ISO 8601 time duration format (i.e. "PT12H" for 12 hours or "P1W" for 1 week). With this property in place, the user may add "`?schedule=automatic`" to the existing REST API call `POST rason.net/api/model/{nameorid}/solve`. The RASON Server will consult past runs to determine the maximum time a solve usually takes to complete (max-time), and will arrange to run the model at intervals of time-since-last-run – max-time. If the "modelRecency" property is omitted, the default value is infinity; the "`?schedule=automatic`" will simply cause the model to run one time.

More importantly, when a workflow of multiple stages is run using `POST rason.net/api/model/{nameorid}/solve?schedule=automatic`, the RASON Server will take into account `modelRecency` information for each *stage* in the *workflow*. For example, suppose a workflow has a final stage dependent on two intermediate stages A and B: If both A and B have runs that satisfy their recency requirement, the RASON Server will just run the final stage, using the previous results from A and B. If A has `modelRecency=PT24H`, takes 2 hours to run and last finished 23 hours ago, and B has `modelRecency=Pt8H`, takes 1 hour to run and last finished 6.5 hours ago, the server will re-run both A and B, and use the latest results (2 hours from now) to run the final stage. (The `modelRecency` property must be added to each stage in the decision flow.)

- Calling `POST rason.net/api/model/{name}/solve?keep-intermediate-results=true&schedule=automatic` solves the flow and persists the stage results on the server. These results may be accessed using the OData endpoint described in the next section.
- To obtain the results from the last scheduled run, click the Editor tab at www.RASON.com. The results will be listed under Results beneath the model name. In order to obtain results from an earlier scheduled

run, click My Account, then Run Details on the Overview tab. Highlight and copy the Model ID of the desired model instance and use this Model ID while calling the REST API Endpoint, GET `rason.net/api/model/{nameorid}/result`.

Simplify-final-results = True/False

When True, this parameter tells the engine to store and report the results of, only, terminal stages as simplest possible JSON object: a scalar for 1x1 dataframe, 1d - array for Nx1 dataframe and 2d – array for NxM dataframe, No headers or index columns are stored. This option does not affect the results of intermediate stages, these are always dataframes. These simple results are reported in JSON Response may be queried with a REST call. Currently OData does not recognize these objects. The default setting is False.

Stage=<stage-name>

Use this parameter to retrieve information related to a specific stage of a decision flow. If this parameter is passed, the endpoint solves only the part of the decision flow required to solve the specified terminal stage, <stage-name>. The information produced by this optional parameter may be helpful during the creation or debugging phase. If a stage is not specified, a full Directed Acyclic Graph (DAG) solve is performed. To view all stages in a decision flow, click or use the REST endpoint GET:

`https://rason.net/api/model/{nameorid}/stages/all`.

When submitted together, the two parameters, data-storage and keep-intermediate-results, create four cases.

Example: POST: `https://rason.net/api/model/{nameorid}/solve?data-storage=json&keep-intermediate-results=true`

- data-storage = JSON and keep-intermediate-results = False

Intermediate results are discarded after the decision flow solve and are not reported in JSON response and are not available for querying. Results for terminal stages are stored as JSON. OData view is available for terminal stages only.

- data-storage = DATABASE and keep-intermediate-results = False

Same as above, since terminal stage results are never stored in the database. OData view is available for terminal stages only.

- data-storage = JSON and keep-intermediate-results = True

SQLite database is not used to store results. Results of all stages (intermediate and terminal) are stored as JSON. Odata view is available for all stages, intermediate and terminal.

- data-storage = DATABASE and keep-intermediate-results = True

SQLite database is used for solving and storing the results of intermediate stages. Results of all stages (intermediate and terminal) are stored in JSON. OData view is available for all stages, intermediate and terminal.

Model Data Parameters

In the Introduction to this guide, within the Binding to Data and Dimensional Flexibility section, the "binding" property was introduced. This property allows new data (parameters) to be passed directly in the REST API call allowing users to perform "what if" analysis without changing the original RASON model. In the example RASON model, AirlineHubConic5.json (in the RASON example model list under Optimization – Conic

Optimizations), the "binding" parameter (in red below) has been added to the data section for the x coordinates and the y coordinates. (Recall that this example model finds the optimal location for an airline hub.)

```
"data": {
  "corx": {
    "dimensions": [ 6 ],
    "binding": "get",
    "value": [ 1, 0.5, 2, 2, 2, 0.5 ]
  },
  "cory": {
    "dimensions": [ 6 ],
    "binding": "get",
    "value": [ 4, 3, 4, 2, 5, 6 ]
  }
},
```

The Query Parameters dialog on the RASON Editor page of www.RASON.com, allows easy access to this data. Simply select the data parameters that you would like to alter, and then type in the new value(s).

The image shows two screenshots of the 'Query Parameters' dialog box. The left screenshot shows the 'Model Data Parameters' section with 'corx' and 'cory' selected. The right screenshot shows the same dialog with new values entered for 'corx' and 'cory'. A blue arrow points from the left to the right, labeled 'Enter New Value(s)'.

Then Save and use the POST rason.net/api/model/{nameorid}/solve endpoint to solve the model.

Table	obj	Table	obj
ID	finalValue	ID	finalValue
0	2.1360009792267305	0	2.1360009792267305

Model Data Source Parameters

The "binding" parameter may also be used within the datasources section of a RASON model. When used here, a data parameter can be changed outside of the RASON model providing "what if analysis" functionality. In the example model DT Loan Strategy Model and Datasource.json (located under Decision Tables in the RASON Examples List), the "cust_data" and "loan_data" datasources allow access to the parameters cuID and loID outside of the RASON model.

```
"datasources": {
  "cust_data": {
```

```

        "type": "csv",
        "connection": "customers.txt",
        "selection": "custID = ?",
        "parameters": {
            "cuID": {
                "binding": "get",
                "value": "c1"
            }
        },
        "indexCols": ["custID"],
        "valueCols": ["age", "maritalStatus", "employmentStatus",
            "creditScore", "bankrupt"],
        "direction": "import"
    },
    "loan_data": {
        "type": "csv",
        "connection": "loans.txt",
        "selection": "loanID = ?",
        "parameters": {
            "loID": {
                "binding": "get",
                "value": "l1"
            }
        },
        "indexCols": ["loanID"],
        "valueCols": ["type", "rate", "term"],
        "direction": "import"
    }
},
}

```

This example model takes two input parameters cuID (customer id) and loID (loan id) and determines if a loan will be granted or not, based on the customer's financial information. By assigning a "binding" parameter to the input parameters, a loan officer is able to vary the customer and loan type. She could do this easily on the Editor tab of www.RASON.com by clicking the Select Query Parameters button on the Properties pane (on the right) to bring up the Query Parameters dialog.

Query Parameters

Select REST query parameters and set values to use when solving the DTLLoanStrategyExampleDataSource version 2021-04-01-21-00-37-484236 model.

API Parameters

- ☐ stage: Single stage name to solve
- ☐ keep-intermediate-results: false
- ☐ simplify-final-results: false
- ☐ data-storage: Database
- ☐ response-format: Workflow
- ☐ schedule: Schedule for solving model

Model Data Parameters

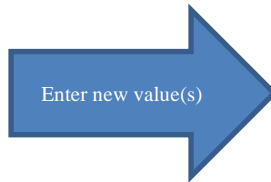
- ☐ monthIncome: 2500
- ☐ monthExpenses: 1000
- ☐ loanAmnt: 100000

Model Data Source Parameters

- ☒ cuID: c1
- ☒ loID: l1

Model Input Parameters

No input parameter properties in the model.



Select REST query parameters and set values to use when solving the DTLLoanStrategyExampleDataSource version 2021-04-01-21-00-37-484236 model.

API Parameters

- ☐ stage: Single stage name to solve
- ☐ keep-intermediate-results: false
- ☐ simplify-final-results: false
- ☐ data-storage: Database
- ☐ response-format: Workflow
- ☐ schedule: Schedule for solving model

Model Data Parameters

- ☐ monthIncome: 2500
- ☐ monthExpenses: 1000
- ☐ loanAmnt: 100000

Model Data Source Parameters

- ☒ cuID: c2
- ☒ loID: l2

Model Input Parameters

No input parameter properties in the model.

Cancel Save

Then Save and use the POST rason.net/api/model/{nameorid}/solve endpoint to solve the model.

Table	routing
ID	routing
0	accept

Model Input Parameters

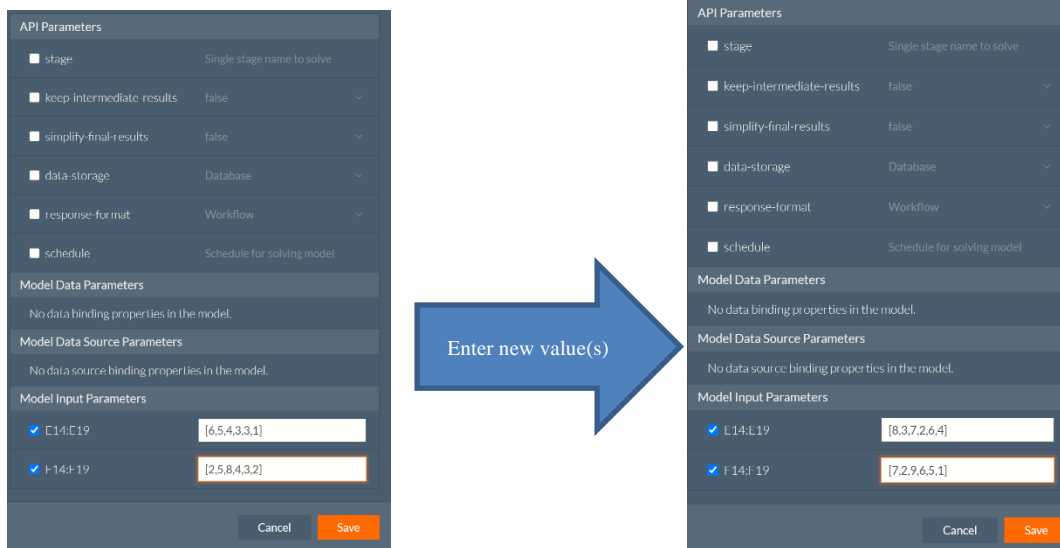
Input parameters do not require the "binding": "get" property in order to be accessed outside of the RASON model. In the example code below, the "E14:E19" and "F14:F19" input parameters allow access to the parameters to each of the data parameters in the RASON model.

This example model has been modified to utilize 12 input parameters, E14:E19 and F14:F19. These are the coordinates of each city in the BoxFunOptimize example model (downloadable by clicking Rason example models – Decisions – Custom Box Functions -- Optimization Model with Box Function). The purpose of this example is to find a location for a new airport that is equidistant to six different cities. Using an inputParameter, a public planner would be able to vary the locations of the six cities that the proposed airport is hoping to serve. She could do this easily on the Editor tab of www.RASON.com by clicking the Select Query Parameters button on the Properties pane (on the right) to bring up the Query Parameters dialog.

```
...
"inputParameters": {
  "E14:E19": {
    "type": "array",
    "value": [6, 5, 4, 3, 3, 1]
  },
  "F14:F19": {
    "type": "array",
    "value": [2, 5, 8, 4, 3, 2]
  }
}
```

...

The Query Parameters dialog on the Rason Editor page of www.Rason.com, allows easy access to this data. Simply select the input parameters that you would like to alter, and then type in the new value(s).



Then Save and use the POST rason.net/api/model/{nameorid}/solve endpoint to solve the model.

Table		e13_f13
ID	:	finalValue
0	:	5.4998911227470035
1	:	5.0000408289698735

Note: New values for input parameters are passed programmatically in the same way as a query parameter is passed. In this case, the new input values can be passed programmatically as: E14:E19=[8,3,7,2,6,4]. Passing a single element in a range is not supported. If an input parameter is defined as a "range", the entire range must be passed, i.e. in this example E15=[3] would not be supported.

Solving An "In-Line" Decision Flow and Display Results

A decision flow stage may either be made up of "inline" models, where the model is written within the decision flow stage script, or a stage may invoke a "reusable model", which is a slightly modified standalone model. (A standalone RASON model is one that can be successfully solved "on its own", meaning that it contains all required elements of a RASON model: data, decision variables, constraints, uncertain variables, uncertain functions, etc. An example of a standalone model is the Product Mix model provided in RASON Examples folder (and solved in the example above.) A decision flow may contain both types of stages.

An example of an "inline" decision flow appears below. This decision flow contains two stages 1. optStage (where the stochastic optimization takes place) and 2. simStage (where the simulation (only) takes place). To open this model, navigate to the RASON.com Editor tab, click the RASON Examples folder on the ribbon, then click Decision Flows – Inline – "Optimization – Simulation Inline Decision Flow". Note that Stage 1 receives no data from an outside source. All model elements are contained within optStage. However, stage 2 contains a datasources section which receives the final decision variable values from optStage as input.

```
{
  flowName: 'optSimWorkflow',
  optStage: {
    "comment": "Uncertain optimization by transformation",
```

Decision Flow Name

```

"modelSettings": {
  "transformStochastic": "robustCounterpart",
  "numTrials": 100,
  "randomSeed": 1
},
"variables": {
  "x": {
    "dimensions": [8],
    "type": "binary",
    "initialValue": [],
    "finalValue": [],
    "dualValue": []
  }
},
"uncertainVariables": {
  "c": {
    "dimensions": [8]
  },
  "c[1]": {
    "formula": "PsiTriangular(400000, 500000, 900000)"
  },
  "c[2]": {
    "formula": "PsiTriangular(500000, 750000, 1250000)"
  },
  "c[3]": {
    "formula": "PsiTriangular(500000, 1000000, 1500000)"
  },
  "c[4]": {
    "formula": "PsiTriangular(400000, 600000, 900000)"
  },
  "c[5]": {
    "formula": "PsiTriangular(250000, 500000, 750000)"
  },
  "c[6]": {
    "formula": "PsiTriangular(300000, 500000, 600000)"
  },
  "c[7]": {
    "formula": "PsiTriangular(200000, 450000, 700000)"
  },
  "c[8]": {
    "formula": "PsiTriangular(400000, 500000, 700000)"
  },
  "d": {
    "dimensions": [8],
    "formula": "PsiBinomial(1, 0.9)"
  }
},
"formulas": {
  "f": {
    "value": [325000, 450000, 550000, 300000, 150000, 250000, 150000, 325000]
  },
  "cash": {
    "formula": "sumproduct(c * d - f, x)"
  }
},
"constraints": {
  "invest": {
    "formula": "sumproduct(f, x)",
    "upper": 1500000
  }
},
"objective": {

```

Stage 1: optStage
Solves a stochastic optimization model containing 8 decision variables in the x array, 16 uncertain variables, c[1] thru c[8] and d (which is an array of 8 uncertain variables all using the same distribution, PsiBinomial(1, 0.9)), 1 constraint and an objective function.

In order to receive a specific output in your results, you must ask for it in the RASON script. For example in optStage, the x array under "variables" asks for three outputs, the initial variable value (initialValue), the final variable value (finalValue) and sensitivity information via the dual value (dualValue). Under simStage, the uncertain function, cash, asks for the mean value in the output, ("mean": []).

```

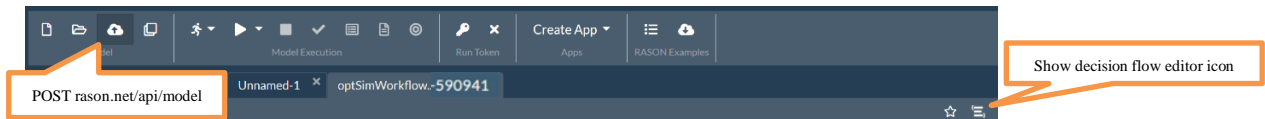
    "totalObjective": {
      "type": "maximize",
      "formula": "cash",
      "chanceType": "ExpVal",
      "finalValue": []
    }
  },
  "simStage":
  {
    "comment": "Simulation at optimization result
    point",
    "inputParameters": {
      "x": {
        "value": "optStage.x.finalValue"
      }
    },
    "uncertainVariables" : {
      "c": {
        "dimensions": [8]
      },
      "c[1]": {
        "formula": "PsiTriangular(400000, 500000, 900000)"
      },
      "c[2]": {
        "formula": "PsiTriangular(500000, 750000,
        1250000)"
      },
      "c[3]": {
        "formula": "PsiTriangular(500000,
        1000000,1500000)"
      },
      "c[4]": {
        "formula": "PsiTriangular(400000, 600000,
        900000)"
      },
      "c[5]": {
        "formula": "PsiTriangular(250000, 500000,
        750000)"
      },
      "c[6]": {
        "formula": "PsiTriangular(300000, 500000,
        600000)"
      },
      "c[7]": {
        "formula": "PsiTriangular(200000, 450000,
        700000)"
      },
      "c[8]": {
        "formula": "PsiTriangular(400000, 500000,
        700000)"
      },
      "d": {
        "dimensions": [8],
        "formula": "PsiBinomial(1, 0.9)"
      }
    },
    "data": {
      "f": { value: [325000, 450000, 550000, 300000,
        150000, 250000, 150000, 325000] },
      "x": { "binding": "xsrc" }
    },
    "uncertainFunctions": {
      "cash": { "formula": "sumproduct(c * d - f, x)",
      "mean": []
    }
  }
}

```

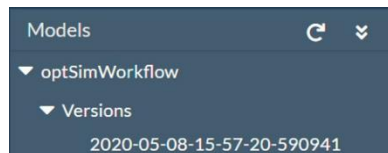
Stage 2: simStage
Receives the final variable values from Stage 1 as input to Stage 2.

Notice the inputParameters section in simStage where the final variable values from optStage are passed. For more information on the inputParameters section in the RASON Reference Guide.

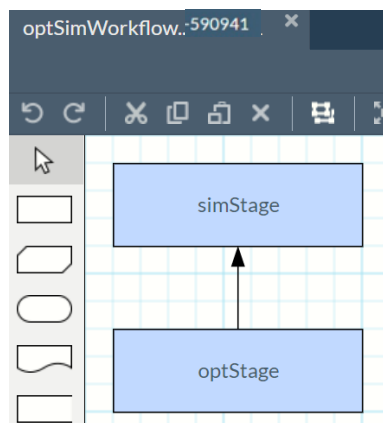
Click POST rason.net/api/model to post the decision flow to the RASON Server, then click the "Show decision flow editor" icon to toggle from the RASON model editor to the RASON Decision Flow editor.



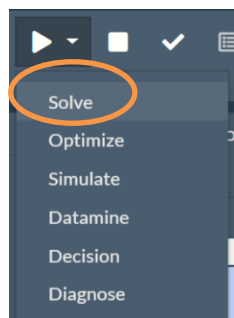
The decision flow is listed in the Models task pane under the decision flow name, optSimWorkflow. Recall that the top level property, flowName, assigns the name to the decision flow. While this property is optional it is recommended for easier model retrieval later.



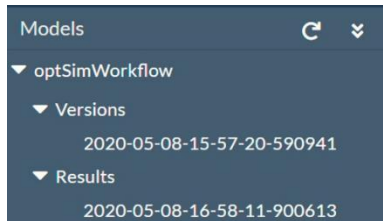
Once "Show decision flow editor" is clicked the RASON model editor toggles to the RASON Decision Flow editor displaying the OptSimWorkflow decision flow. This graphical representation displays the two stages in the decision flow, simStage and optStage, and also shows the direction of flow, i.e. optStage is the beginning node, or first stage, and simStage is the second and terminal (end) stage.



To solve the decision flow, you must use the POST rason.net/api/model/{nameorid}/solve endpoint. Endpoints such as POST rason.net/api/model/{nameorid}/solvetype = optimize, simulate, calculate or datamine, do not support the solving of decision flows.



Click Solve to solve the decision flow. The results will be updated on the Models task pane.



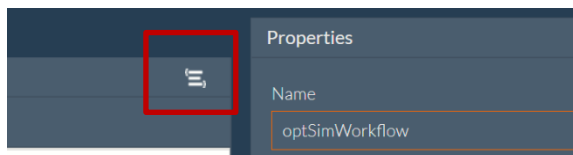
Using the OData Endpoint

Double click the entry under Results to open.

The OData results viewer opens and displays the results from the decision flow in an easy to read table. Click the down arrow next to Table to view the selected results. You can filter the columns displayed in the table by clicking the down arrow next to Columns.

Table	simStage_cash_statistics
ID	mean
0	1391678.557729035

Click the "Show JSON results" icon on the top right corner of the Editor window to display the results in a JSON response.



```
{
  "status": {
    "id": "2590+optSimWorkflow+2020-05-08-16-58-11-900613",
    "code": 0,
    "codeText": "Success",
    "solveTime": 2661
  },
  "results": {
    "simStage.cash.statistics": ["mean"],
    "simStage.d.statistics": ["mean"]
  },
  "simStage": {
    "status": {
      "id": "2590+simStage+2020-05-08-16-58-15-620182",
      "code": 0,
      "codeText": "Solver has completed the simulation.",
      "solveTime": 133
    },
    "cash": {
      "statistics": {
        "objectType": "dataFrame",
        "name": "simStage.cash.statistics",
        "order": "col",
        "colNames": ["mean"],
        "colTypes": ["double"],
        "indexCols": null,
        "data": [
          [1379603.7262184166]
        ]
      }
    }
  }
}
```

"Status" returns the results ID and the Decision Flow result, in this case Success, which means that the decision flow was able to execute successfully, and the total solve time.

"Results" returns each requested statistic in the simulation model.

"simStage" begins the results section for the 2nd stage. Notice that the simulation finished successfully in 133 milliseconds.

"cash" returns the mean for the uncertain function, cash.

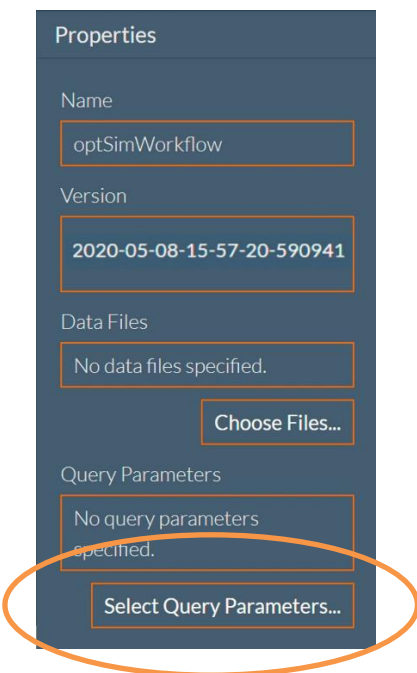
}

Notice that no results for optStage are returned. By default, when the POST `rason.net/api/model/{nameorid}/solve` endpoint is used to solve a workflow, RASON will compute, store and report results for terminal stages only (in addition to a status for the overall workflow). Although RASON computes the results that are required to solve terminal stages, these results are not stored or reported *unless two* URL parameters are passed to the REST API endpoint POST `rason.net/api/model/{nameorid}/solve`: `keep-intermediate-results=true` and `data-storage=JSON`. See the important note on using the ODATA endpoint, GET `rason.net/odata/result` below.

- `keep-intermediate-results=true` determines whether the RASON server stores the results of the *intermediate* stages in the decision flow, returns them with JSON response and makes them available for REST or OData querying later. Currently, this parameter not only affects the "pipeline solve" (i.e. when a single terminal stage is specified) but also the entire decision flow solve, which might have multiple terminal stages. The default setting is False.
- `data-storage=JSON` controls whether the SQLite DATABASE or JSON file storage is used to store the results of *intermediate* stages. If the default, `data-storage=database` is used, intermediate results can still be stored and examined using the REST API ODATA endpoint GET `rason.net/odata/result`, but the results will *not* appear in the JSON response as serialized data frames.

When both URL parameters are submitted together, `data-storage=JSON` and `keep-intermediate-results=true`, JSON files are used for solving and storing, responses and querying. The results of intermediate and terminal stages are reported in the JSON response as well as being available for REST/OData querying.

To supply both parameters to the RASON graphical editor, simply click Select Query Parameters in the bottom of the Properties pane to open the Query Parameters dialog.



Select `keep-intermediate-results` and set to True. Set `data-storage` and set to JSON.

Query Parameters

Select REST query parameters and set values to use when solving the optSimWorkflow 2020-05-08-15-57-20-590941 model.

API Parameters

stage

Single stage name to solve

keep-intermediate-results

true

simplify-final-results

false

data-storage

JSON

response-format

Workflow

schedule

Schedule for solving model

Model Data Parameters

No data binding properties in the model.

Model Data Source Parameters

For more information on the remaining options on this dialog, see the section What is a Decision Flow? above. Use POST rason.net/api/model/{nameorid}/solve to solve the model on the RASON server, check the status using GET rason.net/api/model/{nameorid}/status and obtain the results using GET rason.net/api/model/{nameorid}/result.

The results open in the OData viewer. Notice the additional information available under "Table" pertaining to "optStage".

Table	optStage_x	Co
ID	initialValue	finalValue
0	0	1
1	0	0
2	0	1
3	0	1
4	0	1
5	0	0
6	0	1
7	0	0

Table	optStage_totalObjective
ID	finalValue
0	1390897.9539955165

Click "Show JSON results" and notice the optStage status of "Integer solution found within tolerance.", along with the initial and final variable values and the final objective value are now returned.

```

{
  "status": {
    "id": "2590+optSimWorkflow+2020-05-11-14-21-56-113221",
    "code": 0,
    "codeText": "Success",
    "solveTime": 353
  },
  "results": {
    "optStage.totalObjective": ["finalValue"],
    "optStage.x": ["initialValue", "finalValue"]
    "simStage.cash.statistics": ["mean"],
    "simStage.d.statistics": ["mean"]
  },
  "optStage": {
    "status": {
      "id": "2590+optStage+2020-05-11-14-21-58-248297",
      "code": 14,
      "codeText": "Integer solution found within tolerance.",
      "solveTime": 166
    }
  }
}

```

• 119

```

    },
    "engine": "Gurobi Solver",
    "x": {
        "objectType": "dataFrame",
        "name": "x",
        "order": "col",
        "colNames": ["initialvalue", "finalvalue"],
        "colTypes": ["double", "double"],
        "indexCols": null,
        "data": [
            [1, 0, 1, 1, 1, 0, 1, 0]
        ]
    },
    "totalObjective": {
        "objectType": "dataFrame",
        "name": "totalObjective",
        "order": "col",
        "colNames": ["finalvalue"],
        "colTypes": ["double"],
        "indexCols": null,
        "data": [
            [1390897.9539955165]
        ]
    }
},
"simStage": {
    ...
}

```

Important Note: Another option is to pass `keep-intermediate-results = true` as a URL parameter to the REST API endpoint `POST rason.net/api/model/{nameorid}/solve` (POST `rason.net/api/model/{nameorid}/solve?keep-intermediate-results=true`) and access those results using the ODATA endpoint `GET rason.net/ODATA/result`. For more information on this endpoint, see the section *Using the OData Endpoints* within the **Using the REST API** chapter that appears later in this guide.

The REST API Endpoint `GET rason.net/ODATA/result` requires two headers. The first is the Authorization header and the second is the resource ID. When both headers are passed to this endpoint, all results for the given model will be returned.

```

{
  "@odata.context": "https://rason.net/odata/$metadata#Result(Data())",
  "value": [
    {
      "Name": "optStage_invest",
      "Data": [
        {
          "_ID": 0,
          "finalvalue": 1475000.0
        }
      ]
    },
    {
      "Name": "optStage_totalObjective",
      "Data": [
        {
          "_ID": 0,
          "finalvalue": 1390897.9539955165
        }
      ]
    },
    {
      "Name": "optStage_x",
      "Data": [
        {
          "_ID": 0,
          "initialvalue": 0.0,
          "finalvalue": 1.0
        }
      ]
    }
  ]
}

```

```

        "_ID": 1,
        "initialvalue": 0.0,
        "finalvalue": 0.0
    },
    {
        "_ID": 2,
        "initialvalue": 0.0,
        "finalvalue": 1.0
    },
    {
        "_ID": 3,
        "initialvalue": 0.0,
        "finalvalue": 1.0
    },
    {
        "_ID": 4,
        "initialvalue": 0.0,
        "finalvalue": 1.0
    },
    {
        "_ID": 5,
        "initialvalue": 0.0,
        "finalvalue": 0.0
    },
    {
        "_ID": 6,
        "initialvalue": 0.0,
        "finalvalue": 1.0
    },
    {
        "_ID": 7,
        "initialvalue": 0.0,
        "finalvalue": 0.0
    }
]
},
{
    "Name": "simStage_c[1]_statistics",
    "Data": [
        {

```

Given these results, call GET `rason.net/odata/result/optStage_XXX` to query the calculation of the invest formula (`optStage_invest`), your objective function (`optStage_totalobjective`) or inspect the final decision variables (`optStage_x`). For example, calling GET `rason.net/odata/result/optStage_x`, returns both the final and initial decision variable values, as shown below.

```

{
    "@odata.context": "https://RASONrestapi2-test.azurewebsites.net/odata/$metadata#Result(Data())/ $entity",
    "Name": "optStage_x",
    "Data": [
        {
            "_ID": 0,
            "initialvalue": 0.0,
            "finalvalue": 1.0
        },
        {
            "_ID": 1,
            "initialvalue": 0.0,
            "finalvalue": 0.0
        },
        {
            "_ID": 2,
            "initialvalue": 0.0,
            "finalvalue": 1.0
        },
        {
            "_ID": 3,
            "initialvalue": 0.0,
            "finalvalue": 1.0
        }
    ]
}

```

```

        "_ID": 4,
        "initialvalue": 0.0,
        "finalvalue": 1.0
    },
    {
        "_ID": 5,
        "initialvalue": 0.0,
        "finalvalue": 0.0
    },
    {
        "_ID": 6,
        "initialvalue": 0.0,
        "finalvalue": 1.0
    },
    {
        "_ID": 7,
        "initialvalue": 0.0,
        "finalvalue": 0.0
    }
]
}

```

Solving a Decision Flow with Reusable Models

Now let's create this same decision flow by invoking a model in each stage, rather than supplying the model within the stage script. Invoking reusable models in a decision flow allows more flexibility in that the reusable models are written once but may be used in multiple decision flows.

1st Stage Reusable Model

Let's start with the model for our first stage, `optStage`. Recall that this stage solves a stochastic optimization model. If you compare the model provided "inline" to stage `optStage` and the reusable `optStage` model, you'll notice only two lines of code are different, the top two lines of code: `modelName` and `modelType`.

- `modelName` – Assigns a name to a model, reusable or standalone for easy retrieval later.
- `modelType`: Assigns a model type such as optimization, simulation, datamine or calculation. For stochastic optimization models, use "optimization". For more information on this high level property, see below.

The rest of the RASON script in the reusable model is identical to the inline RASON model.

To invoke an optimization, simulation or decision table as a reusable model, use the `invokeModel` property within the decision flow.

For example:

```

{
  "flowName": "optDMWorkflow",
  "optStage": {
    "invokeModel": "product-mix-
reusable:product-
mix-reusable.xlsx!Product Mix Example 2",
    "modelType": "optimization",
    "outputResults": {
      "Number_to_Build": {...

```

The `invokeModel` property must be of the form

"<modelName>:<workbook>!<worksheet>"

See the subsequent section Invoking an Excel Model in a Decision Flow for more information.

```

{
  "comment": "Uncertain optimization by transformation",
  "modelName": "opt_StageReusable",
  "modelType": "optimization",

```

```

"modelDescription": "gas-company-chance converted into a reusable model",
"modelSettings": {
  "transformStochastic": "robustCounterpart",
  "numTrials": 100,
  "randomSeed": 1
},
"variables": {
  "x": {
    "dimensions": [8],
    "type": "binary",
    "initialValue": [],
    "finalValue": [],
    "dualValue": []
  }
},
"uncertainVariables": {
  "c": {
    "dimensions": [8]
  },
  "c[1]": {
    "formula": "PsiTriangular(400000, 500000, 900000)"
  },
  "c[2]": {
    "formula": "PsiTriangular(500000, 750000, 1250000)"
  },
  "c[3]": {
    "formula": "PsiTriangular(500000, 1000000, 1500000)"
  },
  "c[4]": {
    "formula": "PsiTriangular(400000, 600000, 900000)"
  },
  "c[5]": {
    "formula": "PsiTriangular(250000, 500000, 750000)"
  },
  "c[6]": {
    "formula": "PsiTriangular(300000, 500000, 600000)"
  },
  "c[7]": {
    "formula": "PsiTriangular(200000, 450000, 700000)"
  },
  "c[8]": {
    "formula": "PsiTriangular(400000, 500000, 700000)"
  },
  "d": {
    "dimensions": [8],
    "formula": "PsiBinomial(1, 0.9)"
  }
},
"formulas": {
  "f": {
    "value": [325000, 450000, 550000, 300000, 150000, 250000, 150000, 325000]
  },
  "cash": {
    "formula": "sumproduct(c * d - f, x)"
  }
},
"constraints": {
  "invest": {
    "formula": "sumproduct(f, x)",
    "upper": 1500000
  }
},
"objective": {
  "totalObjective": {
    "type": "maximize",
    "formula": "cash",
    "chanceType": "ExpVal",
    "finalValue": []
  }
}
}

```

2nd Stage Reusable Model

Now let's create the reusable model that will be invoked from the 2nd stage, simStage. If you've already correctly deduced that the model will be the same as the model within simStage, pat yourself on the back! Since we are simply running a simulation after the stochastic optimization has finished, the code for the simulation reusable model, we'll call it simStageReusable, will be the same as the original simStage, with the addition of the inputParameters section. (See below.)

```
{
  "modelName": "simStageReusable",
  "modelType": "simulation",
  "modelDescription": "gas-company-chance converted into a reusable model",
  "inputParameters": {
    "x": {
      "type": "array",
      "value": null,
      "comment": "pass the final variable values from optStage"
    }
  },
  "uncertainVariables": {
    "c": {
      "dimensions": [8]
    },
    "c[1]": {
      "formula": "PsiTriangular(400000, 500000, 900000)"
    },
    "c[2]": {
      "formula": "PsiTriangular(500000, 750000, 1250000)"
    },
    "c[3]": {
      "formula": "PsiTriangular(500000, 1000000, 1500000)"
    },
    "c[4]": {
      "formula": "PsiTriangular(400000, 600000, 900000)"
    },
    "c[5]": {
      "formula": "PsiTriangular(250000, 500000, 750000)"
    },
    "c[6]": {
      "formula": "PsiTriangular(300000, 500000, 600000)"
    },
    "c[7]": {
      "formula": "PsiTriangular(200000, 450000, 700000)"
    },
    "c[8]": {
      "formula": "PsiTriangular(400000, 500000, 700000)"
    },
    "d": {
      "dimensions": [8],
      "formula": "PsiBinomial(1, 0.9)"
    }
  },
  "formulas": {
    "f": {
      "value": [325000, 450000, 550000, 300000, 150000, 250000, 150000, 325000]
    },
    "x": {
      "value": "finalVarValues"
    }
  },
  "uncertainFunctions": {
    "cash": {
      "formula": "(c * d - f)",
      "mean": []
    }
  }
}
```

Reusable models
receive input using the
inputParameters
section.

Recall from the original inline decision flow that the final decision variables from optStage were passed to simStage using the inputParameters section.

```
"inputParameters": {
  "x": {
    "value": "optStage.x.finalValue"
  }
},
```

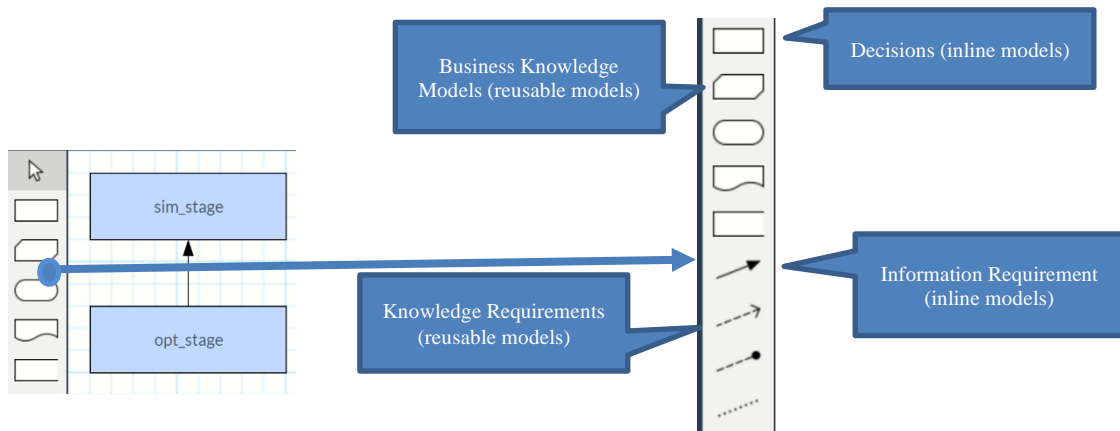
This is also the case in simStageReusable.

In both inline decision flows and decision flows that invoke reusable models, the input parameters section passes input data from previous stages.

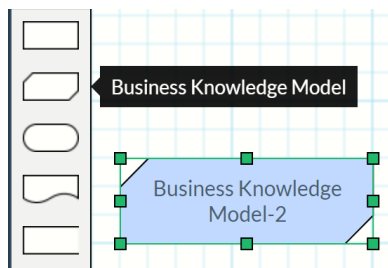
In the reusable model, you see the inputParameters section passing the final decision variable values from the stochastic optimization model, optStageReusable, to the simulation model in simStageReusable. Notice that three properties, "type", "value" and "comment" are passed for finalVarValues. Only "value": null is mandatory, "type" and "comment" are optional. The "type" property is used to inform RASON of the type of data being passed for the given input parameter. Supported properties are: number, string or text, Boolean, array or dataset (for data science flows only). The "comment" property optionally allows for more details, if desired. Note that the name(s) for the inputParameters, finalVarValues, is user-defined.

Creating the Decision Flow

We are now ready to create our decision flow. When creating a decision flow using the graphical editor, it's important to understand that Business Knowledge Models are for use with reusable models while Decisions are for use with inline models.



Recall from our previous example, the inline decision flow, that the flow was made up of two decisions. However, since we will now be invoking reusable models, we will select the Business Knowledge icon and drag it onto the grid. Notice also the connect (the arrow) between these two stages is an Information Requirement (solid arrow) while Business Knowledge stages must be connected using a Knowledge Requirement (a dotted arrow)



On the Properties pane (top right), type "optStageReusable" and then press <enter>.

Properties

Invoke Model

opt_stageReusable

If model is saved within a OneDrive account, use:
Name=MyExcelConnection where *MyExcelConnection* is the name given to the Data Connection in the user's MyAccount. For more information on Data Connections, see the previous chapter.

The pane will update and populate the fields. Notice that the "type" of model is an "optimization", that there are no input arguments to this first stage and the list of output results available as inputs to the next stage.

Properties

Invoke Model

opt_stageReusable

Name

opt_stageReusable

Type

Optimization

Recency

Comment

This interface has been generated by Psi for a

Inputs

No inputs defined.

Outputs

x.initialValue
x.finalValue
x.dualValue
totalObjective.finalValue

Model Invoked in Decision Flow

Name of Reusable Model

A Quick note on Recency

This property informs the RASON server of how recently the reusable model was run in order to determine if the results (in JSON or OData form) can be considered "current". The object "time-since-last run" must be in ISO 8601 time duration format (i.e. "PT12H" for 12 hours or "P1W" for 1 week). The RASON Server will consult past runs to determine the maximum time a solve usually takes to complete (max-time), and will arrange to run the model at intervals of time-since-last-run – max-time. If the "Recency" is omitted, the default value is infinity; causing the model to run one time. See the next section to see an example which uses Recency.

More importantly, when a workflow of multiple stages is run using POST `reason.net/api/model/{nameorid}/solve`, the RASON Server will take into account modelRecency information for each *stage* in the *workflow*. For example, suppose a workflow has a final stage dependent on two intermediate stages A and B: If both A and B have runs that satisfy their recency requirement, the RASON Server will just run the final stage, using the previous results from A and B. If A has modelRecency=PT24H, takes 2 hours to run and last finished 23 hours ago, and B has modelRecency=Pt8H, takes 1 hour to run and last finished 6.5 hours ago, the server will re-run both A and B, and use the latest results (2 hours from now) to run the final stage. (The Recency property must be added to each stage in the decision flow.) For more information on scheduling workflows, see the entry for POST `reason.net/api/model/{nameorid}/solve` within the chapter, **Using the REST API** or continue on to the next section, *More on Scheduling*.

Click Show RASON model editor to toggle back to the RASON editor to view the automatically produced code.

```
{
  "flow": "Unnamed-3",
  "comment": "",
  "optStageReusable": {
    "comment": "This interface has been generated by Psi for a RASON model",
    "invokeModel": "optStageReusable",
    "modelType": "optimization",
    "outputResults": {
      "x": {
        "evaluations": ["initialValue", "finalValue"],
        "type": "array/number",
        "comment": "decision variable block"
      },
      "totalObjective": {
        "evaluations": ["finalValue"],
        "type": "number",
        "comment": "objective function"
      }
    }
  },
  "modelDescription": "gas-company-chance converted into a reusable model"
}
```

Rename to "optsimReusableFlow"

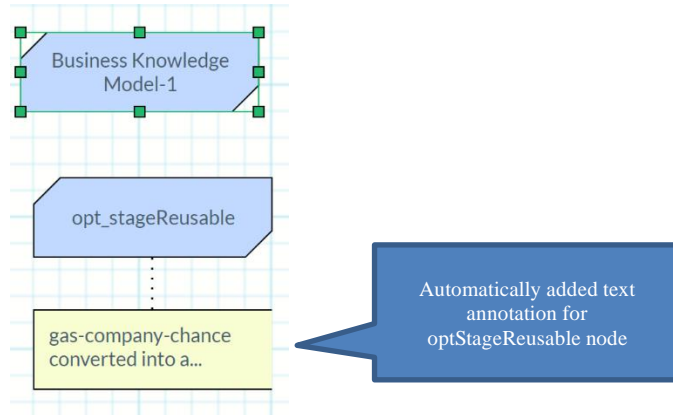
outputResults –
Exposes results
available for input
to subsequent
nodes.

}

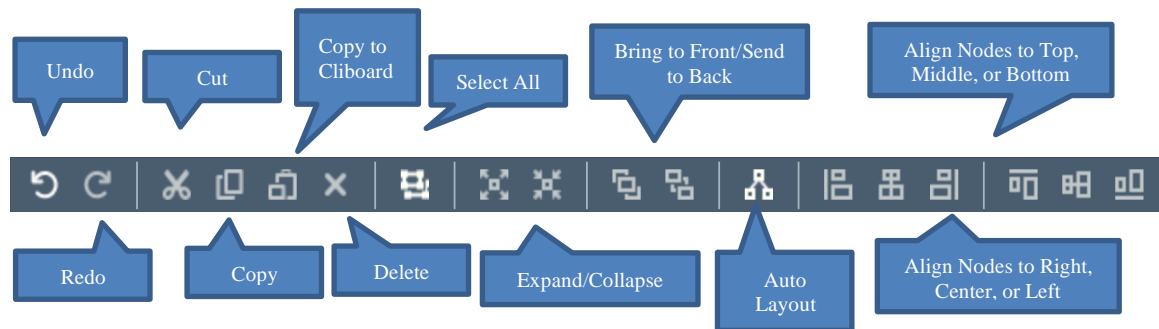
Note that the outputResults section lists the results that may be used as inputs to subsequent nodes. If an output is not listed here, then subsequent nodes will not have access to this result.

We can name our decision flow by deleting "Unnamed-1" and typing "OptSimReusable". Notice that under optStageReusable we see our output results, the x array (which holds our initial variable values and final variable values) and totalObjective (which holds our final objective value).

Toggle back to the RASON model editor and drag a 2nd Business Knowledge onto the grid.



Recall that you can use the icons at the top of the editor to undo changes, copy, paste, delete and align nodes.



Under Properties (top right) enter "simStageReusable" and click <enter> to populate the pane.

Properties

Invoke Model

sim_stageReusable

Name

sim_stageReusable

Type

Simulation

Recency

Comment

This interface has been generated by Psi for a

Inputs

finalVarValues

Outputs

cash.mean

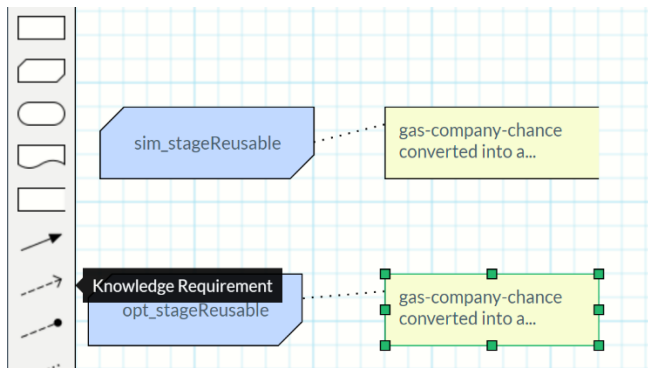
Model Type property – Tied to modelType property within reusable model.

Add a comment here.

Inputs to simStageReusable – Corresponds to inputParameters in RASON model script.

Lists output results for reusable model – Corresponds to outputResults in RASON model script.

To connect the two stages, click the Knowledge Requirement arrow (the dotted arrow) and draw the arrow from optStageReusable to simStageReusable.



It's important that the flow travels from the optimization model to the simulation model as the stochastic optimization must be performed before the simulation in our example. In the properties pane, select finalVarValues as the Input Variable and x.finalValue for Invocable Output Value. Recall that finalVarValues is an "inputParameter" to simStageReusable and x.finalValue is an "outputResult" from optStageReusable.

Properties		
Business Knowledge Model Name	sim_stageReusable	2 nd stage invoked model name
Input Variable	finalVarValues	Input variable for invoked model.
Input Column		Used for data science models.
Invocable Name	opt_stageReusable	Output from optStageReusable becomes input for simStageReusable .
Invocable Output Value	x.finalValue	Output variable from previous stage.

Toggle back to the RASON model editor. Notice that a 2nd stage has been added to the decision flow, **simStageReusable**. This stage accepts inputs (via **inputParameters**) and produces outputs (via **OutputResults**).

Notice under **simStageReusable** – **inputParameters** the "value" property for "finalVarValues" is "**optStageReusable.x.finalValue**". This was set automatically to complete the connection between the two stages.

```
{
  "flow": "OptSimReusable",
  "flowDescription": "opt-sim decision flow invoking reusable models",
  "optStageReusable": {
    "comment": "This interface has been generated by Psi for a RASON model",
    "invokeModel": "optStageReusable",
    "modelType": "optimization",
    "outputResults": {
      "x": {
        "evaluations": ["initial_Value", "finalValue", "dualValue"],
        "type": "array/number",
        "comment": "decision variable block"
      },
      "totalObjective": {
        "evaluations": ["finalValue"],
        "type": "number",
        "comment": "objective function"
      }
    },
    "modelDescription": "gas-company-chance converted into a reusable model",
  },
  "simStageReusable": {
    "comment": "This interface has been generated by Psi for a RASON model",
    "invokeModel": "simStageReusable",
    "modelType": "simulation",
    "inputParameters": {
      "finalVarValues": {
        "value": "optStageReusable.x.finalValue",
        "type": "array",
        "comment": "pass the final variable values from optStage"
      }
    },
    "outputResults": {

```

```

    "cash": {
      "evaluations": ["mean"],
      "type": "number",
      "comment": "uncertain function block"
    },
  },
  "modelDescription": "gas-company-chance converted into a reusable model"
}
}

```

Solving the Decision Flow

Now we are ready to POST and SOLVE the workflow.

- Click POST rason.net/api/model to post the model to the RASON Server
- Click Select Query Parameter on the bottom of the Properties pane to pass the two query parameters "keep-intermediate-results = true and data-storage=JSON.

Query Parameters

Select REST query parameters and set values to use when solving the optSimWorkflow 2020-05-12-21-19-40-697018 model.

API Parameters	
<input type="checkbox"/> stage	Single stage name to solve
<input checked="" type="checkbox"/> keep-intermediate-results	true
<input type="checkbox"/> simplify-final-results	false
<input checked="" type="checkbox"/> data-storage	JSON
<input type="checkbox"/> response-format	Workflow
<input type="checkbox"/> schedule	Schedule for solving model

- Click POST rason.net/api/model/{nameorid}/solve to solve the workflow. Recall that this is the ONLY endpoint that solves decision flows.
- To mark the version as a "champion", select the entry under Versions and click the Mark Champion button.

Models

- opt_stageReusable
- OptSimReusable
 - Versions
 - ★ 2020-05-12-21-19-40-697018
 - Results
 - 2020-05-12-21-19-47-011536
- optSimWorkflow
- sim_stageReusable

Properties

Name: OptSimReusable

Version: 2020-05-12-21-19-40-697018

Data Files: No data files specified.
 [Choose Files...](#)

Query Parameters: No query parameters specified.
 [Select Query Parameters...](#)

Select the entry under OptSimReusable – Results – to open the decision flow results.

Table		opt_stageReusable_x			
ID	:	initialValue	:	finalValue	:
0	:	0	:	1	:
1	:	0	:	0	:
2	:	0	:	1	:
3	:	0	:	1	:
4	:	0	:	1	:
5	:	0	:	0	:
6	:	0	:	1	:
7	:	0	:	0	:

Table		opt_stageReusable_totalObjective	
ID	:	finalValue	:
0	:	1390897.9539955165	:

Click the "Show JSON results" icon in the top right to view the solution in JSON form.

```

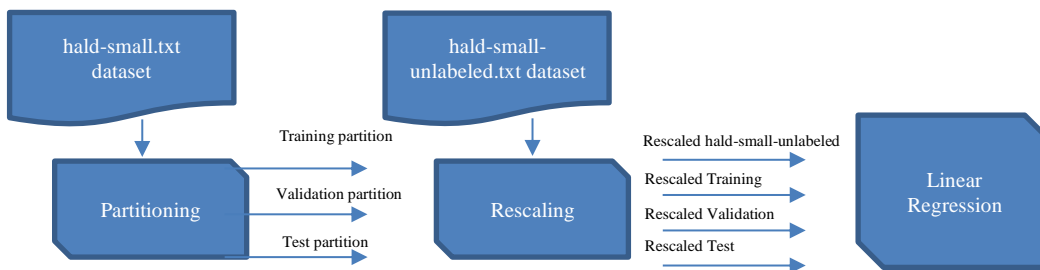
1 {
2   "status": {
3     "id": "2590+opt_stageReusable+2020-11-13-20-19-32-830145",
4     "code": 14,
5     "codeText": "Integer solution found within tolerance.",
6     "solveTime": 214
7   },
8   "results": {
9     "totalObjective": ["finalValue"],
10    "x": ["finalValue"]
11  },
12  "engine": "Gurobi Solver",
13  "x": {
14    "objectType": "dataFrame",
15    "name": "x",
16    "order": "col",
17    "colNames": ["finalValue"],
18    "colTypes": ["double"],
19    "indexCols": null,
20    "data": [
21      [1, 0, 1, 1, 1, 0, 1, 0]
22    ]
23  },
24  "totalObjective": {
25    "objectType": "dataFrame",
26    "name": "totalObjective",
27    "order": "col",
28    "colNames": ["finalValue"],
29    "colTypes": ["double"],
30    "indexCols": null,
31    "data": [
32      [1390897.9539955165]
33    ]
34  }
35 }

```

To open the completed decision flow and reusable models, click the RASON Examples folder on the Editor tab (at www.RASON.com) and then Decision Flows – "Optimization Simulation Decision Flow", "Optimization Stage (1st stage)" and "Simulation Stage (2nd stage)".

Data Science Decision Flow Example

This example walks users through the creation of a data science decision flow. To view the completed decision flow, click the RASON Examples folder on the Editor ribbon and select "Data Science Decision Flow". This decision flow will contain three nodes, a partitioning node, a rescaling node, and an MLR regression node.



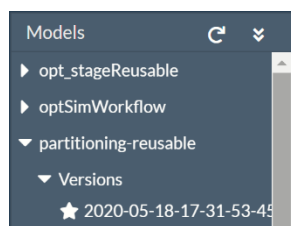
- The **partitioning** node will import the hald-small.txt datasource and partition the dataset into training, validation and test partitions.
- As input, the rescaling node will use the training, validation and test partitions created in the partitioning node, along with a dataset containing unlabeled (new) data, hald-small-unlabeled.txt. This node will rescale all three partitions and the new data.
- As input, the mlrModel node will accept the rescaled training, validation and test partitions along with the rescaled unlabeled dataset, will fit a model to the training partition and will then use this fitted model to score the three partitions and unlabeled data and return various scoring statistics.

Each stage, or node, will invoke a reusable model.

Reusable Partitioning Model

Open "Partitioning Stage (1st stage) for use with Data Science DF" from the RASON Examples folder under Decision Flows. You'll notice that a re-usable model looks a whole lot like a standalone data science model, as described in the chapter, Defining a Data Science model, and since this reusable model accepts raw data as input, rather than another reusable model, this reusable model looks exactly like the standalone partitioning model described in the section, Partitioning Example. See this section for a complete discussion on this example.

POST the model to the RASON Server by clicking POST rason.net/api/model. The partitioning-reusable model appears in the Models pane under "partitioning-reusable". Select this version and click "Mark Champion" to designate this version of the model as the "champion". Now any API call referring to this named model will use this version.



The first 3 lines of code are made up of the modelName, modelType and modelDescription. Recall the modelName must be a unique name, modelType transmits the model type as "datamining" (recall this property can be set to "optimization", "simulation", "datamining" or "calculation") and modelDescription just gives a quick description of the model. Note that modelName, modelType and modelDescription are all RASON key words.

```

1{
2  "modelName": "partitioning-reusable",
3  "modelType": "datamining",

```



```

4  "modelDescription": "transformation: partitioning, independent on
input parameters",

```

The next line of code (line 5) is the start of the datasources section, which is where the external data source is defined. In this section, the contents of hald-small.txt is bound to the datasource, dataToPartitionSource. Notice that the datasource type is CSV and the file name, passed to the connection property, is "hald-small.txt". Currently the RASON modeling language supports nine different data types: "excel" (Microsoft Excel), "access" or "msaccess" (Microsoft Access), "odbc" (ODBC database), "OData" (OData endpoint), "mssql" (Microsoft SQL), "oracle" (Oracle database), CSV (Comma Separated Value), "json" (JSON file), or "xml" (XML file). Note that line 6 ("dataToPartitionSource") is a user defined name while the rest of the properties (type and connection) are RASON key words that can not be changed.

```

5  "datasources": {
6    "dataToPartitionSource": {
7      "type": "csv",
8      "connection": "hald-small.txt"
9    }
10 },

```

Line 11 starts the datasets section, which is where the external dataset (hald-small.txt) is tied to a RASON dataset. In this example, the data source dataToPartitionSource (referring to hald-small.txt) is bound to the dataset, dataToPartition. This STEP is mandatory, the datasource must first be converted to a dataset in order to be manipulated by the Transformer in "actions". Note that line 12 ("dataToPartition") is a user defined name while "binding" is a RASON key word.

```

11 "datasets": {
12   "dataToPartition": {
13     "binding": "dataToPartitionSource"
14   }
15 },

```

Line 16 starts the transformer section. A data science model may either contain an estimator or a transformer section but not both. A "transformer" applies to estimators that do not fit a model but rather transform data, such as partitioning. Line 17 is a user defined name given to the transformer, myPartitioner. The next line of code, line 18, sets the transformer type to transformation. The type must be one of the following: "affinityAnalysis", "bigData", "featureSelection" or "transformation". Line 19 sets the algorithm to "partitioning". Since type=transformation, algorithm may be: "sampling", "stratifiedSampling", "partitioning", "oversamplePartitioning" or "categoryReduction". For more information on estimators/transformers and all essential RASON Data Science model sections see the chapter, **Defining a Data Science Model**. Note that line 17 ("myPartitioner") is a user defined name while the rest of the properties (type, algorithm, parameter settings and seed) are RASON key words.

```

16 "transformer": {
17   "myPartitioner": {
18     "type": "transformation",
19     "algorithm": "partitioning",
20     "parameters": {
21       "partitionMethod": "RANDOM",
22       "ratios": [
23         [ "Training", 0.5 ],
24         [ "Validation", 0.3 ],
25         [ "Testing", 0.2 ]
26       ],
27       "seed": 123
28     }
29   }
30 },

```

The reusable model, partitioning-reusable, partitions the hald-small.txt dataset into three partitions (training, validation and test partitions).



Lines 20-28 set options for the partitioning algorithm. See the RASON Reference Guide to see all options related to partitioning under *Algorithm Parameters: Partitioning for Transformation*.

Line 31 is the start of the "actions" section which is where the estimator or transformer is applied to the data. Three actions are performed by applying the transformer "myPartitioner" to the dataToPartition dataset (hald-small.txt) to create three partitions, Training Validation and Testing, using the options specified for the partitioning algorithm: 50% of records assigned to the training partition, 30% of records assigned to the validation partition and 20% of records assigned to the test partition. Note that lines 32 ("trainingPartition"), 40 ("validationPartition") and 48 ("testPartition") are user defined names while the rest of the properties (data, action, parameters, etc) are RASON key words that can not be changed.

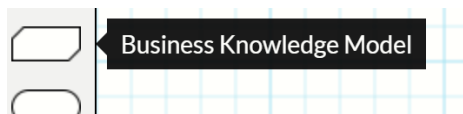
```

31  "actions": {
32    "trainingPartition": {
33      "data": "dataToPartition",
34      "action": "transform",
35      "parameters": {
36        "partition": "Training"
37      },
38      "evaluations": [ "transformation" ]
39    },
40    "validationPartition": {
41      "data": "dataToPartition",
42      "action": "transform",
43      "parameters": {
44        "partition": "Validation"
45      },
46      "evaluations": [ "transformation" ]
47    },
48    "testPartition": {
49      "data": "dataToPartition",
50      "action": "transform",
51      "parameters": {
52        "partition": "Testing"
53      },
54      "evaluations": [ "transformation" ]
55    }
56  }
57}

```

Create a new blank model document by clicking the  icon on the Editor ribbon, toggle to decision flow editor by clicking the "Show decision flow editor" icon  at the top right of the Flow Editor.

Drag a Business Knowledge Model to the grid.



Enter "partitioning-reusable" under Invoke Model and press <enter>. The Properties pane will populate as shown below.

Properties

Invoke Model
partitioning-reusable

Name
partitioning-reusable

Type
Data Mining

Recency

Comment
This interface has been generated by Psi for a

Inputs
No inputs defined.

Outputs
trainingPartition.transformation
validationPartition.transformation
testPartition.transformation

Name of invoked reusable model.

Model name given by modelName property in RASON model script.

This property informs the RASON server of how recently the reusable model was run in order to determine if the results can be considered "current". The object "time-since-last run" must be in ISO 8601 time duration format (i.e. "PT12H" for 12 hours or "P1W" for 1 week). The RASON Server will consult past runs to determine the maximum time a solve usually takes to complete (max-time), and will arrange to run the model at intervals of time-since-last-run – max-time. If the "Recency" is omitted, the default value is infinity; causing the model to run one time.

Reusable model type (optimization, simulation, datamining or calculation), given by ModelType property in the RASON model script.

A comment may be entered here.

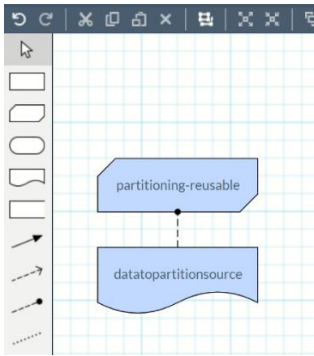
Model inputs given in inputParameters section. Since this is the first stage in the decision flow, no inputs appear here.

Results available to subsequent nodes in the decision flow.

Scheduling

To enter a Recency value of 10 mins, enter "PT10M". With this Recency setting, when the decision flow is executed, the RASON Server will check when this stage has last been executed. If this stage completed less than 10 minutes ago, then this stage will be skipped in the decision flow execution. Otherwise, it will be executed.

Notice that a Knowledge Source (datatopartitionsource) has been added to the grid with a Knowledge Requirement connection. Recall that datatopartitionsource is the source data for the Business Knowledge Model, partitioning-reusable.json.



Congratulations! You've just created the first node in the decision flow. Toggle back to the Decision Flow Editor to view the decision flow code.

```

1 {
2   "flowName": "mlrDF",
3   "partitioning-reusable": {
4     "comment": "This interface has been generated by Psi for a RASON model",
5     "invokeModel": "partitioning-reusable",
6     "modelType": "datamining",
7     "datasources": {
8       "dataToPartitionSource": {
9         "type": "csv",
10        "connection": "hald-small.txt",
11        "direction": "import"
12      }
13    },
14    "outputResults": {
15      "trainingPartition": {
16        "evaluations": ["transformation"],
17        "type": "dataset",
18        "comment": "datamining evaluation"
19      },
20      "validationPartition": {
21        "evaluations": ["transformation"],
22        "type": "dataset",
23        "comment": "datamining evaluation"
24      },
25      "testPartition": {
26        "evaluations": ["transformation"],
27        "type": "dataset",
28        "comment": "datamining evaluation"
29      }
30    },
31    "modelRecency": "PT10M",
32    "modelDescription": "transformation: partitioning, independent on input parameters"
33  },

```

These are the results from the Partitioning Node that are available to subsequent nodes in the decision flow.

Notice that the partitioning-reusable node invokes (or refers to) the reusable model, "partitioning-reusable". Anything changed within this RASON Decision Flow script will be reflected in the Decision Flow node and vice versa.

Properties	Corresponding RASON Script Properties
Invoke Model partitioning-reusable	"invokeModel"
Name partitioning-reusable	"modelName"
Type Data Mining	"modelType"
Recency PT10M	"modelRecency"
Comment This interface has been generated by Psi for a	"comment"
Inputs No inputs defined.	"inputParameters"
Outputs trainingPartition.transformation validationPartition.transformation testPartition.transformation	"outputResults"

The section "outputResults" (located in lines 16-32) identifies the output from this node, or in other words the contents of outputResults indicate the results that are available to subsequent nodes in the decision flow. The results from the partitioning-reusable node are three partitions: trainingPartition.transformation, validationPartition.transformation and testPartition.transformation. Each of these partitions is of type "dataset".

Each node in a partition must contain an outputResults section. If receiving input data, then a node will include both an inputParameters section and an outputResults section.

Toggle back to the Decision Flow Editor.

Reusable Rescaling Model

Now lets move on to the next stage in example, rescaling. Open "Rescaling Stage (2nd stage) for use with Data Science DF" from the RASON Examples folder under Decision Flows – Reusable Models: What's the path? POST the model to the RASON Server by clicking POST rason.net/api/model. The rescaling-reusable model appears in the Models pane under "rescaling-reusable". Again, mark this version as the "Champion".

```

Models
├── opt_stageReusable
├── optSimWorkflow
├── partitioning-reusable
│   └── Versions
│       └── 2020-05-18-17-31-53-45
├── rescaling-reusable
│   └── Versions
│       └── 2020-05-18-19-22-37-38

```

In this node, the reusable model, rescaling-reusable, accepts the inputParameters, which are the three partitions created in the partitioning node, and returns 5 items: 1. A fitted rescaling model, 2. A rescaled training

partition, 3. A rescaled validation partition, 4. A rescaled test partition and 5. A rescaled new data (the hald-small-unlabeled.txt dataset).

```
1{
2  "modelName": "rescaling-reusable",
3  "modelType": "datamining",
4  "modelDescription": "transformation: rescaling",
5  "inputParameters": {
6    "trainData": {
7      "type": "dataset",
8      "value": null,
9      "comment": ""
10   },
11   "validData": {
12     "type": "dataset",
13     "value": null,
14     "comment": ""
15   },
16   "testData": {
17     "type": "dataset",
18     "value": null,
19     "comment": ""
20   }
21 },
22 "datasources": {
23   "newDataSource": {
24     "type": "csv",
25     "connection": "hald-small-unlabeled.txt"
26   }
27 },
28 "datasets": {
29   "newData": {
30     "binding": "newDataSource"
31   }
32 },
33 "estimator": {
34   "myRescaler": {
35     "type": "transformation",
36     "algorithm": "rescaling",
37     "parameters": {
38       "technique": "UNIT_NORMALIZATION",
39       "normType": "L1",
40       "excludedCols": [ "Y" ]
41     }
42   }
43 },
```

The reusable model rescaling-reusable, accepts the inputParameters, which are the three partitions created in the partitioning node, and returns 5 items: 1. A fitted rescaling model, 2. A rescaled training partition, 3. A rescaled validation partition, 4. A rescaled test partition and 5. Rescaled new data (the hald-small-unlabeled.txt dataset).

The first 3 lines of code are made up of the modelName, modelType and modelDescription. Recall the modelName must be a unique name, modelType transmits the model type as "datamining" (recall this property can be set to "optimization", "simulation", "datamining" or "calculation") and modelDescription just gives a quick description of the model.

The next line of code (line 5) is the start of the inputParameters section, which is where the input data (output from a preceding node) is introduced. In this example, the three partitions created in the earlier "partitioning" node are passed, the training partition under "trainData", the validation partition under "validData" and the test partition under "testData". Notice that under each, there are three properties, "type", "value" and "comment". Only "value": null is mandatory, "type" and "comment" are optional. The "type" property may be used to inform the type of data being passed for the given input parameter while the "comment" property optionally

allows for more details, if desired. Note that the name(s) for inputParameters (trainData, validData and testData) are user defined.

```
6 "trainData": {
7   "type": "dataset",
8   "value": null,
9   "comment": ""
10 },
```

The next section (line 22), within rescaling-reusable, is "datasources". Within this section, one datasource is created, newDataSource (user defined name). This datasource will be used as "new data". The datasource type is CSV and the file name, passed to the connection property, is "hald-small-unlabeled.txt".

Line 28 starts the datasets section, which is where the external dataset (hald-small-unlabeled.txt) is tied to a RASON dataset. In this example, the data source newDataSource is bound to a new dataset, newData (user defined name). As discussed earlier, this step is mandatory, the datasource must first be converted to a RASON dataset in order to be manipulated by the Transformer in "actions".

Line 33 begins the "estimator" section. Recall that a data science model may either contain an estimator *or* a transformer section but not both. An "estimator" fits a model to a dataset. Line 34 is a user defined name given to the estimator, myRescaler. The next line of code, line 35, sets the estimator type to transformation. This type may be one of the following: "classification", "regression", "clustering", "textMining", "transformation" or "timeSeries". Line 36 sets the algorithm to "rescaling". Since type=transformation, algorithm may be: "oneHotEncoding", "imputation", "rescaling", "principalComponentAnalysis", "binning", "factorization", "canonicalVariateAnalysis". For more information on estimators/transformers see the chapter, Defining a Data Science Model.

Lines 37 - 43 set options for the rescaling algorithm. See the RASON Reference Guide to see all options related to rescaling under Algorithm Parameters: Rescaling.

```
44 "actions": {
45   "rescalerModel": {
46     "trainData": "trainData",
47     "estimator": "myRescaler",
48     "action": "fit",
49     "evaluations": [ "statistics" ]
50   },
51   "rescaledTrainData": {
52     "data": "trainData",
53     "fittedModel": "rescalerModel",
54     "action": "transform",
55     "evaluations": [ "transformation" ]
56   },
57   "rescaledValidData": {
58     "data": "validData",
59     "fittedModel": "rescalerModel",
60     "action": "transform",
61     "evaluations": [ "transformation" ]
62   },
63   "rescaledTestData": {
64     "data": "testData",
65     "fittedModel": "rescalerModel",
66     "action": "transform",
67     "evaluations": [ "transformation" ]
68   }
69   "rescaledNewData": {
70     "data": "newData",
71     "fittedModel": "rescalerModel",
72     "action": "transform",
```

```

73     "evaluations": [ "transformation" ]
74   }
75 }
76 }

```

Line 44 is the start of the "actions" section which is where the estimator or transformer is applied to the data. Five actions are performed.

1. rescalerModel – Fits the myRescaler model to the training data in "rescalerModel".
2. rescaledTrainData – Applies the fit rescaler model, rescalerModel, to the training partition.
3. rescaledValidData – Applies the fit rescaler model, rescalerModel, to the validation partition.
4. rescaleTestData – Applies the fit rescaler model, rescalerModel, to the test partition
5. rescaledNewData – Applies the fit rescaler model, rescalerModel, to the new data.

In the first action, rescalerModel (a user defined name), the estimator "myRescaler" is fit using the training partition, trainData. In return, several fit statistics will be returned in the results.

```

45   "rescalerModel": {
46     "trainData": "trainData",
47     "estimator": "myRescaler",
48     "action": "fit",
49     "evaluations": [ "statistics" ]
50   },

```

In the second through 4th actions (rescaledTrainData, rescaledValidData and rescaledTestData – all user defined names), the fitted model (fit in the previous rescalerModel action) is applied to the training, validation and test partitions, respectively. Each action transforms the respective data.

```

51   "rescaledTrainData": {
52     "data": "trainData",
53     "fittedModel": "rescalerModel",
54     "action": "transform",
55     "evaluations": [ "transformation" ]
56   },

```


In the fifth and last action, rescaledNewData (a user defined name), the fitted model (fit in the previous action, rescalerModel) is applied to the new dataset.

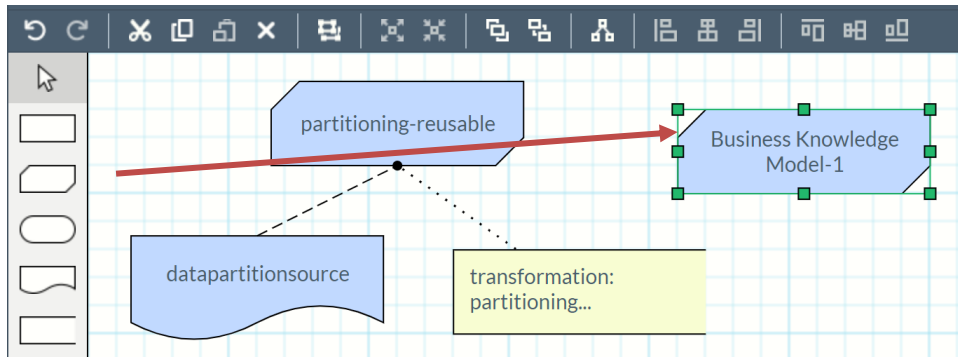
```

69   "rescaledNewData": {
70     "data": "newData",
71     "fittedModel": "rescalerModel",
72     "action": "transform",
73     "evaluations": [ "transformation" ]
74   },

```

Note that lines 45, 51, 57, 63 and 69 are user defined names while the rest of the properties (data, fittedModel, action, parameters, etc) are RASON key words that can not be changed.

Toggle back to the Decision Flow Editor, by clicking the "Show decision flow editor" icon  at the top right of the RASON model editor, and drag a 2nd Business Knowledge Model onto the grid.



Enter "rescaling-reusable" under Invoke Model and press <enter>. The Properties pane will populate as shown below.

Model name corresponds to "modelName" property in RASON Decision Flow script

If the reusable model has not been executed within this time interval, the stage will be executed. This text box corresponds to the "modelRecency" property in the RASON Decision Flow script.

Outputs from previous nodes, or stages, become Inputs to subsequent nodes. Corresponds to "inputParameters" in RASON Decision Flow script.

Properties

Invoke Model
rescaling-reusable

Name
rescaling-reusable

Type
Data Mining

Recency

Comment
This interface has been generated by Psi for a

Inputs

trainData
validData
testData

Outputs

rescalerModel.statistics
rescaledTrainData.transfor
mation
rescaledValidData.transfor
mation
rescaledTestData.transfor
mation
rescaledNewData.transfor
mation

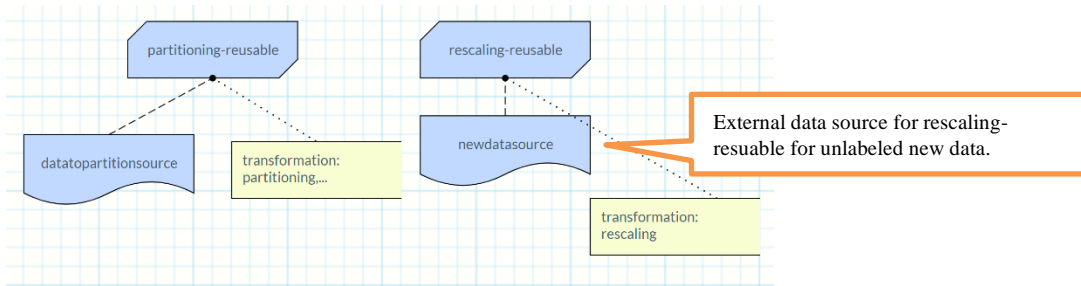
Name of invoked reusable model.

Reusable model type (optimization, simulation, datamining or calculation), corresponds to "modelType" property in the RASON Decision Flow script.

A comment may be entered here. This field corresponds to "comment" in the RASON Decision Flow script.

Results available to subsequent nodes in the decision flow. Corresponds to "outputResult"s in RASON Decision Flow script.

The rescaling-reusable node appears in the grid along with it's external data source "newdatasource".



Click the connection between rescaling-reusable and newdatasource. The Properties pane displays the Target Node's name (rescaling-reusable), the data source name (newdatasource) and the connection or filename (hald-small-unlabeled.txt).

Properties	
Target Name	rescaling-reusable
Source Name	newdatasource
Source Variable	hald-small-unlabeled.txt

Target Node

Data Source Name

Data Source Connection or File Name

Notice that the newly created rescaling-reusable node is not yet connected to the partitioning-rescaling node. To make this connection, click the Knowledge Requirement. (Recall that this connector supports connections between Business Knowledge Models (reusable models).)

Since there are three inputs to this node (training partition, validation partition and the test partition), three connections are required. For the first connection, click the "partitioning-reusable" node and draw a connection from this node to the rescaling-reusable node. In the Properties pane, select "traindata" for "Input Variable" and "trainingPartition.transformation" for "Invocable Output Value".

Properties	
Business Knowledge Model Name	rescaling-reusable
Input Variable	trainData
Input Column	
Invocable Name	partitioning-reusable
Invocable Output Value	trainingPartition.transformation

Toggle back to the RASON model editor and scroll down to the newly added rescaling-reusable stage. Notice that "value" for "traindata" within "inputParameters" has been changed from "null" to "partitioning-reusable.trainingPartition.transformation". This is a direction result of the connection that was just completed.

```

33 },
34 "rescaling-reusable": {
35   "comment": "This interface has been generated by Psi for a RASON model",
36   "invokeModel": "rescaling-reusable",
37   "modelType": "datamining",
38   "inputParameters": {
39     "trainData": {
40       "value": "partitioning-reusable.trainingPartition.transformation",
41       "type": "dataset",
42       "comment": ""
43     },
44     "validData": {
45       "value": "partitioning-reusable.validationPartition.transformation",
46       "type": "dataset",
47       "comment": ""
48     },
49     "testData": {
50       "value": "partitioning-reusable.testPartition.transformation",
51       "type": "dataset",
52       "comment": ""
53     }
54   },
55   "datasources": {
56     "newDataSource": {
57       "type": "csv",
58       "connection": "hald-small-unlabeled.txt",
59       "direction": "import"
60     }
61   },
62   "outputResults": {
63     "rescalerModel": {
64       "evaluations": ["statistics"],
65       "type": "dataset",
66       "comment": "datamining evaluation"
67     },
68     "rescaledTrainData": {

```

Toggle back to the RASON Decision Flow Editor to make the two remaining connections, validation partition connection and the test partition connection. For the validation partition connection, select "validdata" for "Input Variable" and "validationpartition.transformation" for "Invocable Output Value". For the test partition connection, select "testdata" for "Input Variable" and "testpartition.transformation" for "Invocable Output Value".

Validation Partition Connection

Properties	
Business Knowledge Model	Name
	rescaling-reusable
Input Variable	validData
Input Column	
Invocable Name	partitioning-reusable
Invocable Output Value	validationPartition.transfor

Test Partition Connection

Properties	
Business Knowledge Model	Name
	rescaling-reusable
Input Variable	testData
Input Column	
Invocable Name	partitioning-reusable
Invocable Output Value	testPartition.transformation

Toggle back to the Decision Flow Editor to view the decision flow code and scroll down to the rescaling-reusable node, input Parameters section. Notice that the value property for all three input parameters (traindata, validdata and testdata) have been changed from "null" to "partitioning-reusable.xxxpartition.transformation" where xxx is either "training", "validation" or "test".

```

34  "rescaling-reusable": {
35    "comment": "This interface has been generated by Psi for a RASON model",
36    "invokeModel": "rescaling-reusable",
37    "modelType": "datamining",
38    "inputParameters": {
39      "trainData": {
40        "value": "partitioning-reusable.trainingPartition.transformation",
41        "type": "dataset",
42        "comment": ""
43      },
44      "validData": {
45        "value": "partitioning-reusable.validationPartition.transformation",
46        "type": "dataset",
47        "comment": ""
48      },
49      "testData": {
50        "value": "partitioning-reusable.testPartition.transformation",
51        "type": "dataset",
52        "comment": ""
53      }
54    },
55    "datasources": {
56      "newDataSource": {
57        "type": "csv",
58        "connection": "hald-small-unlabeled.txt",
59        "direction": "import"
60      }
61    },

```

Scroll down to outputResults. This is the output that is available for input to subsequent nodes.

```

62  "outputResults": {
63    "rescalerModel": {
64      "evaluations": ["statistics"],
65      "type": "dataset",
66      "comment": "datamining evaluation"
67    },
68    "rescaledTrainData": {
69      "evaluations": ["transformation"],
70      "type": "dataset",
71      "comment": "datamining evaluation"
72    },
73    "rescaledValidData": {
74      "evaluations": ["transformation"],
75      "type": "dataset",
76      "comment": "datamining evaluation"
77    },
78    "rescaledTestData": {
79      "evaluations": ["transformation"],
80      "type": "dataset",
81      "comment": "datamining evaluation"
82    },
83    "rescaledNewData": {
84      "evaluations": ["transformation"],
85      "type": "dataset",
86      "comment": "datamining evaluation"
87    }
88  },
89  "modelDescription": "transformation: rescaling"
90 },

```

These are the results from the Rescaling Node that are available to subsequent nodes in the decision flow.

Congratulations you've just added a 2nd stage to your decision flow! Let's continue with the third and final stage.

mlrModel Reusable Model

In this node, the invoked model, mlr-reusable, accepts four inputParameters, which are the three rescaled partitions (created in the partitioning node and rescaled in the rescaling node) and the rescaled new data (created in the rescaling node); and returns 5 items: 1. "myModel" which fits the model to the training partition and returns the fit statistics, 2. "trainScore" which uses the fitted model to score the rescaled training partition, 3. "validScore" which uses the fitted model to score the rescaled validation partition, 4. "testScore" which uses the fitted model to score the rescaled test partition and 5. "newScore" which uses the fitted model to score the new data.

Open the mlr-reusable example by clicking the RASON Examples folder on the Editor tab ribbon and selecting "MLR Stage (3rd stage) for use with Data Science DF" from the list, then POST the model to the RASON Server using the REST API endpoint, POST rason.net/api/model.

```

1{
2  "modelName": "mlr-reusable",
3  "modelType": "datamining",
4  "modelDescription": "regression: linear model",
5  "inputParameters": {
6    "trainData": {
7      "type": "dataset",
8      "value": null,
9      "targetCol": "",
10     "comment": ""
11   },
12   "validData": {
13     "type": "dataset",
14     "value": null,
15     "targetCol": "",
16     "comment": ""
17   },
18   "testData": {
19     "type": "dataset",
20     "value": null,
21     "targetCol": "",
22     "comment": ""
23   },
24   "newData": {
25     "type": "dataset",
26     "value": null,
27     "comment": ""
28   }
29 },
30 "estimator": {
31   "mlrEstimator": {
32     "type": "regression",
33     "algorithm": "linearRegression",
34     "parameters": {
35       "fitIntercept": true
36     }
37   }
38 },
39 "actions": {
40   "myModel": {
41     "trainData": "trainData",
42     "estimator": "mlrEstimator",
43     "action": "fit",
44     "evaluations": [
45       "anova",
46       "influenceDiagnostics",
47       "detailedResiduals",
48       "coefficients",
49       "regressionSummary",
50       "detailedCoefficients",
51       "multicollinearityDiagnostics",
52       "varianceCovariance",
53       "predictorScreeningInfo",

```

```

54         "entranceTolerance"
55     ]
56 },
57 "trainScore": {
58     "data": "trainData",
59     "fittedModel": "myModel",
60     "action": "predict",
61     "evaluations": [
62         "prediction",
63         "residuals",
64         "intervals",
65         "sse",
66         "ss",
67         "sst",
68         "mse",
69         "rmse",
70         "mad",
71         "r2"
72     ]
73 },
74 "validScore": {
75     "data": "validData",
76     "fittedModel": "myModel",
77     "action": "predict",
78     "evaluations": [
79         "prediction",
80         "residuals",
81         "intervals",
82         "sse",
83         "ss",
84         "sst",
85         "mse",
86         "rmse",
87         "mad",
88         "r2"
89     ]
90 },
91 "testScore": {
92     "data": "testData",
93     "fittedModel": "myModel",
94     "action": "predict",
95     "evaluations": [
96         "prediction",
97         "residuals",
98         "intervals",
99         "sse",
100        "ss",
101        "sst",
102        "mse",
103        "rmse",
104        "mad",
105        "r2"
106    ]
107 },
108 "newScore": {
109     "data": "newData",
110     "fittedModel": "myModel",

```

```

111   "action": "predict",
112   "evaluations": [
113     "prediction"
114   ]
115 }
116 }
117 }

```

The first 3 lines of code are made up of the modelName, modelType and modelDescription. Recall the modelName must be a unique name, modelType transmits the model type as "datamining" and modelDescription simply gives a quick description of the model.

The next line of code (line 5) is the start of the inputParameters section. In this example, the three partitions created in the earlier "partitioning" node and rescaled in the rescaling node, along with the rescaled new data are passed to the mlr-reusable node; the training partition under "trainData", the validation partition under "validData", the test partition under "testData" and the new data under "newData".

Notice that under trainingData, validData, and testData, there are four properties, "type", "value", "targetCol" and "comment". Since the MLR algorithm requires an output variable, two properties are mandatory, "value" and "targetCol", while "type" and "comment" are optional. The "value": null property within the inputParameters section in a reusable model receives data from the matching value property within the inputParameters sections in the decision flow. Since newData contains the new data to be scored, this dataset does not contain an output variable, as this is the variable that will be scored. Note that the name(s) for inputParameters (trainData, validData, testData and newData) are user defined.

```

6  "trainData": {
7    "type": "dataset",
8    "value": null,
9    "targetCol": "",
10   "comment": ""
11 },

```

Line 30 begins the "estimator" section. Recall that a data science model may either contain an estimator *or* a transformer section but not both. This estimator will fit a model using the MLR algorithm. Note: "mlrEstimator" is a user-defined name.

```

30 "estimator": {
31   "mlrEstimator": {
32     "type": "regression",
33     "algorithm": "linearRegression",
34     "parameters": {
35       "fitIntercept": true
36     }
37   }
38 },

```

- "type": "regression" sets the estimator type to Regression. Type is a RASON keyword that may set to one of the following: "classification", "regression", "clustering", "textMining", "transformation" or "timeSeries".
- "algorithm": "linearRegression" sets the estimator to use the MLR algorithm to fit the model. "Algorithm" is a RASON keyword.
- "parameters" (RASON key word) sets any options for the selected algorithm. For a list of all options belonging to each algorithm, see the RASON DM chapter within the RASON Reference Guide.

The actions section begins on line 39. The action "myModel" fits the MLR model, mlrEstimator, to the rescaled training partition, trainData, requests various statistics to be returned.

```

39 "actions": {
40   "myModel": {

```

```

41     "trainData": "trainData",
42     "estimator": "mlrEstimadtor",
43     "action": "fit",
44     "evaluations": [
45         "anova",
46         "influenceDiagnostics",
47         "detailedResiduals",
48         "coefficients",
49         "regressionSummary",
50         "detailedCoefficients",
51         "multicollinearityDiagnostics",
52         "varianceCovariance",
53         "predictorScreeningInfo",
54         "entranceTolerance"
55     ]
56 },

```

- "trainData": "trainingData" sets the training partition to the inputParameters, trainData. Note "trainData" is a RASON key word while "trainingData" is user defined
- "estimator": "mlrEstimator" sets the MLR estimator created in "estimator". Note: "estimator" is a RASON key word.
- "action": "fit", sets the action to "fit" to "fit" the mlrEstimator to the training partition. Note: "action" and "fit" are both RASON key words.
- "evaluations" requests various statistics and results in the output. Note: All evaluations are RASON key words.

The next four actions, trainScore, validScore, testScore and newData, use the fitted model, myModel to score each of the three partitions, trainData, validData and testData, and the new data.

```

57 "trainScore": {
58     "data": "trainData",
59     "fittedModel": "myModel",
60     "action": "predict",
61     "evaluations": [
62         "prediction",
63         "residuals",
64         "intervals",
65         "sse",
66         "ss",
67         "sst",
68         "mse",
69         "rmse",
70         "mad",
71         "r2"
72     ]
73 },

```

- "data": "" sets the partition to be scored trainData, validData or testData. Note: "data" is a RASON key word.
- "fittedModel": "myModel" sets the fitted model, myModel. Note: "fittedModel" is a RASON key word.
- "action": "predict", sets the action to "predict" to score the dataset . Note: "action" and "predict" are RASON key words.
- "evaluations" requests various statistics and results in the output. Note: All evaluations are RASON key words.

Click back to the Unnamed-1 tab and toggle back to the Decision Flow Editor. Drag a third Business Knowledge Model (reusable model) node to the grid and enter "mlr-reusable" for Invoke Model on the Properties pane.

To connect the rescaling-reusable node to the mlr-reusable node, select the Knowledge Requirement connection on the left and draw 1 of 4 connections from the rescaling-reusable node to the mlr-reusable node. Select the appropriate Input Variable and Invocable Output Value for each connection as shown in the screenshots below.

- 1st Connection: trainData – rescaledTrainData.transformation
- 2nd Connection: validData – rescaledValidData.transformation
- 3rd Connection: testData – rescaledTestData.transformation
- 4th Connection: newData – rescaledNewData.transformation

Recall that mlr-reusable fits a model using the multiple linear regression classification method. This method requires an output variable, in this case the Y variable. Enter "Y" for Input Column to specify the target variable, Y, for each partition.

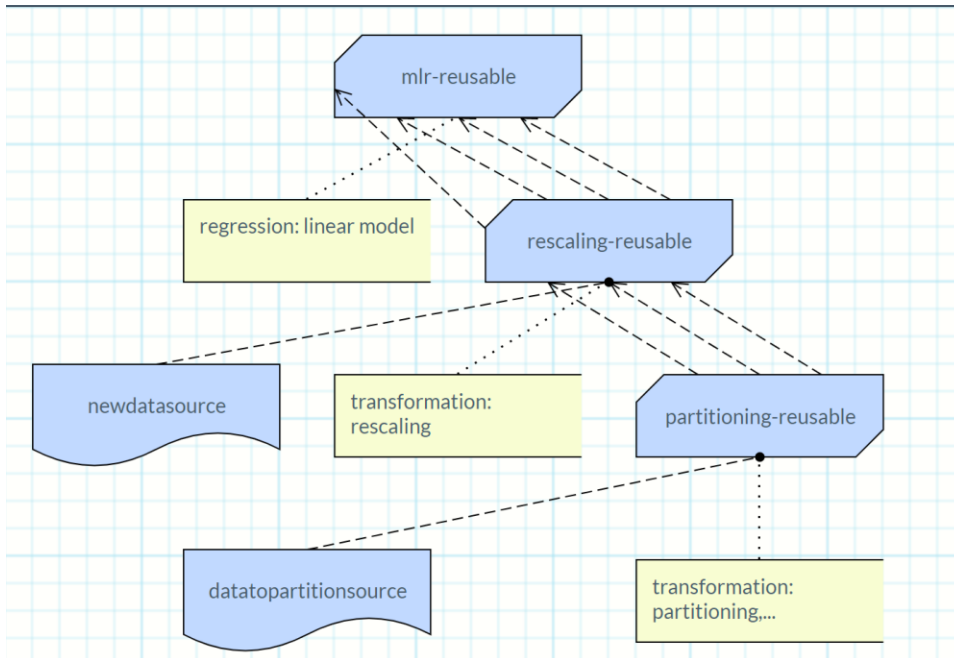
Training Partition Connection

Validation Partition Connection

Test Partition Connection

New (Unlabeled) Data Connection

Selects the output or target



Toggle back to the RASON model editor to view the decision flow script and scroll down to the third stage, "mlr-reusable". The mlrModel node begins by invoking the model "mlr-reusable" (line 92: "invokeModel": "mlr-reusable"), setting the model type to "datamining" (line 95: "modelType": "datamining") and using the "modelDescription" (line 96) property to add a short user-define string.

```

91  "mlr-reusable": {
92    "comment": "This interface has been generated by Psi for a RASON model",
93    "invokeModel": "mlr-reusable",
94    "modelType": "datamining",
95    "inputParameters": {
96      "trainData": {
97        "value": "rescaling-reusable.rescaledTrainData.transformation",
98        "type": "dataset",
99        "comment": "",
100       "targetCol": "Y"
101      },
102      "validData": {
103        "value": "rescaling-reusable.rescaledValidData.transformation",
104        "type": "dataset",
105        "comment": "",
106        "targetCol": "Y"
107      },
108      "testData": {
109        "value": "rescaling-reusable.rescaledTestData.transformation",
110        "type": "dataset",
111        "comment": "",
112        "targetCol": "Y"
113      },
114      "newData": {
115        "value": "rescaling-reusable.rescaledNewData.transformation",
116        "type": "dataset",
117        "comment": "",
118        "targetCol": ""
119      }
120    },
  
```

Since this node receives input data, the "inputParameters" section is present, beginning on line 97. In this example, the input data is made up of the three rescaled partitions, rescaling-reusable.rescaledTrainData.transformation, rescaling-reusable.rescaledValidData.transformation and rescaling-reusable.rescaledTestData.transformation, and the new data imported from within the rescaling node, rescaling-reusable.rescaledNewData.transformation.

Notice the new property "targetCol" that appears for each input argument. As discussed, this property passes the output variable to the MLR algorithm. In this example, the output variable is the Y variable. The "value"

property passes the data within each partition while "type" passes the data type as "dataset" ("type":"dataset" applicable to data science models only).

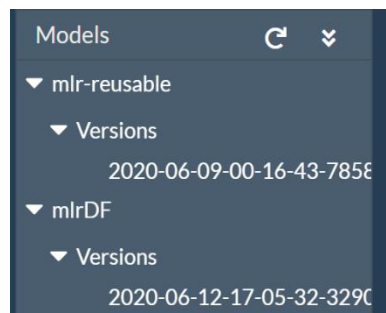
Since this node will be returning results (obviously), the outputResults section is present, beginning on line 123.

```
121  "outputResults": {
122    "myModel": {
123      "evaluations": ["anova", "influenceDiagnostics", "detailedResiduals", "coefficients", "regressionSummary", "detailedCoe
124      "type": "dataset",
125      "comment": "datamining evaluation"
126    },
127    "trainScore": {
128      "evaluations": ["prediction", "residuals", "intervals", "sse", "ss", "sst", "mse", "rmse", "mad", "r2"],
129      "type": "dataset",
130      "comment": "datamining evaluation"
131    },
132    "validScore": {
133      "evaluations": ["prediction", "residuals", "intervals", "sse", "ss", "sst", "mse", "rmse", "mad", "r2"],
134      "type": "dataset",
135      "comment": "datamining evaluation"
136    },
137    "testScore": {
138      "evaluations": ["prediction", "residuals", "intervals", "sse", "ss", "sst", "mse", "rmse", "mad", "r2"],
139      "type": "dataset",
140      "comment": "datamining evaluation"
141    },
142    "newScore": {
143      "evaluations": ["prediction"],
144      "type": "dataset",
145      "comment": "datamining evaluation"
146    }
147  },
148  "modelDescription": "regression: linear model"
149 }
150 }
```

This section returns the results from myModel (the fitted model), trainScore (the scored training partition), validScore (the scored validation partition), testScore (the scored test partition), and newScore (the scored new data dataset).

Scroll up to the top of the script, enter a proper name for the "flowName" property, say "mlrDF" and enter a description for "flowDescription", say "DF that partitions a dataset, rescales the partitions and runs mlr".

Then POST the decision flow using POST rason.net/api/model. The decision flow is posted to the RASON Server and appears on the model pane.



Note that if your RASON decision flow, or model, is unnamed, both would appear under Unnamed in the Models pane.

You'll receive similar output:

Executing ajax call...: POST <https://rason.net/api/model>

```
{
  "ModelId": "2590+mlrDF+2020-06-12-17-05-32-329097",
  "ModelName": "mlrDF",
  "ModelDescr": "DF that partitions a dataset into three partitions,
  rescales the partitions and runs MLR",
  "ModelFiles": [
    { "fileName": "hald-small.txt", "isOnServer": false },
    { "fileName": "hald-small-unlabeled.txt", "isOnServer": false }
  ], "RuntimeToken": "",
  "ModelType": "Origin",
  "ModelKind": "RASON",
```

Notice that the output includes a reference to the required input files.

```

    "IsChampion": false,
    "ParentModelId": null,
    "QueryString": "",
    "ModelContainer": null
  }

```

At this point the decision flow has been posted but we have not yet attached our data files. Notice that on the Properties pane, our data files are listed in white.

Click the "Choose Files" button on the Properties page and upload the two external input files hald-small.txt (defined in the partitioning-reusable node) and hald-small-unlabeled.txt (defined in the rescaling-reusable node). Then use the PUT `rason.net/api/model/id` to attach the files.

Note that at this point we could skip calling PUT and immediately call the POST `rason.net/api/model/{nameorid}/Solve` endpoint. However, in this instance, the files would NOT be attached to that version of the model, i.e. the next time you call `/Solve`, you would need to re-attach the input files. Whereas if you attach them when you POST the model (POST `rason.net/api/model`) or by using the PUT endpoint (PUT `rason.net/api/model/nameorid`), the files are attached to that version of the model.

Notice that after we attach the files using the PUT endpoint, the names of the Data Files turn red and also have been converted into links. If you click either of them, the actual input file will be downloaded locally.

Solve the decision flow using POST `rason.net/api/model/{nameorid}/solve`. Again, you'll receive similar output.

Executing asynchronous solve: POST `https://rason.net/api/model`

```

{
  "ModelId": "2590+mlrDF+2020-12-15-18-01-00-259506",
  "ModelName": "mlrDF",
  "ModelDescr": "",
  "ModelFiles": [{

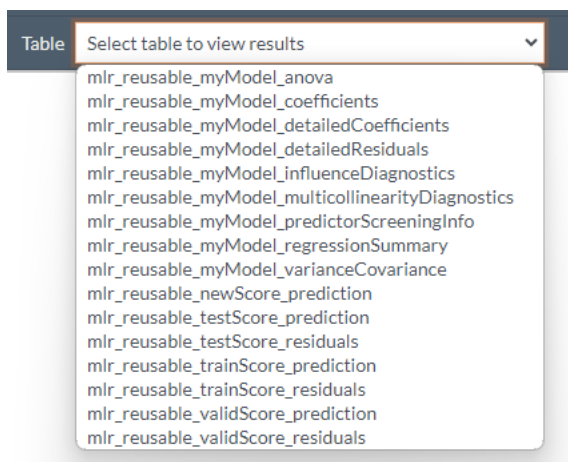
```

```

        "fileName": "hald-small.txt",
        "isOnServer": true
    }, {
        "fileName": "hald-small-unlabeled.txt",
        "isOnServer": true
    }
  ],
  "RuntimeToken": "",
  "ModelType": "Instance",
  "ModelKind": "RASON",
  "IsChampion": false,
  "ParentModelId": "2590+mlrDF+2020-12-15-18-00-54-591982",
  "QueryString": "keep-intermediate-results=true&data-storage=json",
  "ModelContainer": null
}

```

Click the down arrow next to Table to display the selected result.



Click the "Show JSON results" icon on the top left of the Editor window to display the results in the JSON response.

```

{
  "status": {
    "id": "2590+mlrDF+2020-12-15-18-01-00-259506",
    "code": 0,
    "codeText": "Success",
    "solveTime": 994
  },
  "results": {
    "mlr-reusable.myModel.anova": [],
    "mlr-reusable.myModel.anova": [],
    "mlr-reusable.myModel.coefficients": [],
    "mlr-reusable.myModel.detailedCoefficients": [],
    "mlr-reusable.myModel.detailedResiduals": [],
    ...
  }
}

```

To obtain results from the two earlier stages, partitioning-reusable and rescaling-reusable, use the URL parameters `keep-intermediate-results = true` and `data-storage=JSON`.

By default, when the POST `ration.net/api/model/{nameorid}/solve` endpoint is used to solve a workflow, RASON will compute, store and report results for terminal stages only (in addition to a status for the overall

workflow). Although RASON computes the results that are required to solve terminal stages, these results are not stored or reported *unless two* URL parameters are passed to the REST API endpoint POST `rason.net/api/model/{nameorid}/solve`: `keep-intermediate-results=true` and `data-storage=JSON`. *See the important note on using the ODATA endpoint, GET `rason.net/odata/result` below.*

- `keep-intermediate-results=true` determines whether the RASON server stores the results of the *intermediate* stages in the decision flow, returns them with JSON response and makes them available for REST or OData querying later. The default setting is False.
- `data-storage=JSON` controls whether the SQLite DATABASE or JSON file storage is used to store the results of *intermediate* stages. If the default, `data-storage=database` is used, intermediate results can still be stored and examined using the REST API ODATA endpoint GET `rason.net/odata/result`, but the results will *not* appear in the JSON response as serialized data frames.

When both URL parameters are submitted together, `data-storage=JSON` and `keep-intermediate-results=true`, JSON files are used for solving and storing, responses and querying. The results of intermediate and terminal stages are reported in the JSON response as well as being available for REST/OData querying.

To supply both parameters to the RASON graphical editor, click **Select Query Parameters** at the bottom of the Properties pane, select **keep-intermediate-results** and set the parameter to **True**, then select **data-storage** and set the parameter to **JSON**. Afterwards, click **Save** to close the dialog and set the parameters.

Query Parameters

Select REST query parameters and set values to use when solving the `optSimWorkflow 2020-06-09-00-16-43-785E` model.

API Parameters

<input type="checkbox"/> stage	Single stage name to solve
<input checked="" type="checkbox"/> keep-intermediate-results	true
<input type="checkbox"/> simplify-final-results	false
<input checked="" type="checkbox"/> data-storage	JSON
<input type="checkbox"/> response-format	Workflow
<input type="checkbox"/> schedule	Schedule for solving model

Model Data Parameters

Then use PUT `rason.net/api/model/{nameorid}` to update the model on the RASON server (make sure to re-upload the two data files first), check the status using GET `rason.net/api/model/{nameorid}/status` and obtain the results using GET `rason.net/api/model/{nameorid}/result`. Notice the additional information within the Table Results drop down menu pertaining to "partitioning_reusable" and "rescaling_reusable".



Click Show JSON results to view the results in JSON.

```
{
  "status": {
    ...
  },
  "results": {
    "mlr-reusable.mymodel.anova": [],
    ...
    "partitioning-reusable.testPartition.transformation": [],
    "partitioning-reusable.trainingPartition.transformation": [],
    "partitioning-reusable.validationPartition.transformation": [],
    "rescaling-reusable.rescaledNewData.transformation": [],
    "rescaling-reusable.rescaledTestData.transformation": [],
    "rescaling-reusable.rescaledTrainData.transformation": [],
    "rescaling-reusable.rescaledValidData.transformation": [],
    "rescaling-reusable.rescalerModel.statistics": []
  },
  "partitioning-reusable": {
    "status": {
      "id": "2590+partitioning-reusable+2020-05-21-01-12-20-155841",
      "code": 0,
      "codeText": "Success",
      "solveTime": 4
    },
    "trainingPartition": {
      ...
    },
    "validationPartition": {
      ...
    },
    "testPartition": {
```

```

    ...
  },
  "rescaling-reusable": {
    "status": {
      "id": "2590+rescaling-reusable+2020-05-21-01-12-20-228350",
      "code": 0,
      "codeText": "Success",
      "solveTime": 76
    },
    "rescalerModel": {
      ...
    }
    "rescaledTrainData": {
      ...
    },
    "rescaledValidData": {
      ...
    }
  }
}

```

Another option is to pass `keep-intermediate-results = true` as a URL parameter to the REST API endpoint `POST rason.net/api/model/{nameorid}/solve` (POST `rason.net/api/model/{nameorid}/solve?keep-intermediate-results=true`) and access those results using the OData endpoint `GET rason.net/ODATA/result`. For more information on this endpoint, see the section *Using the OData Endpoints* within the **Using the REST API** chapter that appears later in this guide.

The REST API Endpoint `GET rason.net/ODATA/result` requires two headers. The first is the Authorization header and the second is the resource ID. When both headers are passed to this endpoint, all results for the given model will be returned.

For example, the REST API Endpoint `GET https://rason.net/odata/result/` returns all available results in OData format. To drill down, you can ask for specific results using `https://rason.net/odata/result/<odata result>`. For example to obtain the OData results for the MLR Coefficients table, use:
https://rason.net/odata/result/mlr_reusable_mymodel_coefficients

Note: Substitute an underscore ("_") for a period (".") when getting the results using `GET Odata/result/`; aka `mlr-reusable.mymodel.anova` becomes `mlr-reusable_mymodel_anova`, `mlr-reusable.mymodel.anova` becomes `mlr-reusable_mymodel_anova`, etc.

```

{
  "@odata.context": "https://rason.net/odata/$metadata#Result(Data())/Sensitivity",
  "Name": "mlr_reusable_mymodel_coefficients",
  "Data": [
    {
      "_ID": 0,
      "_Name": "Intercept",
      "Estimate": -23.487082869309749
    },
    {
      "_ID": 1,
      "_Name": "X1",
      "Estimate": 149.01166572428048
    },
    {
      "_ID": 2,
      "_Name": "X2",
      "Estimate": 415.960477398573
    }
  ]
}

```



```

    },
    {
      "_ID": 3,
      "_Name": "X3",
      "Estimate": 121.07640892657103
    },
    {
      "_ID": 4,
      "_Name": "X4",
      "Estimate": 151.68899484535496
    },
    {
      "_ID": 5,
      "_Name": "Weights",
      "Estimate": -7.9279668096112808
    }
  ]
}

```

Invoking an Excel Model in a Decision Flow

It's possible to invoke an Excel model as a stage in a reusable model. In the example decision flow below, the Excel model, Product-Mix-Reusable.xlsx, which contains a simple optimization model, is invoked during the first stage, optStage. Notice that the Excel model is invoked using

"invokeModel": "product-mix-reusable:product-mix-reusable.xlsx!Product Mix Example 2",

which must be of the form "<modelname>:<filename>!<worksheetname>".

```

{
  "flowName": "optDMWorkflow",
  "optStage": {
    "invokeModel": "product-mix-reusable:product-mix-reusable.xlsx!Product Mix Example 2",
    "modelType": "optimization",
    "outputResults": {
      "Number_to_Build": {
        "evaluations": [
          "finalValue"
        ],
        "type": "array",
        "comment": "vector of dec vars"
      },
      "Total_profit": {
        "evaluations": [
          "finalValue"
        ],
        "type": "number",
        "comment": "scalar objective"
      }
    }
  },
  "rescalingStage": {
    "invokeModel": "rescaling-reusable",
    "modelType": "datamining",
    "inputParameters": {
      "myData": {
        "type": "dataset",
        "value": "optStage.Number_to_Build.finalValue",
        "comment": "an array/dataset of scoring data"
      }
    }
  },
  "outputResults": {
    "rescalerModel": {
      "type": "dataset",

```

```

    "evaluations": [
      "statistics"
    ],
  },
  "rescaledData": {
    "type": "dataset",
    "evaluations": [
      "transformation"
    ]
  }
}
}
}

```

To create the decision flow within the Editor tab you must first have posted the Excel model to the RASON Server either by using Create App within Analytic Solver or by using the API endpoint, POST RASON.com/api/model.

For more information on deploying your model from Analytic Solver, see the Analytic Solver User Guide. For more information on posting an Excel model to the RASON Server see the API Endpoint description for POST RASON.com/api/model that appears later in this guide.

If model is saved within a OneDrive or SharePoint account, use:

invokeModel: "Name=<NamedExcelConnection!Worksheet>"

where *NamedExcelConnection* is the name given to the Data Connection in the user's MyAccount. For more information on Data Connections, see the previous chapter.

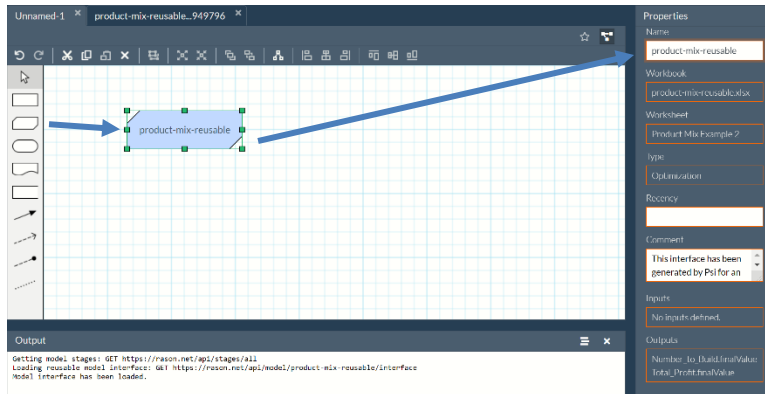
Once the Excel model is deployed to the RASON Server, the model will appear on the Models pane on the left of the Editor tab under the name given. (In this case, product-mix-reusable.)

The screenshot shows the RASON Editor interface. On the left, the 'Models' pane lists various models, with 'product-mix-reusable' highlighted. On the right, the Excel model grid is displayed, showing production decisions and part requirements. The grid includes columns for Part Name, LCD TV, Stereo, Speakers, Used, and Available. The 'Number to Build' row shows values of 0 for LCD TV, Stereo, and Speakers. The 'Part Requirements by Product' section lists requirements for Chassis, LCD Screen, Speaker, Power Supply, and Electronics. The 'Part Inventory Levels' section shows 'Used' and 'Available' quantities for each part.

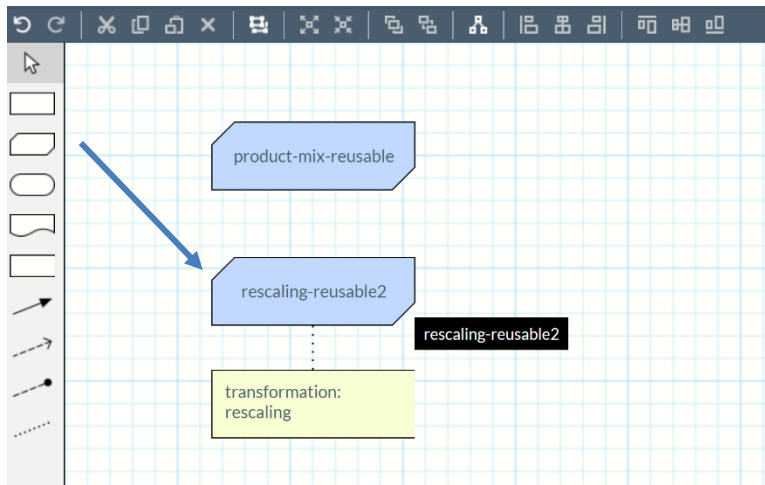
Production Decisions				Part Inventory Levels	
Part Name	LCD TV	Stereo	Speakers	Used	Available
Chassis	1	1	0	0	450
LCD Screen	1	0	0	0	250
Speaker	2	2	1	0	800
Power Supply	1	1	0	0	450
Electronics	2	1	1	0	600

Click the icon, on the Ribbon to create a new blank document. Then click the icon on the top right to open the Decision Flow Editor.

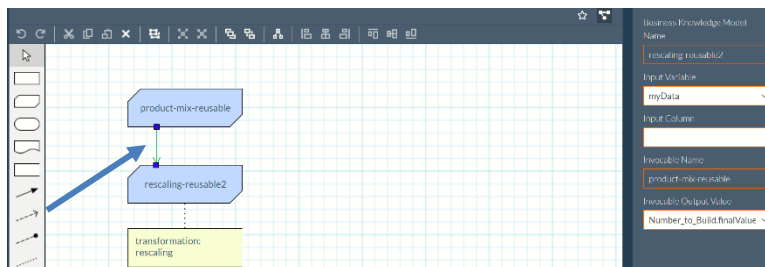
Select a Business Knowledge Model and drag it onto the grid. Under Invoke Model, enter the name of the Excel model, product-mix-reusable. Notice that this model has no Inputs but it does contain two outputs, Number_to_Build.finalValue and Total_Profit.finalValue.



Now the 2nd reusable model is added to the grid, rescaling-reusable2.

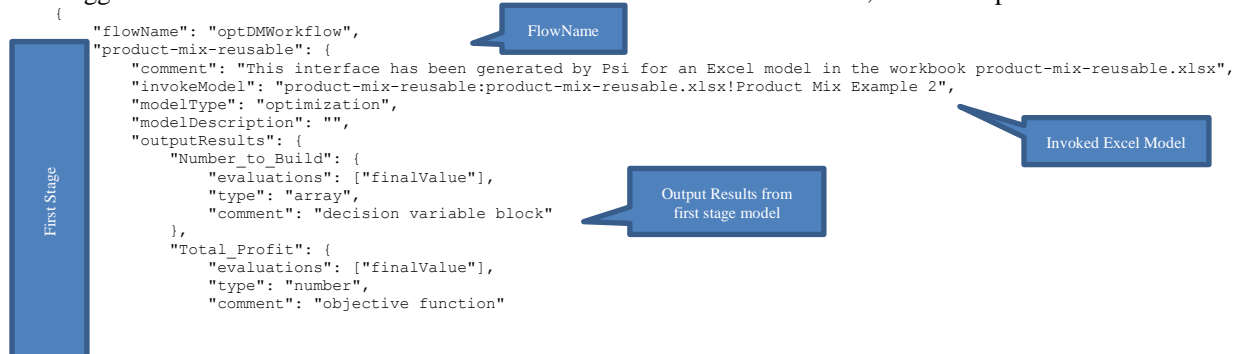


Next, we will connect the two stages. Select a Knowledge Requirement connection type and draw a connection from product-mix-reusable to rescaling-reusable2.



Select myData for Input Variable and Number_to_Build.finalValue for Invocable Output Value on the Properties pane on the left



Toggle back to the RASON Model Editor and enter a name for "flowName", such as "OptDMWorkflow".



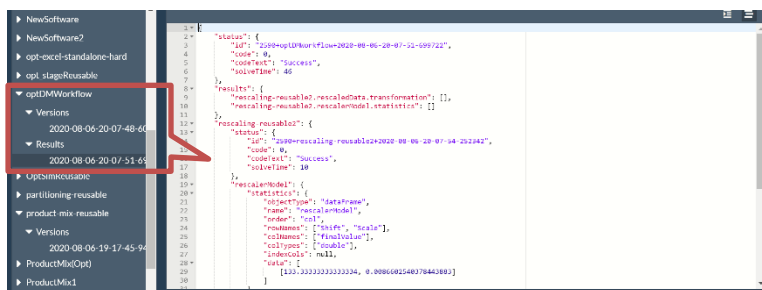
```

    },
    "rescaling-reusable2": {
      "comment": "This interface has been generated by Psi for a RASON model",
      "invokeModel": "rescaling-reusable2",
      "modelType": "datamining",
      "inputParameters": {
        "myData": {
          "value": "product-mix-reusable.Number_to_Build.finalValue",
          "type": "dataset",
          "comment": "a dataset"
        }
      },
      "outputResults": {
        "rescalerModel": {
          "evaluations": ["statistics"],
          "type": "dataset",
          "comment": "datamining evaluation"
        },
        "rescaledData": {
          "evaluations": ["transformation"],
          "type": "dataset",
          "comment": "datamining evaluation"
        }
      }
    },
    "modelDescription": "transformation: rescaling"
  }
}

```

Now click the Cloud icon () to POST the model, then click the down arrow next to the Play button  and select Solve to solve the decision flow.

Click the entry beneath Results to view the results of the solve.



Decision Flow using SQL Server

RASON enables users to easily define multi-stage ‘decision flows’, where a stage can perform a SQL operation, apply a data transformation, train a machine learning model, apply it to score new data, run a simulation, solve a mathematical optimization problem, or evaluate one or more linked decision tables. Results are passed between stages in a rich, standard “Indexed Data Frame” form. RASON uses an in-memory SQL database engine, plus “smart evaluation” to run only the stages that have new input data, to execute these multi-stage flows as efficiently as possible.

The example decision flow discussed in this section, dmMLRSQLWorkflow, is an inline decision flow made up of five stages:

- sqlTransformStage: Queries the hald-small-SQL.txt dataset for values of x4 greater than 6 which happens to be all records but record #7.

Y	X1	X2	X3	X4
78.5	7	26	6	60
74.3	1	29	15	52
104.3	11	56	8	20
87.6	11	31	8	47
95.9	7	52	6	33
109.2	11	55	9	22
102.7	3	71	17	6
72.5	1	31	22	44
93.1	2	54	18	22
115.9	21	47	4	26
83.8	1	40	23	34
113.3	11	66	9	12
109.4	10	68	8	12

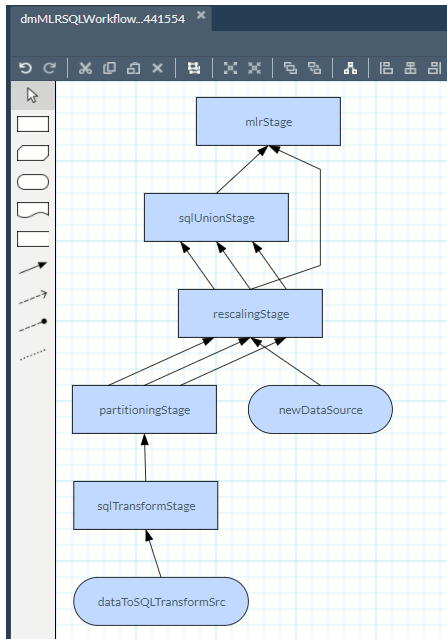
- **partitioningStage:** Partitions the resultant dataset from sqlTransformStage into 3 partitions: training, validation and test.
- **rescalingStage:** Rescales all three partitions from partitioningStage and also a new dataset, hald-small-unlabeled.txt.

X1	X2	X3	X4	
7	26	6	60	
1	29	15	52	
11	56	8	20	
11	31	8	47	
7	52	6	33	
11	55	9	22	
3	71	17	6	
1	31	22	44	
2	54	18	22	
21	47	4	26	
1	40	23	34	
11	66	9	12	
10	68	8	12	

- **sqlUnionStage:** Joins the rescaled validation and test partitions with the rescaled hald-small-unlabeled dataset to create one large combined dataset.
- **mlrStage:** Fits a model using multiple linear regression using the training partition and then uses this fitted model to score the combined dataset.

The screenshot below shows a graphical representation of the decision flow as depicted in the decision flow editor on www.RASON.com. To open this example click the "Open RASON example model" icon on the RASON.com ribbon, click Decision Flows – Inline and open the SQL Transform Decision Flow example.

This section gives a quick description of the decision flow showcasing the SQL commands in the 1st and 4th stages. For more information on how to construct a data science model, see the Defining a Data Science Model chapter that appears later in this guide.



The decision flow begins with the flowName property which assigns the name "dmMLRSQLWorkflow" to the decision flow and the flowDescription which simply gives a quick description of the decision flow. The 1st stage, "sqlTransformStage", begins immediately afterward.

```
{
  "flowName": "dmMLRSQLWorkflow",
  "flowDescription": "Example illustrates how SQL server commands may
    be utilized to manipulate data within a decision flow.",
```

Flow name

First Stage: sqlTransformStage

In the first stage of the decision flow, sqlTranformStage, a raw dataset, hald-small-SQL.txt, is imported into the RASON model as a datasource and then imported from a datasource into a dataset. Afterwards, a transformer is created which grabs all records in the dataset where the value for the x4 variable is greater than 6. In this instance the output of this stage is all records imported from the hald-small-SQL.txt dataset into the dataToSQLTransform dataset, except for 1.

1st Stage
Name

```
"sqlTransformStage": {
  "comment": "transformation: SQL",
  "modelName": "sqlTransformModel",
  "datasources": {
    "dataToSQLTransformSrc": {
      "type": "csv",
      "connection": "hald-small-SQL.txt"
    }
  },
  "datasets": {
    "dataToSQLTransform": {
      "binding": "dataToSQLTransformSrc",
      "colNames": [
        "Y",
        "X1",
        "X2",
        "X3",
        "X4"
```

"datasources"

Imports the hald-small-SQL.txt dataset into the dataToSQLTransformSrc datasource.

"datasets"

Binds all columns in dataToSQLTransformSrc data source to the dataToSQLTransform dataset.

```

    ]
  },
  "transformer": {
    "mySQLTransformer": {
      "type": "transformation",
      "algorithm": "sql",
      "parameters": {
        "query": "SELECT * FROM [dataToSQLTransform] WHERE
X4>6;"
      }
    }
  },
  "actions": {
    "sqlTransformedData": {
      "action": "transform",
      "evaluations": [
        "transformation"
      ]
    }
  }
},

```

"transformer"
Creates a transformer that selects all records within dataToSQLTransform (hald-small-SQL.txt) where the feature x4 has a value greater than 6.

"actions"
Applies the transformer to dataToSQLTransform and creates the new dataset, sqlTransformedData.

2nd Stage: partitioningStage

In the second stage of the decision flow, partitioningStage, the output from the 1st stage is partitioned into three partitions: training, validation and test using ratios of 50% of records assigned to the training partition, 30% of records assigned to the validation partition and the remaining 20% of records assigned to the test partition.

```

"partitioningStage": {
  "comment": "transformation: partitioning",
  "modelName": "partitioningModel",
  "inputParameters": {
    "dataToPartition": {
      "value": "sqlTransformStage.sqlTransformedData.tr
    }
  },
  "transformer": {
    "myPartitioner": {
      "type": "transformation",
      "algorithm": "partitioning",
      "parameters": {
        "partitionMethod": "RANDOM",
        "ratios": [
          "Training",
          0.5
        ],
        "Validation",
        0.3
      ]
    }
  }
}

```

2nd Stage Name

"inputParameters"
Imports input data from previous stage into current stage.

"modelName" and "comment"
Optional but it is considered best practice to include a model name and a short description for each stage in a decision flow.

"transformer"
Creates a transformer, myPartitioner that partitions a dataset into three partitions: training, validation and test.

```

        "Testing",
        0.2
    ]
    ],
    "seed": 123
}
}
},
"actions": {
    "trainingPartition": {
        "data": "dataToPartition",
        "action": "transform",
        "parameters": {
            "partition": "Training"
        },
        "evaluations": [
            "transformation"
        ]
    },
    "validationPartition": {
        "data": "dataToPartition",
        "action": "transform",
        "parameters": {
            "partition": "Validation"
        },
        "evaluations": [
            "transformation"
        ]
    },
    "testPartition": {
        "data": "dataToPartition",
        "action": "transform",
        "parameters": {
            "partition": "Testing"
        },
        "evaluations": [
            "transformation"
        ]
    }
}
},

```

"actions"
Executes the
transformer and
creates the new
dataset,
trainingPartition.

"actions"
Executes the
transformer and
creates the new
dataset,
validationPartition.

"actions"
Executes the
transformer and
creates the new
dataset, testPartition.

3rd Stage: rescalingStage

The third stage of the decision flow accepts the three partitions from the 2nd stage as input as well as a new data source, the hald-small-unlabeled.txt. Each of the three partitions and the new dataset are rescaled using unit normalization (with norm type = L1). The output from this stage are 4 rescaled datasets: rescaledTrainData, rescaledValidData, rescaledTestData and rescaledNewData.

```

"rescalingStage": {
    "comment": "transformation: rescaling",
    "modelName": "rescalingModel",
    "inputParameters": {
        "trainData": {
            "value": "partitioningStage.trainingPartition.transformation"
        },

```

"inputParameters"
Imports input data
from previous stage
into current stage.

3rd Stage Name


```

"validData": {
  "value": "partitioningStage.validationPartition.transformation"
},
"testData": {
  "value": "partitioningStage.testPartition.transformation"
}
},
"datasources": {
  "newDataSource": {
    "type": "csv",
    "connection": "hald-small-unlabeled.txt"
  }
},
"datasets": {
  "newData": {
    "binding": "newDataSource"
  }
},
"estimator": {
  "myRescaler": {
    "type": "transformation",
    "algorithm": "rescaling",
    "parameters": {
      "technique": "UNIT_NORMALIZATION",
      "normType": "L1",
      "excludedCols": [
        "Y"
      ]
    }
  }
},
"actions": {
  "rescalerModel": {
    "trainData": "trainData",
    "estimator": "myRescaler",
    "action": "fit",
    "evaluations": [
      "statistics"
    ]
  },
  "rescaledTrainData": {
    "data": "trainData",
    "fittedModel": "rescalerModel",
    "action": "transform",
    "evaluations": [
      "transformation"
    ]
  },
  "rescaledValidData": {
    "data": "validData",
    "fittedModel": "rescalerModel",
    "action": "transform",
    "evaluations": [
      "transformation"
    ]
  },
  "rescaledTestData": {

```

"datasources" Imports the hald-small-unlabeled.txt dataset into the current stage as the new data source, newDataSource.

"datasets"
Binds the dataset in newDataSource to the newData dataset.

"estimator"
Creates an estimator, myRescaler, that rescales a dataset, except for the Y column, using Unit normalization with a normType of L1.

"actions"
Fits a rescaled model, rescalerModel, to the trainData partition.

"actions"
Uses rescalerModel to rescale the training partition.

"actions"
Uses rescalerModel to rescale the validation partition.

"actions"
Uses rescalerModel to rescale the test partition.

```

    "data": "testData",
    "fittedModel": "rescalerModel",
    "action": "transform",
    "evaluations": [
      "transformation"
    ]
  },
  "rescaledNewData": {
    "data": "newData",
    "fittedModel": "rescalerModel",
    "action": "transform",
    "evaluations": [
      "transformation"
    ]
  }
}
},

```

"actions"

Uses rescalerModel to rescale the new data.

4th Stage: sqlUnionStage

The fourth stage of the decision flow accepts the three rescaled partitions and the rescaled newdata dataset from the 3rd stage, then performs a union (combines) of the validation partition, test partition and new data dataset into one large dataset.

```

"sqlUnionStage": {
  "comment": "transformation: SQL",
  "modelName": "sqlUnionModel",
  "inputParameters": {
    "validData": {
      "value": "rescalingStage.rescaledValidData.transformation",
      "targetCol": "Y"
    },
    "testData": {
      "value": "rescalingStage.rescaledTestData.transformation",
      "targetCol": "Y"
    },
    "newData": {
      "value": "rescalingStage.rescaledNewData.transformation"
    }
  },
  "transformer": {
    "mySQLTransformer": {
      "type": "transformation",
      "algorithm": "sql",
      "parameters": {
        "query": "SELECT * FROM [validData] UNION SELECT *
          FROM [testData] UNION SELECT * FROM [newData];"
      }
    }
  },
  "actions": {
    "joinedData": {
      "action": "transform",
      "evaluations": [
        "transformation"
      ]
    }
  }
}

```

4th Stage Name

"inputParameters"

Imports input data from previous stage into current stage.

"transformer"

Creates a transformer, mySQLTransformer that combines the rescaled validation partition (validData), rescaled test partition (testData) and the new data (newData) into one large dataset.

"actions"

Applies the transformer, mySQLTransformation to create one large combined dataset.

```

    }
  }
},

```

5th Stage: mlrStage

The fifth stage of the decision flow accepts the training partition and the combined dataset, joinedData, as inputs, fits a multiple linear regression model to the training partition and then uses this model to score the combined dataset.

```

"mlrStage": {
  "comment": "regression: linear model",
  "modelName": "mlrModel",
  "inputParameters": {
    "trainData": {
      "value": "rescalingStage.rescaledTrainData.transformation",
      "targetCol": "Y"
    },
    "joinedData": {
      "value": "sqlUnioStage.joinedData.transformation"
    }
  },
  "estimator": {
    "mlrEstimator": {
      "type": "regression",
      "algorithm": "linearRegression",
      "parameters": {
        "fitIntercept": true
      }
    }
  },
  "actions": {
    "myModel": {
      "trainData": "trainData",
      "estimator": "mlrEstimator",
      "action": "fit",
      "evaluations": [
        "anova",
        "influenceDiagnostics",
        "detailedResiduals",
        "coefficients",
        "regressionSummary",
        "detailedCoefficients",
        "multicollinearityDiagnostics",
        "varianceCovariance",
        "predictorScreeningInfo",
        "entranceTolerance"
      ]
    }
  },
  "trainScore": {
    "data": "trainData",
    "fittedModel": "myModel",
    "action": "predict",

```

5th Stage Name

"inputParameters"
Imports input data from previous stage into current stage.

"estimator"
Creates an estimator, mlrEstimator, to fit a dataset using multiple linear regression.

"actions"
Fits a model using the estimator, mlrEstimator, to the trainData partition.

"actions"
Using the fitted model, myModel, the trainScore action scores the trainData partition.

```

    "evaluations": [
      "prediction",
      "residuals",
      "newIntervals",
      "sse",
      "ss",
      "sst",
      "mse",
      "rmse",
      "mad",
      "r2"
    ],
    "joinedScore": {
      "data": "joinedData",
      "fittedModel": "myModel",
      "action": "predict",
      "evaluations": [
        "prediction"
      ]
    }
  }
}

```

"actions"

Using the fitted model, myModel, the joinedScore action scores the joinedData dataset.

More on Explicit and Automatic Scheduling

In production use, models may be run or "solved" many times. At present, a model is run only in response to a REST API request from an external application; this is often, but not always sufficient. Training a machine learning model, or solving a complex optimization model can take hours, whereas scoring a new case for a machine learning model, or retrieving selected results from an optimization model, may need to finish in milliseconds!

Explicit Scheduling

To support explicit scheduling, a new parameter, *?schedule=interval*, has been introduced to the existing REST API call, POST `rason.net/api/model/{name}/solve` where interval is an ISO 8601 time or repeating time interval (only of the form `Rn/starttime/duration`).

- If a time is specified, the RASON Server will run the model once on that day and time.
- If a repeating time interval is specified, the server will run the model at the starttime. After the model finishes and duration has elapsed, the server will run it again, for the specified number of repetitions.

Applications will typically use a model name (rather than an ID) in API calls such as GET `rason.net/api/model/name/status` to refer to the most recent run.

- POST `rason.net/api/model/{name}/stop` will stop an individual run,

Automatic Scheduling

RASON Decision Services utilizes a new property, "modelRecency": "time-since-last-run", at the same level as the existing modelName, modelDescription and modelType properties. This new property informs the RASON server of how recently the model was run in order to determine if the results (in JSON or OData form) can be

considered "current". The object "time-since-last run" must be in ISO 8601 time duration format (i.e. "PT12H" for 12 hours or "P1W" for 1 week). With this *property in place, the user may add "?schedule=automatic" to the existing REST API call POST rason.net/api/model/{name}/solve*. The RASON Server will consult past runs to determine the maximum time a solve usually takes to complete (max-time), and will arrange to run the model at intervals of (time-since-last-run – max-time). If the "modelRecency" property is omitted, the default value is infinity; the "?schedule=automatic" will simply cause the model to run one time.

More importantly, when a workflow of multiple stages is run using POST `rason.net/api/model/{nameorid}/solve?schedule=automatic`, the RASON Server will take into account modelRecency information for each stage in the workflow. For example, suppose a workflow has a final stage dependent on two intermediate stages A and B: If both A and B have runs that satisfy their recency requirement, the RASON Server will just run the final stage, using the previous results from A and B. If A has modelRecency=PT24H, takes 2 hours to run and last finished 23 hours ago, and B has modelRecency=Pt8H, takes 1 hour to run and last finished 6.5 hours ago, the server will re-run both A and B, and use the latest results (2 hours from now) to run the final stage. (The modelRecency property must be added to each stage in the decision flow.)

Calling POST `rason.net/api/model/{name}/solve?keep-intermediate-results=true&schedule=automatic` solves the flow and persists the stage results on the server.

Note: A new model instance is NOT created each time the workflow is run, one model is created on the initial POST `rason.net/api/model/{nameorid}/solve` and that model is reused for each run.

More on the Model Type Property

A new optional top-level property "modelType"={"optimization", "simulation", "datamining", "calculation"} has been introduced in RASON Decision Services for use with decision flows and standalone models. This property defines the model type as optimization, simulation, datamining or calculation within the RASON script and is used in three ways.

1. Explicit: If "modelType"={"optimization", "simulation", "datamining", "calculation"} exists in the RASON model, then the RASON interpreter will use this setting as the model type. If the specified type is incompatible with the actual model, a mismatch error will be returned.
2. Endpoint: If "modelType"={"optimization", "simulation", "datamining", "calculation"} is **not** present in the RASON model, the REST API endpoint called dictates the type and desired engine action. For example, if POST `rason.net/api/model/{nameorid}/simulate` is called to solve a stochastic optimization model, the workflow engine will run a simulation, only. In order to solve the model using stochastic optimization, you must specify "modelType" = "optimization". If the endpoint is incompatible with the actual model, a mismatch error will be returned.
3. Inferred: If "modelType"={"optimization", "simulation", "datamining", "calculation"} is **not** present in the RASON model and the REST API endpoint POST `rason.net/api/model/{nameorid}/solve` is called, the RASON interpreter will execute the engine according to the model type inferred by the RASON interpreter.

Note: The property "modelType":"" has no effect on the REST API endpoints POST `rason.net/api/model/interface` or POST `rason.net/api/model/{nameorid}/interface` as both endpoints generate universal objects which cover all possible results.

Defining Your Optimization Model

Introduction

This chapter introduces conventional optimization (with no uncertainty), simulation, simulation optimization and stochastic optimization using the RASON Modeling Language with a series of examples. This chapter takes you step-by-step through the process of creating an optimization RASON model from scratch.

Defining an Optimization Model

This section assumes that you're familiar with optimization, but you've never used the RASON Modeling language to define and solve an optimization model. This chapter translates a linear model from the algebraic statement of an optimization problem to a RASON model.

Setting Up a Model

When setting up an optimization model in the RASON Modeling language, your model will typically contain these essential segments:

1. **modelName:** RASON supports both named and unnamed models. A named model includes the `modelName: "name"` property in the text. A model name must be a string that can be URL-encoded and must be unique among models within a user's account. An unnamed model has no name. It is strongly recommended that a model be named to allow for easy recognition and retrieval when multiple models, instances and versions exist in a user's account. See the chapter [Using the REST API](#) for more information on named and unnamed models.
2. **modelType:** A new optional top-level property `"modelType"={"optimization", "simulation", "datamining", "calculation"}` has been introduced in RASON Decision Services for use with decision flows and standalone models. This property defines the model type as optimization, simulation, datamining or calculation within the RASON script and is used in three ways.
3. **data:** Where arrays to be used in the calculation of constraints and the objective function may be defined.
4. **variables:** Where the decision variables will be defined. In this example, we will define three variables in the X array, each with a lower bound of 0.
5. **objective:** Where the objective function to be maximized or minimized will be defined. In this example, we will maximize profit by multiplying the decision variables by the profit margin for each product.
6. **constraints:** Where the constraints will be defined. In this example, we define five constraints to ensure that the number of parts used are less than the number of parts in inventory.

Within this overall framework, you have a great deal of flexibility in choosing how to define and calculate your objective function and constraints. For example, the objective function will ultimately depend on the decision variable cells, but you don't have to calculate the entire function all at once. You can use any number of calculations in the data section to compute intermediate results, and use these to calculate the objective function or the constraints.

The RASON modeling language supports all but a few of Excel's functions¹ which means that you can write a linear expression easily with Excel's SUMPRODUCT function, or you can use + and * operators. You can define arrays and use Excel functions that return vector and matrix results and access your data from within Excel or a database.

The Model in Algebraic Form

Consider the following simple linear programming problem. Our factory is building three products: TV sets, stereos and speakers. Each product is assembled from parts in inventory, and there are five types of parts: chassis, picture tubes, speaker cones, power supplies and electronics units. Our goal is to produce the mix of products that will maximize profits, given the inventory of parts on hand.

The Algebraic Form

The problem can be described in algebraic form as follows. The decision variables are the number of products of each type to build: x_1 for TV sets, x_2 for stereos and x_3 for speakers. There is a fixed profit per unit for each product, so the objective function (the quantity we want to maximize) is:

Maximize $75 x_1 + 50 x_2 + 35 x_3$ (Profit)

Building each product requires a certain number of parts of each type. For example, TV sets and stereos each require one chassis, but speakers don't use one. The number of parts used depends on the mix of products built (the left hand side of each constraint), and we have a limited number of parts of each type on hand (the corresponding constraint right hand side):

Subject to:

$1 x_1 + 1 x_2 + 0 x_3 \leq 450$ (Chassis)

$1 x_1 + 0 x_2 + 0 x_3 \leq 250$ (Picture tubes)

$2 x_1 + 2 x_2 + 1 x_3 \leq 800$ (Speaker cones)

$1 x_1 + 1 x_2 + 0 x_3 \leq 450$ (Power supplies)

$2 x_1 + 1 x_2 + 1 x_3 \leq 600$ (Electronics)

Since the number of products built must be nonnegative, we also have the constraints $x_1, x_2, \text{ and } x_3 \geq 0$. Note that terms like $0 x_3$ are included purely to show the structure of the model – they can be either omitted or included in your data. Using this algebraic notation, we can start writing our RASON™ model.

The Basic Model

An easily understood way to formulate this model (though not necessarily the *best* way to formulate the model), can be found below. This model can be opened by clicking Examples – Example models discussed in RASON User Guide on the Editor tab at www.RASON.com. Note: All RASON arrays are 1-based.

```
{
  modelName: "BasicLinearModel",
```

¹ Note: Excel functions not supported by the RASON modeling language are: Call(), Cell(), CubeX(), EuroConvert(), GetPivotData(), HyperLink(), Indirect(), Info(), Offset(), RegisterID(), PivotDim(), PivotCube(), FormulaText(), Dollar(), Fixed(), Replace(), Search(), Text() and SqlRequest().

```

modelType: "optimization",
variables: {
  x: {
    lower: [0, 0, 0],
    finalValue: []
  }
},
constraints: {
  chassisUsed: {
    formula: "x[1] * 1 + x[2] * 1 + x[3] * 0",
    upper: 450
  },
  screensUsed: {
    formula: "x[1] * 1 + x[2] * 0 + x[3] * 0",
    upper: 250
  },
  speakersUsed: {
    formula: "x[1] * 2 + x[2] * 2 + x[3] * 1",
    upper: 800
  },
  powerUsed: {
    formula: "x[1] * 1 + x[2] * 1 + x[3] * 0",
    upper: 450
  },
  electronicsUsed: {
    formula: "x[1] * 2 + x[2] * 1 + x[3] * 1",
    upper: 600
  }
},
objective: {
  obj: {
    type: "maximize",
    formula: "x[1] * 75 + x[2] * 50 + x[3] * 35",
    finalValue: []
  }
}
}

```

The model begins with the `modelName` and `modelType` high-level properties. In `variables`, a column vector `x` of size 3 is defined (`x[1]`, `x[2]` and `x[3]`) to hold the number of products to produce. These are our decision variables. The dimensions of an array can be explicitly defined by the `dimensions` property. However, in the absence of the `dimensions` property (as in this example), the `value`, `lower`, or `upper` properties can implicitly define the shape of the array. The empty array, `finalValue[]` tells RASON that we want the results from solving to include the final values of each variable.

The 5 constraints are calculated in `Constraints` by multiplying the decision variables (`x[1]`, `x[2]` and `x[3]`) by the number of parts required for each product as shown in our algebraic form (above). The number of parts used must be less than or equal to the number of parts in inventory, so the upper bound for our first constraint is: $x[1] * 1 + x[2] * 1 + x[3] * 0 \leq 450$. The other constraints are similar. Note that both the upper and lower properties only support constant values, no formulas.

In `objective`, we must multiply the number of products to produce by the profit of each product using the formula: $x[1] * 75 + x[2] * 50 + x[3] * 35$. Since we want to maximize profit, we set `type` to `maximize` and ask for the final value of the function in the results.

This model works well for the business situation initially described, with exactly three products and five types of parts but it's not very flexible. What if more products were added and more parts were required? In the next section, we'll begin to generalize this model so it can handle different numbers of products and parts.

The Improved Model

One way to make our product mix model more flexible is to use some of RASON's built-in functions that operate on all the elements of an array at once, treating it as a vector or matrix. This model can be opened from the Editor tab at www.RASON.com by clicking RASON Examples – Example models discussed in RASON User Guide.

```
{
  modelName: "ImprovedLinearModel",
  modelType: "optimization",
  variables: {
    x: { dimensions: [3], lower: 0, finalValue: [] }
  },
  data: {
    profits: { dimensions: [3, 1], value: [75, 50, 35],
              finalValue: [] },
    parts: { dimensions: [5, 3], value: [[1, 1, 0], [1, 0, 0],
                                         [2, 2, 1], [1, 1, 0], [2, 1, 1]] },
    inventory: { value: [450, 250, 800, 450, 600] }
  },
  constraints: {
    num_used: { dimensions: [5], formula: "MMULT(parts, x)",
               upper: 'inventory' }
  },
  objective: {
    total: { formula: "sumproduct(x, profits)", type: "maximize",
             finalValue: [] }
  }
}
```

In the `variables` section, we again define a column vector `x` of size 3 but this time the `dimensions` property is used to explicitly define the size of the array. As a result, we can pass a single value to the `lower` property applying a lower bound of 0 to all three elements of the `x` array. Again, the empty array, `finalValue: []` tells the RASON system that you want the results from solving to include the final values of each variable.

In our example above, we calculated our five constraints in the `constraints` section using explicit formulas such as "`x[1] * 1 + x[2] * 1 + x[3] * 0`". In this model, we have created three arrays in `data` to hold the information for our model: `profits` (to hold the profit margin of each product), `inventory` (to hold the available inventory) and `parts` (a matrix holding the part requirements for each product). Note: The RASON™ Language Interpreter currently ONLY supports constants, a name with constant values, or a named array of constants for the *lower* and *upper* arguments.

The `MMULT` function in the `num_used` constraint multiplies the matrix `parts` (with 5 rows and 3 columns) by the column vector `x`. The product of this multiplication is a column vector with 5 rows which must be less than the column vector, `inventory`.

With this version of the model, if the number of parts or products are increased or decreased, you will need to adjust the size of your arrays, but no changes will be needed for the `constraints` formula which can sometimes be the hardest part of the model to define.

Using A Simple For() with Index Sets and a Table Assignment

The example above uses arrays to store the data used in the model. Arrays in the RASON Modeling language are limited to 2-dimensions. If more than 2-dimensions are required, you must use a table, instead. There are several advantages to using a table over an array for data storage.

- Tables may hold more than 2 dimensions.
- A table result may be used in an indexed array formula.

- A table can be sparse while an array is dense.
- The evaluation of a table is less expensive (time consuming) than the evaluation of an array with more than 2 dimensions.

RASON uses index sets to establish a basis of order for each dimension appearing in a table (or array). An index set is always a 1-dimensional array and must be created within the `indexSets` portion of the model.

The example code below creates two ordered sets, *part* and *prod*. The *part* set contains five items (in order as entered): chas, tube, cone, psup and elec while the *prod* set contains 3 items: tv, stereo and speakers. For more information on index sets, see *Index Sets* in the *RASON Reference Guide*. Note: All RASON arrays are 1-based.

```
{
  modelName:"RGProductMixTable4",
  modelType: "optimization",
  indexSets: {
    part: { value: ['chas', 'tube', 'cone', 'psup', 'elec'] },
    prod: { value: ['tv', 'stereo', 'speaker'] }
  },
  data: {
    parts: { indexCols: ['part', 'prod'],
      value: [
        ['chas', 'tv', 1],
        ['elec', 'stereo', 1],
        ['tube', 'tv', 1],
        ['cone', 'tv', 2],
        ['cone', 'stereo', 2],
        ['chas', 'stereo', 1],
        ['cone', 'speaker', 1],
        ['psup', 'tv', 1],
        ['psup', 'stereo', 1],
        ['elec', 'tv', 2],
        ['elec', 'speaker', 1]
      ]
    },
    profits: { dimensions: ['prod'], value: [75, 50, 35] },
    inventory: { indexCols: ['part'],
      value: [
        ['chas', 450],
        ['tube', 250],
        ['cone', 800],
        ['psup', 450],
        ['elec', 600]
      ]
    }
  },
  variables: {
    x: { dimensions: ['prod'], value: 0, lower: 0, finalValue: [] }
  },
  constraints: {
    "for(p in 'part')": {
      "cons[p]": { formula: "sumproduct(parts[p, ], x) -
        inventory[p]", upper: 0 }
    }
  },
  objective: {
    total: { formula: "sumproduct(x, profits)", type: "maximize",
      finalValue: [] }
  }
}
```

```
}
}
```

In the *data* section, an inline table object, *parts*, is created containing two index columns (*part* and *prod*) and a value column. (To open this full example, ProductMixTab2, click RASON examples on the Editor page ribbon and click Optimization – Optimization with Data Binding – ProductMixTab2.json.)

Since the index set *prod* exists, we can dimension the *profits* array according to this set in order to assign the correct profit values to the appropriate products. The inventory values are contained in the table, *inventory*. This table has one index column, *part* (*indexCols*: ['part']). Note: *indexCols* and *valueCols* properties describe a RASON Table.

Within *constraints*, we define 5 constraints for each *p* in the index set '*part*', then assign the evaluation values to *cons[5]*. The component, *cons[5]*, is defined as a *table* since neither the *dimensions* property nor the *upper/lower/value* properties are used to implicitly define *cons[5]* as an array. The variable *p* belongs to an index set. To reference a table element, we would use, say, *cons['chas']* rather than *cons[1]* as the latter is a typical array reference. In practice, when defining model functions with index sets, the formula result identifier will likely be a table, rather than an array. For more information on the use of index sets, tables, *for()* and *loop()*, see the later chapter *Using Arrays, For(), Loops and Tables*.

If the *sortIndexCols* (or *sort*) property is used, all *indexCols* will be sorted alphabetically. (Note: The properties *sort* and *sortIndexCols* perform the same function.) Otherwise, the table will be sorted as entered. In the example below, the order for *prod* will be: speaker, stereo, tv. While the order for *part* will be: chas, cone, elec, psup, tube.

```
data : {
  parts: {indexCols: ['part', 'prod'], sortIndexCols: true,
    value: [
      ['chas', 'tv', 1], ['chas', 'stereo', 1], ['chas', 'speaker', 0],
      ['tube', 'tv', 1], ['tube', 'stereo', 0], ['tube', 'speaker', 0],
      ['cone', 'tv', 2], ['cone', 'stereo', 2], ['cone', 'speaker', 1],
      ['psup', 'tv', 1], ['psup', 'stereo', 1], ['psup', 'speaker', 0],
      ['elec', 'tv', 2], ['elec', 'stereo', 1], ['elec', 'speaker', 1]]
    },
```

Using the Binding Property

What if you wanted to temporarily change the values of the Profit array in order to (say) create a "what if" scenario: "What if", we raised the price of each product by \$25? The binding property within the RASON Modeling language serves two functions, one of which is to allow write access to a table or array outside of the RASON model environment. For example, passing binding: "get" to the profits array within the data section would allow us to change the values for this array directly, through a REST API call.

```
data: {
  profits: {
    dimensions: [3],
    value: [75, 50, 35],
    binding: "get",
    finalValue: []
  },
```

To change the array elements in *profits* to 100, 75, 50; pass the following to a REST API call, via standard HTTP GET parameters, for example:

```
$.get(https://rason.net/api/optimize?profits=100,75,50...
```

To change only one element, say the middle element from 50 to 60, use:

```
$.get (https://rason.net/api/optimize?profits[2]=60...
```

What if our data is contained within an outside datasource, such as an SQL database? As mentioned earlier, RASON makes it exceptionally easy to work with data sources in the Microsoft ecosystem, such as:

- OneDrive and OneDrive for Business
- Common Data Service for Dynamics 365, Power Apps and Power Automate
- OData and CDS support for Power BI
- CData Cloud Hub support for access to 100+ enterprise data sources.

External data sources may be defined in the (optional) `datasources` section. Data from an outside datasource is imported into parametric tables or arrays to be used in 1. formula calculations or 2. as initial starting points for decision variables in a nonlinear optimization model. Currently the RASON modeling language supports nine different data types: "excel" (Microsoft Excel), "access" or "msaccess" (Microsoft Access), "odbc" (ODBC database), "OData" (OData endpoint), "mssql" (Microsoft SQL), "oracle" (Oracle database), CSV (Comma Separated Value), "json" (JSON file), or "xml" (XML file).

Data sources such as "Access", "SQL", "ODBC" and "CSV" contain data in tables with records described by index and value columns. Binding to these data sources results in table objects. Data source types such as Excel and CSV may contain data in 2-dimensional arrays without any descriptions. Binding to these data sources results in array objects. Objects must be bound to data sources within the `data` section. However, if exporting the results of a solve, we must bind to objects within the `variables`, `constraints`, `objective`, `uncertainVariables`, or `uncertainFunctions` sections. See the chapter "Using the RASON WEB IDE" for detailed information on how to create a new data connection and then use this connection in your RASON model. See the RASON Reference Guide for examples calling an external data file, with or without a named data connection.

Running an Optimization

Optimization models written in the RASON modeling language can be solved in two different ways: 1. Using the RASON REST Server or 2. Calling the Solver SDK directly. You can call the RASON REST server by using the RASON Web IDE on the website www.RASON.com or by using the RASON REST API from within your own application. You can call the Solver SDK directly using the RASON Modeling language by using the RASON Desktop IDE or by calling `prob.load` within a programming language such as C# or Java. No matter how you solve the model, the result will always be valid JSON. See the chapter "Solving RASON Models" for complete details on how to run a simulation model from the Editor page on www.RASON.com.

Defining Your Simulation Model

Introduction

This chapter introduces Monte Carlo simulation using the RASON Modeling Language with a series of examples. “Defining a Simulation Model” takes you step-by-step through the process of creating a Business Planning simulation model along with instructions on how to create a simulation parameter and perform a simulation optimization.

Defining a Simulation Model

Building a simulation model using the RASON modeling language is much like building an optimization model. As we learned above, in an optimization model, our RASON model contains components such as variables, data, constraints and objective. In a simulation model, our RASON model will contain components such as data, uncertainVariables and uncertainFunctions. You will use Psi Distribution functions to model the uncertainty as *inputs* and identify the *outputs* of special interest (such as Net Profit) to examine or summarize how they behave in light of the uncertainty.

Setting Up a Model

When setting up a simulation model in the RASON Modeling language, your model will typically contain these essential segments:

1. **modelName:** RASON Decision Services supports named and unnamed models. A named model includes the modelName: "name" property in its text. "name" must be a string that can be URL encoded and must be unique among models within a user's account. It is strongly recommended that a model be named to allow for easy recognition and retrieval when multiple models, instances and versions exist in a user's account. See the chapter Using the REST API for more information on named and unnamed models.
2. A new optional top-level property "modelType"={ "optimization", "simulation", "datamining", "calculation" } has been introduced in RASON Decision Services for use with decision flows and standalone models. This property defines the model type as optimization, simulation, datamining or calculation within the RASON script and is used in three ways.
3. **data:** Where arrays to be used in the calculation of uncertain function and/or uncertain variables will be defined.
4. **uncertainVariables:** Where the uncertain variables will be defined. In this example, we will define two uncertain variables using the Psi Distribution functions PsiTriangular and PsiIntUniform.

5. **uncertainFunctions:** Where the uncertain functions will be defined. In this example, we define one uncertain function which calculates the expected (or average) profit.

As mentioned above, within this overall framework, you have a great deal of flexibility in choosing how to define and calculate your uncertain variables and uncertain functions. For example, the uncertain function will ultimately depend on the uncertain variables, but you don't have to calculate the entire function all at once. You can use any number of calculations in the `data` section to compute intermediate results, and use these to calculate the uncertain function.

The RASON modeling language supports all but a few of Excel's functions² which means that you can write a formula easily using functions such as SUM, SUMPRODUCT, etc. along with operators such as + and *. You can define arrays and use Excel functions that return vector and matrix results and access your data from within an Excel worksheet or a database.

Uncertain Variables

In any problem, there are factors or inputs that you can control – for example, the price you set for a product, and factors or inputs that you *cannot* control – for example, customer demand, interest rates, etc. **Uncertain variables** (*random variables* in mathematics) are used to represent inputs that are uncertain and beyond your control. (It uses decision variables to represent factors or inputs that you *can* control.)

Uncertain Functions

You will also have outputs or results of interest – such as Net Profit – that you can compute, using formulas that depend on the factors influencing the problem – possibly both decision variables and uncertain variables. We'll use the term **uncertain functions** for quantities whose calculation depends on uncertain variables (in mathematics these are called *functions of random variables*).

A Business Planning Example

We'll illustrate the process of building a simulation model step by step, using a simple business planning example. Imagine you are the marketing manager for a firm that is planning to introduce a new product. You need to estimate the first year profit from this product, which will depend on:

- Sales in units
- Price per unit sold
- Unit manufacturing cost
- Fixed costs and overhead

Profit will be calculated as **Profit = Sales * (Price - Unit cost) - Fixed costs**. Fixed costs are known to be **\$120,000**. But the other factors all involve some uncertainty. Sales in units can cover quite a range, and the selling price per unit will depend on competitor actions. Unit manufacturing costs will also vary depending on vendor prices and production experience. We will build the simulation model step-by-step but if you want to follow along you can open the completed model by browsing to C:\Program Files\Frontline Systems\Solver SDK Platform\Examples\RASON and opening the file UGForecast.json if using the Desktop IDE or select the UGForecast.json example model on the Editor page at www.RASON.com.

Uncertain Variables: Sales and Price

Based on your market research, you believe that there are equal chances that the market demand will be Slow, OK, or Hot for this product:

² Note: Excel functions not supported by the RASON modeling language are: Call(), Cell(), CubeX(), EuroConvert(), GetPivotData(), HyperLink(), Indirect(), Info(), Offset(), RegisterID(), PivotDim(), PivotCube(), FormulaText(), Dollar(), Fixed(), Replace(), Search(), Text() and SqlRequest().

- In the Slow market demand scenario, you expect to sell **50,000** units at an average price of **\$11.00** per unit.
- In the OK market scenario, you expect to sell **75,000** units, but you'll likely realize a lower average selling price of **\$10.00** per unit.
- In the Hot market scenario, you expect to sell **100,000** units, but this will bring in competitors who will drive down the average selling price to **\$8.00** per unit.

Since the scenarios are equally likely, your *average* volume is **75,000** units, and your *average* price per unit is **\$9.67**. But think: How likely is this *average* case? (Will it ever actually occur?)

Uncertain Variables: Unit Cost

Your firm's production manager advises you that unit costs may be anywhere from **\$4.50** to **\$8.50**, with a most likely cost of **\$6.50**. The most likely cost is also the *average* cost.

Uncertain Function: Net Profit

Net Profit is calculated as **Profit = Sales * (Price - Unit cost) - Fixed costs**. Sales, Price and Unit costs are all uncertain variables, so Net Profit is an uncertain function.

Defining the Simulation Model

Since there are equal chances that the market will be Slow, OK, or Hot, we want to create an uncertain variable that selects among these three possibilities, by drawing a random number – say 1, 2 or 3 – with equal probability. We can do this easily in the RASON modeling language using an **integer uniform** probability distribution or `PsiIntUniform()`. We'll then base our Sales Volume and Selling Price on this uncertain variable. See the `marketType` uncertain variable in our example code below. A sample drawn from a discrete distribution is always one of a set of discrete values, such as integer values.

Next, we'll deal with Unit Cost. We have not just three, but *many* possible values for this variable: It can be anywhere from \$4.50 to \$8.50, with a most likely cost of \$6.50. A crude but effective way to model this is to use a *triangular* distribution. The RASON modeling language provides a function called **PsiTriangular()** for this distribution (see the `unitCost` uncertain variable below).

When creating any RASON model, our model definition begins with a simple open bracket. Next we see three optional, but strongly recommended, high level properties, `modelName`, `modelDescription` and `modelType`. `modelName` assigns a name to the model for easy retrieval later, `modelDescription` gives a short description of the problem and `modelType` informs the RASON Server that this model is a simulation model.

Since we are creating a simulation model, rather than an optimization model, our first section will be `uncertainVariables` where we will define our two uncertain variables using `PsiIntUniform()` and `PsiTriangular()` followed by `uncertainFunctions` where we will calculate the Net Profit.

```
{
  modelName: "UGForecastSimExample",
  modelType: "simulation",
  uncertainVariables: {
    marketType: {
      formula: "PsiIntUniform(1,3)"
    },
    unitCost: {
      formula: "PsiTriangular(4.5, 6.5, 8.5)"
    }
  },
}
```

The next section, `formulas`, calculates our Sales Volume (`salesVolume`), Selling Price (`sellingPrice`), Gross Sales (`grossSales`) and Total Unit Cost (`totalCosts`). Our Sales Volume and Selling Price are based on the value of the `marketType` uncertain variable. If `marketType` equals 1, the Sales Volume and Selling Price are 50,000 and \$11, if `marketType` equals 2, the Sales Volume and Selling Price are 75,000 and \$10 and if `marketType` equals 3, the Sales Volume and Selling Price are 100,000 and \$8. Gross Sales is calculated by multiplying the Selling Price by the Sales Volume and `totalCosts` is calculated by multiplying Sales Volume by Unit Cost. Note: All RASON arrays are 1-based.

```
formulas: {
  salesVolume: { formula: "if(marketType = 1, 50000, if(marketType = 2,
    75000, if(marketType = 3, 100000)))"
  },
  sellingPrice: { formula: "if(marketType = 1, 11, if(marketType = 2,
    10,
    if(marketType = 3, 8)))"
  },
  grossSales: { formula: "sellingPrice * salesVolume" },
  totalCosts: {
    formula: "salesVolume * unitCost"
  }
},
```

The data section holds the constant `fixedCosts`, which equals \$120,000.

```
data: {fixedCosts: { value: 120000 }
},
```

Finally, the last section, `uncertainFunctions`, calculates the Net Profit by subtracting `totalCosts` and `fixedCosts` from `grossSales`. Since we want to know the expected (or average) net profit, we ask for the mean in our result.

```
uncertainFunctions: {
  netProfit: { formula: "grossSales - totalCosts - fixedCosts",
    mean: [] }
}
```

Note that the result of the formula (`"grossSales - totalCosts - fixedCosts"`) is 1,000 scenarios or trials, rather than a single value. The mean that will be returned in the result will be the **true average** of Net Profit across 1,000 or more scenarios or trials – not a single calculation from *average* values of the *inputs*. We could have asked for additional result measures for `netProfit` such as the minimum function value, maximum function value, standard deviation, variance, median, mode, etc. One, several, or all can be returned in a result. For a complete list of measures that may be returned please see the chapter *RASON Model Components* in the **RASON Reference Guide**.

See below for the complete RASON model.

```
{
  modelName: "UGForecastSimExample",
  modelType: "simulation",
  uncertainVariables: {
    marketType: {
      formula: "PsiIntUniform(1,3)"
    },
    unitCost: {
      formula: "PsiTriangular(4.5, 6.5, 8.5)"
    }
  },
  formulas: {
    salesVolume: {
      formula: "if(marketType = 1, 50000, if(marketType = 2, 75000,
        if(marketType = 3, 100000)))"
    }
  }
}
```



```

    },
    sellingPrice: {
        formula: "if(marketType = 1, 11, if(marketType = 2, 10,
        if(marketType = 3, 8)))"
    },
    grossSales: {
        formula: "sellingPrice * salesVolume"
    },
    totalCosts: {
        formula: "salesVolume * unitCost"
    }
},
data: {
    fixedCosts: {
        value: 120000
    }
},
uncertainFunctions: {
    netProfit: {
        formula: "grossSales - totalCosts - fixedCosts",
        mean: []
    }
}
}

```

Running a Simulation

As with optimization models, simulation models written in the RASON modeling language can be simulated in two different ways: 1. Using the RASON REST Server or 2. Calling the Solver SDK directly. You can call the RASON REST server by using the RASON Web IDE on the website www.RASON.com or by using the RASON REST API from within your own application. You can call the Solver SDK directly using the RASON Modeling language by using the RASON Desktop IDE or by calling `prob.load` within a programming language such as C# or Java. As with optimization, no matter how you solve the model, the result will always be valid JSON. See the topic "Running Example Models on the "Editor" Page" within the "RASON Services WEB IDE" chapter for complete details on how to run a simulation model from the Editor page on www.RASON.com.

An Airline Management Revenue Model

In this section, we'll explore a simple airline revenue management model, also known as a yield management model. We'll start with a simple simulation model, like the one in the previous section. But in this section, we'll use a `PsiSimParam()` to run *multiple parameterized simulations* and also solve *simulation optimization* model.

We will setup these examples step by step but if you'd like to follow along, you can open each of them in either the Desktop IDE or Web IDE. The example files that will be used in this section are:

UGYieldManagement1.json (a simple simulation model), UGYieldManagement2.json (a model with multiple parameterized simulations) and UGYieldManagement3.json (a simulation optimization model). You can open each example by clicking RASON Examples on the Editor page, then *Example models discussed in RASON User Guide* and the file.

A Single Simulation

The model UGYieldManagement1.json is shown below.

```

{
    "modelName": "UGYieldManagement1Example",
    "modelDescription": "UGYieldManagement1 Example - Simulation",
    "modelType": "simulation",

```

```

"modelSettings": {
  "numSimulations": 1,
  "numTrials": 1000,
  "randomSeed": 1
},
"data": {
  "price": {
    "value": 200
  },
  "capacity": {
    "value": 100
  },
  "sold": {
    "value": 110
  },
  "refund_no_shows": {
    "value": 0.5
  },
  "refund_overbook": {
    "value": 1.25
  }
},
"uncertainVariables": {
  "no_shows": {
    "formula": "PsiLogNormal(0.1*sold, 0.06*sold)",
    "mean": []
  }
},
"formulas": {
  "show_ups": {
    "formula": "sold - Round(no_shows, 0)"
  },
  "overbook": {
    "formula": "Max(0, show_ups - capacity)"
  }
},
"uncertainFunctions": {
  "revenue": {
    "formula": "price*(sold - refund_no_shows*Round(no_shows, 0) - refund_overbook*overbook)",
    "mean": [],
    "percentiles": [],
    "trials": []
  }
}
}

```

The model depicts a hypothetical airline flight from San Francisco to Seattle. The flight has 100 seats, and tickets are \$200 per seat. Some passengers who purchase tickets are “no-shows” whose seats will be empty; in this example we assume that such passengers receive a refund of 50% of their purchase price. To utilize their ‘perishable inventory’ of seats, the airline would like to sell more than 100 tickets for each flight. But we assume that Federal regulations require that any ticketed passenger who is unable to board the flight due to overbooking is entitled to compensation of 125% of the ticket price.

The airline would like to know how much revenue it will generate from each flight, less refunds for no-shows and compensation for ‘bumped’ passengers. As shown above, this net revenue amount is calculated in

revenue within uncertainFunctions, for any specific number of tickets sold (110 above) and number of no-shows (5 above).

The uncertain quantity in this model is the number of no-shows; hence we should model this with an *uncertain variable*. We quickly realize that the **number of no-shows** will depend on the **number of tickets sold**. After some research, we decide that we can use a LogNormal distribution for the number of no-shows: no_shows within uncertainVariables uses the formula **=PsiLogNormal(0.1*sold,0.06*sold)** .

Within formulas, the formula for show_ups contains the ROUND(no_shows,0) function to ensure that the number of no-shows is an integer value – and this is used to compute the net revenue in the revenue uncertain function .

Performing the Simulation

If you haven't already, open either the WEB IDE on www.RASON.com or the Desktop IDE (typically installed at C:\Program Files\Frontline Systems\Solver SDK Platform\Bin) and open the file, UGYieldManagement1.json. To perform a single simulation (with 1,000 Monte Carlo trials) using the WEB IDE, click:

- POST rason.net/api/model to POST the model to the RASON Server
- POST rason.net/api/model/{nameorid}/simulate to run the simulation.

The following result will be returned. Note: All RASON arrays are 1-based.


```
{
  "status": {
    "code": 0,
    "id": "2590+UGYieldManagement1Example+2020-02-07-21-29-25-683482",
    "codeText": "Solver has completed the simulation."
  },
  "uncertainFunctions": {
    "revenue": {
      "mean": 20445.30,
      "percentiles": [
        18599, 19098, 19300, 19500, 19695, 19800, 19800, 19900,
        19950, 19950
        ...
        20900, 21000, 21000, 21000, 21000, 21000, 21000, 21000]
      ],
      "trials": [
        20700, 20900, 20400, 20900, 19700, 20700, 19800, 20250,
        ...,
        19400, 19950, 20800, 20600, 20250, 19800, 20100, 20800]
      ]
    }
  },
  "uncertainVariables": {
    "no_shows": {
      "mean": 10.9922
    }
  }
}
```

The mean or expected net revenue is about \$20,000. The 10th percentile equals \$19,950 which tells us that if we sell 110 tickets, we'll earn almost as much revenue as a full flight (100 seats * \$200) 90% of the time. However, what if we asked the question, "How many tickets should we sell?"

Changing Parameter Values

We can answer this question by changing the value for `sold` and resolving. If we are using either the WEB or Desktop IDE, it's easy to change the value of this constant, we can simply delete the value of 110 and type in a



new value, say 125, then either click  in the Desktop IDE or, in the WEB IDE,

- PUT `rason.net/api/model/{nameorid}` to update the model on the RASON Server then
- POST `rason.net/api/model/{nameorid}/simulate` to run a simulation.

Note: If we were calling this model from within our App and it wasn't as easy to modify the model, we could pass new data directly during a REST API call via standard HTTP GET parameters such as:

```
$.get(https://rason.net/api/simulate?sold=125.....
```

Recall that we would need to also add a `"binding": "get"` property to the `sold` data parameter.

```
"sold": {
  "value": 110,
  "binding": "get"
},
```

The returned result is as follows.

```
{
  "status": {
    "code": 0,
    "id": "2590+UGYieldManagement1Example+2020-02-10-19-48-23-720358",
    "codeText": "Solver has completed the simulation."
  },
  "uncertainFunctions": {
    "revenue": {
      "mean": 20505.2,
      "percentiles": [
        [19200, 19200, 19350, 19350, 19491, 19500, 19500, 19500,
          ...
          21900, 22190, 22050, 22200, 22200, 22300, 22350, 22400]
      ],
      "trials": [
        [20250, 20550, 19800, 20700, 22400, 20100, 19200, 19650, 20850,
          ...
          19650, 22100, 19200, 20700, 21000, 19500, 22500, 21900, 20700]
        ]
      }
    },
    "uncertainVariables": {
      "no_shows": {
        "mean": 12.4911
      }
    }
  }
}
```

Surprisingly, our expected revenue did not increase by much, only \$60 ($\$20,505 - \$20,445 = \60). Selling 125 tickets does seem to be a bit better than selling 110 tickets, but we're not yet sure that this is the *best* number of tickets to sell. We could again change the value for `sold` to, say, 150 and resolve, then do it again for 175, etc.

Multiple Parameterized Simulations


However, what if we could vary a parameter over a range, and run a simulation for each different number of tickets sold and summarize the results? With the RASON Modeling language you can! Make the following changes to `UGYieldManagement1.json`.

1. Set `numSimulations`: 41 within `modelSettings`. Forty-one simulations will be ran, each with 1,000 trial values.
2. Remove the constant `sold`:

```
"data": {
  "price": {
    value: 200
  },
  "capacity": {
    value: 100
  },
  "refund_no_shows": {
    value: 0.5
  },
  "refund_overbook": {
    value: 1.25
  }
},
```

3. Add a `parameters` section which uses `PsiSimParam` to automatically modify the value for `sold` on each simulation starting with 110 and ending with 150. The value that we specify for `numSimulations`, within `modelSettings`, determines the incremental value for `sold` for each simulation. In this example `sold` will have a value of 110 on the first simulation, 111 on the second simulation, and so on, through 150 on the 41st simulation.

```
"parameters": {
  "sold": {
    "formula": "PsiSimParam(110, 150)",
    "finalValue": []
  }
},
```

4. Solve by clicking  in the Desktop IDE or PUT `rason.net/api/model/{nameorid}` then POST `rason.net/api/model/{nameorid}/simulate` in the WEB IDE: 41 simulations are completed, each with a different value for `sold`. The results for the first three simulations are below.

```
{
  "simulations": [{
    "status": {
      "code": 0,
      "id": "2590+UGYieldManagement1Example+2020-02-11-18-32-41-055855",
      "codeText": "Solver has completed the simulation."
    },
    "simulation": 1,
    "parameters": {
      "sold": {
        "finalValue": 110
      }
    },
    "uncertainFunctions": {
      "revenue": {
```

```

    "mean": 20445.3,
    "percentiles": [
      [18098.5, 18100, 18250, 18250, 18250, 18250, 18400, 18400,
        ...
        21700, 21850, 22000, 22300, 22600, 22900, 23353, 24101]
    ],
    "trials": [
      [19300, 19750, 18700, 19750, 22300, 19150, 17950, 18550,
        ...
        18100, 19900, 20200, 18400, 21850, 21250, 19900]
    ]
  },
  "uncertainVariables": {
    "no_shows": {
      "mean": 10.9922
    }
  },
  {
    "status": {
      "code": 0,
      "id": "2590+UGYieldManagement1Example+2020-02-11-18-32-41-055855",
      "codeText": "Solver has completed the simulation."
    },
    "simulation": 2,
    "parameters": {
      "sold": {
        "finalValue": 111
      }
    },
    "uncertainFunctions": {
      "revenue": {
        "mean": 20500.2,
        "percentiles": [
          [18098.5, 18100, 18250, 18250, 18250, 18250, 18400, 18400,
            ...
            21550, 21700, 21850, 22000, 22300, 22600, 22900, 23353, 24101]
          ],
          "trials": [
            [19300, 19750, 18700, 19750, 22300, 19150, 17950, 18550,
              ...
              22750, 18100, 19900, 20200, 18400, 21850, 21250, 19900]
            ]
          ],
          "uncertainVariables": {
            "no_shows": {
              "mean": 11.0921
            }
          }
        },
        {
          "status": {
            "code": 0,

```

```

      "id": "2590+UGYieldManagement1Example+2020-02-11-18-32-41-055855",
      "codeText": "Solver has completed the simulation."
    },
    "simulation": 3,
    "parameters": {
      "sold": {
        "finalValue": 112
      }
    },
    "uncertainFunctions": {
      "revenue": {
        "mean": 20543.4,
        "percentiles": [
          18098.5, 18100, 18250, 18250, 18250, 18250, 18400,
          ...
          21850, 22000, 22300, 22600, 22900, 23353, 24101]
        ],
      "trials": [
        [19300, 19750, 18700, 19750, 22300, 19150, 17950,
          ...
          18100, 19900, 20200, 18400, 21850, 21250, 19900]
        ]
      }
    },
    "uncertainVariables": {
      "no_shows": {
        "mean": 11.1921
      }
    }
  },
},

```

Since we asked for the `finalValue` for `sold`, this value is printed for each simulation, along with the expected value and percentile values.

5. To see the result for simulation #5 *only*, add `simulationIndex: 5` to `modelSettings` as shown below, then update the model and resolve.

```

modelSettings: {
  numSimulations: 41,
  numTrials: 1000,
  randomSeed: 1,
  simulationIndex: 5
},

```

The Result will contain information for simulation #5 only.

```

{
  "simulations": [{
    "status": {
      "code": 0,
      "id": "2590+UGYieldManagement1Example+2020-02-11-19-24-31-042646",
      "codeText": "Solver has completed the simulation."
    },
    "simulation": 5,
    "parameters": {
      "sold": {
        "finalValue": 114
      }
    }
  ]
}

```

```

    }
  },
  "uncertainFunctions": {
    "revenue": {
      "mean": 20597.1,
      "percentiles": [
        19299, 19600, 19750, 19750, 19750, 19900, 19900, 19900,
        ...,
        21300, 21300, 21300, 21400, 21400, 21400, 21400]
      ],
      "trials": [
        20650, 20950, 20200, 20950, 20400, 20500, 19600, 20200,
        ...,
        20100, 19750, 21100, 21400, 20050, 20600, 20900, 21100]
      ]
    }
  },
  "uncertainVariables": {
    "no_shows": {
      "mean": 11.3919
    }
  }
}
]]
}

```

Simulation Optimization

Running multiple parameterized simulations, and examining the results has given us a good deal of information about the behavior of this simulation model. However, if we take just a few more steps we can answer the question, "How many tickets should we sell to realize the *maximum expected* net revenue?"

Open the third example, UGYieldManagement3.json or follow the steps below to create UGYieldManagement3.json from UGYieldManagement2.json.

1. Change the `modelType` property to "optimization". If left at `"modelType": "simulation"`, only a simulation will be performed, not a simulation optimization.
2. Replace parameters with variables and define `sold` as an integer variable (since we can't sell fractional tickets) with a lower bound of 0 (since we can't sell a negative number of tickets). We are interested in the value of this variable in the solution, so we must ask for the `finalValue` to be returned in the Result.

```

sold: {
  lower: 0,
  upper: 500,
  type: "integer",
  finalValue: []
},

```

3. Replace `uncertainFunctions` with `objective` and maximize the expected value of the formula.

```

objective: {
  revenue: {
    formula: "price*(sold - refund_no_shows*Round(no_shows, 0)
- refund_overbook*overbook)",
    chanceType: "ExpVal",
    type: "maximize",
    finalValue: []
  }
}

```


4. Replace numSimulations: 41 with simulationOptimization: True in modelSettings.

```
modelSettings: {  
    simulationOptimization: True,  
    numTrials: 1000  
},
```

5. Select the Evolutionary engine by adding "engine": "Evolutionary" in engineSettings.

```
engineSettings: {  
    engine: "Evolutionary",  
    randomSeed: 1  
},
```

Here is the complete model. Note: All RASON arrays are 1-based.

```
{  
    modelName: "UGYieldManagement3",  
    modelType: "optimization",  
    modelSettings: {  
        simulationOptimization: true,  
        numTrials: 1000  
    },  
    engineSettings: {  
        engine: "Evolutionary",  
        randomSeed: 1  
    },  
    variables: {  
        sold: {  
            lower: 0,  
            upper: 500,  
            type: "integer",  
            finalValue: []  
        }  
    },  
    data: {  
        price: {  
            value: 200  
        },  
        capacity: {  
            value: 100  
        },  
        refund_no_shows: {  
            value: 0.5  
        },  
        refund_overbook: {  
            value: 1.25  
        }  
    },  
    uncertainVariables: {  
        no_shows: {  
            formula: "PsiLogNormal(0.1*sold, 0.06*sold)"  
        }  
    },  
    formulas: {  
        show_ups: {  
            formula: "sold - Round(no_shows, 0)"  
        },  
    },  
}
```

```

        overbook: {
            formula: "Max(0, show_ups - capacity)"
        }
    },
    objective: {
        revenue: {
            formula: "price*(sold - refund_no_shows*Round(no_shows, 0) -
            refund_overbook*overbook)",
            chanceType: "ExpVal",
            type: "maximize",
            finalValue: []
        }
    }
}

```

Solve once again by clicking **Optimize** in the Desktop IDE or, if using the RASON Web IDE, first post the model using *POST* rason.net/api/model then run a simulation optimization using *POST* rason.net/api/model/{nameorid}/optimize in the WEB IDE. The returned Result is shown below.

```

{
  "status": {
    "code": 2,
    "id": "2590+UGYieldManagement3+2020-03-20-14-19-27-538120",
    "codeText": "Solver cannot improve the current solution. All
    constraints are satisfied."
  },
  "variables": {
    "sold": {
      "finalValue": 117
    }
  },
  "objective": {
    "revenue": {
      "finalValue": 20620.2
    }
  }
}

```

As shown on the right in the above screenshot, the decision variable, `sold`, has a final value of 117 and the expected value of net revenue for this number of tickets sold is little over \$20,000. We now have an answer to our question "How many tickets should we sell to realize the *maximum expected* net revenue?"

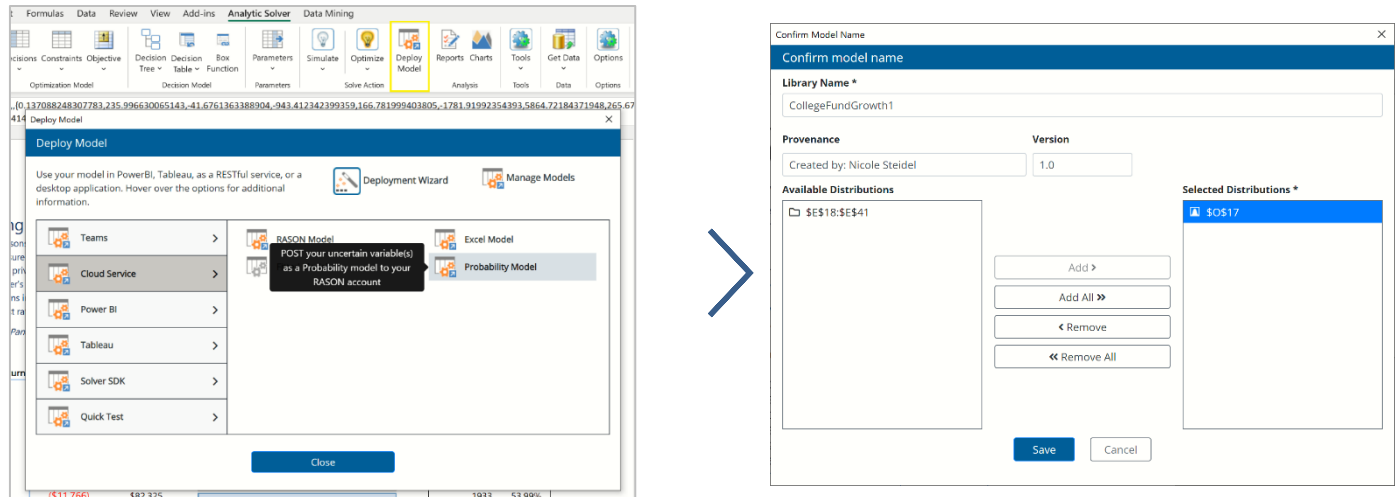
Using a Probability Model within RASON.

With the latest version of RASON Decision Services, a Probability Model may be deployed from Analytic Solver (for Excel or the Cloud) and utilized within a RASON model.

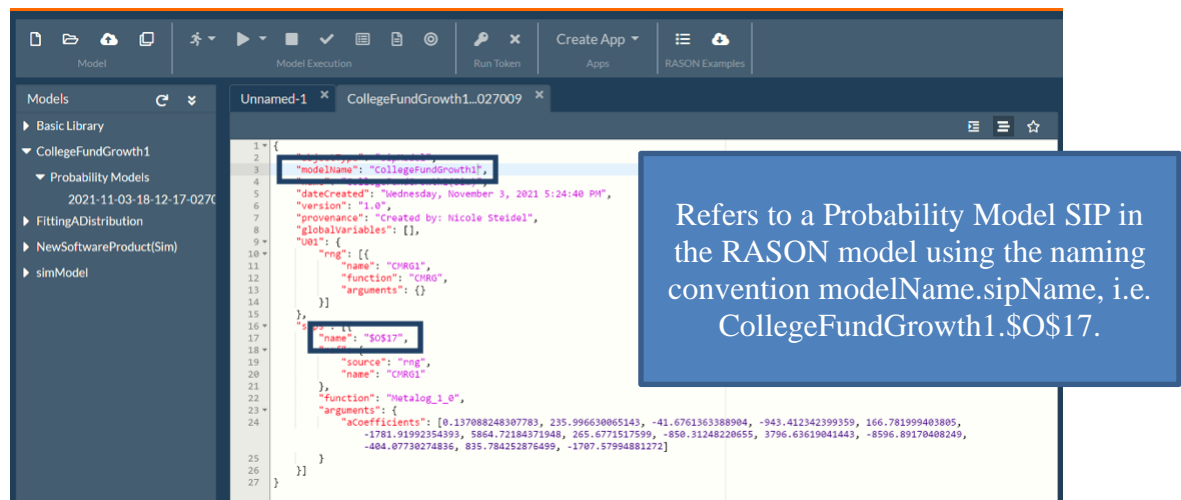
This example begins in Analytic Solver for Excel where a Probability Model is created from the CollegeFundGrowth1(Sim) example model. Starting with Analytic Solver V2021.5, users can deploy and share probability models, following the open Probability Management 3.0 standard. Using these Shared Information Probability resources (SIPs, SLURPs and DISTs), users can ensure that their group or organization uses consistent data about uncertain/risky variables across simulation or decision models, enabling model results to be meaningfully compared. Users of Analytic Solver can deploy Probability Models directly to the RASON server via the Deploy Model feature.

This example begins with the CollegeFundGrowth1(Sim) example simulation model in Analytic Solver for Excel. This example emulates the accumulation of funds in a college savings account.

- The yearly investment return is modeled using the past performance of the S&P 500. A distribution has been fit to this historical data, using PsiFit in Analytic Solver.
- The Deploy Model feature in Analytic Solver is used to export the fitted distribution to the RASON Server so that a RASON model may be developed using the exported probability model. For more information on exporting Probability Models in Analytic Solver, see the Deploying your Model chapter within Analytic Solver User Guide.



The exported Probability Model is shown as it appears in the Editor tab of RASON.com. Notice that the model appears under Probability Model on the Models pane.



The identical distribution used for the yearly returns in CollegeFundGrowth1(Sim).xlsx will also be used in the CollegeFundGrowth1 model in RASON.

The beginning of the college fund growth model in RASON is shown below. Note that in this example, each yearly return uses the same distribution, the PsiMetalog distribution.

```
{
  "modelName": "CollegeFundGrowth1",
  "modelDescription": "Emulates a college fund savings model.",
  "modelSettings": {
    "randomGenerator": "Lecuyer",
    "samplingMethod": "Latin"
```

```

},
"uncertainVariables": {
  "Year1": {
    "formula": "PsiMetalog(,,{0.137088248307783,235.996630065143,-
41.6761363388904,-943.412342399359,166.781999403805,-
1781.91992354393,5864.72184371948,265.6771517599,-
850.31248220655,3796.63619041443,-8596.89170408249,-
404.07730274836,835.784252876499,-1707.57994881272})"
  },
  "Year2": {
    "formula": "PsiMetalog(,,{0.137088248307783,235.996630065143,-
41.6761363388904,-943.412342399359,166.781999403805,-
1781.91992354393,5864.72184371948,265.6771517599,-
850.31248220655,3796.63619041443,-8596.89170408249,-
404.07730274836,835.784252876499,-1707.57994881272})"
  },
  "Year3": {
    "formula": "PsiMetalog(,,{0.137088248307783,235.996630065143,-
41.6761363388904,-943.412342399359,166.781999403805,-
1781.91992354393,5864.72184371948,265.6771517599,-
850.31248220655,3796.63619041443,-8596.89170408249,-
404.07730274836,835.784252876499,-1707.57994881272})"
  },
  "Year4": {
    "formula": "PsiMetalog(,,{0.137088248307783,235.996630065143,-
41.6761363388904,-943.412342399359,166.781999403805,-
1781.91992354393,5864.72184371948,265.6771517599,-
850.31248220655,3796.63619041443,-8596.89170408249,-
404.07730274836,835.784252876499,-1707.57994881272})"
  },
  "Year5": {
    "formula": "PsiMetalog(,,{0.137088248307783,235.996630065143,-
41.6761363388904,-943.412342399359,166.781999403805,-
1781.91992354393,5864.72184371948,265.6771517599,-
850.31248220655,3796.63619041443,-8596.89170408249,-
404.07730274836,835.784252876499,-1707.57994881272})"
  },
  "Year6": {
    "formula": "PsiMetalog(,,{0.137088248307783,235.996630065143,-
41.6761363388904,-943.412342399359,166.781999403805,-
1781.91992354393,5864.72184371948,265.6771517599,-
850.31248220655,3796.63619041443,-8596.89170408249,-
404.07730274836,835.784252876499,-1707.57994881272})"
  },
  ...

```

Rather than risk an input error when entering in each year's formula (there are 24!), the PsiLibSip() function can be used to refer to the Probability Model already POSTed to the RASON Server. Now, users can be confident that the uncertain variables used in the RASON model are identical to the uncertain variables used in the Analytic Solver model.

In the example snippet below, the PsiLibSip function has replaced the PsiMetalog distribution function. Using the naming convention defined above, the argument must be of the form: modelName.SIPName. In this instance the model name is "CollegeFundGrowth1" and the SIP name is \$O\$17, thus the example uses CollegeFundGrowth1.\$O\$17 for the argument to PsiLibSip().

```
{
  "modelName": "CollegeFundGrowth1",
  "modelDescription": "Emulates a college fund savings model.",
  "modelSettings": {
    "randomGenerator": "Lecuyer",
    "samplingMethod": "Latin"
  },

  "uncertainVariables": {
    "Year1": {
      "formula": "PsiLibSip('CollegeFundGrowth1.$O$17') "
    },
    "Year2": {
      "formula": "PsiLibSip('CollegeFundGrowth1.$O$17') "
    },
    "Year3": {
      "formula": "PsiLibSip('CollegeFundGrowth1.$O$17') "
    },
    "Year4": {
      "formula": "PsiLibSip('CollegeFundGrowth1.$O$17') "
    },
    "Year5": {
      "formula": "PsiLibSip('CollegeFundGrowth1.$O$17') "
    },
    "Year6": {
      "formula": "PsiLibSip('CollegeFundGrowth1.$O$17') "
    },
    ...
  }
}
```

Note: This example uses the same SIP (CollegeFundGrowth1.O17) to model each year's return. However, it would be possible for each uncertain variable to utilize a different SIP either from the same Probability Model or even from a different Probability Model.

If anything changes with the Metalog distribution, the user can simply update the Probability Model on the RASON server. As long as neither the modelName or SIPName as been modified, no further adjustments to the RASON model will be required.

Sensitivity Analysis and Model Parameters

Introduction

RASON includes facilities for sensitivity analysis of your model that can be used before even starting an optimization or simulation model. It is especially easy to identify the model parameters with the most impact on your computed results – you can simply append the "identParams" property to *any formula* to quickly find the input cells with the greatest impact on this formula.

You can choose some of these input cells to serve as Sensitivity Parameters, and then produce results that show the impact on computed results of varying these parameters over a range you specify. You can also turn these parameters into (or define other cells as) Simulation Parameters or Optimization Parameters and produce simulation and optimization results as your parameters are automatically varied.

Note: Sensitivity Analysis is currently not supported within a workflow.

Parameter Identification

The model below is similar to the UGYieldManagement1.json model discussed in the previous chapter, except that this model does not contain any uncertain variables or uncertain functions. The data section contains constant (unchanging) parameters such as 1. The price of a ticket (price), 2. The capacity (or number of seats) of the plane (capacity), 3. The number of tickets sold (sold), 4. The refund percentage for a customer who misses the flight (refund_no_shows) and 5. The percentage of compensation a passenger must receive if "bumped" or removed from the flight (refund_overbook). The formulas section contains the computations to 1. calculate the number of passengers who do not show up for a flight (no_shows), 2. number of customers who arrive to board the flight (show_ups), 3. number of overbooked passengers (overbook) and 4. total revenue.

To identify which model parameters have the most impact on the computed results, simply append the identParams:[] property to the formula as shown below for the revenue formula. Note: The identParams:[] property may only appear once within a RASON model.

```
{
  "modelName": "identParamsExample",
  "modelType": "calculation",
  "modelDescription": "Ident Params calculation example",
  "modelSettings": {
    "sensitivityPercent": 10
  },
  "data": {
```

```

    "price": {
      "value": 200
    },
    "capacity": {
      "value": 100
    },
    "sold": {
      "value": 111
    },
    "refund_no_shows": {
      "value": 0.5
    },
    "refund_overbook": {
      "value": 1.25
    }
  },
  "formulas": {
    "no_shows": {
      "formula": "0.1*sold"
    },
    "show_ups": {
      "formula": "sold - Round(no_shows, 0)"
    },
    "overbook": {
      "formula": "Max(0, show_ups - capacity)"
    },
    "revenue": {
      "formula": "price*(sold - refund_no_shows*Round(no_shows, 0) - refund_overbook*overbook)",
      "identParams": []
    }
  }
}

```

To quickly find the inputs (or parameters) with the greatest impact on this formula, use the REST API Endpoint **POST rason.net/api/model** to post the model to the RASON server and then use **POST rason.net/api/model/{nameorid}/decision** endpoint to calculate the RASON model.

Note: Sensitivity Analysis is currently not supported in a workflow. As a result, in order to use the POST rason.net/api/model/{nameorid}/solve endpoint to perform the sensitivity analysis, the query parameter `response-format=standalone` must be used. If using the Editor on RASON.com, simply enter `?response-format=standalone` into the Query Parameters field on the right of the screen. If calling from your own application, use `POST rason.net/api/model/{nameorid}/solve?response-format=standalone`.

```

{
  "status": {
    "code": 0,
    "id": "2590+identParamsExample+2020-04-24-21-23-05-982452",
    "codeText": "Solver has completed the calculation."
  },
  "observations": {
    "revenue": {
      "value": 21100
    }
  },
  "identParams": {
    "revenue": {
      "value": 21100,

```

```

    "paramNames": ["sold", "capacity", "price", "refund_no_shows",
    "refund_overbook"],
    "paramLower": [99.9, 90, 180, 0.45, 1.125],
    "paramUpper": [122.1, 110, 220, 0.55, 1.375],
    "lower": [18980, 18600, 18990, 21210, 21100],
    "upper": [20695, 21100, 23210, 20990, 21100]
  }
}

```

The identParams results may be found under "identParams" section in the results. Here, we see that the final value for the revenue formula based on the data is \$21,100 ("value": 21000). The array "paramNames" holds the name of the five parameters defined in the data section of the RASON model: sold, capacity, price, refund_no_shows, refund_overbook. The remaining results (paramLower, paramUpper, lower and upper) will be reported in this order.

Using the value under "observations" coupled with the "lower" and "upper" properties under identParams, we can calculate the positive or negative impact each parameter has on the revenue formula. The value listed under observations is the reference value with which we will gauge the impact. (Notice that the observations value is equal to the final formula value.)

To calculate the negative impact of the parameter "sold", calculate the difference between the "observations" value and the lower value, i.e. $21100 - 18980 = 2,120$. In other words, if we decrease the value of the sold parameter by 10%, revenue will decrease by \$2,120.

Notice that the "upper" value for the sold parameter is 20,695 which is actually lower than the reference "observations" value of 21100. This means that as the number of tickets sold is increased by 10%, revenue does not increase, but rather decreases by \$405. This is because as the number of tickets sold increases from 111, the number of overbooked passengers increases which decreases the total revenue. (You can confirm this for yourself by simply changing the parameter for "sold" from 111 to 115 in the RASON model, updating the model on the RASON server by clicking `PUT rason.net/api/model/{nameorid}` and then re-calculating the model using `POST rason.net/api/model/{nameorid}/decision`.)

The paramLower and paramUpper results show the lower value and upper value that were used for each input when computing the formula value. A formula is set to $-n\%$ from its current value for paramLower and $+n\%$ from its current value for paramUpper, where $n\%$ (initially 10%) is specified for the model setting, sensitivityPercent.

Using Sensitivity Parameters

The Parameters Identification feature, described above performs a simple kind of sensitivity analysis; in some situations, this may be all that you need. But it is often useful to document the sensitivity of some computed result to systematic variations in the values of one or more parameters. RASON Services can produce a report to document these results.

How Parameters are Varied

RASON Services can automatically vary your parameters' values in two ways:

1. Varying **all parameters simultaneously**, from their respective lower to upper limits, for the number of steps that you specify. This can be performed with one or more PsiSenParams and one or more result cells.
2. Varying **two parameters independently**, from their lower to upper limits, computing a result for all combinations of the two parameters. This can be performed with exactly one result function (uncertain function) and *exactly two* PsiSenParams().

Varying All Parameters Simultaneously

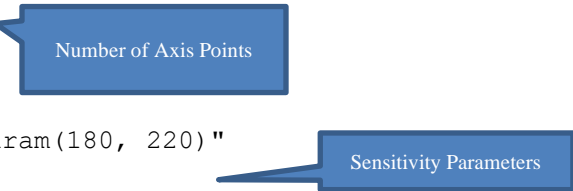
When creating a sensitivity analysis report by varying *all* parameters simultaneously, you'll first need to define *at least* one `PsiSenParam` function within the "parameters" section of the RASON model. In this example, two parameters (price and sold) are created within "parameters". The "price" parameter can be varied between 180 and 220 and the "sold" parameter can be varied between 99 and 121.

Note that `modelType` is equal to decision. A calculation will be performed, not an optimization or simulation.

Under `modelSettings`, "sensitivityPoints": 11 varies both the "price" and "sold" parameters by 11 axis points. This means that 11 calculations will be performed. Increments are calculated using the formula: (upper bound – lower bound)/(axis points – 1). As a result, "price" will be varied from 180 to 220, incrementing by 4 units (180, 184, 188, 192, etc.) and "sold" will be varied from 99 to 121 incrementing by 2.2 units.

Note, it's also possible to supply the exact values that you want the parameter to equal by using the formula syntax: price: { formula: "PsiSenParam({val1, val2, val3, val4})" }. In this case, if "sensitivityPoints" is set to 4, the price parameter will be set equal to only the values provided in the list. If the number of axis points is greater than the number of values supplied, the values used in the sensitivity analysis will repeat, i.e. val1, val2, val3, val4, val1, val2, etc.

```
{
  "modelName": "simultaneousVariation",
  "modelType": "decision",
  "comment": "Simultaneous variation of any params",
  "modelSettings": {
    "sensitivityPoints": 11
  },
  "parameters": {
    "price": {
      "formula": "PsiSenParam(180, 220)"
    },
    "sold": {
      "formula": "PsiSenParam(99, 121)"
    }
  },
  "data": {
    "capacity": {
      "value": [100, 120, 130]
    },
    "refund_no_shows": {
      "value": 0.5
    },
    "refund_overbook": {
      "value": 1.25
    }
  },
  "formulas": {
    "no_shows": {
      "formula": "0.1*sold"
    },
    "show_ups": {
      "formula": "sold - Round(no_shows, 0)"
    },
    "overbook": {
      "formula": "Max(0, show_ups - capacity[1])"
    },
    "revenue": {
      "formula": "price*(sold - refund_no_shows*Round(no_shows, 0) - refund_overbook*overbook)",
    }
  }
}
```



```

        "plotParams": []
    }
}

```

The keyword plotparams must be added to the formula definition for 1 (and only 1) result function.

To Post the model and create the sensitivity report, use the REST API Endpoint **POST rason.net/api/model** to post the model to the RASON server and then use **POST rason.net/api/model/{nameorid}/decision** endpoint to calculate the RASON model. To use the POST **~/solve** endpoint to perform the sensitivity analysis, see the note in the Parameter Identification section, above.

```

{
  "status": {
    "code": 0,
    "id": "2590+SensitivityCalculationExampleIndependently+2020-10-19-21-22-16-475993",
    "codeText": "Solver has completed the calculation."
  },
  "observations": {
    "revenue": {
      "value": 20100
    }
  },
  "plotParams": {
    "revenue": {
      "paramNames": ["price", "sold"],
      "paramLower": [180, 99],
      "paramUpper": [220, 121],
      "value": [
        17100, 18160.8, 18687.2, 18163.2, 20148.8, 19700,
        21154.8, 21455.2, 22705.2, 22042.8, 23155
      ]
    }
  }
}

```

Varying Two Parameters Independently

When creating a sensitivity analysis report by varying *two* parameters independently, you'll first need to define two (and only two) PsiSenParams within the "parameters" section of the RASON model. In this example, two parameters (price and sold) are created within "parameters". The "price" parameter can be varied between 180 and 220 and the "sold" parameter can be varied between 99 and 121.

Note that modelType is "decision". RASON Services will be performing a calculation rather than a simulation or optimization.

Under modelSettings, "sensitivityPoints": 11 varies the "price" parameter by 11 axis points and "sensitivityPoints2": 11 varies the "sold" parameter by 11 axis points. This means that 121 calculations will be performed. Increments are calculated using the formula: $(\text{upper bound} - \text{lower bound}) / (\text{axis points} - 1)$. As a result:

- "price" will be varied from 180 to 220, incrementing by 4 units (180, 184, 188, 192, etc.) calculated by $[(220-180)/(11-1)]$.
- "sold" will be varied from 99 to 121, incrementing by 2.2 units (99, 101.2, 103.4, etc) calculated by $[(121-99)/(11-1)]$.

Notice the keyword "plotParams" has been added to the revenue formula under in the "formulas" section of the RASON model. The keyword "plotParams" denotes the required result function.

```

{

```

Model Type is "decision".

```

modelName: "SensitivityCalculationExample",
modelType: "decision",
modelSettings: { sensitivityPoints: 11, sensitivityPoints2: 11 },
parameters: {
  price: { formula: "PsiSenParam(180, 220)" },
  sold: { formula: "PsiSenParam(99, 121)" }
},
data: {
  capacity: { value: 100 },
  refund_no_shows: { value: 0.5 },
  refund_overbook: { value: 1.25 }
},
formulas: {
  no_shows: { formula: "0.1*sold" },
  show_ups: { formula: "sold - Round(no_shows, 0)" },
  overbook: { formula: "Max(0, show_ups - capacity)" },
  revenue: { formula: "price*(sold - refund_no_shows*Round(no_shows, 0) - refund_overbook*overbook)", plotParams: [] }
}
}

```

Axis Points

Sensitivity Parameters

The keyword plotParams must be added to the formula definition for 1 (and only 1) result function.

To Post the model and create the sensitivity report, use the REST API Endpoint **POST rason.net/api/model** to post the model to the RASON server and then use **POST rason.net/api/model/{nameorid}/decision** endpoint to calculate the RASON model. To use the POST ~/solve endpoint to perform the sensitivity analysis, see the note in the Parameter Identification section, above.

```

{
  "status": {
    "code": 0,
    "id": "2590+SensitivityCalc+2020-09-30-21-30-12-345197",
    "codeText": "Solver has completed the calculation."
  },
  "observations": {
    "revenue": {
      "value": 20900
    }
  },
  "plotParams": {
    "revenue": {
      "paramNames": ["price", "sold"],
      "paramLower": [180, 99],
      "paramUpper": [220, 121],
      "value": [
        16920, 17296, 17672, 18048, 18424, 18800, 19176, 19552,
        19928, 20304, 20680],
        17316, 17700.8, 18085.6, 18470.4, 18855.2, 1240,
        19624.8, 20009.6, 20394.4, 20779.2, 21164],
        17712, 18105.6, 18499.2, 18892.8, 19286.4, 19680,
        20073.6, 20467.2, 20860.8, 21254.4, 21648],
        18018, 18418.4, 18818.8, 19219.2, 19619.6, 20020,
        20420.4, 20820.8, 21221.2, 21621.6, 22022],
        18414, 18823.2, 19232.4, 19641.6, 20050.8, 20460,
        20869.2, 21278.4, 21687.6, 22096.8, 22506],
        18810, 19228, 19646, 20064, 20482, 20900, 21318, 21736,
        22154, 22572, 22990],
        18936, 19356.8, 19777.6, 20198.4, 20619.2, 21040,

```

Sold = 99 and Price varied from 180 to 220

Sold = 103.4 and Price varied from 180 to 220

Sold =121 and Price varied from 180 to 220

21460.8, 21881.6, 22302.4, 22723.2, 23144],
 [18837, 19255.6, 19674.2, 20092.8, 20511.4, 20930,
 21348.6, 21767.2, 22185.8, 22604.4, 23023],
 [18873, 19292.4, 19711.8, 20131.2, 20550.6, 20970,
 21389.4, 21808.8, 22228.2, 22647.6, 23067],
 [18774, 19191.2, 19608.4, 20025.6, 20442.8, 20860,
 21277.2, 21694.4, 22111.6, 22528.8, 22946],
 [18675, 19090, 19505, 19920, 20335, 20750, 21165, 21580,
 21995, 22410, 22825]

]
 }
 }
 }

The results can be organized into the following chart.

	99	101.2	103.4	105.6	107.8	110	112.2	114.4	116.6	118.8	121
\$180.00	\$16,920.00	\$17,316.00	\$17,712.00	\$18,018.00	\$18,414.00	\$18,810.00	\$18,936.00	\$18,837.00	\$18,873.00	\$18,774.00	\$18,675.00
\$184.00	\$17,296.00	\$17,700.80	\$18,105.60	\$18,418.40	\$18,823.20	\$19,228.00	\$19,356.80	\$19,255.60	\$19,292.40	\$19,191.20	\$19,090.00
\$188.00	\$17,672.00	\$18,085.60	\$18,499.20	\$18,818.80	\$19,232.40	\$19,646.00	\$19,777.60	\$19,674.20	\$19,711.80	\$19,608.40	\$19,505.00
\$192.00	\$18,048.00	\$18,470.40	\$18,892.80	\$19,219.20	\$19,641.60	\$20,064.00	\$20,198.40	\$20,092.80	\$20,131.20	\$20,025.60	\$19,920.00
\$196.00	\$18,424.00	\$18,855.20	\$19,286.40	\$19,619.60	\$20,050.80	\$20,482.00	\$20,619.20	\$20,511.40	\$20,550.60	\$20,442.80	\$20,335.00
\$200.00	\$18,800.00	\$19,240.00	\$19,680.00	\$20,020.00	\$20,460.00	\$20,900.00	\$21,040.00	\$20,930.00	\$20,970.00	\$20,860.00	\$20,750.00
\$204.00	\$19,176.00	\$19,624.80	\$20,073.60	\$20,420.40	\$20,869.20	\$21,318.00	\$21,460.80	\$21,348.60	\$21,389.40	\$21,277.20	\$21,165.00
\$208.00	\$19,552.00	\$20,009.60	\$20,467.20	\$20,820.80	\$21,278.40	\$21,736.00	\$21,881.60	\$21,767.20	\$21,808.80	\$21,694.40	\$21,580.00
\$212.00	\$19,928.00	\$20,394.40	\$20,860.80	\$21,221.20	\$21,687.60	\$22,154.00	\$22,302.40	\$22,185.80	\$22,228.20	\$22,111.60	\$21,995.00
\$216.00	\$20,304.00	\$20,779.20	\$21,254.40	\$21,621.60	\$22,096.80	\$22,572.00	\$22,723.20	\$22,604.40	\$22,647.60	\$22,528.80	\$22,410.00
\$220.00	\$20,680.00	\$21,164.00	\$21,648.00	\$22,022.00	\$22,506.00	\$22,990.00	\$23,144.00	\$23,023.00	\$23,067.00	\$22,946.00	\$22,825.00

Defining Your Stochastic Optimization Model

Introduction

This chapter takes you step-by-step through the process of creating a stochastic optimization model that can be solved using stochastic programming or robust optimization.

Defining a Stochastic Optimization Model

This chapter teaches you how to model stochastic optimization models using the RASON Modeling Language. This guide will give you step-by-step instructions on how to create and solve a stochastic optimization model but if you'd like to open the example and follow along, you can do so by clicking RASON examples on the ribbon, then *Example models discussed in RASON User Guide* - UGProjectSelect0.json.

We use the term stochastic optimization to mean optimization of models that include *uncertainty*, using any solution method. Solver SDK Platform and the RASON Modeling Language offer an exceptional level of power to find robust optimal solutions to models with uncertainty, using three different solution methods:

- Simulation optimization
- Stochastic programming
- Robust optimization

The first method, **simulation optimization**, uses the Evolutionary engine to handle very general models, but it is not scalable to large models (with thousands of variables and constraints), and it doesn't support the important modeling concept of recourse decisions. We used this method in the section above to solve the UGYieldManagement3.json model.

Stochastic programming and **robust optimization** can be applied only to linear and quadratic programming models with uncertainty, but they are scalable to large models.

A Project Selection Model

Our next example is a capital budgeting problem, where the projects being considered for funding have uncertain future cash flows. First, we will use simulation optimization to find a solution to the model using the Evolutionary Engine. Afterwards, we will use calls to the RASON Interpreter to analyze the structure of the model, transform the model into a linear model and solve again using the LP/Quadratic engine.

In this example, eight different capital projects are proposed for funding. Each one has a known initial investment. Each project has a 90% chance of success, and if a project succeeds, it will have an uncertain (but positive) future cash flow. Funding all eight projects would require a total initial investment of \$2.5 million, but our capital budget is only \$1.5 million. Hence, we must choose a subset of the projects to fund that will maximize our expected total future cash flow, while ensuring that our total initial investment doesn't exceed our \$1.5 million budget.

To model the uncertainty in this problem, we use PSI functions that define *uncertain variables* with probability distributions within the `uncertainVariables` section. To model the cash flow if a project is successful, we first create the uncertain variable `successCashFlow` with 8 elements then populate this array using the `PsiTriangular()` function, specifying a minimum, most likely and maximum cash flow for each of the 8 projects.

The `x` array will hold our 0-1 (binary) decision variables. If a variable is set to 1, the project is selected. If a variable is set to 0, the project is not selected.

To model the chance of success we create the variable `successProbability` (within formulas) using the probability distribution, `=PsiBinomial(1,0.9)`. On each trial, this distribution returns 1 with probability 90% and 0 with probability 10%.

In formulas, cash multiplies the net cash flow from each project, expected cash flow minus the initial investment, by the `x` array which holds our binary decision variables. Inside `constraints`, `invest` calculates the constraint that ensures that the total initial investment does not exceed \$1,500,000, (`initialInvest` multiplied by the 0-1 decision variables in the `x` array).

The objective maximizes the expected mean of cash by setting the `chanceType` property to `ExpVal`.

```
objective: {
  total: {
    type: "maximize",
    formula: "sumproduct(successCashFlow * successProbability -
      initialInvest, x)",
    chanceType: "ExpVal",
    finalValue: []
  }
}
```

Inside of `modelSettings` we set `simulationOptimization: True`.

```
modelSettings: {
  simulationOptimization: "true",
  numTrials: "1000",
  randomSeed: "1"
},
```

To solve via simulation optimization, the model option `simulationOptimization` must be set to `True` within `modelSettings`, and the Evolutionary engine must be selected within `engineSettings`. The idea behind simulation optimization is straightforward: **For each** set of values for the decision variables considered by the optimizer, we perform **one simulation**, a compute user-specified *summary measure* – such as `PsiMean(cash)` in the Project Selection model – for the constraints and/or objective that depend on uncertainty. The optimizer uses these summary measures to decide what set of values it should try *next* for the decision variables – and the process is repeated. The great strength of simulation optimization is its **generality** – but this is also its weakness: It requires a new simulation **at each step** of the optimization, and because the method assumes no structure in the model, in general the **number of steps** can grow **exponentially** with the number of variables and constraints.

Note that `modelType` is set to "optimization", rather than "simulation". If `modelType=simulation` and `POST rason.net/api/model/{nameorid}/solve` is called to solve a stochastic optimization model, the workflow engine will run a simulation, only. In order to solve the model using stochastic optimization, you must specify `"modelType" = "optimization"`.

This model formulation will find the best combination of projects – by finding 0 or 1 values for the selection variables in the `x` array – to maximize the expected value of total net cash flow, subject to the constraint that the total initial investment doesn't exceed our \$1.5 million budget. The full model formulation is below.

```
{
  modelName: "UGProjectSelect0",
  modelType: "optimization",
  modelSettings: {
```

```

    simulationOptimization: "true",
    numTrials: "1000",
    randomSeed: "1"
  },
  engineSettings: {
    engine: "Evolutionary",
    randomSeed: 1
  },
  variables: {
    x: {
      dimensions: [8],
      type: "binary",
      finalValue: []
    }
  },
  uncertainVariables: {
    successCashFlow: {
      dimensions: [8]
    },
    "successCashFlow[1]": {
      formula: "PsiTriangular(400000, 500000, 900000)"
    },
    "successCashFlow[2]": {
      formula: "PsiTriangular(500000, 750000, 1250000)"
    },
    "successCashFlow[3]": {
      formula: "PsiTriangular(500000, 1000000, 1500000)"
    },
    "successCashFlow[4]": {
      formula: "PsiTriangular(400000, 600000, 900000)"
    },
    "successCashFlow[5]": {
      formula: "PsiTriangular(250000, 500000, 750000)"
    },
    "successCashFlow[6]": {
      formula: "PsiTriangular(300000, 500000, 600000)"
    },
    "successCashFlow[7]": {
      formula: "PsiTriangular(200000, 450000, 700000)"
    },
    "successCashFlow[8]": {
      formula: "PsiTriangular(400000, 500000, 700000)"
    },
    "successProbability": {
      dimensions: [8],
      formula: "PsiBinomial(1, 0.9)"
    }
  },
  formulas: {
    initialInvest: {
      value: [325000, 450000, 550000, 300000, 150000, 250000, 150000, 325000]
    },
    cash: {
      formula: "sumproduct(successCashFlow * successProbability - initialInvest, x)"
    }
  }
}

```

```

    },
    constraints: {
      invest: {
        formula: "sumproduct(initialInvest, x)",
        upper: 1500000
      }
    },
    objective: {
      total: {
        type: maximize,
        formula: cash,
        chanceType: ExpVal,
        finalValue: []
      }
    }
  }
}

```

Solving with Simulation Optimization

We'll first solve this problem using **simulation optimization** using either the Desktop IDE or the Web IDE. If using the Desktop IDE, simply click the Solve icon at the top of the application.

If using the Web IDE,

- Click **POST rason.net/api/model** to call the endpoint `POST rason.net/api/model` to post the model.
- Click **POST rason.net/api/model/id/optimize**, or **/solve**, to start the simulation optimization.
- Click **GET rason.net/api/model/id/status** to obtain the status of the solve.
- Click **GET rason.net/api/model/id/result** to obtain the final result.

The results are shown below. Note: All RASON arrays are 1-based.

```

{
  "status": {
    "code": 2,
    "id": "2590+UGProjectSelect0+2020-03-20-14-44-27-831535",
    "codeText": "Solver cannot improve the current solution. All constraints are satisfied."
  },
  "variables": {
    "x": {
      "finalValue": [1, 0, 1, 1, 1, 0, 1, 0]
    }
  },
  "objective": {
    "total": {
      "finalValue": 1.39564e+06
    }
  }
}

```

We've decided to fund projects #1, 3, 4, 5 and 7 for an expected total net cash flow of \$1.36808M. That's a great result but for a 'scaled-up' model that might involve hundreds or thousands of projects, the solution might have taken far more time to solve.

Stochastic Transformation using Deterministic Equivalent

As we discussed above, there are two additional methods besides simulation optimization that can solve Stochastic LPs – stochastic programming and robust optimization. For this model, we will choose a transformation to Stochastic Programming Deterministic Equivalent form by replacing `simulationOptimization: True` with `transformStochastic: deterministicEquivalent`.

The `modelSettings` section changes to ...

```
"modelSettings": {
  "transformStochastic": "deterministicEquivalent",
  "numtrials": 1000
},
```

If using the Desktop IDE, simply click the Solve icon at the top of the application.

If using the Web IDE,

- Click POST `rason.net/api/model` to post the model.
- Click POST `rason.net/api/model/id/optimize` (or `/solve`) to begin the simulation optimization.
- Click GET `rason.net/api/model/id/status` to obtain the status of the solve.
- Click GET `rason.net/api/model/id/result` to obtain the final result.

The results are shown below.

```
{
  "status": {
    "code": 0,
    "id": "2590+UGProjectSelect0+2020-03-25-14-18-50-873215",
    "codeText": "Solver found a solution. All constraints and \
optimality conditions are satisfied."
  },
  "variables": {
    "x": {
      "finalValue": [1, 0, 1, 1, 1, 0, 1, 0]
    }
  },
  "objective": {
    "total": {
      "finalValue": 1.39564e+06
    }
  }
}
```

In a *fraction of a second*, a solution appears with a slightly better objective value of \$1.39564M and the same projects selected, with the message “Solver found a solution. All constraints and optimality conditions are satisfied.” This means that Analytic Solver Platform found a *proven* globally optimal solution – whereas with simulation optimization, we never know whether the solution we found is optimal. To use the robust counterpart method solve this model, simply replace `"deterministicEquivalent"` within `modelSettings` to `"robustCounterpart"`.

```
"modelSettings": {
  "transformStochastic": "robustCounterpart",
  "numtrials": "1000"
},
```

Defining Your Data Science Model

Introduction

RASON® DM is a comprehensive data science **modeling language**. Data science is a discovery-driven data analysis technology used for identifying patterns and relationships in data sets. With overwhelming amounts of data now available from transaction systems and external data sources, organizations are presented with increasing opportunities to understand and gain insights into their data. Data science is still an emerging field, and is a convergence of Statistics, Machine Learning and Artificial Intelligence.

Often, there may be more than one approach to a problem. RASON DM provides a high-level API “tool belt” that offers a variety of methods to analyze data. It has extensive coverage of statistical and machine learning techniques for classification, regression, affinity analysis, data exploration and reduction.

The worlds of commerce, research and government are huge and varied. No single data analysis pattern can possibly be right for everyone. RASON DM provides a fast, solid and well-tested foundation on which organizations can build and execute data analysis tasks to suit their needs precisely. RASON DM focuses on Data science tasks and provides robust implementations of industry-standard Data science algorithms.

This chapter assumes that you're familiar with data science methods and techniques, but you've never used the RASON Modeling language to define such a problem. This chapter begins with a discussion of the different components that comprise a RASON model and then moves on to several example models illustrating how to sample from a dataset, create training and validation partitions, perform feature selection and fit a classification model. For more specific information such as what types of options are available for each data science method, see the RASON Reference Guide.

Supported Features

RASON supports all facets of the data science process, including data exploration and transformation, visualization, feature selection, text mining, time series forecasting, affinity analysis and unsupervised and supervised learning.

- Featuring an innovative (and patent pending) new capability for automated risk analysis of machine learning models. A further benefit of this feature is a general-purpose, easy to use new tool for synthetic data generation, to augment the data you already have.
- Data may be acquired from any source (file system, Web, data streams, databases, Big Data sources, ODATA, generic ODBC connection or specific DB connections for MS access, MS SQL and Oracle) using any tool available for your desired programming language. Exception reporting helps you to quickly find and fix problems with your model and numerous built in output and report capabilities help you to evaluate its performance.
- Clean and transform your data with a comprehensive set of data handling utilities including categorizing data and handling missing values. Use Principal Components Analysis to reduce columns and K-Means Clustering or Hierarchical Clustering to group data by rows.

- The following transformation utilities are currently available:
 - Factorizing
 - Binning
 - Missing values
 - One-hot-encoding (Transforming Categorical Features)
 - Reducing Categories
 - Rescaling
 - PCA
 - Sampling
 - Canonical Variate Analysis (CVA)
 - Summarization
 - Synthetic Data Generation – An innovative (and patent pending) capability for automated risk analysis of machine learning models.
- You can clusterize your data into a set of cohesive groups using:
 - k-Means Clustering
 - Hierarchical Clustering
- Automatically transform free-form text into structured data, identify the most frequently occurring terms and extract key concepts with latent semantic indexing. RASON Data Science Text Miner supports:
 - TF-IDF vectorization
 - Latent Semantic Analysis
- Apply the most popular exponential smoothing and Box-Jenkins ARIMA methods, with seasonality, to forecast time series, such as sales and inventory, from historical data. The following time series analysis tools are supported in RASON DM:
 - Lag Analysis
 - ARIMA
 - Exponential, Double Exponential, Moving Average and Holt Winters Smoothing
- Easily partition your data into training, validation and test datasets, with no limits on dataset size.
- Use feature selection to automatically identify columns or variables with the greatest explanatory power for your desired classification or regression task.
- Use powerful Multiple Linear Regression with variable selection and other supervised regression algorithms including Ensemble methods (using any regression engine as a weak learner).
- RASON DM includes four regression algorithms and three different ensemble methods: Bagging, Boosting and Random Trees. Boosting and Bagging ensemble methods can use any prediction method as a base learner. RASON DM provides extensive functionality for evaluating the performance of supervised models, including the goodness of fit metrics and charts (Lift Charts, Gain Charts, Decile Tables, RROC curves).
 - Multiple Linear Regression
 - k-Nearest Neighbors
 - Regression Trees

- Neural Networks
- Use classical Discriminant Analysis and Logistic Regression and other supervised classification algorithms, including Ensemble methods (using any classification engine as a weak learner).
- RASON DM includes six classification algorithms and three different ensemble methods. Boosting and Bagging ensemble methods can use any classification method as a base learner. RASON DM provides extensive functionality for evaluating the performance of supervised models, including the goodness of fit metrics and charts (Lift Charts, Gain Charts, Decile Tables, ROC curves).
 - Discriminant Analysis
 - k-Nearest Neighbors
 - Logistic Regression
 - Classification Trees
 - Naïve Bayes
 - Neural Networks
- Use Affinity Analysis to discover association rules and perform market basket analysis.
- Automatic POSTing of fitted models, in PMML and JSON format, to the RASON Server, as well as fitted model export/import, for all supported models:
 - Regression Models (Linear and Logistic)
 - Decision Trees (Classification and Regression)
 - Neural Networks (Classification and Regression)
 - k-Nearest Neighbors (Classification and Regression)
 - Discriminant Analysis
 - Naïve Bayes
 - Random Trees (Classification and Regression)
 - Ensemble Bagging (Classification and Regression)
 - Ensemble Boosting (Classification and Regression)
 - Clustering
 - Time Series
 - Transformations
 - Text Mining
- Feature-parity and model interoperability between XLMinerSDK, Analytic Solver Data Science and RASON Data Science.

Defining a Data Science Model

This chapter teaches you how to model data science models using the RASON Modeling Language by giving step-by-step instructions on how to create and solve various data science models. If you'd like to open the examples and follow along, you can do so on the Editor page on www.RASON.com under RASON Examples – Data Science.

All algorithms featured in Analytic Solver and XLMiner SDK can be expressed using a standardized structure in RASON DM. This basic structure includes four major "sections" or "segments": datasources, datasets,

estimator/transformer and actions. While both model Name and modelType are optional high-level properties, both are strongly recommended.

```
{
  "modelName": "",
  "modelType": "",
  "datasources": {},
  "datasets": {},
  "estimator"/"transformer": {
    "type": "",
    "algorithm": "",
    "parameters": {}
  },
  "actions": {}
}
```

- modelName - RASON Decision Services supports named and unnamed models. A named model includes the modelName: "name" property in its text; "name" must be a string that can be URL encoded and must be unique among models within a user's account. Although this property is optional, it is strongly recommended that a model be named to allow for easy recognition and retrieval when multiple models, instances and versions exist in a user's account. This property also identifies the model within an inline decision flow and fitted model, if POSTed to the RASON Server. (Reusable model use the "invokeModel" field; which in turn invokes another standalone model containing the "modelName" property.)
- modelType –This top-level property defines the model type as optimization, simulation, datamining or calculation within the RASON script. (For more information on this property see the section *More on the modelType Property* within the **Creating and Running a Decision Flow** chapter.) This property is optional but strongly recommended.
- datasources - The "datasources" section is where the data for the model is acquired. Most times, the data is contained in an external data source such as a database or delimited file.
- datasets - The "datasets" section is where the external dataset is "bound" to a RASON dataset. Note: Data science methods can not be applied directly to a datasource. The datasource must first be "bound" to a dataset, then data science methods can be applied to the dataset.
- estimator/transformer – These sections are mutually exclusive - A model may contain **one** of either a transformer or an estimator, not both. An "estimator" object *estimates* a model from the training data and stores the fitted model, which may be used later. Examples of estimators are classification or regression algorithms. These algorithms fit a model which can be used later to score new data. A "transformer" applies to estimators that do not fit a model but rather transform data, such as Feature Selection or Sampling.
- actions - The function of the estimator or transformer is carried out within the "actions" section. If a model was "fit" within the estimator section, then the model is applied to the desired dataset (training, validation, test partitions or to new data) within this section. If a transformer was initiated, then the actual data transformation will be performed within this segment.

It is important to note that the order inside the RASON model is *very* important as the RASON interpreter does *not* parse the model to determine the correct order beforehand. Therefore, "actions" may not appear before "estimator", "estimator" may not appear before "datasets" and so on.

Continue reading to learn more about both the major and minor sections that may appear in a RASON DM model.

Essential RASON Data Science Model Sections

As mentioned in Defining a Data Science Model, there are four essential sections that must exist in a single RASON DM model. This portion of the User Guide describes each section, besides modelName and modelType, in more detail.

"datasources"

As mentioned above, this section is used to specify how the data will be acquired. Typically, data will be contained in an external data source such as a delimited file, Excel workbook, or database.

This section, "datasources", is an object with user defined attributes where each attribute defines an object with "type", "connection" and "direction" properties. The following example defines 3 data sources:

myTrainingData, myValidationData and myTestData.

```
"datasources": {
  "myTrainingData": {
    "type": "csv",
    "connection": "PathToDataFilesOrTrainingData.txt",
    "direction": "import"
  },
  "myValidationData": {
    "type": "csv",
    "connection": "PathToDataFilesOrValidationData.txt",
    "direction": "import"
  },
  "myTestData": {
    "type": "csv",
    "connection": "PathToDataFilesOrTestData.txt",
    "direction": "import"
  }
}
```

In this example code snippet, three data sources are initialized: myTrainingData, myValidationData and myTestData. The "type" property describes the file type of the data file being imported into the RASON model. In this case, the data for all three data sources is contained within a "CSV" file. The "connection" property describes the location of each data file and the "direction" property specifies whether the file is being imported or exported. The default for "direction" is "import".

Aside from "type" and "connection" properties, additional properties exist for specific types of data sources such as "headerExists" (or simply "header") for delimited files or "selection" for SQL database selection. For the full list of properties for the "datasource" section, see the RASON Reference Guide. For examples on how to import from various data sources, see both the RASON Reference Guide or the Editor page on RASON.com.

Note: RASON Decision Services makes it exceptionally easy to work with data sources in the Microsoft ecosystem, by creating a Data Connection on the user's My Account page on www.RASON.com. The RASON service supports the following data connections.

- OneDrive and OneDrive for Business
- Common Data Service for Dynamics 365, Power Apps and Power Automate
- OData and CDS support for Power BI
- CData Cloud Hub support for access to 100+enterprise data sources.

For more information on how to create and maintain Data Connections, see the previous Data Connections section within the chapter, RASON Services Web IDE.

"datasets"

The component, "datasets", is an object with user defined attributes where each attribute defines an object with a "binding" property. The following example defines 2 data sets: myTrainData and myValidData.

```
"datasets": {
  "myTrainData": {
    "binding": "myTrainSrc",
    "targetCol": "Y"
  },
  "myValidData": {
    "binding": "myValidSrc",
    "targetCol": "Y"
  }
},
```

In this example code snippet, two datasets are initialized, "myTrainData" and "myValidData". Within "myTrainData", the dataSource "myTrainSrc" is bound to the "myTrainData" dataset. Likewise, the dataSource "myValidSrc" is bound to the "myValidData" dataset.

The "binding" property specifies the data source to be bound. This attribute can be bound to the output of, or data sources in, other stages. "Binding" is not applicable if the user provides the data inline, i.e. enters data manually into the RASON model. For a list of all properties that may appear in a given data set definition, see the RASON Reference Guide.

"estimator"/"transformer"

The "estimator" object *estimates* a model from the training data and stores the fitted model, which may be used later. The "estimator" object implements the "fit" interface. The "transformer" object is used to differentiate the algorithms that do not have a model, i.e. they do not implement the "fit" interface. Rather, these algorithms implement the "transform" interface (only).

"estimator" The "estimator" section defines the estimator used to fit the model. Estimators extract a model from the input data. This model can be used in other RASON models using a dataset binding to the output. This element is mutually exclusive with the "transformer" element. Both may not appear in the same stage definition. (Only one transformer or estimator may appear in a given RASON DM model.) An example of the estimator "Find Best Model" is shown below.

```
"estimator": {
  "cfbmEstimator": {
    "type": "classification",
    "algorithm": "findBestModel",
  }
},
```

In this example, a new estimator, cfbmEstimator, is initialized. This estimator will perform run all available classification learners and will determine the learner with the "best fit" to the dataset based on user specifications. See the classification and regression examples below for more information.

Properties for "estimator" are:

- "type" – Must be one of the following: "classification", "regression", "clustering", "textMining", "transformation", "timeSeries".
- "algorithm" – The selection for this property varies with the selected "type". See the chart below to see which algorithms correspond to the selected "type".

Type	Algorithm Choice
"classification"	"findBestModel", "boosting", "bagging", "neuralNetwork", "decisionTree", "randomTrees", "nearestNeighbors", "naiveBayes", "discriminantAnalysis" or "logisticRegression"
"regression"	"findBestModel", "boosting", "bagging", "neuralNetwork", "decisionTree", "randomTrees", "nearestNeighbors", "linearRegression"
"clustering"	"kMeans" or "hierarchical"
"textMining"	"tfIdf" or "latentSemanticAnalysis"
"transformation"	"oneHotEncoding", "imputation", "rescaling", "principalComponentAnalysis", "binning", "factorization", "canonicalVariateAnalysis", "syntheticDataGenerator", "summarization"
"featureSelection"	"univariate", "linearWrapping" or "logisticWrapping"
"timeSeries"	"addHoltWinters", "mulHoltWinters", "noTrendHoltWinters", "doubleExponential", "exponential", "movingAverage", "arima" or "lagAnalysis"

- "parameters" – The property options for "parameters" will vary depending on the algorithm selected. For a complete list of properties for each algorithm, see the RASON Reference Guide.
- In order to run the synthetic data generator, described later in this chapter, "simulation":{} must be called within the estimator. All parameters applying to the synthetic data generator are passed within "simulation". For example:

```

"estimator": {
  "mlrEstimator": {
    "type": "regression",
    "algorithm": "linearRegression",
    "parameters": {
      "fitIntercept": true
    },
    "simulation": {
      "metalogAuto": true,
      "numMetalogTerms": [
        ["CRIM", 5],
        ["ZN", 5],
        ["INDUS", 5],
        ...
      ]
    }
  }
}

```

"transformer" A "transformer" applies to estimators that do not fit a model but rather transform data, such as Feature Selection or Sampling. Since no data is stored (i.e. transformers take data in and return data out), transformation algorithms are

represented by a single object. For example, when applying a sampling algorithm to a dataset, there is nothing to estimate from the training data which results in nothing to store in a model for future actions.

This element is mutually exclusive with the "estimator" element. Both may not appear in the same RASON model. (Only one transformer or estimator may appear in a given RASON DM model.) An example of the transformer "mySampler" (appearing in the Transformation - Sampling.json RASON example on RASON.com) is shown below.

```
"transformer": {
  "mySampler": {
    "type": "transformation",
    "algorithm": "sampling",
    "parameters": {
      "sampleSize": 4,
      "replaceOption": "false",
      "sortIndexes": "false",
      "seed": 123
    }
  }
},
```

In the example code snippet above, the transformer "mySampler" is initialized. This transformer will perform a "transformation" (type: transformation) using the sampling algorithm (algorithm: sampling). Four options, sampleSize, replaceOption, sortIndexes and seed, are specified.

The "transformer" objects are:

- "type" – Must be one of the following: "affinityAnalysis", "bigData", "featureSelection", or "transformation".
- "algorithm" – The selection for this property varies with the selected "type". See the chart below to see which algorithms correspond to the selected "type".

Type	Algorithm Choice
"affinityAnalysis"	"associationRules"
"bigData"	"sampling" or "summarization"
"transformation"	"sampling", "stratifiedSampling", "partitioning", "oversamplePartitioning" or "categoryReduction", "summarization", "syntheticDataGenerator"

- "parameters" – The property options for "parameters" will vary depending on the algorithm selected. For a complete list of properties for each algorithm, see the RASON Reference Guide.

"actions"

The estimator or transformer is applied to the data within the "actions" section. An example of the action "nnpModel" (RASON Example Models – Data Science – Regression – NeuralNetwork.json) is shown below.

```

actions: {
  "nnpModel": {
    "trainData": "myTrainData",
    "estimator": "lennpEstimator",
    "export": "json",
    "action": "fit",
    "evaluations": [
      "trainingLog",
      "neuronWeights",
      "numEpochsUsed",
      "trainingTime",
      "stoppingReason",
      "partitionCausedStopping"
    ]
  }
}

```

In the example code snippet above, the "nnpModel" action is initialized. The model created from the "nnpEstimator" (estimator: nnpEstimator) will be applied or "fit" (action: fit) to the "myTrainData" dataset. Several results or "evaluations" are requested in the final results: the training log (trainingLog), the neuron weights (neuronWeights), number of epochs (numEpochsUsed), the time spent training the model (trainingTime), the reason the algorithm stopped (stoppingReason) and the partition used to evaluate the performance of the algorithm (partitionCausedStopping).

Note the "export" property. This property posts the fitted model, in either JSON or PMML format, to the RASON Server under the "modelName" property setting. Replace "export" ("export": "json/pmml") property with "binding" property to export the fitted model contained within a JSON or XML file. If neither "export" or "binding" properties are included within "actions", then the fitted model will only be produced in-memory. An in-memory fitted model may be used in a decision flow. For more information on POSTing a fitted model to the RASON server or exporting a fitted model, see the section POSTing/Exporting Fitted Models, below.

A second example of an action for a transformer is below. Notice there is no keyword to replace "estimator" within "actions", as in the example above. When using a transformer, there's no estimator/model, so the actions can unambiguously refer to the transformer object only.

```

transformer: {
  mySampler: {
    type: 'transformation',
    algorithm: 'sampling',
    parameters: {
      sampleSize: 4,
      replaceOption: false,
      sortIndexes: false,
      seed: 123
    }
  }
},
actions: {
  sampleData: {
    data: 'myData',
    action: 'transform',
    export: 'json',
    evaluations: [
      'transformation'
    ]
  }
}

```

The following properties are available for the "actions" section:

- "trainData" – This property may be used interchangeably with the property, "data". In some algorithms, it is possible to provide both "trainData" and "validData" i.e. for classification and regression algorithms.
- "estimator" – This property is used to reference the estimator defined in the "estimator" object.
- "export" or "binding" – Use the "export" property to POST the fitted model to the RASON Server. Use "binding" to export the fitted model to a XML or JSON file. If neither property exists, then the fitted model will only be produced in-memory. For more information on POSTing a fitted model to the RASON server or exporting a fitted model, see the section POSTing/Exporting Fitted Models, below.

Notes:

- The following transformation methods do not generate a fitted model: sampling, partitioning and SQL transformation.
 - The following transformation methods only produce a fitted model in JSON format: categoryreduction, factorization, imputation, and principalcomponentsanalysis.
 - The following estimators do not generate a fitted model: Feature Selection (logisticAnalysis, linearWrapping, univariate)
 - The following estimators only produce a fitted model in JSON format: hierarchical and kmeans clustering.
 - The following estimator only produces a fitted model in PMML: affinityAnalysis.
- "fittedModel" – Used when scoring a model, this property is used to reference the model generated inside of the "model" object. For more information on scoring, see the example below.
- "action" - Valid values for this property are "fit", "predict", "transform", or "forecast". As the name suggests, "fit" fits the model given "estimator" and "trainData". The remaining options, "predict", "transform" and "forecast", apply the fitted model for further options on partitions or new data.
- "parameters" – The selection for this property depends on the "model" or "estimator" selected. If these options are directly applicable to the prediction/transformation/forecast of the data within this action specifically (i.e. the "successProbability" when classifying different datasets), you may use different values for scoring each dataset using the same model. If using "numPrincipalComponents" when running Principal Components Analysis, you may request a different number of components when transforming each dataset using the same PCA model. For all valid parameters and evaluations for each algorithm, see the RASON Reference Guide.
- "evaluations" – This property specifies the results to be reported back to the user. Only those evaluations specified for this property will be *computed* or *reported*. Evaluation results may either be 1. A part of the RASON response or 2. Bound to a writeable datasource. In the example below, "fittedModelJson" and "regressionSummary" are part of the RASON response while "influenceDiagnostics" is bound to the writeable datasource "myExportSrc". To view this complete example, see LinearRegression.json on the Editor page on RASON.com. Note: Some code has been removed from the example below for simplicity.

```
{
  "modelName": "LinearRegression",
  "comment": "regression: linear model; scoring examples
JSONLinearRegression.json and PMMLRegressor.json use
Exported fitted model, mlrModel, to score new data",
  "modelType": "datamining",
  "datasources": {
```

```

    "myTrainSrc": {
      "type": "csv",
      "connection": "hald-small-train.txt",
      "direction": "import"
    },
    ...
    "myExportSrc": {
      "type": "csv",
      "content": "export",
      "connection": "influence-diagnostics.csv",
      "direction": "export"
    }
  },
  "datasets": {
    "myTrainData": {
      "binding": "myTrainSrc",
      "targetCol": "Y"
    },
    ...
  },
  "estimator": {
    "mlrEstimator": {
      "type": "regression",
      "algorithm": "linearRegression",
      "parameters": {
        "fitIntercept": true
      }
    }
  },
  "actions": {
    "mlrModel": {
      "trainData": "myTrainData",
      "estimator": "mlrEstimator",
      "action": "fit",
      "evaluations": [
        "fittedModelJson",
        {
          "name": "influenceDiagnostics",
          "binding": "myExportSrc"
        },
        "regressionSummary"
      ],
      ...
    },
    ...
  }
}

```

Notes on exporting to a writable data source.

1. "Type": "CSV" and "Type": "JSON" simply create or overwrite the files with the dataframe/table evaluation.
2. The "selection" property specifies the Excel worksheet and is optional when "Type": "Excel". If not provided, the worksheet name will be automatically assigned based on the RASON script's name, action and evaluation, i.e mlr-mlrmodel-influenceDiagnostics.
3. The "selection" property specifies the Database table name and is optional for all database types. If not provided, the table name will be automatically assigned as in 2 above.

4. Users can write to the same Excel workbook or same database – adding new worksheets/tables with subsequent evaluations.
5. It is also possible to create a new MS Access database file and write evaluations there.
6. Creating a new database for MS SQL/Oracle types or when using an ODBC connection string is not supported. As a result, "connection" must point to an existing database.

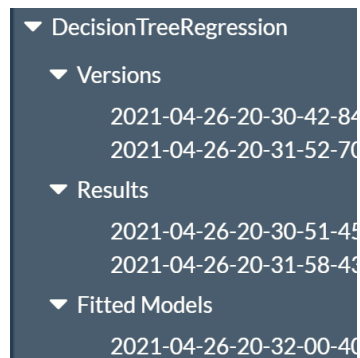
For more examples on exporting results to a writeable datasource, see the "datasources" section in the RASON Reference Guide.

POSTing/Exporting Fitted Models

In the latest version of RASON Decision Services, fitted models are now treated as first-class model objects on the RASON server inheriting the model name of their parent RASON model instance. Fitted models inherit all properties of RASON models such as the model name and the ability to designate a "champion" fitted model. If a fitted model is referred to in a RASON Scoring model script, the champion or the most recent fitted model given the <modelName> (either PMML or JSON) will be used. There is no need to download the exported fitted model file and attach it to the RASON scoring model as was required in the past. However, this mechanism remains unchanged and may still be utilized.

Notes:

- Fitted models receive their own container and table entry and are listed on the RASON Editor tab as shown in the screenshot below.



- Fitted models can be POSTed or PUT directly to the server creating model Origin and Versions with ModelKind=Fitted. See the API Endpoints POST/PUT `ration.net/api/model/{nameorid}` in the subsequent chapter, Using the RASON API.
- Multiple solves of the same named model instance would result in multiple versions of the fitted model. As discussed above, when a fitted model is referred to in a RASON Scoring model script, the champion or the most recent fitted model given the <modelName> (either PMML or JSON) will be used.
- When using the latest syntax for fitted models, there is no need to specify model type or content, only the modelName; everything else is inferred automatically given the model stored on the RASON server.
- When a fitted model is produced by a Decision flow or model defining a file to be exported, the fitted model is also POSTed to the Rason server.

The two examples below illustrate how to produce a fitted model by either: POSTing the fitted model to the RASON Server or exporting a file containing the fitted model. While both approaches are valid and produce the same fitted model, the method that POSTs the fitted model to the RASON server is strongly recommended for its ease of use.

Note that some code has been excluded for simplicity.

RASON Example – POSTs Fitted Model to RASON Server

To open this example, click the RASON example files icon on the RASON ribbon (on the Editor tab at www.RASON.com), then click Data Science – Classification - DecisionTree.json.

When this method is used, only the data files are imported within the datasources section of the RASON Model. The "export" property within "actions" POSTs the fitted model, in either "JSON" or "PMML" format, to the RASON Server where it will persist until deleted by the user. The name of the fitted model will be the same as the modelName setting, in this instance "DecisionTreeClassification".

```
{
  "modelName": "DecisionTreeClassification",
  "modelDescription": "classification: decision tree; fitted model
    generated and POSTed to RASON Server",
  "modelType": "datamining",
  "datasources": {
    "myTrainSrc": {
      "comment": "imports training and validation partitions",
      "type": "csv",
      "connection": "hald-small-binary-train.txt",
      "direction": "import"
    },
    "myValidSrc": {
      "type": "csv",
      "connection": "hald-small-binary-valid.txt",
      "direction": "import"
    }
  },
  ...
  "actions": {
    "comment": "Using export: json to POST the fitted model using
      the modelName setting and fits the model to myTrainData",
    "treeModel": {
      "trainData": "myTrainData",
      "validData": "myValidData",
      "estimator": "treeEstimator",
      "action": "fit",
      "export": "pmml",
      "evaluations": ["trainingLog", "pruningLog",
        "featureImportance",
        "treeRules"]
    },
    ...
  }
}
```

Notice that only the imported data sources appear here. In this method, the fitted model is not exported, rather the fitted model is POSTed to the RASON Server.

The "export" property POSTs the fitted model to the RASON Server in either "json" or "pmml" format.

RASON Example – Fitted Models as Data Sources

The example below exports the fitted model in PMML format to the XML file, classification-decision-tree.xml. To export a fitted model in JSON format, replace "myPMMLSrc" with :

```
"myJSONSrc": {
  "type": "json",
  "content": "json-model",
  "connection": "<file name>",
  "direction": "export"
}
```

To open this example, click the RASON example files icon on the RASON ribbon (on the Editor tab at www.RASON.com), then click Data Science -- Classification - Fitted Models as Datasource –

DecisionTree.json. The "binding" property within "actions" binds the fitted model to "myPMMLSrc" which exports the fitted model to the file, classification-decision-tree.xml using "direction": "export". This file persists on the server until deleted by the user.

```
{
  "modelName": "DecisionTreeClassification",
  "modelDescription": "classification: decision tree; fitted model
    generated and POSTed to RASON Server",
  "modelType": "datamining",
  "datasources": {
    "myTrainSrc": {
      "type": "csv",
      "connection": "hald-small-binary-train.txt",
      "direction": "import"
    },
    "myValidSrc": {
      "type": "csv",
      "connection": "hald-small-binary-valid.txt",
      "direction": "import"
    },
    "myPMMLSrc": {
      "type": "xml",
      "content": "pmml-model",
      "connection": "classification-decision-tree.xml",
      "direction": "export"
    }
  },
  "actions": {
    "comment": "Using export: json to POST the fitted model using the
      modelName setting and fits the model to myTrainData",
    "treeModel": {
      "trainData": "myTrainData",
      "validData": "myValidData",
      "estimator": "treeEstimator",
      "action": "fit",
      "binding": "myPMMLSrc",
      "evaluations": ["trainingLog", "pruningLog", "featureImportance",
        "treeRules"]
    }
  }
}
```

When exporting the fitted model, you must provide the exported filename in the datasource section of the RASON model with the property "direction": "export". This example exports the fitted model in PMML format.

The "binding" property binds the fitted model to myPMMLSrc which exports the fitted model to the classification-decision-tree.xml file. To download the file, use GET rason.net/api/model/{nameorid}/result/data. This file is persisted on the Server until deleted by the user.

After the fitted model is either POSTed or exported, the model may be used for scoring new data. See the section on Scoring for more information.

Optional RASON Sections

RASON DM also features several additional optional sections that can be used to further refine the RASON model. These additional sections are: data, fittedModel, preProcessor and weakLearner.

"data"

Data arrays may be defined and calculated in this optional section, to be used later in a data science method. Scalars, arrays or tables containing scalars maybe be defined in the data section. If pulling data from an external source, this section may be used to "bind" the data to an array or table.

In the example code below, data from the `qty` column from the `parts_data` data source is assigned to the `parts` table. Note: A table is created here, rather than an array, by the use of the `valueCol` property.

```
"data": {
  "parts": {
    "binding": "parts_data", "valueCol": "qty"
  }
},
```

Properties available for data, are:

- "name" – Used to define the table, array or scalar name.
- "dimensions" – Defines a 1 or 2-dimensional array or table.
- "value" – Defines the value of the array or table
- "valueCol" - Used with `binding` property to bind imported values from a readable data source.
- "binding" - Allows data to be edited outside of the model from a URL or when calling the RASON™ interpreter to solve an optimization or simulation model.
- "comment" – Use this property to enter a comment to describe the data.

"fittedModel"

Used (only) when scoring a model. This section is similar to "datasets" but rather than refining imported data, this section defines a model that you can bind to when performing an "action" such as "forecast", "predict", "fit" or "transform".

In the example below, a previously fit linear regression model previously POSTed to the RASON server is used to score the `hald-small-score.txt` dataset (which was imported into RASON as "dataSrc" and then bound to "myData").

```
{
  "modelName": "PMMLRegressor",
  "modelDescription": "regression: linear model scoring from
pmml",
  "modelType": "datamining",
  "datasources": {
    "dataSrc": {
      "type": "csv",
      "connection": "hald-small-score.txt",
      "direction": "import"
    }
  },
  "datasets": {
    "myData": {
      "binding": "dataSrc"
    }
  },
  "fittedModel": {
    "mlrModel": {
      "modelName": "LinearRegression"
    }
  },
  "actions": {
    "myDataPrediction": {
```



```

        "data": "myData",
        "fittedModel": "mlrModel",
        "action": "predict",
        "evaluations": [
            "prediction"
        ]
    }
}
}

```

This section includes two properties: "modelName" and "binding".

- Use "modelName" when the fitted model is residing on the RASON Server.
- Use "binding" when importing a file containing the fitted model.

See the section above for an example of each.

"preProcessor"

This optional section may be used for preliminarily data preparation or to compute values of some properties, which are passed later, at parse-time, to the RASON DM engine. This section is parsed once, before the model is parsed.

In the example below, "numLeafRecords", is defined within the "preProcessor" section and is then referenced within the estimator, "treeEstimator", to set the parameter, "minNumRecordsInLeaves".

```

"preprocessor": {
    "numLeafRecords": {
        "formula": "INT (MAX (1, ROWS (myTrainData) / 10))"
    }
},
"estimator": {
    "treeEstimator": {
        "type": "classification",
        "algorithm": "decisionTree",
        "parameters": {
            "priorProbMethod": "EMPIRICAL",
            "minNumRecordsInLeaves": "numLeafRecords",
            "maxNumNodes": 5,
            "maxNumLevels": 3,
            "maxNumSplits": 10,
            "categoricalFeaturesNames": [ "X1" ],
            "prunedTreeType": "MIN_ERROR"
        }
    }
}

```

The properties available for this section include:

- "name" - Use this property to define the array name.
- "dimensions" – Use this property to define a 1 or 2 dimensional array.
- "value" – Use this property to set the value of the array.
- "formula" – Use this property to enter the formula being calculated.
- "comment" - Use this property to enter a comment describing the formula.

"weakLearner"

This section is only required when a bagging or boosting estimator is specified in "estimator" and is used to define the weak learner used in these algorithms.

The following example defines the treeWeakLearner data source.

```
"weakLearner": {
  "treeWeakLearner": {
    "type": "classification",
    "algorithm": "decisionTree",
    "parameters": {
      "minNumRecordsInLeaves": 2
    }
  }
},
```

In this example code snippet, weakLearner, "treeWeakLearner", is initialized to perform a classification (type: classification) using the decision tree algorithm as the weak learner for a bagging or boosting algorithm defined in within "estimator".

The following parameters are available for this section:

- "type" - Type must be either "classification" or "regression".
- "algorithm" - Algorithm must be one of the following: "neuralNetwork", "decisionTree", "nearestNeighbors", "naiveBayes", "discriminantAnalysis", "logisticRegression", or "linearRegression".
- "parameters" - Properties for "parameters" will vary depending on the selected algorithm.

Example RASON Models

In this segment of the User Guide, several example RASON models will be examined including an example on sampling a dataset.

Sampling Example

Let's step through a few RASON data science examples now. Let's start with an example of how to sample from a dataset.

```
{
  modelName:"Sampling",
  modelDescription: 'transformation: sampling',
  modelType: "datamining",
  "datasources": {
    "mySrc": {
      "type": "csv",
      "connection": "hald-small-binary.txt",
      "direction": "import"
    }
  },
  "datasets": {
    "myData": {
      "binding": "mySrc"
    }
  },
  "transformer": {
```

```

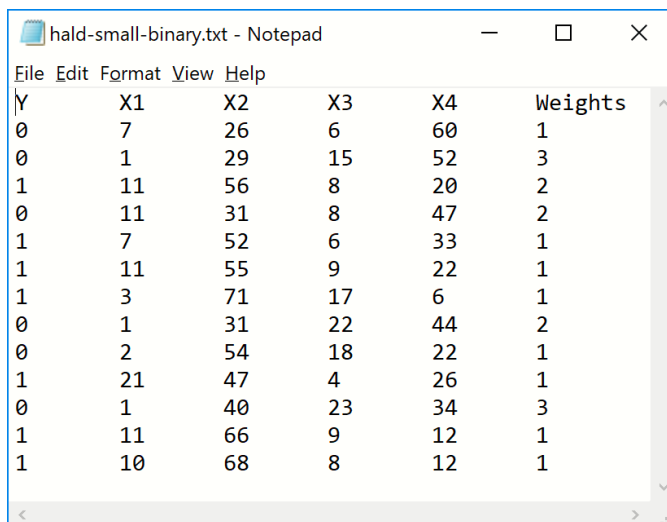
    "mySampler": {
      "type": "transformation",
      "algorithm": "sampling",
      "parameters": {
        "sampleSize": 4,
        "replaceOption": "false",
        "sortIndexes": "false",
        "seed": 123
      }
    },
    "actions": {
      "sampleData": {
        "data": "myData",
        "action": "transform",
        "evaluations": [
          "transformation"
        ]
      }
    }
  }
}

```

Within "datasources" a sample is taken from the hald-small-binary dataset (contained within hald-small-binary.txt) and given the name "mySrc". This file contains data in CSV format.

Note: Input files in a Data Science RASON model must not contain a path to a file location.

hald-small-binary.txt



Y	X1	X2	X3	X4	Weights
0	7	26	6	60	1
0	1	29	15	52	3
1	11	56	8	20	2
0	11	31	8	47	2
1	7	52	6	33	1
1	11	55	9	22	1
1	3	71	17	6	1
0	1	31	22	44	2
0	2	54	18	22	1
1	21	47	4	26	1
0	1	40	23	34	3
1	11	66	9	12	1
1	10	68	8	12	1

Within "datasets", the data sampled in mySrc is bound to myData. The sampling transformer, mySampler, is specified within the "transformer" attribute. Since we are sampling, the "type" is specified as "transformation" and the algorithm is specified as "sampling". Various sampling options under "parameters": sample size (sampleSize), sample with replacement (replaceOption), index sorting (sortIndexes) and the random seed value (seed). For a complete list of options associated with the sampling transformer, please see the RASON Reference Guide.

Finally, under "actions", the transformation (the sampling) is performed on the myData dataset. Under "evaluations" we see the quantities to be computed and reported. In this example, the sample is the result.

```

Getting model results: GET https://rason.net /api/model/
2590+Sampling+2020-01-20-01-07-51-620648/result
{
  "status": {
    "id": "2590+Sampling+2020-01-20-01-07-51-620648"
    "code":0,
    "codeText": "Success"
  },
  "results": [sampleData.transformation"],
  "sampleData": {
    "transformation": {
      "objectType": "dataFrame",
      "name": "Sample:mydata",
      "order": "col",
      "rowNames": ["Record 5", "Record 3", "Record 9", "Record
7"],
      "colNames": ["Y", "X1", "X2", "X3", "X4", "Weights"],
      "colTypes": ["double", "double", "double", "double",
"double",
"double"],
      "indexCols": null,
      "data":[
        [1,1,0,1],
        [7,11,2,3],
        [52,56, 54, 71],
        [6,8,18,17],
        [33,20,22,6],
        [1,2,1,1]
      ]
    }
  }
}

```

A Note about Data Frames

A **DataFrame**, in RASON DM, is a collection of data organized into named columns of equal length and homogeneous type. RASON DM uses DataFrames to deliver input data to an algorithm and to deliver the results of the algorithm back to the user. DataFrames hold heterogeneous data across columns (variables): numeric, categorical, or textual.

Examples of basic DataFrame tasks are:

- Creating and filling DataFrames
- Selecting a subset of columns/rows
- Appending columns or rows
- Selecting subsets for training and verification models

RASON Decision Services introduces two new REST API endpoints POST rason.net/api/solve and POST rason.net/api/model/{nameorid}/solve which automatically create an OData endpoint which returns the result in a dataframe object. For more information, see the Using the REST API chapter.

According to the results, the records sampled were:

Index	Y	X1	X2	X3	X4	Weights
5	1	7	52	6	33	1

3	1	11	56	8	20	2
9	0	2	54	18	22	1
7	1	3	71	17	6	1

Partitioning Example

Now let's take a look at a partitioning example.

```
{
  "modelName": "Partitioning",
  "modelType": "datamining",

  "modelDescription": "transformation: partitioning",
  "datasources": {
    "mySrc": {
      "type": "csv",
      "connection": "hald-small-binary.txt",
      "direction": "import"
    }
  },
  "datasets": {
    "myData": {
      "binding": "mySrc"
    }
  },
  "transformer": {
    "myPartitioner": {
      "type": "transformation",
      "algorithm": "partitioning",
      "parameters": {
        "partitionMethod": 'RANDOM',
        "ratios": [
          [ "training", 0.5 ],
          [ "validation", 0.3 ],
          [ "test", 0.2 ]
        ]
      },
      "seed": 123
    }
  },
  "actions": {
    "partitions": {
      "data": "myData",
      "action": 'transform',
      "evaluations": [ 'transformation' ]
    }
  }
}
```

The "datasources" section in this example is identical to the Sampling example above. Inside of "datasets", the datasource "mySrc" is bound to the "myData" dataset. Since partitioning performs a transformation of the data, the "transformer" attribute is used with type "transformation" and algorithm "partitioning". Within "parameters", "RANDOM" is specified for "partitionMethod" which selects random partitioning as the type of partitioning to be performed.

In simple random sampling, every observation in the main dataset has equal probability of being selected for the partition dataset. For example, if you specify 60% for the training dataset, then 60% of the total observations are randomly selected for the training dataset. In other words, each observation has a 60% chance of being selected. Random partitioning uses the system clock as a default to initialize the random number seed. Alternatively, the random seed can be manually set which will result in the same observations being chosen for the training/validation/test sets each time a standard partition is created.

In this example, 50% of the records will be included in the training partition, 30% will be included in the validation partition and 20% will be included in the test partition.

Within "actions", the partitioning (or transformation) is performed on the MyData data set. The returned result will be the three different partitions: training, validation and test.

Getting model results: GET

<https://rason.net/api/model/2590+Partitioning+2020-01-20-01-18-37-436902/result>

```
{
  "status": {
    "id": "2590+Partitioning+2020-01-20-01-18-37-436902"
    "code":0,
    "codeText":"Success"
  },
  "results":["partitions.transformation"],
  "partitions": {
    "transformation": {
      "objectType": "dataFrameVector",
      "name": "myData - Partitioned",
      "data": {
        "training": {
          "objectType": "dataFrame",
          "name": "training",
          "order": "col",
          "rowNames": ["Record 1", "Record 12", "Record 6", "Record 13",
            "Record 9", "Record 4", "Record 2"],
          "colNames": ["Y", "X1", "X2", "X3", "X4", "Weights"],
          "colTypes": ["double", "double", "double", "double",
            "double",
            "double"],
          "indexCols": null,
          "data": [
            [0, 1, 1, 1, 0, 0, 0] },
            [7, 11, 11, 10, 2, 11, 1],
            [26, 66, 55, 68, 54, 31, 29],
            [6, 9, 9, 8, 18, 8, 15],
            [60, 12, 22, 12, 22, 47, 52],
            [1, 1, 1, 1, 1, 2, 3]
          ]
        },
        "validation": {
          "objectType": "dataFrame",
          "name": "validation",
          "order": "col",
          "rowNames": ["Record 11", "Record 3", "Record 10",
            "Record 7"],
          "colNames": ["Y", "X1", "X2", "X3", "X4", "Weights"],
          "colTypes": ["double", "double", "double", "double",
```

From the results, we can see the records allocated to the training, validation and test partitions.

Index	Y	X1	X2	X3	X4	Weights
1	0	7	26	6	60	1
12	1	11	66	9	12	1
6	1	11	55	9	22	1
13	1	10	68	8	12	1
9	0	2	54	18	22	1
4	0	11	31	8	47	2
2	0	1	29	15	52	3

Index	Y	X1	X2	X3	X4	Weights
11	0	1	40	23	34	3
3	1	11	56	8	20	2
10	1	21	47	4	26	1
7	1	3	71	17	6	1

Index	Y	X1	X2	X3	X4	Weights
5	1	7	52	6	33	1

8	0	1	31	22	44	2
---	---	---	----	----	----	---

Example Performing Feature Selection

Next, we will look at an example of how to call Feature Selection in RASON.

Feature Selection allows users the ability to rank and select the most relevant variables for inclusion in a classification or prediction model. In many cases the most accurate models, or the models with the lowest misclassification or residual errors, have benefited from better feature selection, using a combination of human insights and automated methods. RASON Data Science provides a facility to compute all of the following metrics, described in the literature, to give users information on what features should be included, or excluded, from their models.

- Correlation-based
 - Pearson product-moment correlation
 - Spearman rank correlation
 - Kendall concordance
- Statistical/probabilistic independence metrics
 - Welch's statistic
 - F statistic
 - Chi-square statistic
- Information-theoretic metrics
 - Mutual Information (Information Gain)
 - Gain Ratio
- Other
 - Cramer's V
 - Fisher score
 - Gini index

Only some of these metrics can be used in any given application, depending on the characteristics of the input variables (features) and the type of problem. In a supervised setting, if we classify data science problems as follows:

- $\mathbb{R}^n \rightarrow \mathbb{R}$: real-valued features, prediction (regression) problem
- $\mathbb{R}^n \rightarrow \{0, 1\}$: real-valued features, binary classification problem
- $\mathbb{R}^n \rightarrow \{1..C\}$: real-valued features, multi-class classification problem
- $\{1..C\}^n \rightarrow \mathbb{R}^n$: nominal categorical features, prediction (regression) problem
- $\{1..C\}^n \rightarrow \{0, 1\}$: nominal categorical features, binary classification problem
- $\{1..C\}^n \rightarrow \{1..C\}$: nominal categorical features, multi-class classification problem

then we can describe the applicability of the Feature Selection metrics by the following table:

	R-R	R-{0,1}	R-{1..C}	{1..C}-R	{1..C}-{0,1}	{1..C}-{1..C}
Pearson	N					
Spearman	N					
Kendall	N					
Welch's	D	N				
F-Test	D	N	N			
Chi-squared	D	D	D	D	N	N

Mutual Info	D	D	D	D	N	N
Gain Ratio	D	D	D	D	N	N
Fisher	D	N	N			
Gini	D	N	N			

"N" means that metrics can be applied naturally, and "D" means that features and/or the outcome variable must be discretized before applying the particular filter.

Here is an example of a RASON model calling Feature Selection using linear wrapping.

```
{
  modelName: 'LinearWrapping',
  modelType: "datamining",

  modelDescription: 'feature selection: linear wrapping',
  datasources: {
    myTrainSrc: {
      type: 'csv',
      connection: 'hald-small.txt',
      direction:"import"
    }
  },
  datasets: {
    myTrainData: {
      binding: 'myTrainSrc',
      targetCol: 'Y'
    }
  },
  estimator: {
    linearFSEstimator: {
      type: 'featureSelection',
      algorithm: 'linearWrapping',
      parameters: {
        fitIntercept: true,
        method: 'EXHAUSTIVE_SEARCH'
      }
    }
  },
  actions: {
    linearFSModel: {
      data: 'myTrainData',
      estimator: 'linearFSEstimator',
      action: 'fit',
      evaluations: [
        'bestSubsets',
        'bestSubsetsDetails'
      ]
    },
    reducedTrainData: {
      data: 'myTrainData',
      fittedModel: 'linearFSModel',
      parameters: {
        numTopFeatures: 2
      },
      action: 'transform',
    }
  }
}
```

```

        evaluations: [
            'transformation'
        ]
    }
}
}

```

The "datasources" section in this example creates one datasource, "myTrainSrc" which imports the hald-small dataset within the imported hald-small.txt file. Inside of "datasets", the datasource "myTrainData" is bound to the "myTrainSrc" dataset. An output column is specified as the "Y" column (targetCol: 'Y'). *Note: Input files in a Data Science RASON model must not contain a path to a file location.*

A new estimator, linearFSEstimator, is created within "estimator". This estimator is of type "featureSelection" using algorithm "linearWrapping" and the parameters "fitIntercept" set to True and method set to "EXHAUSTIVE_SEARCH". For a complete list of all parameters associated with Feature Selection, see the RASON Reference Guide.

Two new action items, "linearFSModel" and "reducedTrainData" are listed within "actions". The "linearFSModel" action item "fits" the model while "reducedTrainData" uses the resultant model to transform the dataset.

The "linearFSModel" action item uses the estimator, linearFSEstimator, to "fit" a feature selection model to the "myTrainData" dataset and return the best subsets and their details in the results.

The "reducedTrainData" action item uses the fitted "linearFSModel" model to perform feature selection to reduce (transform) the "myTrainData" dataset from 5 features (not including the output variable, Y) to 2 (three including the output variable, Y).

Here are the results:

Getting model results: GET
<https://rason.net/api/model/2590+LinearWrapping+2020-01-20-01-35-50761786/result>

```

{
  "status": {
    "id": "2590+LinearWrapping+2020-01-20-01-35-50-761786"
    "code": 0,
    "codeText": "Success"
  },
  "results": ["linearFSModel.bestSubsets",
    "linearFSModel.bestSubsetsDetails",
    "reducedTrainData.transformation"],
  "linearFSModel": {
    "bestSubsets": {
      "objectType": "dataFrame",
      "name": "Best Subsets",
      "order": "col",
      "rowNames": ["Subset 1", "Subset 2", "Subset 3", "Subset 4",
        "Subset 5", "Subset 6"],
      "colNames": ["Intercept", "X1", "X2", "X3", "X4", "Weights"],
      "colTypes": ["double", "double", "double", "double",
        "double", "double"],
      "indexCols": null,
      "data": [
        [1, 1, 1, 1, 1, 1],
        [0, 0, 1, 1, 1, 1],
        [0, 0, 1, 1, 1, 1],
        [0, 0, 0, 0, 1, 1],

```

Status of RASON model

Results for linearFSModel begin here.

Notice that the results are printed in *column* order. The first subset (Subset 1) contains only the intercept term. The second subset (Subset 2) contains the intercept term plus 1 feature, X4. The third subset (Subset 3) contains the intercept term plus 2 features, X1 and X2 and so on.

```

        [0, 1, 0, 1, 1, 1],
        [0, 0, 0, 0, 0, 1]
    ]
},
"bestSubsetsDetails": {
    "objectType": "dataFrame",
    "name": "Best Subsets Details",
    "order": "col",
    "rowNames": ["Subset 1", "Subset 2", "Subset 3", "Subset 4",
    "Subset 5", "Subset 6"],
    "colNames": ["#Coefficients", "RSS", "Mallows's Cp", "R2",
    "Adjusted R2", "Probability"],
    "colTypes": ["integer", "double", "double", "double",
    "double", "double"],
    "indexCols": null,
    "data": [
        [1, 2, 3, 4, 5, 6] ,
        [2715.7630769230273, 883.86691689923498,
        57.904483176071921,
        47.972729400348463, 47.863639350457937,
        47.800255339956486],
        [386.70376545604654, 120.43588636278355, 1.479690732816751,
        2.0252575726668702, 4.009282127686447, 6],
        [2.5424107263916085e-14, 0.67454196413163481,
        0.97867837453564743, 0.98233545120044441,
        0.98237562040770021,
        0.98239895970818625],
        [2.5424107263916085e-14, 0.64495486996177454,
        0.97441404944277676, 0.97644726826725459,
        0.97356343061154282,
        0.96982678807116807],
        [5.4992942301671353e-06, 0.00015856276306674378,
        0.69819260728365373, 0.98747306627232578,
        0.9259478188926249, 0]
    ]
}
},
"reducedtraindata": {
    "transformation": {
        "objectType": "dataFrame",
        "name": "mytraindata:Reduced",
        "order": "col",
        "rowNames": ["Record 1", "Record 2", "Record 3", "Record 4",
        "Record 5", "Record 6", "Record 7", "Record 8", "Record 9",
        "Record 10", "Record 11", "Record 12", "Record 13"],
        "colNames": ["X1", "X2", "Y"],
        "colTypes": ["double", "double", "double"],
        "indexCols": null,
        "data": [
            [7, 1, 11, 11, 7, 11, 3, 1, 2, 21, 1, 11, 10],
            [26, 29, 56, 31, 52, 55, 71, 31, 54, 47, 40, 66, 68],
            [78.5, 74.299999999999997, 104.3, 87.599999999999994,
            95.900000000000006, 109.2, 102.7, 72.5,
            93.099999999999994, 115.90000000000001,
            83.799999999999997, 113.3,
            109.40000000000001]
        ]
    }
}
]

```

Best subset result details (bestSubsetsDetails) for linearFSModel. Here are the calculated statistics for the six subsets. The statistics returned are: RSS, Mallows's Cp, R2, Adjusted R2 and Probability.

Results for reducedTrainData begin here.

```
}
}
}
```

The dataset has been reduced from 5 features and 1 output variable to 2 features and 1 output variable. The resulting object is a data frame.

	X1	X2	Y
Record 1	7	26	78.5
Record 2	1	29	74.29999999999997
Record 3	11	56	104.3
Record 4	11	31	87.59999999999994
Record 5	7	52	95.900000000000006
Record 6	11	55	109.2
Record 7	3	71	102.7
Record 8	1	31	72.5
Record 9	2	54	93.09999999999994
Record 10	21	47	115.90000000000001
Record 11	1	40	83.79999999999997
Record 12	11	66	113.3
Record 13	10	68	109.40000000000001

Find Best Model for Classification and Prediction

RASON Decision Services includes comprehensive, powerful support for data science and machine learning. Using these tools, you can “train” or fit your data to a wide range of statistical and machine learning models: Classification and regression trees, neural networks, linear and logistic regression, discriminant analysis, naïve Bayes, k-nearest neighbors and more. But the task of choosing and comparing these models, and selecting parameters for each one was up to you...until now...

With the Find Best Model options, you can automate this work as well! Find Best Model uses methods similar to those in (expensive high-end) tools like DataRobot and RapidMiner, to automatically choose types of ML models and their parameters, validate and compare them according to criteria that you choose, and deliver the model that best fits your data.

Continue reading to learn how to run the Find Best Model method and to learn how to run a classification or prediction learner independently.

Classification/Prediction using Find Best Model

Rason Decision Services includes the Find Best Model method for both classification and regression models. This method fits the model to all available classification or regression learners and then determines which model is the best fit to the data.

RASON Decision Services allows differing levels of control when specifying the FindBestModel estimator (using "algorithm": "findBestModel").

1. Full control which allows users to select the learners and edit the learner parameters.

Example code for full control over Find Best Model method - Classification

```
"estimator": {
  "type": "classification",
  "algorithm": "findBestModel",
  "learners": {
    "logisticRegression": {
      "fitIntercept": false,
    },
    "nearestNeighbors": {
```

```

        "numNeighbors": 5,
    }, ...
}, ...
}

```

Example code for full control over Find Best Model method - Prediction

```

"estimator": {
  "type": "regression",
  "algorithm": "findBestModel",
  "learners": {
    "linearRegression": {
      "fitIntercept": false,
    },
    "nearestNeighbors": {
      "numNeighbors": 5,
    }, ...
  }, ...
}

```

2. Partial control which allows users to select the learners with their default parameters.

Example code for partial control over Find Best Model method-Classification

```

"estimator": {
  "type": "classification",
  "algorithm": "findBestModel",
  "learners": { "logisticRegression", "decisionTree",
  "nearestNeighbors" }
},

```

Example code for partial control over Find Best Model method-Prediction

```

"estimator": {
  "type": "reression",
  "algorithm": "findBestModel",
  "learners": { "linearRegression", "decisionTree",
  "nearestNeighbors" }
},

```

3. Default Specification which selects all eligible learners and uses all default parameter settings.

Example code selecting all Find Best Model learners at their parameter default settings-Classification

```

"estimator": {
  "type": "classification",
  "algorithm": "findBestModel",
},

```

Example code selecting all Find Best Model learners at their parameter default settings-Classification

```

"estimator": {
  "type": "regression",
  "algorithm": "findBestModel",
},

```

Default Specification

The example below creates a new estimator using the Find Best Model Classification method which uses all eligible learners with all default parameters. Then that estimator fits a model to the bh-class-

train.txt dataset. This model is then used to score both the bh-class-train.txt and bh-class-valid.txt datasets.

```
{
  "comment": "regression: find best model",
  "modelName": "fbmFullSpecs",
  "datasources": {
    "myTrainSrc": {
      "type": "csv",
      "connection": "bh-class-train.txt"
    },
    "myValidSrc": {
      "type": "csv",
      "connection": "bh-class-valid.txt"
    }
  },
  "datasets": {
    "myTrainData": {
      "binding": "myTrainSrc",
      "targetCol": "CATMEDV"
    },
    "myValidData": {
      "binding": "myValidSrc",
      "targetCol": "CATMEDV"
    }
  },
  "estimator": {
    "fbmEstimator": {
      "type": "classification",
      "algorithm": "findBestModel"
    }
  },
  "actions": {
    "fbmModel": {
      "trainData": "myTrainData",
      "estimator": "fbmEstimator",
      "action": "fit",
      "evaluations": ["messages"]
    },
    "trainScore": {
      "data": "myTrainData",
      "fittedModel": "fbmModel",
      "parameters": {
        "bestLearnerMetric": "ACCURACY"
      },
      "action": "predict",
      "evaluations": ["modelPerformance", "bestLearner",
        "prediction", "learnerForScoring"]
    },
    "validScore": {
      "data": "myValidData",
      "fittedModel": "fbmModel",
      "action": "predict",
      "parameters": {
        "bestLearnerMetric": "ACCURACY",
        "useForScoring": true
      }
    }
  }
}
```

Two files are imported as datasources
1. bh-class-train.txt imported as myTrainSrc
2. bh-class-valid.txt as myValidSrc.

Both datasources are bound to a dataset.
1. myTrainSrc (bh-class-train.txt) to myTrainData
2. myValidSrc (bh-class-valid.txt) to myValidData

A new estimator is created, fbmEstimator, using the Find Best Model Classification method. Use "type": "classification" to use Find Best Model for prediction.

A model is fit (fbmModel) to the myTrainData dataset (bh-class-train.txt) using all eligible learners with all parameters set at their defaults.

The parameter "bestLearnerMetric" is applied to the "predict" actions resulting in two evaluations: "modelPerformance" and "bestLearner". (See below for more details.)

The "learnerForScoring" evaluator in both "trainScore" and "validScore", returns the learner used to perform scoring for each "predict" action in the output.

The parameter "useForScoring"=true in validScore results in the best learner found for validScore, scoring **all** "predict" actions, which in this example is validScore and validTrain.

```

    "evaluations": ["modelPerformance", "bestLearner",
    "prediction", "learnerForScoring"]
  }
}
}

```

- Notice that the Default Specification estimator only contains two lines of code, the estimator type (classification) and the estimator algorithm (findBestModel). This is because with this specification, all eligible learners are selected using all parameter defaults. See below for estimators that allow specific learners to be selected and learner parameters to be modified.
- Within "actions" the fbMModel is fit to the myTrainData (bh-class-train.txt) dataset. The "fit" action has only one evaluation, "messages", which reports the fitting log, where any failures/warnings will appear. *Note: If there are any missing learners in the model performance table, check here to see why.*
- The "predict" actions that bind to the labeled data (trainScore and ValidScore) accept the (optional) parameter "bestLearnerMetric". This parameter specifies the metric used to select the best learner, i.e. accuracy for classification or R2 for regression. This metric, which can appear in multiple "predict" actions, returns the "modelPerformance" report and the name of the "bestLearner", the name of the best learner for a given "predict" action. In order for both "modelPerformance" and "bestLearner" to appear in the output, both must be listed under "evaluations" in the "predict" action.

Results for "modelPerformance" evaluator in ODATA viewer*

ID	Name	Accuracy__correct_	Accuracy__pctcorrect_	Specificity	Sensitivity	Precision	F1_score
0	Logistic Regression	190	94.05940594059405	0.94545454...	0.9189189...	0.7906976...	0.85
1	Decision Tree	187	92.57425742574257	0.93930393...	0.8648648...	0.7619047...	0.81012658...
2	Nearest Neighbors	172	85.14851485148515	0.90303030...	0.6216216...	0.5897435...	0.60526315...
3	Neural Network	163	80.6930693069307				
4	Linear Discriminant	189	93.56435643564357	0.95151515...	0.8648648...	0.8	0.83116883...
5	Bagging	193	95.54455445544554	0.95151515...	0.9729729...	0.8181818...	0.88888888...
6	Boosting	194	96.03960396039604	0.96969696...	0.9189189...	0.8717948...	0.89473684...
7	Random Trees	193	95.54455445544554	0.95151515...	0.9729729...	0.8181818...	0.88888888...

*Note that "bestLearner" result does not appear in the ODATA viewer. See JSON results for "bestLearner" and "learnerForScoring" output.

Results for "modelPerformance", "learnerForScoring" and "bestLearner" evaluators in JSON viewer

```

...
"fbmModel": {
  "messages": null
},
"validScore": {
  "modelPerformance": {
    "objectType": "dataFrame",
    "name": "myValidData",
    "order": "col",
    "rowNames": ["Logistic Regression", "Decision Tree",
    "Nearest Neighbors", "Neural Network", "Discriminant
    Analysis", "Bagging", "Boosting", "Random Trees"],
    "colNames": ["Accuracy (#correct)", "Accuracy
    (%correct)", "Specificity", "Sensitivity", "Precision",
    "F1 score"],
    "colTypes": ["double", "double", "double", "double",
    "double", "double"],
    "indexCols": null,
    "data": [
      [190, 187, 172, 163, 189, 193, 194, 193],

```

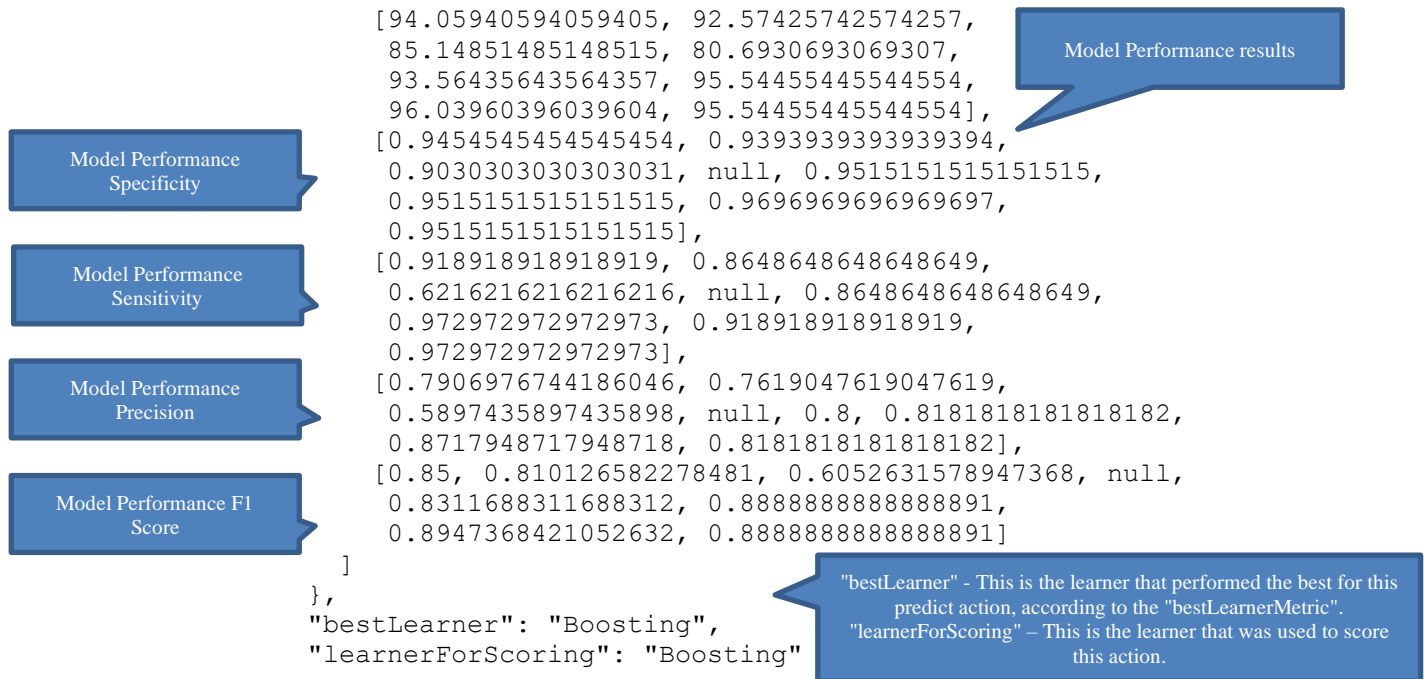
Note that any messages/errors that occurred during Find Best Model would appear here. Note: If there are any missing learners in the model performance table, check here to find out why.

Model Performance Learners

Model Performance Metric Headings

Model Performance Accuracy # Correct

Model Performance Accuracy % Correct



- The parameter "useForScoring=true" may be submitted to any *one* "predict" action. This parameter instructs the findBestModel estimator to use the best learner found in this action to score the data in this action, and all other "predict" actions. The name of the learner can be returned for each "predict" action if "learnerForScoring" appears as an evaluator.

In this example, "useForScoring = true" appears in the validScore predict action. As a result, the best learner found for this action will be used to score the data within this action (myValidData) and the data in the trainScore action (myTrainData).

- If no "predict" action contains the parameter "useForScoring = True", then the best learner is selected using the first "predict" action with the "bestLearner" evaluation.
- If multiple "predict" actions contain the parameter "useForScoring=True", then the best learner is selected from the first listed action containing "useForScoring=True".
- Using the best learner that applies to all "predict" actions, users can ask for all typical evaluations such as "prediction", "posteriorProbability", "residuals", etc.
- Currently, it is not possible to export a fitted model using the Find Best Model algorithm, i.e. it's not possible to use "export": "pmml/json" within the "fit" action.

Find Best Model Results

The results below were produced by the Find Best Model Default specification (code above).

- Notice that a model performance table is returned for both "validScore" and "trainScore" because the "modelPerformance" evaluator is included for both actions.
- The best learner was determined for both "validScore" and "trainScore" using the Accuracy metric ("bestLearnerMetric"="ACCURACY")
- Since the parameter "useforscoring=true" is contained within "validScore", the best learner found in this action will be used to score all, in this case both, predict actions, i.e. "validScore" and "validTrain".

```
{
  "status": {
    "id": "2590+fbmFullSpecs+2021-06-21-23-57-22-920899",
    "code": 0,
    "codeText": "Success",
  }
}
```

The status of the solve.


```

"solveTimestamp": "2021-06-21-23-57-30-312839",
"solveTime": 2741
},
"results": {
  "fbmModel.messages": [],
  "trainScore.bestLearner": [],
  "trainScore.modelPerformance": [],
  "trainScore.prediction": [],
  "validScore.bestLearner": [],
  "validScore.modelPerformance": [],
  "validScore.prediction": []
},
"fbmModel": {
  "messages": null
},
"validScore": {
  "modelPerformance": {
    "objectType": "dataFrame",
    "name": "myValidData",
    "order": "col",
    "rowNames": ["Logistic Regression", "Decision Tree",
      "Nearest Neighbors", "Neural Network", "Discriminant
      Analysis", "Bagging", "Boosting", "Random Trees"],
    "colNames": ["Accuracy (#correct)", "Accuracy
      (%correct)", "Specificity", "Sensitivity", "Precision",
      "F1 score"],
    "colTypes": ["double", "double", "double", "double",
      "double", "double"],
    "indexCols": null,
    "data": [
      [190, 187, 172, 163, 189, 193, 194, 193],
      [94.05940594059405, 92.57425742574257,
        85.14851485148515, 80.6930693069307,
        93.56435643564357, 95.54455445544554,
        96.03960396039604, 95.54455445544554],
      [0.9454545454545454, 0.9393939393939394,
        0.9030303030303031, null, 0.9515151515151515,
        0.9515151515151515, 0.9696969696969697,
        0.9515151515151515],
      [0.918918918918919, 0.8648648648648649,
        0.6216216216216216, null, 0.8648648648648649,
        0.972972972972973, 0.918918918918919,
        0.972972972972973],
      [0.7906976744186046, 0.7619047619047619,
        0.5897435897435898, null, 0.8, 0.8181818181818182,
        0.8717948717948718, 0.8181818181818182],
      [0.85, 0.810126582278481, 0.6052631578947368, null,
        0.8311688311688312, 0.8888888888888889,
        0.8947368421052632, 0.8888888888888889]
    ]
  },
  "bestLearner": "Boosting",
  "learnerForScoring": "Boosting",
  "prediction": {
    "objectType": "dataFrame",
    "name": "Prediction",
    "order": "col",

```

Available results.

Warnings/failures that occurred during "fit" will appear here.

Results for "validScore" include the model performance table, the best learner, and scoring results using the best learner.

Note that actual scoring was performed using the BOOSTING ensemble method as specified in the "learnerForScoring" output field.

```

"rowNames": ["Record 1", "Record 2", "Record 3", "Record
...
Record 199", "Record 200", "Record 201", "Record 202"],
"colNames": ["Prediction: CATMEDV"],
"colTypes": ["wstring"],
"indexCols": null,
"data": [
  ["1", "0", "1", "0", "0", "0", "0", "0", "0", "0",
  ...
  "0", "0", "1", "1", "0", "0", "0", "0", "0", "0"]
]
},
"trainScore": {
  "modelPerformance": {
    "objectType": "dataFrame",
    "name": "myTrainData",
    "order": "col",
    "rowNames": ["Logistic Regression", "Decision Tree",
      "Nearest Neighbors", "Neural Network", "Discriminant
      Analysis", "Bagging", "Boosting", "Random Trees"],
    "colNames": ["Accuracy (#correct)", "Accuracy
      (%correct)", "Specificity", "Sensitivity", "Precision",
      "F1 score"],
    "colTypes": ["double", "double", "double", "double",
      "double", "double"],
    "indexCols": null,
    "data": [
      [290, 304, 304, 257, 282, 302, 304, 302],
      [95.39473684210526, 100, 100, 84.53947368421053,
        92.76315789473685, 99.3421052631579, 100,
        99.3421052631579],
      [0.9727626459143969, 1, 1, null, 0.9610894941634242,
        0.9961089494163424, 1, 0.9961089494163424],
      [0.851063829787234, 1, 1, null, 0.7446808510638298,
        0.9787234042553191, 1, 0.9787234042553191],
      [0.851063829787234, 1, 1, null, 0.7777777777777778,
        0.9787234042553191, 1, 0.9787234042553191],
      [0.8510638297872339, 1, 1, null, 0.7608695652173912,
        0.9787234042553191, 1, 0.9787234042553191]
    ]
  },
  "bestLearner": "Decision Tree",
  "learnerForScoring": "Boosting",
  "prediction": {
    "objectType": "dataFrame",
    "name": "Prediction",
    "order": "col",
    "rowNames": ["Record 1", "Record 2", "Record 3", "Record
    ...
    Record 302", "Record 303", "Record 304"],
    "colNames": ["Prediction: CATMEDV"],
    "colTypes": ["wstring"],
    "indexCols": null,
    "data": [
      ["0", "1", "0", "0", "0", "0", "0", "0", "0", "0",
      ...

```

Results for "trainScore" include the model performance table, the best learner, and scoring results using the best learner. Note that scoring was performed using the BOOSTING ensemble method. Since the "useForScoring" parameter was entered for "validScore", the best performing learner from "validScore" is used to score all "predict" actions.

Note: Although the Decision Tree learner was deemed the best learner in "validTrain", since the "useForScoring" parameter exists for "validScore", the "validScore" best learner (BOOSTING) is used to score all predict actions as specified in the "learnerForScoring" output field.

```

        "0", "0", "0", "0", "0", "1", "0", "0"]
    ]
}
}
}

```

Partial Specification

Use this specification of the Find Best Model estimator to select specific learners while using their default parameters. The example code below runs Find Best Model for only three learners: logistic regression, decision trees and the bagging ensemble method. Decision Tree ("decisionTree") is the default weak learner for the bagging and boosting ensemble.

```

...
"estimator": {
  "fbmEstimator": {
    "type": "regression",
    "algorithm": "findBestModel",
    "learners": ["linearRegression", "decisionTree", "bagging"]
  }
},
...

```

Available classification learner names are: "boosting", "bagging", "neuralNetwork", "decisionTree", "randomTrees", "nearestNeighbors", "DiscriminantAnalysis" and "logisticRegression". Note that not all classification/regression learners will be eligible for all classification/regression models. If a classification or regression learner does not appear within the model performance table, see "messages" in the output for the reason.

Full Specification

Use this specification of the Find Best Model estimator to select specific learners and edit their default parameter values.

The example code below runs Find Best Model for three learners: logistic regression, nearest neighbors and the boosting ensemble method. Nearest neighbors is selected for the boosting weak learner with the number of neighbors set to 5. Decision tree ("decisionTree") is the default weak learner for the bagging and boosting ensemble methods. See WeakLearner in the RASON Reference Guide for a complete list of all parameters available for all classification and regression weak learners.

```

...
"estimator": {
  "fbmEstimator": {
    "type": "classification",
    "algorithm": "findBestModel",
    "learners": {
      "logisticRegression": {
        "fitIntercept": false,
      },
      "nearestNeighbors": {
        "numNeighbors": 5,
      },
      "boosting": {
        "weakLearner": {
          "type": "classification",
          "algorithm": "nearestNeighbors",
          "parameters": {
            "numNeighbors": "5"
          }
        }
      }
    }
  }
}

```

```
},
```

```
...
```

As mentioned above, available classification learner names are: "boosting", "bagging", "neuralNetwork", "decisionTree", "randomTrees", "nearestNeighbors", "DiscriminantAnalysis" and "logisticRegression". Note that not all classification/regression learners will be eligible for all classification/regression models. If a classification or regression learner does not appear within the model performance table, see "messages" in the output for the reason.

Classification Using Independent Learner

Next, we will look at an example of how to call a classification model to score data using the RASON modeling language. To open this example, click RASON Example Models on the Editor tab of www.RASON.com, Data Science – Classification – NeuralNetwork.json.

This example imports two datasources (hald-small-binary-train.txt and hald-small-binary-valid.txt) and binds both to a dataset (myTrainData and myValidData, respectively). A fitted model is created, nncEstimator, using Neural Networks which is then fit to the myTrainData dataset. This model, nncModel, is used to score both datasets and produce statistics on how well the model fits each dataset.

```
{
  modelName:  'NueralNetwork',
  modelType:  "datamining",

  modelDescription: 'classification: neural network',
  datasources: {
    myTrainSrc: {
      type: 'csv', connection: 'hald-small-binary-train.txt',
      direction:"import"
    },
    myValidSrc: {
      type: 'csv', connection: 'hald-small-binary-valid.txt',
      direction:"import"
    }
  },
  datasets: {
    myTrainData: {
      binding: 'myTrainSrc', targetCol: 'Y'
    },
    myValidData: {
      binding: 'myValidSrc', targetCol: 'Y'
    }
  },
  estimator: {
    nncEstimator: {
      type: 'classification',
      algorithm: 'neuralNetwork',
      parameters: {
        priorProbMethod: 'EMPIRICAL',
        numNeurons: [ 4 ],
        numEpochs: 100,
        errorTolerance: 0.01,
        weightDecay: 0.0,
        learningRate: 0.4,
        weightMomentum: 0.7,
        hiddenLayerActivation: 'LOGISTIC_SIGMOID',
        outputLayerActivation: 'SOFTMAX',
        weightInitSeed: 123,
        learningOrder: 'RANDOM',

```

```

        learningOrderSeed: 1234,
        dataForErrorComputation: 'TRAIN_AND_VALID',
        maxNumEpochsWithNoImprovement: 30,
        minRelativeErrorChange: 0.00001,
        maxTrainingTimeSeconds: 5
    }
}
},
actions: {
    nncModel: {
        trainData: 'myTrainData',
        validData: 'myValidData',
        export: 'json',
        estimator: 'nncEstimator',
        action: 'fit',
        evaluations: [
            'trainingLog',
            'neuronWeights',
            'numEpochsUsed',
            'trainingTime',
            'stoppingReason',
            'partitionCausedStopping'
        ]
    },
    trainScore: {
        data: 'myTrainData',
        fittedModel: 'nncModel',
        action: 'predict',
        parameters: {
            successClass: '1',
            successProbability: 0.6
        },
        evaluations: [
            'prediction',
            'posteriorProbability',
            'confusionMatrix',
            'accuracy',
            'specificity',
            'sensitivity',
            'recall',
            'precision',
            'f1'
        ]
    },
    validScore: {
        data: 'myValidData',
        fittedModel: 'nncModel',
        action: 'predict',
        parameters: {
            successClass: '1',
            successProbability: 0.6
        },
        evaluations: [
            'prediction',
            'posteriorProbability',
            'confusionMatrix',
            'accuracy',

```

Note: If neither "export" nor "binding" is present within "actions", the fitted model will only be produced in-memory. (An in-memory fitted model may be used in a decision flow.)

```

        'specificity',
        'sensitivity',
        'recall',
        'precision',
        'f1'
    ]
}
}
}

```

In the example above, two datasets are imported in "datasources". The first dataset, hald-small-binary-train.txt, is used to train the Neural Network while hald-small-binary-valid.txt is used to evaluate, or "validate" the fitted model. The first dataset, hald-small-binary-train.txt is imported as myTrainSrc and the second dataset, hald-small-binary-valid.txt, is imported as myValidSrc.

Within "datasets", "myTrainSrc" is bound to "myTrainData" and "myValidSrc" is bound to "myValidData". *Note: Input files in a Data Science RASON model must not contain a path to a file location.*

Within "estimator", the estimator "nncEstimator" is created using the neural network classification algorithm ("type": "classification", "algorithm": "neuralNetwork"). Various options are set for this algorithm under "parameters". For a complete list of the options available to the neural network classification algorithm, see the RASON reference guide.

Three action items are created within "actions", nncModel, trainScore and validScore.

The nncModel action item, fits the neural network classification model, using the estimator nncEstimator, to the training dataset, "myTrainData" and automatically posts the fitted model, in JSON format, to the RASON server using "export":"json". The fitted model is assigned the name given to the modelName setting, i.e. modelName: 'NueralNetwork'. The fitted model will be run on the dataset, myValidData, in order to evaluate the performance of the model. The following results are requested within "evaluations": the training log (trainingLog), the neuron weights (neuronWeights), number of epochs (numEpochsUsed), time to train the model (trainingTime), reason for stopping (stoppingReason) and the partition that caused the algorithm to stop (partitionCausedStopping).

The trainScore action item, scores the myTrainData dataset, using the fitted model nncModel, using a Success Class equal to 1 and a success probability equal to 0.6. The following results are requested for the training partition scoring action item: predictions (prediction), the posterior probabilities (posteriorProbability), the confusion matrix (confusionMatrix), and the 6 statistics (accuracy, specificity, sensitivity, recall, precision and f1).

Lastly, the validScore action item scores the myValidData dataset using the fitted model nncModel. The following results are requested for the validation partition scoring action item: predictions (prediction), the posterior probabilities (posteriorProbability), the confusion matrix (confusionMatrix), and the 6 statistics (accuracy, specificity, sensitivity, recall, precision and f1).

Here are the results from this RASON model.

Getting model results: GET
<https://rason.net/api/model/2590+NeuralNetworkClassification+2020-01-20-01-53-41-109078/result>

```

{
  "status": {
    "id": "2590+NeuralNetworkClassification+2020-01-20-01-53-41-109078",
    "code": 0,
    "codeText": "Success"
  },
  "results": {
    "results": [
      "nncModel.trainingLog",
      "nncModel.neuronWeights",
      "nncModel.numEpochsUsed",
      "nncModel.trainingTime",
      "nncModel.stoppingReason",

```

Start of nncModel action results

```

"nncModel.partitionCausedStopping", "trainScore.prediction",
"trainScore.posteriorProbability", "trainScore.confusionMatrix",
"trainScore.accuracy", "trainScore.specificity",
"trainScore.sensitivity",
"trainScore.recall", "trainScore.precision", "trainScore.f1",
"validScore.prediction", "validScore.posteriorProbability",
"validScore.confusionMatrix", "validScore.accuracy",
"validScore.specificity",
"validScore.sensitivity", "validScore.recall", "validScore.precision",
"validScore.f1"],
"nncModel": {
  "trainingLog": {
    "objectType": "dataFrame",
    "name": "Training Log",
    "order": "col",

```

The trainingLog results, as requested, for nncModel contains 100 rows and 4 columns.

Row names are found under "rowNames", column headings under "colNames" and data for all 4 columns are found under "data".

```

"rowNames": ["Epoch 1", "Epoch 2", "Epoch 3", "Epoch 4", "Epoch 5", "Epoch
12", "Epoch 6", "Epoch 7", "Epoch 8", "Epoch 9", "Epoch 10", "Epoch 11", "Epoch
18", "Epoch 13", "Epoch 14", "Epoch 15", "Epoch 16", "Epoch 17", "Epoch
24", "Epoch 19", "Epoch 20", "Epoch 21", "Epoch 22", "Epoch 23", "Epoch
"Epoch 25", "Epoch 26", "Epoch 27", "Epoch 28", "Epoch 29", "Epoch
30", "Epoch 31", "Epoch 32", "Epoch 33", "Epoch 34", "Epoch 35",
"Epoch 36", "Epoch 37", "Epoch 38", "Epoch 39", "Epoch 40", "Epoch 41",
"Epoch 42", "Epoch 43", "Epoch 44", "Epoch 45", "Epoch 46", "Epoch 47",
"Epoch 48", "Epoch 49", "Epoch 50", "Epoch 51", "Epoch 52", "Epoch 53",
"Epoch 54", "Epoch 55", "Epoch 56", "Epoch 57", "Epoch 58", "Epoch 59",
"Epoch 60", "Epoch 61", "Epoch 62", "Epoch 63", "Epoch 64", "Epoch 65",
"Epoch 66", "Epoch 67", "Epoch 68", "Epoch 69", "Epoch 70", "Epoch 71",
"Epoch 72", "Epoch 73", "Epoch 74", "Epoch 75", "Epoch 76", "Epoch 77",
"Epoch 78", "Epoch 79", "Epoch 80", "Epoch 81", "Epoch 82", "Epoch 83",
"Epoch 84", "Epoch 85", "Epoch 86", "Epoch 87", "Epoch 88", "Epoch 89",
"Epoch 90", "Epoch 91", "Epoch 92", "Epoch 93", "Epoch 94", "Epoch 95",
"Epoch 96", "Epoch 97", "Epoch 98", "Epoch 99", "Epoch 100"],
"colNames": ["Training: Network Error (Cross Entropy)", "Training:
Data Error (Misclassification)",
"Validation: Network Error (Cross Entropy)",
"Validation: Data Error (Misclassification)"],
"colTypes": ["double", "double", "double", "double"],
"indexCols": "null",

```

```
"data": [[0.66303106458569439, 0.61046150383331155,
0.56944006636282729,
  0.52779162039300587, 0.48417504066980516, 0.44362260680789289,
  0.40638475616914688, 0.37215396361664815, 0.3411592111360856,
  0.31431372121567258, 0.29131909802834266, 0.27102389766093798,
  0.25353556252847897, 0.23813822487069872, 0.22470097813635537,
  0.21280936293863378, 0.20228703681028415, 0.19291095351064058,
  0.1845022151930211, 0.1769364291783426, 0.17008566901396463,
  0.16385202086310643, 0.15815736165219668, 0.1529302492035593,
  0.14811227303762814, 0.14365570720572687, 0.13951863318194754,
  0.13566514472547134, 0.13206511877311614, 0.12869251061694048,
  0.12552454521962747, 0.1225415301884923, 0.11972637230850892,
  0.11706403575912855, 0.11454124699820405, 0.11214629679558265,
  0.10986881018953637, 0.10769953657381337, 0.10563020106791242,
  0.1036533877314311, 0.10176242944089263, 0.099951309541604375,
  0.098214582156025687, 0.096547306563537702, 0.09494498943028741,
  0.093403533610519077, 0.091919194140355126, 0.090488540652579968,
  0.089108424598987587, 0.087775950263900768, 0.086488449223837263,
  0.085243457964565922, 0.084038698196981468, 0.082872059416664465,
  0.08174158339000176, 0.080645450347748282, 0.07958196668651682,
  0.078549553981234477, 0.077546739135733622, 0.076572145533259556,
  0.075624485073849626, 0.074702550998768227, 0.073805211411634764,
  0.072931403416608628, 0.072080127805153577, 0.071250444232266827,
  0.070441466830221316, 0.069652360213760117, 0.068882335836058628,
  0.068130648659690554, 0.067396594111116953, 0.066679505290812818,
  0.065978750414207185, 0.065293730461298868, 0.064623877015205694,
  0.063968650272027958, 0.063327537206275833, 0.062700049877748992,
  0.062085723867207059, 0.061484116829451943, 0.06089480715359024,
  0.060317392721253721, 0.059751489754459056, 0.059196731745587997,
  0.058652768462681568, 0.058119265023883551, 0.057595901035437588,
  0.057082369788157505, 0.056578377507750292, 0.056083642654785618,
  0.055597895270477579, 0.055120876364782037, 0.054652337343614694,
  0.05419203947226759, 0.053739753372352525, 0.053295258549820014,
  0.052858342951806907, 0.052428802550251424, 0.052006440950377902,
  0.051591069022308984],

[0.15384615384615385, 0.076923076923076927, 0.076923076923076927,
0.076923076923076927, 0.076923076923076927, 0.076923076923076927,
0.076923076923076927, 0.076923076923076927, 0.076923076923076927,
0.076923076923076927, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[0.66303106458569439, 0.61046150383331155, 0.56944006636282729,
0.52779162039300587, 0.48417504066980516, 0.44362260680789289,
0.40638475616914688, 0.37215396361664815, 0.3411592111360856,
0.31431372121567258, 0.29131909802834266, 0.27102389766093798,
0.25353556252847897, 0.23813822487069872, 0.22470097813635537,
0.21280936293863378, 0.20228703681028415, 0.19291095351064058,
0.1845022151930211, 0.1769364291783426, 0.17008566901396463,
0.16385202086310643, 0.15815736165219668, 0.1529302492035593,
0.14811227303762814, 0.14365570720572687, 0.13951863318194754,
```



```

0.13566514472547134, 0.13206511877311614, 0.12869251061694048,
0.12552454521962747, 0.1225415301884923, 0.11972637230850892,
0.11706403575912855, 0.11454124699820405, 0.11214629679558265,
0.10986881018953637, 0.10769953657381337, 0.10563020106791242,
0.1036533877314311, 0.10176242944089263, 0.099951309541604375,
0.098214582156025687, 0.096547306563537702, 0.09494498943028741,
0.093403533610519077, 0.091919194140355126, 0.090488540652579968,
0.089108424598987587, 0.087775950263900768, 0.086488449223837263,
0.085243457964565922, 0.084038698196981468, 0.082872059416664465,
0.08174158339000176, 0.080645450347748282, 0.07958196668651682,
0.078549553981234477, 0.077546739135733622, 0.076572145533259556,
0.075624485073849626, 0.074702550998768227, 0.073805211411634764,
0.072931403416608628, 0.072080127805153577, 0.071250444232266827,
0.070441466830221316, 0.069652360213760117, 0.068882335836058628,
0.068130648659690554, 0.067396594111116953, 0.066679505290812818,
0.065978750414207185, 0.065293730461298868, 0.064623877015205694,
0.063968650272027958, 0.063327537206275833, 0.062700049877748992,
0.062085723867207059, 0.061484116829451943, 0.06089480715359024,
0.060317392721253721, 0.059751489754459056, 0.059196731745587997,
0.058652768462681568, 0.058119265023883551, 0.057595901035437588,
0.057082369788157505, 0.056578377507750292, 0.056083642654785618,
0.055597895270477579, 0.055120876364782037, 0.054652337343614694,
0.05419203947226759, 0.053739753372352525, 0.053295258549820014,
0.052858342951806907, 0.052428802550251424, 0.052006440950377902,
0.051591069022308984],

[0.15384615384615385, 0.076923076923076927, 0.076923076923076927,
0.076923076923076927, 0.076923076923076927, 0.076923076923076927,
0.076923076923076927, 0.076923076923076927, 0.076923076923076927,
0.076923076923076927, 0.076923076923076927, 0.076923076923076927,
0.076923076923076927, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0,
0, 0,0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0,
0, 0, 0, 0,0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0,
0, 0, 0, 0, 0, 0,0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
]
},
"neuronWeights": {
  "objectType": "dataFrameVector",
  "name": "Neuron Weights",
  "data": {
    "Neuron Weights: Input Layer - Hidden Layer 1": {
      "objectType": "dataFrame",
      "name": "Neuron Weights: Input Layer - Hidden Layer 1",
      "order": "col",
      "rowNames": ["Neuron 1", "Neuron 2", "Neuron 3", "Neuron 4"]
      "colNames": ["X1", "X2", "X3", "X4", "Weights", "Bias"],
      "colTypes": ["double", "double", "double", "double", "double", "double",
      "double"],
      "indexCols": null,
      "data": [
        [-0.75069145845261487, -0.27887307054565263, -
0.79378896450938241,
        0.093324509265532637],

```

```

        [-0.43066792161670547, -0.5410528516228591, -
0.1673473671984495,
        0.36763405948607503],
        [-0.66269427879746901, 0.44816597640320865,
0.022985461449100159, -
        1.1276745743539944],
        [0.01091317921324568, 0.5809695393525508, 0.023724515541765721,
-
        0.23574636017890654],
        [-0.11534712105085403, 0.51441944511523519, -
0.19651109483154025, -
        0.49304953437134286],
        [-0.0056227144792817983, 0.00024935739304918469, -
        0.0009730192415791769, -0.00075515790996579716]
    ]
},
"Neuron Weights": "Hidden Layer 1 - Output Layer": {
    "objectType": "dataFrame",
    "name": "Neuron Weights: Hidden Layer 1 - Output Layer",
    "order": "col",
    "rowNames": ["0", "1"],
    "colNames": ["Neuron 1", "Neuron 2", "Neuron 3", "Neuron 4",
"Bias"],
    "colTypes": ["double", "double", "double", "double", "double"],
    "indexCols": null,
    "data" :[
        [-0.77478453633089717, -0.15371632782460409],
        [1.1850172724323604, -1.8065785521236266],
        [-0.65417688123264595, -0.64439128694926806],
        [-1.7117601902255974, 2.5736303856604104],
        [0.48784434452201758, -0.48784434452201725]
    ]
}
},
"numEpochsUsed": 100,
"trainingTime": 0.1417815,
"stoppingReason": "Maximum number of epochs reached",
"partitionCausedStopping": null"
}
"trainScore": {
    "prediction": {
        "objectType": "dataFrame",
        "name": "Prediction",
        "order": "col",
        "rowNames": ["Record 1", "Record 2", "Record 3", "Record 4", "Record
5",
        "Record 6", "Record 7", "Record 8", "Record 9", "Record 10", "Record
11",
        "Record 12", "Record 13"],
        "colNames": ["Prediction: Y"],
        "colTypes": ["wstring"],
        "indexCols": null,
        "data":[
            ["0", "0", "1", "0", "1", "1", "1", "1", "0", "0", "1", "0", "1", "1"]
        ]
    }
},

```

Under "nncModel" – "actions" – "evaluations", in the RASON model above, we asked for the following results: 'trainingLog', 'neuronWeights', 'numEpochsUsed', 'trainingTime', 'stoppingReason' and 'partitionCausedStopping'.

Results for posteriorProbability contains 13 rows and 2 columns. Row names are found under "rowNames", column headings under "colNames" and data is found under "data".

```

"posteriorProbability": {
  "objectType": "dataFrame",
  "name": "posteriorProbability",
  "order": col,
  "rowNames": ["Record 1", "Record 2", "Record 3", "Record 4", "Record
5",
  "Record 6", "Record 7", "Record 8", "Record 9", "Record 10", "Record
11",
  "Record 12", "Record 13"],
  "colNames": ["0", "1"],
  "colTypes": ["double", "double"],
  "indexCols": null,
  "data": [
    [0.98142563041514053, 0.98142506620239767, 0.035388177536841307,
    0.98141295352694591, 0.036583192233183064, 0.035889568070065693,
    0.035968315202947687, 0.98142532181709696, 0.72414124228947319,
    0.035283978729137452, 0.98142313253302671, 0.035240926303000457,
    0.035240035931403735],
    [0.018574369584859543, 0.018574933797602357, 0.9646118224631588,
    0.018587046473054213, 0.96341680776681693, 0.96411043192993429,
    0.96403168479705237, 0.018574678182903086, 0.27585875771052676,
    0.96471602127086253, 0.018576867466973224, 0.96475907369699965,
    0.96475996406859632]
  ]
},
"confusionMatrix": {
  "objectType": "dataFrame",
  "name": "Confusion Matrix",
  "order": "col",
  "rowNames": ["0", "1"],
  "colNames": ["0", "1"],
  "colTypes": ["double", "double"],
  "indexCols": null,
  "data": [
    [6, 0],
    [0, 7]
  ]
},
"accuracy": 100,
"specificity": 1,
"sensitivity": 1,
"recall": 1,
"precision": 1,
"f1": 1
},
"validScore": {
  "prediction": {
    "objectType": "dataFrame",
    "name": "prediction",
    "order": "col",
    "rowNames": ["Record 1", "Record 2", "Record 3", "Record 4", "Record
5",
    "Record 6", "Record 7", "Record 8", "Record 9", "Record 10", "Record
11",
    "Record 12", "Record 13"],
    "colNames": ["Prediction: Y"],
    "colTypes": ["wstring"],

```

Results for confusion matrix.

```

    "indexCols": null,
    "data": [["0", "0", "1", "0", "1", "1", "1", "0", "0", "1", "0", "1",
"1"]]
  },
  "posteriorProbability": {
    "objectType": "dataFrame",
    "name": "Posterior Probability",
    "order": col,
    "rowNames": ["Record 1", "Record 2", "Record 3", "Record 4", "Record
5", "Record 6", "Record 7", "Record 8", "Record 9", "Record 10",
"Record 11", "Record 12", "Record 13"],
    "colNames": ["0", "1"],
    "colTypes": ["double", "double"],
    "indexCols": null,
    "data": [
      [0.98142563041514053, 0.98142506620239767, 0.035388177536841307,
0.98141295352694591, 0.036583192233183064, 0.035889568070065693,
0.035968315202947687, 0.98142532181709696, 0.72414124228947319,
0.035283978729137452, 0.98142313253302671, 0.035240926303000457,
0.035240035931403735],
      [0.018574369584859543, 0.018574933797602357, 0.9646118224631588,
0.018587046473054213, 0.96341680776681693, 0.96411043192993429,
0.96403168479705237, 0.018574678182903086, 0.27585875771052676,
0.96471602127086253, 0.018576867466973224, 0.96475907369699965,
0.96475996406859632]
    ]
  },
  "confusionMatrix": {
    "objectType": "dataFrame",
    "name": "Confusion Matrix",
    "order": "col",
    "rowNames": ["0", "1"],
    "colNames": ["0", "1"],
    "colTypes": ["double", "double"],
    "indexCols": null,
    "data": [[6, 0], [0, 7]]
  },
  "accuracy": 100,
  "specificity": 1,
  "sensitivity": 1,
  "recall": 1,
  "precision": 1,
  "f1": 1
}
}

```

Results for confusionMatrix contain 2 rows and 2 columns. Row names are found under "rowNames", column headings are found under "colNames" and data is found under "data".

Synthetic Data Generation

The newly released Synthetic Data Generation feature included in the latest version of RASON Decision Services allows users to generate synthetic data by automated Metalog probability distribution selection and parameter fitting, Rank Correlation or Copula fitting, and random sampling. This can be beneficial for several reasons such as when the actual training data is limited or when the data owner is unwilling to release the actual, full dataset but agrees to supply a limited copy or a synthetic version that statistically resembles the properties of the actual dataset.

This process consists of three main steps.

1. Fit and select a marginal probability distribution to each feature – by automated and semi-automated search within the family of bounded, semi-bounded or unbounded Metalog distributions.
2. Identify correlations among features, by using Rank Correlation or one of available Copulas – Clayton, Gumbel, Frank, Student or Gauss.
3. Generate the random sample consistent with the best-fit probability distributions and correlations.

Additional to the generated synthetic data, RASON Decision Services can optionally provide the details of the fitting process – fitted coefficients and goodness-of-fit metrics for all fitted candidate Metalog distributions, selected distribution for each feature and fitted correlation matrix.

To further explore the original/synthetic data and compare them, one may easily compute basic and advanced statistics for original and/or synthetic data, including but not limited to percentiles and Six Sigma metrics.

Three examples are provided to illustrate how to use Synthetic Data Generation. Each example demonstrates the use of the SyntheticDataGenerator object in the RASON Modeling language.

All examples can be downloaded, POSTed and Solved using the Editor tab at www.RASON.com. To download, click the Open RASON Example Model icon (on the Editor tab at www.RASON.com), then click Data Mine – Summarizer – Summarization.json, to download Summarization.json, Data Mine – Simulation – SyntheticDataGeneration.json and Data Mine – Simulation – LinearRegressionSimulation to download the Linear Regression Simulation example. The data file, bh-scale-reg.txt, can be downloaded by clicking the Download RASON Example Data icon.

1. The first example, Summarization Data Example, illustrates how to obtain statistics, percentiles and six sigma metrics on a dataset.
2. The second example, Synthetic Data Generator, is a standalone example illustrating the use of the SyntheticDataGenerator object.
3. The third example, Linear RegressionSimulation, illustrates how to perform the Risk Analysis of a fitted Multiple Regression model using synthetic data generation techniques.

Summarization Data Example

This example illustrates how a user could obtain statistics, percentiles and six sigma metrics on a dataset.

Summarization Example Code

```
{
  "modelName": "summarization",
  "datasources": {
    "mySrc": {
      "type": "csv",
      "connection": "hald-small.txt"
    }
  },
  "datasets": {
    "myData": {
      "binding": "mySrc"
    }
  },
  "transformer": {
    "summarizer": {
      "type": "transformation",
      "algorithm": "summarization",
      "parameters": {
        "numBins": [
          [ "X2", 4 ],
          [ "X4", 5 ]
        ]
      }
    }
  }
}
```

Specifies number of bins to be used for each variable. This is optional.

```

    ]
  }
},
"actions": {
  "mySummaries": {
    "data": "myData",
    "action": "transform",
    "evaluations": [
      "summary",
      "advancedSummary",
      "sixSigma",
      "percentiles",
      "histogram"
    ],
    "parameters": {
      "semiVariancePower": 2,
      "semiDeviationPower": 2,
      "varPercentile": 0.95,
      "cvarPercentile": 0.95,
      "meanConfidencePercentile": 0.95,
      "sixSigmaPercentileLB": 0.05,
      "sixSigmaPercentileUB": 0.95,
      "sixSigmaShift": 0.0,
      "sixSigmaNumStdDev": 6.0,
      "sixSigmaTarget": 0.0
    }
  }
}
}
}

```

Summary statistics, advanced summary statistics, six sigma metrics, percentiles and variable histograms will be included in the output.

Parameter values are included for select statistics and six sigma metrics. All are optional, default settings will be used if not specified.

Note: In order to obtain statistics, percentiles, Six Sigma metrics, histograms, etc, on the results of synthetic data generator, predictions, etc., the Summarizer above may be called as a stage in a decision flow. For more information on decision flows, see the [Creating and Running a Decision Flow](#) chapter that appears earlier in this guide.

Computed Statistics, Percentiles and Six Sigma Metrics

All statistics generated using the Summarizer evaluations are briefly described below.

Summary

- **Mean**, the average of all the values.
- **Standard Deviation**, the square root of variance.
- **Variance**, the spread of the distribution of values.
- **Skewness**, which describes the *asymmetry* of the distribution of values.
- **Kurtosis**, which describes the *peakedness* of the distribution of values.
- **Mode**, the most frequently occurring single value.
- **Minimum**, the minimum value attained.
- **Maximum**, the maximum value attained.
- **Range**, the difference between the maximum and minimum values.

Advanced Summary

- **Mean Abs. Deviation**, returns the average of the absolute deviations.
- **SemiVariance**, measure of the dispersion of values.
- **SemiDeviation**, *one-sided* measure of dispersion of values.
- **Value at Risk 95%**, the maximum loss that can occur at a given confidence level.
- **Cond. Value at Risk**, is defined as the *expected value* of a loss *given that* a loss at the specified percentile occurs.
- **Mean Confidence**, returns the confidence “half-interval” for the estimated mean value (returned by the PsiMean() function).
- **Std. Dev. Confidence 95%**, returns the confidence ‘half-interval’ for the estimated standard deviation of the simulation trials (returned by the PsiStdDev() function).
- **Coefficient of Variation**, is defined as the ratio of the standard deviation to the mean.
- **Standard Error**, defined as the standard deviation of the sample mean.
- **Expected Loss**, returns the average of all negative data multiplied by the percentrank of 0 among all data.
- **Expected Loss Ratio**, returns the expected loss ratio.
- **Expected Gain** returns the average of all positive data multiplied by 1 - percentrank of 0 among all data.
- **Expected Gain Ratio**, returns the expected gain ratio.
- **Expected Value Margin**, returns the expected value margin.

Percentiles

- Generates numeric percentile values (from 1% to 99%) computed using all values for the variable. For example, the 75th Percentile value is a number such that three-quarters of the values occurring in the last simulation are less than or equal to this value.

Six Sigma Metrics

Generates various computed Six Sigma measures, described below. These functions compute values related to the Six Sigma indices used in manufacturing and process control.

- **SigmaCP**: SixSigmaCP (cell,lower_limit,upper_limit)

A Six Sigma index, SixSigmaCP predicts what the process is capable of producing if the process mean is centered between the lower and upper limits. This index assumes the process output is normally distributed.

$$Cp = \frac{UpperSpecificationLimit - LowerSpecificationLimit}{6\hat{\sigma}}$$

where $\hat{\sigma}$ is the estimated standard deviation of the process.

- **SigmaCPK**: SixSigmaCPK (cell,lower_limit,upper_limit)

A Six Sigma index, SixSigmaCPK predicts what the process is capable of producing if the process mean is not centered between the lower and upper limits. This index assumes the process output is normally distributed and will be negative if the process mean falls outside of the lower and upper specification limits.

$$Cpk = \frac{MIN(UpperSpecificationLimit - \hat{\mu}, \hat{\mu} - LowerSpecificationLimit)}{3\hat{\sigma}}$$

where $\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.

- **SigmaCPKLower:** SixSigmaCPKLower(cell,lower_limit)

A Six Sigma index, SixSigmaCPKLower calculates the one-sided Process Capability Index based on the lower specification limit. This index assumes the process output is normally distributed.

$$Cp, lower = \frac{\hat{\mu} - LowerSpecificationLimit}{3\hat{\sigma}}$$

where $\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.

- **SigmaCPKUpper:** SixSigmaCPKUpper(cell,upper_limit)

A Six Sigma index, SixSigmaCPKUpper calculates the one-sided Process Capability Index based on the upper specification limit. This index assumes the process output is normally distributed.

$$Cp, upper = \frac{UpperSpecificationLimit - \hat{\mu}}{3\hat{\sigma}}$$

where $\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.

- **SigmaCPM:** SixSigmaCPM(cell,lower_limit,upper_limit,target)

A Six Sigma index, SixSigmaCPM calculates the capability of the process around a target value. This index is referred to as the Taguchi Capability Index. This index assumes the process output is normally distributed and is always positive.

$$Cpm = \frac{\hat{Cp}}{\sqrt{1 + (\frac{\hat{\mu} - T}{\hat{\sigma}})^2}}$$

where \hat{Cp} is the process capability (SigmaCP), $\hat{\mu}$ is the process mean, $\hat{\sigma}$ is the standard deviation of the process and T is the target process mean.

- **SigmaDefectPPM:** SixSigmaDefectPPM(cell,lower_limit,upper_limit)

A Six Sigma index, SixSigmaDefectPPM calculates the Defective Parts per Million.

$$DPMO = (\delta^{-1}(\frac{LowerSpecificationLimit - \hat{\mu}}{\hat{\sigma}}) + 1 - \delta^{-1}(\frac{UpperSpecificationLimit - \hat{\mu}}{\hat{\sigma}})) * 1000000$$

where $\hat{\mu}$ is the process mean, $\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.

- **SigmaDefectShiftPPM:** SixSigmaDefectPPM(cell,lower_limit,upper_limit)

A Six Sigma index, SixSigmaDefectShiftPPM calculates the Defective Parts per Million with an added shift.

$$DPMOshift = (\delta^{-1}(\frac{LowerSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift) + 1 - \delta^{-1}(\frac{UpperSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift)) * 1000000$$

where $\hat{\mu}$ is the process mean, $\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.

- **SigmaDefectShiftPPMLower:** SixSigmaDefectShiftPPMLower(cell,lower_limit,shift)

A Six Sigma index, SixSigmaDefectShiftPPMLower calculates the Defective Parts per Million, with a shift, below the lower specification limit.

$$DPMOshift, lower = (\delta^{-1}(\frac{LowerSpecificationLimit}{\hat{\sigma}} - Shift) * 1000000$$

Where $\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.

- **SigmaDefectShiftPPMUpper:** SixSigmaCPKUpper(cell,upper_limit)

A Six Sigma index, SixSigmaDefectShiftPPMUpper calculates the Defective Parts per Million, with a shift, above the lower specification limit.

$$DPMOshift, upper = (\delta^{-1}(\frac{UpperSpecificationLimit}{\hat{\sigma}} - Shift) * 1000000$$

where $\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.

- **SigmaK: SixSigmaK(cell,lower_limit,upper_limit)**

A Six Sigma index, SixSigmaK calculates the Measure of Process Center and is defined as:

$$1 - \frac{2 * MIN(UpperSpecificationLimit - \hat{\mu}, \hat{\mu} - LowerSpecificationLimit)}{UpperSpecificationLimit - LowerSpecificationLimit}$$

where $\hat{\mu}$ is the process mean.

- **SigmaLowerBound: SixSigmaLowerBound(cell,number_stdev)**

A Six Sigma index, SixSigmaLowerBound calculates the Lower Bound as a specific number of standard deviations below the mean and is defined as:

$$\hat{\mu} - \hat{\sigma} * \#StandardDeviations$$

where $\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.

- **SigmaProbDefectShift: SixSigmaProbDefectShift(cell,lower_limit, upper_limit, shift)**

A Six Sigma index, SixSigmaProbDefectShift calculates the Probability of Defect, with a shift, outside of the upper and lower limits. This statistic is defined as:

$$\delta^{-1} \left(\frac{LowerSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift \right) + 1 - \delta^{-1} \left(\frac{UpperSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift \right)$$

where $\hat{\mu}$ is the process mean, $\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.

- **SigmaProbDefectShiftLower: SixSigmaProbDefectShiftLower(cell,lower_limit, shift)**

A Six Sigma index, SixSigmaProbDefectShiftLower calculates the Probability of Defect, with a shift, outside of the lower limit. This statistic is defined as:

$$\delta^{-1} \left(\frac{LowerSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift \right)$$

where $\hat{\mu}$ is the process mean, $\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.

- **SigmaProbDefectShiftUpper: SixSigmaProbDefectShiftUpper(cell,upper_limit, shift)**

A Six Sigma index, SixSigmaProbDefectShiftUpper calculates the Probability of Defect, with a shift, outside of the upper limit. This statistic is defined as:

$$1 - \delta^{-1} \left(\frac{UpperSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift \right)$$

where $\hat{\mu}$ is the process mean, $\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.

- **SigmaSigmaLevel: SixSigmaSigmaLevel(cell,lower_limit,upper_limit, shift)**

A Six Sigma index, SixSigmaSigmaLevel calculates the Process Sigma Level with a shift. This statistic is defined as:

$$-\delta \left(\delta^{-1} \left(\frac{LowerSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift \right) + 1 - \delta^{-1} \left(\frac{UpperSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift \right) \right)$$

where $\hat{\mu}$ is the process mean, $\hat{\sigma}$ is the standard deviation of the process δ is the standard normal cumulative distribution function, and δ^{-1} is the standard normal inverse cumulative distribution function.

- **SigmaUpperBound: SixSigmaUpperBound(cell,number_stdev)**

A Six Sigma index, SixSigmaUpperBound calculates the Upper Bound as a specific number of standard deviations above the mean and is defined as:

$$\hat{\mu} - \hat{\sigma} * \#StandardDeviations$$

where $\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.

- **SigmaYield: SixSigmaYield(cell,lower_limit,upper_limit,shift)**

A Six Sigma index, SixSigmaYield calculates the Six Sigma Yield with a shift, or the fraction of the process that is free of defects. This statistic is defined as:

$$\delta^{-1}\left(\frac{UpperSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift\right) - \delta^{-1}\left(\frac{LowerSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift\right)$$

where $\hat{\mu}$ is the process mean, $\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.

- **SigmaZLower: SixSigmaZLower(cell,lower_limit)**

A Six Sigma index, SixSigmaZLower calculates the number of standard deviations of the process that the lower limit is below the mean of the process. This statistic is defined as:

$$\frac{\hat{\mu} - LowerSpecificationLimit}{\hat{\sigma}}$$

where $\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.

- **SigmaZMin: SixSigmaZMin(cell,lower_limit,upper_limit)**

A Six Sigma index, SixSigmaZMin calculates the minimum of SigmaZLower and SigmaZUpper. This statistic is defined as:

$$\frac{MIN(\hat{\mu} - LowerSpecificationLimit, UpperSpecificationLimit - \hat{\mu})}{\hat{\sigma}}$$

where $\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.

- **SigmaZUpper: SixSigmaZUpper(cell,upper_limit)**

SixSigmaZUpper(cell,upper_limit,simulation)

A Six Sigma index, SigmaZUpper calculates the number of standard deviations of the process that the upper limit is above the mean of the process. This statistic is defined as:

$$\frac{UpperSpecificationLimit - \hat{\mu}}{\hat{\sigma}}$$

Where $\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.

Synthetic Data Generator Example

This example demonstrates the API of the SyntheticDataGenerator object in the RASON modeling language, consistent with the 3-step process defined above: fit and select distributions and correlations for all columns (variables) in the dataset and generate the synthetic data. The original/synthetic data can be further analyzed using “summarization” functionality as shown in the previous example.

Synthetic Data Example Code

```
{
  "modelName": "syntheticDataGeneration",
  "datasources": {
    "mySrc": {
      "type": "csv",
```

```

    "connection": "bh-scale-reg.txt"
  },
  "datasets": {
    "myData": {
      "binding": "mySrc"
    }
  },
  "transformer": {
    "sdgTransformer": {
      "type": "transformation",
      "algorithm": "syntheticDataGenerator",
      "params": {
        "metalogAuto": true,
        "numMetalogTerms": [
          ["CRIM", 5],
          ["ZN", 5],
          ["INDUS", 5],
          ["NOX", 5],
          ["RM", 5],
          ["DIS", 5],
          ["AGE", 5],
          ["TAX", 5],
          ["PTRATIO", 5],
          ["B", 5],
          ["LSTAT", 5],
          ["MEDV", 5]
        ],
        "metalogLowerBound": [
          ["CRIM", 0.00632],
          ["ZN", 0],
          ["INDUS", 0.46],
          ["NOX", 0.385],
          ["RM", 3.561],
          ["DIS", 1.1296],
          ["AGE", 2.9],
          ["TAX", 187],
          ["PTRATIO", 12.6],
          ["B", 0.32],
          ["LSTAT", 1.73],
          ["MEDV", 5]
        ],
        "metalogUpperBound": [
          ["CRIM", 88.9762],
          ["ZN", 100],
          ["INDUS", 27.74],
          ["NOX", 0.871],
          ["RM", 8.78],
          ["DIS", 12.1265],
          ["AGE", 100],
          ["TAX", 711],
          ["PTRATIO", 22],
          ["B", 396.9],
          ["LSTAT", 37.97],
          ["MEDV", 50]
        ]
      },
      "useMinMaxAsBounds": true,

```

Specifies number of terms in the Metalog distributions

Lower bounds for Metalog distributions

Upper bounds for Metalog distributions

```

    "metalogGoodnessOfFitType": [
      ["CRIM", "ANDERSON_DARLING"],
      ["ZN", "ANDERSON_DARLING"],
      ["INDUS", "ANDERSON_DARLING"],
      ["NOX", "ANDERSON_DARLING"],
      ["RM", "ANDERSON_DARLING"],
      ["DIS", "ANDERSON_DARLING"],
      ["AGE", "ANDERSON_DARLING"],
      ["TAX", "ANDERSON_DARLING"],
      ["PTRATIO", "ANDERSON_DARLING"],
      ["B", "ANDERSON_DARLING"],
      ["LSTAT", "ANDERSON_DARLING"],
      ["MEDV", "ANDERSON_DARLING"]
    ],
    "computeMetalogCurves": true,

    "correlationType": "RANK",
    "claytonCopula" : true,
    "frankCopula" : true,
    "gumbelCopula" : true,
    "gaussCopula" : true,
    "studentCopula" : true,

    "sampleSize": 10,
    "randomSeed": 12345,
    "randomGeneratorType": "MERSENNE_TWISTER",
    "samplingMethodType": "LATIN_HYPERCUBE",
    "randomStreamType": "INDEPENDENT"
  }
},
"actions": {
  "syntheticData": {
    "data": "myData",
    "action": "transform",
    "evaluations": [
      "log",
      "bounds",
      "metalogFitted",
      "bestMetalog",
      "metalogCoefficients",
      "metalogGOF",
      "metalogCurve",
      "correlationFitted",
      "selectedCopula",
      "correlationSigma",
      "copulaTheta",
      "copulaDF",
      "transformation"
    ]
  }
}
}

```

Goodness-of-Fit tests for
Metalog distributions

Options settings, see below for
details.

Evaluations to be included in the
output

Synthetic Data Generation Options

RASON Decision Services provides the following options for Synthetic Data Generation.

Metalog Distribution Fitting Options

- **Metalog Terms**

"metalogAuto": true or false

- If **False**, Rason Decision Services will attempt to fit the Metalog distribution with the specified number of terms.

Example: "metalogAuto": true

- **Number of Terms for Metalog Distributions**

"numMetalogTerms": [colname, N], where N = integer from [2, 16]

Example:

```
"numMetalogTerms": [
    ["CRIM", 5],
    ["ZN", 5],
    ["INDUS", 5],
    ["NOX", 5],
    ["RM", 5],
    ["DIS", 5],
    ["AGE", 5],
    ["TAX", 5],
    ["PTRATIO", 5],
    ["B", 5],
    ["LSTAT", 5],
    ["MEDV", 5]
],
```

Sets the number of terms for the Metalog Distribution for a given column.

- If **True**, Rason Decision Services will attempt to fit *all possible* Metalog distributions, with the number of terms limited by the specified value, and select the best Metalog distribution according to the chosen Goodness-of-Fit test.

Example: "MetalogAuto": true

- **Max Terms for Metalog Distributions**

NumMetalogTerms[colname, N], where N = integer from [2, 16]

Example:

```
"numMetalogTerms": [
    ["CRIM", 5],
    ["ZN", 5],
    ["INDUS", 5],
    ["NOX", 5],
    ["RM", 5],
    ["DIS", 5],
    ["AGE", 5],
    ["TAX", 5],
    ["PTRATIO", 5],
]
```

```

["B", 5],
["LSTAT", 5],
["MEDV", 5]
],

```

Sets the max number of terms for Metalog distribution for a given column.

- **Goodness of Fit Test**

```

MetalogGoodnessOfFitType[colname,type]

```

```

Type = CHI_SQUARE, KOLMOGOROV_SMIRNOV, ANDERSON_DARLING, AIC,
BIC, AICc, BICc, MAX_LIKELIHOOD

```

Example:

```

"metalogGoodnessOfFitType": [
    ["CRIM", "CHI_SQUARE"],
    ["ZN", "KOLMOGOROV_SMIRNOFF"],
    ["INDUS", "ANDERSON_DARLING"],
    ["NOX", "AIC"],
    ["RM", "AICc"],
    ["DIS", "BIC"],
    ["AGE", "BICc"],
    ["TAX", "MAX_LIKELIHOOD"],
],

```

The Goodness of Fit test is used to select the best Metalog form for each column among the candidate distributions defined by a different number of terms, from 2 to the value passed for NumMetalogTerms. The default Goodness-of-Fit test is Anderson-Darling.

RASON Decision Services offers the following Goodness of Fit Tests:

- **Chi Square** – Uses the chi-square statistic to rank the distributions. Sample data is first divided into intervals using either equal probability, then the number of points that fall into each interval are compared with the expected number of points in each interval. The null hypothesis is rejected using a 90% significance level, if the chi-squared test statistic is greater than the critical value statistic.
- **Kolmogorov-Smirnoff** – This test computes the difference (D) between the continuous distribution function (CDF) and the empirical cumulative distribution function (ECDF). The null hypothesis is rejected if, at the 90% significance level, D is larger than the critical value statistic.
- **[Default] Anderson -Darling** – Ranks the fitted distributions using the Anderson Darling statistic, A2. The null hypothesis is rejected using a 90% significance level, if A2 is larger than the critical value statistic. This test awards more weight to the distribution tails than the Kolmogorov-Smirnoff test.
- **AIC** – The AIC test is a Chi Squared test corrected for the number of distribution parameters and sample size. $AIC = \text{Chi-Square Statistic} + 2 * k + 2 * k * (k + 1) / (n - k - 1)$ where k is the number of distribution parameters and n is the sample size.
- **AICc** – When the sample size is small, there is a significant chance that the AIC test will select a model with many parameters. In other words, AIC will overfit the data. AICc was developed to reduce the possibility of overfitting by applying a penalty to the number of parameters. Assuming that the model is univariate, is linear in the parameters and has normally distributed residuals, the formula for AICc is: $AICc = AIC + 2k / (n - k - 1)$ where n = sample size, k = # of parameters. As the sample size approaches infinity, the penalty on the number of parameters converges to 0 resulting in AICc converging to AIC.
- **BIC** – The Bayesian information criterion (BIC) is defined as: $BIC = k \ln(n) = 2 \ln(L)$ where L = the maximized value of the likelihood function of the model M. $L = p(x|\theta, M)$

where θ are the parameter values that maximize the likelihood function and x is the observed data. n = Sample size k = Number of parameters

BICc – The BICc is the alternative version of BIC, corrected for the sample size $BICc = BIC + 2 * p * (p + 1) / (n - p - 1)$

- Maximum Likelihood (ML) – The (negated) raw value of the estimated maximum log likelihood utilized in tests described above.

- **Set Upper and Lower Bounds for Metalog distribution**

As mentioned above, RASON Decision Services supports unbounded, semi-bounded and bounded Metalog distributions. The API for setting up semi-bounded or bounded distributions is explained below.

"UseMinMaxAsBounds": True sets the lower/upper bounds as minimum/maximum of each variable. However, if a lower or upper bound is manually set, RASON Decision Services will give priority to the manually set bounds while keeping the minimum/maximum for those variables where the bounds were not set manually.

To check the Metalog bounds for all columns at once use "Bounds" in "Evaluations".

- Manually: Set each upper and lower bound separately using:

```
MetalogLowerBound[colname, lowerBound]
```

```
MetalogUpperBound[colname, upperBound]
```

Example:

```
"metalogLowerBound": [
    ["CRIM", 0.00632],
    ["ZN", 0],
    ["INDUS", 0.46],
    ["NOX", 0.385],
    ["RM", 3.561],
    ["DIS", 1.1296],
    ["AGE", 2.9],
    ["TAX", 187],
    ["PTRATIO", 12.6],
    ["B", 0.32],
    ["LSTAT", 1.73],
    ["MEDV", 5]
],
"metalogUpperBound": [
    ["CRIM", 88.9762],
    ["ZN", 100],
    ["INDUS", 27.74],
    ["NOX", 0.871],
    ["RM", 8.78],
    ["DIS", 12.1265],
    ["AGE", 100],
    ["TAX", 711],
    ["PTRATIO", 22],
    ["B", 396.9],
    ["LSTAT", 37.97],
    ["MEDV", 50]
],
```

- Automatically: Set UseMinMaxAsBounds to True to use the minimum and maximum values in each column as the lower and upper bound for each variable.

"UseMinMaxAsBounds": True or False

Example: `"UseMinMaxAsBounds": True`

If a bound has been set manually, `UseMinMaxAsBounds` will not overwrite the existing bound.

- **Generate Metalog Curves**

Set `computeMetalogCurves` to true to compute Metalog PDF curves the selected Metalog distribution for all columns.

`"computeMetalogCurves": True or False`

Example: `"computeMetalogCurves": true,`

Correlation Fitting Options

- **Correlation Method**

Use `CorrelationType` to fit a correlation between the variables. If this option is set to "None", then no correlation fitting will be performed.

`"CorrelationType": "None"`

Otherwise, there are two options for correlation fitting: rank (`"CorrelationType": "RANK"`) and copula (`"CorrelationType": "COPULA"`).

- If *Rank* is selected, Spearman rank order correlation will be used to fit a correlation matrix for all columns. To select Rank use: `"CorrelationType": "RANK"`.
- If *Copula* is selected, correlation will be fit using specified copulas. To select a Copula use:
`"CorrelationType": "COPULA"`.
 - If `"CorrelationType": "COPULA"`, then copulas may be specified by setting the individual copula to true. If multiple copulas are selected, the first successfully fit copula will be used in the sample generation.

RASON Decision Services offers 5 types of copulas:

- STUDENT
- CLAYTON
- FRANK
- GUMBEL
- GAUSS



The Spearman rank order correlation coefficient is a *nonparametric* measure of correlation that is computed from a rank ordering of the trial values drawn for all variables. It can be used to induce correlations between *any* two uncertain variables, whether they have the same or different analytic distributions, or even custom distributions. This correlation coefficient ranges in value from -1 to +1.



RASON Decision Services includes copulas to improve the method of defining the correlation or dependence between two or more variables. Copulas offer more flexibility over the rank order correlation method, and are able to capture complex correlations.

An n – dimensional copula C is a multi-variate probability distribution where the marginal probability distribution of each variable follows the Uniform(0,1) distribution. A major benefit of copulas is that they allow two or more uncertain variables to be correlated without changing the shape of the original uncertain variable distributions.

For some copula C , a multi-variate distribution F with distributions of F_1, F_2, \dots, F_n can be written as:

$$F(x_1, \dots, x_n) = C(F_1(x_1), F_2(x_2), \dots, F_n(x_n))$$

RASON Decision Services supports five types of copulas: three Archimedean

Generating Data Options

- Sample Size

`"sampleSize": N`, where N is any positive integer

Use this option to set the size of the generated sample. The default is 100.

Example: `"sampleSize": = 10`,

- Random Seed

`"randomSeed": N`, where N is any positive integer

Setting the random number seed to a nonzero value (any number of your choice is OK) ensures that the *same* sequence of random numbers is used for each simulation. When the seed is zero or RandomSeed is not specified, the random number generator is initialized from the system clock, so the sequence of random numbers will be different in each simulation. Set the seed to ensure that the results from one simulation to another are strictly comparable.

Example: `"randomSeed": = 12345`,

- Random Generator

`"randomGeneratorType": <Type>`

Type = HDR, LECUYER_CMGR, MERSENNE_TWISTER, PARK_MILLER, WELL

Use this option to select a random number generation algorithm. RASON Decision Services includes an advanced set of random number generation capabilities.

Computer-generated numbers are never truly “random,” since they are always computed by an algorithm – they are called *pseudorandom* numbers. A random number generator is designed to quickly generate sequences of numbers that are as close to being statistically independent as possible. Eventually, an algorithm will generate the same numbers seen sometime earlier in the sequence, and at this point the sequence will begin to repeat. The *period* of the random number generator is the number of values it can generate before repeating.

A long period is desirable, but there is a tradeoff between the length of the period and the degree of statistical independence achieved within the period. Hence, RASON Decision Services offers a choice of five random number generators:

- *Park-Miller* (PARK_MILLER) “Minimal” Generator with Bayes-Durham shuffle and safeguards. This generator has a period of $2^{31}-2$. Its properties are good, but the following choices are usually better.

EXAMPLE: `"randomGeneratorType": "PARK_MILLER"`,

- Combined Multiple Recursive Generator of L’Ecuyer (LECUYER_CMGR). This generator has a period of 2^{191} , and excellent statistical independence of samples within the period.

EXAMPLE: `"randomGeneratorType": "LECUYER_CMGR"`,

- Well Equidistributed Long-period Linear (WELL) generator of Panneton, L’Ecuyer and Matsumoto. This generator combines a long period of 2^{1024} with very good statistical independence.

EXAMPLE: "randomGeneratorType": "WELL",

- *Mersenne Twister* (default setting - MERSENNE_TWISTER) generator of Matsumoto and Nishimura. This generator has the longest period of $2^{19937}-1$, but the samples are not as “equidistributed” as for the WELL and L-Ecuyer-CMRG generators.

EXAMPLE: "randomGeneratorType": "MERSENNE_TWISTER",

- The *HDR* Random Number Generator (HDR), designed by Doug Hubbard. Permits data generation running on various computer platforms to generate identical or independent streams of random numbers.

EXAMPLE: "randomGeneratorType": "HDR",

- Sampling Method

"samplingMethodType": <type>,

Type = MONTE_CARLO, LATIN_HYPERCUBE, SOBOLO_RQMC

Use this option to select *Monte Carlo*, *Latin Hypercube*, or *Sobol RQMC* sampling.

- *Monte Carlo*: In standard Monte Carlo sampling, numbers generated by the chosen random number generator are used directly to obtain sample values. With this method, the variance or estimation error in computed samples is inversely proportional to the square root of the number of trials (controlled by the Sample Size); hence to cut the error in half, four times as many trials are required.

EXAMPLE: "samplingMethodType": "MONTE_CARLO",

RASON provides two other sampling methods than can significantly improve the ‘coverage’ of the sample space, and thus reduce the variance in computed samples. This means that you can achieve a given level of accuracy (low variance or error) with fewer trials.

- *Latin Hypercube (default)*: Latin Hypercube sampling begins with a stratified sample in each dimension (one for each selected variable), which constrains the random numbers drawn to lie in a set of subintervals from 0 to 1. Then these one-dimensional samples are combined and randomly permuted so that they ‘cover’ a unit hypercube in a stratified manner.

EXAMPLE: "samplingMethodType": "LATIN_HYPERCUBE",

- *Sobol RQMC (Randomized QMC)*. Sobol numbers are an example of so-called “Quasi Monte Carlo” or “low-discrepancy numbers,” which are generated with a goal of coverage of the sample space rather than “randomness” and statistical independence. Analytic Solver Data Science adds a “random shift” to Sobol numbers, which improves their statistical independence.

EXAMPLE: "samplingMethodType": "SOBOL_RQMC",

- Random Stream

"RandomStreamType": <type>

Type = INDEPENDENT or SINGLE

Use this option to select a *Single Stream* or an *Independent Stream (the default)* for each variable.

EXAMPLE: "randomStreamType": "SINGLE"

EXAMPLE: "randomStreamType": "INDEPENDENT"

If *Single Stream* is selected, a single sequence of random numbers is generated. Values are taken consecutively from this sequence to obtain samples for each selected variable. This introduces a subtle dependence between the samples for all distributions in one trial. In many applications, the effect is too small to make a difference – but in some cases, better results are obtained if *independent* random number sequences (streams) are used for each distribution in the model. RASON Decision Services offers this capability for Monte Carlo sampling and Latin Hypercube sampling; it does not apply to Sobol numbers.

Synthetic Data Generation Results

The following Evaluations allow users to obtain the results of the Synthetic Data Generation.

Metalog Fitting Results

- MetalogFitted – Checks whether there was at least one feasible Metalog distribution fitted for all or specific columns.

Example: "metalogFitted"

- MetalogCoefficients – Obtains the coefficients for fitted Metalog Distributions for all or specific columns.

Example: "metalogCoefficients"

Example ODATA Results:

Table		syntheticData_metalogCoefficients_Metalog_Coefficients__AGE					Columns		All columns
ID	Name	C1	C2	C3	C4				
0	2	2.7500811463024633	2.7373582179938087						
1	3	0.571734746449768	2.7373582179938083	4.362691831206549					
2	4	0.5717347464497686	4.592869737160488	4.362691831206548	-12.139935960902505				

- BestMetalog – Use BestMetalog to obtain the number of terms for the best fitted Metalog distribution for each column.

Example: "bestMetalog"

Example ODATA Results:

Table		syntheticData_bestMetalog		
ID	Name	Terms		
0	CRIM	5		
1	ZN	2		
2	INDUS	3		
3	NOX	5		
4	RM	3		
5	AGE	4		
6	DIS	3		
7	TAX	3		
8	PTRATIO	3		
9	B	3		
10	LSTAT	2		
11	MEDV	2		

- MetalogGOF – Gets the detailed report of Goodness of Fit tests for fitted Metalog Distributions for all or specific columns.

Example: "MetalogGOF"

Example ODATA Results:

Table		syntheticData_metalogGOF_Metalog_Goodness_of_Fit_AGE							Columns		All columns	
ID	Name	CS	KS	AD	ML	AIC	AICc					
0	2	443.32806324110675	0.28131933767883655	176.8019000584984	2316.352724879155	4640.70544975831	4640.785290077672					
1	3	385.699604743083	0.19067978228804291	149.14985835777884	2277.707114799885	4565.41422959977	4565.53422959977					
2	4	292.02371541501975	0.13338016112716378	129.48387125903446	2225.1797473423685	4462.359494684737	4462.527831358084					

Correlation Results

- CorrelationFitted – Checks if the correlations have been fitted.

Example: "correlationFitted"

Obtains information on the correlation fitting.

- "selectedCopula" – Determines what copula was selected if copula fitting was requested.
- "correlationSigma" – Obtains the correlation matrix , when applicable – i.e. if rank correlation or Gauss/Student Copula was used.
- "copulaTheta" – Use this result to obtain the fitted correlation theta value when selected copula is Clayton, Frank or Grumbel.
- "copulaDF" – Use this result to obtain the degrees of freedom when copula is Student.

Linear Regression Simulation Example

The example below illustrates how to perform the Risk Analysis of a fitted Multiple Regression model using synthetic data generation techniques. The methods demonstrated below are applicable to any supervised classification or regression method.

See the Synthetic Data Generation example above for explanations on Step 1, Step 2 and Step 3.

Important Note

Steps 1, 2 and 3, described in the introduction to this section, may be condensed to a single line of code by simply calling

```
"simulation": {}
```

within the estimator. In this example, the estimator would change to:

```
"estimator": {
  "mlrEstimator": {
    "type": "regression",
    "algorithm": "linearRegression",
    "parameters": {
      "fitIntercept": true
    },
    "simulation": {
      "metalogAuto": true,
      "useMinMaxAsBounds": true,
      "correlationType": "RANK",
      "sampleSize": 506,
      "randomSeed": 12345,
      "expression": "IF (CRIM<10, MEDV, 2*MEDV)"
    },
  },
}
```

If a user opts to forgo setting options for the Synthetic Data Generator then the data will be fit to unbounded Metalog functions with up to 5 terms, a sample size of 100 and rank correlation.

```
"simulation": {}
```

Linear Regression Simulation Example Code

```
{
  "comment": "regression: linear model with simulation",
  "modelDescription": "This example demonstrates the automated risk
    analysis of a machine learning model",
  "modelName": "mlr-sim",
  "datasources": {
    "myTrainSrc": {
      "type": "csv",
      "connection": "bh-scale-reg.txt"
    }
  },
  "datasets": {
    "myTrainData": {
      "binding": "myTrainSrc",
      "targetCol": "MEDV"
    }
  },
  "estimator": {
    "mlrEstimator": {
```

```

"type": "regression",
"algorithm": "linearRegression",
"parameters": {
  "fitIntercept": true
},
"simulation": {
  "metalogAuto": true,
  "numMetalogTerms": [
    ["CRIM", 5],
    ["ZN", 5],
    ["INDUS", 5],
    ["NOX", 5],
    ["RM", 5],
    ["DIS", 5],
    ["AGE", 5],
    ["TAX", 5],
    ["PTRATIO", 5],
    ["B", 5],
    ["LSTAT", 5]
  ],
  "metalogLowerBound": [
    ["CRIM", 0.00632],
    ["ZN", 0],
    ["INDUS", 0.46],
    ["NOX", 0.385],
    ["RM", 3.561],
    ["DIS", 1.1296],
    ["AGE", 2.9],
    ["TAX", 187],
    ["PTRATIO", 12.6],
    ["B", 0.32],
    ["LSTAT", 1.73]
  ],
  "metalogUpperBound": [
    ["CRIM", 88.9762],
    ["ZN", 100],
    ["INDUS", 27.74],
    ["NOX", 0.871],
    ["RM", 8.78],
    ["DIS", 12.1265],
    ["AGE", 100],
    ["TAX", 711],
    ["PTRATIO", 22],
    ["B", 396.9],
    ["LSTAT", 37.97]
  ],
  "useMinMaxAsBounds": true,
  "metalogGoodnessOfFitType": [
    ["CRIM", "ANDERSON_DARLING"],
    ["ZN", "ANDERSON_DARLING"],
    ["INDUS", "ANDERSON_DARLING"],
    ["NOX", "ANDERSON_DARLING"],
    ["RM", "ANDERSON_DARLING"],
    ["DIS", "ANDERSON_DARLING"],
    ["AGE", "ANDERSON_DARLING"],
    ["TAX", "ANDERSON_DARLING"],
    ["PTRATIO", "ANDERSON_DARLING"],
  ]
}

```

The keyword, "simulation", within the estimator, invokes the synthetic data generator.

See the Synthetic Data Generator example above for explanations and option descriptions found within the contents of "simulation":{ }.

```

        ["B", "ANDERSON_DARLING"],
        ["LSTAT", "ANDERSON_DARLING"]
    ],

    "correlationType": "RANK",
    "claytonCopula" : true,
    "frankCopula" : true,
    "gumbelCopula" : true,
    "gaussCopula" : true,
    "studentCopula" : true,

    "sampleSize": 10,
    "randomSeed": 12345,
    "randomGeneratorType": "MERSENNE_TWISTER",
    "samplingMethodType": "LATIN_HYPERCUBE",
    "randomStreamType": "INDEPENDENT",

    "expression": "IF(CRIM<10, MEDV, 2*MEDV)"
  }
},
"actions": {
  "mlrModel": {
    "trainData": "myTrainData",
    "estimator": "mlrEstimator",
    "action": "fit",
    "evaluations": [
      "simulationLog",
      "simulationPrediction",
      "simulationData",
      "simulationExpression",
      "summary",
      "advancedSummary",
      "sixSigma",
      "percentiles",
      "histogram"
    ]
  },
  "training": {
    "data": "myTrainData",
    "fittedModel": "mlrModel",
    "action": "predict",
    "evaluations": [
      "prediction",
      "trainingExpression",
      "summary",
      "advancedSummary",
      "sixSigma",
      "percentiles",
      "histogram"
    ]
  }
}
}

```

Use “expression” to calculate a business outcome or decision. It is a function of input features and a predicted response, and may be any valid Excel formula that references a variable and the response.

mlrModel: In this action, the linear regression model, mlrEstimator, is used to fit a model to the training data. Evaluations requested to be included in the output are the simulation log, the simulation predictions, the synthetic data, the results of the expression when applied to the synthetic data and summary and advanced statistics, Six Sigma metrics, percentiles and histogram.

training: In this action, the linear regression from, mlrModel, is used to score the training dataet. Similar evaluations from mlrModel are requested to be included in the output. See output examples below.

The keyword “simulation” is called within the estimator, mlrEstimator, to invoke synthetic data generation. Options and parameters pertaining to the synthetic data generator are called within this section. (See above for

descriptions on all available options and parameters.) Use “expression” to calculate a business outcome or decision in the output. In this example the expression is: `IF (CRIM<10, MEDV, 2*MEDV)`. If the value of the CRIM feature is less than 10, in each record, then return the value of the MEDV feature, otherwise return the value of the MEDV variable, multiplied by 2.

Summary statistics, advanced summary statistics, six sigma metrics, percentiles and variable histograms will be included in the output.

There are four available evaluations for synthetic data generator: `simulationLog`, `simulationPrediction`, `simulationData` and `simulationExpression`. Note that the summarizer evaluations (summary, advancedSummary, sixSigma, percentiles and histogram - all described previously) are also present in this example illustrating that these evaluations are also available for output when the simulation keyword is present within the estimator.

- `simulationLog`: Any errors, warnings or general messages generated during the synthetic data generation will appear in the simulation log in the output. For example, if `numMetalogTerms` is not specified for any variables, the error, “Number of Metalog terms is not set for <varname>” will be generated in the log. If no errors occurred or no messages or warnings were generated, `simulationLog` will report as “null”.
- `simulationPrediction`: Returns the predictions, produced by the fitted model, for the output variable in the synthetic data.

SimulationPrediction ODATA Results

Table: mlrModel_simulationPrediction			
ID	Name	Prediction__MEDV	
0	Record 1	7.4859485550743665	
1	Record 2	19.33800158862268	
2	Record 3	28.80607239500627	
3	Record 4	25.0210131769636	
4	Record 5	36.9549320722286	
5	Record 6	26.397748675281722	
6	Record 7	25.414320831469844	
7	Record 8	24.99997157823908	
8	Record 9	15.650310886852973	
9	Record 10	7.749433211263474	

- `simulationData`: Returns the synthetically generated data.

SimulationData ODATA Results

Table: mlrModel_simulationData				Columns	All columns		
ID	Name	CRIM	ZN	INDUS	NOX	RM	AG
0	Record 1	3.0173948744659613	0.000001992200954126554	12.15049025239329	0.5442180626015405	4.812844155157227	97.7130745321
1	Record 2	1.6845962479531356	3.613179109905842e-11	21.184745133545587	0.8576092603190157	5.925323751552714	54.2457723449
2	Record 3	1.117614650663902	3.352215317960864e-8	10.092911449686719	0.536574512922895	7.425321170332638	73.3791299582
3	Record 4	0.49751696953100283	3.6873003557906403e-7	6.970254430585414	0.5353726899055936	6.129281777459723	50.9023773700
4	Record 5	0.02706598869987465	0.0003126767941516971	2.5186212436805246	0.4455545319860708	8.160977851369024	46.9650210066
5	Record 6	0.1507294719087859	0.00007261630631265232	19.34507059003818	0.5481893391783497	6.844534387246951	99.786895221
6	Record 7	0.06643543439223107	0.000013017962673342018	3.69317787045981	0.5235672386444615	6.59806198421498	37.1162177235
7	Record 8	0.10151637096289569	18.528301974918165	6.1290381312539415	0.4658275085484027	6.378467126498372	93.9220811284
8	Record 9	0.30608566989236385	0.03667927052153165	1.5828033668582833	0.39865802357640395	5.676958785141758	3.85720699359
9	Record 10	60.198117995355	5.552783028808233e-9	27.593931959314364	0.6179938029366677	5.289697724659054	99.9992630233

- `simulationExpression`: Results from the formula provided to “expression”.

Note: If `simulationExpression` is present in “evaluations” but no expression is passed in the estimator, `simulationExpression` will fail and an error will be reported in the `simulationLog`.

SimulationExpression ODATA Results

Table mlrModel_simulationExpression		
ID	Name	Expression
0	Record 1	7.4859485550743665
1	Record 2	19.33800158862268
2	Record 3	28.806072395006275
3	Record 4	25.021013176963603
4	Record 5	36.9549320722286
5	Record 6	26.397748675281722
6	Record 7	25.414320831469848
7	Record 8	24.99997157823908
8	Record 9	15.650310886852973
9	Record 10	15.498866422526946

Recall the expression:
"expression": "IF (CRIM<10,
MEDV, 2*MEDV) "

Data Science Risk Analysis Decision Flow

Here is an example of a decision flow computing simulation-related evaluations, predictions and summary, six sigma metrics, percentiles and histograms for the predicted values in the training dataset and in the simulated data. All output is reported as a Data Frame with ODATA immediately available. This file can be opened on the Editor tab of www.RASON.com under Open RASON Example Model – Decision Flows – Inline – Data Science – Synthetic Data Generation”.

The evaluation, “trainingExpression”, in “predict” action is now available in the latest release of RASON Decision Services. This evaluation returns the results of the expression, if present, for the training partition.

1st Stage, mlr-sim-stage: This stage includes two actions: “mlr-model” and “training”.

Within the “mlr-model” action, a linear regression model is fit to the my-train-data dataset using the mlr-estimator estimator. The keyword “simulation” is called within mlr-estimator to invoke synthetic data generation.

This action requests four evaluations, all pertaining to the simulated data: simulationLog, simulationPrediction, simulationData and simulationExpress. (See above for descriptions of each output type.) These evaluations may be called in a subsequent stage(s) using:

- mlr-sim-stage.mlr-model.simulationLog
- mlr-sim-stage.mlr-model.simulationPrediction
- mlr-sim-stage.mlr-model.simulationData
- mlr-sim-stage.mlr-model.simulationExpression

The “training” action in this 1st stage uses the fitted model from mlr-model to predict the values for the output variable in the training dataset, my-train-data. This action requests two evaluations, prediction and trainingExpression. Prediction results will contain the predicted values for the output variable in the my-train-data dataset and trainingExpression will contain the results of the expression also applied to the my-train-data dataset.

2nd Stage, summarization-stage: The 2nd stage accepts four input parameters, all are output from the 1st stage, mlr-sim-stage: mlr-sim-stage.training.prediction, mlr-sim-stage.training.trainingExpression, mlr-sim-stage.mlr-model.simulationPrediction and mlr-sim-stage.mlr0model.simulationExpression. The transformer, summarizer, uses the summarization algorithm to compute summary and advanced summary statistics, six sigma metrics percentile values and histograms.

This stage includes 4 “actions”. Each action has a set of evaluations.

- The first action, training-prediction-summary, generates summary and advanced summary statistics, six-sigma metrics, percentile values and histograms for the records in the *training* dataset (mlr-sim-stage.training.prediction).
- The second action, train-express-summary, generates summary and advanced summary statistics, six-sigma metrics, percentile values and histograms for the predictions in the records in the *training dataset filtered using the expression*, (mlr-sim-stage.training.trainingExpression).
- The third action, sim-prediction-summary, generates summary and advanced summary statistics, six-sigma metrics, percentile values and histograms for the records in the *simulated*, or synthetic, data (mlr-sim-stage.mlr-model.simulationPrediction).
- The fourth action, sim-expression-summary, generates summary and advanced summary statistics, six-sigma metrics, percentile values and histograms for records in the *simulated data filtered using the expression*, (mlr-sim-stage.mlr-model.simulationExpression).

DM-RISK-ANALYSIS-FLOW EXAMPLE CODE

```
{
  "flowName": "dm-risk-analysis-flow",
  "mlr-sim-stage":
  {
    "comment": "regression: linear model with simulation",
    "modelName": "mlr-sim",
    "datasources":
    {
      "my-train-src":
      {
        "type": "csv",
        "connection": "bh-scale-reg.txt"
      }
    },
    "datasets":
    {
      "my-train-data":
      {
        "binding": "my-train-src",
        "targetCol": "MEDV"
      }
    },
    "estimator":
    {
      "mlr-estimator":
      {
        "type": "regression",
        "algorithm": "linearRegression",
        "parameters":
        {
          "fitIntercept": true
        },
        "simulation":
        {
          "metalogAuto": true,
          "useMinMaxAsBounds": true,
          "correlationType": "RANK",
          "sampleSize": 506,
          "randomSeed": 12345,
          "expression": "IF (CRIM<10, MEDV, 2*MEDV) "
```

Beginning of 1st stage

Input datasource

Datasource converted to dataset
(This step can not be skipped.)

mlr-estimator: Creates a regression
estimator using Linear Regression; one
option is set, fit intercept is set to true.

simulation: If this key word is present,
synthetic data will be generated using the
options settings specified. In this
example, 5 options are set: metalogAuto,
useMinMaxAsBounds, correlationType,
sampleSize and randomSeed. See above
for descriptions of all available options.

Use "expression" to apply any valid
Excel formula that references a variable
and the response, to both the synthetic
and training data. This is optional.

```

    }
  },
  "actions":
  {
    "mlr-model":
    {
      "data": "my-train-data",
      "estimator": "mlr-estimator",
      "action": "fit",
      "evaluations":
      [
        "simulationLog",
        "simulationPrediction",
        "simulationData",
        "simulationExpression"
      ]
    },
    "training":
    {
      "data": "my-train-data",
      "fittedModel": "mlr-model",
      "action": "predict",
      "evaluations":
      [
        "prediction",
        "trainingExpression"
      ]
    }
  },
  "summarization-stage":
  {
    "modelName": "summarization",
    "inputParameters":
    {
      "train-prediction":
      {
        "value": "mlr-sim-stage.training.prediction"
      },
      "train-expression":
      {
        "value": "mlr-sim-stage.training.trainingExpression"
      },
      "sim-prediction":
      {
        "value": "mlr-sim-stage.mlr-model.simulationPrediction"
      },
      "sim-expression":
      {
        "value": "mlr-sim-stage.mlr-model.simulationExpression"
      }
    },
    "transformer":
    {
      "summarizer":
      {
        "type": "transformation",

```

mlr-model: Uses the linear regression estimator, mlr-estimator, to fits a model to the training dataset (bh-scale-reg.txt). Four evaluations, simulationLog, simulationPrediction, simulationData and simulationExpression, will be returned.

training: The linear regression model fit in mlr-model is then used to score the training dataset, bh-scale-reg.txt. Two evaluations, prediction and trainingExpression, will be returned.

summarization-stage: This stage accepts four inputs. All four are outputs from the 1st stage, mlr-stage. For more information, see description above.

summarizer: Creates the summarization transformer, summarizer, which, when applied to each input, will generate summary and advanced statistics, six sigma metrics and percentile values.

```

        "algorithm": "summarization"
    },
    "actions":
    {
        "train-prediction-summary":
        {
            "data": "train-prediction",
            "action": "transform",
            "evaluations":
            [
                "summary",
                "advancedSummary",
                "sixSigma",
                "percentiles",
                "histogram"
            ]
        },
        "train-expression-summary":
        {
            "data": "train-expression",
            "action": "transform",
            "evaluations":
            [
                "summary",
                "advancedSummary",
                "sixSigma",
                "percentiles",
                "histogram"
            ]
        },
        "sim-prediction-summary":
        {
            "data": "sim-prediction",
            "action": "transform",
            "evaluations":
            [
                "summary",
                "advancedSummary",
                "sixSigma",
                "percentiles",
                "histogram"
            ]
        },
        "sim-expression-summary":
        {
            "data": "sim-expression",
            "action": "transform",
            "evaluations":
            [
                "summary",
                "advancedSummary",
                "sixSigma",
                "percentiles",
                "histogram"
            ]
        }
    }
}

```

actions: All four actions, train-prediction, train-expression, sim-prediction and sim-expression, use the summarizer transformer to generate summary and advanced statistics, Six Sigma metrics, percentiles and histograms for the four inputs. See results below.

DM-RISK-ANALYSIS-FLOW EXAMPLE RESULTS

See below to find a portion of the DM-Risk-Analysis-Flow example decision flow. All results are also returned in JSON. To view the JSON results in the Editor tab of www.RASON.com, click the Show JSON results icon on the top right of the Editor window.



- ODATA results for the “summary” evaluation for “train-prediction-summary” action: Summary statistics for the predicted values in the training dataset.

Table	summarization_stage_train_prediction_summary_summary							Columns	All columns
ID	Name	Type	Mean	Sum	Abs__Sum	StdDev	Variance		
0	Prediction: MEDV	Numeric	22.532806324110634	11401.59999999998	11409.94664039504	7.817305898643934	61.110271512973235	0.095125	

- ODATA results for “advancedsummary” evaluation for “train-expression-summary” action: Advanced statistics based on the results of the expression, "IF (CRIM<10, MEDV, 2*MEDV) " as applied to the training dataset.

Table	summarization_stage_train_expression_summary_advancedSummary						Columns	All columns
ID	Name	Mean_Abs__Deviation	SemiVariance	SemiDeviation	Value_at_Risk_95_000000pct	Cond_Value_at_Risk_95_0000		
0	Expression	6.131737957870959	27.549188768921358	5.248732110607414	-12.659725501045761	22.9534485		

- OData results for “sixSigma” evaluation for “sim-prediction-summary” action: Six Sigma metrics for the predicted values in the generated synthetic data.

Table	summarization_stage_sim_prediction_summary_sixSigma							Columns	All columns
ID	Name	Cp	Cpk	Cpk_lower	Cpk_upper	Cpm	PPM		
0	MEDV	0.5618540683927047	0.5096394785257552	0.614068658259654	0.5096394785257552	0.22380747967668782	95864.84609459223	3	

- OData results for “percentiles” evaluation for “sim-expression-summary” action: Percentiles values for the results of the expression, "IF (CRIM<10, MEDV, 2*MEDV) ", when applied to the synthetic data.

Table	summarization_stage_sim_expression_summary_percentiles		
ID	Name	Expression	
0	1%	-5.437969827708645	
1	2%	-0.7747499864575728	
2	3%	0.3050346212965677	
3	4%	4.120136142718392	
4	5%	6.051699946643482	
5	6%	8.197483845843752	
6	7%	8.560891490649729	
7	8%	9.050702639447954	
8	9%	10.26031088234907	
9	10%	11.01573158825201	
10	11%	11.670626374508474	
11	12%	12.577957980486973	
12	13%	12.759796045947148	
13	14%	13.106522202753348	
14	15%	13.4857229351416	

- ODataResults for “histogram” evaluation for “sim-expression-summary” action: Bin values and Count information for the results of the expression, "IF (CRIM<10, MEDV, 2*MEDV, when applied to the synthetic data.


```

"modelDescription": "classification: NN model scoring from JSON",
"modelType": "datamining",
"datasources": {
  "labeledDataSrc": {
    "type": "csv",
    "connection": "hald-small-binary.txt",
    "direction": "import"
  },
  "unlabeledDataSrc": {
    "type": "csv",
    "connection": "hald-small-binary-score.txt",
    "direction": "import"
  }
}

```

The datasets section binds both data sources (labeledDataSrc and unlabeledDataSrc) to two datasets (labeledData and unlabeledData).

```

"datasets": {
  "labeledData": {
    "binding": "labeledDataSrc",
    "targetCol": "Y"
  },
  "unlabeledData": {
    "binding": "unlabeledDataSrc"
  }
},

```

The name of the fitted model previously POSTed to the RASON Server, in the neural network classification example above, is used in the creation of the nncModel fitted model.

```

"fittedModel": {
  "nncModel": {
    "modelName": "NueralNetwork"
  }
},

```

The name of the fitted model residing on the RASON server.

Within the actions section, labeledDataPrediction scores the data within hald-small-binary.txt (the labeled data) using the fitted model, nncModel. The success class is "1" and the success Probability is "0.6". Four evaluations are requested in the results: the predictions, posterior probabilities, confusion matrix and fitting metrics.

```

"actions": {
  "labeledDataPrediction": {
    "data": "labeledData",
    "fittedModel": "nncModel",
    "export": "json",
    "action": "predict",
    "parameters": {
      "successClass": "1",
      "successProbability": 0.6
    },
    "evaluations": [
      "prediction",
      "posteriorProbability",
      "confusionMatrix",
      "metrics"
    ]
  },
},

```

While labeledDataPrediction scores the data within hald-small-binary-score.txt (the unlabeled or new data) using the fitted model, nncModel. The success class is "1" and the success Probability is "0.6". Two evaluations are requested in the results: the predictions and the posterior probabilities.

```
"unlabeledDataPrediction": {
  "comment": "scores data in hald-small-binary-score.txt using fitted
  model,
  classification-nn-model.json and asks for results.",
  "data": "unlabeledData",
  "fittedModel": "nncModel",
  "action": "predict",
  "parameters": {
    "successClass": "1",
    "successProbability": 0.6
  },
  "evaluations": [
    "prediction",
    "posteriorProbability"
  ]
}
}
```

Scoring with Fitted Model as Datasource

The next example illustrates how to provide a fitted model as a datasource when scoring new data.

The model begins by importing 3 data files, the first two are the same as the two files used in the previous scoring example, the third file contains the fitted model exported by the example model found on the Editor tab at www.RASON.com (RASON example models – Data Science – Classification – Fitted Models as Datasources – NeuralNetwork.json).

```
{
  "modelName": "JSONClassifierNN",
  "modelDescription": "classification: NN model scoring from JSON",
  "modelType": "datamining",
  "datasources": {
    "labeledDataSrc": {
      "type": "csv",
      "connection": "hald-small-binary.txt",
      "direction": "import"
    },
    "unlabeledDataSrc": {
      "type": "csv",
      "connection": "hald-small-binary-score.txt",
      "direction": "import"
    },
    "jsonModelSrc": {
      "comment": "This file produced by NeurelNetwork.json classification
      model (under Rason Example Models-Data Science-Classification-Fitted
      models as data sources)",
      "type": "json",
      "content": "json-model",
      "connection": "classification-neural-network.json",
      "direction": "import"
    }
  }
}
```


The datasets section binds both data sources (labeledDataSrc and unlabeledDataSrc) to two datasets (labeledData and unlabeledData). This code remains unchanged from above.

```
"datasets": {
  "labeledData": {
    "binding": "labeledDataSrc",
    "targetCol": "Y"
  },
  "unlabeledData": {
    "binding": "jsonModelSrc"
  }
},
```

The name of the fitted model imported in datasources is bound to the fitted model, nncModel.

```
"fittedModel": {
  "nncModel": {
    "binding": "NueralNetwork"
  }
},
```

The rest of the RASON model script remains unchanged.

```
"actions": {
  "labeledDataPrediction": {
    "data": "labeledData",
    "fittedModel": "nncModel",
    "export": "json",
    "action": "predict",
    "parameters": {
      "successClass": "1",
      "successProbability": 0.6
    },
    "evaluations": [
      "prediction",
      "posteriorProbability",
      "confusionMatrix",
      "metrics"
    ]
  },
},
```

While labeledDataPrediction scores the data within hald-small-binary-score.txt (the unlabeled or new data) using the fitted model, nncModel. The success class is "1" and the success Probability is "0.6". Two evaluations are requested in the results: the predictions and the posterior probabilities.

```
"unlabeledDataPrediction": {
  "comment": "scores data in hald-small-binary-score.txt using fitted
  model,
  classification-nn-model.json and asks for results.",
  "data": "unlabeledData",
  "fittedModel": "nncModel",
  "action": "predict",
  "parameters": {
    "successClass": "1",
    "successProbability": 0.6
  },
  "evaluations": [
    "prediction",
    "posteriorProbability"
  ]
}
```

```

    }
  }
}

```

Perform Risk Analysis Given Fitted Model Stored in PMML/JSON Format

In the latest version of RASON Decision Services a full risk analysis may be performed given an already fitted model that is stored in either PMML or JSON format. Note that the RASON model below does not include the “fit” action. Training data is supplied in order to generate the synthetic dataset. (It is possible to use the new data for this task but in many cases, this data is limited.) The set of (optional) evaluations is the same as if the RASON model included the actual fitted step for a supervised model – synthetic data, predictions, expressions, statistics, etc. – all are included in the “evaluations” section. (See below.) This example uses the “linear-fitted” PMML model which must *first* be POSTed on the RASON Decision Services server.

```

{
  "modelDescription": "regression: linear model scoring from pmml and risk
    analysis",
  "modelName": "pmmlRiskScorer",
  {
    "trainDataSource":
    {
      "type": "csv",
      "connection": "bh-scale-reg.txt"
    },
    "newDataSource":
    {
      "type": "csv",
      "connection": "bh-scale-reg-score.txt"
    }
  },
  "datasets":
  {
    "trainData":
    {
      "binding": "trainDataSource",
      "targetCol": "MEDV"
    },
    "newData":
    {
      "binding": "newDataSource"
    }
  },
  "fittedModel":
  {
    "mlrModel":
    {
      "modelName": "linear-fitted",
      "simulation":
      {
        "metalogAuto": true,
        "useMinMaxAsBounds": true,
        "correlationType": "RANK",
        "sampleSize": 506,
        "randomSeed": 12345,
        "expression": "IF (CRIM<10, MEDV, 2*MEDV) "
      }
    }
  },
  "actions":

```

Datasource containing training data

Datasource containing new data

Training dataset

New data

mlrModel – Uses “linear-fitted” fitted model to score the data. In this example, “linear-fitted” is used to score both the new data and the generated synthetic data. The fitted model, “linear-fitted”, is contained within the JSON file, linear-fitted.xml. This fitted model must first be POSTed to the RASON Decision Server. See below for instructions on how to obtain and POST this model.

Because the keyword “simulation” is present in mlrModel, synthetic data will be generated. See above for explanations of all simulation options.

```

{
  "trainPrediction":
  {
    "data": "trainData",
    "fittedModel": "mlrModel",
    "action": "predict",
    "evaluations":
    [
      "prediction",
      "simulationLog",
      "simulationData",
      "simulationPrediction",
      "simulationExpression",
      "trainingExpression",
      "summary",
      "advancedSummary",
      "sixSigma",
      "percentiles",
      "histogram"
    ]
  },
  "newPrediction":
  {
    "data": "newData",
    "fittedModel": "mlrModel",
    "action": "predict",
    "evaluations":
    [
      "prediction",
      "summary",
      "advancedSummary",
      "sixSigma",
      "percentiles",
      "histogram"
    ]
  }
}

```

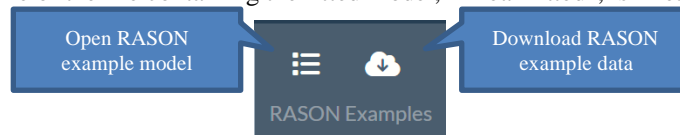
“trainPrediction”: Action uses “mlrModel” to score the training data and the synthetic data, and returns the evaluations in the output.

“newPrediction”: Action uses “mlrModel” to score the new data and returns the evaluations in the output.

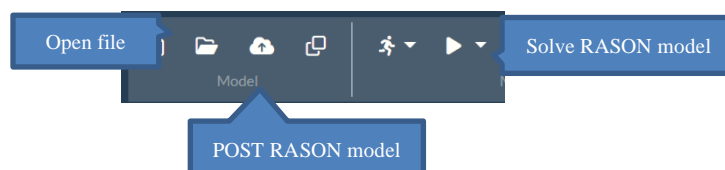
POSTing and Solving the Model

To generate the synthetic data, you must first POST the fitted model to the RASON server.

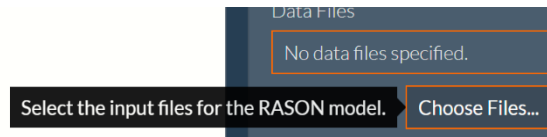
1. Click the “Download RASON example data” icon on the Editor tab ribbon at www.RASON.com, to download the example data files required for the RASON example models. All data files are contained in datafiles.zip. The name of the file containing the fitted model, “linear-fitted”, is linear-fitted.xml.



2. Click the Open file icon on the Editor tab ribbon and browse to the location where linear-fitted.xml is saved.
3. Once the model appears in the Editor window, click the POST icon to post the fitted model to the RASON Server.



- Click Data Science – Simulation – RiskScoringPMML.json, to open the example model shown above.
- Click Choose Files on the Properties pane (on the right of the Editor tab) to select the data source files, bh-scale-reg.txt and bh-scale-reg-score.txt, also contained in datafiles.zip.



- Click the POST icon to post the RASON model and the data files to the RASON Server, then click the Solve icon to solve the model. See below for a portion of the results of the evaluations.

A portion of the evaluations from the two actions, trainPrediction and newPrediction, can be found below.

- Evaluation from trainPrediction: simulationData – Risk analysis results (synthetic data) for the training dataset.

Unnamed-1 x pmmLRiskScorer...846549 x							
Table		Columns		All columns			
ID	Name	CRIM	ZN	INDUS	NOX	RM	
0	Record 1	3.051293250167163	1.2132789143592572e-11	26.15493353462016	0.6318625251564732	5.880710737012649	99.98415
1	Record 2	0.128231589825542	1.4180269614236255e-7	6.458641253213398	0.5373955023179724	6.87497852442618	98.84603
2	Record 3	0.7328779825326913	0.0006017166203319334	6.869914419414779	0.5342879348484294	7.286696090645346	54.2807
3	Record 4	0.9496692312964846	0.000020856436488746126	17.085106992279094	0.5499475560865241	5.18163813220474	91.97629
4	Record 5	0.06961096228804975	4.597239848117763e-7	9.746238992152625	0.5322100382693336	5.018179063055442	56.23651
5	Record 6	2.914469162856663	2.0907571647353632e-8	18.341418517738898	0.5374149765341572	6.523825026994264	77.62994
6	Record 7	9.739568015565638	1.4542733468951713e-15	27.098036915237014	0.8369797929323161	4.588144147211613	99.99606
7	Record 8	12.657347152390678	8.881018565416679e-7	27.399093003909343	0.8267389750555221	4.9514732017298	99.25318
8	Record 9	0.0830146444273958	8.399346116220465e-7	7.786819477335855	0.5093696729532454	6.910992499231959	64.73867
9	Record 10	7.1673935742269705	5.111332798776512e-8	19.865703701759188	0.5416088430679327	6.445913394012469	71.0941
10	Record 11	0.048725594036333676	0.0000025852608215993505	15.53412183455224	0.48738715103720154	5.387339597724165	58.774661
11	Record 12	0.4395416648077273	8.260384938596446e-12	6.960456559162599	0.7810259827873098	6.849641625934261	96.09192
12	Record 13	0.5965140751431866	0.0002548995443051326	9.411821722669297	0.5493908307293172	6.03342225157295	90.56399

- Evaluation from trainPrediction: Predicted values for the training data (actual data).

Table			trainPrediction_prediction
ID	Name	Prediction__MEDV	
0	Record 1	31.505258861679994	
1	Record 2	25.182369161689998	
2	Record 3	31.071081301709995	
3	Record 4	28.82287332863	
4	Record 5	28.18985636595	
5	Record 6	25.219812847149996	
6	Record 7	22.844416122710008	
7	Record 8	19.34123597045001	
8	Record 9	11.096661282760003	
9	Record 10	18.68029704196	
10	Record 11	18.87863920811001	
11	Record 12	21.367169105530014	
12	Record 13	20.77237058822	

- Evaluation from trainPrediction: Predicted values for the synthetic data.

Table trainPrediction_simulationPrediction		
ID	Name	MEDV
0	Record 1	19.725772163987607
1	Record 2	24.863313840828873
2	Record 3	35.690842341213006
3	Record 4	11.211072337911457
4	Record 5	13.09005158834971
5	Record 6	19.542505030633993
6	Record 7	-1.1750105674506903
7	Record 8	1.5503478728428954
8	Record 9	30.287913712000034
9	Record 10	18.1161215503364
10	Record 11	11.094098559737033
11	Record 12	20.86405719798458
12	Record 13	23.346388366521026

- Evaluation from newPrediction: Predicted values for the new data

Table newPrediction_prediction		
ID	Name	Prediction__MEDV
0	Record 1	31.505258861679994
1	Record 2	25.182369161689998
2	Record 3	31.071081301709995
3	Record 4	28.82287332863
4	Record 5	28.18985636595
5	Record 6	25.219812847149996
6	Record 7	22.844416122710008
7	Record 8	19.34123597045001
8	Record 9	11.096661282760003
9	Record 10	18.68029704196

Custom Type Definitions

Introduction

In past versions of RASON, type definitions were not required as optimization and simulation models dealt exclusively with numeric values. However, with the recent introduction of decision tables and custom functions, RASON Decision Services is now supporting custom type definitions. With this new service, RASON Decision Services now conforms to DMN Specification Level 2. Custom Type definitions can be applied to all sorts of RASON model problem types including optimization, simulation and stochastic models along with decision tables and custom functions. Note: Custom Type definitions are not supported in RASON data science models.

Data-source Binding

One key benefit to using a custom type definition is the ability to bind a single variable-structure, containing multiple named components, to the entire record.

In the past, the following data source declaration would require three different variables in order to bind to each value column.

```
"datasources": {
  "dsc_loan": {
    "type": "csv",
    "connection": "loan_data.txt",
    "selection": "loanID=?",
    "parameters": {
      "ID": {
        "binding": "get",
        "value": "L1"
      }
    },
    "indexCols": ["loanID"],
    "valueCols": ["principal", "rate", "termMonths"]
  }
}
```

However, once a custom type definition has been defined...

```
"typeDefs": {
  "tLoan": {
    "language": "FEEL",
    "components": {
      "principal": {"typeRef": "number", "allowedValues": [ ">0" ]},
      "rate": {"typeRef": "number", "allowedValues": [ "0..1" ]},
      "termMonths": {"typeRef": "number", "allowedValues": [ "0>" ]}
    }
  }
}
```

...then a new variable can be introduced with "type" set to the custom type definition (in this example "loan") and that variable can be bound to the data-source (in this example "dsc_loan") as shown in the code below.

```
"data": {
  "loan": {
    "type": "tLoan",
    "binding": "dsc_loan"
  }
}
```

The binding property feeds the components of the variable, loan, with the values in the data-source record. Later in formulas, the components may be referenced through the "." operator, for example:

```
"formulas": {
  "payment": "(loan.principal * loan.rate/12) / (1-(1+loan.rate/12)^(loan.termMonths))",
  "finalValue": []
}
```

Note: The variable "loan" of this custom type definition can be alternatively initialized inline or through the existing binding "get".

```
"data"
  "loan": {
    "type": "tLoan",
    "value": [100000, 0.0375, 360],
    "binding": "get"
  }
}
```

Custom Types in RASON

Custom Type is a major feature in the DMN/FEEL specification (Conformance Level 2) utilized heavily in the development of Decision Trees and Custom Functions. However, this feature may be used in RASON Decision services beyond these two applications.

There are two different structures for custom types: custom types with constraints on values and custom types with components.

- Custom types with constraints on values

In this structure, all members of tEmploymentStatus and tAge are of the same typeRef, either "string" for tEmploymentStatus or "number" for tAge.

Custom Type with Constraints Example

```
"typeDefs": {
  "tEmploymentStatus": {
    "type": "string",
    "allowedValues": ["UNEMPLOYED", "EMPLOYED", "SELF-EMPLOYED", "STUDENT"]
  },
  "tAge": {
    "language": "FEEL",
    "typeRef": "number",
    "allowedValues": ["[18..21]", ">65"]
  }
},
```

- Custom types with components

This custom type uses the components property to define a list of components for the custom type structure. Notice that this structure allows different types to be passed to each component in the type definition.

Custom Type with Components Example

```
"typeDefs": {
```

```

    "tLoan": {
      "language": "FEEL",
      "components": {
        "principal": {"typeRef": "number", "allowedValues": [ ">0" ]},
        "rate": {"typeRef": "number", "allowedValues": [ "0..1" ]},
        "termMonths": {"typeRef": "number", "allowedValues": [ "0>" ]}
      }
    }
  }
}

```

Custom Type Specifications

A custom type must be defined within the "typeDefs": {} section of the RASON model.

The components of a custom type definition are:

- "language": Select the syntax (Excel or FEEL) by using "language": "FEEL" or "language": "Excel". The supported type is determined by the language setting. If missing, the default is "Excel". Type definitions within the same RASON model can be different. In other words, two type definitions within the same RASON model using two different language settings may exist.
- "isCollection": Use the "isCollection" property to allow multiple records to be passed to the variable with the given "type". See the Advanced Features section below for more information on this property.
- "typeRef" or "type": Assigns a variable to a given type.
- "language": FEEL or Excel
- If "language": "FEEL", use the "typeRef" property.

```

"typeDefs": {
  "tEmploymentStatus": {
    "language": "FEEL",
    "typeRef": "string",
    "allowedValues": [ "UNEMPLOYED", "EMPLOYED", "SELF-EMPLOYED",
      "STUDENT" ]
  }
}

```

- If "language": "Excel", use the "type" property rather than "typeRef".

```

"typeDefs": {
  "tEmploymentStatus": {
    "language": "Excel",
    "type": "string",
    "allowedValues": [ "UNEMPLOYED", "EMPLOYED", "SELF-EMPLOYED",
      "STUDENT" ]
  }
}

```

Supported Types when Formula Language = Excel

Boolean: The entered words TRUE and FALSE are interpreted as Boolean reserved words, not strings.

Number: May be an integer or fraction.

String or Text: Any string

Support Types when Language = FEEL

Boolean: The entered words TRUE and FALSE are interpreted as Boolean reserved words, not strings.

Date: Any valid date, such as 05-05-1964

Duration: There are two formats for duration, one measuring periods in months and another measuring periods in seconds. For example, P1DT1H2M3S denotes:

- P for "period"
- 1D for 1 day
- T for "time"
- 1H for 1 hour
- 2M for 2 minutes and 3S for 3 seconds.

Note: Since the basic component types depend on the "language" (Excel or FEEL) used, it is important for users to note the assigned type values. For example, if "language": "Excel", "typeRef" may not be set to "duration" since this type reference is not supported for this language.

- "components": Use this property to list the members in the type definition. This example contains 3 components: principal, rate, and termMonths. Each of these components is of type "number".

```
"typeDefs": {
  "tLoan": {
    "language": "FEEL",
    "components": {
      "principal": {"typeRef": "number", "allowedValues": [ ">0" ]},
      "rate": {"typeRef": "number", "allowedValues": [ "0..1" ]},
      "termMonths": {"typeRef": "number", "allowedValues": [ "0>" ]}
    }
  }
},
```

- "allowedValues": Use this property to specify the exact values that a type definition can take on, for example, a value greater than 0. In this example,

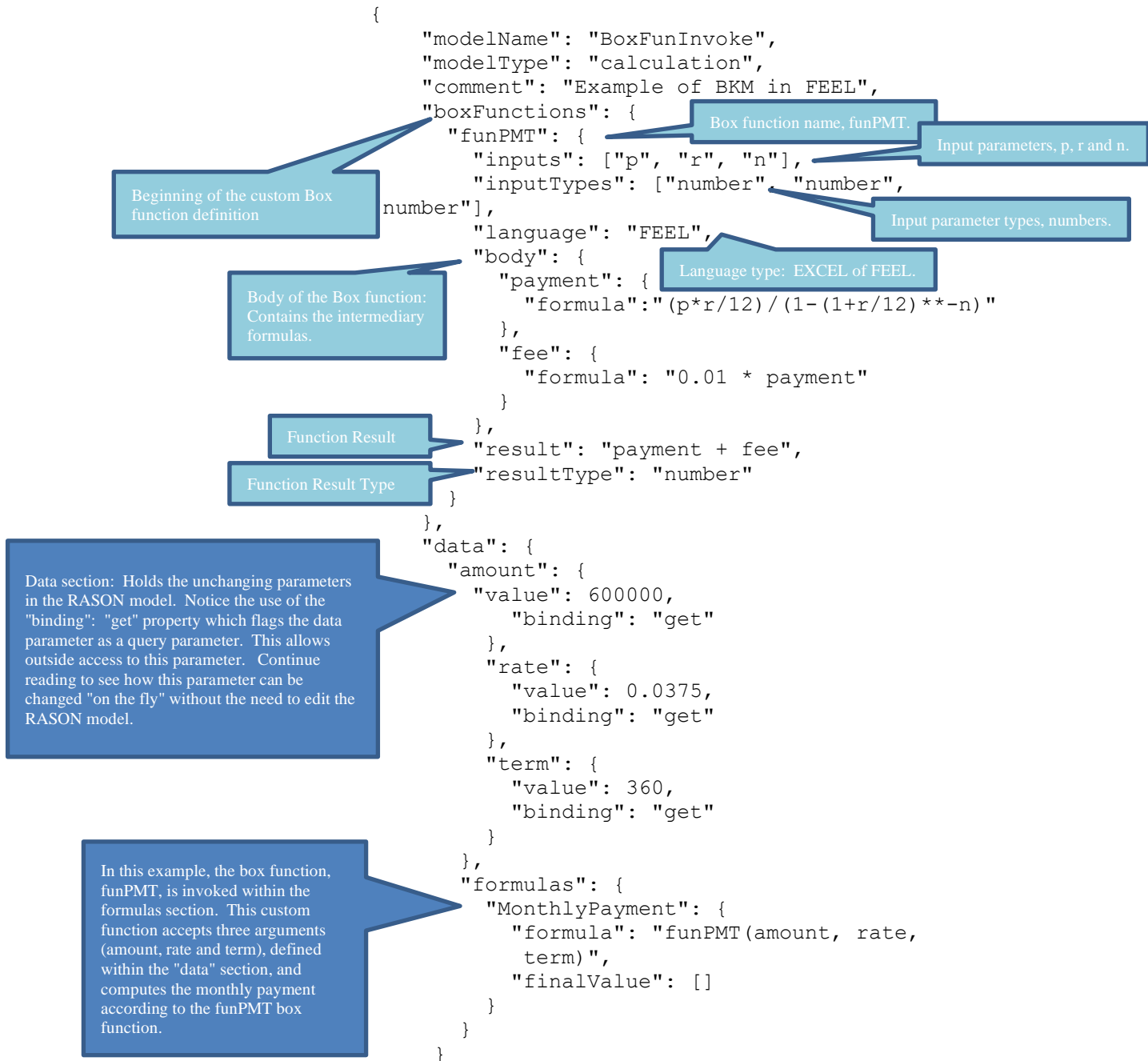
```
"typeDefs": {
  "tLoan": {
    "language": "FEEL",
    "components": {
      "principal": {"typeRef": "number", "allowedValues": [ ">0" ]},
      "rate": {"typeRef": "number", "allowedValues": [ "0..1" ]},
      "termMonths": {"typeRef": "number", "allowedValues": [ "0>" ]}
    }
  }
},
```

Using Type Definitions with Box Functions

RASON Decision Services supports custom, reusable functions in their RASON models. While custom functions can be defined using either a BOX function or a LAMBDA function, type definitions can only be used with Box functions. This section walks through an example that utilizes a custom definition in conjunction with a custom Box function. For more information on using a Box Function in a RASON model, see the chapter **Defining Custom Functions** in this guide.

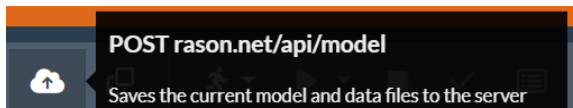
A Simple Box Function Example

This simple example defines the box function, funPMT, using FEEL syntax, and then calls the function to compute the monthly payment of an installment loan.

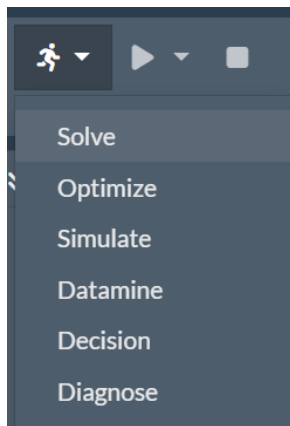


Posting and Solving the Model

POST the model to the RASON Server by clicking the POST rason.net/api/model icon.

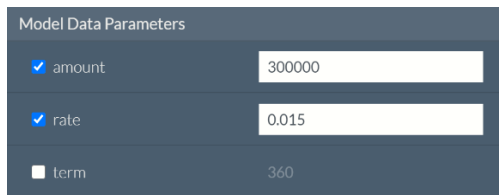


To solve the model "as is", click the Solve icon in the Ribbon and select Solve from the list of actions. The final value for the monthly payment will be calculated as \$2,806.48.



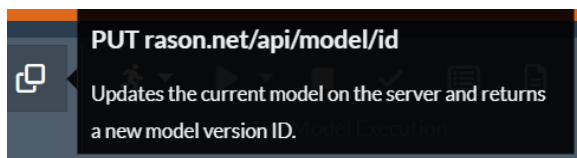
Click Select Query Parameters in the Properties pane, on the right, to open the Query Parameters dialog.

Under Model Data Parameters, select rate and amount, then change each of these to 0.015 and 300000, respectively.



Then click Save at the bottom of the dialog to change the amount parameter from 600,000 to 300,000 and the rate from 0.0375 to 0.015.

Update the model on the server with the two query parameters by clicking the PUT icon on the RASON ribbon.



Then click Solve – Decision to solve the model. (Solve – Solve could also be used to solve the model.) Notice that the monthly payment for these parameters, amount = 300,000, rate = 0.015 and term = 360, is \$1,045.71.

```
{
  "status": {
    "code": 0,
    "id": "2590+BoxFunInvoke+2021-03-23-23-09-07176359",
    "codeText": "Solver has completed the calculation."
  },
  "observations": {
    "MonthlyPayment": {
      "value": 1045.71
    }
  }
}
```

For more information on how to send a query parameter value programmatically through the \Solve endpoint, see the documentation for POST rason.net/api/model/id/solvetype found in the next chapter, Using the REST API. See the next chapter for an in-depth discussion on Box Functions in RASON Decision Services.

Box Function Example with Type Definitions

The example extends the previous example by introducing the "typeDefs" section to create a custom type definition.

As in the previous example, this example defines the box function, funPMT, using FEEL syntax, and then calls this function to compute the monthly payment of an installment loan. The difference between this example and the previous example, is that this example uses the "typeDefs" section to define the input parameters and their types. By introducing the "typeDefs" section into the RASON model, a user can specify the accepted values for the input parameters.

The "tLoan" type definition in this example is a flexible definition that allows only three components: principal, rate and termMonths. All three components must be numbers with specific allowed values. In this example, the input parameters are: principal = \$600,000, rate = 0.0375 and termMonths = 360.

Example with Type Definitions & Allowable Values; FEEL Syntax

In this example, the "allowedValues" property is used within the "typeDefs" section to not only limit the input parameters to numbers, but also to limit the values that each input parameter can be equal to.

- The principal input parameter must be greater than 0.
- The rate input parameter must be a value greater than or equal to 0 and less than or equal to 1.
- The termMonths input parameter must be a value greater than 0.

If a value outside of the allowed value bounds is passed to the function during the solving process, for example, if $r = 1.0375$, the error "Value mismatches its data type [loan must be of type tLoan]" will be returned.

```
{
  "modelName": "BoxFunInvokeTypeDef",
  "modelType": "calculation",
  "comment": "Box Function Example with Custom Type Definition",
  "typeDefs": {
    "tLoan": {
      "language": "FEEL"
      "components": {
        "principal": {
          "typeRef": "number",
          "allowedValues": [ ">0" ]
        },
        "rate": {
          "typeRef": "number",
          "allowedValues": [ "[0..1]" ]
        },
        "termMonths": {
          "typeRef": "number",
          "allowedValues": [ ">0" ]
        }
      }
    }
  },
  "boxFunctions": {
    "funPMT": {
      "inputs": [ "p", "r", "n" ],
      "inputTypes": [ "number", "number", "number" ],
```

The "typeDefs" section defines three components: principal, rate and termMonths. Notice that the input parameter types are not only defined as "numbers" but there are constraints on the values that the components may equal, i.e. principal > 0.

```

    "language": "FEEL",
    "resultType": "number",
    "body": {
      "payment": { "formula": "(p*r/12) / (1 - (1 + r/12)**-n)" },
      "fee": { "formula": "0.01 * payment" }
    },
    "result": "payment + fee"
  }
},
"data": {
  "loan": {
    "type": "tLoan",
    "value": [600000, 0.0375, 360],
    "binding": "get"
  }
},
"formulas": {
  "MonthlyPayment": {
    "formula": "funPMT(loan.principal, loan.rate, loan.termMonths)",
    "finalValue": []
  }
}
}
}

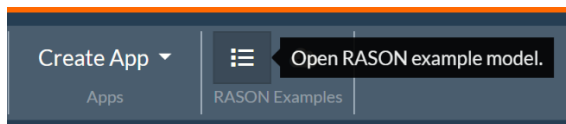
```

Example with Type Definition Passed as JSON Object

This next example further extends the Simple Example by passing two JSON objects, "loan" and "tLoan", to "inputs" and "inputTypes". The number of "inputs" and "inputTypes" must be equal. Passing "loan" to "inputs" passes the JSON object created within the "data" section. Passing "tLoan" for "inputTypes" passes the entire custom type definition as a JSON object.

Note that passing "inputType":["number", "number", "number"] would result in the error: Problem could not be loaded. The number of inputTypes must equal the number of inputs in box function definitions [inputTypes]."

To open this example, click the Open RASON example model icon,



then Decisions – Type Definition Examples – Custom Function with Type Definition Example. Email support@solver.com for access to any of the previous examples.

RASON requires that the number of input types must be equal to the number of inputs. In this example there is one input, "loan", a JSON object. In addition, there is one "inputType", "tLoan", also a JSON object.

```

...
"boxFunctions": {
  "funPMT": {
    "inputs": ["loan"],
    "inputTypes": ["tLoan"],
    "language": "FEEL",
    "resultType": "number",
    "body": {
      "payment": {
        "formula":
          "(loan.principal*loan.rate/12)
          /(1 - (1 + loan.rate/12)**-
            loan.termMonths)"
      },
      "fee": {
        "formula": "0.01 * payment"
      }
    }
  }
}

```

The JSON object "loan" is created within the "data" section and passed to the box function, funPMT, as the "inputs" argument.

Since there is now only one input, the "MonthlyPayment" formula has been simplified to simply "funPMT(loan)".

```

    },
    "result": "payment + fee"
  }
},
"data": {
  "loan": {
    "type": "tLoan",
    "value": [600000, 0.0375, 360],
    "binding": "get"
  }
},
"formulas": {
  "MonthlyPayment": {
    "formula": "funPMT(loan)",
    "finalValue": []
  }
}
}

```

...

Example with Type Definitions using Excel Syntax

To write this same example using Excel Syntax,

- Change the "typeRef" argument for the tLoan custom type definition components to simply "type". (The property "typeDefs" is not supported for "components" when Language is Excel.)

```

"typeDefs": {
  "tLoan": {
    "components": {
      "principal": {
        "type": "number",
        "allowedValues": [ ">0" ]
      },
      "rate": {
        "type": "number",
        "allowedValues": [ "[0..1]" ]
      },
      "termMonths": {
        "type": "number",
        "allowedValues": [ ">0" ]
      }
    }
  }
}

```

- Change "language" to "Excel" and the payment formula, within the funPMT box function body, to the following.

```

"boxFunctions": {
  "funPMT": {
    "inputs": [ "loan" ], "inputTypes": [ "tLoan" ],
    "language": "Excel", "resultType": "number",
    "body": {
      "payment": {
        "formula":
          "(loan.principal*loan.rate/12)
          /(1 - (1 + loan.rate/12))"
      }
    }
  }
}

```

```

        ^-loan.termMonths)"
    },
    "fee": {
        "formula": "0.01 * payment"
    }
},
"result": "payment + fee"
}

```

All remaining code will stay the same.

Using Type Definitions with Decision Tables

This section walks through an example of a decision table that includes a type definition.

A decision table contains a set of rules which specify actions to perform based on specific conditions. Decision tables are a good tool to use when there is a consistent number of rules, or conditions, to be evaluated followed by a specific set of actions to be performed once a rule, or condition, is met.

This decision table example determines the medical risk rating for a patient based on the patients age and medical history. The original formulation for the model is shown below. This example may be opened from www.RASON.com by clicking the Editor tab and then Rason Example Models – Decisions – Decision Tables – DT Unique example. For more information on Decision Tables, see the chapter, **Defining Decision Tables in RASON**.

```

{
  "modelName": "DTUniqueExample",
  "modelDescription": "'U' policy example",
  "modelType": "calculation",
  "data": {
    "comment": "use binding to feed dif. values",
    "age": {
      "value": 54
    },
    "medHistory": {
      "value": "good"
    }
  },
  "decisionTables": {
    "tblRisk": {
      "inputs": ["age", "medHistory"],
      "outputs": ["riskRating", "rule"],
      "rules": [
        [ ">60,<25", "good", "Medium", "r1"],
        [ ">60", "bad", "High", "r2"],
        [ "[25..60]", "-", "Medium", "r3"],
        [ "<25", "bad", "Medium", "r4"]
      ],
      "hitPolicy": "Unique",
      "default": [
        ["High", "r0"]
      ]
    }
  },
  "formulas": {
    "res": {

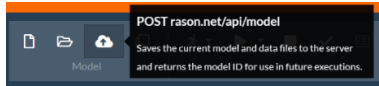
```

```

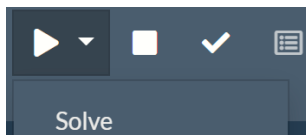
    "formula": "tblRisk(,,age,
medHistory)",
    "finalValue": []
  }
}
}

```

POST the model to the RASON Server by clicking the POST rason.net/api/model icon.



To solve the model with the inputs age = 54 and medHistory = "good", click the Solve icon in the Ribbon and select Solve from the list of actions.



With these inputs, only the 3rd rule is successful, and therefore "Medium" is returned for the patient risk.

Note: Recall that the finalValue for "res" is displayed in the result because the "finalValue": [] property is present.

Decision Table with Type Definition with Allowed Values

In the next RASON example model, the "typeDefs" section has been added (in red) in order to ensure the inputs to the tblRisk decision table are of type number (age) and string (medHistory).

In this example, age must be greater than 0 and medHistory can be either "good" or "bad". This is achieved by the use of the allowedValues property. This property allows users to specify the specific values that a component can take on. If an unsupported value is entered for age or medHistory, say "medium" for medHistory, the error *Problem could not be loaded. Value mismatches its data type [cust must be of type tCustomer]* will be returned.

To open this example from www.RASON.com, click the Open RASON Example model icon, then click Decisions – Custom Type Reference – Decision Table with Type Reference Example.

```

{
  "modelName":
  "DTUniqueExampleTypeDefAllowedValues",
  "modelDescription": "\"U\" policy example with Type
  Definition",
  "modelType": "calculation",
  "typeDefs": {
    "tCustomer": {
      "language": "FEEL",
      "components": {
        "age": {
          "typeRef": "number",
          "allowedValues": [ ">0" ]
        },
        "medHistory": {
          "typeRef": "string",
          "allowedValues": [ "bad", "good" ]
        }
      }
    }
  }
}

```

The tCustomer type definition defines two components, age and medHistory, and sets their type to number and string, respectively. In addition, the allowedValues property restricts the values that can be entered for each to >0 and specific strings: bad and good.

Within the "data" section, the "cust" JSON object is set to type "tCustomer" which includes two components, age and medHistory. Here, age is set to 54 and medHistory is set to "good".

The binding property for the "cust" JSON object allows access to tCustomer outside of the RASON


```

    },
    "data": {
      "comment": "use binding to feed dif. values",
      "cust": {
        "type": "tCustomer",
        "value": [54, "good"],
        "binding": "get"
      }
    },
    "decisionTables": {
      "tblRisk": {
        "inputs": ["age", "medHistory"],
        "outputs": ["riskRating", "rule"],
        "rules": [
          [">60,<25", "good", "Medium", "r1"],
          [">60", "bad", "High", "r2"],
          ["[25..60]", "-", "Medium", "r3"],
          ["<25", "bad", "Medium", "r4"]
        ],
        "hitPolicy": "Unique",
        "default": [
          ["High", "r0"]
        ]
      }
    },
    "formulas": {
      "res": {
        "formula": "tblRisk(, , cust.age, cust.medHistory)",
        "finalValue": []
      }
    }
  }
}

```

The tblRisk decision table accepts two inputs, age and medHistory, and returns a result based on the "rules".

Since "cust" is of type "tCustomer" and "tCustomer" has the properties of "age" and "medHistory", inputs to the decision table are easily referred to as cust.age and cust.medHistory.

With inputs of age = 54 and medHistory = good, only the 3rd rule is successful, and therefore "Medium" is returned for the patient risk.

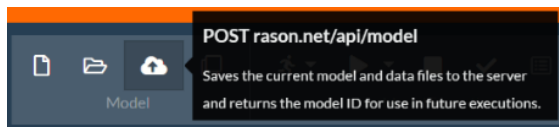
Notice that we cannot pass data of custom types as inputs to decision tables.

"formula": "tblRisk(, , cust.age, cust.medHistory)",

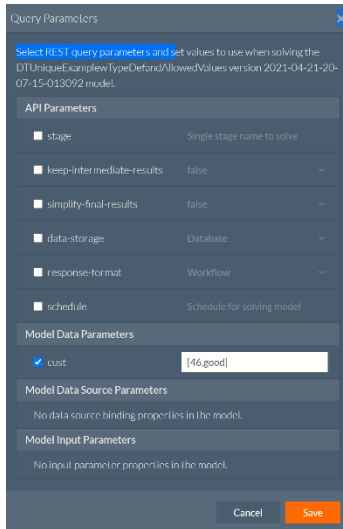
Cust.age and cust.medHistory are basic (non-custom) types and are therefore supported.

"formula": "tblRisk(, , cust)", is **NOT** supported.

POST the model to the RASON Server by clicking the POST rason.net/api/model icon.



Rather than using the hard coded values for age and medHistory, click Select Query Parameters on the Properties pane to open the Query Parameters dialog.



Query Parameters

Select REST query parameters and set values to use when solving the DT UniqueExampleTypeDefAndAllowedValues version 2021-04-21-20-07-15-013092 model.

API Parameters

- stage: Single stage name to solve.
- keep-intermediate-results: false
- simplify-final-results: false
- data-storage: Database
- response-format: Workflow
- schedule: Schedule for solving model

Model Data Parameters

- cust: [46,good]

Model Data Source Parameters

No data source binding properties in the model.

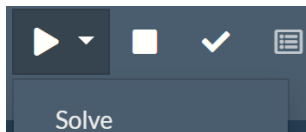
Model Input Parameters

No input parameter or properties in the model.

Cancel Save

Select "cust" under Model Data Parameters and enter a new age and medical history rating, such as "46" and "good". Then click Save to save the new data parameters and close the dialog.

Afterwards, click the POST `rason.net/api/model/id/solvetype` icon and select Solve from the menu to solve the RASON model using the new data parameters of age = 46 and medHistory = good.



RASON Decision Services returns a patient medical risk rating of "medium" via rule 3.

Table res_finalValue			
ID	riskRating	rule	
0	Medium	r3	

Note: Recall that the finalValue for "res" is displayed in the result because the "finalValue": [] property is present.

For information on passing data/query parameters via the RASON Solve endpoint, see the POST `rason.net/api/model/id/solvetype` endpoint description within the chapter, Using the Rest API.

Advanced Features: Type Definitions with Collections

If the "isCollection" property is set to True, multiple records may be passed to the variable of the given type, for example:

```
"typeDefs": {
  "language": "FEEL",
  "tStats": {
    "typeRef": "string",
    "isCollection": true,
```

```

    "allowedValues": ["UNEMPLOYED", "EMPLOYED", "SELF-EMPLOYED",
    "STUDENT"],
  },
  "tLoan": {
    "isCollection": true,
    "components": {
      "principal": {"type": "number", "allowedValues": [ ">0" ] },
      "rate": {"type": "number", "allowedValues": [ "[0..1]" ] },
      "termMonths": {"type": "number", "allowedValues": [ ">0" ] }
    }
  }
}

```

A variable can now be defined with "type" set to the collection definition. This variable can be bound to a data source containing multiple records. Recall the "dsc_loan" datasource from the beginning of this chapter.

```
"loan": {"type": "tLoan", "binding": "dsc_loan"}
```

The variable "loan" can be used within the "formulas" section to represent the entire collection (array formulas) or by extracting/filtering specific information.

In the following example, the formula will be computed for all records in the collection.

```

"payment": {
  "formula": "(loan.principal * loan.rate/12) / (1-(1
+ loan.rate/12)^-loan.termMonths)"
}

```

Example: Custom Type with a Collection

```

{
  "modelName": "CustomTypewCollection",
  "modelDescription": "Example of a component type
  definition with a binding collection",
  "typeDefs": {
    "tLoan": {
      "isCollection": true,
      "language": "Excel",
      "components": {
        "principal": {"type": "number", "allowedValues": [ ">0" ] },
        "rate": {"type": "number", "allowedValues": [ "[0..1]" ] },
        "termMonths": {"type": "number", "allowedValues": [ ">0" ] }
      }
    }
  },
  "data": {
    "loan": {
      "type": "tLoan",
      "binding": "get",
      "value": [
        [600000, 0.0375, 360],
        [100000, 0.0375, 360]
      ]
    }
  },
  "formulas": {
    "payment": {
      "formula": "(loan.principal * loan.rate/12) /
      (1 - (1 + loan.rate/12)^-loan.termMonths)",

```

Note that there are two records in the collection. However, any number of records may exist in the collection.

Since there are two records in the collection, this formula will be computed twice, once for each record. Since the finalValue property is present, the final value of the formula will be displayed in the results.

```

        "finalValue": []
    }
}

```

Example Results in JSON

```

{
  "status": {
    "id": "2590+CustomTypewCollection+2021-04-23-00-34-37-613311",
    "code": 0,
    "codeText": "Calculation executed.",
    "solveTimestamp": "2021-04-23-00-34-40-356413",
    "solveTime": 55
  },
  "results": {
    "payment.finalValue": []
  },
  "payment": {
    "finalValue": {
      "objectType": "dataFrame",
      "name": "payment",
      "order": "col",
      "colNames": ["Col1"],
      "colTypes": ["double"],
      "indexCols": null,
      "data": [
        [2778.693549432746, 463.1155915721243]
      ]
    }
  }
}

```

Since there were two records in the collection, two results have been computed.

Since the finalValue property was included in the definition for "payment", the final value of the decision table calculation is included in the results.

Computing Specific Results

To only compute the formula for a specific record, use the index operator. The example below only computes the payment formula for the 1st record.

```

"formulas": {
  "payment": {
    "formula": "(loan[1].principal *
    loan[1].rate/12)/(1-(1 + loan[1].rate/12)^-
    loan[1].termMonths)"
  }
}

```

Negative Indexing

Using loan[-1] (i.e. loan[-1].principal) refers to the last record in the collection. (Negative indexing refers to counting from the last to the first, otherwise known as "reverse order".) In FEEL, the size of any collection can be obtained by the special function "count(loan)", while in RASON, users may use "value": rows(loan) or value: "loan.rows" within a "data" definition.

Negative Indexing Example 1

...

```
"formulas": {
  "payment": {
    "feelFormula": "(loan[-1].principal * loan[-1].rate/12) / (1 - (1 + loan[-1].rate/12)**-loan[-1].termMonths)",
    "finalValue": []
  }
}
```

Negative Indexing Example 2

...

```
"data": {
  "loan": {
    "type": "tLoan",
    "binding": "get",
    "value": [
      [600000, 0.0375, 360],
      [100000, 0.0375, 360]
    ]
  }
},
"formulas": {
  "numrows": {"formula": "rows(loan)"},
  "payment": {
    "feelFormula": "(loan[numRows].principal * loan[numRows].rate/12) / (1 - (1 + loan[numRows].rate/12)**-loan[numRows].termMonths)",
    "finalValue": []
  }
}
```

Clause Selection

A clause may be used to filter or exclude records for a given criteria. RASON Decision Services supports numbered **indices** (positive and negative), **variables** with integer values plus the **and/or** joins for both FEEL and Excel syntax. If no data matches the clause, a NULL will be returned.

- The filter `loan[principal > 200000]` will filter records where "principal" is greater than \$200,000.

Clause Example 1

```
"formulas": {
  "payment": {
    "formula": "(loan[principal>200000].principal * loan[principal>200000].rate/12) / (1-(1 + loan[principal>200000].rate/12)^-loan[principal>200000].termMonths)"
  }
}
```

- AND/OR conditions may also be used to create a more advanced filter, such as: `loan[principal > 100000 and principal < 500000]`.

Clause Example 2

```
"formulas": {
```

```

"myLoan": {
  "formula": "loan[principal>=100000 and principal<=500000]"
},
"payment": {
  "feelFormula": "(myLoan.principal * myLoan.rate/12)
/ (1 - (1 + myloan.rate/12)**-myLoan.termMonths)",
  "finalValue": []
}
}

```

- It's also possible to exclude records from the computation. In the following example, "item" is used to exclude records marked as "STUDENT". Note that "item" is a special name for "types without components".

Clause Example 3

```

{
  "comment": "Example of a constraint type
Definition with a Collection and Exclusion
filter",
  "modelName":
  "ConstraintTypeDefwithCollectionFilter",
  "typeDefs": {
    "tEmploymentStatus": {
      "type": "string",
      "allowedValues":
      ["UNEMPLOYED", "EMPLOYED", "SELF-
EMPLOYED", "STUDENT"]
    }
  },
  "data": {
    "Employee1": {
      "type": "tEmploymentStatus",
      "value": [
        "EMPLOYED",
        "EMPLOYED",
        "SELF-EMPLOYED",
        "UNEMPLOYED",
        "UNEMPLOYED",
        "STUDENT",
        "STUDENT",
        "STUDENT"
      ],
      "binding": "get"
    }
  },
  "formulas": {
    "EmploymentStatement": {
      "formula": "You are " &
EmploymentStatus[item<> "STUDENT"]",
      "finalValue": []
    }
  }
}

```

Excel Syntax to exclude records marked as "student"

[item <> "STUDENT"]

FEEL Syntax to exclude records marked as "student"

```
[item != "STUDENT"
```

Defining Custom Functions

Introduction

Users of RASON Decision Services now have the option of defining and using custom, reusable functions in their RASON models. Custom functions can be defined using a custom BOX function or a LAMBDA function. This chapter contains a walk-through of several examples using both Box and Lambda functions.

Using the Lambda Function

Use the Lambda function to create a custom, reusable function within your RASON model. See the Analytic Solver User Guide for an explanation of how to define and use Microsoft's Lambda function in an Excel model. Continue reading to see an example of how to use this function in a RASON model.

Function Definition:

```
Lambda (Parameter1,  
Parameter2, ... ParameterN,  
Calculation)  
  
Lambda(x, y, x + y)
```

Lambda Function Defined

The Lambda function supported in RASON uses the definition of Microsoft Excel's newly implemented Lambda function. For a complete definition of this function, click [here](#).

Suppose a user wanted to create a function that computes the Total Profit of a company that produces TV, Stereos and Speakers. (Recall this is exactly what the Product Mix example model does in the Optimization chapter that occurs previously in this guide.)

=LAMBDA(TV,Stereos,Speakers,75*TV + 50*Stereos + 35* Speakers)

The LET function can be used in conjunction with the LAMBDA function to give:

=LAMBDA(TV,Stereos,Speakers,LET(TV,TV*75,Stereos,50*Stereos,Speakers,35*Speakers,TV+Stereos+Speakers))

The function takes three arguments named TV, Stereos and Speakers, binds the value of 75 * TV to the name TV, the value of 50*Stereos to Stereos, the value of 35*Speakers to Speakers and TV + Stereos + Speakers as the result.

Once the function is created within the boxFunctions section of the RASON model, the function can be called by that name thereby eliminating the need to enter this formula multiple times in the model.

Note: Although in Excel a LAMBDA can be an argument to another LAMBDA, this behavior is not supported in RASON Decision Services.

Optimization Example

To open this example click RASON Examples – Decisions – Custom Lambda Functions – Optimization Model with Lambda Function.

This example attempts to find a location for an airport that minimizes the distance between the proposed airport location and six cities.

This model can be written in standard form as:

MIN: MinVar

Subject To:

Henderson Coord: $\text{SQRT}((6 - \text{Xone})^2 + (2 - \text{Yone})^2) - \text{MinVar} \leq 0$

Winchester Coord: $\text{SQRT}((5 - \text{Xone})^2 + (5 - \text{Yone})^2) - \text{MinVar} \leq 0$

Sunrise Manor Coord: $\text{SQRT}((4 - \text{Xone})^2 + (8 - \text{Yone})^2) - \text{MinVar} \leq 0$

Paradise Coord: $\text{SQRT}((3 - \text{Xone})^2 + (4 - \text{Yone})^2) - \text{MinVar} \leq 0$

Enterprise Coord: $\text{SQRT}((3 - \text{Xone})^2 + (3 - \text{Yone})^2) - \text{MinVar} \leq 0$

Blue Diamond Coord: $\text{SQRT}((1 - \text{Xone})^2 + (2 - \text{Yone})^2) - \text{MinVar} \leq 0$

where MinVar, Xone and Ytwo are decision variables ≥ 0 .

In this example, the Lambda function, found within the boxFunctions section in the RASON model below, uses the distance formula to calculate the distance between the proposed airport coordinates (the decision variables Xone, Yone and MinVar) and the coordinates of each of the six cities: Henderson: (6, 2), Winchester (5, 5), Sunrise Manor (4,8), Paradise (3, 4), Enterprise (3, 3) and Blue Diamond (1, 2) . The third variable, MinVar which is minimized in the objective, "squishes" the distance formulas so that the distance between the proposed airport location and each city is as small as possible.

Notice that this example has been transformed from an Excel example using Analytic Solver's Create App feature. For more information on how to convert a model created in Excel using Analytic Solver into a RASON model with just one click, see the Analytic Solver User Guide.

```
{
  "comment": "This model has been generated by Psi from an Excel model in
the workbook BoxFunctionAirport.xlsx",
  "modelName": "LambdaFunOptimize",
  "modelType": "Optimization",
  "modelSettings": {
    "worksheets": ["Original_Formulation", "Box_Function",
    "Lambda_Function"],
    "activeSheet": "Lambda_Function"
  },
  "engineSettings": {
    "engine": "GRG Nonlinear",
    "scaling": -1
  },
  "boxFunctions": {
    "DistanceFormula": {
      "result":
        "LAMBDA(xone,xtwo,yone,ytwo,SQRT((xtwo-
xone)^2+(ytwo-yone)^2))"
    }
  },
  "variables": {
    "e13:f13": {
      "value": 1,
      "finalValue": []
    },
    "g20": {
      "value": 1,
      "finalValue": []
    }
  }
},
```

The Lambda function appears within the boxFunctions section of the RASON model. Notice the definition follows: LAMBDA (parameter1, parameter 2, parameter 3, parameter 4, calculation) .

The variables section of the RASON model holds the three variables: E13, F13 and G20.

```

"data": {
  "e14": {
    "value": 6
  },
  "f14": {
    "value": 2
  },
  "e15": {
    "value": 5
  },
  "f15": {
    "value": 5
  },
  "e16": {
    "value": 4
  },
  "f16": {
    "value": 8
  },
  "e17": {
    "value": 3
  },
  "f17": {
    "value": 4
  },
  "e18": {
    "value": 3
  },
  "f18": {
    "value": 3
  },
  "e19": {
    "value": 1
  },
  "f19": {
    "value": 2
  }
},

```

The data section holds the x and y coordinates of each city.

Henderson: E14 and F14 – (6, 2)
 Winchester: E15 and F15 – (5, 5)
 Sunrise Manor: E16 and F16 – (4, 8)
 Paradise: E17 and F17 – (3, 4)
 Enterprise: E18 and F18 – (3, 3)
 Blue Diamond: E19 and F19 – (1, 2)

```

"formulas": {
  "g14": {
    "formula":
      "DistanceFormula($E$13,E14,$F$13,F14) "
  },
  "g15": {
    "formula":
      "DistanceFormula($E$13,E15,$F$13,F15) "
  },
  "g16": {
    "formula":
      "DistanceFormula($E$13,E16,$F$13,F16) "
  },
  "g17": {
    "formula":
      "DistanceFormula($E$13,E17,$F$13,F17) "
  },
  "g18": {
    "formula":

```

The Lambda functions all appear within the formulas section of the model.

Note the Box function could also have been called within the constraints section by changing the code to:

```

"constraints" : {
  "G14_": {
    "formula":
      "DistanceFormula($E$13,E14,$F$13,F14) -
      G20",
    "upper": 0
  }
}

```

If the Box function is called within the constraints section, the formulas section is not needed.

```

        "DistanceFormula($E$13,E18,$F$13,F18)"
    },
    "g19": {
        "formula":
        "DistanceFormula($E$13,E19,$F$13,F19)"
    }
},
"constraints": {
    "g14_": {
        "formula": "g14 - g20",
        "upper": 0
    },
    "g15_": {
        "formula": "g15 - g20",
        "upper": 0
    },
    "g16_": {
        "formula": "g16 - g20",
        "upper": 0
    },
    "g17_": {
        "formula": "g17 - g20",
        "upper": 0
    },
    "g18_": {
        "formula": "g18 - g20",
        "upper": 0
    },
    "g19_": {
        "formula": "g19 - g20",
        "upper": 0
    }
},
"objective": {
    "g20_": {
        "formula": "g20",
        "type": "min",
        "finalValue": []
    }
}
}

```

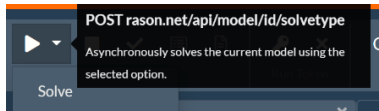
The objective minimizes the decision variable G20 which has an "squishing" effect on the Lambda functions to find an airport location that minimizes the distance between the proposed location and all six cities.

Posting and Solving the Model

POST the model to the RASON Server by clicking the POST rason.net/api/model icon.



To solve the model using the OData endpoint /Solve, click the Solve icon in the Ribbon and select Solve from the list of actions. For more information on the POST rason.net/api/model/id/Solve endpoint, see the next chapter Using the REST API.



After the model is loaded, parsed and solved, the OData Viewer appears. Click the down arrow next to Table to view the results. The final coordinate values are: 3.5 and 4.5.

Table e13_f13	
ID	finalValue
0	3.499999428887126
1	4.500000285328616

Simulation Example Using a Lambda Function

The simulation model below uses a custom Lambda function to calculate the predicted daily price of a stock for five consecutive days. A similar example model can be opened under RASON Examples – Decisions – Custom Lambda Functions – Simulation Model with Lambda Function.

The LAMBDA function accepts five 5 parameters: Volatility, NumofDays, NormDist, Appreciation and Previous. Then uses those 5 parameters to calculate the following formula:

$$\text{Previous} * (\text{Volatility} * \text{SQRT}(1/\text{NumofDays}) * \text{NormDist} + \text{EXP}((\text{Appreciation} - \text{Volatility}^2 * 0.5) * (1/\text{NumofDays}))))$$

```
{
  "comment": "Simulation Model with Lambda
    Function",
  "modelName": "LambdaFunSimulate",
  "modelType": "simulation",
  "boxFunctions": {
    "PredDailyPrice": {
      "result":
        "LAMBDA (Volatility,NumofDays, NormDist,Appreciation,Previous,Previous*(Vola
          tility*SQRT(1/NumofDays)*NormDist + EXP((Appreciation-
            Volatility^2*0.5)*(1/NumofDays))))"
    }
  },
  "uncertainVariables": {
    "NormDist1": {
      "formula": "PsiNormal(0,1)"
    },
    "NormDist2": {
      "formula": "PsiNormal(0,1)"
    },
    "NormDist3": {
      "formula": "PsiNormal(0,1)"
    },
    "NormDist4": {
      "formula": "PsiNormal(0,1)"
    },
    "NormDist5": {
      "formula": "PsiNormal(0,1)"
    }
  }
},
```

The Lambda function appears within the boxFunctions section of the RASON model. Notice the definition follows:
LAMBDA (parameter1, parameter 2, parameter 3, parameter 4, parameter 5 calculation) .

The uncertainVariables section contains the uncertain variables.

```

"data": {
  "FirstClose": {
    "value": 20.33
  },
  "Volatility": {
    "value": 0.09486,
    "comment": "volatility"
  },
  "NumofDays": {
    "value": 100,
    "comment": "numofdays"
  },
  "Appreciation": {
    "value": 0.0234992098,
    "comment": "appreciation"
  }
},
"uncertainFunctions": {
  "DayOne": {
    "formula":
"PredDailyPrice (Volatility,NumofDays, NormDist1,Appreciation,FirstClose) ",
    "mean": [],
    "trials": []
  },
  "DayTwo": {
    "formula":
"PredDailyPrice (Volatility,NumofDays, NormDist2,Appreciation,DayOne) ",
    "mean": [],
    "trials": []
  },
  "DayThree": {
    "formula":
"PredDailyPrice (Volatility,NumofDays, NormDist3,Appreciation,DayTwo) ",
    "mean": [],
    "trials": []
  },
  "DayFour": {
    "formula":
"PredDailyPrice (Volatility,NumofDays, NormDist4,Appreciation,DayThree) ",
    "mean": [],
    "trials": []
  },
  "DayFive": {
    "formula":
"PredDailyPrice (Volatility,NumofDays, NormDist5,Appreciation,DayFour) ",
    "mean": [],
    "trials": []
  }
}
}

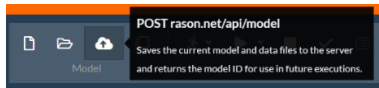
```

The data section contains the unchanging (constant) parameters used in the Lambda function.

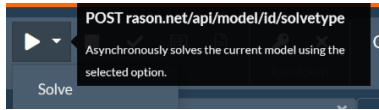
The Lambda functions are called within the uncertainFunctions section. In return, all trials and the expected value of the function will be displayed.

Posting and Solving the Model

POST the model to the RASON Server by clicking the POST rason.net/api/model icon.



Solve the model using the OData endpoint /Solve.



As soon as the model is loaded, parsed and solved, the results are immediately displayed in the Output window.

Table DayOne_statistics		
ID	mean	
0	20.33386818422745	

Table DayOne_trials		
ID	Trial	DayOne
0	1	19.982162566868467
1	2	20.628033116963696
2	3	20.44309385098142
3	4	20.104518609083545
4	5	20.349967129242575
5	6	20.351316810114685
6	7	20.591441724579536
7	8	20.221387388011582
8	9	20.58939402077483
9	10	20.326251968019278
10	11	20.43737511186305
11	12	19.88813461983043
12	13	20.154727606074595

Click Show JSON results in the upper right hand corner to view the same results in JSON.

Elements of a Box Function

A Box Function creates a custom, reusable function which can be used in an Analytic Solver Excel model or within an RASON model. See the Analytic Solver User Guide for an explanation of how to define and use a custom Box function in an Excel model. Continue reading to see an example of how to create a custom Box function within a RASON model.

Below is an example of a Box Function implemented in an Excel workbook. Notice that the box function contains seven components: Function Name (required), Formula Language (required), Input Parameters (required), Result Type (optional), Input Parameter Types (optional), Function Body (optional) and Function Result (required).

Function Name	Distance					Input Parameters
Formula Language	EXCEL	Xone	Xtwo	Yone	Ytwo	Input parameters types
Result Type	number	number	number	number	number	
	Formula1	(Xtwo-Xone)^2				Function Body
	Formula2	(Ytwo-Yone)^2				
	SQRT(Formula1+Formula2)					Function Result

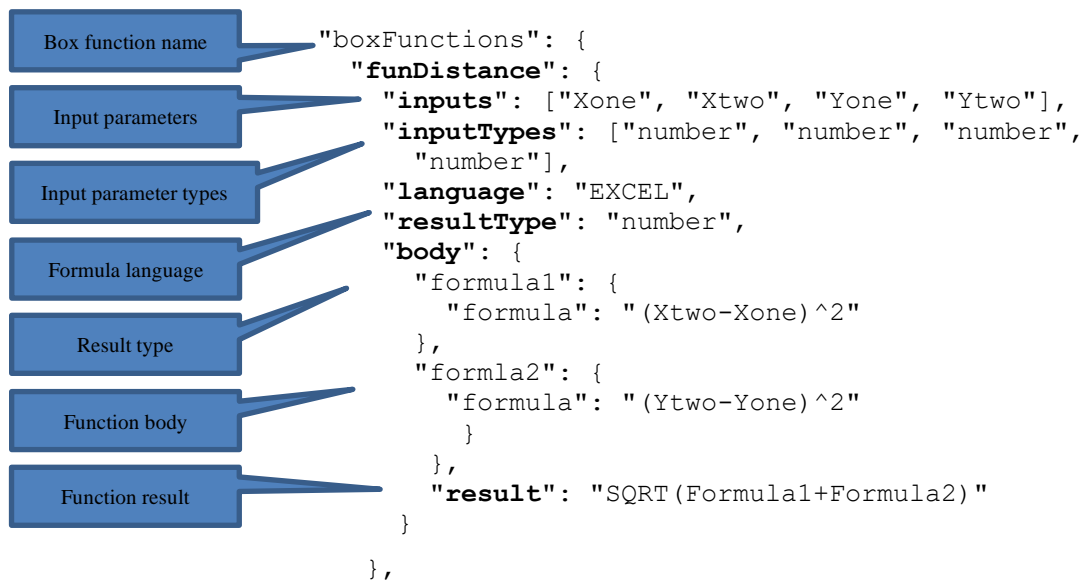
A custom function is created in RASON using the newly introduced `boxFunction` RASON section, as shown in the example code below, however additional sections such as `data`, `formulas`, or `dataSource` will also be called into play to pass the calculation parameters, get results and import data respectively.

In this example, the custom function, named `Distance`, accepts 4 numbers as input parameters: `Xone`, `Xtwo`, `Yone` and `Ytwo`. The result formula will calculate as:

$\text{SQRT}((\text{Formula1})^2 + (\text{Formula2})^2)$ where

- $\text{Formula1} = (\text{Xtwo} - \text{Xone})^2$
- $\text{Formula2} = (\text{Ytwo} - \text{Yone})^2$.

This is the same Box Function within a RASON model. Note the use of the prefix "fun" in front of the name, `Distance`. Using the prefix "fun" for Box functions in RASON is considered best practice and should be employed in order to prohibit naming conflicts between custom named functions and internal RASON functions. Note: Box functions converted from Excel will automatically be given the "fun" prefix.



The table below explains each component of the custom function.

Box Function Component	Rason Model Component	Description
Function name		(Required) The name of the custom function.
Formula language	language	(Required) Specifies the formula language specified. Supported options are FEEL or EXCEL. If FEEL is entered as the formula language, FEEL syntax is expected. (See top screenshot.)
Returned value type	resultType	(Required if Formula Language = FEEL, otherwise optional.) The returned value type is specified below the formula language. Custom functions return only one output. Supported types are: <u>Formula Language = Excel</u> Array: Any Excel cell reference, i.e. A1:C1.

		<p>Note: This can be used for a Box function that, say, computes the SUMPRODUCT(A1:A3, B1:B3) where A1:A3 is a range for the first input parameter and B1:B3 is a range for the second input parameter.</p> <p>Boolean: The entered words TRUE and FALSE are interpreted as Boolean reserved words, not strings.</p> <p>Empty: Select "empty" if Formula Language = EXCEL and no Data Type is being specified.</p> <p>Error: Any Excel error such as #N/A, #Number, etc.</p> <p>Number: May be an integer or fraction.</p> <p>String or Text: Any string</p> <p><u>Language = FEEL</u></p> <p>Boolean: The entered words TRUE and FALSE are interpreted as Boolean reserved words, not strings.</p> <p>Date: Any valid date, such as 05-05-1964</p> <p>Duration: There are two formats for duration, one measuring periods in months and another measuring periods in seconds. For example, P1DT1H2M3S denotes:</p> <p>P for "period"</p> <p>1D for 1 day</p> <p>T for "time"</p> <p>1H for 1 hour</p> <p>2M for 2 minutes and</p> <p>3S for 3 seconds.</p>
Input parameters	inputs	(Required) Specifies the input parameters. Input Parameters may be changing (i.e. decision variables, recourse variable, uncertain variables) or unchanging (i.e constant values).
Parameter types	inputTypes	<p>(Required if Formula Language = FEEL, otherwise optional.) The returned value type is specified below the formula language. Custom functions return only one output.</p> <p>Supported types are:</p> <p><u>Formula Language = Excel</u></p> <p>Array: Any Excel cell reference, i.e. A1:C1.</p> <p>See Note above for Returned Value Type.</p> <p>Boolean: The entered words TRUE and FALSE are interpreted as Boolean reserved words, not strings.</p>

		<p>Empty: Select "empty" if Formula Language = EXCEL and no Data Type is being specified.</p> <p>Error: Any Excel error such as #N/A, #Number, etc.</p> <p>Number: May be an integer or fraction.</p> <p>String or Text: Any string</p> <p><u>Language = FEEL</u></p> <p>Boolean: The entered words TRUE and FALSE are interpreted as Boolean reserved words, not strings.</p> <p>Date: Any valid date, such as 05-05-1964</p> <p>Duration: There are two formats for duration, one measuring periods in months and another measuring periods in seconds. For example, P1DT1H2M3S denotes:</p> <p>P for "period"</p> <p>1D for 1 day</p> <p>T for "time"</p> <p>1H for 1 hour</p> <p>2M for 2 minutes and</p> <p>3S for 3 seconds.</p>
Function body	body	<p>(Optional) Contains the intermediary formulas used in the Function result.</p> <p>All formulas are executed in the order they are entered.</p>
Function result	result	(Required) The formula for the function.

Invoking the Function in the RASON Model

Once the custom Box has been defined within the boxFunctions section, this function can be called *anywhere* within the RASON model. In this example, the Box function, Distance, is called within the "formulas" section. Note that in this example, Xvar and Yvar are the decision variables in the optimization model.

```
"formulas": {
  "g14": {
    "formula": "funDistance(6,Xvar,2,Yvar)"
  },
  "g15": {
    "formula": "funDistance(5,Xvar,5,Yvar)"
  },
  "g16": {
    "formula": "funDistance(4, Xvar,8,Yvar)"
  },
  "g17": {
```

```

    "formula": "funDistance(3, Xvar,4,Yvar)"
  },
  "g18": {
    "formula": "funDistance(3, Xvar,3,Yvar)"
  },
  "g19": {
    "formula": "funDistance(1, Xvar,2,Yvar)"
  },
},

```

Full RASON Optimization Model

The full RASON model is found below. This model has been automatically converted using Analytic Solver in Excel. For more information on converting an Analytic Solver Excel model to a RASON model, see the Deploying Your Model chapter within the Analytic Solver User Guide.

Name of RASON model.

Model Settings: There are three worksheets in the workbook; this model appears on the Box Function worksheet.

Custom Box Function: See component descriptions above.

Decision Variables: For more information on RASON optimization models, see the Optimization chapter that appears previously in this guide.

```

{
  "comment": "This model has been generated by Psi
from an Excel model in the workbook
BoxFunctionAirport.xlsx",
  "modelName": "BoxFunctionAirport",
  "modelDescription": "",
  "modelSettings": {
    "worksheets": ["Original_Formulation",
"Box_Function",
"Lambda_Function"],
    "activeSheet": "Box_Function"
  },
  "engineSettings": {
    "engine": "GRG Nonlinear",
    "scaling": -1
  },
  "boxFunctions": {
    "funDistance": {
      "inputs": ["Xone", "Xtwo", "Yone", "Ytwo"],
      "inputTypes": ["number", "number", "number",
"number"],
      "language": "EXCEL",
      "resultType": "number",
      "body": {
        "formula1": {
          "formula": "(Xtwo-Xone)^2"
        },
        "formula2": {
          "formula": "(Ytwo-Yone)^2"
        }
      },
      "result": "SQRT(Formula1+Formula2)"
    }
  },
  "variables": {
    "e13:f13": {
      "value": [
        [3.4999999752995, 4.5000000123503]
      ],
      "finalValue": []
    }
  },
},

```

Data: Where unchanging model parameters appear in a RASON model.

```

    "g20": {
      "value": 3.53553,
      "finalValue": []
    },
  },
  "data": {
    "e14": {
      "value": 6
    },
    "f14": {
      "value": 2
    },
    "e15": {
      "value": 5
    },
    "f15": {
      "value": 5
    },
    "e16": {
      "value": 4
    },
    "f16": {
      "value": 8
    },
    "e17": {
      "value": 3
    },
    "f17": {
      "value": 4
    },
    "e18": {
      "value": 3
    },
    "f18": {
      "value": 3
    },
    "e19": {
      "value": 1
    },
    "f19": {
      "value": 2
    }
  },

```

Formulas: Where intermediary formulas are computed in a RASON model.

```

  },
  "formulas": {
    "g14": {
      "formula": "funDistance(E13,E14,F13,F14)"
    },
    "g15": {
      "formula":
        "funDistance($E$13,E15,$F$13,F15)"
    },
    "g16": {
      "formula":
        "funDistance($E$13,E16,$F$13,F16)"
    },
    "g17": {
      "formula":

```

Constraints: Where the constraints are defined in an optimization model.

Note that in this example, the Box function, funPMT, is called within the "formulas" section. This function could have also been called within the "constraints". For example, the constraint for G14 could also have been written as:

```
...
"constraints": {
  "g14_": {
    "formula":
      "funDistance(E13,E14,F13,F14) - g20",
    "upper": 0
  },
  ...
}
```

Objective: Where the objective is defined in an optimization model.

```
      "funDistance($E$13,E17,$F$13,F17)"
    },
    "g18": {
      "formula":
        "funDistance($E$13,E18,$F$13,F18)"
    },
    "g19": {
      "formula":
        "funDistance($E$13,E19,$F$13,F19)"
    }
  },
  "constraints": {
    "g14_": {
      "formula": "g14 - g20",
      "upper": 0
    },
    "g15_": {
      "formula": "g15 - g20",
      "upper": 0
    },
    "g16_": {
      "formula": "g16 - g20",
      "upper": 0
    },
    "g17_": {
      "formula": "g17 - g20",
      "upper": 0
    },
    "g18_": {
      "formula": "g18 - g20",
      "upper": 0
    },
    "g19_": {
      "formula": "g19 - g20",
      "upper": 0
    }
  },
  "objective": {
    "g20_": {
      "formula": "g20",
      "type": "min",
      "finalValue": []
    }
  }
}
```

Posting and Solving the Model

POST the model to the RASON Server by clicking the POST rason.net/api/model icon.



To solve the model "as is", click the Solve icon in the Ribbon and select Solve from the list of actions. The final coordinate values are: 3.5 and 4.5.

Table e13_f13		
ID	finalValue	
0	3.499999428887126	
1	4.500000285328616	

See the Simulation example below to discover how to use a query parameter to supply inputs to a box function.

Simulation Example Using a Box Function

The simulation model below uses a custom Box function to calculate the predicted daily price of a stock for five consecutive days. A similar example model can be opened under RASON Examples – Decisions – Custom Box Functions – Simulation Model with Box Function.



The "data" section holds the unchanging or constant parameters used in the model. Note the use of the binding property. This property allows these parameters to be accessed outside of the RASON model.

The Box function, funDailyPrice, is called within the "formulas" section of the RASON model.

The uncertainFunctions section calculates the expected value for each DailyPrice formula above.

Note the Box function could also have been called within the uncertainFunction section by changing the code to:

```
"ExpDailyPrice1": {
  "formula":

"funDailyPrice (FirstClose, NormDist
1)",
  "trials": [],
  "mean": []
}
```

```
},
"NormDist5": {
  "formula": "PsiNormal(0,1)"
},
},
"data": {
  "Volatility": {
    "value": 0.09486,
    "binding": "get",
    "comment": "vol"
  },
  "NumDays": {
    "value": 100,
    "binding": "get",
    "comment": "numdays"
  },
  "AppRate": {
    "value": 0.0234992098,
    "binding": "get",
    "comment": "apprate"
  },
  "FirstClose": {
    "value": 20.33,
    "binding": "get",
    "comment": "firstclose"
  }
},
"formulas": {
  "DailyPrice1": {
    "formula":
    "funDailyPrice (FirstClose, NormDist1) "
  },
  "DailyPrice2": {
    "formula":
    "funDailyPrice (DailyPrice1, NormDist2) "
  },
  "DailyPrice3": {
    "formula":
    "funDailyPrice (DailyPrice2, NormDist3) "
  },
  "DailyPrice4": {
    "formula":
    "funDailyPrice (DailyPrice3, NormDist4) "
  },
  "DailyPrice5": {
    "formula":
    "funDailyPrice (DailyPrice4, NormDist5) "
  }
},
"uncertainFunctions": {
  "ExpDailyPrice1": {
    "formula": "DailyPrice1",
    "trials": [],
    "mean": []
  },
  "ExpDailyPrice2": {
```

```

        "formula": "DailyPrice2",
        "trials": [],
        "mean": []
    },
    "ExpDailyPrice3": {
        "formula": "DailyPrice3",
        "trials": [],
        "mean": []
    },
    "ExpDailyPrice4": {
        "formula": "DailyPrice4",
        "trials": [],
        "mean": []
    },
    "ExpDailyPrice5": {
        "formula": "DailyPrice5",
        "trials": [],
        "mean": []
    }
}
}
}

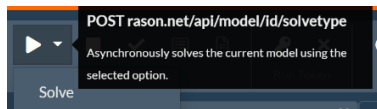
```

Posting and Solving the Model

POST the model to the RASON Server by clicking the POST `rason.net/api/model` icon.



To solve the model "as is", click the Solve icon in the Ribbon and select Solve from the list of actions.

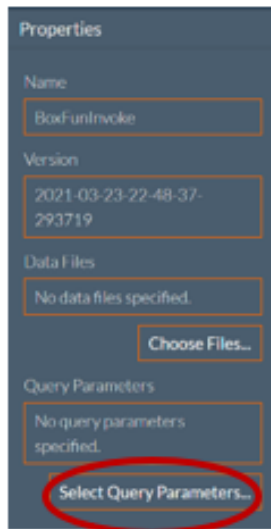


Click the down arrow next to Table to display the expected values of the predicted daily rates as found in the OData display.

Table ExpDailyPrice1_statistics ▾		
ID	mean	
0	20.334846025555592	

Table ExpDailyPrice1_trials ▾		
ID	Trial	ExpDailyPrice1
0	1	20.517207003591356
1	2	20.283302219383923
2	3	20.183416307182203
3	4	20.09631211985559
4	5	20.233236919248967
5	6	20.366653552781052
6	7	20.196838823084423
7	8	20.4472609829898
8	9	20.161814056889785
9	10	20.56272689872328
10	11	20.33360396012343

Go back to the original model but this time click Select Query Parameters in the Properties pane, on the right, to open the Query Parameters dialog.



Under Model Data Parameters, select all four parameters, Volatility, NumDays, AppRate and FirstClose, then change each of these values to the values shown in the screenshot below.



Then click Save at the bottom of the dialog to change the Volatility parameter to 0.10, NumDays to 120, AppRate to 0.03 and FirstClose to 15.67.

Update the model on the server with the four query parameters by clicking the PUT icon on the RASON ribbon.



Then click Solve – Solve to solve the model and view the results.

Table ExpDailyPrice2_trials		
ID	Trial	ExpDailyPrice2
0	1	15.520794728932701
1	2	15.750412132995763
2	3	15.458056144050795
3	4	15.767006017848063
4	5	15.521481302339147
5	6	15.458056144050795

Table ExpDailyPrice2_statistics ▾	
ID	mean
0	15.677849993953915

For more information on how to send a query parameter value programmatically through the \Solve endpoint, see the documentation for POST `rason.net/api/model/id/solvetype` found in the next chapter, Using the REST API.

Creating independent DMN/Feel models

In the latest version of RASON Decision Services, DMN/Feel functionality no longer requires Excel formulas when representing a DMN decision model. The entire model may now be represented using only FEEL formulas, which are referred to as **literal expressions**. Such models are entirely independent of Excel syntax. These models are referred to as **pure DMN models**. Notice that pure DMN models can only be decision/calculation models. Currently, Rason Decision Services does not support optimization, simulation or data science models as pure DMN models.

The main consequence of avoiding Excel formulas is to preserve the authentic DMN/Feel types in formula assignments.

For example,

```
dt: { feelFormula: "date('05-05-2021')"} }
```

preserves the specific Feel type 'feel date' in the variable **dt**, so we may use in a later feelFormula: **"dt.day"**.

Box functions, which can be similarly formatted, are reusable functions with input arguments and usually a single output. The result of a box function could be a scalar or array value. DMN CL3 suggests formatting the result of a box function as a custom component type where components match the formula variables in the body and ignore the result formula, which can be set to a dummy value, 0 or "".

```
{
  "comment": "Example of BKM in FEEL",
  "typeDefs": {
    "tLoan": {
      "language": "FEEL",
      "components": ['principal', 'rate', 'termMonths'],
      "types": ['number', 'number', 'number']
    },
    "tPayment": {
      "language": "FEEL",
      "components": ['payment', 'fee', 'total'],
      "types": ['number', 'number', 'number']
    }
  },
  "boxFunctions": {
```

```

"funPMT": {
  "inputs": ['p', 'r', 'n'],
  "inputTypes": ['number', 'number', 'number'],
  "language": "FEEL",
  "resultType": "tPayment",
  "body": {
    "payment": { "formula": "(p*r/12) / (1 - (1 + r/12)**-n)" },
    "fee": { "formula": "0.01 * payment" },
    "total": { "formula": "payment + fee" }
  },
  "result": "0"
},
"data": {
  "loan": {
    "type": 'tLoan',
    "value": [600000, 0.0375, 360],
    "binding": 'get'
  },
  "formulas": {
    "monthlyPayment": {
      "feelFormula": "funPMT(loan.principal, loan.rate,
        loan.termMonths)",
      "finalValue": []
    },
    "total": {
      "feelFormula": "monthlyPayment.total",
      "finalValue": []
    }
  }
}

```

The **data** variable, **loan**, is of type **tLoan**. The result of the box function will be of type **tPayment**. Notice that the components of **tPayment** (payment, fee, total) must exactly match the names of the formulas in the body of the funPMT Box Function; order is not important. Since the result of the box function is formatted as **tPayment**, this type is extended to the **monthlyPayment** variable.

The typed variable allows the desired component to be extracted like so in later formulas:

feelFormula: "monthlyPayment.total",

Without the typed result feature, the decision variable would have had to be called multiple times for different outputs and separate box functions would have been required for each component result.

See the chapter *Defining Custom Functions* within the [*RASON User Guide*](#) for more information on Box Functions.

Defining Decision Tables in RASON

Introduction

Users of RASON have the ability to create and evaluate Decision Tables. A decision table contains a set of rules which specify actions to perform based on specific conditions. Decision tables are a good tool to use when there is a consistent number of rules, or conditions, to be evaluated followed by a specific set of actions to be performed once a rule, or condition, is met. For example, the simple decision table below returns an employee's pay based on the number of hours worked.

C+	hours	pay
	number	number
1	[1..40]	25*hours
2	>40	45*(hours – 40)

If the employee worked from 1 to 40 hours in a week, the pay is \$25 * hours. If any overtime is worked, the employee gets paid \$45 an hour for any hour above 40.

A RASON decision table has a name, input/output parameters and a body implementing the logic through a structured framework. In RASON, a decision table is created in the `decisionTable` section.

This same decision table written in the RASON modeling language is below.

```
{
  modelName: "DTIntro",
  modelType: "calculation",
  decisionTables: {
    "tblPayDay": {
      inputs: ["hours"],
      outputs: ["pay"],
      refTypes: ["number", "number"],
      hitPolicy: "C+",
      rules: [
        ["[1..40]", "25 * hours"],
```

```

    [">40", "45*(hours - 40)"]
  ]
}
},

```

Use the formulas section to calculate the decision table given the input parameter, hours = 45.

```

formulas: {
  "paycheck": {formula: "tblPayDay(, ,45)", comment: "calculate pay
per
employee", finalValue: []}
}
}

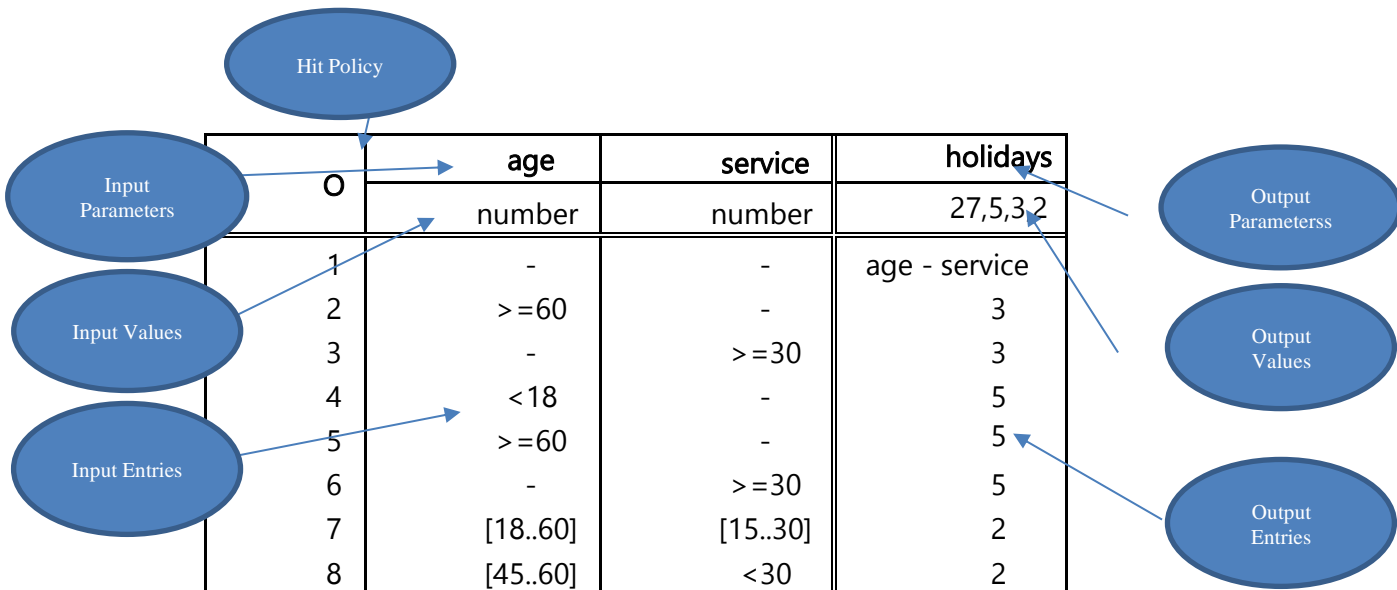
```

The modeling language inside of RASON's Decision Table functionality is S-FEEL extended to standard conversion functions in FEEL. For more information on Decision Tables, we invite you to reference the following: DMN Method and Style by Bruce Silver (Cody Cassidy Press, September 28, 2018 and DMN Cookbook by Bruce Silver & Edson Tirelli (Cody-Cassidy Press, April 4, 2018).

See the chapter RASON DMN/FEEL at Conformance Level 3 within the RASON Reference Guide for updated information using a Decision Table within RASON Decision Services with Conformance Level 3. A example with Conformance Level 3 can be found at the end of this chapter.

Decision Table Structure

Open the example, DT Output Order Example.json, on either the Editor tab on RASON.com or in the RASON IDE. The decision table contained within this example returns the number of holidays apportioned to an employee based on his/her age and number of years in service. This file defines the regions of the decision table as illustrated in the screenshot below. A visual representation of the decision table is shown below.



A decision table is created in RASON using the newly introduced `decisionTables` RASON section as shown in the example code below, however additional sections such as `data`, `formulas`, or `dataSource` will also be called into play to pass the calculation parameters, get results and import data respectively. According to the rules of the table, a 58 year old employee with 31 years of service is allotted 27 + 3 + 5 vacation days via rules 1, 3 and 6, respectively. However, since the Hit Policy specifies that the results must be returned priority order, the result collection is listed as 27, 5 and 3.

This example code creates the decision table shown above.

```
{
  modelName: "DTOutputOrderExample",
  modelDescription: "'O' output order policy example",
  modelType: "calculation",
  data: {
    age: { value: 58 },
    service: { value: 31 }
  },
  decisionTables: {
    tblHolidays: {
      hitPolicy: 'outputOrder',
      inputs: ['age', 'service'],
      outputs: ['holidays'],
      refTypes: ['number', 'number', ''],
      outputValues: [27, 5, 3, 2]
      rules: [
        ['- ', '- ', age-service],
        ['>=60', '- ', 3],
        ['- ', '>=30', 3],
        ['<18', '- ', 5],
        ['>=60', '- ', 5],
        ['- ', '>=30', 5],
        ['[18..60]', '[15..30]', 2],
        ['[45..60]', '<30', 2]
      ],
      default: [1]
    }
  },
  formulas: {
    result: { formula: "tblHolidays(,age, service)", finalValue: [] }
  }
}
```

modelName

RASON Decision Services supports named and unnamed models. A named model includes the `modelName: "name"` property in its text. "name" must be a string that can be URL encoded and must be unique among models within a user's account. It is strongly recommended that a model be named to allow for easy recognition and retrieval when multiple models, instances and versions exist in a user's account. See the chapter "Using the REST API" for more information on named and unnamed models.

modelDescription

This optional high level property offers users the ability to enter a user – defined string containing a description of the RASON model.

modelType

A new optional top-level property `"modelType"={"optimization", "simulation", "datamining", "calculation"}` has been introduced in RASON Decision Services for use with decision flows and standalone models. This property defines the model type as optimization, simulation, datamining or calculation within the RASON script. Since this model is a decision table model, the `modelType` is set to "calculation". For more information on this property, see the next chapter, Using the REST API.

data

The `data` section passes the parameters required to evaluate, or calculate, the decision table: `age` (58) and `service` (31). Note: Decision tables only accept scalar (constant) arguments as inputs. (No components of a decision table can be bound to an external database.)

decisionTables

In RASON, tables are defined as objects in the newly introduced section **decisionTables**: { }. Each table component is defined as a **component** with a scalar or array value assigned to it.

The steps required to create the decision table above are listed below.

Step 1: Enter a *unique* Name to define the table and create the decision table object. *In this example, the table name is specified as `tblHolidays`.*

Step 2: Enter a Hit Policy. `hitPolicy: 'outputOrder'`

Specifies how the table will be evaluated when multiple rules are applicable and hence, multiple output values are returned.

The Hit Policy value identifies the supported policies by a capital letter and an operator, when applicable. You may enter the 1st letter of the policy or the whole word. The currently supported Hit types and their meanings are:

Unique (U): A unique rule must be successful, or "hit", evaluating to a unique result. If multiple rules are "hit", an error will be returned.

Any (A): If rules overlap, but point to the same result, that unique result is returned.

Priority(P): If multiple rules are "hit" and multiple results collected, the result with the highest priority is returned. Priorities are defined by their order in the `outputValues` component (see below).

First (F): Only 1 result is returned for this policy. Once a rule is evaluated successfully, or a hit occurs, the search stops.

Rule Order (R) - If multiple rules are hit, the collection of results is returned according to the rule order, as specified in `outputValues`.

Output Order (O) – If multiple rules are hit, return the collection of results in the priority order as listed for `outputValues`.

Collect (C) – The same as **(R)**. However, we may make this policy more specific by adding an operator to it in order to allow aggregation.

Note: If aggregating a date, a scalar is returned. If using an operator, output must not be a string, but only a numerical value.

C+ - sum the matched output values

C< - return the min of matched output values

C> - return the max of matched output values

C# - return the number of matched output values

Step 3: Enter Input Parameters. `inputs: ['age', 'service']`

Two headers are contained in the table above, `age` and `service`. These are the input components, or the input data that will be passed to the decision table during calculation.

Step 4: (Optional) Enter Data Types using `refTypes`.

`refTypes: ['number', 'number', ''],`

The component `refType` describes the data type for each input. When defining `refTypes`, all input and output columns must be included. It's possible to enter empty strings or null for any input or output

column. This example uses the refTypes component to stipulate that the age and service input parameters will be of type "number" and passes an empty string to the output parameter. The output parameter's domain is defined using the outputValues component (see below.)

Data Types

Boolean: The entered words TRUE and FALSE are interpreted as Boolean reserved words, not strings.

Number: May be an integer or fraction.

Text: Any string

Date: Any valid date, such as 05-05-1964

Time: Any valid time

Duration: There are two formats for duration, one measuring periods in months and another measuring periods in seconds. For example, P1DT1H2M3S denotes:

- P for "period"
- 1D for 1 day
- T for "time"
- 1H for 1 hour
- 2M for 2 minutes and
- 3S for 3 seconds.

Step 5: Optional: Enter Input Values using inputValues.

The component inputValues describes the domain covered by all input entries in the decision table rules (see below). Each input value must relate to a given input parameter. Input Values may be a list of values separated by commas (i.e. 27, 5, 3, 2), a list of unary tests (i.e. <10, >=20, [18..20]), or a data type (i.e. boolean, number, text, data, time and duration).

Both refValues and inputValues/outputValues may exist within the same RASON model but inputValues will override the refValues component. Use refValues when stipulating the value type accepted by the column and input/outputValues when stipulating the domain of the column.

When testing a value against a list of values or unary tests, the OR operator is used. A list of values is evaluated as 27 OR 5 OR 3 OR 2. Likewise, the list of unary tests is evaluated as <10 OR >=20. It's possible to negate a list as well. For example, NOT(27, 5, 3, 2) would result in a selection of a record that does not includes 27 OR 5 OR 3 OR 2. Similarly, NOT(<10, >=20) would equate to neither <10 OR >=20 being selected.

All input entries in the relevant input column should cover the entered domain, otherwise, an error will be generated indicating that the table is not complete. If an input value does not exist, the completeness test is not performed.

Step 6: Enter Output Parameters outputs: ['holidays']

In this example, holidays is the output parameters. After calculation, a decision table returns a few selected or all output parameter values in the form of an array. If a decision table has a single output or a single output has been selected, the result will be a scalar value. These input parameters belong to the local scope of this table.

Step 7: (Optional) Enter the Output Values

outputValues: [27, 5, 3, 2]

An output value may be a list of values separated by commas (i.e. 27, 5, 3, 2) listing the priority of returned results. If a value appears in the table that does not match the output value, an error will be returned.

In this example, the list "27, 5,3,2" is entered beneath "holidays" and "text" is entered beneath "rule". The Output Values list (27, 5, 3, 2) specifies the priority when returning the results. In other words, the results are to be returned largest to smallest.

All output entries in the relevant output column should cover the entered domain, otherwise, an error will be generated indicating that the table is not complete. If an output value does not exist, the completeness test is not performed.

Step 8: Enter the Decision Table Rules (or Unary tests).

```
rules: [
    ['-', '-', age-service],
    ['>=60', '-', 3],
    ['-', '>=30', 3],
    ['<18', '-', 5],
    ['>=60', '-', 5],
    ['-', '>=30', 5],
    ['[18..60]', '[15..30]', 2],
    ['[45..60]', '<30', 2]
],
```

Rules consists of Input Entries and Output Entries. These entries consist of unary tests which return information (true or false) about the rule. Supported unary tests may have one of the following syntax forms: value(Boolean, number, text, date, time, duration), < value, > value, <= value, >= value, [value..value], (value..value), [value..value), (value..value), "-". Note: Currently, rules *may only* be entered as rows.

The first test examines whether the value being tested is equal to the value inside the parenthesis. For example, if the Unary test consists of the single value 'medium', the resulting test would ensure that the variable being tested was equal to 'medium'. The forms, [value..value], (value..value), [value..value) and (value..value), are interval tests. The [] operators denote a closed interval while the () operators denote an open interval. The last test, "-" returns TRUE against any value.

Step 9: (Optional) Define a default result.

```
default: [1]
```

There are two ways to return a default value for a decision table.

1. Simply use "-" for all unary tests.
2. Use the default component as shown in the example above.

In this example a default of "1" holiday is returned if no tests are successful.

Expressions

Variables and constants can be combined through operations called literal expressions. Literal expressions in S-FEEL are similar to formulas in Excel and in the RASON modeling language.

The following operators are supported in combination with decision table rules: addition (+), subtraction (-), multiplication (*), division (/) and exponentiation (**). Variables and constants can be combined using only these supported operators and parentheses. An example of an expression is: 2 * age – service where two variables, age and service and a constant, 2, are linked by two arithmetic operations (* and -). Note that an expression is a FEEL expression, NOT an Excel formula. In this example, the expression "age – service" appears in the first rule where "age" and "service" refer to 58 and 31, respectively, as defined in the data section. This expression does not refer to any appearances of "age" or "service" outside of the scope of this table. For more information on supported conversion expressions, please see Decision Tables in the RASON Reference Guide.

See the **Decision Table Containing Duration** section below for an illustration on using expressions within decision table rules. For a complete list of supported operators, see the Decisions Table section within the RASON Reference Guide.

formulas

Use the formulas section to obtain the final results from a decision table. The complete signature of the decision table function is:

```
"tblDecTable(,[string ret_output], [bool ret_header], variant
Input1, variant Input2, ..., variant InputN)"
```

where:

- The first argument [string ret_output] is an optional argument that, when passed, returns only the desired columns in the output.
- The second argument [bool ret_header] is an optional argument that, when True, returns the column headings in the output.
- The third and remaining arguments pass the input parameters to the decision table.

In this example, we are passing the parameters age and service (defined within the data section) to the decision table, tblHolidays. In return, we are asking for the finalValue, the result, from the table. Notice the two placeholders for ret_output and ret_header.

```
formulas: {
result: { formula: "tblHolidays(,age, service)", finalValue: [] }
      }
}
```

Output

As with optimization, simulation and data science models written in the RASON modeling language, RASON Decision Table models can be calculated by calling the RASON REST server by using the Editor page on the website www.RASON.com or by using the RASON REST API from within your own application. As with optimization, simulation and data science, no matter how you solve the model, the result will always be valid JSON. See the chapter, Solving RASON Models, for step by step instructions on how to solve a decision table in RASON.

If solving on the Editor page on www.RASON.com, click POST rason.net/api/model to POST the model and POST rason.net/api/model/{nameorid}/decision to calculate the table. According to the rules of the table, a 58 year old employee with 31 years of service is allotted 27 + 3 + 5 vacation days via rules 1, 3 and 6, respectively. However, since the Hit Policy specifies that the results must be returned in priority order, the result collection is listed as 27, 5 and 3.

```
{
  "status": {
    "code": 0,
    "id": "2590+DTOutputOrderExample+2020-02-25-19-16-35-647046",
    "codeText": "Solver has completed the calculation."
  },
  "observations": {
    "res": {
      "value": [27, 5, 3]
    }
  }
}
```

Optional Arguments

Two optional arguments may be passed to a decision table: `output` and `ret_header`.

To return the result for a given output only, pass the output heading in quotes. For example, to only receive the number of holidays, rather than both holidays and the rule, add "holidays" as the second argument to the existing formula:

```
{
result: { formula: "tblHolidays('holidays',,age, service)",
finalValue: [] }
}
```

In this instance, only the result collection for "holidays" will be returned, 27, 5, 3.

You can pass as many output arguments as needed.

A 2nd optional argument, `ret_header`, is a Boolean argument, that, if True, returns a header for the result collection.

```
{
result: { formula: "tblHolidays('holidays', True, age, service)",
finalValue: [] }
}
```

In this instance, the result collection will include only the "holidays" output parameter with the header "holidays" as the first element in the collection: "holidays", 27, 5, 3.

To only include the optional `ret_header` argument and not both optional arguments, use:

```
{
result: { formula: "tblHolidays(,True, age, service)", finalValue:
[] }
}
```

Decision Table Hit Policies

This section examines the 7 currently supported hit policies: Any, Collect, First, Rule Order, Output Order, Priority Order and Unique. Recall from earlier, that the hit policy specifies how the table will be evaluated when multiple rules are applicable and hence, multiple output values are returned.

The Hit Policy value identifies the supported policies by a capital letter and an operator, when applicable. You may enter the 1st letter of the policy or the whole word. The currently supported Hit types and their meanings are:

Unique (U): A unique rule must "hit" evaluating to a unique result. If multiple rules are "hit", an error will be returned.

Any (A): If rules overlap, but point to the same result, that unique result is returned.

Priority(P): If multiple rules are "hit" and multiple results collected, return only one result with the highest priority; the priorities are defined by the order of the output values.

First (F): Only 1 result is returned for this policy. Once a rule is evaluated successfully, or a hit occurs, the search stops.

Rule Order (R) - If multiple rules are hit, return the collection of results as created in the rule order

Output Order (O) –If multiple rules are hit, return the collection of results in the priority order of the listed output values.

Collect (C) – The same as (R). However, we may make this policy more specific by adding an operator to it in order to allow aggregation.

Note: If aggregating a date, a scalar is returned. If using an operator, output must not be a string, but only a numerical value.

C+ - sum the matched output values

C< - return the min of matched output values

C> - return the max of matched output values

C# - return the number of matched output values

Open the example DT Hit Policy examples.json on the Editor tab of RASON.com.

This example RASON model creates and recalculates seven decision tables; one table for each of the 7 supported hit policies: tblPolicyAny, tblPolicyCollect, tblPolicyUnique, tblPolicyFirst, tblPolicyRuleOrder, tblPolicyOutputOrder and tblPolicyPriorityOrder.

```
{
  modelName: "DTHitPolicyExample",
  modelType: "calculation",
  modelDescription: "Decision Table Hit Policy Example",
  modelType: "calculation",
  decisionTables: {
    "tblPolicyAny": {
      inputs: ["creditRating", "creditCardBalance",
"studentLoanBalance"],
      outputs: ["loanCompliance"],
      refTypes: ["", "", "", ""],
      hitPolicy: "A",
      rules: [
        ["A", "<10000", "<50000", "compliant"],
        ["Not(A)", "-", "-", "not compl"],
        ["-", ">=10000", "-", "not compl"],
        ["-", "-", ">=50000", "not compl"]
      ]
    },
    "tblPolicyCollect": {
      inputs: ["age", "service"],
      outputs: ["holidays"],
      refTypes: ["number", "number", "number"],
      hitPolicy: "C+",
      rules: [
        ["-", "-", 22],
        [">=60", "-", 3],
        ["-", ">=30", 3],
        ["< 18", "-", 5],
        [">=60", "-", 5],
        ["-", ">=30", 5],
        ["[18..60]", "[15..30]", 2],
        ["[45..60]", "<30", 2]
      ]
    },
    "tblPolicyUnique": {
      inputs: ["age", "medHistory"],
      outputs: ["riskRating", "rule"],
      refTypes: ["", "", "", ""],
      hitPolicy: "U",
      rules: [
        [">60,<25", "good", "medium", "r1"],
```

```

        [">60", "bad", "high", "r2"],
        ["[25..60]", "-", "medium", "r3"],
        ["<25", "good", "low", "r4"],
        ["<25", "bad", "medium", "r5"]
    ]
},
"tblPolicyFirst": {
    inputs: ["order", "location", "customer"],
    outputs: ["discount"],
    refTypes: ["", "", "", ""],
    hitPolicy: "F",
rules: [
    ["web", "US", "wholesaler", 10],
    ["phone", "-", "-", 2],
    ["-", "non_US", "-", 0],
    ["-", "-", "retailer", 5]
]
},
"tblPolicyRuleOrder": {
    inputs: ["gpa", "act_count", "honor_member"],
    outputs: ["eligibility", "rule"],
    refTypes: ["number", "number", "boolean", "text", "text"],
    hitPolicy: "R",
    rules: [
        [">3.5", ">=4", true, "20% scolar", "r1"],
        [">3.0", "-", true, "30% loan", "r2"],
        [">3.0", ">=2", false, "20% work", "r3"],
        ["<=3.0", "-", "-", "20% work", "r4"]
    ]
},
"tblPolicyOutputOrder": {
    inputs: ["age", "service"],
    outputs: ["holidays"],
    outputValues: ["27", "5", "3", "2"],
    hitPolicy: "O",
rules: [
    ["-", "-", "age - service"],
    [">=60", "-", 3],
    ["-", ">=30", 3],
    ["<18", "-", 5],
    [">=60", "-", 5],
    ["-", ">=30", 5],
    ["[18..60]", "[15..30]", 2],
    ["[45..60]", "<30", 2]
]
},
"tblPolicyPriorityOrder": {
    inputs: ["age", "medHistory"],
    outputs: ["riskRating"],
    refTypes: [ "", "", "" ],
    outputValues: ["low", "medium", "high"],
    hitPolicy: "P",
    rules: [
        [">=25", "good", "medium"],
        [">60", "bad", "high"],
        ["-", "bad", "medium"],
        ["<25", "good", "low"]
    ]
}

```

```

    ]
  }
},
formulas: {
  "Any": { formula: "tblPolicyAny(,, 'B', 12000, 75000)", finalValue:
    [] },
  "Collect": { formula: "tblPolicyCollect(,, 58, 31)", finalValue: []
  },
  "Unique": { formula: "tblPolicyUnique(,, 54, 'good')",
    finalValue: [] },
  "First": { formula: "tblPolicyFirst(,, 'web', 'non_US', 'retailer')",
    finalValue: [] },
  "Rule_Order": { formula: "tblPolicyRuleOrder('eligibility',, 3.6, 4,
    TRUE)", finalValue: [] },
  "Output_Order": { formula: "tblPolicyOutputOrder(,, 58, 31)",
    finalValue: [] },
  "Priority_Order": { formula: "tblPolicyPriorityOrder(,, 61, 'bad')",
    finalValue: [] }
  }
}

```

To execute all seven decision tables given the input data passed to each table within the formulas section, click POST rason.net/api/model to post the model, then POST rason.net/api/model/{nameorid}/decision to calculate the decision table.

```

Getting model results: GET
https://rason.net/api/model/2590+DTHitPolicyExample+2020-01-20-02-49-27-
161414/result
{ "status": {
  "code": 0,
  "id": "2590+DTHitPolicyExample+2020-01-20-02-49-27-161414",
  "codeText": "Solver has completed the calculation."
},
  "observations": {
    "Any": {
      "value": "not compl"
    },
    "Collect": {
      "value": 30
    },
    "Unique": {
      "value": [ ["medium", "r3"] ]
    },
    "First": {
      "value": 0
    },
    "Rule_Order": {
      "value": ["20% scolar", "30% loan"]
    },
    "Output_Order": {
      "value": [27, 5, 3]
    },
    "Priority_Order": {
      "value": ["medium"]
    }
  }
}

```

Let's look at each table individually.

Hit Policy: Any using the Decision Table, tblPolicyAny

The tblPolicyAny decision table returns the loan compliance for an individual given their credit rating, current card balance and current student loan balance.

This table uses three input parameters, creditRating, creditCardBalance and studentLoanBalance and one output parameter, loanCompliance.

```
{decisionTables: {
  "tblPolicyAny": {
    inputs: ["creditRating", "creditCardBalance",
"studentLoanBalance"],
    outputs: ["loanCompliance"],
    hitPolicy: "A",
    rules: [
      ["A", "<10000", "<50000", "compliant"],
      ["Not (A)", "-", "-", "not compl"],
      ["-", ">=10000", "-", "not compl"],
      ["-", "-", ">=50000", "not compl"]
    ]
  }
},
formulas: {
  "Any": { formula: "tblPolicyAny(, 'B', 12000, 75000)",
    finalValue: [] }
}
}
```

Given the input data, passed to the RASON server within the `formulas` section, of `creditRating = B`, `creditCardBalance = 12,000` and `studentLoanBalance = 75,000`, all rules except for rule 1 are evaluated successfully for a "hit". However, since the result collection only includes 1 "not compl" entry, "not compl" is returned. If there had been more than 1 result in the result collection, say "not compl" and "compliant", an error would have been returned from the RASON Server.

```
"Any": { "value": "not compl"},
```

Hit Policy: Collect using tblPolicyCollect

The tblPolicyCollect decision table returns the total number of holidays allotted to an employee based on their age and number of years in service. In this example, the Collect policy is used with the "+" operator. This operator returns the sum of the collection of results in rule order when multiple rules are successful. The Collect Policy allows 4 operators, + (sum), < (return the minimum of matched output values), > (return the maximum of matched output values) and # (return the number of matched output values).

This decision table specifies two input parameters, age and service and one output parameter, holidays.

```
{decisionTables: {
  "tblPolicyCollect": {
    inputs: ["age", "service"],
    outputs: ["holidays"],
    hitPolicy: "C+",
    rules: [
      ["-", "-", 22],
      [">=60", "-", 3],
      ["-", ">=30", 3],
      ["<18", "-", 5],
      [">=60", "-", 5],
    ]
  }
}
```

```

        ["-", ">=30", 5],
        ["[18..60]", "[15..30]", 2],
        ["[45..60]", "<30", 2]
    ]
}
},
formulas: {
  "Collect": {
    formula: "tblPolicyCollect(, , 58, 31)",
    finalValue: [] }
  }
}

```

Given the input data of age = 58 and service = 31, within the formulas section, three rules, 1, 3 and 6, are successful which return the result values of 22, 3 and 5, respectively. Since this Hit policy totals the results, the result, 30, is returned.

Let's see what happens when we change the Hit Policy to C<. Simply change hitPolicy: "C+", to hitPolicy: "C<" then recalculate the decision table. When the Hit Policy is equal to C<, the minimum result value, 3, is returned.

```
"Collect": { "value": 3 }
```

Let's repeat these steps for "C>" to find the maximum result value of 22,

```
"Collect": { "value": 22 }
```

and "C#" to find the number of returned results, 3.

```
"Collect": { "value": 3 }
```

Hit Policy: First using tblPolicyFirst

In this decision table, a percentage discount is returned based on where an order originated, on the web or by phone, the location of the order, inside or outside the US, and customer type, wholesaler or retail according to the First Hit Policy. When the hit policy, First is in use, only the first entry in the result collection is returned.

This decision table takes three inputs, order, location, and customer, and produces one output discount.

```

{decisionTables: {
  "tblPolicyFirst": {
    inputs: ["order", "location", "customer"],
    outputs: ["discount"],
    hitPolicy: "F",
    rules: [
      ["web", "US", "wholesaler", 10],
      ["phone", "-", "-", 2],
      ["-", "non_US", "-", 0],
      ["-", "-", "retailer", 5]
    ]
  }
},
formulas: {
  "First": { formula: "tblPolicyFirst(, , 'web',      'non_US',
    'retailer')", finalValue: [] }
  }
}

```

Given the input data passed in formulas, order = web, location = non_US and customer = retailer, two rules are "hit", rule 3 and rule 4. However, since we are using the First Hit Policy, which returns the first successful output, only the result for rule 3 is returned, 0.

```
"First": { "value": 0 }
```

Hit Policy: Rule Order using tblPolicyRuleOrder

This decision table returns the amount of funding available to a student based on their grade point average and number of times they took the ACT. Funds may be provided via a scholarship, loan, or work study program. The "Rule Order" Hit policy returns the collection of results as created in rule order.

This decision table has three input parameters, gpa, act_count and honor_member, and two output parameters, eligibility and rule. Notice that this decision table also has input and output values. The input values are number, number and Boolean and pertain to the input parameters, gpa, act_count and honor_member respectively. The two output values are both "text" and pertain to both output parameters. Recall that input and output values specify the input and output type such as number or text.

```
{decisionTables: {
  "tblPolicyRuleOrder": {
    inputs: ["gpa", "act_count", "honor_member"],
    outputs: ["eligibility", "rule"],
    refTypes: ["number", "number", "boolean", "text", "text"],
    hitPolicy: "R",
    rules: [
      [ ">3.5", ">=4", true, "20% scolar", "r1"],
      [ ">3.0", "-", true, "30% loan", "r2"],
      [ ">3.0", ">=2", false, "20% work", "r3"],
      [ "<=3.0", "-", "-", "20% work", "r4"]
    ]
  }
},
formulas: {
  "Rule_Order": { formula: "tblPolicyRuleOrder('eligibility',, 3.6,
4,
TRUE)", finalValue: [] }
}
}
```

Given an honor student with a gpa = 3.6 taking the ACT for the 4th time (see the formulas section above), this student would be eligible to receive a scholarship covering 20% of tuition (rule 1) and a student loan covering 30% of tuition (rule 2). Rules 3 and 4 are not applicable because the student is an honor student (honor_member = TRUE) and the student's GPA is greater than 3.0.

```
"Rule_Order": { "value": ["20% scolar", "30% loan"] }
```

Hit Policy: Output Order using tblPolicyOutputOrder

This decision table returns the number of holidays allotted to an employee based on the employee's age and number of years of service. The output is returned in the order as given for the Output Values: 27, 5, 3, 2,

This table takes two inputs, age and service and returns one output, holidays. This table includes the outputValues argument which determines the order of the output.

```
{decisionTables: {
  "tblPolicyOutputOrder": {
    inputs: ["age", "service"],
    outputs: ["holidays"],
    outputValues: ["27", "5", "3", "2"],

```



```

        hitPolicy: "O",
        rules: [
            ["-", "-", "age - service"],
            [">=60", "-", 3],
            ["-", ">=30", 3],
            ["<18", "-", 5],
            [">=60", "-", 5],
            ["-", ">=30", 5],
            ["[18..60]", "[15..30]", 2],
            ["[45..60]", "<30", 2]
        ]
    },
    formulas: {
        "Output_Order": { formula: "tblPolicyOutputOrder(,,58, 31)",
        finalValue: [] }
    }
}

```

According to the rules of the table, a 58 year old employee with 31 years of service is allotted 27 + 3 + 5 vacation days via rules 1, 3 and 6, respectively. However, since the Hit Policy specifies that the results must be returned in priority order according to the list passed to `outputValues`, the result collection is listed as 27, 5 and 3.

```
"Output_Order": { "value": [27, 5, 3] }
```

Hit Policy: Priority using tblPolicyPriorityOrder

According to the Priority Hit Policy, if multiple rules are "hit" and multiple results are collected, only one result will be returned. The priority of the returned result is defined by the order of the output values in the header.

The `tblPolicyPriorityOrder` table returns a risk of disease rating of low, medium, or high based on a person's age and medical history. This table requires two inputs, age and `medHistory`, and returns one result, `riskRating`, which must be returned in the following order: low, medium, high.

```

{decisionTables: {
    "tblPolicyPriorityOrder": {
        inputs: ["age", "medHistory"],
        outputs: ["riskRating"],
        refTypes: [ "", "", "" ],
        outputValues: ["low", "medium", "high"],
        hitPolicy: "P",
        rules: [
            [">=25", "good", "medium"],
            [">60", "bad", "high"],
            ["-", "bad", "medium"],
            ["<25", "good", "low"]
        ]
    }
},
formulas: {
    "Priority_Order": {
        formula: "tblPolicyPriorityOrder(,,61, 'bad')",
        finalValue: []
    }
}
}

```

Note that according to our criteria of age = 61 and medHistory = bad, two rules are returned as successful, rule 2 (age > 60 and medHistory = "bad") and rule 3 (age = "-" and medHistory = "bad"). Why isn't "high" returned? Because the priority list given for outputValues (low, medium, high) specifies that "medium" should have higher priority than "high".

```
"Priority_Order": { "value": [medium] }
```

Hit Policy: Unique

According to the Unique Hit Policy, only one rule may be returned. If multiple rules are successful and multiple results are collected, an error will be returned.

This table takes two inputs, age and medHistory, and returns two outputs, riskRating and rule.

```
{decisionTables: {
  "tblPolicyUnique": {
    inputs: ["age", "medHistory"],
    outputs: ["riskRating", "rule"],
    hitPolicy: "U",
    rules: [
      [ ">60,<25", "good", "medium", "r1"],
      [ ">60", "bad", "high", "r2"],
      [ "[25..60]", "-", "medium", "r3"],
      [ "<25", "bad", "medium", "r4"]
    ]
  }
},
formulas: {
  "Unique": { formula: "tblPolicyUnique(,54, 'good')", finalValue:
    [] }
}
```

In this example, given an age of 54 and a "good" medical history, a "medium" risk rating is returned by rule 3.

```
"Unique": { "value": [ ["medium", "r3"] ] }
```

Decision Tables with Dates and Times

This section presents a special class of tables, in which the specific Date, Time and Duration data types are utilized. Open the example, DT Date & Time examples.json on the Editor page on RASON.com to display the following example RASON code. For a list of supported operators, see the Decision Tables section within the RASON Reference Guide.

This example model invokes three decision tables: tblRiskRating, tblTollTax and tblParkingFee. In the first decision table, tblRiskRating, a decision table is created to determine the medical risk of an individual given their age and medical history. The second decision table, tblTollTax, determines the rate of traffic and the toll, given the time entering the toll road and the third decision table, tblParkingFee, determines a parking fee based on the time spent in the parking facility. Let's take a closer look at all three.

```
{
  "modelName": "DTDateTimeExample",
  "modelType": "calculation",
  "modelDescription": "Date and Time Decision Table
Examples",
  "decisionTables": {
```

```

    "tblRiskRating": {
      "inputs": [ "birthDate", "medHistory" ],
      "outputs": [ "riskRating" ],
      "refTypes": [ "date", "text", "text" ],
      "rules": [
        [ ">'1970-05-05'", "good", "medium" ],
        [ ">'1970-05-05'", "bad", "high" ],
        [ ["'1935-05-05'..'1970-05-05'"], "-", "medium"
],
        [ "<'1935-05-05'", "good", "low" ],
        [ "<'1935-05-05'", "bad", "medium" ]
      ],
      "hitPolicy": "U"
    },
    "tblTollTax": {
      "inputs": [ "operTime" ],
      "outputs": [ "traffic", "toll" ],
      "refTypes": [ "time", "text", "number" ],
      "rules": [
        [ ">'19:00:00'", "low", 5 ],
        [ ["'15:00:00'..'19:00:00'"], "high", 7 ],
        [ ["'09:00:00'..'15:00:00'"), "medium", 6 ],
        [ ["'06:00:00'..'09:00:00'"), "high", 7 ],
        [ "<'06:00:00'", "low", 5 ]
      ],
      "hitPolicy": "U"
    },
    "tblParkingFee": {
      "inputs": [ "dtDuration" ],
      "outputs": [ "parkingFee" ],
      "refTypes": [ "duration", "number" ],
      "rules": [
        [ "<'PT20M'", 0 ],
        [ ["'PT20M'..'PT1H'"),
          "2
*ceiling(duration(dtDuration)/duration('PT20M'))"
        ],
        [ ["'PT1H'..'PT4H'"),
          "6
*ceiling(duration(dtDuration)/duration('PT1H'))"
        ],
        [
          ">='PT4H'",
          "30*ceiling(duration(dtDuration)/duration('P1D'))"
        ]
      ],
    },

```

```

        "hitPolicy": "U"
    }
},
"data": {
    "Date1": {
        "value": "'1964-05-05'",
        "comment": "bd"
    },
    "Date2": {
        "value": "good",
        "comment": "mh"
    },
    "actualTimeEntered": {
        "value": "'18:50:05'",
        "comment": "bynow"
    },
    "durationParked": {
        "value": "PT25M",
        "comment": "period"
    }
},
"formulas": {
    "dateResult": {
        "formula": "tblRiskRating(,,Date1, Date2)",
        "finalValue": []
    },
    "tollResult": {
        "formula": "tblTollTax('toll',,actualTimeEntered)",
        "finalValue": []
    },
    "trafficResult": {
        "formula":
"tblTollTax('traffic',,actualTimeEntered)",
        "finalValue": []
    },
    "durationResult": {
        "formula": "tblParkingFee(,,durationParked)",
        "finalValue": []
    }
}
}

```

Decision Table Containing Dates

The decision table, `tblRiskRating` returns the medical risk rating of an individual based on their age and medical history. This decision table takes two types of input data, `birthDate` and `medHistory`. The `refTypes` argument stipulates that `birthDate` must be a date, and `medHistory` and `riskRating` (the output) must be text strings. The hit policy is "U" for Unique which means that a unique rule must resolve to a unique result. If multiple rules are "hit", an error will be returned.

```

"tblRiskRating": {
  inputs: ["birthDate", "medHistory"],
  outputs: ["riskRating"],
  refTypes: ["date", "text", "text"],
  hitPolicy: "U",
  rules: [
    [ ">'1970-05-05'", "good", "medium"],
    [ ">'1970-05-05'", "bad", "high"],
    [ ["'1935-05-05'.. '1970-05-05'"], "-", "medium"],
    [ "<'1935-05-05'", "good", "low"],
    [ "<'1935-05-05'", "bad", "medium"]
  ]
},

```

- The first rule states that if the patient's birthDate occurs after 1970-05-05 and the patient's medical history is "good", the returned riskRating will be "medium".
- The second rule states that if the patient's birthDate occurs after 1970-05-05 and the patient's medical history is "bad", the returned riskRating will be "high".
- The third rule states that if the patient's birthDate occurs on or between 1935-05-05 and 1970-05-05 with any type of medical history ("-"), the returned riskRating will be "medium".
- The fourth rule states that if the patient's birthDate occurs before 1935-05-05 and the patient's medical history is "good", the returned riskRating will be "low".
- The fifth rule states that if the patient's birthDate occurs before 1935-05-05 and the patient's medical history is "bad", the returned riskRating will be "medium".

Notice that all dates within the rules are surrounded by single quotes. S-FEEL defines a single format for the **Date** type '**yyyy-mm-dd**'. Optionally a date may begin with the letter D, for example "D1964-05-05". Other date type formats are not supported.

Input data is passed in the data section: birthDate = 1964-05-05 and medHistory = "good".

```

"Date1": { value: "'1964-05-05'", comment: "bd" },
"Date2": { value: "good", comment: "mh" },

```

The final result request, dateResult, passes the input data to the decision table, tblRiskRating, asks for the final values.

```

"dateResult": { formula: "tblRiskRating(, ,Date1, Date2)",
finalValue: [] },

```

Given this input data, only one rule is successful, rule 3, which states that if the patient's birthDate occurs on or between 1935-05-05 and 1970-05-05 with any type of medical history ("-"), the returned riskRating will be "medium".

```

"observations": { "dateresult": { "value": "medium" },

```

Decision Table Containing Time

The decision table, tblTollTax returns the type of traffic one might encounter and the toll cost incurred when entering a toll road. This decision table takes one input, operTime or the time the car is expected to enter the toll road. Two outputs are returned, traffic, which specifies if traffic will be high medium or low, and the toll cost. According to our refTypes component, the input operTime must be a time, the output traffic will return a text string and the output toll will return a number. The hit policy is "U" for Unique.

```

"tblTollTax": {
  inputs: ["operTime"],
  outputs: ["traffic", "toll"],

```

```

    refTypes: ["time", "text", "number"],
    hitPolicy: "U",
    rules: [
      [">'19:00:00'", "low", 5],
      ["['15:00:00'..'19:00:00']", "high", 7],
      ["['09:00:00'..'15:00:00']", "medium", 6],
      ["['06:00:00'..'09:00:00']", "high", 7],
      ["<'06:00:00'", "low", 5]
    ]
  },

```

- The first rule states that if a vehicle enters the toll road after 19:00 (or 7 pm), traffic will be low and the assessed toll will be \$5.
- The second rule states that if a vehicle enters the toll road at or between 15:00 (or 3 pm) and 19:00, inclusive, (7 pm), traffic will be high and the assessed toll will be \$7.
- The third rule states that if a vehicle enters the toll road at or after 9:00 (or 9 am) and before 15:00 (3 pm), traffic will be medium and the assessed toll will be \$6.
- The fourth rule states that if a vehicle enters the toll road at or after 6:00 (or 6 am) and before 9:00 (9 am), traffic will be high and the assessed toll will be \$7.
- The fifth rule states that if a vehicle enters the toll road before 6:00 (or 6 am) traffic will be low and the assessed toll will be \$5.

Again, notice that all times must be surrounded by single quotes. Optionally, we could enter the times using the format beginning with the letter **T** such as in “**T**18:29:30”. Usually this time format denotes local time. If we want to specify the UTC time, add **z** at the end of the format, for example “**T**18:29:30z”. However, it’s important to note that the **z** time format is just a readable format. The RASON Server cannot effectively convert local and zulu times, because the location of the local time is not specified in the supported format.

Input data is passed in the data section: actualTimeEntered = 18:50:05.

```

"actualTimeEntered": { value: "'18:50:05'", comment: "bynow" },

```

The final result requests, tollResult and trafficResult, pass the input data to the decision table, tblTollTax, and request the final values.

```

"tollResult": { formula: "tblTollTax('toll',,actualTimeEntered)",
  finalValue: [] },
"trafficResult": { formula:
"tblTollTax('traffic',,actualTimeEntered)", finalValue: [] },

```

Given this input data, only one rule is successful, rule 2, which states that if the vehicle enters the toll road at exactly 15:00 (or 3 pm) or till and 19:00, inclusive, (7 pm), traffic will be high and the assessed toll will be \$7.

```

"tollResult": { "value": 7 },
"trafficResult": { "value": "high" },

```

Decision Table Containing Duration

Finally, let’s take a look into an example using the data type, duration. The table, tblParkingFee, computes a parking fee based on how long the vehicle is parked in the parking facility. This decision table requires just one input, dtDuration, and one output, parkingFee. According to the refType component, the input dtDuration must be a duration and a number will be returned for the output, parkingFee.

```

"tblParkingFee": {
  inputs: ["dtDuration"],

```

```

    outputs: ["parkingFee"],
    refTypes: ["duration", "number"],
    hitPolicy: "U",
    rules: [
      ["<'PT20M'", 0],
      ["['PT20M'..'PT1H')", "2
      *ceiling(duration(dtDuration)/duration('PT20M'))"],
      ["['PT1H'..'PT4H')", "6
      *ceiling(duration(dtDuration)/duration('PT1H'))"],
      [">='PT4H'",
      "30*ceiling(duration(dtDuration)/duration('P1D'))"]
    ]
  }
},

```

There are two types of formats for Duration:

- **P1Y2M** - measures duration in months
- **P1DT1H20M10S** – measures duration in seconds

The individual components in these formats are optional; if we have 0 hours to represent, we may skip the hour component. For example, we can represent a 20 minute duration using either: PT0H20M0S or PT20M.

While we can skip a duration component in the format, the order of components in a format must be followed; entering first minutes and then hours; PT20M1H is not a currently supported syntax.

- The first rule states that if a vehicle is parked for less than 20 minutes, no parking fee will be assessed.
- The second rule states that if a vehicle is parked between 20 minutes less than 1 hour, a parking fee will be assessed equal to 2 *ceiling(duration(dtDuration)/duration('PT20M')).
- The third rule states that if a vehicle is parked between 1 hour (inclusive) and less than 4 hours, a parking fee will be assessed equal to 6 *ceiling(duration(dtDuration)/duration('PT1H')).
- The fourth rule states that if a vehicle is parked for four or more hours, a parking fee will be assessed equal to 30*ceiling(duration(dtDuration)/duration('P1D')).

Input data is passed in the data section: durationParked = 25 minutes (PT25M).

```
"durationParked": { value: "PT25M", comment: "period" }
```

The final result requests, durationResult, passes the input data to the decision table, tblParkingFee, and requests the finalValue, in this case, the parking fee.

```
"durationResult": { formula: "tblParkingFee(, , durationParked)",
finalValue: [] }
```

Given this input data, only one rule is successful, rule 2, which states that if the vehicle is parked 20 minutes or up to 1 hour (not inclusive) the calculated toll will be 2 *ceiling(duration(dtDuration)/duration('PT20M')) or \$4.

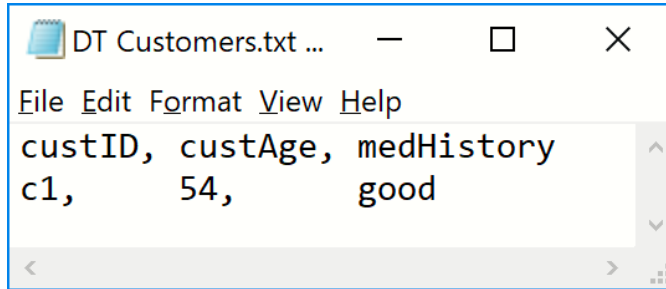
```
"durationResult": { "value": 4 }
```

Using an External Data Source

In RASON, it's possible to import input parameters from an external data-source within the dataSource section by using the binding "get" method within the data section. In the example below (see Datasource example.json on the Editor page) the CSV file, DT Customers.txt is imported

into the RASON model within the datasources section. DT Customers.txt contains the following fields, as shown in the screenshot below: custID, custAge and medHistory. The indexCols property is set to custID, which is, of course, the customer or patient ID. The valueCols property is set to custAge and medHistory. Within the data section of the RASON model, custAge is bound to the age variable and medHistory is bound to the med variable.

Note: Recall that indexCols and valueCols properties describe a RASON Table while colIndex and rowIndex properties describe a dataframe. These properties may not be used interchangeably.



```
{ modelName: "DTDataSourceExample",
  modelDescription: "Decision table inputs from a datasource",
  modelType: "calculation",
  datasources: {
    cust_data: {
      type: "csv",
      connection: "DT Customers.txt",
      selection: "",
      indexCols: ['custID'],
      valueCols: ['custAge', 'medHistory'],
      direction: "import"
    }
  },
  data: {
    age: { binding: 'cust_data', valueCol: 'custAge' },
    med: { binding: 'cust_data', valueCol: 'medHistory' }
  },
  decisionTables: {
    "PolicyUnique": {
      hitPolicy: "U",
      inputs: ['age', 'medHistory'],
      outputs: ['riskRating', 'rule'],
      rules: [
        ['>60', '<25', 'good', 'medium', 'r1'],
        ['>60', 'bad', 'high', 'r2'],
        ['[25..60]', '-', 'medium', 'r3'],
        ['<25', 'good', 'low', 'r4'],
        ['<25', 'bad', 'medium', 'r5']
      ]
    }
  },
  formulas: {
    "res": {
      formula: "PolicyUnique(, , age[1], med[1])", comment:
"policy",
      finalValue: [] }
  }
}
```


}

Decision Table Components

This table returns a risk rating for an individual based on their age and medical history (medHistory). A graphical representation of this Decision Table is below. Notice that no Input or Output Values exist.

U	age	medHistory	riskRating	rule
1	>60,<25	Good	medium	r1
2	>60	Bad	high	r2
3	[25..60]	-	medium	r3
4	<25	Good	low	r4
5	<25	bad	medium	r5

Decision Table Name: "PolicyUnique"

Hit Policy: "U" which stands for "Unique". A unique rule must "hit" evaluating to a unique result. If multiple rules are "hit", an error will be returned.

Input Parameters. inputs: ['age', 'medHistory']

Input Values (Optional): No input values exist in this decision table.

Output Parameters outputs: ['riskRating', 'rule'],

This example contains two output Parameters, riskRating and rule. After calculation, a table returns a few selected or all output parameter values in the form of an array. If a table has a single output or a single output has been selected, the result will be a scalar value. These input parameters belong to the local scope of this table.

Output Values (Optional): No output values exist in this decision table.

Decision Table Rules (or Unary tests):

```
rules: [  
  ['>60,<25','good','medium','r1'],  
  ['>60','bad','high','r2'],  
  ['[25..60]','-','medium','r3'],  
  ['<25','good','low','r4'],  
  ['<25','bad','medium','r5']  
]
```

Decision Table Results

In the formulas section, the table is calculated based on the variables age and med. Recall that these variables are bound to the data source valuecols of custAge and medHistory, respectively.

```
"res": { formula: "PolicyUnique(,age[1], med[1])", comment:  
"policy", finalValue: [] }
```

Decision Tables accept only scalar arguments as inputs. Since “age” and “med” were created by binding to an array, we pass only the first element as in age[1] and med[1]. (Alternatively, we could have also bound a variable to a table.)

In this example, given an age of 54 and a "good" medical history, a "medium" risk rating is returned by rule 3.

Getting model results: GET
https://rason.net/api/model/2590+DTDataSourceExample+2020-02-27-17-19-41-239540/result

```
{
  "status": {
    "code": 0,
    "id": "2590+DTDataSourceExample+2020-02-27-17-19-41-239540",
    "codeText": "Solver has completed the calculation."
  },
  "observations": {
    "res": {
      "value": [
        ["medium", "r3"]
      ]
    }
  }
}
```

The result is an array despite the ‘U’ policy, because 2 outputs are returned: "medium" and "r3".

This example demonstrates an **implicit array return** from a Decision table. In RASON, array results can be processed later through the index operator []. To access the first element res array, use res.value[1], to access the 2nd element, use res.value[2].

RASON Decision Services makes it exceptionally easy to work with data sources in the Microsoft ecosystem, by creating a Data Connection on the user's My Account page on www.RASON.com. The RASON service supports the following data connections.

- OneDrive and OneDrive for Business
- Common Data Service for Dynamics 365, Power Apps and Power Automate
- OData and CDS support for Power BI
- CData Cloud Hub support for access to 100+enterprise data sources.

For more information on how to create and maintain Data Connections, see the previous RASON Services Web IDE chapter.

Loan Strategy Example

Open the DT Loan Strategy Model to learn how to use cascading decision tables where the output of one decision table is the input to another. This model reflects a loan strategy laid out by a financial institution in accepting or declining a loan application.

Eight decision tables are invoked in the example code below:

- tblAppRiskScore: This table returns a partial "risk of default" score based on a loan applicant's age, marital and employment status.
- tblBureauRiskCat: This table returns a risk category of high, medium, or low based on their customer status (existing or not) and their application risk score.
- tblBureauCallType: This table returns a bureau call type of "full", "mini" or "none" based on a bureau risk category of "high,medium", "low", or "veryLow, decline"

- `tblEligibility`: This table returns the applicant's eligibility based on three criteria: bureau risk category, an affordability rating and customer age.
- `tblCreditContFactor`: This table returns a credit factor based on a customer's risk rating.
- `tblStrategy`: This table returns a loan strategy for the financial institution based on an applicant's eligibility and loan type.
- `tblPostBureauRiskCat`: This table returns a risk category based on various customer features.
- `tblRouting`: This table recommends to either decline or accept an applicant's loan.

The data section contains the input parameters for the decision tables. Input parameters are all constant values.

- `custExist = False`
- `custAge = 40`
- `maritalStatus = "s"`
- `employmentStatus = "selfEmployed"`
- `creditScore = 610`
- `bankrupt = False`
- `monthIncome = 2500`
- `monthExpenses = 1000`
- `loanType = "standard"`
- `loanRate = 5.0`
- `loanTerm = 30`
- `loanAmnt = 100000`

We will examine each table in turn.

```
{
  modelName: "DTloanStrategyFromScratch",
  modelDescription: "Loan Strategy model",
  modelType: "calculation",
  data: {
    comment: "use binding to feed dif. values",
    custExist: { value: false },
    custAge: { value: 40 },
    maritalStatus: { value: 's' },
    employmentStatus: { value: 'selfEmployed' },
    creditScore: { value: 610 },
    bankrupt: { value: false },
    monthIncome: { value: 2500 },
    monthExpenses: { value: 1000 },
    loanType: { value: 'standard' },
    loanRate: { value: 5.0 },
    loanTerm: { value: 30 },
    loanAmnt: { value: 100000.0 }
  },
  decisionTables: {
    tblAppRiskScore: {
      inputs: ['custAge', 'maritalStatus', 'employmentStatus'],
      outputs: ['partialScore'],
      inputValues: [
```

```

        ['[18..120]',      's', 'unemployed'],
        [null,            'm', 'employed'],
        [null,            null, 'selfEmployed'],
        [null,            null, 'student']],
    rules: [
        ['[18..21]',      '-', '-',      32],
        ['(21..25]',      '-', '-',      35],
        ['(25..35]',      '-', '-',      40],
        ['(35..49]',      '-', '-',      43],
        ['>49',           '-', '-',      48],
        ['-',             's', '-',      25],
        ['-',             'm', '-',      45],
        ['-',             '-', 'unemployed', 15],
        ['-',             '-', 'student',    18],
        ['-',             '-', 'employed',   45],
        ['-',             '-', 'selfEmployed', 36]],
    hitPolicy: 'collect+'
  },
  tblBureauRiskCat: {
    inputs: ['custExist', 'appRiskScore'], outputs:
    ['bureauRiskCat'],
    rules: [
        [false,          '<100',      'high'],
        [false,          '[100..120)', 'medium'],
        [false,          '[120..130]', 'low'],
        [false,          '>130',      'veryLow'],
        [true,           '<80',        'decline'],
        [true,           '[80..90)',   'high'],
        [true,           '[90..110]',  'medium'],
        [true,           '>110',      'low']],
    hitPolicy: 'unique'
  },
  tblBureauCallType: {
    inputs: ['bureauRiskCat'], outputs: ['bureauCallType'],
    rules: [
        ['high,medium',  'full'],
        ['low',          'mini'],
        ['veryLow,decline', 'none']],
    hitPolicy: 'unique'
  },
  tblEligibility: {
    inputs: ['bureauRiskCat', 'bureauAfford', 'custAge'],
    outputs: ['eligibility'],
    outputValues: ['ineligible', 'eligible'],
    rules: [
        ['decline',      '-',      'ineligible'],
        ['-',            false,     'ineligible'],
        ['-',            '-',      '<18', 'ineligible'],
        ['-',            '-',      '-',  'eligible']],
    hitPolicy: 'priority'
  },
  tblCreditContFactor: {
    inputs: ['bureauRiskCat'], outputs: ['creditContFactor'],
    rules: [
        ['high,decline', 0.6],
        ['medium',       0.7],
        ['low,veryLow',  0.8]],
  },

```

```

        hitPolicy: 'unique'
    },
    tblStrategy: {
        inputs: ['eligibility', 'bureauCallType'], outputs:
['strategy'],
        rules: [
            ['ineligible',          '-',          'decline'],
            ['eligible',            'full,mini',    'bureau'],
            ['eligible',            'none',         'through']],
        hitPolicy: 'unique'
    },
    tblPostBureauRiskCat: {
        inputs: ['custExist', 'appRiskScore', 'creditScore'],
        outputs: ['postBureauRiskCat'],
        rules: [
            [false,          '<120',          '<590',          'high'],
            [false,          '<120',          '[590..610]',      'medium'],
            [false,          '<120',          '>610',          'low'],
            [false,          '[120..130]',    '<600',          'high'],
            [false,          '[120..130]',    '[600..625]',    'medium'],
            [false,          '[120..130]',    '>625',          'low'],
            [false,          '>130',          '-',          'veryLow'],
            [true,           '<=100',          '<580',          'high'],
            [true,           '<=100',          '[580..600]',    'medium'],
            [true,           '<=100',          '>600',          'low'],
            [true,           '>100',          '<590',          'high'],
            [true,           '>100',          '[590..615]',    'medium'],
            [true,           '>100',          '>615',          'low']],
        hitPolicy: 'unique'
    },
    tblRouting: {
        inputs:
['postBureauRiskCat', 'postBureauAfford', 'bankrupt', 'creditScore'],
        outputs: ['routing'],
        inputValues: [
            [null,          null,          true, '[0..999]'],
            [null,          null,          false, null]],
        outputValues: ['decline', 'refer', 'accept'],
        rules: [
            ['-',          false,          '-', '-', 'decline'],
            ['-',          '-',          true, '-', 'decline'],
            ['high', '-',          '-', '-', 'refer'],
            ['-',          '-',          '-', '<580', 'refer'],
            ['-',          '-',          '-', '-', 'accept']],
        hitPolicy: 'priority'
    }
},
    formulas: {
        monthFee: { formula: "IF(loanType = 'standard', 20, IF(loanType =
'special', 25, 0))" },
        monthRepay: { formula: "-PMT(loanRate%/12, loanTerm*12,
loanAmnt)" },
        monthInstall: { formula: "monthRepay + monthFee" },
        disposIncome: { formula: "monthIncome - (monthRepay +
monthExpenses)" },
        appRiskScore: { formula: "tblAppRiskScore(, , custAge,
maritalStatus, employmentStatus)" },
    }
}

```

```

        bureauRiskCat: { formula: "tblBureauRiskCat(,,custExist,
        appRiskScore)" },
bureauAfford: { formula: "IF(disposIncome *
tblCreditContFactor(,,bureauRiskCat) > monthInstall, true, false)"
},
strategy: { formula: "tblStrategy(,,tblEligibility(,,bureauRiskCat,
bureauAfford, custAge), tblBureauCallType(,,bureauRiskCat))",
finalValue: [] },comment: "additional stage follows",
postBureauRiskCat: { formula: "tblPostBureauRiskCat(,,custExist,
appRiskScore, creditScore)" },
postBureauAfford: { formula: "IF(disposIncome *
tblCreditContFactor(,,postBureauRiskCat) > monthInstall, true,
false)" },
routing: { formula: "tblRouting(,,postBureauRiskCat,
postBureauAfford, bankrupt, creditScore)", finalValue: [] }
}
}

```

Scroll down to the formulas section. The first few entries here are simple preliminary formulas that are used to initialize four values: monthFee, monthRepay, monthInstall and disposIncome.

1. The first formula:

```

monthFee: { formula: "IF(loanType = 'standard', 20, IF(loanType =
'special', 25, 0))" },

```

inspects loanType in the data section. In this example, for loanType = standard "20" is returned for monthFee.

2. The second formula:

```

monthRepay: { formula: "-PMT(loanRate%/12, loanTerm*12, loanAmnt)"
},

```

calculates a monthly payment using the internal PMT function based on the loanRate (5.0), loanTerm (30) and loanAmnt (100000) or monthRepay = -PMT(.05/12, 30 * 12, 100000) , or \$536.82

3. The third formula:

```

monthInstall: { formula: "monthRepay + monthFee" },

```

calculates the monthly installment payment by adding monthRepay (calculated in 2, \$536.82) with monthFee (calculated in 1, 20) or = \$536.82 + 20.

4. The fourth formula:

```

disposIncome: { formula: "monthIncome - (monthRepay +
monthExpenses)" },

```

calculates the customer's disposable income using the formula = monthIncome (2500) – (monthRepay (536.82) + monthExpenses (1000)) = 963.18. Both monthIncome and monthExpenses are given in data and monthRepay is calculated in 2.

5. The fifth formula is the first formula to invoke a decision table.

```

appRiskScore: { formula: "tblAppRiskScore(,,custAge, maritalStatus,
employmentStatus)" },

```

The first table, tblAppRiskScore, appearing in order in the RASON code above, returns a partial "risk of default" score based on a loan applicant's age, marital and employment status. Higher scores are given to older, employed, married applicants.

All input parameters are constant values, given in the data section.

Notice the inputValues component which lists the domains of each input: [18..120] for custAge, 's' or 'm' (only) for maritalStatus, 'unemployed', 'employed', 'selfEmployed', or 'student' (only) for employmentStatus. If data outside of these domains (i.e. 'p' for maritalStatus) is passed to this decision table, an error will be returned.

```
tblAppRiskScore: {
  inputs: ['custAge', 'maritalStatus', 'employmentStatus'],
  outputs: ['partialScore'],
  inputValues: [
    ['[18..120]', 's', 'unemployed'],
    [null, 'm', 'employed'],
    [null, null, 'selfEmployed'],
    [null, null, 'student']],
  rules: [
    ['[18..21]', '-', '-', 32],
    ['(21..25]', '-', '-', 35],
    ['(25..35]', '-', '-', 40],
    ['(35..49]', '-', '-', 43],
    ['>49', '-', '-', 48],
    ['-', 's', '-', 25],
    ['-', 'm', '-', 45],
    ['-', '-', 'unemployed', 15],
    ['-', '-', 'student', 18],
    ['-', '-', 'employed', 45],
    ['-', '-', 'selfEmployed', 36]],
  hitPolicy: 'collect+'
},
```

There are three applicable rules for the input parameters: custAge = 40, maritalStatus = "s" and employmentStatus = "selfEmployed" or 4, 6 and 11, respectively. Which return the partial scores of 43, 25 and 36, respectively. Since the Hit Policy is "C+" the result collection is added together, 43 + 25 + 36 = 104 appRiskScore.

6. The sixth formula invokes the tblBureauRiskCat decision table.

```
bureauRiskCat: { formula: "tblBureauRiskCat(, , custExist,
appRiskScore)" },
```

The tblBureauRiskCat decision table returns a risk category of high, medium, low, very low or decline based on the customer status (existing or not) and the customer's application risk score. Non-existing customers are allowed a higher risk score. This table has two input parameters, custExist (as given in the data section) and appRiskScore, which is the output from the tblAppRiskScore decision table. We will discuss this output in depth below.

```
tblBureauRiskCat: {
  inputs: ['custExist', 'appRiskScore'],
  outputs: ['bureauRiskCat'],
  rules: [
    [false, '<100', 'high'],
    [false, '[100..120)', 'medium'],
    [false, '[120..130]', 'low'],
    [false, '>130', 'veryLow'],
    [true, '<80', 'decline'],
    [true, '[80..90)', 'high'],
    [true, '[90..110]', 'medium'],
    [true, '>110', 'low']],
  hitPolicy: 'unique'
},
```

Based on the conditions given to bureauRiskCat within formulas (custExist = False and appRiskScore = 104), one rule is "hit", rule 2. This rule returns "medium" for the bureau risk category.

7. The seventh formula invokes a decision table within an IF statement.

```
bureauAfford: { formula: "IF(disposIncome *  
tblCreditContFactor(,,bureauRiskCat) > monthInstall, true, false)"  
},
```

Given the input parameter bureauRiskCat (calculated in 6), tblCreditContFactor returns 0.7 for bureauRiskCat = "medium".

```
tblCreditContFactor: {  
  inputs: ['bureauRiskCat'], outputs: ['creditContFactor'],  
  rules: [  
    ['high,decline', 0.6],  
    ['medium', 0.7],  
    ['low,veryLow', 0.8]],  
  hitPolicy: 'unique'  
},
```

The formula, =IF(disposIncome (963.18) * 0.7 > monthInstall (556.82), then True, else False, returns TRUE.

The eight formula illustrates how one could invoke multiple decision tables at once. This decision table returns the bureau call type based on the bureau risk category returned from 6.

8. The next formula illustrates how one could write a cascading formula where the input of one decision table becomes an input to another. In this case, the results from tblEligibility and tblBureauCallType form the input parameters to tblStrategy.

```
strategy: { formula: "tblStrategy(,,tblEligibility(,,bureauRiskCat,  
bureauAfford, custAge), tblBureauCallType(,,bureauRiskCat))",  
finalValue: [] },
```

Given the input parameter bureauRiskCat, or medium – calculated in 6, tblBureauCallType returns "full".

```
tblBureauCallType: {  
  inputs: ['bureauRiskCat'],  
  outputs: ['bureauCallType'],  
  rules: [  
    ['high, medium', 'full'],  
    ['low', 'mini'],  
    ['veryLow, decline', 'none']],  
  hitPolicy: 'unique'  
},
```

Given the input parameters bureauRiskCat (medium), bureauAfford (TRUE) and custAge (40), tblEligibility returns "eligible".

```
tblEligibility: {  
  inputs: ['bureauRiskCat', 'bureauAfford', 'custAge'],  
  outputs: ['eligibility'],  
  outputValues: ['ineligible', 'eligible'],  
  rules: [  
    ['decline', '-', '-', 'ineligible'],  
    ['-', false, '-', 'ineligible'],  
    ['-', '-', '<18', 'ineligible'],  
    ['-', '-', '-', 'eligible']],  
  hitPolicy: 'priority'  
},
```


Finally, given the input parameters "eligible" and "full", tblStrategy returns "bureau",

```
tblStrategy: {
  inputs: ['eligibility','bureauCallType'],
  outputs: ['strategy'],
  rules: [
    ['ineligible', '-', 'decline'],
    ['eligible', 'full,mini', 'bureau'],
    ['eligible', 'none', 'through']],
  hitPolicy: 'unique'
},
```

If the result of the Decision Table tblStrategy in cell J26 would have been either "decline" or "through", then our work would be finished. We would either decline or accept the applicant, respectively. However, since "bureau" was returned, more analysis is required and on we must go.

9. The next formula,

```
postBureauRiskCat: { formula: "tblPostBureauRiskCat(,,custExist,
appRiskScore, creditScore)" },
```

returns a Post Bureau Risk Category based on if the applicant is an existing customer, his/her application Risk Score and credit score.

```
tblPostBureauRiskCat: {
inputs: ['custExist','appRiskScore','creditScore'], outputs:
['postBureauRiskCat'],
  rules: [
    [false, '<120', '<590', 'high'],
    [false, '<120', '[590..610]', 'medium'],
    [false, '<120', '>610', 'low'],
    [false, '[120..130]', '<600', 'high'],
    [false, '[120..130]', '[600..625]', 'medium'],
    [false, '[120..130]', '>625', 'low'],
    [false, '>130', '-', 'veryLow'],
    [true, '<=100', '<580', 'high'],
    [true, '<=100', '[580..600]', 'medium'],
    [true, '<=100', '>600', 'low'],
    [true, '>100', '<590', 'high'],
    [true, '>100', '[590..615]', 'medium'],
    [true, '>100', '>615', 'low']],
  hitPolicy: 'unique'
},
```

- The input parameter custExist is given in the data section as "False".
- The input parameter appRiskScore was calculated in 5, as 104.
- The input parameter creditScore = 610 as given in the data section.

Given these input values, 1 rule is "hit", rule 2 and returns "medium".

10. The next formula inserts the calculation of a decision table into an IF statement.

```
postBureauAfford: { formula: "IF(disposIncome *
tblCreditContFactor(,,postBureauRiskCat) > monthInstall, true,
false)" },
```

Let's take the first term;

```
=(disposIncome * tblCreditContFactor(,,PostBureauRiskCat)
```

The first variable, `disposIncome`, was calculated earlier as 963.18. The decision table, `tblCreditContFactor()` returns 0.7 given the input parameter `postBureauRiskCat = medium`. Therefore, the first term evaluates to: 674.22.

The next variable, `monthInstall`, was calculated earlier as 556.82. Therefore, this IF statement evaluates to TRUE.

11. Finally, we come to the final decision table, `tblRouting`.

```
routing: { formula: "tblRouting(,,postBureauRiskCat,
postBureauAfford, bankrupt, creditScore)", finalValue: [] }
```

This table returns a final strategy based on two decision table results (`postBureauRiskCat` and `postBureauAfford`) and two input parameters, `bankrupt` and `creditScore`. The input parameters are: `PostBureauRiskCat = medium`, `postBureauAfford = True`, `bankrupt = FALSE` (never bankrupt) and `creditScore = 610`.

```
tblRouting: {
  inputs:
  ['postBureauRiskCat', 'postBureauAfford', 'bankrupt', 'creditScore'],
  outputs: ['routing'],
  inputValues: [
    [null, null, true, '[0..999]'],
    [null, null, false, null]],
  outputValues: ['decline', 'refer', 'accept'],
  rules: [
    ['-', false, '-', '-', 'decline'],
    ['-', '-', true, '-', 'decline'],
    ['high', '-', '-', '-', 'refer'],
    ['-', '-', '-', '<580', 'refer'],
    ['-', '-', '-', '-', 'accept']],
  hitPolicy: 'priority'
},
```

With these criteria, the last rule, rule 5, is successful, the loan application should be accepted.

```
{
  "status": {
    "code": 0,
    "id": "2590+DTLoanStrategyFromScratch+2020-02-28-17-46-35-419297",
    "codeText": "Solver has completed the calculation."
  },
  "observations": {
    "strategy": {
      "value": "bureau"
    },
    "routing": {
      "value": "accept"
    }
  }
}
```

Using a PMML

The example model, `DT Loan Strategy Model + Predictive.json`, extends the `DT Loan Strategy Model` to accept a data science model saved in PMML format, as an argument to a decision table. The

formula for appRiskScore returns the minimum of 2 outputs. The first output is produced from a linear regression model scoring new data contained in Input!F26:I27. The second output is from the decision table, tblAppRiskScore given the inputs: custAge, maritalStatus, employmentStatus.

The decisionTables section and the remaining data section remains the same and thus does not appear here.

Note the existence of the full PMML model for "Predictive_PMML_Model". When RASON Decision Services calculates prediction or classification results, internal values and coefficients are generated and used in the computations.

Analytic Solver saves these values to an additional output sheet, termed the Stored Model Sheet, which uses the output sheet name, XX_Stored_N where XX are the initials of the classification or prediction method and N is the number of generated stored sheets. The actual PMML model is saved to the worksheet beginning with cell B12 and ending with the last populated cell in the B column. (The contents of the last cell will be: "</PMML>".) For more information on how to save a data science or forecasting model in PMML format, see the Data Science User Guide for Analytic Solver.

XLMiner SDK saves these values to an XML file. For more information on XLMiner SDK, see the XLMiner SDK User Guide.

A PMML model can be passed in two ways to the RASON model, inline or through an external data source.

Passing PMML Model In-Line

In order to enter the PMML model in line to the RASON model, you'll need to copy the contents of each of these cells, surround them by "", and separate them by commas. In formulas, the PsiPredict() (or PsiForecast()/PsiPosteriors()/PsiTransform()), function can be called to score new data using this PMML model.

In the example code below, cells B12 through B14 contained the following contents.

B12 = <?xml version='1.0' encoding='utf-8'?>

B13 = <PMML version='4.2' xsi:schemaLocation='http://www.dmg.org/PMML-4_2 http://www.dmg.org/v4-2/pmml-4-2.xsd' xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance' xmlns='http://www.dmg.org/PMML-4_2'>

B14 = <Header copyright='Copyright (c) 2021 Frontline Systems Inc.' description='RegressionModel'>

Notice that the contents of these cells appear in the definition of Predictive_PMML_Model!B12:B44 surrounded by quotes and separated by commas.

```
data: {
  "Predictive_PMML_Model": { value: ["<?xml version='1.0'
encoding='utf-8'?>", "<PMML version='4.2'
xsi:schemaLocation='http://www.dmg.org/PMML-4_2
http://www.dmg.org/v4-2/pmml-4-2.xsd'
xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
xmlns='http://www.dmg.org/PMML-4_2'>", "<Header copyright='Copyright
(c) 2021 Frontline Systems Inc.' description='RegressionModel'>",
"<Application name='XLMinerSDK' version='21.0.0.0'/>",
"<Timestamp>2019-1-12 21:32:7</Timestamp>", "</Header>",
"<DataDictionary numberOfFields='5'>", "<DataField
optype='continuous' dataType='double' name='X1'/>", "<DataField
optype='continuous' dataType='double' name='X2'/>", "<DataField
optype='continuous' dataType='double' name='X3'/>", "<DataField
optype='continuous' dataType='double' name='X4'/>", "<DataField
optype='continuous' dataType='double' name='Score'/>",
```

```

"</DataDictionary>", "<RegressionModel modelName='RegressionModel'
functionName='regression' algorithmName='LinearRegression'>",
"<MiningSchema>", "<MiningField name='Score'
usageType='predicted'/>", "<MiningField name='X1'
usageType='active'/>", "<MiningField name='X2'
usageType='active'/>", "<MiningField name='X3'
usageType='active'/>", "<MiningField name='X4'
usageType='active'/>", "</MiningSchema>", "<Output>", "<OutputField
optype='continuous' dataType='double' name='Predicted_Score'
feature='predictedValue'/>", "</Output>", "<LocalTransformations/>",
"<RegressionTable intercept='0'>", "<NumericPredictor name='X1'
exponent='1' coefficient='-2.3617607737378745'/>",
"<NumericPredictor name='X2' exponent='1'
coefficient='1.4606586382766593'/>", "<NumericPredictor name='X3'
exponent='1' coefficient='-1.0552806547103726'/>",
"<NumericPredictor name='X4' exponent='1'
coefficient='0.090913941664884437'/>", "</RegressionTable>",
"</RegressionModel>", "</PMML>"] },
    "predictiveInput": { value: [["X1", "X2", "X3", "X4"], [7, 26,
6,
    60]] },
    "custAge": { value: 40, comment: "custage" },
    "maritalStatus": { value: "s", comment: "maritalst" },
    "employmentStatus": { value: "selfEmployed", comment:
"employst" },
    "custExist": { value: false, comment: "custexist" },
    "monthIncome": { value: 2500, comment: "monthincome" },
    "loanRate": { value: 5, comment: "loanrate" },
    "loanTerm": { value: 30, comment: "loanterm" },
    "loanAmnt": { value: 100000, comment: "loanamnt" },
    "monthExpenses": { value: 1000, comment: "monthexpenses" },
    "loanType": { value: "standard", comment: "loantype" }
    },
    formulas: {
    "appRiskScore": { formula:
"MIN(PsiPredict(Predictive_PMML_Model,predictiveInput),tblAppRiskSco
re(,custAge, maritalStatus, employmentStatus))", comment:
"tblAppRiskScore", finalValue: [] },
    ...
    }
}

```

Importing PMML Model through External Data File

Alternatively, the PMML model may be contained within an CSV or XML file and imported into the RASON model within datasources, as shown in the example code below .

```

datasources: {
    "pmml_src": { type: "xml", connection: "pmml_src.xml", content:
"pmml-
    model", direction:"import" }
},

```

where

- Type: Passes the file type, which can be either "csv" or "xml".
- Connection: Passes the name for the external data file.
- Content: Passes the content type of the XML model, 'pmml-model', 'xml-model' or 'json-model'.

- Direction: Specifies the contents of the file will be imported.

***PsiPredict()* Function**

The PsiPredict() function predicts the response, target, output or dependent variable for Input_Data whether it is continuous (Regression) or categorical (Classification) when the model is stored in PMML format. In addition, this function also computes the fitted values for a Time Series model when the model is stored in PMML format.

`PsiPredict(Model, Input_Data)`

Where Model is an array containing the stored Classification, Regression or TimeSeries model in PMML format and Input_Data is a range containing the new data for computing predictions. This range must contain a header row with column names and at least one row of data containing the exact same features (or columns) as the data used to create the model.

The output of this function is a single column array containing the header and predicted/fitted values for each record in Input_Data.

Analytic Solver supports four scoring functions: **PsiPredict()**, **PsiForecast()**, **PsiTransform()** and **PsiPosteriors()**. PsiPredict() and PsiForecast() provide functionality such as storing and scoring prediction/forecasting models, including ensemble methods with any available weak learner, and computing the fitted values for new time series data. PsiTransform() and PsiPosteriors() provide functionality for storing or scoring. All four functions take two arguments, the model saved in PMML format, and the new input data.

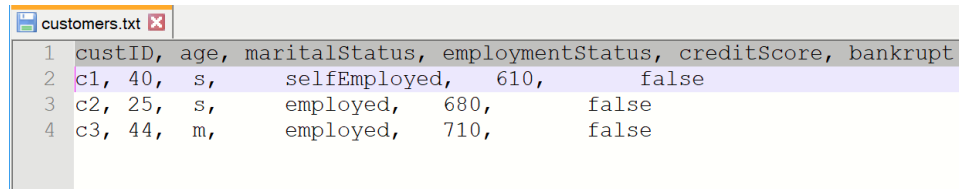
PSI Scoring Function	Description
PsiPredict()	Predicts the target for input data using a Classification or Regression model and computes the fitted values for a Time Series model stored in PMML format.
PsiForecast()	Computes the forecasts for the input data using a Time Series model stored in PMML format.
PsiPosteriors()	Computes the posterior probabilities for the input data using a Classification model stored in PMML format.
PsiTransform()	Transforms the input data using a Transformation model stored in PMML format.

See Appendix II in the RASON Reference Guide for formula signatures of each of these four functions.

Parametric Selection Feature

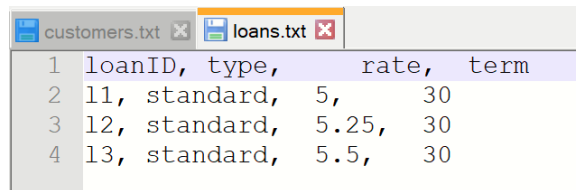
In the previous example, all data used in the model was simply passed as constants in the file. The example model DT Loan Strategy Model and Datasource.json, demonstrates how to import this same data from an external data file using a parametric selection criteria. A Parametric Selection allows a single record to be selected from an external datasource file as an input. *Parametric selection in data-sources is universal, it is critical to decision tables, which expect a single record for their inputs. All supported data types may be used with this feature.*

Most of the customer and loan data is imported from the two datasources: loan_data and cust_data. The datasources section creates two datasources, cust_data and loan_data. The datasource, cust_data, binds to the customers.txt CSV file. This file contains five input parameters, age, maritalStatus, employmentStatus, creditScore and bankrupt. A screenshot of this file is shown below.



	custID	age	maritalStatus	employmentStatus	creditScore	bankrupt
1	c1	40	s	selfEmployed	610	false
2	c2	25	s	employed	680	false
3	c3	44	m	employed	710	false

The loan_data datasource binds to the loans.txt CSV file. This file contains three input parameters: type, rate and term.



	loanID	type	rate	term
1	l1	standard	5	30
2	l2	standard	5.25	30
3	l3	standard	5.5	30

```
{
  modelName: "DTLoanStrategyExamplewDatasource",
  modelDescription: "Loan Strategy model",
  modelType: "calculation",
  datasources :{
    cust_data: {
      type: "csv",
      connection: "customers.txt",
      selection: "custID = ?",
      parameters: {
        cuID: {
          binding: 'get',
          value: 'c1'
        }
      }
    },
    indexCols: ['cust ID'],
    valueCols: ['age' 'maritalStatus', 'employmentStatus',
      'creditScore', 'bankrupt'] ,
    direction: "import"
  },
}
```

Inside of the cust_data datasource, we see the connection argument passing the CSV file, connection: "customers.txt" (screenshot above). The selection argument selects the "CustID" column , from the customers.txt file, and replaces "custID = ?" with "custID = cuID". The "binding" property allows the cuID parameter to be queried outside of the RASON Model

environment (see example below); 'c1' is the default custID in this example; if a query parameter is used outside of the RASON model environment, this value will be replaced by the query parameter.

In addition, indexCols is set to "cust ID" and valueCols are set to 'age', 'maritalStatus', 'employmentStatus', 'creditScore' and 'bankrupt'. This means that cust ID is the index column and 'age', 'maritalStatus', 'employmentStatus', 'creditScore' and 'bankrupt' are the value columns.

```
loan_data: {
  type: "csv",
  connection: "loans.txt",
  selection: "loanID = ?",
  parameters: {
    loID: {
      binding: 'get',
      value: '11' }
  },
  indexCols: ['loanID'],
  valueCols: ['type', 'rate', 'term'],
  direction: "import"
},
```

Inside of the loan_data datasource, we see the connection argument passing the CSV file, connection: "loans.txt" (screenshot above). The selection argument selects the "loanID" column, from the loans.txt file, and replaces "loanID = ?" with "loanID = loID". In this example, the "binding" property allows the loanID parameter to be queried outside of the RASON Model environment (see example below); '11' is the default loanID in this example; if a query parameter is used outside of the RASON model environment, this value will be replaced by the query parameter. In addition, indexCols is set to "loanID" and valueCols are set to 'type', 'rate' and 'term'. This means that loanID is the index column and 'type', 'rate' and 'term' are the value columns.

If we wanted to use multiple selection criteria, such as cust ID, maritalStatus and age, we would change the code to the following:

```
datasources :
{
  cust_data: {
    type: "csv",
    connection: "customers.txt",
    selection: "custID = ? and maritalStatus=? or age=?",
    parameters: {
      cuID: { binding: 'get', value: 'c1' },
      marry: {binding: 'get', value: 's' },
      yrold: {binding: 'get', value: '44' }
    },
    indexCols: ['custID'], valueCols: ['age', 'maritalStatus',
    'employmentStatus', 'creditScore', 'bankrupt'],
    direction: "import"
  },
},
```

The selection component indicates which column or columns in the database/table are to be queried. There is no limit on the amount of parameters queried using "and" or "or".

In this example, the "binding" property allows cuID, marry and yrold parameters to be queried outside of the RASON Model environment (see example below); 'c1' is the default custID in this example; 's' is the default maritalStatus and '44' is the default 'age'; if a query parameter is used outside of the RASON model environment, this value will be replaced by the query parameter.

Parameters may be matched to the selection arguments "by order" or "by name". In the example code above, the parameters are matched to the selection arguments "by order". This means that the order in

which the selection arguments are given must match the order of the parameters i.e. the selection argument order in the example above is custID, maritalStatus, age; therefore, parameters must be listed in the same order: cuID, marry and yrold. If matching by name, the order is not relevant. An example of how to match by name is below.

```
selection: "custID=$cuID and maritalStatus=$marry or age=$yrold",
```

Here, "custID" is matched with "cuID", "maritalStatus" is matched with "marry" and "age" is matched with "yrold" because they have been explicitly matched.

Note: A parameter must have a different name from the column name in the table to which it refers. In the example above, we use "cuID" as a parameter to distinguish it from the column name "custID" because "custID" is a column name marked as an index column. Index columns are index set objects used in the model to dimension variables. They cannot be mixed with parameters. The parameter "cuID" is a single value, an element of a table column used to select records. However, "custID" is the set of all possible elements of a column.

- o indexCols indexes the data first by the custID column
- o valueCols defines the value columns age, maritalStatus, employmentStatus, creditScore and bankrupt.

To perform a query using custID, maritalStatus and age outside of the RASON model environment, use the query parameter:

```
$.get(https://rason.net/api/decision?custID=c2&maritalStatus=s&age=40.....
```

Or in general,

```
$.get(https://rason.net/api/decision?par1=val1&par2=val2.....
```

The RASON Server will map "custID=?" with "custID=c2", "maritalStatus=?" with "maritalStatus=s" and "age=?" with "age=44".

The remaining data is passed in the data section. The input data custExist is passed as a constant within the RASON model.

```
data: {
  comment: "use binding to feed dif. values",
  custExist: { value: false },
  custAge: {
    value: 40,
    binding: 'cust_data',
    valueCol: 'age'
  },
  maritalStatus: {
    value: 's',
    binding: 'cust_data',
    valueCol: 'maritalStatus'
  },
  employmentStatus: {
    value: 'selfEmployed',
    binding: 'cust_data',
    valueCol: 'employmentStatus'
  },
  creditScore: {
    value: 610,
    binding: 'cust_data',
    valueCol: 'creditScore'
  },
  bankrupt: {
```



```

        value: false,
        binding: 'cust_data',
        valueCol: 'bankrupt'
    },
    monthIncome: {
        value: 2500,
        binding: 'get'
    },
    monthExpenses: {
        value: 1000,
        binding: 'get'
    },
    loanType: {
        value: 'standard',
        binding: 'loan_data',
        valueCol: 'type'
    },
    loanRate: {
        value: 5.0,
        binding: 'loan_data',
        valueCol: 'rate'
    },
    loanTerm: {
        value: 30,
        binding: 'loan_data',
        valueCol: 'term'
    },
    loanAmnt: {
        value: 100000.0,
        binding: 'get'
    }
}

```

The decision table results return the recommended loan strategy for customer 1.

```

{
  "status": {
    "code": 0,
    "id": "2590+DTLoanStrategyExamplewDatasource+2020-03-02-18-43-
04-186842",
    "codeText": "Solver has completed the calculation."
  }
  "observations": {
    "strategy": {
      "value": "bureau"
    },
    "routing": {
      "value": "accept"
    }
  }
}

```

Changing Table Components Outside of the RASON Model

Input data monthIncome, monthExpenses and loanAmnt are passed within the RASON model as "get" only. This property allows write access to the data outside of the model environment using the keyword "get". For example, let's say we wanted to increase the value for monthIncome but we did

not want to do so within the RASON model. Rather we could pass this new parameter in the call to the "decision" endpoint.

```
$.get (https://rason.net/api/decision?monthIncome=3000...
```

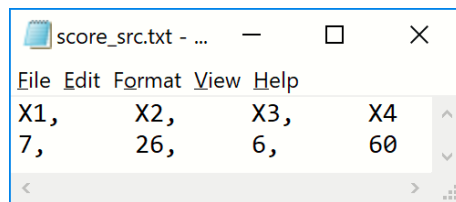
Note: Changing a decision table component outside of the RASON model is not supported.

Using PMML Model to Score New Data

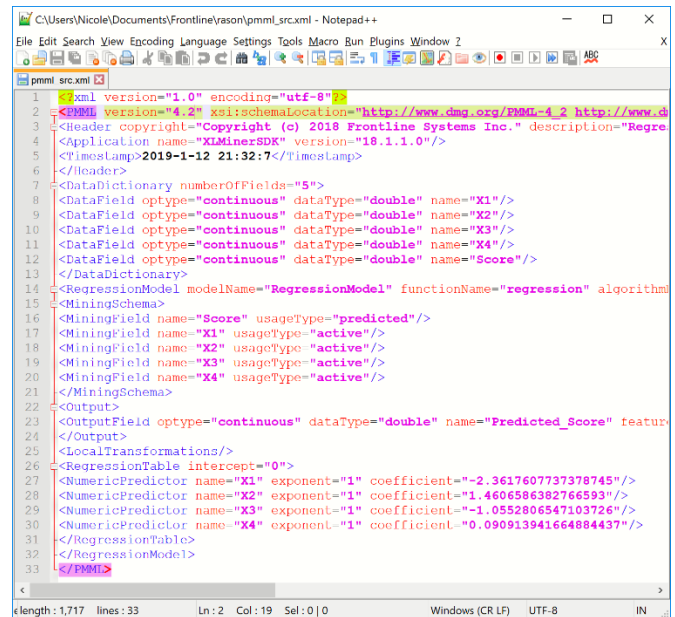
Open the example model, DT Loan Strategy model and Predictive CSV-XML.json, on the Editor page. This model imports a prediction model saved in PMML format and then uses that model to score new data contained in an external data file.

The RASON model begins with the `dataSources` section. The first datasource, `score_src`, holds the new data. See a screenshot of the data file below. The second datasource, `pmml_src`, contains the saved prediction model in PMML format.

Click Download RASON Example Data (beneath RASON Examples) to download both files. Before posting this model to the RASON Server, you'll need to first click Choose Files and select both `score_src` and `pmml_src` to upload to the RASON Server.



X1,	X2,	X3,	X4
7,	26,	6,	60



```
1 <?xml version="1.0" encoding="utf-8"?>
2 <PMML version="4.2" xsi:schemaLocation="http://www.dmg.org/PMML-4_2 http://www.d
3 <Header copyright="Copyright (c) 2018 Frontline Systems Inc." description="Regre
4 <Application name="XLMinerSDK" version="18.1.1.0"/>
5 <Timestamp>2019-1-12 21:32:7</Timestamp>
6 </Header>
7 <DataDictionary numberOfFields="5">
8 <DataField optype="continuous" dataType="double" name="X1"/>
9 <DataField optype="continuous" dataType="double" name="X2"/>
10 <DataField optype="continuous" dataType="double" name="X3"/>
11 <DataField optype="continuous" dataType="double" name="X4"/>
12 <DataField optype="continuous" dataType="double" name="Score"/>
13 </DataDictionary>
14 <RegressionModel modelName="RegressionModel" functionName="regression" algorithm
15 <MiningSchema>
16 <MiningField name="Score" usageType="predicted"/>
17 <MiningField name="X1" usageType="active"/>
18 <MiningField name="X2" usageType="active"/>
19 <MiningField name="X3" usageType="active"/>
20 <MiningField name="X4" usageType="active"/>
21 </MiningSchema>
22 <Output>
23 <OutputField optype="continuous" dataType="double" name="Predicted_Score" featur
24 </Output>
25 </LocalTransformations>
26 <RegressionTable intercept="0">
27 <NumericPredictor name="X1" exponent="1" coefficient="-2.3617607737378745"/>
28 <NumericPredictor name="X2" exponent="1" coefficient="1.4606586382766593"/>
29 <NumericPredictor name="X3" exponent="1" coefficient="-1.0552806547103726"/>
30 <NumericPredictor name="X4" exponent="1" coefficient="0.090913941664884437"/>
31 </RegressionTable>
32 </RegressionModel>
33 </PMML>
```

```
{
  modelName: "DTLoanStrategyPredictiveCSVXML",
  modelType: "calculation",
  datasources: {
    "score_src": {
      type: "csv",
      connection: "score_src.txt",
      colIndex: "score_h",
      direction: "import"
    },
    "pmml_src": {
      type: "xml",
      connection: "pmml_src.xml",
      content: "pmml- model",
      direction: "import"
    }
  },
}
```

The `score_src` data source, which holds the data to be scored, passes three arguments: `type`, `connection` and `colIndex`.

- `type` specifies the file type, in this instance "csv", but any supported file type may be used including XML and JSON.
- `connection` passes the name of the external data source.
- `colIndex` passes the column headings in the external data source. Note: When using a PMML model to score new data, headings must be used in order to correctly map, or match, the variables in the new data with the variables in the saved PMML model.
- `direction` specifies that the data will be imported.

The `pmml_src` data source, which contains the saved prediction model in PMML format, also passes three arguments: `type`, `connection` and `content`.

- `type` specifies the file type, which can be "xml", "json", or "csv".
- `connection` passes the name of the external data source.
- `content` passes the content type within the data file which can be either 'pmml-model', 'xml-model' or 'json-model'. See the RASON Reference Guide for more information on supported XML/JSON formats.
- `direction` specifies that the pmml model will be imported.

The `decisionTables` section remains the same, so it is not displayed here. See the example above for more information on this section.

In the data section we use the "binding" property to bind the two data sources, `score_src` and `pmml_src`, to `scoreData` and `pmodel`, respectively.

```
data: {
  "scoreData": { binding: "score_src" },
  "pmodel": { binding: "pmml_src", comment: "pmodel" },
... (See the example above for the complete contents of
this section)...
}
```

Both `scoreData` and `pmodel` are used as inputs to the `PsiPredict()` function within formulas. For more information on this function see the example above or Appendix II in the RASON Reference Guide.

```
formulas: {
  "appRiskScore": { formula: "MIN(PsiPredict(pmodel,
scoreData),tblAppRiskScore(, , custAge, maritalStatus,
employmentStatus))", comment: "tblAppRiskScore",
finalValue: [] },
...(See the example above for the complete contents of
this section)...
}
```

To solve on the Editor page:

- Click `Post rason.net/api/model` to post the model to the RASON Server.
- Click `POST rason.net/api/model/id/decision` (or `/solve`).

- Click GET rason.net/api/model/id/status to check the status of the solve.
- Click GET rason.net/api/model/id/result to obtain the result.

```
Getting model results: GET
https://rason.net/api/model/2590+DTLoanStrategyPredictive
CSVXML+2020-01-20-03-28-59-759652/result
{ "status": {
  "code": 0,
  "id": "2590+DTLoanStrategyPredictiveCSVXML+2020-01-
20-03-28-59-759652",
  "codeText": "Solver has completed the calculation."
},
  "observations": {
    "appRiskScore": { "value": 20.568 },
    "bureauRiskCat": { "value": "high" },
    "bureauCallType": { "value": "full" },
    "eligibility": { "value": "eligible" },
    "strategy": { "value": "bureau" },
    "creditContFactor": { "value": 0.6 }
  }
}
```

Merging Decision Table Results

The DT Loan Recommend Model.json contains one decision table, tblLoanTypes and one "regular" table, bankRates. The decision table returns a loan or loans meeting four input parameters, reqObj (repayment type), reqType (loan type), conforming (conventional or conforming) and reqDown (percentage down) and returns three outputs, loanType (type of loan), confType (conventional or jumbo) and downPct (down payment percentage required).

The refTypes component requires the conforming input parameter to be Boolean (True or False), the reqDown input parameter to be a number and specifies that the downPct output parameter will return a number.

The inputValues component allows only 1 type of input, the text string "payment". The outputValues component allows only two types of text strings, "conventional" or "jumbo". Notice two arrays were utilized to enter two different strings for the same output. Notice the difference between refTypes and inputValues. The first component, refTypes, specifies the input type, Boolean or number, while inputValues and outputValues specifies the input and output parameter domains, "payment" and "conventional" or "jumbo", respectively.

The hit policy, C for Collect will return all results in the result collection.

The non-table array, bankRates, lists six banks offering loans in the area. Notice that no bank is offering the Fixed 40 loan. The remaining input parameters in the data section are defined constant values or text strings, reqObj = "payment", loanType = "fixed 30", conforming = True and downPct = 30.

Note that the "reqObj" data object is not named "objective" so as to not clash with the defined RASON property, "objective".

```
{
  "modelName": "DTLoanRecommendExample",
```

```

    "modelDescription": "This model has been created from an Excel
model
    in the workbook DT Loan Recomend model.xlsx",
    "modelType": "calculation",
    decisionTables: {
    "tblLoanTypes": {
        inputs: ["reqObj", "reqType", "conforming", "reqDown"],
        outputs: ["loanType", "confType", "downPct"],
        refTypes: ["", "", "boolean", "number", "", "", "number"],
        inputValues: [["payment", "", "", ""]],
        outputValues: [[["", "conventional", ""], ["", "jumbo", ""]],
        rules: [
            ["payment", "Fixed 30, Fixed 20", "-", "<20.0", "ARM 3/1",
            "conventional", 20],
            ["payment", "Fixed 30, Fixed 20", false, ">=20", "ARM 3/1",
            "jumbo", "reqDown"],
            ["payment", "Fixed 30, Fixed 20", true, ">=20", "'ARM 3/1'",
            "conventional", "reqDown"],
            ["payment", "Fixed 15", "-", "<20", "Fixed 20",
            "conventional", 20],
            ["payment", "Fixed 15", false, ">=20", "Fixed 20", "jumbo",
            "reqDown"],
            ["payment", "Fixed 15", true, ">=20", "Fixed 20",
            "conventional", "reqDown"],
            ["payment", "Fixed 20", "-", "<20", "Fixed 30",
            "conventional", 20],
            ["payment", "Fixed 20", false, ">=20", "Fixed 30", "jumbo",
            "reqDown"],
            ["payment", "Fixed 20", true, ">=20", "Fixed 30",
            "conventional", "reqDown"],
            ["payment", "Fixed 30", "-", "<20", "Fixed 40",
            "conventional", 20],
            ["payment", "Fixed 30", false, ">=20", "Fixed 40", "jumbo",
            "reqDown"],
            ["payment", "Fixed 30", true, ">=20", "Fixed 40",
            "conventional", "reqDown"],
            ["payment", "'ARM 7/1','ARM 3/1'", "-", "<20", "ARM 3/1",
            "conventional", 20],
            ["payment", "'ARM 7/1','ARM 3/1'", false, ">=20", "ARM 3/1",
            "jumbo", "reqDown"],
            ["payment", "'ARM 7/1','ARM 3/1'", true, ">=20", "ARM 3/1",
            "conventional", "reqDown"]
        ],
        hitPolicy: "C"
    }
    },
    data: {
        "reqObj": { value: "payment", comment: "reqobj" },
        "loanType": { value: "fixed 30", comment: "reqtype" },
        "conforming": { value: true, comment: "conforming" },
        "downPct": { value: 30, comment: "reqdown" },
        bankRates: { value: [
            ['Lender','loanType', 'confType',
            'minDown%','Term','APR%','Rate%','Points','feesAmt'],
            ['MDL','"ARM 3/1"', 'conventional', 20, 360, 3.002, 3, 0, 0],
            ['AL', "'ARM 3/1'", 'conventional', 20, 360, 3.103, 2.875, 0,
1995],

```

```

      ['AO', "'ARM 3/1'", 'conventional', 20, 360, 3.13, 2.875, 1,
899],
      ['MDL', 'Fixed 30', 'conventional', 20, 360, 3.629, 3.625, 0,
0],
      ['AL', 'Fixed 30', 'conventional', 20, 360, 3.682, 3.625, 0,
1068],
      ['AO', 'Fixed 30', 'conventional', 20, 360, 3.79, 3.75, 0,
799]]
    }
  },
  formulas: {
    loanTypes: { formula: "tblLoanTypes(,True,reqObj, loanType,
conforming, downPct)", comment: "it's important to include the
header", finalValue: [] },
    recom: { formula: "PsiJoin(loanTypes, bankRates, 'loanType =
loanType, confType = confType, downPct >= minDown%')", finalValue:
[] }
  }
}

```

The `formulas` section contains two calculations: `loanTypes` and `recom`. The first, `loanTypes`, passes the input data, found in `data`, `reqObj = payment`, `loanType = "fixed 30"`, `conforming = True` and `downPct = 30` **along with the "ret_header" optional argument set to TRUE**. As discussed earlier, with this argument set to `True`, the RASON Server will return the header for each column in the result collection.

The results for `loanTypes` includes two loans, an ARM loan and a Fixed 40 loan along with the headers for each output: `loanType`, `confType` and `downPct`.

```

"loantypes": { "value":
  [
    ["loanType", "confType", "downPct"],
    ["'ARM 3/1'", "conventional", 30],
    ["Fixed 40", "conventional", 30]
  ]
},

```

The next formula, `recom`, uses the `PsiJoin()` function to join the `loanTypes` result with the `bankRates` array. This function takes three arguments, `Table1`, `Table2` and `Clause`. The `Table1` argument passes the first table, `tblLoanTypes`. The `Table2` argument passes the second table or array, `bankRates`. The third argument, `Clause`, passes the conditions for joining the two. In this example, `loanTypes`, decision table results and the `bankRates` are joined via three columns: `loanType`, `confType` and `downPct`. The `Clause` argument must be surrounded by quotes.

Note that in this example, two conditions are the same, i.e. `loanType = loanType` and `confType = confType`. This is not a requirement; table headers can be different.

The third condition, `downPct >= 'minDown %'` requires that the `downPct` in the Loan Types decision table must be greater than the "minDown %" in the Bank Rates table. In other words, the down payment percentage in the Loan Types table must be greater than the minimum percentage down in the Bank Rates array in order for the loan to be selected and displayed by `PsiJoin()`.

Note: Recall that the decision table formula for `loanTypes` included the optional "header" argument. If the results had not included this argument, it would not have been possible to use the `PsiJoin()` function to complete merge.

Notice how string values, which look like expressions are entered, " 'ARM 3/1' ", using inner single quotes. As discussed earlier, decision table values may be expressions and the way to distinguish expressions from text is by using the inner single quotes. "APR 3/1" will produce an error concerning a misspelled expression. Alternatively " 'ARM 3\1' " would have been a better choice since the back slash is not a valid operator in FEEL.

Another important point to notice is the double quotes surrounding each element in the bankRates table. Although typically there is no need to double quote the values in a non-table array, in this model, since we are joining the tblLoanTypes decision table with the non-table array "bankRates", double quotations in the non-table array must be present.

All three columns are common to both the loanTypes result and the Bank Rates table. Three banks offer a conventional ARM 3/1 loan: Mount Diablo, AimLoan and America One. The first three columns are passed to the Clause argument: loanType, confType and downPct. The remaining columns contain the remaining information for each offered loan: Lender, loanType, confType, minDown%, Term, APR %, Rate %, Points and feesAmt.

```
"recom": {
  "value": [
    ["loanType", "confType", "downPct", "Lender",
     "loanType", "confType", "minDown%", "Term", "APR%",
     "Rate%", "Points", "feesAmt"],
    ["'ARM 3/1'", "conventional", 30, "MDL", "'ARM 3/1'",
     "conventional", 20, 360, 3.002, 3, 0, 0], ["'ARM 3/1'",
     "conventional", 30, "AL", "'ARM 3/1'", "conventional",
     20, 360, 3.103, 2.875, 0, 1995],
    ["'ARM 3/1'", "conventional", 30, "AO", "'ARM 3/1'",
     "conventional", 20, 360, 3.13, 2.875, 1, 899]
  ]
}
```

Decision Table Using DMN Conformance Level 3

In the latest version of RASON Decision Services, DMN/Feel functionality no longer requires Excel formulas when representing a DMN decision model. The entire model may now be represented using only FEEL formulas, which are referred to as **literal expressions**. Such models are entirely independent of Excel syntax. These models are referred to as **pure DMN models**. Notice that pure DMN models can only be decision/calculation models. Currently, Rason Decision Services does not support optimization, simulation or data science models as pure DMN models.

The main consequence of avoiding Excel formulas is to preserve the authentic DMN/Feel types in formula assignments.

For example,

```
dt: { feelFormula: "date('05-05-2021') " }
```

preserves the specific Feel type 'feel date' in the variable **dt**, so we may use in a later feelFormula: "**dt.day**".

Formatting results of box objects as custom types

Decision tables are key box objects in decision modeling. A decision table may have one or more outputs.

- If the result is a single successful rule evaluation, depending on the single or many outputs, we currently return either a scalar value or a horizontal vector.

- If the result is multiple successful rule evaluations (with the Collect hit policy) and the output is single, we return a vertical vector.
- In case of many outputs and many successful rule evaluations, we return a 2D array which resembles a table with columns for each output and records for each success. In any case, the result is either scalar or pure array and we can reference this information in later formulas by the standard rules: scalars as scalars and arrays by whole names or by the index operator.

DMN CL3 introduces a way of formatting the results of decision tables and box functions through custom types. This approach allows users to reference the result more efficiently. Here is an example in which a decision table result is formatted.

```
{
  typeDefs: {
    tParkingFee: {
      language: "FEEL",
      components: ['parkingFee', 'durVal'],
      types: ['number', 'duration']
    }
  },
  decisionTables: {
    "tblParkingFee": {
      inputs: ["dtDuration"],
      outputs: ["durVal", "parkingFee"],
      refTypes: ["duration", "duration", "number"],
      rules: [
        ["<'PT20M'", "duration(dtDuration)", 0],
        ["['PT20M'..'PT1H')", "duration(dtDuration)",
          "2 *ceiling(duration(dtDuration)/duration('PT20M'))"],
        ["['PT1H'..'PT4H')", "duration(dtDuration)", "6
          *ceiling(duration(dtDuration)/duration('PT1H'))"],
        [">='PT4H'", "duration(dtDuration)",
          "30*ceiling(duration(dtDuration)/duration('P1D'))"]],
      hitPolicy: "U",
      resultType: "tParkingFee"
    }
  },
  data: {
    "dur": { value: "PT25M", comment: "period" }
  },
  formulas: {
    "fee": { feelFormula: "tblParkingFee(.,dur)", finalValue: [] },
    "res": { feelFormula: "fee.durVal.minutes", finalValue: [] }
  }
}
```

The decision table (**tblParkingFee**) has two outputs (**durVal** and **parkingFee**) and a hit policy of ‘U’. The result is a single record with two columns. We define a custom component type **tParkingFee** with the same component names as the outputs of the decision table. Then we set the table property **resultType: “tParkingFee”** to that custom type. The results, **durVal** and **parkingFee** are formatted using **refTypes**, "duration" and "number", respectively.

This example uses only feelFormulas in order to preserve the Feel types. The first formula assigns the decision result to the variable “**fee**”, which is a 1D array of 2 elements – the outputs. However, this time “**fee**” has the **tParkingFee** custom type attached to it. Without the custom type, the variable “**fee**” may be referenced only by its name or through the index operator.

With the more flexible custom type and reference, only the desired component in the next formula **fee.durVal** is required. Since feel formulas preserve Feel types, the component **fee.durVal** is of type

duration. The number of minutes that the car was parked can be extracted from the duration:
feeFormula: “fee.durVal.minutes”.

Defining Contexts in RASON

Introduction: Relation to Custom Types

Context objects resemble component custom types (defined in a previous chapter). Recall that component custom types are defined as custom types with components such as names, types and, optionally, allowed values (i.e. domain). If a component type is attached to a variable, then that variable defines values in its array structure according to the component type description. The variable may be referenced component-wise using the `'.'` operator. For example,

```
typeDefs: {
  tPmt: {
    components: {
      payment: { typeRef: 'number' },
      fee: { typeRef: 'number' },
      total: { typeRef: 'number' }
    }
  }
}
data: {
  loan: { type: 'tPmt', value: [600000, 1000, 601000], binding:
'get' }
}
```

Subsequently, `loan.payment`, `loan.fee`, and/or `loan.total` may be referenced within later sections (for example the formulas section) in the RASON model. The restriction here is that the variable must define only constant values. These constant values may be obtained using the binding: `'get'` mechanism or by fetching a record from an external table or even simply including them inline, as shown in the example code above. Regardless of how they are obtained, these values may not be computed. However, in the latest version of RASON Decision Services, the concept of variables with components has been extended to the next level where components may compute formulas. This type of component is referred to as a "context" which is defined within the new "contexts" section.

The Context Definition

A context is a single object which determines the type of the structure and, at the same time, defines the values. Recall that with component types we have two objects – the type and the variable to which we attach the type. This results in an abstract type definition that many different variables may be attached to. However, *context* is a single object encompassing both the type and the variable.

Contexts are defined within the special section **contexts**: { }. Each object has a unique name in the global scope. The context object, "language", defines the formula language using the syntax "language": "Excel" or "language": "FEEL". (Currently, only FEEL and Excel are supported formula types.)

Components are defined by a unique name in the local scope and through component properties.

The **typeRef/type** property must be a supported type in Excel or FEEL; custom types are not allowed. Each component must have either a **value** or **formula** property to define the value attached to it. These

two properties plus the holding mechanism, makes the context type variables distinguishable from the component type variables. See the example below.

```
contexts: {
  cPmt: {
    language: "FEEL",
    components: {
      payment: {
        typeRef: 'number',
        formula: "(loan.principal*loan.rate/12)
/ (1 - (1 + loan.rate/12)**-loan.termMonths)"
      },
      fee: {
        typeRef: 'number',
        value: 10
      },
      total: {
        typeRef: 'number',
        formula: "payment + fee"
      }
    }
  }
}
```

The components “payment”, “fee”, and “total” are in the local scope of the context variable cPmt. Outside that context, the same names can be used either in the global scope or another local scope and they will be distinguishable. Notice that the formulas in the context can reference variables from both local and global scopes.

Defined in this way, the context object is a variable itself and can be used in formulas as a whole – it will be treated as a vertical array with all component values. The context object may also be used component-wise through the ‘.’ operator. For example, **cPmt.total** will return the **total** component.

The component property **typeRef** comes from the DMN syntax, but the Excel property **type** is also supported. Since the basic component types depend on the language, users must pay attention to the assigned type values. For example, if a context uses language: Excel, then the FEEL type “duration” may not be assigned to a component. Frontline encourages the usage of *typeRef* with language: “FEEL” and *type* with language: “Excel”.

Notice that the context object **resembles box functions** without arguments. The difference between the two is that box functions have one more result formula and that is the only value they are able to compute in return. Box functions reference box function components as results. Since contexts have no result or default components, when used without the ‘.’ operator, contexts return all component values in a vertical array.

There are two functions which can be used optionally with Context objects but both are supported only in FEEL, i.e. “language”: “FEEL”. RASON Decision Services has implemented them in order to obtain compliance for the DMN specification.

- **getValue(contextObject, componentName)** is equivalent to **contextObject.componentName**
- **getEntries(contextObject)** is equivalent to simply referencing the **contextObject**

FEEL and DMN RASON Examples

For many more FEEL and DMN RASON examples, visit www.RASON.com, click the Editor tab and then click the *Download RASON example data* icon.

te App ▾

Apps

☰

RASON Examples

☁

Download RASON example data.

Using the REST API

Introduction

This chapter introduces the RASON REST API function calls that may be used to analyze a model, solve an optimization, run a simulation, perform a data science task, calculate a decision table or solve a decision flow using either the Editor Page on www.RASON.com or from within your own application.

For small, simple models that can be solved in limited memory within 30 seconds of CPU time, you can run an optimization, simulation, data science task, decision table recalculation or solve a decision flow and get results with a single API call: POST rason.net/api/optimize for optimization, POST rason.net/api/simulate for simulation, POST rason.net/api/datamine for data science, POST rason.net/api/decision to recalculate a decision table or POST rason.net/api/solve to solve a decision flow (and any problem type) and obtain an ODATA end point for the results. To keep your application response time acceptable to your user, you should use this endpoint judiciously.

For larger models that require more time or memory or to solve a decision flow, you can use separate API calls to (i) upload your model plus data files, create a **resource ID** on the RASON server, (ii) start an optimization, simulation, data science task, decision table or decision flow recalc (iii) check on its status and get progress information, and finally (iv) retrieve results when the solution process is complete. If necessary, you can stop a long run in progress with an API call.

Other API calls allow you to *diagnose* a model to determine its type (linear, quadratic, nonlinear, etc.) and size before solving it (mostly useful in model design and development), retrieve a resource ID that you've previously created (including the full text of the model), update the model text of an existing resource ID, get a list of all the resource IDs you've created, and delete a resource ID.

RASON REST API

This section gives a detailed account of how to use the RASON REST API beginning with how to pass the authorization header (required on each call to the API), then moving on to obtaining and passing the resource ID, what to expect as a response to a call to a REST API endpoint, a discussion on CPU time limits, an explanation on throttling and unauthorized errors and ending with a description of each REST API endpoint.

Authorization Headers

Every RASON REST API call must include an 'Authorization' header with value 'Bearer' followed by your account OAuth token. To obtain your OAuth token, visit www.RASON.com to register for a *free* account, then click the MyAccount link to copy and paste the token into your application. An example of an Authorization Header is below. For more information on creating your OAuth token, please see the earlier chapter, *RASON Services Web IDE*.

```
'Authorization': 'Bearer ' + '{OAuth Token}', 'Content-Type':  
'application/json' },
```

Resource IDs

When you create a resource ID (referred to as Model IDs in RASON V1) with POST `rason.net/api/model`, the response will include a 'Location' header whose value is the resource URI. This resource URL includes the **resource ID**. A resource ID identifies not only **model versions** of RASON models/decision flows (where the text of a RASON model is different) but also **model instances** (where a model/decision flow is run with new data). Each ID contains an embedded timestamp to identify version history and also for auditability purposes, i.e. to identify the model instance used to generate the RASON response.

The response below shows the resource id labeled as "id".

```
"status": {
  "code": 0,
  "id": "2590+TestModel1+2019-11-22-16-48-00-704633",
  "codeText": "Solver found a solution. All constraints and optimality conditions are satisfied."
},
```

Model Names

RASON V2 supports both named and unnamed models. *Note: A decision flow must be named.* A model/decision flow becomes "named" either by including `modelName/flowName: "name"` in the RASON model text and then calling POST `rason.net/api/model` or by simply calling POST `rason.net/api/model/<name>` or both. Either call returns a Location header with a new resource ID that identifies this unique model/flow instance. The "name" must be unique among models with a user's account. An unnamed model, or a model not containing `modelName`, has no name.

- A model/flow may be named by using a REST API call to POST `rason.net/api/model/{name}`, i.e. POST `rason.net/api/model/TestModel2`. See the example response below, note the resource ID (2590+TestModel2+2019-11-22-17-53-20-176350) that identifies the unique model instance.

Location: `https://rason.net/api/model/2590+TestModel2+2019-11-22-17-53-20-176350`

- A model may also be named by using the property `modelName: "name"` in the body of the RASON model. A decision flow may be named in a similar way by using the property `flowName: "name"`. In the example code below, the model name is "TestModel1". The RASON model will be "named" with a call to POST `rason.net/api/model` or POST `rason.net/api/model/{name}`. If using POST `rason.net/api/model/<name>`, the name given to the `modelName` property and the name passed to POST `rason.net/api/model/<name>` *must* be identical, otherwise, an error will be returned.

```
{
  modelName: "TestModel1",
  comment: "Example of using CSV table binding",
  datasources: {
    parts_data: {
      type: "csv",
      connection: "ProductMixParts.txt; header",
      indexCol: 1,
      ...
    }
  }
}
```

See the example response below, again, note the resource ID (2590+TestModel1+2019-11-22-18-25-40-015004) that identifies the unique model instance.

Location: `https://rason.net/api/model/2590+TestModel1+2019-11-22-18-25-40-015004`

The RASON Server maintains a simple, one-level directory of named models as ordinary text files using Azure file storage.

Additionally, a RASON model (standalone or reusable) may be saved to the user's OneDrive for Business account. If the model is stored in OneDrive for Business, the user must give the RASON server permission to access the account on OneDrive by adding a Data Connection on the MyAccount page at www.RASON.com.

Data Connections

Create one or more named Data Connections, and reference them in your RASON model data sources with connection: "NAME=myname".

	Name	Type	URI
<input type="checkbox"/>			

Create New
Edit Selected
Delete Selected

Data Connection Credentials are stored securely in an Azure vault.

For more information on creating a Data Connection, see the *Data Connections* section within the previous *RASON Services Web IDE* chapter. Note: There is a 4 MB limit on the size of files written back to OneDrive or OneDrive for Business.

Model Versions

Every RASON model (named or unnamed) or decision flow that is posted to the RASON Server will have a *resource ID* of the *model instance* that was run (including "quick" or synchronous calls). Although for synchronous calls (such as POST rason.net/api/optimize, POST rason.net/api/simulate, etc.) the resource ID will never be referenced by the RASON client; it only appears in the JSON response and the RASON event log (see below). This new resource ID is now included in the "status" returned for each response as "id": resourceId; for example:

```
"status":{
  "code": 0,
  "id": "11+2019-03-15-00-56-32-664488",
  "codeText": "Solver found a solution. All constraints and
optimality
conditions are satisfied."
},
```

Every model/flow that is updated on the RASON Server (via PUT rason.net/api/model/{nameorid}) will have a new ID for the new version; the old ID will identify the old version. GET rason.net/api/model/id can be used to retrieve the text of the current or any past version. Resources are persisted until explicitly deleted by a user and model text is stored under the resource ID. The result is a simple "version control system". Users can reference any model/flow version desired.

Fitted Models (in RASON DM)

Just as decision flows, optimization, simulation and decision table models can be named, fitted data science models (fittedModels) may also be named. A named fittedModel with filename "fitted-name" is created when a RASON data science model containing name: {..action:fit} is run. "fitted-name" must be unique among models within the user's account. The fittedModel inherits the resource ID of the model instance run to create it; if this RASON model or another one that fits the same name is run again, a new version of fitted-name is created, inheriting the ID of that new model instance. GET rason.net/api/model/nameorid/result/fitted-name is used to retrieve the text of the fittedModel.

Model Instances

When you make an API call that requires a resource ID, you simply substitute the string you get from the Location header for *id*. Afterwards, you can call additional RASON REST API endpoints to place the model/flow in the queue on the Server such as GET

`rason.net/api/model/{id}/solvetype` where `solvetype` can be `solve`, `optimize`, `simulate`, `datamine`, `decision` or `diagnose`. (Use `solvetype=solve` to solve decision flows or any type of problem type: optimization, simulation, data science or decision.) After the model is placed in the queue, you should periodically check the status of the model, using `GET rason.net/api/model/{id}/status`, which will either be "complete" or "incomplete". Once the model/flow has finished solving, or the status returned is "complete", then you can call `GET rason.net/api/model/id/result` to obtain the final solution. The model/flow is merged with the model instance and the RASON log will have only a timestamp to distinguish different runs or model instances – the IDs will be the same.

In RASON Decision Services, when a model/flow is solved via the new REST API endpoint `POST rason.net/api/model/id/solvetype` (`solvetype` can once more be `solve`, `optimize`, `simulate`, `datamine`, `decision` or `diagnose`), where `{id}` is again the ID of the (desired version of) model/flow. The call will return a Location header with a new resource ID that identifies this unique model/flow instance (model bound to data). The client should use this new ID in subsequent calls to `GET rason.net/api/model/id/status`, `GET rason.net/api/model/id/result` etc. and the RASON log will uniquely identify this model/flow instance via the new resource ID.

The timestamp embedded in this ID records the date and time when the model was run, and when any regularly-updated data sources (OneDrive, CDS, CData CloudHub) were accessed by the model instance.

Timestamps and RASON Event Log

RASON Decision Services includes an update for auditability purposes by adding a timestamp to the model instance ID for all stages used to run the decision flow. The property, "id", in the results (obtained from `GET rason.net/api/model/{name}/results`) has been modified to include an expanded status for each stage in the decision flow.

```
"mlr-reusable": {
  "status": {
    "id": "2590+mlr-reusable+2020-07-07-18-40-46-097257",
    "code": 0,
    "codeText": "Success",
    "solveTime": 595
  },
}
```

Note: The reusable model, "mlr-reusable", runs multiple linear regression on the input data. See the Creating and Running a Decision Flow chapter for more information on this reusable model.

Using Model Names in API Calls

As mentioned above, RASON models may be named or unnamed while decision flows must be named. (Unnamed decision flows are not supported.) As a "best practice" it is recommended to always use named models, though unnamed models are still supported (i.e. for synchronous operations). In any RASON API call that creates, updates, deletes, retrieves model text, or executes a resource ID, the model name can be used *instead* of the ID in the URL. The new API call `POST rason.net/api/model/{name}/solvetype` returns an ID for (only) that model/flow instance. Subsequent calls to `GET rason.net/api/model/name/status` or `GET rason.net/api/model/name/result` (etc.) will point to this instance. (To get the result of the original model/flow, you can either replace "name" with the full resource ID of the original model, or make the original model the "champion". For more information on champion models, see the next section.)

The new API endpoint `POST rason.net/api/model/name/solve` solves both single models and decision flows and can be used in place of `POST rason.net/api/solvetype`, `GET rason.net/api/model/nameorid/solvetype` and the new call to `POST rason.net/api/model/nameorid/solvetype`.

Champion and Challenger Models

An API call using a RASON model name or decision flow always refers to the "champion" version or the most recent version of that model or flow. The user can designate a specific version (resource ID) as the current "champion" via the API call `PATCH rason.net/api/model/nameorid` with "champion" in the body, marking that resource ID as the current "champion" (i.e. the model instance to use when future API calls specify the model name; then the user can create newer versions as "challengers"). An API call to `PATCH rason.net/api/model/{nameorid}` with "challenger" in the body, or an empty body if just posting new data files, unmarks the resource ID as the champion.

Similarly, when a RASON model uses a fittedModel for scoring by referencing its name in a datasource declaration, the champion or latest version will be retrieved. The user can also designate a specific fittedModel version as the current "champion" via the API call `PATCH rason.net/api/model/{id}` with the string "champion" in the body, where ID designates the model instance where the relevant fittedModel version was produced; then the user can create newer versions as "challengers". As with RASON models, `PATCH rason.net/api/model/{id}` with the string "challenger" in the body, or an empty body, "unmarks" the fittedModel version as the champion.

POST for Runs: Separating "Fixed" Data from "Run-Specific" Data

When initially creating a resource ID, `POST rason.net/api/model/{name}` allows CSV and XLSX data files to be attached and "streamed up" to the RASON Server. But with RASON 2020 and later, you can now access your data from online data sources such as OneDrive, the Common Data Service and CData Cloud Hub. Only "fixed data" (data that does not change from run to run) should be attached to your model using this call.

With the new API call to `POST rason.net/api/model/nameorid/solvetype` (solvetype = solve, optimize, simulate, datamine, decision or diagnose), the POST body may be empty or it may supply "run-specific" CSV and XLSX files when starting the execution of a new model *instance*. These files will be attached to the new resource ID returned by this call, in the Location header.

Deleting Resource IDs

As in RASON V1, **it is your responsibility to delete resource IDs** when you no longer need them. This is easy to do: just use the API call `DELETE rason.net/api/model/nameorid`. Deleting a resource ID also removes any data files that were uploaded when it was created. For Basic (free) accounts, the RASON Server will delete "aging" resource IDs; for Platform and other accounts, Frontline Systems reserves the right to charge for resource ID storage.

Responses: Solution Status and Results

When you call `GET rason.net/api/model/id/status` to check the solution status, the JSON response format is fixed, and easy to examine. For example, if the solution process has completed, you'll get { "status": "Complete" }, so you can check for this with code such as if (response.status == "Complete"). When you call `GET rason.net/api/model/id/result` to retrieve results, the response includes a fixed portion describing the solution status, and a variable portion that depends on *what you ask for in the RASON model*.

For example, if you run the example optimization model AirlineHub.json (available as AirlineHub.json in the dropdown list on the Editor page at www.RASON.com or at C:\Program Files\Frontline Systems\Solver SDK Platform\Examples\RASON if using the Desktop IDE), you'll see the following response:

```
var modelRequest = {
  "variables" :
  {
    "x" : { "value": 1.0, "finalValue": [] },
```

```

...
"objective" :
{
  "z" : { "type": "minimize", "finalValue": [] }
}
};

```

The fixed part of the response includes the "status" property, which tells you the outcome of the optimization. You can test this with JavaScript code such as: `if (response.status.code == 0)`. The variable part of the response includes final values for the three decision variables x, y and z, and the objective being minimized (which is the third variable, z). Recall from the chapter *Defining Your Model*, we got this part of the response *because we asked for it* in the RASON model - we included the property `finalValue: []` in the definition of the objective and each decision variable.

To retrieve the final value of the objective in your JavaScript code, you would simply reference `response.objective.z.finalValue`. If the model had asked for the dual value of x by including the property `dualValue: []`, your JavaScript code could obtain this value with a reference to `response.variables.x.dualValue`.

CPU Time Limits and Charges

Large optimization and simulation models, or challenging mixed-integer, global optimization, or stochastic optimization models can consume significant amounts of CPU time (especially) and memory. Initially, the RASON service back-end workers run on Azure cloud services with 7GB memory, but we can offer much larger memory service instances, up to 112GB. See the RASON Subscription chapter for a discussion on how the RASON service handles CPU time.

Throttling

The RASON REST API ensures the Server operates at peak performance and continually maintains reliability by utilizing throttling, which limits the number of calls to the REST API. When a throttling limit is reached, the RASON REST API will throttle any further requests from that resource ID for one minute.

Synchronous REST API Endpoints (Quick Solve)

The following six RASON REST API calls may be used to quickly invoke a data science feature (`/datamine`), recalculate a decision table (`/decision`), diagnose an optimization, stochastic optimization or simulation optimization model (`/diagnose`), solve an optimization model (`/optimize`), create an OData endpoint for the results of any type of RASON model (`/solve`) or run a simulation (`/simulate`) as long as the task finishes in *30 seconds or less of CPU time*. Each of these calls are synchronous and will allow you to perform the function (datamine, decision, diagnose, optimize, create ODATA endpoint or simulate) and obtain the results with a single API call. A resource ID is returned by RASON's synchronous endpoints, but this ID will never be referenced by the RASON client. It only appears in the JSON response and the RASON event log.

- [POST rason.net/api/datamine](https://rason.net/api/datamine) – Solves data science/forecasting models
- [POST rason.net/api/decision](https://rason.net/api/decision) - Solves DMN business rules, custom types and box functions.
- [POST rason.net/api/diagnose](https://rason.net/api/diagnose) - Diagnoses conventional/stochastic/simulation optimization and simulation models.
- [POST rason.net/api/optimize](https://rason.net/api/optimize) – Solves conventional optimization models (only).
- [POST rason.net/api/simulate](https://rason.net/api/simulate) – Solves simulation models (only).

- POST rason.net/api/solve – Solves all models types.

REST Endpoint: POST: <https://rason.net/api/datamine>

A REST API endpoint that accepts a RASON model, runs a data science task and returns a result in JSON format. This call should only be used with models that run very quickly, i.e. within a few seconds. There is a time limit of 30 seconds on all synchronous calls.

- **URL**

<https://rason.net/api/datamine>

- **Method:**

POST

- **URL Params**

Required: None

Optional: Any datasource component may be passed as a query parameter if a binding property exists for that datasource. For example, see the RASON example code snippet below.

```
"datasources": {
  "srcCustomers": {
    "type": "csv",
    "connection": "customers_dt.txt",
    "selection": "custID = $parCustID1 or custID = $parCustID2",
    "parameters": {
      "parCustID1": {
        "binding": "get",
        "value": "c1"
      },
      "parCustID2": {
        "binding": "get",
        "value": "c3"
      }
    }
  }
},
```

To query `custID=$parCustID1 or custID=$parCustID2` outside of the RASON model environment, use:

```
$.get('https://rason.net/api/datamine?parCustID1=c1&parCustID2=c2...')
```

Or in general,

```
$.get('https://rason.net/api/datamine?par1=val1&par2=val2.....')
```

RASON Server will map `"custID=$parCustID1"` with `"parCustID1=c1"` and `"custID=$parCustID2"` with `"parCustID2=c3"`. A query can return any number of rows that satisfy the filtering condition, from 0 to infinity.

- **Headers**

Required: Authorization - Example: `Authorization: bearer {your RASON token}`

Optional: None

- **Data Params**

A RASON Data Science Model

Example:

```
{
  modelName: "oneHotEncodingInLineData",
  modelDescription: 'transformation: one-hot encoding',
  modelType: "datamining",
  datasets: {
    trainData: {
      value: [
        [ '1', 2.0 ],
        [ '0', 3.0 ],
        [ '1', 6.0 ]
      ]
    },
    newData: {
      value: [
        [ '0', 3.0 ],
        [ '1', 6.0 ],
        [ '1', 6.0 ]
      ]
    }
  },
  estimator: {
    myEncoder: {
      type: 'transformation',
      algorithm: 'oneHotEncoding'
    }
  },
  actions: {
    encoderModel: {
      trainData: 'trainData',
      estimator: 'myEncoder',
      action: 'fit'
    },
    encodedTrainData: {
      data: 'trainData',
      fittedModel: 'encoderModel',
      action: 'transform',
      evaluations: [
        'transformation'
      ]
    },
    encodedNewData: {
      data: 'newData',
      fittedModel: 'encoderModel',
      action: 'transform',
      evaluations: [
        'transformation'
      ]
    }
  }
}
```

- **Success Response:**

Code: 200 (OK)

Example Response:

```
{
  "status": {
    "id": "2590+2019-11-18-20-23-58-998833",
    "code": 0,
    "codeText": "Success"
  },
  "results": [
    "encodedTrainData.transformation",
    "encodedNewData.transformation"
  ],
  "encodedTrainData": {
    "transformation": {
      "objectType": "dataFrame",
      "name": "Encoded - traindata",
      "order": "col",
      "rowNames": [
        "Record 1",
        "Record 2",
        "Record 3"
      ],
      "colNames": [
        "Feature 1_0",
        "Feature 1_1",
        "Feature 2_2",
        "Feature 2_3",
        "Feature 2_6"
      ],
      "colTypes": [
        "integer",
        "integer",
        "integer",
        "integer",
        "integer"
      ],
      "indexCols": null,
      "data": [
        [
          0,
          1,
          0
        ],
        [
          1,
          0,
          1
        ],
        [
          1,
          0,
          0
        ],
        [
          0,
          1,
          0
        ],
        [
          0,
          0,
          1
        ]
      ]
    }
  },
}
```

```

"encodingNewData": {
  "transformation": {
    "objectType": "dataFrame",
    "name": "Encoded - newdata",
    "order": "col",
    "rowNames": [
      "Record 1",
      "Record 2",
      "Record 3"
    ],
    "colNames": [
      "Feature 1_0",
      "Feature 1_1",
      "Feature 2_2",
      "Feature 2_3",
      "Feature 2_6"
    ],
    "colTypes": [
      "integer",
      "integer",
      "integer",
      "integer",
      "integer"
    ],
    "indexCols": null,
    "data": [
      [
        1,
        0,
        0
      ],
      [
        0,
        1,
        1
      ],
      [
        0,
        0,
        0
      ],
      [
        1,
        0,
        0
      ],
      [
        0,
        1,
        1
      ]
    ]
  }
}

```

Error Response:

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.

Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.
--	--

REST Endpoint: POST: *https://rason.net/api/decision*

A REST API endpoint that accepts a RASON decision table, calculates the decision table and returns a result in JSON format. This call should only be used with models that run very quickly, i.e. within a few seconds. There is a time limit of 30 seconds on all synchronous calls.

- **URL**

`https://rason.net/api/decision`

- **Method:**

POST

- **URL Params**

Required: None

Optional: Any data component may be passed as a query parameter if a binding property exists for that data component. See the previous topic "Parametric Selection Feature" for a complete example illustrating how to pass a query parameter from outside or inside your RASON model.

- **Headers**

Required: Authorization - Example: `Authorization: bearer {your RASON token}`

Optional: None

- **Data Params**

A RASON Decision Table Model

```
{
  "modelName": "DTUniqueExample",
  "modelDescription": "'U' policy example",
  "modelType": "calculation",
  "data": {
    "comment": "use binding to feed dif. values",
    "age": {
      "value": 54
    },
    "medHistory": {
      "value": "good"
    }
  },
  "decisionTables": {
    "tblRisk": {
      "inputs": [
        "age",
        "medHistory"
      ],
      "outputs": [
        "riskRating",

```

```

        "rule"
      ],
      "rules": [
        [
          ">60,<25",
          "good",
          "Medium",
          "r1"
        ],
        [
          ">60",
          "bad",
          "High",
          "r2"
        ],
        [
          "[25..60]",
          "-",
          "Medium",
          "r3"
        ],
        [
          "<25",
          "bad",
          "Medium",
          "r4"
        ]
      ],
      "hitPolicy": "Unique",
      "default": [
        [
          "High",
          "r0"
        ]
      ]
    }
  },
  "formulas": {
    "res": {
      "formula": "tblRisk(,,age, medHistory)",
      "finalValue": []
    }
  }
}

```

- **Success Response:**

Code: 200 (OK)

Example Response:

```

{
  "status": {
    "code": 0,
    "id": "2590+2019-11-18-21-57-42-631488",
    "codeText": "Solver has completed the calculation."
  },
  "observations": {
    "res": {

```

```

    "value": [
      [
        "Medium",
        "r3"
      ]
    ]
  }
}

```

Error Response:

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

REST Endpoint: POST: <https://rason.net/api/diagnose>

A REST API endpoint that accepts an optimization, stochastic optimization or simulation optimization RASON model, runs a model diagnosis and returns a result in JSON format. This call should only be used with models where the model diagnosis completes very quickly, i.e. within a few seconds. There is a time limit of 30 seconds on all synchronous calls.

- **URL**

`https://rason.net/api/diagnose`

- **Method:**

POST

- **URL Params**

Required: None

Optional: None

- **Headers**

Required: Authorization - Example: Authorization: bearer {your RASON token}

Optional: None

- **Data Params**

An optimization, stochastic optimization or simulation optimization RASON Model.

```
{
  modelName: "ProductMix",
  modelDescription: "Version of ProductMix using a 2-
dimensional
matrix",
  modelType: "optimization",

  variables: {
    x: {
      dimensions: [3],
      value: 0, lower: 0,
      finalValue: []
    }
  },

  data: {
    profits: {
      dimensions: [3],
      value: [75, 50, 35],
      binding: "get",
      finalValue: []
    }
  }
}
```

```

    },

    parts: {
      dimensions: [5, 3],
      value: [
        [1, 1, 0],
        [1, 0, 0],
        [2, 2, 1],
        [1, 1, 0],
        [2, 1, 1]
      ]
    },

    inventory: {
      value: [450, 250, 800, 450, 600]
    }
  },

  constraints: {
    num_used: {
      dimensions: [5],
      formula: "MMULT(parts, x)",
      upper: 'inventory'
    }
  },

  objective: {
    total: {
      formula: "sumproduct(x, profits)",
      type: "maximize",
      finalValue: []
    }
  }
}

```

- **Success Response:**

Code: 200 (OK)

Example Response:

```

{
  "status": {
    "code": 100,
    "id": "2590+2020-01-03-17-36-54-557510",
    "codeText": "Model Diagnosis Complete."
  },
  "ModelType": "LP Convex",
  "AllVars": 3,
  "SmoothVars": 3,
  "LinearVars": 3,
  "RecourseVars": 0,
  "UncertainVars": 0,
  "AllFcns": 6,
  "SmoothFcns": 6,
  "LinearFcns": 6,
  "RecourseFcns": 0,
  "UncertainFcns": 0,
  "AllDpns": 14,
  "SmoothDpns": 14,
  "LinearDpns": 14,
  "RecourseDpns": 0,
  "UncertainDpns": 0,

```

```

    "Bounds": 3,
    "Integers": 0,
    "ChanceConst": 0,
    "Sparsity": 77.77777777777786,
    "DummyObjective": false,
    "LinearEngine": true,
    "RangeInfo": false,
    "NumExceptions": 0,
    "NumUEExceptions": 0,
    "Exceptions": [],
    "UEExceptions": [],
    "modeltype": "linear",
    "variables": "3",
    "functions": "6",
    "bounds": 3,
    "integers": "0"
}

```

Error Response:

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

REST Endpoint: POST: <https://rason.net/api/optimize>

A REST API endpoint that accepts a RASON optimization, simulation optimization or stochastic optimization model, performs an optimization and returns a result in JSON format. This call should only be used with models that solve very quickly, i.e. within a few seconds. There is a time limit of 30 seconds on all synchronous calls.

- **URL**

`https://rason.net/api/optimize`

- **Method:**

POST

- **URL Params**

Required: None

- **Optional:** Any data component may be passed as a query parameter if a binding property exists for that parameter. For example, see the RASON example code snippet below.

```
data: {
  profits: {
    value: [[75, 50, 35]],
    binding: 'get'
  }
}
```

To change a query parameter outside of the RASON model environment, use:

```
$.Post("https://rason.net/api/optimize?profits=100,150,75", ...
```

RASON Server will map "profits=" with "profits = 100, 150, 75". A query can return any number of rows that satisfy the filtering condition, from 0 to infinity.

To perform a query using multiple parameters (say custID, maritalStatus and age) outside of the RASON model environment, use the query parameter:

```
$.get(https://rason.net/api/decision?custID=c2&maritalStatus=s&age=40 .....
```

Or in general,

```
$.get(https://rason.net/api/decision?par1=val1&par2=val2.....
```

- **Headers**

Required: Authorization - Example: Authorization: bearer {your RASON token}

Optional: None

- **Data Params**

An optimization, stochastic optimization or simulation optimization RASON Model.

```
{
  "modelName": "productMix4Example",
```

```

    "modelDescription": "Version of ProductMix using a 2-
dimensional
    matrix",
    "modelType": "optimization",
    "variables": {
        "x": {
            "dimensions": [ 3 ],
            "value": 0,
            "lower": 0,
            "finalValue": []
        }
    },
    "data": {
        "profits": {
            "dimensions": [ 3 ],
            "value": [ 75, 50, 35 ],
            "binding": "get",
            "finalValue": []
        },
        "parts": {
            "dimensions": [ 5, 3 ],
            "value": [
                [ 1, 1, 0 ],
                [ 1, 0, 0 ],
                [ 2, 2, 1 ],
                [ 1, 1, 0 ],
                [ 2, 1, 1 ]
            ]
        },
        "inventory": {
            "value": [ 450, 250, 800, 450, 600 ]
        }
    },
    "constraints": {
        "num_used": {
            "dimensions": [
                5
            ],
            "formula": "MMULT(parts, x)",
            "upper": "inventory"
        }
    },
    "objective": {
        "total": {
            "formula": "sumproduct(x, profits)",
            "type": "maximize",
            "finalValue": []
        }
    }
}

```

- **Success Response:**

Code: 200 (OK)

Example Response:

```
{
```



```

    "status": {
      "code": 0,
      "id": "2590+2020-03-18-16-16-32-113535",
      "codeText": "Solver found a solution. All constraints and optimality
conditions are satisfied."
    },
    "parametricInputs": {
      "profits": {
        "finalValue": [75, 50, 35]
      }
    },
    "variables": {
      "x": {
        "finalValue": [200, 200, 0]
      }
    },
    "objective": {
      "total": {
        "finalValue": 25000
      }
    }
  }
}

```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

REST Endpoint: POST: <https://rason.net/api/simulate>

A REST API endpoint that accepts a RASON model, runs a simulation and returns a result in JSON format. This call should only be used with models that run very quickly, i.e. within a few seconds. There is a time limit of 30 seconds on all synchronous calls.

- **URL**

`https://rason.net/api/simulate`

- **Method:**

POST

- **URL Params**

Required: None

- **Optional:** Any data component may be passed as a query parameter if a binding property exists for that parameter. For example, see the RASON example code snippet below.

```
data: {  
  tktPrice: {  
    value: [150],  
    binding: 'get'  
  }  
}
```

To change a query parameter outside of the RASON model environment, use:

```
$.Post("https://rason.net/api/simulate?tktpPrice=300", ...
```

RASON Server will map "tktpPrice=?" with "tktpPrice = 300". A query can return any number of rows that satisfy the filtering condition, from 0 to infinity.

To perform a query using multiple parameters (say custID, maritalStatus and age) outside of the RASON model environment, use the query parameter:

```
$.get(https://rason.net/api/decision?custID=c2&maritalStatus=s&age=40 .....)
```

Or in general,

```
$.get(https://rason.net/api/decision?par1=val1&par2=val2.....)
```

- **Headers**

Required: Authorization - Example: Authorization: bearer {your RASON token}

Optional: None

- **Data Params**

A simulation RASON Model

```
{  
  comment: "Simulation example",
```

```

modelSettings: {
  numSimulations: 1,
  numTrials: 100,
  randomSeed: 1
},
data: {
  price: {
    value: 200
  },
  capacity: {
    value: 100
  },
  sold: {
    value: 110
  },
  refund_no_shows: {
    value: 0.5
  },
  refund_overbook: {
    value: 1.25
  }
},
uncertainVariables: {
  no_shows: {
    formula: "PsiLogNormal(0.1*sold, 0.06*sold)",
    mean: []
  }
},

formulas: {
  show_ups: {
    formula: "sold - Round(no_shows, 0)"
  },
  overbook: {
    formula: "Max(0, show_ups - capacity)"
  }
},
uncertainFunctions: {
  revenue: {
    formula: "price*(sold -
refund_no_shows*Round(no_shows, 0) - refund_overbook*overbook)",
    percentiles: [],
    "trials": []
  }
}
}

```

- **Success Response:**

- **Code: 200 (OK)**

Example Response:

```

{
  "status": {
    "code": 0,
    "id": "2590+2019-11-18-21-51-05-591690",
    "codeText": "Solver has completed the simulation."
  },

```

[illegible]

```

20700,
20700,
20700,
20700,
20700,
20727,
20800,
20800,
20800,
20800,
20800,
20810.5,
20850,
20850,
20850,
20850,
20850,
20850,
20857,
20900,
20900,
20900,
20900,
20900,
20908,
21000,
21000,
21000,
21000,
21000,
21000,
21000
]
],
"trials": [
[
19900,
20250,
20100,
20550,
20700,
20600,
20500,
20700,
20550,
21000,
19700,
20700,
20250,
20600,
20300,
20850,
20550,
18800,
20900,
20700,
19950,
20550,
20700,
20900,
20250,
19600,
20400,
20600,
20700,
20700,
20550,
21000,
20800,
18400,
20800,

```

```
20250,
20700,
20600,
20800,
20400,
20400,
20200,
20400,
20400,
20400,
20700,
20250,
20250,
20700,
20250,
20850,
19200,
20900,
20400,
21000,
20100,
21000,
21000,
20300,
20550,
20200,
20800,
20100,
20400,
20700,
20800,
20900,
20900,
20550,
20500,
21000,
20400,
20900,
20100,
20100,
19900,
20700,
19950,
20100,
20500,
20100,
20400,
20700,
19500,
20550,
20800,
20000,
20550,
19950,
21000,
19950,
20400,
20400,
20850,
20850,
20850,
20850,
20500,
20850,
21000
    ]
  }
},
"uncertainVariables": {
  "no_shows": {
```

```
"mean": 10.9432
```

```
}
```

```
}
```

```
}
```

Error Response:

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

REST Endpoint: POST: <https://rason.net/api/solve>

A REST API endpoint that accepts analytic models (optimization, simulation, data science or decision table) in JSON, returns the results of solving these models in JSON and automatically creates an OData endpoint for the results. (A decision flow is a combination of 1 or more stages involving the solving or diagnosing of an optimization, simulation, decision table, or data science models.) This call should only be used with models that run very quickly, i.e. within a few seconds. There is a time limit of 30 seconds on all synchronous calls.

What is a Decision Flow?

RASON Decision Services enables users for the first time to define multiple "stages" in a **single** RASON file, where a stage can perform an SQL operation, apply a data transformation, train a machine learning model, apply it to score new data, run a simulation, solve a mathematical optimization problem or evaluate one or more linked decision tables. Results are passed between stages in a rich, standard "Indexed Data Frame" form.

A dataframe, in XLMiner SDK; the workhorse of the RASON Server, is a collection of data organized into named columns of equal length and homogeneous type. RASON uses DataFrames to deliver input data to an algorithm and to deliver the results of the algorithm back to the user. DataFrames hold heterogeneous data across columns (variables): numeric, categorical, or textual. When solving a decision flow containing optimization or simulation models, the columns that are indexed over the same dimensions and that belong to the same entity are reported in a single dataframe with multiple columns rather than multiple dataframes, i.e. final, dual, initial, etc for optimization results and statistics for uncertain variables or functions in simulation models. RASON can still bind to the individual results such as `optModel.x.finalValue` but will also consider the possibility of the last segment being a dataframe column rather than a separate dataframe. As a result, JSON responses are concise which greatly simplifies OData representation and querying.

Notes:

- A decision flow stage is terminal if no other stage in the decision flow depends on any of the results produced by that stage. Results for all terminal stages are currently always stored as JSON and returned within the JSON response and are available for REST/ODATA querying.
- The final value of an objective function in an optimization, stochastic optimization or simulation optimization model is currently not reported by default. This result must be explicitly requested using `finalValue: []` in the `objective` section in the RASON model. See the Defining your Optimization Model section that appears previously in this guide for more information.
- Custom JSON objects like fitted data science models are reported (if requested during evaluation) in JSON response and not database. This is also true for scalar data mining metrics, such as R^2 , accuracy, etc. These metrics are always reported in JSON and as a simple JSON scalar, not a dataframe.
- Row names are currently omitted in the serialized dataframe results from optimization, simulation and decision table stages and stored only for data science stages in order to reduce the verbosity of JSON responses. OData automatically adds a key field `_ID` that can be used for accessing records using row Id.
- OData handles row names, when present, by adding a `_Name` field to the entity. When data-storage = DATABASE, row names are stored as table columns, but ignored for inter-stage bindings.

- As mentioned above, the RASON Server supports decision flows containing any supported solving method, i.e. optimization, stochastic optimization, simulation, data science or decision table models. Decision flows may subsist of 1 stage or many stages. Typically, later stages in a decision flow are dependent on the results of stages occurring previously in the same decision flow, i.e. stage 3 depends on the results of stage 2 which depends on the results of stage 1. The final, or terminal, stage of a decision flow has no dependents, or children, i.e. stages that depend on it's results.
- Results for intermediate stages may be persisted on Azure storage and available for retrieval via RASON's OData (Open Data Protocol) endpoints, described later in this chapter. OData is a data access protocol for the Web providing a uniform way to query and manipulate data sets through CRUD (Create, read, update and delete) operations. Two layers of data storage are available, DATABASE or JSON. Results for terminal stages are currently always stored as JSON and returned within a JSON response; allowing the results to be available for REST/OData querying.

- **URL**

`https://rason.net/api/solve`

- **Method:**

POST

- **URL Params**

Required: None

Optional:

This endpoint offers five optional parameters controlling the content of the final results and where intermediate results are stored: stage, data-storage, keep-intermediate-results, simplify-final-results and response-format.

- Stage=<stage-name>
 - Use this parameter to retrieve information related to a specific stage of a decision flow. If this parameter is passed, the endpoint solves only the part of the decision flow required to solve the specified terminal stage, <stage-name>. The information produced by this optional parameter may be helpful during the creation or debugging phase. If a stage is not specified, a full Directed Acyclic Graph (DAG) solve is performed. To view all stages in a decision flow, click or use the REST endpoint GET:
`https://rason.net/api/model/{nameorid}/stages/all`.
 - Example: POST: `https://rason.net/api/solve?stage=imputation`
- Simplify-final-results = True/False
 - When True, this parameter tells the engine to store and report the results of, only, terminal stages as simplest possible JSON object: a scalar for 1x1 dataframe, 1d - array for Nx1 dataframe and 2d – array for NxM dataframe, No headers or index columns are stored. This option does not affect the results of intermediate stages, these are always dataframes. These simple results are reported in JSON Response may be queried with a REST call. Currently OData does not recognize these objects. The default setting is False.

- Example: POST: <https://rason.net/api/solve?simplify-final-results=true>
- Response-format = STANDALONE/WORKFLOW
 - Use this parameter to switch between the default dataframe-based decision flow reporting format (for both single- and multi-stage decision flows) and the default reporting format (for single-stage decision flows only). When response-format= WORKFLOW and simplify-final-results = false, the formula outputs are reported as dataframes. The default setting is WORKFLOW.
 - Note: If a formula refers to a decision table, the dataframe will be complete with column headers-as defined in the "outputs" section of the decision table definition, and property types – that will be available in both the JSON response and later in REST/OData queries. If a generic formula, RASON will use default column names and the best possible type for each result column.
 - Example: POST: <https://rason.net/api/solve?response-format=standalone>
 - Example: POST: <https://rason.net/api/solve?response-format=workflow>
- Keep-intermediate-results = True/False
 - This parameter determines whether the RASON server stores the results of the *intermediate* stages in the decision flow, returns them with JSON response and makes them available for REST or OData querying later. Currently, this parameter not only affects the "pipeline solve" (i.e. when a single terminal stage is specified) but also the entire decision flow solve, which might have multiple terminal stages. The default setting is False.
 - Example: POST: <https://rason.net/api/solve?keep-intermediate-results=true>
- Data-storage = DATABASE/JSON
 - This parameter controls whether the SQLite DATABASE or JSON file storage is used to store the results of *intermediate* stages. Database mode does not affect terminal stages. When database mode is selected, the in-memory SQLite database is used to store the results of intermediate stages and will spill on disk in out-of-memory conditions. The default setting is Database.
 - DATABASE mode can be orders of magnitude faster than JSON mode. In JSON mode, costs are incurred when reading/writing to/from disk. However, in DATABASE mode all in-memory database read/write operations are much more efficient. Intermediate results can still be stored and examined in DATABASE mode. These results may be queried with REST API or OData but they will *not* appear in the JSON response as serialized data frames.
 - If data-storage = database and
 - keep-intermediate-results =False, the database stays in memory and will be discarded after terminal stages are processed.
 - keep-intermediate-results = True, an in-memory database is saved to disk to enable later querying.
 - Example: POST: <https://rason.net/api/solve?data-storage=json>

When submitted together, the two parameters, data-storage and keep-intermediate-results, create four cases.

Example: POST: <https://rason.net/api/solve?data-storage=json&keep-intermediate-results=true>

- data-storage = JSON and keep-intermediate-results = True
 - SQLite database is not used to store results. Rather, JSON files are used for solving, storing, responses and querying. The results of intermediate and terminal stages are reported in JSON response as well as being available for REST/OData querying.
- data-storage = JSON and keep-intermediate-results = False
 - Same as when data-storage = JSON and keep-intermediate-results = True, except intermediate results are discarded after the decision flow solve. These results are not reported in JSON response and are not available for querying.
- data-storage = DATABASE and keep-intermediate-results = True
 - SQLite database is used for solving and storing the results of intermediate stages. Results are not returned within JSON response and are available for REST/OData querying only from the stored database. The results of terminal stages are initially stored in JSON files, reported within JSON response and available for REST/OData querying from stored JSON files.
- data-storage = DATABASE and keep-intermediate-results = False
 - Same as if data-storage = DATABASE and keep-intermediate-results = True except the database with intermediate results is discarded after the decision flow is solved. These results are not reported in JSON response and are not available for later querying.

- **Headers**

Required: Authorization - Example: Authorization: bearer {your RASON token}

Optional: None

- **Data Params**

A decision flow RASON Model

```
{
  "workflow": "imputationWorkflow",
  "imputation": {
    "datasets": {
      "trainData": {
        "value": [
          [
            "black",
            null,
            6,
            2,
            1,
            "nan",
```

```

        1
      ],
      [
        "",
        3,
        9,
        5.1,
        null,
        "",
        2
      ],
      [
        "red",
        7,
        8,
        null,
        9.2,
        "small",
        3
      ],
      [
        "red",
        10000,
        null,
        4.4,
        4.4,
        "large",
        -1
      ],
      [
        "blue",
        2,
        3,
        5.6,
        3.4,
        "unknown",
        5
      ]
    ],
    "colNames": [
      "A",
      "B",
      "C",
      "D",
      "E",
      "F",
      "G"
    ]
  },
  "estimator": {
    "myImputer": {
      "type": "transformation",
      "algorithm": "imputation",
      "parameters": {
        "imputationStrategy": [

```

```

        "A",
        "MODE"
    ],
    [
        "B",
        "MEAN"
    ],
    [
        "C",
        "MEDIAN"
    ],
    [
        "D",
        "DELETE_RECORD"
    ],
    [
        "E",
        "DELETE_RECORD"
    ],
    [
        "F",
        "VALUE"
    ],
    [
        "G",
        "MEAN"
    ]
],
"placeholder": [
    [
        "B",
        10000
    ],
    [
        "F",
        "unknown"
    ],
    [
        "G",
        -1
    ]
]
}
},
"actions": {
    "imputingModel": {
        "data": "trainData",
        "estimator": "myImputer",
        "action": "fit",
        "evaluations": [
            "fittedModelJson"
        ]
    }
}
},
"transforming": {

```

```

    "comment": "imputing missing values based on json model from
the parent stage",
    "datasources": {
        "myModelSrc": {
            "type": "stage",
            "connection": "imputation",
            "selection": "imputingModel.fittedModelJson",
            "direction": "import"
        }
    },
    "datasets": {
        "newData": {
            "value": [
                [
                    "blue",
                    null,
                    null,
                    4,
                    1,
                    "unknown",
                    -1
                ]
            ],
            "colNames": [
                "A",
                "B",
                "C",
                "D",
                "E",
                "F",
                "G"
            ]
        },
        "imputingModel": {
            "binding": "myModelSrc"
        }
    },
    "actions": {
        "fixedNewData": {
            "data": "newData",
            "fittedModel": "imputingModel",
            "parameters": {
                "imputation": [
                    [
                        "F",
                        "medium"
                    ]
                ]
            },
            "action": "transform",
            "evaluations": [
                "transformation"
            ]
        }
    }
}

```

- **Success Response:**

Code: 200 (OK)

```
{
  "status": {
    "id": "2590+2019-11-18-22-06-58-637468",
    "code": 0,
    "codeText": "Success"
  },
  "results": {
    "transforming.fixedNewData.transformation": []
  },
  "transforming": {
    "status": {
      "code": 0,
      "codeText": "Success"
    },
    "fixedNewData": {
      "transformation": {
        "objectType": "dataFrame",
        "name": "newdata",
        "order": "col",
        "rowNames": [
          "Record 1"
        ],
        "colNames": [
          "Treated_A",
          "Treated_B",
          "Treated_C",
          "D",
          "E",
          "Treated_F",
          "Treated_G"
        ],
        "colTypes": [
          "wstring",
          "double",
          "double",
          "double",
          "double",
          "wstring",
          "double"
        ],
        "indexCols": null,
        "data": [
          [
            "blue"
          ],
          [
            4
          ],
          [
            7
          ]
        ]
      }
    }
  }
}
```

- **Error Response:**

Request sent to the RASON Server was incorrect or corrupted.

Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.

Request forbidden; user may be trying to access protected data.

User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.

Something has gone wrong on the RASON Server, contact technical support.

Asynchronous RASON REST API Endpoints

For all but the smallest / most quickly solved models, you'll need to use the second set of REST API calls. Each of these calls are asynchronous. You'll first need to make a call to the REST API endpoint `POST rason.net/api/model` which will create a "resource ID". Then you'll need to start an optimization, simulation, decision table recalculation, data science task via the REST API endpoints `GET rason.net/api/model/{nameorid}/optimize`, `GET rason.net/api/model/{nameorid}/simulate`, `GET rason.net/api/model/{nameorid}/decision` or `GET rason.net/api/model/{nameorid}/datamine`. Additionally, you can use `GET rason.net/api/model/{nameorid}/solve` to solve any type of RASON model (optimization, simulation, data science, or decision table) which will automatically create an OData endpoint for the results.

You can check on the model's progress at any time with the REST API endpoint `GET rason.net/api/model/{nameorid}/status` and obtain results when completed with the REST API endpoint `GET rason.net/api/model/{nameorid}/result`. To stop an optimization, simulation, decision table recalculation or data science task that is in progress, use `POST rason.net/api/model/{nameorid}/stop`. To delete a model or decision table, use `DELETE rason.net/api/model/{nameorid}/delete`.

- [DELETE rason.net/api/model/{nameorid}](#)
- [DELETE rason.net/api/model/{nameorid}?delete-champion=true/false](#)
- [DELETE rason.net/api/model/{nameorid}?kind=fitted,excel,RASON or all](#)
- [DELETE rason.net/api/model/{nameorid}?type=origin,version,instance or all](#)
- [DELETE rason.net/api/model/{nameorid}/runtoken](#)
- [GET rason.net/api/interface](#)
- [GET rason.net/api/model](#)
- [GET rason.net/api/model/{nameorid}](#)
- [Get rason.net/api/model/{name}/champion](#)
- [GET rason.net/api/model/{nameorid}/datamine](#)
- [GET rason.net/api/model/{nameorid}/decision](#)
- [GET rason.net/api/model/{nameorid}/diagnose](#)
- [GET rason.net/api/model/{nameorid}/fitted-info](#)
- [Get rason.net/api/model/{nameorid}/interface](#)
- [GET rason.net/api/model/{nameorid}/optimize](#)
- [GET rason.net/api/model/{nameorid}/result](#)
- [GET rason.net/api/model/{nameorid}/result/data](#)

- [GET rason.net/api/model/{nameorid}/runtoken](#)
- [GET rason.net/api/model/{nameorid}/scoring-model](#)
- [GET rason.net/api/model/{nameorid}/simulate](#)
- [GET rason.net/api/model/{nameorid}/solve](#)
- [GET rason.net/api/model/{nameorid}/stages](#)
- [GET rason.net/api/model/{nameorid}/stages/<stage-name or all>](#)
- [GET rason.net/api/model/{nameorid}/status](#)
- [PATCH rason.net/api/model/{nameorid}](#)
- [POST rason.net/api/fitted-info](#)
- [POST rason.net/api/interface](#)
- [POST rason.net/api/score](#)
- [POST rason.net/api/scoring-model](#)
- [POST rason.net/api/model](#)
- [POST rason.net/api/model/excel?connection-name=<NamedExcelConnection!Worksheet>](#)
- [POST rason.net/api/stages](#)
- [POST rason.net/api/stages/{stage}](#)
- [POST rason.net/api/model/{nameorid}/score](#)
- [POST rason.net/api/model/{nameorid}/solvetype](#)
- [POST rason.net/api/model/{nameorid}/stop](#)
- [PUT rason.net/api/model/{nameorid}](#)
- [PUT rason.net/api/model/excel?connection-name=<NamedExcelConnection!Worksheet>](#)

REST Endpoint:

DELETE: *https://rason.net/api/model/{nameorid}*

This REST API endpoint deletes the specified model instance and any attached files.

- If a valid model name is passed for {nameorid}, all model versions, instances, fitted models, attached files and/or results will be deleted for the given model name.
- If a valid fitted model name is passed for {nameorid}, all fitted models with that name, their versions and all corresponding PMML/JSON files will be deleted.
- If a valid model id is passed for {nameorid}, the specified model and attached files and results will be deleted.

Response format:

```
{
  "deletedModels": [ - Information pertaining to all successfully
    deleted models
      "ModelId": "", - ID of deleted model
      "ModelName": "", - Name of deleted model
      "ModelDescr": "", - Text from "modelDescription" property of
        deleted model (optional).
      "ModelFiles": [], - Files uploaded during model POST (optional).
      "RuntimeToken": "", - Runtime Token (optional) of deleted model.
      "ModelType": "", - Model type as specified by optional
        "modelType" property
      "ModelKind": "", - Kind of model (fitted, Excel or RASON)
      "IsChampion": true/false, - Specifies if model version is the
        champion*
      "ParentModelId": null, - DK for the Parent Model
      "QueryString": "" - Query Parameters, if used
      "ModelContainer": null - Only applicable with an Organization
        Account.
      (For more information on Organization Accounts, see the
        Appendix.)
    ],
  "notDeletedModels": [], - Information concerning all models not
    deleted because they are currently executing.
  "invalidModels": [], - Any invalid resource identifiers listed
    here.
  "status": "OK" - descriptive status menu
}
```

Note: Use ?force=true/false query parameter to force deletion of models that are still executing.

*This field will only return "true" if the model has been specifically set as the champion using the REST API endpoint PATCH *rason.net/api/model/{nameorid}*.

URL

https://rason.net/api/model/{nameorid}

https://rason.net/api/model/{fitted-model}

https://rason.net/api/model/{id}

- **Method:**

DELETE

- **URL Params**

Required: None

Optional: None

- **Headers**

Required: Authorization - Example: Authorization: bearer {your RASON token}

Optional: None

- **Data Params:** None

- **Success Response:**

Code: 200 (OK)

Response Example:

- DELETE api/model/{modelname}

Example: DELETE api/model/NearestNeighborsClassification

Deletes the origin, a version, a fitted model and an instance of the NearestNeighborsClassification model.

```
{
  "deletedModels": [
    {
      "ModelId": "2590+NearestNeighborsClassification+2020-05-22-20-12-800790",
      "ModelName": "NearestNeighborsClassification",
      "ModelDescr": "classification: k nearest neighbors; scoring example JSONClassifier.json uses exported fitted model, knncModel, to score new data.",
      "ModelFiles": [
        {
          "fileName": "hald-small-binary-train.txt",
          "isOnServer": false
        },
        {
          "fileName": "hald-small-binary-valid.txt",
          "isOnServer": false
        },
        {
          "fileName": "classification-nearest-neighbors.xml",
          "isOnServer": false
        },
        {
          "fileName": "classification-nearest-neighbors.json",
          "isOnServer": false
        }
      ]
    },
    {
      "RuntimeToken": "",

```

```

        "ModelType": "Origin",
        "ModelKind": "RASON",
        "IsChampion": false,
        "ParentModelId": null,
        "QueryString": ""
    },
    {
20-         "ModelId": "2590+NearestNeighborsClassification+2020-05-22-20-12-
           473765",
        "ModelName": "NearestNeighborsClassification",
        "ModelDescr": "classification: k nearest neighbors; scoring exampl
e JSONC1
assifier.json uses exported fitted model, knncModel, to score new data.",
        "ModelFiles": [
            {
                "fileName": "hald-small-binary-train.txt",
                "isOnServer": false
            },
            {
                "fileName": "hald-small-binary-valid.txt",
                "isOnServer": false
            },
            {
                "fileName": "classification-nearest-neighbors.xml",
                "isOnServer": false
            },
            {
                "fileName": "classification-nearest-neighbors.json",
                "isOnServer": false
            }
        ],
        "RuntimeToken": "",
        "ModelType": "Instance",
        "ModelKind": "RASON",
        "IsChampion": false,
        "ParentModelId": "2590+NearestNeighborsClassification+2020-05-22-
20-12-12-800790",
        "QueryString": "response-format=STANDALONE"
    },
    {
54-         "ModelId": "2590+NearestNeighborsClassification+2020-05-26-19-37-
           664126",
        "ModelName": "NearestNeighborsClassification",
        "ModelDescr": "classification: k nearest neighbors; scoring exampl
e JSONC1
assifier.json uses exported fitted model, knncModel, to score new data.",
        "ModelFiles": [
            {
                "fileName": "hald-small-binary-train.txt",
                "isOnServer": false
            },
            {
                "fileName": "hald-small-binary-valid.txt",
                "isOnServer": false
            },
            {
                "fileName": "classification-nearest-neighbors.xml",
                "isOnServer": false
            },
            {
                "fileName": "classification-nearest-neighbors.json",
                "isOnServer": false
            }
        ],
        "RuntimeToken": "",
        "ModelType": "Version",

```

```

        "ModelKind": "RASON",
        "IsChampion": true,
        "ParentModelId": "2590+NearestNeighborsClassification+2020-05-22-
20-12-12-
        800790",
        "QueryString": ""
    },
    {
        "ModelId": "2590+fitted-myjsonsrc+2020-05-28-20-56-05-705194",
        "ModelName": "fitted-myjsonsrc",
        "ModelDescr": "classification-logistic-model.json",
        "ModelFiles": [],
        "RuntimeToken": "",
        "ModelType": "Origin",
        "ModelKind": "Fitted",
        "IsChampion": false,
        "ParentModelId": "2590+NearestNeighborsClassification+2020-05-22-
20-12-12-
        800790",
        "QueryString": ""
    },
    ],
    "notDeletedModels": [],
    "invalidModels": [],
    "status": "OK"
}

```

- DELETE api/model/{fittedname}

Example: DELETE api/model/fitted-myjsonsrc

```

{
    "deletedModels": [
        {
            "ModelId": "2590+fitted-myjsonsrc+2020-05-22-20-27-55-492264",
            "ModelName": "fitted-myjsonsrc",
            "ModelDescr": "classification-lda-model.json",
            "ModelFiles": [],
            "RuntimeToken": "",
            "ModelType": "Origin",
            "ModelKind": "Fitted",
            "IsChampion": false,
            "ParentModelId": "2590+DiscriminantAnalysis1+2020-05-22-20-27-53-
212764",
            "QueryString": ""
        },
        {
            "ModelId": "2590+fitted-myjsonsrc+2020-05-22-20-29-45-743664",
            "ModelName": "fitted-myjsonsrc",
            "ModelDescr": "regression-bagging.json",
            "ModelFiles": [],
            "RuntimeToken": "",
            "ModelType": "Origin",
            "ModelKind": "Fitted",
            "IsChampion": false,
            "ParentModelId": "2590+BaggingRegression+2020-05-22-20-29-43-
843501",
            "QueryString": ""
        }
    ],
    "notDeletedModels": [],
    "invalidModels": [],
    "status": "OK"
}

```

- DELETE api/model/{id}

Example: DELETE api/model/2590+2020-05-22-20-37-05-032407

```

{
    "deletedModels": [
        {
            "ModelId": "2590+2020-05-22-20-37-05-032407",
            "ModelName": "",

```

```

rescale      "ModelDescr": "DF that partitions a dataset into three partitions,
              s the partitions and runs MLR",
              "ModelFiles": [
                {
                  "fileName": "hald-small.txt",
                  "isOnServer": false
                },
                {
                  "fileName": "hald-small-unlabeled.txt",
                  "isOnServer": false
                }
              ],
              "RuntimeToken": "",
              "ModelType": "Origin",
              "ModelKind": "RASON",
              "IsChampion": false,
              "ParentModelId": null,
              "QueryString": ""
            },
            "notDeletedModels": [],
            "invalidModels": [],
            "status": "OK"
          }
        }

```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

[Return to Asynchronous Endpoint List](#)

REST Endpoint: DELETE:

https://rason.net/api/model/{nameorid}?delete-champion=true/false

If *true* (the default), this endpoint behaves the same as DELETE `rason.net/api/model/{nameorid}` i.e. all versions (including the champion version), instances, fitted models, attached files, results are deleted for the given model name. If *false*, this endpoint behaves the same as DELETE `rason.net/api/model/{nameorid}` **except** the champion version is not deleted.

Response Format:

```
{
  "deletedModels": [ - Information pertaining to all successfully
    deleted
                        models
    "ModelId": "", - ID of deleted model
    "ModelName": "", - Name of deleted model
    "ModelDescr": "", - Text from "modelDescription" property of
      deleted model (optional).
    "ModelFiles": [], - Files uploaded during model POST (optional).
    "RuntimeToken": "", - Runtime Token (optional) of deleted model.
    "ModelType": "", - Model type as specified by optional
      "modelType" property
    "ModelKind": "", - Kind of model (fitted, Excel or RASON)
    "IsChampion": true/false, - Specifies if model version is the
      champion*
    "ParentModelId": null, - DK for the Parent Model
    "QueryString": "" - Query Parameters, if used
  ],
  "notDeletedModels": [], - Information concerning all models not
    deleted because they are currently executing.
  "invalidModels": [], - Any invalid resource identifiers listed
    here.
  "status": "OK" - descriptive status menu
}
```

Note: Use ?force=true/false query parameter to force deletion of models that are still executing.

*This field will only return "true" if the model has been specifically set as the champion using the REST API endpoint PATCH `rason.net/api/model/{nameorid}`.

- **URL**

`https://rason.net/api/model/{nameorid}?delete-champion=true/false`

- **Method:**

DELETE

- **URL Params**

Required: None

Optional: None

- **Headers**

Required: Authorization - Example: Authorization: bearer {your RASON token}

Optional: None

- **Data Params:** None

- **Success Response:**

Code: 200 (OK)

Response Example:

- DELETE RASON.com/api/model/{modelname}?delete-champion=true
Example: DELETE RASON.com/api/model/UGProjectSelect1Example?delete-champion=true
Note that both the champion version and the non champion version are both deleted.

```
{
  "deletedModels": [
    {
      "ModelId": "2590+UGProjectSelect1Example+2020-03-25-14-18-42-513688",
      "ModelName": "UGProjectSelect1Example",
      "ModelDescr": "UGProjectSelect1 Example - Stochastic Optimization",
      "ModelFiles": [],
      "RuntimeToken": "",
      "ModelType": "Origin",
      "ModelKind": "RASON",
      "IsChampion": true,
      "ParentModelId": null,
      "QueryString": ""
    },
    {
      "ModelId": "2590+UGProjectSelect1Example+2020-03-25-14-18-50-873215",
      "ModelName": "UGProjectSelect1Example",
      "ModelDescr": "UGProjectSelect1 Example - Stochastic Optimization",
      "ModelFiles": [],
      "RuntimeToken": "",
      "ModelType": "Instance",
      "ModelKind": "RASON",
      "IsChampion": false,
      "ParentModelId": "2590+UGProjectSelect1Example+2020-03-25-14-18-42-513688",
      "QueryString": "response-format=STANDALONE"
    }
  ],
  "notDeletedModels": [],
  "invalidModels": [],
  "status": "OK"
}
```
- DELETE RASON.com/api/model/{modelname}?delete-champion=false
Example: DELETE RASON.com/api/model/UGProjectSelect1Example?delete-champion=false

```
{
  "deletedModels": [
```

```

{
  "ModelId": "2590+Test2+2020-05-02-02-23-24-778898",
  "ModelName": "Test2",
  "ModelDescr": "",
  "ModelFiles": [],
  "RuntimeToken": "",
  "ModelType": "Origin",
  "ModelKind": "RASON",
  "IsChampion": false,
  "ParentModelId": null,
  "QueryString": ""
},
{
  "ModelId": "2590+Test2+2020-05-02-02-23-32-528113",
  "ModelName": "Test2",
  "ModelDescr": "",
  "ModelFiles": [],
  "RuntimeToken": "",
  "ModelType": "Instance",
  "ModelKind": "RASON",
  "IsChampion": false,
  "ParentModelId": "2590+Test2+2020-05-02-02-23-24-778898",
  "QueryString": ""
}
],
"notDeletedModels": [],
"invalidModels": [],
"status": "OK"
}

```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

[Return to Asynchronous Endpoint List](#)

REST Endpoint: DELETE:

https://rason.net/api/model/{nameorid}?kind={fitted,excel,RASON or all}

If a valid model name is passed for {nameorid}, endpoint behaves the same as DELETE rason.net/api/model/{nameorid} but affects only specified kinds of models: fitted, Excel, RASON or all.

- "fitted" deletes fitted Data Science models in PMML/JSON format
- "excel" deletes models defined in the Excel language, i.e. Excel models.
- "RASON" deletes models defined using the RASON modeling language.
- "all" deletes fitted, excel and RASON models.

Response format:

```
{
  "deletedModels": [ - Information pertaining to all successfully
    deleted models
    "ModelId": "", - ID of deleted model
    "ModelName": "", - Name of deleted model
    "ModelDescr": "", - Text from "modelDescription" property of
    deleted model (optional).
    "ModelFiles": [], - Files uploaded during model POST (optional).
    "RuntimeToken": "", - Runtime Token (optional) of deleted model.
    "ModelType": "", - Model type as specified by optional
    "modelType" property
    "ModelKind": "", - Kind of model (fitted, Excel or RASON)
    "IsChampion": true/false, - Specifies if model version is the
    champion*
    "ParentModelId": null, - DK for the Parent Model
    "QueryString": "" - Query Parameters, if used
  ],
  "notDeletedModels": [], - Information concerning all models not
  deleted because they are currently executing.
  "invalidModels": [], - Any invalid resource identifiers listed
  here.
  "status": "OK" - descriptive status menu
}
```

Note: Use ?force=true/false query parameter to force deletion of models that are still executing.

*This field will only return "true" if the model has been specifically set as the champion using the REST API endpoint PATCH rason.net/api/model/{nameorid}.

- **URL**
 - https://rason.net/api/model/{nameorid}?kind=fitted deletes all fitted models with given {nameorid} or {fitted-name}

Note: DELETE rason.net/api/model/ExampleA/fitted *will not* delete fitted models created from a solve of ExampleA. To delete these models, use: DELETE <https://rason.net/api/model/{fitted-name}?kind=fitted>

- <https://rason.net/api/model/{nameorid}?kind=excel> deletes Excel models with given {nameorid}
- <https://rason.net/api/model/{nameorid}?kind=RASON> deletes RASON models with given {nameorid}
- <https://rason.net/api/model/{nameorid}?kind=all> deletes all fitted, Excel and RASON model versions with given {nameorid} or {fitted-name}

- **Method:**

DELETE

- **URL Params**

Required: None

Optional: kind=fitted, excel, RASON or all

- **Headers**

Required: Authorization - Example: Authorization: bearer {your RASON token}

Optional: None

- **Data Params:** None

- **Success Response:**

Code: 200 (OK)

Response Example:

- DELETE api/model/{nameorid}/?kind=fitted
Example: DELETE <https://rason.net/api/model/2590+fitted-myjsonsrc+2020-05-26-20-43-50-721463?kind=fitted>

```
{
  "deletedModels": [
    {
      "ModelId": "2590+fitted-myjsonsrc+2020-05-26-20-43-50-721463",
      "ModelName": "fitted-myjsonsrc",
      "ModelDescr": "classification-logistic-model.json",
      "ModelFiles": [],
      "RuntimeToken": "",
      "ModelType": "Origin",
      "ModelKind": "Fitted",
      "IsChampion": false,
      "ParentModelId": "2590+LogisticRegression+2020-05-26-20-43-47-987029",
      "QueryString": ""
    }
  ],
  "notDeletedModels": [],
  "invalidModels": [],
  "status": "OK"
}
```

```
}
```

- DELETE api/model/{nameorid}/?kind=excel

Example: DELETE

[https://rason.net/api/model/2StageFarmer1\(Stochastic\)?kind=excel](https://rason.net/api/model/2StageFarmer1(Stochastic)?kind=excel)

```
{
  "deletedModels": [
    {
      "ModelId": "2590+2StageFarmer1(Stochastic)+2020-05-26-21-11-54-342640",
      "ModelName": "2StageFarmer1(Stochastic)",
      "ModelDescr": "",
      "ModelFiles": [],
      "RuntimeToken": "",
      "ModelType": "Origin",
      "ModelKind": "Excel",
      "IsChampion": false,
      "ParentModelId": null,
      "QueryString": ""
    }
  ],
  "notDeletedModels": [],
  "invalidModels": [],
  "status": "OK"
}
```

- DELETE api/model/{nameorid}/?kind=RASON

Example: DELETE

[https://rason.net/api/model/2StageFarmer1\(Stochastic\)?kind=RASON](https://rason.net/api/model/2StageFarmer1(Stochastic)?kind=RASON)

```
{
  "deletedModels": [
    {
      "ModelId": "2590+2StageFarmer1(Stochastic)1+2020-05-26-21-12-54-091292",
      "ModelName": "2StageFarmer1(Stochastic)1",
      "ModelDescr": "UGAirlineHub Nonlinear Optimization Example - See Example walkthrough in RASON User Guide",
      "ModelFiles": [
        {
          "fileName": "AirlineHubData.csv",
          "isOnServer": false
        }
      ],
      "RuntimeToken": "",
      "ModelType": "Origin",
      "ModelKind": "RASON",
      "IsChampion": false,
      "ParentModelId": null,
      "QueryString": ""
    },
    {
      "ModelId": "2590+2StageFarmer1(Stochastic)1+2020-05-26-21-12-58-819840",
      "ModelName": "2StageFarmer1(Stochastic)1",
      "ModelDescr": "UGAirlineHub Nonlinear Optimization Example - See Example walkthrough in RASON User Guide",
      "ModelFiles": [
        {
          "fileName": "AirlineHubData.csv",
          "isOnServer": false
        }
      ],
      "RuntimeToken": "",
      "ModelType": "Version",
      "ModelKind": "RASON",
      "IsChampion": false,
      "ParentModelId": "2590+2StageFarmer1(Stochastic)1+2020-05-26-21-12-
```

```

        "54-091292",
        "QueryString": ""
    }
},
"notDeletedModels": [],
"invalidModels": [],
"status": "OK"
}

```

- DELETE api/model/{nameorid}/?kind=all

Example: DELETE

[https://rason.net/api/model/2StageFarmer1\(Stochastic\)?kind=all](https://rason.net/api/model/2StageFarmer1(Stochastic)?kind=all)

Notice that ModelKind for the first deleted model is "RASON" and "fitted" for the 2nd deleted model.

```

{
  "deletedModels": [
    {
      "ModelId": "2590+LogisticRegression1+2020-05-26-21-51-54-409999",
      "ModelName": "LogisticRegression1",
      "ModelDescr": "classification: logistic regression; scoring example PMMLC1
assifier.json uses exported fitted model, lrModel, to score new data",
      "ModelFiles": [
        {
          "fileName": "hald-small-binary-train.txt",
          "isOnServer": false
        },
        {
          "fileName": "hald-small-binary-valid.txt",
          "isOnServer": false
        },
        {
          "fileName": "classification-logistic-model.xml",
          "isOnServer": false
        },
        {
          "fileName": "classification-logistic-model.json",
          "isOnServer": false
        }
      ],
      "RuntimeToken": "",
      "ModelType": "Origin",
      "ModelKind": "RASON",
      "IsChampion": false,
      "ParentModelId": null,
      "QueryString": ""
    },
    {
      "ModelId": "2590+fitted-myjsonsrc+2020-05-26-20-43-50-721463",
      "ModelName": "fitted-myjsonsrc",
      "ModelDescr": "classification-logistic-model.json",
      "ModelFiles": [],
      "RuntimeToken": "",
      "ModelType": "Origin",
      "ModelKind": "Fitted",
      "IsChampion": false,
      "ParentModelId": "2590+LogisticRegression+2020-05-26-20-43-47-987029",
      "QueryString": ""
    }
  ],
  "notDeletedModels": [],
  "invalidModels": [],
  "status": "OK"
}

```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

[Return to Asynchronous Endpoint List](#)

REST Endpoint: DELETE:

<https://rason.net/api/model/{nameorid}?type={origin,version,instance or all}>

If a valid model name is passed for {nameorid}, endpoint behaves the same as DELETE rason.net/api/model/{nameorid} but affects only specified types of models: origin, version, instance or all.

- "origin" deletes the originally posted model.
- "version" deletes models of the same name created by using PUT rason.net/model/api/{nameorid}.
- "instance" deletes an instance of a model which is created when a model is solved.
- "all" deletes origin models, model versions and model instances.

```
{
  "deletedModels": [ - Information pertaining to all successfully
    deleted models
    "ModelId": "", - ID of deleted model
    "ModelName": "", - Name of deleted model
    "ModelDescr": "", - Text from "modelDescription" property of
      deleted model (optional).
    "ModelFiles": [], - Files uploaded during model POST (optional).
    "RuntimeToken": "", - Runtime Token (optional) of deleted model.
    "ModelType": "", - Model type as specified by optional
      "modelType" property
    "ModelKind": "", - Kind of model (fitted, Excel or RASON)
    "IsChampion": true/false, - Specifies if model version is the
      champion*
    "ParentModelId": null, - DK for the Parent Model
    "QueryString": "" - Query Parameters, if used
  ],
  "notDeletedModels": [], - Information concernting all models not
    deleted because they are currently executing.
  "invalidModels": [], - Any invalid resource identifiers listed
    here.
  "status": "OK" - descriptive status menu
}
```

Note: Use ?force=true/false query parameter to force deletion of models that are still executing.

*This field will only return "true" if the model has been specifically set as the champion using the REST API endpoint PATCH rason.net/api/model/{nameorid}.

- URL
 - <https://rason.net/api/model/{nameorid}?type=origin>

- <https://rason.net/api/model/{nameorid}?type=version>
- <https://rason.net/api/model/{nameorid}?type=instance>
- <https://rason.net/api/model/{nameorid}?type=all>
- **Method:**
DELETE
- **URL Params**
Required: None
Optional: None
- **Headers**
Required: Authorization - Example: Authorization: bearer {your RASON token}
Optional: None
- **Data Params:** None
- **Success Response:**
Code: 200 (OK)
Response Example:
 - DELETE api/model/{nameorid}?type=origin
Example: DELETE
<https://rason.net/api/model/LogisticRegression?type=origin>

```
{
  "deletedModels": [
    {
      "ModelId": "2590+LogisticRegression+2020-05-26-20-41-53-035392",
      "ModelName": "LogisticRegression",
      "ModelDescr": "classification: logistic regression; scoring example PMMLClassifier.json uses exported fitted model, lrModel, to score new data",
      "ModelFiles": [
        {
          "fileName": "hald-small-binary-train.txt",
          "isOnServer": false
        },
        {
          "fileName": "hald-small-binary-valid.txt",
          "isOnServer": false
        },
        {
          "fileName": "classification-logistic-model.xml",
          "isOnServer": false
        },
        {
          "fileName": "classification-logistic-model.json",
          "isOnServer": false
        }
      ]
    },
    {
      "RuntimeToken": "",
      "ModelType": "Origin",
      "ModelKind": "RASON",
      "IsChampion": false,
    }
  ]
}
```

```

        "ParentModelId": null,
        "QueryString": ""
    }
],
"notDeletedModels": [],
"invalidModels": [],
"status": "OK"
}

```

- DELETE api/model/{nameorid}?type=version

Example: DELETE

<https://rason.net/api/model/optSimWorkflow?type=version>

```

{
    "deletedModels": [
        {
            "ModelId": "2590+optSimWorkflow+2020-05-08-16-58-11-900613",
            "ModelName": "optSimWorkflow",
            "ModelDescr": "",
            "ModelFiles": [],
            "RuntimeToken": "Version",
            "ModelType": "",
            "ModelKind": "RASON",
            "IsChampion": false,
            "ParentModelId": "2590+optSimWorkflow+2020-05-08-15-57-20-590941",
            "QueryString": ""
        },
    ],
    "notDeletedModels": [],
    "invalidModels": [],
    "status": "OK"
}

```

- DELETE api/model/{nameorid}?type=instance

Example: DELETE

<https://rason.net/api/model/UGYieldManagement2Example?type=instance>

```

{
    "deletedModels": [
        {
            "ModelId": "2590+UGYieldManagement2Example+2020-03-11-18-37-22-268626",
            "ModelName": "UGYieldManagement2Example",
            "ModelDescr": "UGYieldManagement2 Example - Parametric Simulation",
            "ModelFiles": [],
            "RuntimeToken": "",
            "ModelType": "Instance",
            "ModelKind": "RASON",
            "IsChampion": false,
            "ParentModelId": "2590+UGYieldManagement2Example+2020-03-11-18-37-17-615391",
            "QueryString": "response-format=STANDALONE"
        },
    ],
    "notDeletedModels": [],
    "invalidModels": [],
    "status": "OK"
}

```

- DELETE api/model/{nameorid}?type=all

Example: DELETE <https://rason.net/api/model?type=all>

Note that ModelType is "Origin" for the first deleted model and "version" for the 2nd deleted model.

```

{
    "deletedModels": [
        {
            "ModelId": "2590+productMixExample+2020-05-27-17-35-33-978111",
            "ModelName": "productMixExample",

```

```

        "ModelDescr": "Your company manufactures TVs, stereos and speakers
        , using a common parts inventory of power supplies, speakers, etc.
        Parts are in limited supply and you must determine the
        most profitable mix of products to build. See our Tutorial
        Online for step-by step instructions on formulating this linear
        programming model.",
        "ModelFiles": [],
        "RuntimeToken": "",
        "ModelType": "Origin",
        "ModelKind": "RASON",
        "IsChampion": false,
        "ParentModelId": null,
        "QueryString": ""
    },
    {
        "ModelId": "2590+productMixExample+2020-05-27-17-37-34-667234",
        "ModelName": "productMixExample",
        "ModelDescr": "Your company manufactures TVs, stereos and speakers
        , using
        a common parts inventory of power supplies, speaker cones, etc.
        Parts are in limited supply and you must determine the most
        profitable mix of products to build. See our Tutorial
        Online for step-by-step instructions on formulating this
        linear programming model.",
        "ModelFiles": [],
        "RuntimeToken": "",
        "ModelType": "Version",
        "ModelKind": "RASON",
        "IsChampion": false,
        "ParentModelId": "2590+productMixExample+2020-05-27-17-35-33-
        978111",
        "QueryString": ""
    },
    {
        "ModelId": "2590+productMixExample+2020-05-27-17-38-04-882364",
        "ModelName": "productMixExample",
        "ModelDescr": "Your company manufactures TVs, stereos and speakers
        , using
        a common parts inventory of power supplies, speaker cones, etc.
        Parts are in limited supply and you must determine the
        most profitable mix of products to build. See our Tutorial
        Online for step-by-step instructions on formulating this
        linear programming model.",
        "ModelFiles": [],
        "RuntimeToken": "",
        "ModelType": "Instance",
        "ModelKind": "RASON",
        "IsChampion": false,
        "ParentModelId": "2590+productMixExample+2020-05-27-17-37-34-
        667234",
        "QueryString": "response-format=STANDALONE"
    }
],
"notDeletedModels": [],
"invalidModels": [],
"status": "OK"
}

```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.

Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

[Return to Asynchronous Endpoint List](#)

REST Endpoint: DELETE:
https://rason.net/api/model/{nameorid}/runtoken

A Rest API endpoint that deletes a run time token for a named (or unnamed) model.

- **URL**

`https://rason.net/api/model/{nameorid}/runtoken`

- **Method:**

DELETE

- **URL Params**

Required: None

Optional: None

- **Headers**

Required: Authorization - Example: Authorization: bearer {your RASON token}

Optional: None

- **Data Params:** None

- **Success Response:**

Code: 200 (OK)

Response Example:

`Runtime permissions for model RGProductMixExcel11 removed.`

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted or model not found. See response body for more information.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

[Return to Asynchronous Endpoint List](#)

REST Endpoint: GET:

https://rason.net/api/interface?connection-name=<ConnectionName>

A REST API endpoint that generates an interface from a model saved to a Sharepoint/OneDrive account, that can be utilized in a decision flow. This endpoint accepts Analytic Solver optimization, simulation, stochastic optimization, simulation optimization and decision table models defined in a Microsoft Excel workbook.

ConnectionName must be either a OneDrive Data Connection or a Sharepoint Data Connection that exists on the client's MyAccount page on www.RASON.com. For more information on RASON Data Connections, see the Data Connections section the occurs previously in this guide.

If a model is defined within an Excel workbook with multiple worksheets, use the optional URL parameter !<worksheet> to specify the correct worksheet name. If this query parameter is not present, the active sheet (or the entire workbook) is considered.

Example: GET <https://rason.net/api/interface?connection-name=<ConnectionName>!<WorksheetName>>

- **URL**

<https://rason.net/api/interface?connection-name=<ConnectionName>>

- **Method:**

GET

- **URL Params**

Required: ?connection-name=<ConnectionName>

Optional: !<WorksheetName> - To specify a specific worksheet within an Excel workbook saved on OneDrive or Sharepoint.

- **Headers**

Required: Authorization - Example: Authorization: bearer {your RASON token}

Optional: None

- **Data Params:** An Analytic Solver optimization, simulation, simulation optimization, stochastic optimization or decision table model defined within an Excel workbook saved on a OneDrive or Sharepoint account.

- **Success Response:**

Code: 200 (OK)

Example Request: GET <https://rason.net/api/interface?connection-name=MyConnection!Forecast with Uncertainty>

Response Example:

```

{
  "comment": "This interface has been generated by Psi for an
Excel
  model in the workbook BusinessForecast(Sim).xlsx stored on One
  Drive",
  "invokeModel": "Name=Sim_July24!Forecast with Uncertainty",
  "modelType": "simulation",
  "modelDescription": "",
  "outputResults": {
    "d19": {
      "evaluations": [
        "mean"
      ],
      "type": "number",
      "comment": "uncertain function"
    }
  }
}

```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted or model not found. See response body for more information.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

[Return to Asynchronous Endpoint List](#)

REST Endpoint: GET: <https://rason.net/api/model>

A REST API endpoint that returns all available resource IDs and information for all models. Use a query parameter to filter for:

- specific named model only (name),
- the original posted model (origin),
- model version (version),
- model instance (instance),
- fitted models only (fitted),
- Excel models only (excel),
- RASON models only (RASON),
- named RASON models only (RASON)
- all named models (all)

Below is a brief explanation of the returned properties

- **ModelId:** The unique identifier for the model.
- **ModelName:** The name of the model taken from the model text. When using POST rason.net/api/model/<name> the name from the url is validated against the name in the model, if specified
- **ModelDescr:** The model description taken directly from the model text
- **ModelFiles:** The list of files used by the model as taken from the model text.
- **RuntimeToken:** Lists any runtime token associated with this model
- **ModelType:** Lists the *type* of model such as "origin", "version", "instance".
 - “origin” – the initially posted model
 - “version” – means that this model is a version of a previously POSTed or PUT model. Note: Conceptually these are different versions of the POSTed model.
 - “instance” – these are models created whenever a model is solved.
- **ModelKind:** Lists the *kind* of model such as "fitted", "excel" or "RASON".
 - “fitted” – fitted Data Science model in PMML/JSON format
 - “excel” – models defined in Excel language
 - “RASON” – models defined in RASON language
- **IsChampion:** True *only* if model is marked as the “Champion” using the API endpoint PATCH rason.net/api/model/{nameorid}.

- **ParentModelId:** Identifies the model from which this model is derived. In the case of “Origin” models, this is always empty. Otherwise, this property identifies the model used to create this model. For example, if a user POST’s a model and then PUT’s another version of this model, “parentModelId” of the version will be the modelId of the originally posted model. Similar logic holds for posting versions of version and instances of version.
 - **QueryString:** The query string associated with the model when it was created. In the case of unnamed models, this will change throughout the model’s lifecycle. A user may post an unnamed model with a particular query string but then solve the model with another. In the case of named models, this is the query string associated with the action used to create the model. A model could be posted with one query string, have several versions with different query strings and associated instances with yet another query string. If a particular version or instance doesn’t have a query string, it will inherit the query string of the parent.
 - **ModelContainer: (Property only applies to Organization Account users)** The container within the Azure Storage account where the model is saved (POSTed). Otherwise, this property is reported as “null”. For more information on Organization Accounts, see the Appendix: Organization Accounts.
- **URL**
https://rason.net/api/model
 - **Method:**
GET
 - **URL Params**
Required: None
Optional:
 - GET https://rason.net/api/model?type=origin, version, instance
Lists info for all models of specified type.
"origin" - originally POSTed model
"version" – created by using the API endpoint PUT rason.net/api/model/{nameorid} which creates a new version of the model of the same model name.
"instance" – created when a RASON model is solved.
"all" – Lists info for all original models, versions and instances. May also be omitted.
 - GET https://rason.net/api/model?kind=fitted, excel or RASON
Lists info for all models of specified kind.
"fitted" – fitted Data Science model in PMML/JSON format
"excel" – models defined in Excel
"RASON" – created when a RASON model is solved.
"all" – Lists info for all fitted, Excel or RASON models. May also be omitted.
 - **Headers**
Required: Authorization - Example: Authorization: bearer {your RASON token}
Optional: None
 - **Data Params:** None
 - **Success Response:**
Code: 200 (OK)

- Example Request: GET <https://rason.net/api/model>

Example Response:

```
[
  {
    "ModelId": "2590+DecisionTreeClassification+2020-05-28-21-48-35-064322",
    "ModelName": "DecisionTreeClassification",
    "ModelDescr": "classification: decision tree",
    "ModelFiles": [
      {
        "fileName": "hald-small-binary-train.txt",
        "isOnServer": false
      },
      {
        "fileName": "hald-small-binary-valid.txt",
        "isOnServer": false
      },
      {
        "fileName": "classification-decision-tree.xml",
        "isOnServer": false
      },
      {
        "fileName": "classification-decision-tree.json",
        "isOnServer": false
      }
    ],
    "RuntimeToken": "",
    "ModelType": "Origin",
    "ModelKind": "RASON",
    "IsChampion": false,
    "ParentModelId": null,
    "QueryString": "",
    "ModelContainer": null
  },
  {
    "ModelId": "2590+LogisticRegression+2020-05-28-21-07-59-763658",
    "ModelName": "LogisticRegression",
    "ModelDescr": "classification: logistic regression; scoring example PM
MLClass
ifier.json uses exported fitted model, lrModel, to score new data",
    "ModelFiles": [
      {
        "fileName": "hald-small-binary-train.txt",
        "isOnServer": false
      },
      {
        "fileName": "hald-small-binary-valid.txt",
        "isOnServer": false
      },
      {
        "fileName": "classification-logistic-model.xml",
        "isOnServer": false
      },
      {
        "fileName": "classification-logistic-model.json",
        "isOnServer": false
      }
    ],
    "RuntimeToken": "",
    "ModelType": "Instance",
    "ModelKind": "RASON",
    "IsChampion": false,
    "ParentModelId": "2590+LogisticRegression+2020-05-28-21-04-19-737715",
    "QueryString": "response-format=STANDALONE",
    "ModelContainer": null
  },
  {
    "ModelId": "2590+2StageFarmer1(Stochastic)+2020-05-26-21-46-10-180902",
```

```

        "ModelName": "2StageFarmer1(Stochastic)",
        "ModelDescr": "",
        "ModelFiles": [],
        "RuntimeToken": "",
        "ModelType": "Origin",
        "ModelKind": "Excel",
        "IsChampion": false,
        "ParentModelId": null,
        "QueryString": "",
        "ModelContainer": null
    },
    {
        "ModelId": "2590+fitted-jsonmodelsrc+2020-05-26-20-31-32-407619",
        "ModelName": "fitted-jsonmodelsrc",
        "ModelDescr": "classification-nearest-neighbors.json",
        "ModelFiles": [],
        "RuntimeToken": "",
        "ModelType": "Origin",
        "ModelKind": "Fitted",
        "IsChampion": false,
        "ParentModelId": "2590+NearestNeighborsClassification+2020-05-26-20-31355966",
        "QueryString": "",
        "ModelContainer": null
    }
]

```

- GET <https://rason.net/api/model/{nameorid}?type={origin, version, instance or all}>

Example Request:

<https://rason.net/api/model?type=origin>

Example Response:

```

[
    {
        "ModelId": "2590+DecisionTreeClassification+2020-05-28-21-48-35-064322",
        "ModelName": "DecisionTreeClassification",
        "ModelDescr": "classification: decision tree",
        "ModelFiles": [
            {
                "fileName": "hald-small-binary-train.txt",
                "isOnServer": false
            },
            {
                "fileName": "hald-small-binary-valid.txt",
                "isOnServer": false
            },
            {
                "fileName": "classification-decision-tree.xml",
                "isOnServer": false
            },
            {
                "fileName": "classification-decision-tree.json",
                "isOnServer": false
            }
        ],
        "RuntimeToken": "",
        "ModelType": "Origin",
        "ModelKind": "RASON",
        "IsChampion": false,
        "ParentModelId": null,
        "QueryString": "",
        "ModelContainer": null
    }
]

```

Example Request:

<https://rason.net/api/model?type=version>

Example Response:

```
[
  {
    "ModelId": "2590+mlrDF+2020-05-21-01-12-10-082250",
    "ModelName": "mlrDF",
    "ModelDescr": "DF that partitions a dataset into three partitions, rescales the partitions and runs MLR",
    "ModelFiles": [
      {
        "fileName": "hald-small.txt",
        "isOnServer": true
      },
      {
        "fileName": "hald-small-unlabeled.txt",
        "isOnServer": true
      }
    ],
    "RuntimeToken": "",
    "ModelType": "Version",
    "ModelKind": "RASON",
    "IsChampion": false,
    "ParentModelId": "2590+mlrDF+2020-05-21-01-10-23-882710",
    "QueryString": "keep-intermediate-results=true&data-storage=JSON",
    "ModelContainer": null
  }
]
```

Example Request:

<https://rason.net/api/model?type=instance>

Example Response:

```
[
  {
    "ModelId": "2590+LogisticRegression+2020-05-28-21-07-59-763658",
    "ModelName": "LogisticRegression",
    "ModelDescr": "classification: logistic regression; scoring example PMMLC lassifier.json uses exported fitted model, lrModel, to score new data",
    "ModelFiles": [
      {
        "fileName": "hald-small-binary-train.txt",
        "isOnServer": false
      },
      {
        "fileName": "hald-small-binary-valid.txt",
        "isOnServer": false
      },
      {
        "fileName": "classification-logistic-model.xml",
        "isOnServer": false
      },
      {
        "fileName": "classification-logistic-model.json",
        "isOnServer": false
      }
    ],
    "RuntimeToken": "",
    "ModelType": "Instance",
    "ModelKind": "RASON",
    "IsChampion": false,
    "ParentModelId": "2590+LogisticRegression+2020-05-28-21-04-19-737715",
    "QueryString": "response-format=STANDALONE",
    "ModelContainer": null
  }
]
```

Example Request:

<https://rason.net/api/model?type=all> - Behaves the same as GET rason.net/api/model

Example Response:

```
[
  {
    "ModelId": "2590+DecisionTreeClassification+2020-05-28-21-48-35-064322",
    "ModelName": "DecisionTreeClassification",
    "ModelDescr": "classification: decision tree",
    "ModelFiles": [
      {
        "fileName": "hald-small-binary-train.txt",
        "isOnServer": false
      },
      {
        "fileName": "hald-small-binary-valid.txt",
        "isOnServer": false
      },
      {
        "fileName": "classification-decision-tree.xml",
        "isOnServer": false
      },
      {
        "fileName": "classification-decision-tree.json",
        "isOnServer": false
      }
    ],
    "RuntimeToken": "",
    "ModelType": "Origin",
    "ModelKind": "RASON",
    "IsChampion": false,
    "ParentModelId": null,
    "QueryString": "",
    "ModelContainer": null
  },
  {
    "ModelId": "2590+LogisticRegression+2020-05-28-21-07-59-763658",
    "ModelName": "LogisticRegression",
    "ModelDescr": "classification: logistic regression; scoring example PM MLClass  
ifier.json uses exported fitted model, lrModel, to score new data",
    "ModelFiles": [
      {
        "fileName": "hald-small-binary-train.txt",
        "isOnServer": false
      },
      {
        "fileName": "hald-small-binary-valid.txt",
        "isOnServer": false
      },
      {
        "fileName": "classification-logistic-model.xml",
        "isOnServer": false
      },
      {
        "fileName": "classification-logistic-model.json",
        "isOnServer": false
      }
    ],
    "RuntimeToken": "",
    "ModelType": "Instance",
    "ModelKind": "RASON",
    "IsChampion": false,
    "ParentModelId": "2590+LogisticRegression+2020-05-28-21-04-19-737715",
    "QueryString": "response-format=STANDALONE",
    "ModelContainer": null
  },
  {
    "ModelId": "2590+2StageFarmer1 (Stochastic)+2020-05-26-21-46-10-180902",
    "ModelName": "2StageFarmer1 (Stochastic)",
    "ModelDescr": ""
  }
]
```

```

        "ModelFiles": [],
        "RuntimeToken": "",
        "ModelType": "Origin",
        "ModelKind": "Excel",
        "IsChampion": false,
        "ParentModelId": null,
        "QueryString": "",
        "ModelContainer": null
    },
    {
        "ModelId": "2590+fitted-jsonmodelsrc+2020-05-26-20-31-32-407619",
        "ModelName": "fitted-jsonmodelsrc",
        "ModelDescr": "classification-nearest-neighbors.json",
        "ModelFiles": [],
        "RuntimeToken": "",
        "ModelType": "Origin",
        "ModelKind": "Fitted",
        "IsChampion": false,
        "ParentModelId": "2590+NearestNeighborsClassification+2020-05-26-20-31355966",
        "QueryString": "",
        "ModelContainer": null
    }
]

```

- GET <https://rason.net/api/model?kind={fitted, excel, RASON or all}>

Example Request:

<https://rason.net/api/model?kind=fitted>

Example Response:

```

[
  {
    "ModelId": "2590+fitted-jsonmodelsrc+2020-05-26-20-31-32-407619",
    "ModelName": "fitted-jsonmodelsrc",
    "ModelDescr": "classification-nearest-neighbors.json",
    "ModelFiles": [],
    "RuntimeToken": "",
    "ModelType": "Origin",
    "ModelKind": "Fitted",
    "IsChampion": false,
    "ParentModelId": "2590+NearestNeighborsClassification+2020-05-26-20-31-30-355966",
    "QueryString": "",
    "ModelContainer": null
  }
]

```

Example Request:

<https://rason.net/api/model?kind=excel>

Example Response:

```

[
  {
    "ModelId": "2590+2StageFarmer1(Stochastic)+2020-05-26-21-46-10-180902",
    "ModelName": "2StageFarmer1(Stochastic)",
    "ModelDescr": "",
    "ModelFiles": [],
    "RuntimeToken": "",
    "ModelType": "Origin",
    "ModelKind": "Excel",
    "IsChampion": false,
    "ParentModelId": null,
    "QueryString": "",
    "ModelContainer": null
  }
]

```

Example Request:

<https://rason.net/api/model?kind=RASON>

Example Response:

```
{
  "ModelId": "2590+DecisionTreeClassification+2020-05-28-21-48-35-064322",
  "ModelName": "DecisionTreeClassification",
  "ModelDescr": "classification: decision tree",
  "ModelFiles": [
    {
      "fileName": "hald-small-binary-train.txt",
      "isOnServer": false
    },
    {
      "fileName": "hald-small-binary-valid.txt",
      "isOnServer": false
    },
    {
      "fileName": "classification-decision-tree.xml",
      "isOnServer": false
    },
    {
      "fileName": "classification-decision-tree.json",
      "isOnServer": false
    }
  ],
  "RuntimeToken": "",
  "ModelType": "Origin",
  "ModelKind": "RASON",
  "IsChampion": false,
  "ParentModelId": null,
  "QueryString": "",
  "ModelContainer": null
}
```

Example Request:

<https://rason.net/api/model?kind=all> - Behaves the same as GET rason.net/api/model

Example Response:

```
[
  {
    "ModelId": "2590+DecisionTreeClassification+2020-05-28-21-48-35-064322",
    "ModelName": "DecisionTreeClassification",
    "ModelDescr": "classification: decision tree",
    "ModelFiles": [
      {
        "fileName": "hald-small-binary-train.txt",
        "isOnServer": false
      },
      {
        "fileName": "hald-small-binary-valid.txt",
        "isOnServer": false
      },
      {
        "fileName": "classification-decision-tree.xml",
        "isOnServer": false
      },
      {
        "fileName": "classification-decision-tree.json",
        "isOnServer": false
      }
    ],
    "RuntimeToken": "",
    "ModelType": "Origin",
    "ModelKind": "RASON",
    "IsChampion": false,
    "ParentModelId": null,
    "QueryString": ""
  }
]
```

```

"ModelContainer": null
},
{
  "ModelId": "2590+LogisticRegression+2020-05-28-21-07-59-763658",
  "ModelName": "LogisticRegression",
  "ModelDescr": "classification: logistic regression; scoring example PM
MLClass
ifier.json uses exported fitted model, lrModel, to score new data",
  "ModelFiles": [
    {
      "fileName": "hald-small-binary-train.txt",
      "isOnServer": false
    },
    {
      "fileName": "hald-small-binary-valid.txt",
      "isOnServer": false
    },
    {
      "fileName": "classification-logistic-model.xml",
      "isOnServer": false
    },
    {
      "fileName": "classification-logistic-model.json",
      "isOnServer": false
    }
  ],
  "RuntimeToken": "",
  "ModelType": "Instance",
  "ModelKind": "RASON",
  "IsChampion": false,
  "ParentModelId": "2590+LogisticRegression+2020-05-28-21-04-19-737715",
  "QueryString": "response-format=STANDALONE",
  "ModelContainer": null
},
{
  "ModelId": "2590+2StageFarmer1(Stochastic)+2020-05-26-21-46-10-180902",
  "ModelName": "2StageFarmer1(Stochastic)",
  "ModelDescr": "",
  "ModelFiles": [],
  "RuntimeToken": "",
  "ModelType": "Origin",
  "ModelKind": "Excel",
  "IsChampion": false,
  "ParentModelId": null,
  "QueryString": "",
  "ModelContainer": null
},
{
  "ModelId": "2590+fitted-jsonmodelsrc+2020-05-26-20-31-32-407619",
  "ModelName": "fitted-jsonmodelsrc",
  "ModelDescr": "classification-nearest-neighbors.json",
  "ModelFiles": [],
  "RuntimeToken": "",
  "ModelType": "Origin",
  "ModelKind": "Fitted",
  "IsChampion": false,
  "ParentModelId": "2590+NearestNeighborsClassification+2020-05-26-20-31355966",
  "QueryString": "",
  "ModelContainer": null
}
]

```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
------------------------------	--

Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

[Return to Asynchronous Endpoint List](#)

REST Endpoint: GET: <https://rason.net/api/model/{nameorid}>

If a valid model name is passed for "nameorid", the model text or workbook for the champion or most recent model is returned.

If a valid resource ID is passed for "nameorid", the model text or workbook for that specific resource is returned.

If a valid model name is passed for "nameorid" and a filter such as "?type", "?kind" or "?return", the model itself or the information for a given model name for the specified kind/type will be returned.

- **ModelId:** The unique identifier for the model
- **ModelName:** The name of the model taken from the model text. When using POST rason.net/api/model/<name> the name from the url is validated against the name in the model, if specified
- **ModelDescr:** The model description taken directly from the model text
- **ModelFiles:** The list of files used by the model as taken from the model text.
- **RuntimeToken:** Lists any runtime token associated with this model
- **ModelType:** Lists the *type* of model such as "origin", "version", "instance". Conceptually, "version" refers to different version of the model in question.
 - “origin” – the initially posted model
 - “version” – means that this model is a version of a previously POSTed or PUT model
 - “instance” – these are models created whenever a model is solved.
- **ModelKind:** Lists the *kind* of model such as "fitted", "excel" or "RASON".
 - “fitted” – fitted Data Science model in PMML/JSON format
 - “excel” – models defined in Excel language

Note that GET rason.net/api/model may also return the Excel model workbook as an attachment for ModelKind=Excel.

 - “RASON” – models defined in RASON language
- **IsChampion:** True *only* if model is marked as the “Champion” using the API endpoint PATCH rason.net/api/model/{nameorid}.
- **ParentModelId:** Identifies the model from which this model is derived. In the case of “Origin” models, this is always empty. Otherwise, this property identifies the model used to create this model. For example, if a user POST’s a model and then PUT’s another version of this model, "parentModelId" of the version will be the modelId of the originally posted model. Similar logic holds for posting versions of version and instances of version.
- **QueryString:** The query string associated with the model when it was created. In the case of unnamed models, this will change throughout the model’s lifecycle. A user may post an unnamed model with a particular query string but then solve the model with another. In the case of named models, this is the query string associated with the action used to create the model. A model could be posted with one query string, have several versions with different query strings and associated instances with yet another query string. If a particular version or instance doesn’t have a query string, it will inherit the query string of the parent.
- **ModelContainer: (Property only applies to Organization Account users)** The container within the Azure Storage account where the model is saved (POSTed). Otherwise, this

property is reported as "null". For more information on Organization Accounts, see the Appendix: Organization Accounts.

- **URL**

`https://rason.net/api/model/{nameorid}`

- **Method:**

GET

- **URL Params**

Required: None

Optional:

- GET `https://rason.net/api/model/{nameorid}?type={origin, version, instance or all}` – Lists info for all models of specified type.

"origin" - originally POSTed model

"version" – created by using the API endpoint `PUT rason.net/api/model/{nameorid}` which creates a new version of the model of the same model name.

"instance" – created when a RASON model is solved.

"all" – returns models with type = "origin", "version" and "instance". May be omitted.

- GET `https://rason.net/api/model/{nameorid}?kind=fitted, excel or RASON`

Lifts info for all models of specified kind.

"fitted" – fitted Data Science model in PMML/JSON format

"excel" – models defined in Excel

"RASON" – created when a RASON model is solved.

"all" – Lists info for all fitted, Excel or RASON models. May be omitted.

- GET `https://rason.net/api/model/{nameorid}?return=info or model`

Returns an array of model information objects or model text/workbook.

- **Headers**

Required: Authorization - Example: `Authorization: bearer {your RASON token}`

Optional: None

- **Data Params:** None

- **Success Response:**

Code: 200 (OK)

- `https://rason.net/api/model/{nameorid}`

Example Request

`https://rason.net/api/model/StratifiedSampling`

Example Response

```
{
  "modelName": "StratifiedSampling",
  "modelDescription": "transformation: stratified sampling",
  "modelType": "datamining",
  "datasources": {
    "mySrc": {
      "type": "csv",
      "connection": "hald-small-binary.txt",
      "direction": "import"
    }
  },
  "datasets": {
    "myData": {
      "binding": "mySrc",
      "strataCol": "Y"
    }
  },
  "transformer": {
    "mySampler": {
      "type": "transformation",
      "algorithm": "stratifiedSampling",
      "parameters": {
        "sampleSize": 10,
        "replaceOption": false,
        "sortIndexes": false,
        "seed": 123,
        "StratificationMethod": "PROPORTIONAL"
      }
    }
  },
  "actions": {
    "sampleData": {
      "data": "myData",
      "action": "transform",
      "evaluations": [
        "transformation"
      ]
    }
  }
}
```

Example Request

<https://rason.net/api/model/flow-MyExcelConnection>

Response for an Excel model saved to a user's OneDrive account, using a named connection maintained on the user's www.RASON.com account.

Example Response

```
{
  "flowName": "flow-MyExcelConnection",
  "flowDescription": "This flow has been generated for an Excel model at the location provided by data connection [MyExcelConnection]",
  "excelStage": {
    "comment": "This interfact has been generated...",
    "invokeModel": "Name=MyExcelConnection!ProductMixExample1",
    "modelType": "optimization",
    "modelDescription": "",
    "outputResults": {
      "Number_to_build": {
        "evaluation": [
          "finalValue"
        ],
        "type": "array",
        "comment": "decision variable block"
      }
    },
    "Total_Profit": {
      "evaluations": [
```

```

        "finalValue"
      ],
      "type": "number",
      "comment": "objective function"
    }
  }
}

```

- <https://rason.net/api/model/{modelid}?type={origin, version, instance or all}>

Example Request:

<https://rason.net/api/model/DecisionTreeClassification?type=origin>

Example Response:

```

[
  {
    "ModelId": "2590+DecisionTreeClassification+2020-05-28-21-48-35-064322",
    "ModelName": "DecisionTreeClassification",
    "ModelDescr": "classification: decision tree",
    "ModelFiles": [
      {
        "fileName": "hald-small-binary-train.txt",
        "isOnServer": false
      },
      {
        "fileName": "hald-small-binary-valid.txt",
        "isOnServer": false
      },
      {
        "fileName": "classification-decision-tree.xml",
        "isOnServer": false
      },
      {
        "fileName": "classification-decision-tree.json",
        "isOnServer": false
      }
    ],
    "RuntimeToken": "",
    "ModelType": "Origin",
    "ModelKind": "RASON",
    "IsChampion": false,
    "ParentModelId": null,
    "QueryString": "",
    "ModelContainer": null
  }
]

```

Example Request:

<https://rason.net/api/model/mlrDF?type=version>

Example Response:

```

[
  {
    "ModelId": "2590+mlrDF+2020-05-21-01-12-10-082250",
    "ModelName": "mlrDF",
    "ModelDescr": "DF that partitions a dataset into three partitions, rescales the partitions and runs MLR",
    "ModelFiles": [
      {
        "fileName": "hald-small.txt",
        "isOnServer": true
      },
      {
        "fileName": "hald-small-unlabeled.txt",

```

```

        "isOnServer": true
    },
    ],
    "RuntimeToken": "",
    "ModelType": "Version",
    "ModelKind": "RASON",
    "IsChampion": false,
    "ParentModelId": "2590+mlrDF+2020-05-21-01-10-23-882710",
    "QueryString": "keep-intermediate-results=true&data-storage=JSON",
    "ModelContainer": null
}
]

```

Example Request:

<https://rason.net/api/model/LogisticRegression?type=instance>

Example Response:

```

[
  {
    "ModelId": "2590+LogisticRegression+2020-05-28-21-07-59-763658",
    "ModelName": "LogisticRegression",
    "ModelDescr": "classification: logistic regression; scoring example PMMLC
lassifie
r.json uses exported fitted model, lrModel, to score new data",
    "ModelFiles": [
      {
        "fileName": "hald-small-binary-train.txt",
        "isOnServer": false
      },
      {
        "fileName": "hald-small-binary-valid.txt",
        "isOnServer": false
      },
      {
        "fileName": "classification-logistic-model.xml",
        "isOnServer": false
      },
      {
        "fileName": "classification-logistic-model.json",
        "isOnServer": false
      }
    ],
    "RuntimeToken": "",
    "ModelType": "Instance",
    "ModelKind": "RASON",
    "IsChampion": false,
    "ParentModelId": "2590+LogisticRegression+2020-05-28-21-04-19-737715",
    "QueryString": "response-format=STANDALONE",
    "ModelContainer": null
  }
]

```

Example Request:

<https://rason.net/api/model/mlrDF?type=all>

Example Response:

```

[
  {
    "ModelId": "2590+mlrDF+2020-05-21-01-12-18-469787",
    "ModelName": "mlrDF",
    "ModelDescr": "DF that partitions a dataset into three partitions, res
cales the partitions and runs MLR",
    "ModelFiles": [
      {
        "fileName": "hald-small.txt",
        "isOnServer": true
      },
      {
        "fileName": "hald-small-unlabeled.txt",

```

```

        "isOnServer": true
    }
},
"RuntimeToken": "",
"ModelType": "Instance",
"ModelKind": "RASON",
"IsChampion": false,
"ParentModelId": "2590+mlrDF+2020-05-21-01-12-10-082250",
"QueryString": "keep-intermediate-results=true&data-storage=JSON",
"ModelContainer": null
},
{
    "ModelId": "2590+mlrDF+2020-05-21-01-12-10-082250",
    "ModelName": "mlrDF",
    "ModelDescr": "DF that partitions a dataset into three partitions, rescales the partitions and runs MLR",
    "ModelFiles": [
        {
            "fileName": "hald-small.txt",
            "isOnServer": true
        },
        {
            "fileName": "hald-small-unlabeled.txt",
            "isOnServer": true
        }
    ],
    "RuntimeToken": "",
    "ModelType": "Origin",
    "ModelKind": "RASON",
    "IsChampion": false,
    "ParentModelId": "null",
    "QueryString": "keep-intermediate-results=true&data-storage=JSON",
    "ModelContainer": null
},
{
    "ModelId": "2590+mlrDF+2020-05-19-22-20-06-803420",
    "ModelName": "mlrDF",
    "ModelDescr": "DF that partitions a dataset into three partitions, rescales the partitions and runs MLR",
    "ModelFiles": [
        {
            "fileName": "hald-small.txt",
            "isOnServer": true
        },
        {
            "fileName": "hald-small-unlabeled.txt",
            "isOnServer": true
        }
    ],
    "RuntimeToken": "",
    "ModelType": "Version",
    "ModelKind": "RASON",
    "IsChampion": false,
    "ParentModelId": "2590+mlrDF+2020-05-21-01-12-10-082250",
    "QueryString": "",
    "ModelContainer": null
}
}
]

```

- <https://rason.net/api/model/{modelid}?kind={excel, RASON, fitted or all}>

Example Request:

<https://rason.net/api/model/BusinessForecastPsiInput?kind=excel>
1

Example Response:

Note that this endpoint returns an Excel model workbook as an attachment.

Example Request:

`https://rason.net/api/model/rescaling-reusable?kind=RASON`

Example Response:

```
{
  "modelName": "rescaling-reusable",
  "modelType": "datamining",
  "modelDescription": "transformation: rescaling",
  "inputParameters": {
    "trainData": {
      "type": "dataset",
      "value": null,
      "comment": ""
    },
    "validData": {
      "type": "dataset",
      "value": null,
      "comment": ""
    },
    "testData": {
      "type": "dataset",
      "value": null,
      "comment": ""
    }
  },
  "datasources": {
    "newDataSource": {
      "type": "csv",
      "connection": "hald-small-unlabeled.txt"
    }
  },
  "datasets": {
    "newData": {
      "binding": "newDataSource"
    }
  },
  "estimator": {
    "myRescaler": {
      "type": "transformation",
      "algorithm": "rescaling",
      "parameters": {
        "technique": "UNIT_NORMALIZATION",
        "normType": "L1",
        "excludedCols": [
          "y"
        ]
      }
    }
  },
  "actions": {
    "rescalerModel": {
      "trainData": "trainData",
      "estimator": "myRescaler",
      "action": "fit",
      "evaluations": [
        "statistics"
      ]
    },
    "rescaledTrainData": {
      "data": "trainData",
      "fittedModel": "rescalerModel",
      "action": "transform",
      "evaluations": [
        "transformation"
      ]
    },
    "rescaledValidData": {
      "data": "validData",
      "fittedModel": "rescalerModel",

```



```

        "action": "transform",
        "evaluations": [
            "transformation"
        ]
    },
    "rescaledTestData": {
        "data": "testData",
        "fittedModel": "rescalerModel",
        "action": "transform",
        "evaluations": [
            "transformation"
        ]
    },
    "rescaledNewData": {
        "data": "newData",
        "fittedModel": "rescalerModel",
        "action": "transform",
        "evaluations": [
            "transformation"
        ]
    }
}
}

```

Example Request:

<https://rason.net/api/model/fitted-myjsonsrc?kind=fitted>

Example Response:

```

{
    "modelName": "BaggingClassification",
    "objectType": "fittedModel",
    "type": "classification",
    "algorithm": "bagging",
    "variables": [
        "X1",
        "X2",
        "X3",
        "X4",
        "Weights"
    ],
    "targetColumn": "Y",
    "categoricalVariables": {},
    "fittedModel": {
        "classes": {
            "objectType": "stringVector",
            "name": "StringVector",
            "data": [
                "0",
                "1"
            ]
        }
    },
    "weakLearners": [
        {
            "modelName": "",
            "objectType": "fittedModel",
            "type": "classification",
            "algorithm": "decisionTree",
            "variables": [
                "X1",
                "X2",
                "X3",
                "X4",
                "Weights"
            ],
            "targetColumn": "Y",
            "categoricalVariables": {},
            "fittedModel": {
                "classes": {
                    "objectType": "stringVector",
                    "name": "StringVector",

```

```

    "data": [
      "0",
      "1"
    ]
  },
  "priorProb": {
    "0": 0.46153846153846156,
    "1": 0.53846153846153844
  },
  "root": {
    "id": 0,
    "score": 1,
    "recordCount": 13,
    "predicate": null,
    "scoreDistribution": [
      {
        "category": "0",
        "recordCount": 6
      },
      {
        "category": "1",
        "recordCount": 7
      }
    ],
    "leftChild": {
      "id": 1,
      "score": 1,
      "recordCount": 8,
      "predicate": {
        "type": "continuous",
        "field": "X4",
        "operator": "less",
        "splitValue": 36.5
      },
      "scoreDistribution": [
        {
          "category": "0",
          "recordCount": 1
        },
        {
          "category": "1",
          "recordCount": 7
        }
      ],
      "leftChild": {
        "id": 3,
        "score": 0,
        "recordCount": 1,
        "predicate": {
          "type": "continuous",
          "field": "X1",
          "operator": "less",
          "splitValue": 2.5
        },
        "scoreDistribution": [
          {
            "category": "0",
            "recordCount": 1
          },
          {
            "category": "1",
            "recordCount": 0
          }
        ]
      },
      "rightChild": {
        "id": 4,
        "score": 1,
        "recordCount": 7,
        "predicate": {
          "type": "continuous",

```

```

        "field": "X1",
        "operator": "greaterOrEqual",
        "splitValue": 2.5
    },
    "scoreDistribution": [
        {
            "category": "0",
            "recordCount": 0
        },
        {
            "category": "1",
            "recordCount": 7
        }
    ]
    },
    "rightChild": {
        "id": 2,
        "score": 0,
        "recordCount": 5,
        "predicate": {
            "type": "continuous",
            "field": "X4",
            "operator": "greaterOrEqual",
            "splitValue": 36.5
        },
        "scoreDistribution": [
            {
                "category": "0",
                "recordCount": 5
            },
            {
                "category": "1",
                "recordCount": 0
            }
        ]
    },
    "successClass": "1"
}
},
{
    "modelName": "",
    "objectType": "fittedModel",
    "type": "classification",
    "algorithm": "decisionTree",
    "variables": [
        "X1",
        "X2",
        "X3",
        "X4",
        "Weights"
    ],
    "targetColumn": "Y",
    "categoricalVariables": {},
    "fittedModel": {
        "classes": {
            "objectType": "stringVector",
            "name": "StringVector",
            "data": [
                "0",
                "1"
            ]
        },
        "priorProb": {
            "0": 0.61538461538461542,
            "1": 0.38461538461538464
        },
        "root": {
            "id": 0,
            "score": 0,

```

```

        "recordCount": 13,
        "predicate": null,
        "scoreDistribution": [
            {
                "category": "0",
                "recordCount": 8
            },
            {
                "category": "1",
                "recordCount": 5
            }
        ],
        "leftChild": {
            "id": 1,
            "score": 1,
            "recordCount": 5,
            "predicate": {
                "type": "continuous",
                "field": "X4",
                "operator": "less",
                "splitValue": 30
            },
            "scoreDistribution": [
                {
                    "category": "0",
                    "recordCount": 0
                },
                {
                    "category": "1",
                    "recordCount": 5
                }
            ]
        },
        "rightChild": {
            "id": 2,
            "score": 0,
            "recordCount": 8,
            "predicate": {
                "type": "continuous",
                "field": "X4",
                "operator": "greaterOrEqual",
                "splitValue": 30
            },
            "scoreDistribution": [
                {
                    "category": "0",
                    "recordCount": 8
                },
                {
                    "category": "1",
                    "recordCount": 0
                }
            ]
        },
        "successClass": "1"
    }
}

```

Example Request:

<https://rason.net/api/model/BaggingClassification?kind=all>

Example Response:

```

{
    "modelName": "BaggingClassification",
    "modelDescription": "classification example through bagging",

```

```

"modelType": "datamining",
"datasources": {
  "myTrainSrc": {
    "type": "csv",
    "connection": "hald-small-binary.txt",
    "direction": "import"
  },
  "myPMMLSrc": {
    "type": "xml",
    "content": "pmml-model",
    "connection": "classification-bagging.xml",
    "direction": "export"
  },
  "myJSONSrc": {
    "type": "json",
    "content": "json-model",
    "connection": "classification-bagging.json",
    "direction": "export"
  }
},
"datasets": {
  "myTrainData": {
    "binding": "myTrainSrc",
    "targetCol": "Y"
  }
},
"weakLearner": {
  "treeWeakLearner": {
    "type": "classification",
    "algorithm": "decisionTree",
    "parameters": {
      "minNumRecordsInLeaves": 2
    }
  }
},
"estimator": {
  "baggingEstimator": {
    "type": "classification",
    "algorithm": "bagging",
    "parameters": {
      "numWeakLearners": 2,
      "bootstrapSeed": 10
    }
  }
},
"actions": {
  "baggingModel": {
    "data": "myTrainData",
    "estimator": "baggingEstimator",
    "binding": "myJSONSrc",
    "action": "fit",
    "evaluations": [
      "weakLearnerModels",
      "numWeakLearners"
    ]
  },
  "trainScore": {
    "data": "myTrainData",
    "fittedModel": "baggingModel",
    "action": "predict",
    "parameters": {
      "successClass": "1",
      "successProbability": 0.6
    },
    "evaluations": [
      "prediction",
      "posteriorProbability",
      "confusionMatrix",
      "accuracy",
      "specificity",
      "sensitivity",

```

```

        "recall",
        "precision",
        "f1"
    ]
}
}
}

```

- [https://rason.net/api/model/{modelid}?return={ \"model\" or \"info\"}](https://rason.net/api/model/{modelid}?return={\)

Example Request:

<https://rason.net/api/model/BoxFunctionAirport?return=info>

Example Response:

(Two versions of the BoxFunctionAirport RASON model reside on the RASON server.)

```

[
  {
    "ModelId": "2590+BoxFunctionAirport+2021-04-05-18-12-08-857708",
    "ModelName": "BoxFunctionAirport",
    "ModelDescr": "",
    "ModelFiles": [],
    "RuntimeToken": "",
    "ModelType": "Instance",
    "ModelKind": "Rason",
    "IsChampion": false,
    "ParentModelId": "2590+BoxFunctionAirport+2021-04-05-17-12-38-168966",
    "QueryString": "E14:E19=%5B8,3,7,2,6,4%5D&F14:F19=%5B7,2,9,6,5,1%5D",
    "ModelContainer": null
  },
  {
    "ModelId": "2590+BoxFunctionAirport+2021-04-05-18-08-52-847519",
    "ModelName": "BoxFunctionAirport",
    "ModelDescr": "",
    "ModelFiles": [],
    "RuntimeToken": "",
    "ModelType": "Instance",
    "ModelKind": "Rason",
    "IsChampion": false,
    "ParentModelId": "2590+BoxFunctionAirport+2021-04-05-17-12-38-168966",
    "QueryString": "E14:E19=%5B8,3,7,2,6,4%5D&F14:F19=%5B7,2,9,6,5,1%5D",
    "ModelContainer": null
  }
]

```

Example Request:

<https://rason.net/api/model/BoxFunctionAirport?return=model>

Example Response:

```

{
  "comment": "This model has been generated by Psi from an Excel model in the workbook BoxFunctionAirport.xlsx",
  "modelName": "BoxFunctionAirport",

```

```

"modelDescription": "",
"inputParameters": {
  "E14:E19": {
    "type": "array",
    "value": [6, 5, 4, 3, 3, 1]
  },
  "F14:F19": {
    "type": "array",
    "value": [2, 5, 8, 4, 3, 2]
  }
},
"modelSettings": {
  "worksheets": [
    "Original_Formulation",
    "Box_Function",
    "Lambda_Function"
  ],
  "activeSheet": "Box_Function"
},
"engineSettings": {
  "engine": "GRG Nonlinear",
  "scaling": -1
},
"boxFunctions": {
  "funDistance": {
    "inputs": ["Xone", "Xtwo", "Yone", "Ytwo"],
    "inputTypes": ["number", "number", "number", "number"],
    "language": "EXCEL",
    "resultType": "number",
    "body": {
      "formula1": {
        "formula": "(Xtwo-Xone)^2"
      },
      "formula2": {
        "formula": "(Ytwo-Yone)^2"
      }
    }
  },
  "result": "SQRT(Formula1+Formula2)"
},
"variables": {
  "e13:f13": {
    "value": 1,
    "finalValue": []
  },
  "g20": {
    "value": 1,
    "finalValue": []
  }
},
"data": {
  "e14": {
    "value": 6
  },
  "f14": {
    "value": 2
  },

```

```

        "e15": {
            "value": 5
        },
        "f15": {
            "value": 5
        },
        "e16": {
            "value": 4
        },
        "f16": {
            "value": 8
        },
        "e17": {
            "value": 3
        },
        "f17": {
            "value": 4
        },
        "e18": {
            "value": 3
        },
        "f18": {
            "value": 3
        },
        "e19": {
            "value": 1
        },
        "f19": {
            "value": 2
        }
    },
    "formulas": {
        "g14": {
            "formula": "funDistance($E$13,E14,$F$13,F14)"
        },
        "g15": {
            "formula": "funDistance($E$13,E15,$F$13,F15)"
        },
        "g16": {
            "formula": "funDistance($E$13,E16,$F$13,F16)"
        },
        "g17": {
            "formula": "funDistance($E$13,E17,$F$13,F17)"
        },
        "g18": {
            "formula": "funDistance($E$13,E18,$F$13,F18)"
        },
        "g19": {
            "formula": "funDistance($E$13,E19,$F$13,F19)"
        }
    },
    "constraints": {
        "g14_": {
            "formula": "g14 - g20",
            "upper": 0
        },
        "g15_": {

```



```

        "formula": "g15 - g20",
        "upper": 0
    },
    "g16_": {
        "formula": "g16 - g20",
        "upper": 0
    },
    "g17_": {
        "formula": "g17 - g20",
        "upper": 0
    },
    "g18_": {
        "formula": "g18 - g20",
        "upper": 0
    },
    "g19_": {
        "formula": "g19 - g20",
        "upper": 0
    }
},
"objective": {
    "g20_": {
        "formula": "g20",
        "type": "min",
        "finalValue": []
    }
}
}

```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.

[Return to Asynchronous Endpoint List](#)

REST Endpoint: GET:
https://rason.net/api/model/{name}/champion

A REST API endpoint that retrieves a list of resource IDs for only the champion versions of the named model, marked by using PATCH rason.net/api/model/{nameorid}. If no champions exist, the most recent version of the model is returned.

- If {name} is a valid model name, this endpoint returns the champion or the most recent version of a named model.
- If {name}="fitted", this endpoint returns all champions or the most recent versions of all ModelKind=Fitted models on the user's account.
- If {name} = "excel", this endpoint returns all champions or most recent versions of all ModelKind=Excel models on the user's account.
- If {name} = "all", this endpoint returns all champions or most recent versions of all models on the user's account.

URL

https://rason.net/api/model/{name}/champion

https://rason.net/api/model/fitted/champion

https://rason.net/api/model/RASON/champion

https://rason.net/api/model/excel/champion

https://rason.net/api/model/all/champion

- **Method:**
GET
- **URL Params**
Required: None
Optional: None
- **Headers**
Required: Authorization - Example: Authorization: bearer {your RASON token}
Optional: None
- **Data Params:** None
- **Success Response:**
Code: 200 (OK)

Example Request

<https://rason.net/api/model/StratifiedSampling/champion>

Notice that only the champion version of the StratifiedSampling model was returned.

Example Response

```
[
  {
    "ModelId": "2590+StratifiedSampling+2020-01-04-06-50-23-416510",
    "ModelName": "StratifiedSampling",
    "ModelDescr": "transformation: stratified sampling",
    "ModelFiles": [
      "hald-small-binary.txt"
    ],
    "RuntimeToken": "",
    "ModelType": 0,
    "ModelKind": "Origin",
    "IsChampion": true,
    "ParentModelId": null,
    "QueryString": "?modelID=2590%2bStratifiedSampling%2b2020-01-04-06-50-23-416510&_cb=1578120629964"
  }
]
```

Example Request

<https://rason.net/api/model/fitted/champion> - Notice that the first returned fitted model (fitted-myjsonsrc) has *not* been marked as a champion, so the most recent version has been returned. The second returned fitted model (LogisticRegressionFitted1) has been marked as the champion.

Example Response

```
[
  {
    "ModelId": "2590+fitted-myjsonsrc+2020-06-02-21-10-57-104562",
    "ModelName": "fitted-myjsonsrc",
    "ModelDescr": "classification-logistic-model.json",
    "ModelFiles": [],
    "RuntimeToken": "",
    "ModelType": "Origin",
    "ModelKind": "Fitted",
    "IsChampion": false,
    "ParentModelId": "2590+LogisticRegression+2020-05-29-15-56-23-566547",
    "QueryString": ""
  },
  {
    "ModelId": "2590+LogisticRegressionFitted1+2020-06-01-21-25-05-231970",
    "ModelName": "LogisticRegressionFitted1",
    "ModelDescr": "",
    "ModelFiles": [],
    "RuntimeToken": "",
    "ModelType": "Origin",
    "ModelKind": "Fitted",
    "IsChampion": true,
    "ParentModelId": null,
    "QueryString": ""
  }
]
```

Example Request

<https://rason.net/api/model/RASON/champion> - Notice that out of the two RASON models on the User's account, the first one is simply the most recent version of the Sampling model and the 2nd is the actual champion version of the Logistic Regression model.

Example Response

```
[
```

```

{
  "ModelId": "2590+Sampling+2020-05-29-16-32-56-004293",
  "ModelName": "Sampling",
  "ModelDescr": "",
  "ModelFiles": [
    {
      "fileName": "Sampling.xlsx",
      "isOnServer": true
    }
  ],
  "RuntimeToken": "",
  "ModelType": "Version",
  "ModelKind": "RASON",
  "IsChampion": false,
  "ParentModelId": null,
  "QueryString": ""
},
{
  "ModelId": "2590+LogisticRegression+2020-05-29-15-56-23-566547",
  "ModelName": "LogisticRegression",

  "ModelDescr": "classification: logistic regression; scoring example PMM
LClass
ifier.json uses exported fitted model, lrModel, to score new data",
  "ModelFiles": [
    {
      "fileName": "hald-small-binary-train.txt",
      "isOnServer": true
    },
    {
      "fileName": "hald-small-binary-valid.txt",
      "isOnServer": true
    },
    {
      "fileName": "classification-logistic-model.xml",
      "isOnServer": false
    },
    {
      "fileName": "classification-logistic-model.json",
      "isOnServer": true
    }
  ],
  "RuntimeToken": "",
  "ModelType": "Version",
  "ModelKind": "RASON",
  "IsChampion": true,
  "ParentModelId": "2590+LogisticRegression+2020-05-28-21-04-19-737715",
  "QueryString": ""
}
]

```

Example Request

<https://rason.net/api/model/excel/champion> - Notice that out of the two Excel models on the User's account, the first one has been marked as the champion (Blending(Opt)) and the second is simply the most recent version of the 2StageFarmer1(Stochastic) model.

Example Response

```

[
  {
    "ModelId": "2590+Blending (Opt)+2020-06-01-21-23-30-714655",
    "ModelName": "Blending (Opt)",
    "ModelDescr": "",
    "ModelFiles": [],
    "RuntimeToken": "",
    "ModelType": "Origin",
    "ModelKind": "Excel",
    "IsChampion": true,
    "ParentModelId": null,
    "QueryString": ""
  },

```

```

    {
      "ModelId": "2590+2StageFarmer1 (Stochastic)+2020-05-26-21-46-10-180902",
      "ModelName": "2StageFarmer1 (Stochastic)",
      "ModelDescr": "",
      "ModelFiles": [],
      "RuntimeToken": "",
      "ModelType": "Origin",
      "ModelKind": "Excel",
      "IsChampion": false,
      "ParentModelId": null,
      "QueryString": ""
    }
  ]

```

Example Request

<https://rason.net/api/model/all/champion> - All models on the User's account are marked as champions, except for the last model, 2StageFarmer1(Stochastic) which is the most recent version.

Example Response

```

[
  {
    "ModelId": "2590+StratifiedSampling+2020-01-04-06-50-23-416510",
    "ModelName": "StratifiedSampling",
    "ModelDescr": "transformation: stratified sampling",
    "ModelFiles": [
      "hald-small-binary.txt"
    ],
    "RuntimeToken": "",
    "ModelType": 0,
    "ModelKind": "Origin",
    "IsChampion": true,
    "ParentModelId": null,
    "QueryString": "?modelID=2590%2bStratifiedSampling%2b2020-01-04-06-50-23-416510&_cb=1578120629964"
  },
  {
    "ModelId": "2590+LogisticRegressionFitted1+2020-06-01-21-25-05-231970",
    "ModelName": "LogisticRegressionFitted1",
    "ModelDescr": "",
    "ModelFiles": [],
    "RuntimeToken": "",
    "ModelType": "Origin",
    "ModelKind": "Fitted",
    "IsChampion": true,
    "ParentModelId": null,
    "QueryString": ""
  },
  {
    "ModelId": "2590+LogisticRegression+2020-05-29-15-56-23-566547",
    "ModelName": "LogisticRegression",
    "ModelDescr": "classification: logistic regression; scoring example PMMLClass  
ifier.json uses exported fitted model, lrModel, to score new data",
    "ModelFiles": [
      {
        "fileName": "hald-small-binary-train.txt",
        "isOnServer": true
      },
      {
        "fileName": "hald-small-binary-valid.txt",
        "isOnServer": true
      },
      {
        "fileName": "classification-logistic-model.xml",
        "isOnServer": false
      }
    ]
  }
]

```

```

    },
    {
        "fileName": "classification-logistic-model.json",
        "isOnServer": true
    }
],
"RuntimeToken": "",
"ModelType": "Version",
"ModelKind": "RASON",
"IsChampion": true,
"ParentModelId": "2590+LogisticRegression+2020-05-28-21-04-19-737715",
"QueryString": ""
},
{
    "ModelId": "2590+Blending (Opt)+2020-06-01-21-23-30-714655",
    "ModelName": "Blending (Opt) ",
    "ModelDescr": "",
    "ModelFiles": [],
    "RuntimeToken": "",
    "ModelType": "Origin",
    "ModelKind": "Excel",
    "IsChampion": true,
    "ParentModelId": null,
    "QueryString": ""
},
{
    "ModelId": "2590+2StageFarmer1 (Stochastic)+2020-05-26-21-46-10-180902",
    "ModelName": "2StageFarmer1 (Stochastic) ",
    "ModelDescr": "",
    "ModelFiles": [],
    "RuntimeToken": "",
    "ModelType": "Origin",
    "ModelKind": "Excel",
    "IsChampion": false,
    "ParentModelId": null,
    "QueryString": ""
}
]

```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

[Return to Asynchronous Endpoint List](#)

REST Endpoint: *GET*

https://rason.net/api/model/{nameorid}/datamine

A REST API endpoint that enqueues the previously submitted data science model for solving.

- **URL**

`https://rason.net/api/model/{nameorid}/datamine`

- **Method:**

GET

- **URL Params**

Required: None

- **Optional:** Any datasource component may be passed as a query parameter if a binding property exists for that datasource. For example, see the RASON example code snippet below.

```
"datasources": {
  "srcCustomers": {
    "type": "csv",
    "connection": "customers_dt.txt",
    "selection": "custID = $parCustID1 or custID =
$parCustID2",
    "parameters": {
      "parCustID1": {
        "binding": "get",
        "value": "c1"
      },
      "parCustID2": {
        "binding": "get",
        "value": "c3"
      }
    }
  }
},
```

To query `custID=$parCustID1 or custID=$parCustID2` outside of the RASON model environment, use:

```
$.get(https://rason.net/api/datamine?parCustID1=c1&parCustID2=c2...
```

Or in general,

```
$.get(https://rason.net/api/datamine?par1=val1&par2=val2.....
```

RASON Server will map "custID=\$parCustID1" with "parCustID1=c1" and "custID=\$parCustID2" with "parCustID2=c3". A query can return any number of rows that satisfy the filtering condition, from 0 to infinity.

- **Headers**

Required: Authorization - Example: Authorization: bearer {your RASON token}

Optional: None

- **Data Params:** None

- **Success Response:**

Code: 202 (Accepted)

Example Request:

`https://rason.net/api/model/LogisticRegression/datamine`

Response Example:

```
{
  "Accepted": "2020-06-02+21-10-55"
}
```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

[Return to Asynchronous Endpoint List](#)

REST Endpoint: GET:
https://rason.net/api/model/{nameorid}/decision

A REST API endpoint that enqueues the previously submitted decision table model for solving.

- **URL**

https://rason.net/api/model/{nameorid}/decision

- **Method:**

GET

- **URL Params**

Required: None

Optional: Any data component may be passed as a query parameter if a binding property exists for that data component. See the previous topic "Parametric Selection Feature" within the chapter **Defining Decision Tables in RASON** for a complete example illustrating how to pass a query parameter from outside or inside your RASON model.

- **Headers**

- **Required:** Authorization - Example: Authorization: bearer {your RASON token}

- **Optional:** None

- **Data Params:**

- **Success Response:**

Code: 202 (Accepted)

Response Example:

```
{
  "Accepted": "2019-11-19+18-53-16"
}
```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.

Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

[Return to Asynchronous Endpoint List](#)

REST Endpoint:

GET:https://rason.net/api/model/{nameorid}/diagnose

A REST API endpoint that enqueues the previously submitted optimization, stochastic optimization or simulation optimization model for diagnosing.

- **URL**

https://rason.net/api/model/{nameorid}/diagnose

- **Method:**

GET

- **URL Params**

Required: None

Optional: None

- **Headers**

Required: Authorization - Example: Authorization: bearer {your RASON token}

Optional: None

- **Data Params:** None

- **Success Response:**

Code: 202 (Accepted)

Response Example:

```
{
  "Accepted": "2019-11-19+18-56-26"
}
```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

[Return to Asynchronous Endpoint List](#)

REST Endpoint: GET:

<https://rason.net/api/model/{nameorid}/fitted-info>

The GET <https://rason.net/api/model/{name}/fitted-info> endpoint returns information pertaining to a fitted model that has been POSTed to the RASON server. Information returned includes 1. the *name* of the fitted model, 2. the *format* of the fitted model (JSON or PMML), 3. the model *type* (classification, regression, etc.), 4. the *algorithm* used to fit the model, the included *features* (input variables) and 5. the *output variable* (target variable). This endpoint is intended to help the user understand his/her fitted model and determine what type of data can be scored. See the related API endpoint: POST <https://rason.net/api/model/fitted-info>.

If a fitted model is opened on the Editor tab of www.RASON.com, the information from this endpoint is displayed on the Properties pane, on the right.

The screenshot shows the RASON Editor interface. The main editor displays a JSON object representing a fitted model. The right-hand pane, titled 'Properties', shows the model's details. The 'Variables' section lists the input variables and their types.

```
26      "estimate": 5.0316693922277373
27    }, {
28      "name": "X3",
29      "categorical": false,
30      "estimate": 2.5501718637097612
31    }, {
32      "name": "X4",
33      "categorical": false,
34      "estimate": 3.8639376929625953
35    }, {
36      "name": "Weights",
37      "categorical": false,
38      "estimate": 6.854417131311874
39    }],
40    "encoder": {
41      "modelName": "",
42      "objectType": "fittedModel",
43      "type": "transformation",
44      "algorithm": "oneHotEncoding",
45      "variables": ["X1", "X2", "X3", "X4", "Weights"],
46      "categoricalVariables": {},
47      "fittedModel": {
48        "mapping": null
49      }
50    },
51    "fitIntercept": true,
52    "priorProb": {
53      "0": 0.46153846153846156,
54      "1": 0.53846153846153844
55    },
56    "successClass": "1"
57  }
58 }
```

Properties

Name: Logistic-Regression

Version: 2021-04-30-18-21-14-680873

Created by: Logistic-Regression_872793

Type: Classification

Algorithm: Logistic-Regression

Variables:

X1	Continuous
X2	Continuous
X3	Continuous
X4	Continuous
Weights	Continuous

- **URL**

<https://rason.net/api/model/{nameorid}/fitted-info>

- **Method:**

GET

- **URL Params**

Required: None

Optional: None

- **Headers**

Required: Authorization - Example: Authorization: bearer {your RASON token}

Optional: None

- **Data Params:** None

- **Success Response:**

Code: 200 (OK)

- <https://rason.net/api/model/{nameorid}/fitted-info>

Example Request

<https://rason.net/api/model/LogisticRegression/fitted-info>

Example Response

In the example response below, the model name is "LogisticRegression", the format is JSON, the type of model is "classification", the algorithm used is to fit the model is "logistic regression", the included features are "x1", "x2", "x3", "x4" and "weights" and the output or target variable is "Y".

```
{
  "modelName": "LogisticRegression",
  "format": "JSON",
  "type": "classification",
  "algorithm": "logisticRegression",
  "features": [
    {
      "name": "X1",
      "type": "continuous"
    },
    {
      "name": "X2",
      "type": "continuous"
    },
    {
      "name": "X3",
      "type": "continuous"
    },
    {
      "name": "X4",
      "type": "continuous"
    },
    {
      "name": "Weights",
      "type": "continuous"
    }
  ],
  "targetName": "Y"
}
```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.

[Return to Asynchronous Endpoint List](#)

REST Endpoint: GET:

<https://rason.net/api/model/{nameorid}/interface>

A REST API endpoint that generates an interface from an already posted reusable RASON model, that can be utilized in a decision flow. This endpoint accepts models defined in either a RASON script or an Excel workbook. If a model is defined within an Excel workbook, you can use the REST API endpoint *GET rason.net/api/model/{nameorid}/interface* on a model previously posted using POST *rason.net/api/model*. An optional query parameter exists, *?worksheet=<name>*, which can be used when an Excel workbook contains multiple worksheets. If this query parameter is not present, the active sheet (or the entire workbook) is considered.

See below for an example of how to create a decision flow using this REST API endpoint.

- **URL**

`https://rason.net/api/model/{nameorid}/interface`

- **Method:**

GET

- **URL Params**

Required: None

Optional: *?worksheet=<name>* where *<name>* is the name of the worksheet

- **Headers**

Required: Authorization - Example: `Authorization: bearer {your RASON token}`

Optional: None

- **Data Params:**

A reusable model.

In this example, a reusable model that partitions the `hald-small.txt` dataset into three partitions: training, validation and test.

```
{
  "modelName": "partitioning-reusable",
  "modelType": "datamining",
  "modelDescription": "transformation: partitioning, independent on
input parameters",
  "datasources": {
    "dataToPartitionSource": {
      "type": "csv",
      "connection": "hald-small.txt"
    }
  },
  "datasets": {
    "dataToPartition": {
      "binding": "dataToPartitionSource"
    }
  },
  "transformer": {
```



```

    "myPartitioner": {
      "type": "transformation",
      "algorithm": "partitioning",
      "parameters": {
        "partitionMethod": "RANDOM",
        "ratios": [
          [
            "Training",
            0.5
          ],
          [
            "Validation",
            0.3
          ],
          [
            "Testing",
            0.2
          ]
        ],
        "seed": 123
      }
    },
    "actions": {
      "trainingPartition": {
        "data": "dataToPartition",
        "action": "transform",
        "parameters": {
          "partition": "Training"
        },
        "evaluations": [
          "transformation"
        ]
      },
      "validationPartition": {
        "data": "dataToPartition",
        "action": "transform",
        "parameters": {
          "partition": "Validation"
        },
        "evaluations": [
          "transformation"
        ]
      },
      "testPartition": {
        "data": "dataToPartition",
        "action": "transform",
        "parameters": {
          "partition": "Testing"
        },
        "evaluations": [
          "transformation"
        ]
      }
    }
  }
}

```

- **Success Response:**

Code: 200 (OK)

Response Request: GET rason.net/api/model/partitioning-reusable/interface

Response Example:

Returned results from this REST API endpoint can be copied and pasted into a decision flow. For more information on decision flows and how this REST API Endpoint can help you create a decision flow quickly and easily, see the example below that appears earlier in this guide.

```

{
  "comment": "This interface has been generated by Psi for a RASON model",
  "invokeModel": "partitioning-reusable",
  "modelType": "datamining",
  "modelDescription": "transformation: partitioning, independent on i
nput parameters",
  "datasources": {
    "datatopartitionsources": {
      "type": "csv",
      "connection": "hald-small.txt",
      "direction": "import"
    }
  },
  "outputResults": {
    "trainingPartition": {
      "evaluations": [
        "transformation"
      ],
      "type": "dataset",
      "comment": "datamining evaluation"
    },
    "validationPartition": {
      "evaluations": [
        "transformation"
      ],
      "type": "dataset",
      "comment": "datamining evaluation"
    },
    "testPartition": {
      "evaluations": [
        "transformation"
      ],
      "type": "dataset",
      "comment": "datamining evaluation"
    }
  }
}

```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

Creating a Decision Flow using GET rason.net/api/model/{nameorid}/interface

In the previous chapter, Creating and Solving Decision Flows, a decision flow was created using the Decision flow editor on www.RASON.com. Users can also create a decision flow outside of the Decision flow editor by using the REST API endpoint, GET rason.net/api/interface. To start, each reusable model, that is to be included in the decision flow, must first be posted (separately) using the REST API endpoint POST rason.net/api/model. Use the REST API endpoint GET rason.net/api/model/interface to retrieve the interface in return.

Interface for optStageReusable

```

{

```

```

    "comment": "This interface has been generated by Psi for a RASON model",
    "invokeModel": "optStageReusable",
    "modelType": "optimization",
    "modelDescription": "Project Select converted into a reusable model",
    "outputResults": {
      "x": {
        "evaluations": [
          "initialValue",
          "finalValue",
          "dualValue"
        ],
        "type": "array/number",
        "comment": "decision variable block"
      },
      "totalObjective": {
        "evaluations": [
          "finalValue"
        ],
        "type": "number",
        "comment": "objective function"
      }
    }
  }
}

```

Interface for simStageReusable

```

{
  "comment": "This interface has been generated by Psi for a RASON model",
  "invokeModel": "simStageReusable",
  "modelType": "simulation",
  "modelDescription": "Project Select converted into a reusable model",
  "inputParameters": {
    "finalVarValues": {
      "value": null,
      "type": "array",
      "comment": "pass the final variable values from optStage"
    }
  },
  "outputResults": {
    "cash": {
      "evaluations": [
        "mean"
      ],
      "type": "number",
      "comment": "uncertain function block"
    }
  }
}

```

Combine the two returned interfaces to create the decision flow. You'll need to add the *flowName* property and optionally the *flowDescription* property along with a name for each stage. (See items in red.)

Combined to form a Decision Flow

```

{
  "flowName": "OptSimReusable",
  "flowDescription": "Opt Sim Decision Flow",
  "optStageReusable": {
    "comment": "This interface has been generated by Psi for a RASON model",
    "invokeModel": "optStageReusable",
    "modelType": "optimization",
    "modelDescription": "Project Select converted into a reusable model",
    "outputResults": {
      "x": {
        "evaluations": [
          "initialValue",
          "finalValue",
          "dualValue"
        ],
        "type": "array/number",

```

```

        "comment": "decision variable block"
    },
    "totalObjective": {
        "evaluations": [
            "finalValue"
        ],
        "type": "number",
        "comment": "objective function"
    }
}
},
"simStageReusable": {
    "comment": "This interface has been generated by Psi for a RASON model",
    "invokeModel": "simStageReusable",
    "modelType": "simulation",
    "modelDescription": "Project Select converted into a reusable model",
    "inputParameters": {
        "finalVarValues": {
            "value": optStageReusable.x.finalValue,
            "type": "array",
            "comment": "pass the final variable values from optStage"
        }
    },
    "outputResults": {
        "cash": {
            "evaluations": [
                "mean"
            ],
            "type": "number",
            "comment": "uncertain function block"
        }
    }
}
}
}

```

[Return to Asynchronous Endpoint List](#)

REST Endpoint: GET:
https://rason.net/api/model/{nameorid}/optimize

A REST API endpoint that enqueues the previously submitted optimization model for solving.

- **URL**

https://rason.net/api/model/{nameorid}/optimize

- **Method:**

GET

- **URL Params**

Required: None

- **Optional:** Any data component may be passed as a query parameter if a binding property exists for that parameter. For example, see the RASON example code snippet below.

```
data: {  
  profits: {  
    value: [[75, 50, 35]],  
    binding: 'get'  
  }  
}
```

To change a query parameter outside of the RASON model environment, use:

```
$.Post("https://rason.net/api/optimize?profits=100,150,75", ...
```

RASON Server will map "profits=?" with "profits = 100, 150, 75". A query can return any number of rows that satisfy the filtering condition, from 0 to infinity.

To perform a query using multiple parameters (say custID, maritalStatus and age) outside of the RASON model environment, use the query parameter:

```
$.get(https://rason.net/api/decision?custID=c2&maritalStatus=s&age=40).....
```

Or in general,

```
$.get(https://rason.net/api/decision?par1=val1&par2=val2).....
```

- **Headers**

Required: Authorization - Example: Authorization: bearer {your RASON token}

Optional: None

- **Data Params:** None

- **Success Response:**

Code: 202 (Accepted)

Response Example:

```
{  
  "Accepted": "2019-11-19+18-58-25"  
}
```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

[Return to Asynchronous Endpoint List](#)

REST Endpoint: GET:

https://rason.net/api/model/{nameorid}/result

A REST API endpoint that retrieves the results from the model previously submitted to GET/POST
rason.net/api/model/{nameorid}/optimize, GET/POST
rason.net/api/model/{nameorid}/simulate,
GET/POST rason.net/api/model/{nameorid}/decision, GET/POST
rason.net/api/model/{nameorid}/datamine, GET/POST
rason.net/api/model/{nameorid}/diagnose or GET/POST
rason.net/api/model/{nameorid}/solve.

Use GET rason.net/api/model/{nameorid}/result/fitted-{name} to retrieve the text of the fittedModel.

Note: If solving a hard mixed integer model or nonsmooth model, calling GET
rason.net/api/model/id/result will return the best solution found so far
and the number of subproblems completed.

- **URL**

https://rason.net/api/model/{nameorid}/result

- **Method:**

GET

- **URL Params:** None

Required: None

Optional: None

- **Headers**

Required: Authorization - Example: Authorization: bearer {your RASON token}

Optional: None

- **Data Params:** None

- **Success Response:**

Code: 200 (OK)

Response Example:

Results from a previously solved optimization model.

```
{
  "status": {
    "code": 0,
    "id": "2590+2019-11-19-17-56-14-563925",
    "codeText": "Solver found a solution. All constraints and optimality conditions are satisfied."
  },
  "variables": {
```

```

        "d9:f9": {
            "finalValue": [
                [
                    200,
                    200,
                    0
                ]
            ]
        },
        "objective": {
            "d18": {
                "finalValue": 25000
            }
        }
    }
}

```

Response Example:

Results from a previously solved decision flow using POST
rason.net/api/model/{nameorid}/solve.

Notes:

1. Overall flow "status" contains the id of the actual model instance that was created after calling POST rason.net/api/model/{nameorid}/solve.
2. Overall flow "status" shows "solveTimestamp" indicating the moment in time when the entire flow was solved.
3. The "status" object for a decision flow *stage* contains the id that corresponds to a model on the server:
 - a. For reusable stages, the id refers to origin/version resource id of the RASON/excel reusable model
 - b. For inline stages, the id refers to the resource id of the enclosing flow model instance.
4. The "status" object for a decision flow *stage* displays "solveTimestamp" indicating the moment in time when the stage was solved.

```

{
    "status": {
        "id": "2590+mlrDF+2020-12-23-00-44-21-465964",
        "code": 0,
        "codeText": "Success",
        "solveTimestamp": "2020-12-23-00-44-31-738690",
        "solveTime": 3333
    },
    "results": {
        "mlr-reusable.myModel.anova": [],
        "mlr-reusable.myModel.coefficients": [],
        "mlr-reusable.myModel.detailedCoefficients": [],
        "mlr-reusable.myModel.detailedResiduals": [],
        "mlr-reusable.myModel.entranceTolerance": [],
        "mlr-reusable.myModel.influenceDiagnostics": [],
        "mlr-reusable.myModel.multicollinearityDiagnostics": [],
        "mlr-reusable.myModel.predictorScreeningInfo": [],
        "mlr-reusable.myModel.regressionSummary": [],
        "mlr-reusable.myModel.varianceCovariance": [],
        "mlr-reusable.newScore.prediction": [],
        "mlr-reusable.testScore.intervals": [],
        "mlr-reusable.testScore.mad": [],
        "mlr-reusable.testScore.mse": [],
        "mlr-reusable.testScore.prediction": [],
        "mlr-reusable.testScore.r2": [],
        "mlr-reusable.testScore.residuals": [],
        "mlr-reusable.testScore.rmse": [],
        "mlr-reusable.testScore.ss": [],
        "mlr-reusable.testScore.sse": [],
        "mlr-reusable.testScore.sst": [],
        "mlr-reusable.trainScore.intervals": [],
        "mlr-reusable.trainScore.mad": [],
        "mlr-reusable.trainScore.mse": [],
        "mlr-reusable.trainScore.prediction": [],
        "mlr-reusable.trainScore.r2": [],
        "mlr-reusable.trainScore.residuals": [],
        "mlr-reusable.trainScore.rmse": [],
        "mlr-reusable.trainScore.ss": [],
        "mlr-reusable.trainScore.sse": [],
    }
}

```



```

"mlr-reusable.trainScore.sst": [],
"mlr-reusable.validScore.intervals": [],
"mlr-reusable.validScore.mad": [],
"mlr-reusable.validScore.mse": [],
"mlr-reusable.validScore.prediction": [],
"mlr-reusable.validScore.r2": [],
"mlr-reusable.validScore.residuals": [],
"mlr-reusable.validScore.rmse": [],
"mlr-reusable.validScore.ss": [],
"mlr-reusable.validScore.sse": [],
"mlr-reusable.validScore.sst": []
},
"mlr-reusable": {
  "status": {
    "id": "2590+mlr-reusable+2020-12-22-17-17-28-531289",
    "code": 0,
    "codeText": "Success",
    "solveTimestamp": "2020-12-23-00-44-31-726603",
    "solveTime": 807
  },
  "myModel": {
    "anova": {
      "objectType": "dataFrame",
      "name": "ANOVA",
      "order": "col",
      "rowNames": ["Regression", "Error", "Total"],
      "colNames": ["DF", "SS", "MS", "F-Statistic", "P-Value"],
      "colTypes": ["double", "double", "double", "double", "double"],
      "indexCols": null,
      "data": [
        [5, 1, 6],
        [1481.7379525361716, 21.239190320971829, 1502.9771428571435],
        [296.34759050723432, 21.239190320971829, 250.49619047619058],
        [13.9528666596399, null, null],
        [0.200386619063231, null, null]
      ]
    },
    "influenceDiagnostics": {
      "objectType": "dataFrame",
      "name": "Influence Diagnostics",
      "order": "col",
      "rowNames": ["Record 1", "Record 2", "Record 3", "Record 4",
        "Record 5", "Record 6", "Record 7"],
      "colNames": ["Cook's Distance", "DFITS", "Covariance Ratio",
        "Leverage", "Delete-1 Variance"],
      "colTypes": ["double", "double", "double", "double", "double"],
      "indexCols": null,
      "data": [
        [107.47518518527227, 51.424990138083416, 0.14188702913422824,
          1.0631545936513054, 2.913987270808915, 0.61818025245238661,
          2.9727210098095447],
        [null, null, null, null, null, null, null],
        [null, null, null, null, null, null, null],
        [0.99845165552432136, 0.99676950350136173,
          0.45984550198283475,
          0.86447895149935328, 0.94589893248338786, 0.787644364006993,
          0.94691109100174853],
        [null, null, null, null, null, null, null]
      ]
    },
    "detailedResiduals": {
      "objectType": "dataFrame",
      "name": "Residuals",
      "order": "col",
      "rowNames": ["Record 1", "Record 2", "Record 3", "Record 4",
        "Record 5", "Record 6", "Record 7"],
      "colNames": ["Raw", "Standardized", "Studentized", "Deleted"],
      "colTypes": ["double", "double", "double", "double"],
      "indexCols": null,
      "data": [
        [0.18134382537429872, 0.261941081096154, 3.3870996717127468, -

```

```

        1.6965722329459254, -1.071943501101245, -2.1237376882720014,
        1.0618688441359865],
    [0.039349008573019859, 0.056837456827683354,
    0.73495203790261954, -0.36813183576085,
    -0.23259636178714821, -0.46082061151062514,
    0.23041030575530949],
    [1.00000000000003062, 1.00000000000000773, 1.00000000000000446, -
    1.00000000000000406, -1.00000000000000535, -1.00000000000000457,
    1.00000000000000351],
    [null, null, null, null, null, null, null]
    ]
},
"coefficients": {
    "objectType": "dataFrame",
    "name": "Coefficients",
    "order": "col",
    "rowNames": ["Intercept", "X1", "X2", "X3", "X4", "Weights"],
    "colNames": ["Estimate"],
    "colTypes": ["double"],
    "indexCols": null,
    "data": [
        [-23.487082869309749, 149.01166572428048, 415.960477398573,
        121.07640892657103, 151.68899484535496, -7.9279668096112808]
    ]
},
"regressionSummary": {
    "objectType": "dataFrame",
    "name": "Regression Summary",
    "order": "col",
    "rowNames": ["Residual DF", "R2", "Adjusted R2", "Std. Error
    Estimate", "RSS"],
    "colNames": ["Value"],
    "colTypes": ["double"],
    "indexCols": null,
    "data": [
        [1, 0.98586858727565452, 0.91521152365392722,
        4.6085996051915625, 21.239190320971829]
    ]
},
"detailedCoefficients": {
    "objectType": "dataFrame",
    "name": "Coefficients",
    "order": "col",
    "rowNames": ["Intercept", "X1", "X2", "X3", "X4", "Weights"],
    "colNames": ["Estimate", "Confidence Interval: Lower", "Confidence
    Interval: Upper", "Standard Error", "T-Statistic", "P-Value"],
    "colTypes": ["double", "double", "double", "double", "double",
    "double"],
    "indexCols": null,
    "data": [
        [-23.487082869309749, 149.01166572428048, 415.960477398573,
        121.07640892657103, 151.68899484535496, -7.9279668096112808],
        [-9817.2912678486209, -6420.4203245593817, -30105.25625420154,
        9828.3505418376717, -21462.07830169308, -441.60206686215588],
        [9770.3171021100025, 6718.443656007943, 30937.177208998688,
        10070.503359690812, 21765.456291383787, 425.7461332429333],
        [770.78910566396416, 517.02551050357522, 2402.071851141035,
        783.03688295200607, 1701.0403771476965, 34.130891879765663],
        [-0.030471477472528336, 0.28820950358744446,
        0.17316737515615238,
        0.15462414550655598, 0.089174247056796482, -
        0.23228126699822157],
        [0.980607255587423, 0.82136128652147955, 0.89084075162002208,
        0.90233664276260372, 0.94337967667800116,
        0.85470175050792052]
    ]
},
"multicollinearityDiagnostics": {
    "objectType": "dataFrame",
    "name": "Multicollinearity Diagnostics",

```

```

"order": "col",
"rowNames": ["Eigenvalue", "Condition Number", "Intercept", "X1",
"X2", "X3", "X4", "Weights"],
"colNames": ["Component 1", "Component 2", "Component 3",
"Component 4", "Component 5", "Component 6"],
"colTypes": ["double", "double", "double", "double", "double",
"double"],
"indexCols": null,
"data": [
[3.6730071738696753E-06, 1180.0164678487108,
0.99997945561313217, 0.990267656034644, 0.9991944090799727, 0.99224894926150142,
0.999758767919465, 0.00084224958727100536],
[0.010435918746027308, 22.137756794176649,
1.7240620011455569E-05, 0.0088257997391910665, 0.00078674087194367207,
0.00731719974698259, 6.5640162959100873E-05, 0.12834187943550959],
[0.077015742133040577, 8.1490916865528664,
3.0578560755202531E-06, 0.00017935197320540931, 6.5721043606337055E-07,
0.00011897926922988472, 0.00012275090771918826, 0.83130061038308478],
[0.27182022378006571, 4.3376856294185409, 2.2986685440246014E-
09, 0.00046250118143391085, 8.0394468246562482E-06, 0.000300234112295231,
3.2232605484255242E-05, 3.7656279853642748E-05],
[0.52628650423904266, 3.1173663258103557, 4.9668359866317463E-
08, 0.000252242229389297, 9.4036357441602486E-06, 7.5579438000784511E-06,
1.9453368308914693E-05, 0.03617747936598411],
[5.1144379380946488, 1, 1.9394375241198394E-07,
1.2448842136433821E-05, 7.4975507874111751E-07, 7.0796661908484751E-06,
1.1550360633061476E-06, 0.0033001249482969112]
]
},
"varianceCovariance": {
"objectType": "dataFrame",
"name": "Variance-Covariance Matrix of Coefficients",
"order": "col",
"rowNames": ["Intercept", "X1", "X2", "X3", "X4", "Weights"],
"colNames": ["Intercept", "X1", "X2", "X3", "X4", "Weights"],
"colTypes": ["double", "double", "double", "double", "double",
"double"],
"indexCols": null,
"data": [
[594115.84541025374, -396732.72786361113, -1850505.4001221072,
-601408.49507572281, -1310903.2853242843, 760.22970426963764],
[-396732.72786361113, 267315.37851148256, 1232078.7696380834,
404340.33514438733, 874334.74061642145, -940.12886509650309],
[-1850505.4001221072, 1232078.7696380834, 5769949.1780441189,
1868436.9332559798, 4084730.3785933196, -1657.4586767497408],
[-601408.49507572281, 404340.33514438733, 1868436.9332559798,
613146.76006319362, 1325771.9443042604, -1836.5072377616627],
[-1310903.2853242843, 874334.74061642145, 4084730.3785933196,
1325771.9443042604, 2893538.3646867773, -2052.4309307878093],
[760.22970426963764, -940.12886509650309, -1657.4586767497408,
-1836.5072377616627, -2052.4309307878093, 1164.9177805082536]
]
},
"predictorScreeningInfo": {
"objectType": "dataFrame",
"name": "Predictor Screening",
"order": "col",
"rowNames": ["Intercept", "X1", "X2", "X3", "X4", "Weights"],
"colNames": ["Criteria", "Included"],
"colTypes": ["double", "wstring"],
"indexCols": null,
"data": [
[2.6457513110645907, 0.18910238671962837,
0.0019185935687154323,
0.051188877004303704, 0.21616064003500096,
0.14905171986625759],
["TRUE", "TRUE", "TRUE", "TRUE", "TRUE", "TRUE"]
]
},
"entranceTolerance": 4.1122383477614725E-15
},

```

```

"trainScore": {
  "prediction": {
    "objectType": "dataFrame",
    "name": "Scores: trainData",
    "order": "col",
    "rowNames": ["Record 1", "Record 2", "Record 3", "Record 4",
    "Record 5", "Record 6", "Record 7"],
    "colNames": ["Prediction: Y"],
    "colTypes": ["double"],
    "indexCols": null,
    "data": [
      [78.3186561746257, 113.03805891890384, 105.81290032828726,
      111.09657223294593, 94.171943501101239, 89.723737688272,
      73.238131155864011]
    ]
  },
  "residuals": {
    "objectType": "dataFrame",
    "name": "Residuals: trainData",
    "order": "col",
    "rowNames": ["Record 1", "Record 2", "Record 3", "Record 4",
    "Record 5", "Record 6", "Record 7"],
    "colNames": ["Value"],
    "colTypes": ["double"],
    "indexCols": null,
    "data": [
      [0.18134382537429872, 0.261941081096154, 3.3870996717127468, -
      1.6965722329459254, -1.071943501101245, -2.1237376882720014,
      1.0618688441359865]
    ]
  },
  "sse": 21.239190320973723,
  "ss": 64754,
  "sst": 1502.9771428571432,
  "mse": 3.0341700458533891,
  "rmse": 1.7418869210868395,
  "mad": 1.3977866920911939,
  "r2": 0.9858685872756533
},
"validScore": {
  "prediction": {
    "objectType": "dataFrame",
    "name": "Scores: validData",
    "order": "col",
    "rowNames": ["Record 1", "Record 2", "Record 3", "Record 4"],
    "colNames": ["Prediction: Y"],
    "colTypes": ["double"],
    "indexCols": null,
    "data": [
      [88.386069640329225, 103.28937334867146, 118.19379758053017,
      106.12656662832126]
    ]
  },
  "residuals": {
    "objectType": "dataFrame",
    "name": "Residuals: validData",
    "order": "col",
    "rowNames": ["Record 1", "Record 2", "Record 3", "Record 4"],
    "colNames": ["Value"],
    "colTypes": ["double"],
    "indexCols": null,
    "data": [
      [-4.586069640329228, 1.0106266513285362, -2.2937975805301676,
      3.4265666283212539]
    ]
  },
  "sse": 39.056267173095925,
  "ss": 41881.03,
  "sst": 529.80750000000023,
  "mse": 9.7640667932739813,

```

```

        "rmse": 3.1247506769779219,
        "mad": 2.8292651251272964,
        "r2": 0.92628215498441246
    },
    "testScore": {
        "prediction": {
            "objectType": "dataFrame",
            "name": "Scores: testData",
            "order": "col",
            "rowNames": ["Record 1", "Record 2"],
            "colNames": ["Prediction: Y"],
            "colTypes": ["double"],
            "indexCols": null,
            "data": [
                [93.148609395507151, 82.823761907438126]
            ]
        },
        "residuals": {
            "objectType": "dataFrame",
            "name": "Residuals: testData",
            "order": "col",
            "rowNames": ["Record 1", "Record 2"],
            "colNames": ["Value"],
            "colTypes": ["double"],
            "indexCols": null,
            "data": [
                [2.7513906044928547, -10.323761907438126]
            ]
        },
        "sse": 114.15021017996204,
        "ss": 14453.060000000001,
        "sst": 273.78000000000014,
        "mse": 57.075105089981022,
        "rmse": 7.5548067539799471,
        "mad": 6.53757625596549,
        "r2": 0.58305862305514644
    },
    "newScore": {
        "prediction": {
            "objectType": "dataFrame",
            "name": "Scores: newData",
            "order": "col",
            "rowNames": ["Record 1", "Record 2", "Record 3", "Record 4",
"Record 5", "Record 6", "Record 7", "Record 8", "Record 9", "Record 10", "Record 11",
"Record 12", "Record 13"],
            "colNames": ["Prediction: Y"],
            "colTypes": ["double"],
            "indexCols": null,
            "data": [
                [79.111452855586833, 75.616521198747392, 104.87496671059371,
91.309331050194245, 93.941406076468283, 106.60569700924839, 106.91936330928239,
84.409355269360375, 94.964740182062371, 118.98659426149131, 90.7644596832126,
113.83085559986498, 111.88936891390706]
            ]
        }
    }
}
}
}

```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.

Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

- Notes:

The chart below contains the available "code" and "codetext" responses for RASON for optimization and simulation. For a complete explanation of each Code, see the Solver Result Messages chapter in the RASON Reference Guide. For a list of RASON error codes, see Appendix II in the RASON Reference Guide.

Code	CodeText	Explanation
	"The remote server returned an error: (401) Unauthorized."	Returned if validation token is invalid.
0	Solver found a solution. All constraints and optimality conditions are satisfied.	This means that the Solver has found the optimal or "best" solution under the circumstances. The exact meaning depends on whether you are solving a linear or quadratic, smooth nonlinear, global optimization, or integer programming problem, as outlined in the RASON Reference Guide.
1	Solver has converged to the current solution. All constraints are satisfied	This means that Solver has found a series of "best" solutions that satisfy the constraints, and that have very similar objective function values.
2	Solver cannot improve the current solution. All constraints are satisfied.	This means that the Solver has found solutions that satisfy the constraints, but it has been unable to further improve the objective, even though the tests for optimality and convergence have not yet been satisfied.
3	Stop chosen when the maximum iteration limit was reached.	This result is returned when the Solver has completed the maximum number of iterations, or trial solutions, as specified for "iterations" in "engineSettings". The default setting for this option is unlimited.
4	The objective (Set Cell) values do not converge	This result is returned when the Solver is able to increase (if you are trying to Maximize) or decrease (for Minimize) without limit the value calculated by the objective, while still satisfying the constraints.
5	Solver could not find a feasible solution.	This result is returned when the Solver could not find any combination of values for the decision variables that allows all of the constraints to be satisfied simultaneously.
6	Solver stopped at user's request.	This result is returned only when the REST API endpoint, POST rason.net/api/model/id/stop is called.
7	The linearity conditions required by this Solver engine are not satisfied.	This result is returned if you've selected the LP/Quadratic Solver and the Solver's tests determine that the constraints are not linear functions of the variables or the objective is not a linear or convex quadratic function of the variables; or if you've selected the SOCP Barrier Solver and the Solver's tests determine that the constraints or the objective are not linear or convex quadratic functions of the variables.

8	{ "Exception": "You have _____ variables. Your license allows _____ variables." }	This result is returned when the Solver determines that your model is too large for the selected Solver engine within your Account tier.
9	Solver encountered an error value in a target or constraint cell.	This message appears when the Solver SDK Platform (on the RASON server or on your desktop) evaluates the formulas in your RASON model and discovers an error value while calculating the objective function, uncertain function or one of your constraints.
10	Stop chosen when the maximum time limit was reached.	This result is returned when Solver has run for the maximum time (number of seconds) specified for maxTime within engineSettings. The default setting for this option is unlimited.
11	There is not enough memory available to solve the problem.	This message appears when the Solver could not allocate the memory it needs to solve the problem.
12	No model inputs defined.	This message means that the internal "model" (information about the variable cells, objective, constraints, Solver options, etc.) is not in a valid form.
14	Solver found an integer solution within tolerance. All constraints are satisfied.	If you are solving a mixed-integer programming problem (any problem with integer constraints) with a non-zero value for the intTolerance within engineSettings, the Branch & Bound method has found a solution satisfying the constraints (including the integer constraints) where the relative difference of this solution's objective value from the true optimal objective value does not exceed the integer Tolerance setting.
15	Stop chosen when the maximum number of feasible [integer] solutions was reached	If you are using the Evolutionary Solver, this result is returned when the Solver has found the maximum number of feasible solutions (values for the variables that satisfy all constraints) allowed by the maxFeasibleSols option setting within engineSettings.
16	Stop chosen when the max number of feasible [integer] subproblems was reached.	If you are using the Evolutionary Solver, this result is returned when the Solver has explored the maximum number of subproblems specified for maxSubproblems within engineSettings. You may increase the value for the maxSubproblems, leave the option setting at its default, unlimited.
17	Solver converged in probability to a global solution.	If you are using the multistart methods for global optimization, with the standard LSGRG solver, or a field-installable nonlinear Solver engine, this result is returned when the multistart method's Bayesian test has determined that all of the locally optimal solutions have probably been found; the solution displayed on the worksheet is the best of these locally optimal solutions, and is probably the globally optimal solution to the problem.
18	All variables must have both upper and lower bounds.	If you are using the Interval Global Solver, this message is returned if you have not defined lower and upper bounds on all of the decision variables in the problem.
19	Variable bounds conflict in binary or alldifferent constraint.	This result is returned if you have both a binary or alldifferent constraint on a decision variable and a <= or >= constraint on the same variable (that is inconsistent with the binary or alldifferent specification), or if two or more of the same decision variables appear in more than one alldifferent constraint.

20	Lower and upper bounds on variables allow no feasible solution.	This result is returned if you've defined lower and upper bounds on a decision variable, where the lower bound is greater than the upper bound.
21	Solver encountered an error computing derivatives.	This message appears when the Interpreter in Solver SDK Platform encounters an error when computing derivatives via automatic differentiation.
22	Variable appears in more than one cone constraint.	This result is returned if the same decision variable appears in more than one cone constraint.
23	Formula depends on uncertainties, must be summarized or transformed.	This result is returned if you've defined constraints or an objective computed by formulas that depend on uncertain parameters.
25	Simulation optimization doesn't handle models with recourse decisions.	This result is returned if you've defined a recourse decision variable, but you've set "simulationOptimization": True within "modelSettings".
26	Solver could not find a feasible solution to the robust chance constrained problem.	This message means that the Solver could not find a feasible solution to the robust counterpart problem. It does not <i>necessarily</i> mean that there is no feasible solution to the original problem; the robust counterpart is an <i>approximation</i> to the problem defined by your chance constraints that may yield conservative solutions which <i>over-satisfy</i> the chance constraints.
27	Solver found a conservative solution to the robust chance constrained problem. All constraints are satisfied.	The message means that the Solver found an optimal solution to the robust counterpart model, but when this solution was tested against your <i>original</i> model (using Monte Carlo simulation to test satisfaction of the chance constraints), the solution <i>over-satisfied</i> the chance constraints; this normally means that the solution is 'conservative' and the objective function value can be further improved. An alternative course of action is to <i>manually</i> adjust the Chance measures of selected chance constraints, by setting the option, chanceAutoAdjust: True in modelSettings, and re-solve the problem. The automatic improvement algorithm uses general-purpose methods to find an improved solution; you may be able to do better by adjusting Chance measures based on your knowledge of the problem.
28	Solver has converged to the current solution of the robust chance constrained problem. All constraints are satisfied.	This result may be returned when you solve a model with uncertainty and chance constraints using robust optimization, and you've set "chanceAutoAdjust": True within "modelSettings".
999	Unexpected error. Please contact Technical Support.	This status signifies that an unexpected exception has occurred within Solver.
1000	Interval Solver requires strictly smooth functions.	The Interval Global Solver considers the 'special' functions ABS, IF, MAX, MIN or SIGN nonsmooth.
1001	Function cannot be evaluated for given real or interval arguments	This message may appear (instead of "Solver encountered an error value...") if the Interval Global Solver encounters an arithmetic operation or function that it cannot evaluate for the current values of the decision variables.
1002	Solution found, but not proven globally optimal.	This message indicates that the Interval Global Solver has systematically explored the solution space and has found a solution that is very

probably the global optimum, but it has not been able to “prove global optimality.”

The chart below contains the available "code" and "codetext" responses for RASON DM. For a list of RASON error codes, see Appendix II in the RASON Reference Guide.

Code	CodeText	Explanation
0	"Success"	Data science method was completed successfully.
-1	"Execution Error"	Indicates an error has occurred during the data science process
-2	"Terminated by User Request"	Indicates that solving process was terminated by the User.
-3	"Error with Algorithm Parameters"	An erroneous parameter has been submitted to a data science algorithm.
-4	"RASON Interpreter Error"	Indicates there is an error with RASON model syntax/data types/keywords, etc.

[Return to Asynchronous Endpoint List](#)

REST

Endpoint: GET: `https://rason.net/api/model/{nameid}/result/{data}`
}

A REST API endpoint that retrieves the output file containing results from the model previously submitted to GET/POST `rason.net/api/model/{nameid}/optimize`, GET/POST `rason.net/api/model/{nameid}/decision`, GET/POST `rason.net/api/model/{nameid}/datamine`, GET/POST `rason.net/api/model/{nameid}/simulate` or GET/POST `rason.net/api/model/{nameid}/solve`. You must specify the output file within the "datasources" section of your RASON model.

- **URL**

`https://rason.net/api/model/{nameid}/result/{datafilename}?p-relative/path/from/the/RASON/file`

- **Method:**

GET

- **URL Params**

Required: When getting a data file for the PMML model generated by a RASON model, the request becomes:

`https://rason.net/api/model/{nameid}/result/{datafilename}?p-relative/path/from/the/RASON/file`

For example, if the RASON model contains the following...

```
"pmmlModelSrc": {
  "type": "xml",
  "content": "pmml-model",
  "connection": "results/pmml/transformation-
rescaling.xml",
  "direction": "export"
}
```

The request would become:

```
GET rason.net/api/model/{nameid}/result/transformation-
rescaling.xml?p=Results/PMML
```

Or if the RASON connection contained...

```
"connection": "transformation-rescaling.xml",
```

The request would become:

```
GET rason.net/api/model/{nameid}/result/transformation-
rescaling.xml
```

Optional: None

- **Headers**

Required: Authorization - Example: Authorization: bearer {your RASON token}

Optional: None

- **Data Params:** None

- **Success Response:**

Code: 200 (OK)

Response Example:

Returns the contents of the file specified within the dataSources section of the RASON model.

Error Response:

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

[Return to Asynchronous Endpoint List](#)

REST Endpoint: GET:
https://rason.net/api/model/{nameorid}/runtoken

A REST API endpoint that gets a runtime token for a named (or unnamed) model.

- **URL**

https://rason.net/api/model/{nameorid}/runtoken

- **Method:**

GET

- **URL Params**

Required: None

Optional: None

- **Headers**

Required: Authorization - Example: Authorization: bearer {your RASON token}

Optional: None

- **Data Params:** None

- **Success Response:**

Code: 200 (OK)

Response Example:

NiJ9.eyJlc2VyaWQiOiIyNTkw...IjE1NzUzMTHJY

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted or model cannot be found. See response body for more information.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

REST Endpoint: GET:
<https://rason.net/api/model/{nameorid}/scoring-model>

A REST API endpoint that generates the script for the RASON Scoring stage by parsing the named fitted model residing on the RASON server or in a OneDrive account or SQL Server. This endpoint returns the appropriate action type (i.e. "predict", "transform", "forecast", etc.) and evaluations (i.e. "prediction", "transformation", "forecast", etc.) which can be pasted into an existing decision flow.

This API endpoint can be useful for more advanced scoring cases, for example when data sources such as OneDrive or SQL Server are used, when the users requires auxiliary evaluations such as posterior probabilities, when advanced parameters are used such as success class or success cutoff probability, or when model management (versioning or scheduling) plays a role in the decision flow.

- **URL**

<https://rason.net/api/model/{nameorid}/scoring-model>

- **Method:**

GET

- **URL Params**

Required: None

Optional:

model-name: Populates the "modelName" property in the response.

Example: GET <https://rason.net/api/model/LogisticRegression/scoring-model?model-name=MyScoringExample>

data-type: Placeholder for the "type" property in the response, for example "csv".

Example: <https://rason.net/api/model/LogisticRegression/scoring-model?data-type=csv>

Currently the RASON modeling language supports nine different data types: "excel" (Microsoft Excel), "access" or "msaccess" (Microsoft Access), "odbc" (ODBC database), "OData" (OData endpoint), "mssql" (Microsoft SQL), "oracle" (Oracle database), CSV (Comma Separated Value), "json" (JSON file), or "xml" (XML file). For more information on using external data sources, please see Data Sources within the RASON Reference Guide.

data-connection: Placeholder for the "data-connection" property in the response.

Example: <https://rason.net/api/model/LogisticRegression/scoring-model?data-connection=MyOneDriveConnection>

where "MyOneDriveConnection" is a RASON Data Connection that connects to a RASON model stored on a OneDrive account. For more information on Data Connections in RASON Decision Services, see the [My Account: Data Connections and Creating API Tokens](#) section that appears earlier in this guide.

- **Headers**

Required: Authorization - Example: Authorization: bearer {your RASON token}

Optional: None

- **Data Params:** None
- **Example Request:** Request contains all three optional URL parametes: model-name = *MyScoringExample*, data-type = *csv* and data-connection = *MyOneDriveConnection*.

<https://rason.net/api/model/LogisticRegression/scoring-model?model-name=MyScoringExample&data-type=csv&data-connection=MyOneDriveConnection>

- **Success Response:**

Code: 200 (OK)

Response Example:

In the example auto-generated RASON scoring model shown below, the data contained in "myDataSource" is fit to the "myFittedModel" fitted model using Logistic Regression. From here, users can add parameters and evaluation statistics as desired.

```
{
  "comment": "Auto-generated RASON Scoring model",
  "modelName": "MyScoringExample",
  "datasources": {
    "myDataSource": {
      "type": "csv",
      "connection": "MyOneDriveConnection"
    }
  },
  "datasets": {
    "myDataset": {
      "binding": "myDataSource"
    }
  },
  "fittedModel": {
    "myFittedModel": {
      "modelName": "LogisticRegression"
    }
  },
  "actions": {
    "myScore": {
      "data": "myDataset",
      "fittedModel": "myFittedModel",
      "action": "predict",
      "parameters": {},
      "evaluations": [
        "prediction"
      ]
    }
  }
}
```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
-----------------------	--

Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

[Return to Asynchronous Endpoint List](#)

REST Endpoint: GET:
<https://rason.net/api/model/{nameorid}/simulate>

A REST API endpoint that enqueues a previously submitted simulation model for solving.

- **URL**

`https://rason.net/api/model/{nameorid}/simulate`

- **Method:**

GET

- **URL Params**

Required: None

- **Optional:** Any data component may be passed as a query parameter if a binding property exists for that parameter. For example, see the RASON example code snippet below.

```
data: {  
  tktPrice: {  
    value: [150],  
    binding: 'get'  
  }  
}
```

To change a query parameter outside of the RASON model environment, use:

```
$.Post("https://rason.net/api/simulate?tktpPrice=300", ...
```

RASON Server will map "tktpPrice=?" with "tktpPrice = 300". A query can return any number of rows that satisfy the filtering condition, from 0 to infinity.

To perform a query using multiple parameters (say custID, maritalStatus and age) outside of the RASON model environment, use the query parameter:

```
$.get(https://rason.net/api/decision?custID=c2&maritalStatus=s&age=40.....
```

Or in general,

```
$.get(https://rason.net/api/decision?par1=val1&par2=val2.....
```

- **Headers**

Required: Authorization - Example: Authorization: bearer {your RASON token}

Optional: None

- **Data Params:** None

- **Success Response:**

Code: 202 (Accepted)

Response Example:

```
{
```



```

    "Accepted": "2019-11-19+20-40-19"
  }

```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

[Return to Asynchronous Endpoint List](#)

REST Endpoint: GET:
https://rason.net/api/model/{nameorid}/solve

A REST API endpoint that accepts an analytic model (optimization, simulation, data science or decision table) in JSON and places it in the queue to be solved.

- **URL**

`https://rason.net/api/model/{nameorid}/solve`

- **Method:**

GET

- **URL Params**

Required: None

- **Optional:** Any data component may be passed as a query parameter if a binding property exists for that parameter. For example, see the RASON example code snippet below.

```
data: {  
  tktPrice: {  
    value: [150],  
    binding: 'get'  
  }  
}
```

To change a query parameter outside of the RASON model environment, use:

```
$.Post("https://rason.net/api/simulate?tkPrice=300", ...
```

RASON Server will map "tkPrice=?" with "tkPrice = 300". A query can return any number of rows that satisfy the filtering condition, from 0 to infinity.

To perform a query using multiple parameters (say custID, maritalStatus and age) outside of the RASON model environment, use the query parameter:

```
$.get(https://rason.net/api/decision?custID=c2&maritalStatus=s&age=40 .....)
```

Or in general,

```
$.get(https://rason.net/api/decision?par1=val1&par2=val2.....)
```

In a data science model, any *datasource* component may be passed as a query parameter if a binding property exists for that datasource.

```
"datasources": {  
  "srcCustomers": {  
    "type": "csv",  
    "connection": "customers_dt.txt",  
    "selection": "custID = $parCustID1 or custID =  
$parCustID2",  
    "parameters": {  
      "parCustID1": {  
        "binding": "get",  
        "value": "c1"  
      }  
    }  
  }  
}
```

```

    },
    "parCustID2": {
      "binding": "get",
      "value": "c3"
    }
  }
}
},

```

To query `custID=$parCustID1` or `custID=$parCustID2` outside of the RASON model environment, use:

```
$.get (https://rason.net/api/datamine?parCustID1=c1&parCustID2=c2...
```

Or in general,

```
$.get (https://rason.net/api/datamine?par1=val1&par2=val2.....
```

RASON Server will map "`custID=$parCustID1`" with "`parCustID1=c1`" and "`custID = $parCustID2`" with "`parCustID2=c3`". A query can return any number of rows that satisfy the filtering condition, from 0 to infinity.

- **Headers**

Required: Authorization - Example: `Authorization: bearer {your RASON token}`

Optional: None

- **Data Params**

None

- **Success Response:**

Code: 202 (Accepted)

```

{
  "Accepted": "2019-12-02+18-27-32"
}

```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	REST API unable to map Client's URI to an unknown resource, i.e. passing an erroneous resource ID.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

[Return to Asynchronous Endpoint List](#)

REST Endpoint: GET:
<https://rason.net/api/model/{nameorid}/stages>

An asynchronous REST API endpoint that returns the list of stages from the previously posted decision flow. This endpoint takes three parameters: all, ordered or terminal.

- **URL**

<https://rason.net/api/model/{nameorid}/stages>

- **Method:**

GET

- **URL Params**

Required: None

Optional:

- All (default): Retrieves stage information *from all stages* from the decision flow.
 - Example: <https://rason.net/api/model/{nameorid}/stages?type=all>
- Ordered: Retrieves the stage information in the optimal order of execution
 - Example: <https://rason.net/api/model/{nameorid}/stages?type=ordered>
- Terminal: Retrieves the terminal nodes in the decision flow.
 - Example: <https://rason.net/api/model/{nameorid}/stages?type=terminal>

- **Headers**

Required: Authorization - Example: Authorization: bearer {your RASON token}

Optional: None

- **Data Params**

A decision flow RASON Model

```
{
  "workflow": "imputationWorkflow",
  "imputation": {
    "datasets": {
      "trainData": {
        "value": [
          [
            "black",
            null,
            6,
            2,
            1,

```

```

        "nan",
        1
    ],
    [
        "",
        3,
        9,
        5.1,
        null,
        "",
        2
    ],
    [
        "red",
        7,
        8,
        null,
        9.2,
        "small",
        3
    ],
    [
        "red",
        10000,
        null,
        4.4,
        4.4,
        "large",
        -1
    ],
    [
        "blue",
        2,
        3,
        5.6,
        3.4,
        "unknown",
        5
    ]
],
"colNames": [
    "A",
    "B",
    "C",
    "D",
    "E",
    "F",
    "G"
]
},
"estimator": {
    "myImputer": {
        "type": "transformation",
        "algorithm": "imputation",
        "parameters": {
            "imputationStrategy": [

```

```

        [
            "A",
            "MODE"
        ],
        [
            "B",
            "MEAN"
        ],
        [
            "C",
            "MEDIAN"
        ],
        [
            "D",
            "DELETE_RECORD"
        ],
        [
            "E",
            "DELETE_RECORD"
        ],
        [
            "F",
            "VALUE"
        ],
        [
            "G",
            "MEAN"
        ]
    ],
    "placeholder": [
        [
            "B",
            10000
        ],
        [
            "F",
            "unknown"
        ],
        [
            "G",
            -1
        ]
    ]
}
},
"actions": {
    "imputingModel": {
        "data": "trainData",
        "estimator": "myImputer",
        "action": "fit",
        "evaluations": [
            "fittedModelJson"
        ]
    }
}
},

```

```

    "transforming": {
      "comment": "imputing missing values based on json model from
the parent stage",
      "datasources": {
        "myModelSrc": {
          "type": "stage",
          "connection": "imputation",
          "selection": "imputingModel.fittedModelJson",
          "direction": "import"
        }
      },
      "datasets": {
        "newData": {
          "value": [
            [
              "blue",
              null,
              null,
              4,
              1,
              "unknown",
              -1
            ]
          ],
          "colNames": [
            "A",
            "B",
            "C",
            "D",
            "E",
            "F",
            "G"
          ]
        },
        "imputingModel": {
          "binding": "myModelSrc"
        }
      },
      "actions": {
        "fixedNewData": {
          "data": "newData",
          "fittedModel": "imputingModel",
          "parameters": {
            "imputation": [
              [
                "F",
                "medium"
              ]
            ]
          },
          "action": "transform",
          "evaluations": [
            "transformation"
          ]
        }
      }
    }
  }
}

```



```
}
```

- **Example Request:** <https://rason.net/api/model/imputationWorkflow/stages>

- **Success Response:**

Code: 200 (OK)

```
[  
  "imputation",  
  "transforming"  
]
```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	REST API unable to map Client's URI to an unknown resource, i.e. passing an erroneous resource ID.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

[Return to Asynchronous Endpoint List](#)

REST Endpoint: GET:

https://rason.net/api/model/{nameorid}/stages/{stage-name} or all

An asynchronous REST API endpoint that returns stage information *of a specific stage* from a previously posted decision flow model. This endpoint does *not* solve the model. If 'stage-name' =all, then information concerning all stages in the model is returned.

This endpoint returns complete information about the stages in a decision flow without having to actually solve the decision flow first.

- **name:** The name of the stage.
 - **type:** The type of stage: calculation, data science, optimization or simulation.
 - **pipeline:** Preceding stages required to solve a given stage in the topological order of execution.
 - **allResults:** All outputs expected to be produced by this stage.
 - **resultsForBindings:** Subset of allResults that participate in inter-stage bindings.
 - **dataSources:** Information about all stage inputs such as name, type (i.e. CSV, Excel, ODBC, etc.) connection (file name), selection (selected columns), content (json or pmml model), direction (import/export), isUsed (True/False – if datasource is used in the RASON model or just defined and not used), isStageBinding (True/False), isFittedModel (True/False) . For more information on dataSources components, see the RASON Reference Guide.
 - **isTerminal:** True if the stage is a terminal stage, otherwise False.
- **URL**
 - **all (default):** Retrieves stage information *from all stages* from the decision flow.

Example: `https://rason.net/api/model/{nameorid}/stages/all`
 - **{stage-name}:** Retrieves the stage information for the specific for "stage-name" (i.e. imputation, transforming, etc.) in the decision flow.

Example: `https://rason.net/api/model/{nameorid}/stages/imputation`

- **Method:**

GET

- **URL Params**

Required: None

Optional: None

- **Headers**

Required: Authorization - Example: `Authorization: bearer {your RASON token}`

Optional: None

- **Data Params**

A RASON Model

```
{
  "workflow": "imputationWorkflow",
  "imputation": {
    "datasets": {
      "trainData": {
        "value": [
          [
            "black",
            null,
            6,
            2,
            1,
            "nan",
            1
          ],
          [
            "",
            3,
            9,
            5.1,
            null,
            "",
            2
          ],
          [
            "red",
            7,
            8,
            null,
            9.2,
            "small",
            3
          ],
          [
            "red",
            10000,
            null,
            4.4,
            4.4,
            "large",
            -1
          ],
          [
            "blue",
            2,
            3,
            5.6,
            3.4,
            "unknown",
            5
          ]
        ],
        "colNames": [
          "A",
          "B",
          "C",
          "D",
          "E",
          "F",
          "G"
        ]
      }
    }
  }
}
```

```

    ]
  }
},
"estimator": {
  "myImputer": {
    "type": "transformation",
    "algorithm": "imputation",
    "parameters": {
      "imputationStrategy": [
        [
          "A",
          "MODE"
        ],
        [
          "B",
          "MEAN"
        ],
        [
          "C",
          "MEDIAN"
        ],
        [
          "D",
          "DELETE_RECORD"
        ],
        [
          "E",
          "DELETE_RECORD"
        ],
        [
          "F",
          "VALUE"
        ],
        [
          "G",
          "MEAN"
        ]
      ],
      "placeholder": [
        [
          "B",
          10000
        ],
        [
          "F",
          "unknown"
        ],
        [
          "G",
          -1
        ]
      ]
    }
  }
},
"actions": {
  "imputingModel": {
    "data": "trainData",
    "estimator": "myImputer",
    "action": "fit",
    "evaluations": [
      "fittedModelJson"
    ]
  }
}

```

```

    }
  },
  "transforming": {
    "comment": "imputing missing values based on json model from the
parent stage",
    "datasources": {
      "myModelSrc": {
        "type": "stage",
        "connection": "imputation",
        "selection": "imputingModel.fittedModelJson",
        "direction": "import"
      }
    },
    "datasets": {
      "newData": {
        "value": [
          [
            "blue",
            null,
            null,
            4,
            1,
            "unknown",
            -1
          ]
        ],
        "colNames": [
          "A",
          "B",
          "C",
          "D",
          "E",
          "F",
          "G"
        ]
      },
      "imputingModel": {
        "binding": "myModelSrc"
      }
    },
    "actions": {
      "fixedNewData": {
        "data": "newData",
        "fittedModel": "imputingModel",
        "parameters": {
          "imputation": [
            [
              "F",
              "medium"
            ]
          ]
        },
        "action": "transform",
        "evaluations": [
          "transformation"
        ]
      }
    }
  }
}

```

- **Success Response:**

- GET <https://rason.net/api/model/OneHotEncodingInLine/stages/all>

Code: 200 (OK)

```
[
  {
    "name": "imputation",
    "type": "datamining",
    "pipeline": [
      "imputation"
    ],
    "allResults": [
      "imputingmodel.fittedModelJson"
    ],
    "resultsForBindings": [
      "imputingmodel.fittedmodeljson"
    ],
    "dataSources": [],
    "isTerminal": false
  },
  {
    "name": "transforming",
    "type": "datamining",
    "pipeline": [
      "imputation",
      "transforming"
    ],
    "allResults": [
      "fixednewdata.transformation"
    ],
    "resultsForBindings": [],
    "dataSources": [
      {
        "name": "mymodelsrc",
        "type": "stage",
        "connection": "imputation",
        "selection": "imputingmodel.fittedmodeljson",
        "content": "",
        "direction": "import",
        "isUsed": true,
        "isStageBinding": true,
        "isFittedDMMModel": false
      }
    ],
    "isTerminal": true
  }
]
```

- GET <https://rason.net/api/model/OneHotEncodingInLine/stages/imputation>

Code: 200 (OK)

```
[
  {
    "name": "imputation",
    "type": "datamining",
    "pipeline": [
      "imputation"
    ],
    "allResults": [
      "imputingmodel.fittedModelJson"
    ],
    "resultsForBindings": [
      "imputingmodel.fittedmodeljson"
    ],
    "dataSources": [],
    "isTerminal": false
  }
]
```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	REST API unable to map Client's URI to an unknown resource, i.e. passing an erroneous resource ID.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

[Return to Asynchronous Endpoint List](#)

REST Endpoint: GET:
https://rason.net/api/model/{nameorid}/status

A REST API endpoint that retrieves the status of the model previously submitted to GET/POST
rason.net/api/model/{nameorid}/optimize, GET/POST
rason.net/api/model/{nameorid}/datamine, GET/POST
rason.net/api/model/{nameorid}/decision, GET/POST
rason.net/api/model/{nameorid}/diagnose, GET/POST
rason.net/api/model/{nameorid}/simulate or GET/POST
rason.net/api/model/{nameorid}/solve.

RASON Decision Services tracks and reports the following:

1. *For standalone RASON/Excel models/stages* – The current “message” is reported, similar to the ones reported in Analytic Solver's taskpane: "Loading", "Parsing", "Diagnosing", "Solving", etc. with % progress, when available.
2. *For standalone RASON/Excel Optimization models/stages* – Solving progress reported containing information on the elapsed time, incumbent solutions, objective, etc. when available.
3. *For multi-stage flows* – Completion status of each stage is reported. For the stage that is currently being solved, 1. and 2. (above) are reported when available.

Note: Initially “*status*”: “*Incomplete*” could be reported due to initial model parsing, loading model into memory, etc. or when no message/progress info is yet available or due to a lag in updating the Azure storage with the progress info. In order to avoid frequent table operations, this table is updated every few seconds.

- **URL**

https://rason.net/api/model/{nameorid}/status

- **Method:**

GET

- **URL Params**

Required: None

Optional: None

- **Headers**

Required: Authorization - Example: Authorization: bearer {your RASON token}

Optional: None

- **Data Params:** None

- **Success Response:**

Code: 200 (OK)

Example Request: GET https://rason.net/api/model/HardMip/status

Response Examples:

Example: Solve complete

```
{
  "status": "Complete"
}
```

Example: Solve incomplete, loading workbook

```
{
  "status": "Incomplete",
  "progress": {
    "message": "[Status] Loading Workbook: <workbook name>"
  }
}
```

Example: Solve incomplete, parsing model

```
{
  "status": "Incomplete",
  "progress": {
    "message": "[Progress] Parsing...2.755906%"
  }
}
```

Example: Solve incomplete, diagnosing model

```
{
  "status": "Incomplete",
  "progress": {
    "message": "[Progress] Diagnosing...75.5906%"
  }
}
```

Example: Solve incomplete (optimization model).

```
{
  "status": "Incomplete",
  "progress": {
    "optimization": {
      "elapsedMilliseconds": 0,
      "iterations": 0,
      "subProblems": 0,
      "localSolutions": 0,
      "objective": 0.0
    }
  }
}
```

Example: Solve incomplete, incumbent found (optimization model).

```
{
  "status": "Incomplete",
  "progress": {
    "message": "[Status] Solving...",
    "optimization": {
      "elapsedMilliseconds": 23028,
      "iterations": 32,
      "subProblems": 15240,
      "localSolutions": 5,
      "objective": 57.0
    }
  }
}
```

```
}  
}  
}
```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

[Return to Asynchronous Endpoint List](#)

REST Endpoint: PATCH:
https://rason.net/api/model/{nameorid}

A REST API endpoint that, with "champion" in the body, marks the ID (or the most recent ID if name is given) as the current champion. If used with "challenger", or nothing, in the body, the ID becomes unmarked as a "champion", reverting to the rule that the most recent version is the "champion".

Note: When PATCH <https://rason.net/api/model/{nameorid}> is used to mark a model as a "champion", the model will remain a champion even if you use PUT <https://rason.net/api/model/{nameorid}> to create a new version of the same model. The model will remain the champion until another call is made to PATCH <https://rason.net/api/model/{nameorid}> to unmark the model.

- **URL**

<https://rason.net/api/model/{nameorid}>

- **Method:**

PATCH

- **URL Params**

Required: None

Optional: None

- **Headers**

Required: Authorization - Example: Authorization: bearer {your RASON token}

Optional: None

- **Data Params:** None

- **Success Response:**

Code: 200 (OK)

Response Example:

None

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.

Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

[Return to Asynchronous Endpoint List](#)

REST Endpoint: *POST: <https://rason.net/api/fitted-info>*

The POST <https://rason.net/api/fitted-info> endpoint sends the fitted model contained within the request body to the RASON server and returns a response. Information returned includes 1. the *name* of the fitted model, 2. the *format* of the fitted model (JSON or PMML), 3. the model *type* (classification, regression, etc.), 4. the *algorithm* used to fit the model, the included *features* (input variables) and 5. the *output variable* (target variable).

See the related API endpoint: GET <https://rason.net/api/model/{nameorid}/fitted-info> which returns information on the fitted model already POSTed on the RASON server.

Note that this endpoint does not POST the fitted model to the RASON Server. This endpoint is intended to help the user understand his/her fitted model and determine what type of data can be scored.

URL

<https://rason.net/api/fitted-info>

- **Method:**

POST

- **URL Params**

Required: None

Optional: None

- **Headers**

Required: Authorization - Example: Authorization: bearer {your RASON token}

Optional: None

- **Data Params:** A fitted model in the request body.

Request Body Example: A Logistic Regression fitted model

```
{
  "modelName": "LogisticRegression",
  "objectType": "fittedModel",
  "type": "classification",
  "algorithm": "logisticRegression",
  "variables": ["X1", "X2", "X3", "X4", "Weights"],
  "targetColumn": "Y",
  "categoricalVariables": {},
  "fittedModel": {
    "classes": {
      "objectType": "stringVector",
      "name": "StringVector",
      "data": ["0", "1"]
    },
    "coefficients": [{
      "name": "intercept",
      "categorical": false,
      "estimate": -428.55610291564261
    }
  ]
}
```

```

    },
    {
      "name": "X1",
      "categorical": false,
      "estimate": 4.2446385886748441
    },
    {
      "name": "X2",
      "categorical": false,
      "estimate": 5.0316693922277373
    },
    {
      "name": "X3",
      "categorical": false,
      "estimate": 2.5501718637097612
    },
    {
      "name": "X4",
      "categorical": false,
      "estimate": 3.8639376929625953
    },
    {
      "name": "Weights",
      "categorical": false,
      "estimate": 6.8544171311311874
    }
  ],
  "encoder": {
    "modelName": "",
    "objectType": "fittedModel",
    "type": "transformation",
    "algorithm": "oneHotEncoding",
    "variables": ["X1", "X2", "X3", "X4", "Weights"],
    "categoricalVariables": {},
    "fittedModel": {
      "mapping": null
    }
  },
  "fitIntercept": true,
  "priorProb": {
    "0": 0.46153846153846156,
    "1": 0.53846153846153844
  },
  "successClass": "1"
}

```

- **Success Response:**

Code: 200 (OK)

- <https://rason.net/api/fitted-info>

Example Request

<https://rason.net/api/fitted-info>

Example Response

In the example response below, the model name is "LogisticRegression", the format is JSON, the type of model is "classification", the algorithm used to fit the model is "logistic regression", the included features are "x1", "x2", "x3", "x4" and "weights" and the output or target variable is "Y".

```
{
  "modelName": "LogisticRegression",
  "format": "JSON",
  "type": "classification",
  "algorithm": "logisticRegression",
  "features": [
    {
      "name": "X1",
      "type": "continuous"
    },
    {
      "name": "X2",
      "type": "continuous"
    },
    {
      "name": "X3",
      "type": "continuous"
    },
    {
      "name": "X4",
      "type": "continuous"
    },
    {
      "name": "Weights",
      "type": "continuous"
    }
  ],
  "targetName": "Y"
}
```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.

[Return to Asynchronous Endpoint List](#)

REST Endpoint: POST: <https://rason.net/api/interface>

A REST API endpoint that generates an interface for a reusable RASON model that can be utilized in a decision flow. This endpoint accepts models defined in either a RASON script or an Excel workbook.

This REST API endpoint may be used to post an Analytic Solver optimization, simulation or decision table model within an Excel workbook. Attach the Excel workbook to POST rason.net/api/interface as *Content-Type=multipart/form-data*. (You may also use the REST API endpoint *GET rason.net/api/model/{nameorid}/interface* on a previously posted model.) An optional query parameter exists, *?worksheet=<name>*, which can be used when an Excel workbook contains multiple worksheets. If this query parameter is not present, the active sheet (or the entire workbook) is considered.

- **URL**

<https://rason.net/api/interface>

- **Method:**

POST

- **URL Params**

Required: None

Optional: *?worksheet=<name>* where *<name>* is the name of the worksheet

- **Headers**

Required: Authorization - Example: *Authorization: bearer {your RASON token}*

Optional: None

- **Data Params:**

A reusable model.

In this example, a reusable model that partitions the hald-small.txt dataset into three partitions: training, validation and test.

```
{
  "modelName": "partitioning-reusable",
  "modelType": "datamining",
  "modelDescription": "transformation: partitioning, independent
on
input parameters",
  "datasources": {
    "dataToPartitionSource": {
      "type": "csv",
      "connection": "hald-small.txt"
    }
  },
  "datasets": {
    "dataToPartition": {
```



```

        "binding": "dataToPartitionSource"
    }
},
"transformer": {
    "myPartitioner": {
        "type": "transformation",
        "algorithm": "partitioning",
        "parameters": {
            "partitionMethod": "RANDOM",
            "ratios": [
                [
                    "Training",
                    0.5
                ],
                [
                    "Validation",
                    0.3
                ],
                [
                    "Testing",
                    0.2
                ]
            ],
            "seed": 123
        }
    }
},
"actions": {
    "trainingPartition": {
        "data": "dataToPartition",
        "action": "transform",
        "parameters": {
            "partition": "Training"
        },
        "evaluations": [
            "transformation"
        ]
    },
    "validationPartition": {
        "data": "dataToPartition",
        "action": "transform",
        "parameters": {
            "partition": "Validation"
        },
        "evaluations": [
            "transformation"
        ]
    },
    "testPartition": {
        "data": "dataToPartition",
        "action": "transform",
        "parameters": {
            "partition": "Testing"
        },
        "evaluations": [
            "transformation"
        ]
    }
}

```

```

    }
  }
}

```

- **Success Response:**

Code: 200 (OK)

Response Example:

Returned results from this REST API endpoint can be copied and pasted into a decision flow. For more information on decision flows and how this REST API Endpoint can help you create a decision flow quickly and easily, see the example below.

```

{
  "comment": "This interface has been generated by Psi for a RASON model",
  "invokeModel": "partitioning-reusable",
  "modelType": "datamining",
  "modelDescription": "transformation: partitioning, independent on input parameters",
  "datasources": {
    "datatopartitionsource": {
      "type": "csv",
      "connection": "hald-small.txt",
      "direction": "import"
    }
  },
  "outputResults": {
    "trainingPartition": {
      "evaluations": [
        "transformation"
      ],
      "type": "dataset",
      "comment": "datamining evaluation"
    },
    "validationPartition": {
      "evaluations": [
        "transformation"
      ],
      "type": "dataset",
      "comment": "datamining evaluation"
    },
    "testPartition": {
      "evaluations": [
        "transformation"
      ],
      "type": "dataset",
      "comment": "datamining evaluation"
    }
  }
}

```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.

Code: 500 INTERNAL SERVER ERROR

Something has gone wrong on the RASON Server, contact technical support.

Creating a Decision Flow using POST rason.net/api/interface

In the previous chapter, Creating and Solving Decision Flows, a decision flow was created using the Decision flow editor on www.RASON.com. Users can also create a decision flow outside of the Decision flow editor by using the REST API endpoint, POST rason.net/api/interface. To start, pass each reusable model (separately), that is to be included in the decision flow, in the body of the call to POST rason.net/api/model. Use the REST API endpoint GET rason.net/api/model/interface to retrieve the interface from an already posted model.

Interface for optStageReusable

```
{
  "comment": "This interface has been generated by Psi for a RASON model",
  "invokeModel": "optStageReusable",
  "modelType": "optimization",
  "modelDescription": "Project Select converted into a reusable model",
  "outputResults": {
    "x": {
      "evaluations": [
        "initialValue",
        "finalValue",
        "dualValue"
      ],
      "type": "array/number",
      "comment": "decision variable block"
    },
    "totalObjective": {
      "evaluations": [
        "finalValue"
      ],
      "type": "number",
      "comment": "objective function"
    }
  }
}
```

Interface for simStageReusable

```
{
  "comment": "This interface has been generated by Psi for a RASON model",
  "invokeModel": "simStageReusable",
  "modelType": "simulation",
  "modelDescription": "Project Select converted into a reusable model",
  "inputParameters": {
    "finalVarValues": {
      "value": null,
      "type": "array",
      "comment": "pass the final variable values from optStage"
    }
  },
  "outputResults": {
    "cash": {
      "evaluations": [
        "mean"
      ],
      "type": "number",
      "comment": "uncertain function block"
    }
  }
}
```

Combine the two returned interfaces to create the decision flow. You'll need to add the *flowName* property and optionally the *flowDescription* property along with a name for each stage. (See items in red.)

Combined to form a Decision Flow

```

{
  "flowName": "OptSimReusable",
  "flowDescription": "Opt Sim Decision Flow",
  "optStageReusable": {
    "comment": "This interface has been generated by Psi for a RASON model",
    "invokeModel": "optStageReusable",
    "modelType": "optimization",
    "modelDescription": "Project Select converted into a reusable model",
    "outputResults": {
      "x": {
        "evaluations": [
          "initialValue",
          "finalValue",
          "dualValue"
        ],
        "type": "array/number",
        "comment": "decision variable block"
      },
      "totalObjective": {
        "evaluations": [
          "finalValue"
        ],
        "type": "number",
        "comment": "objective function"
      }
    }
  },
  "simStageReusable": {
    "comment": "This interface has been generated by Psi for a RASON model",
    "invokeModel": "simStageReusable",
    "modelType": "simulation",
    "modelDescription": "Project Select converted into a reusable model",
    "inputParameters": {
      "finalVarValues": {
        "value": optStageReusable.x.finalValue,
        "type": "array",
        "comment": "pass the final variable values from optStage"
      }
    },
    "outputResults": {
      "cash": {
        "evaluations": [
          "mean"
        ],
        "type": "number",
        "comment": "uncertain function block"
      }
    }
  }
}
}

```

[Return to Asynchronous Endpoint List](#)

REST Endpoint: POST: <https://rason.net/api/score>

The POST <https://rason.net/api/score> endpoint produces the scores (prediction, transformation, forecast, etc.) given the fitted PMML/JSON model and new data attached to the request as "form-data" consisting of two content parts: "FittedModel" and "Data".

New data is expected to be either 1D (single record) or 2D (multiple records) JSON array compatible with the fitted model. (Note: Column headings are not supported.) The type of model and headers is inferred automatically. If the request is successful, the response will include the scores serialized in JSON array.

See the related API endpoint: POST <https://rason.net/api/model/{nameorid}/score> which expects only the data to be provided in the request body. The fitted model is located on the RASON server.

URL

<https://rason.net/api/score>

- **Method:**

POST

- **URL Params**

Required: None

Optional:

Header – Set to True if the "Data" data parameter contains column headings and False (the default) if the new data does not contain column headings.

Example:

<https://rason.net/api/model/LogisticRegression/score?header=true>

- **Headers**

Required: Authorization - Example: Authorization: bearer {your RASON token}

Optional: None

- **Data Params:** Form Data consisting of two content parts "FittedModel" and "Data" which can be attached files or inline.

FittedModel

```
{
  "modelName": "LogisticRegression",
  "objectType": "fittedModel",
  "type": "classification",
  "algorithm": "logisticRegression",
  "variables": ["X1", "X2", "X3", "X4", "Weights"],
  "targetColumn": "Y",
  "categoricalVariables": {},
  "fittedModel": {
    "classes": {
      "objectType": "stringVector",
      "name": "StringVector",
      "data": ["0", "1"]
    }
  }
}
```

```

    },
    "coefficients": [{
      "name": "intercept",
      "categorical": false,
      "estimate": -428.55610291564261
    },
    {
      "name": "X1",
      "categorical": false,
      "estimate": 4.2446385886748441
    },
    {
      "name": "X2",
      "categorical": false,
      "estimate": 5.0316693922277373
    },
    {
      "name": "X3",
      "categorical": false,
      "estimate": 2.5501718637097612
    },
    {
      "name": "X4",
      "categorical": false,
      "estimate": 3.8639376929625953
    },
    {
      "name": "Weights",
      "categorical": false,
      "estimate": 6.8544171311311874
    }
  ],
  "encoder": {
    "modelName": "",
    "objectType": "fittedModel",
    "type": "transformation",
    "algorithm": "oneHotEncoding",
    "variables": ["X1", "X2", "X3", "X4", "Weights"],
    "categoricalVariables": {},
    "fittedModel": {
      "mapping": null
    }
  },
  "fitIntercept": true,
  "priorProb": {
    "0": 0.46153846153846156,
    "1": 0.53846153846153844
  },
  "successClass": "1"
}
}

```

Data

X1 X2 X3 X4 Weights

7 26 6 60 1

```

1  29  15  52  3
11 56   8  20  2
11 31   8  47  2
7  52   6  33  1

```

- **Success Response:**

Code: 200 (OK)

Example Request

POST <https://rason.net/api/score?header=true>

Example Response

The example response below contains the scored values.

```

[
  "0",
  "0",
  "1",
  "0",
  "1"
]

```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.

[Return to Asynchronous Endpoint List](#)

REST Endpoint: POST: <https://rason.net/api/scoring-model>

The POST <https://rason.net/api/scoring-model> endpoint sends the fitted model contained within the request body to the RASON server and returns an auto-generated script of the scoring stage of a RASON model.

This API endpoint can be useful for more advanced scoring cases, for example when data sources such as OneDrive or SQL Server are used, when the users requires auxiliary evaluations such as posterior probabilities, when advanced parameters are used such as success class or success cutoff probability, or when model management (versioning or scheduling) plays a role in the decision flow.

See the related API endpoint: GET <https://rason.net/api/model/{nameorid}/scoring-model> which returns an auto-generated script of the scoring stage of a RASON model already POSTed on the RASON server.

URL

<https://rason.net/api/scoring-model>

- **Method:**

POST

- **URL Params**

Required: None

Optional:

- **Headers**

Required: Authorization - Example: Authorization: bearer {your RASON token}

Optional:

model-name: Populates the "modelName" property in the response.

Example: POST <https://rason.net/api/scoring-model?model-name=MyScoringExample>

data-type: Placeholder for the "type" property in the response, for example "csv".

Example: POST <https://rason.net/api/scoring-model?data-type=csv>

Currently the RASON modeling language supports nine different data types: "excel" (Microsoft Excel), "access" or "msaccess" (Microsoft Access), "odbc" (ODBC database), "OData" (OData endpoint), "mssql" (Microsoft SQL), "oracle" (Oracle database), CSV (Comma Separated Value), "json" (JSON file), or "xml" (XML file). For more information on using external data sources, please see Data Sources within the RASON Reference Guide.

data-connection: Placeholder for the "data-connection" property in the response.

Example: POST <https://rason.net/api/scoring-model?data-connection=MyOneDriveConnection>

where "MyOneDriveConnection" is a RASON Data Connection that connects to a RASON model stored on a OneDrive account. For more information on Data Connections in RASON Decision Services, see the [*My Account: Data Connections and Creating API Tokens*](#) section that appears earlier in this guide.

- **Data Params:** A fitted model in the request body.

Request Body Example: A Logistic Regression fitted model

```
{
  "modelName": "LogisticRegression",
  "objectType": "fittedModel",
  "type": "classification",
  "algorithm": "logisticRegression",
  "variables": ["X1", "X2", "X3", "X4", "Weights"],
  "targetColumn": "Y",
  "categoricalVariables": {},
  "fittedModel": {
    "classes": {
      "objectType": "stringVector",
      "name": "StringVector",
      "data": ["0", "1"]
    },
    "coefficients": [{
      "name": "intercept",
      "categorical": false,
      "estimate": -428.55610291564261
    },
    {
      "name": "X1",
      "categorical": false,
      "estimate": 4.2446385886748441
    },
    {
      "name": "X2",
      "categorical": false,
      "estimate": 5.0316693922277373
    },
    {
      "name": "X3",
      "categorical": false,
      "estimate": 2.5501718637097612
    },
    {
      "name": "X4",
      "categorical": false,
      "estimate": 3.8639376929625953
    },
    {
      "name": "Weights",
      "categorical": false,
      "estimate": 6.8544171311311874
    }
  ],
  "encoder": {
    "modelName": "",
    "objectType": "fittedModel",
    "type": "transformation",
    "algorithm": "oneHotEncoding",
    "variables": ["X1", "X2", "X3", "X4", "Weights"],
    "categoricalVariables": {},
    "fittedModel": {
      "mapping": null
    }
  }
}
```

```

    },
    "fitIntercept": true,
    "priorProb": {
      "0": 0.46153846153846156,
      "1": 0.53846153846153844
    },
    "successClass": "1"
  }
}

```

- **Success Response:**

Code: 200 (OK)

- <https://rason.net/api/scoring-model>

Example Request

POST <https://rason.net/api/scoring-model?model-name=MyScoringExample&data-type=csv&data-connection=MyOneDriveConnection>

Example Response

The example response below contains an auto-generated RASON script which scores the new data within the myDataset datasource using the fitted model, myFittedModel. The fitted model was generate using the Logistic Regression algorithm.

```

{
  "comment": "Auto-generated RASON Scoring model",
  "modelName": "MyScoringExample",
  "datasources": {
    "myDataSource": {
      "type": "csv",
      "connection": "MyOneDriveConnection"
    }
  },
  "datasets": {
    "myDataset": {
      "binding": "myDataSource"
    }
  },
  "fittedModel": {
    "myFittedModel": {
      "modelName": "LogisticRegression"
    }
  },
  "actions": {
    "myScore": {
      "data": "myDataset",
      "fittedModel": "myFittedModel",
      "action": "predict",
      "parameters": {},
      "evaluations": [
        "prediction"
      ]
    }
  }
}

```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.

[Return to Asynchronous Endpoint List](#)

REST Endpoint: POST: <https://rason.net/api/model>

A REST API endpoint that creates an unnamed or named resource ID for

- a RASON model,
- an Excel optimization/simulation/decision table/data science model,
- a data science fitted model (RASON or PMML)

and returns a Location header with the resource ID. If "modelName": "myModelName" exists in the body of the RASON model or if the model name is appended to this endpoint (POST <https://rason.net/api/model/{myModelName}>), then this REST API endpoint will create a named resource. The model, "myModelName", must be a RASON model and not a fitted model. *Model names are case sensitive.*

The returned model ID URL facilitates subsequent calls to the RASON REST API endpoints necessary for manipulating your model (i.e. perform an optimization, run a simulation, check the status of the model, get results, etc.).

Note: When using this endpoint to post and solve an Excel model, the Excel model is uploaded and stored on the RASON Server.

Below is a brief explanation of the returned properties

- **modelId:** The unique identifier for the model
- **modelName:** The name of the model taken from the model text. When using POST rason.net/api/model/<name> the name from the url is validated against the name in the model, if specified.
- **modelDescr:** The model description taken directly from the model text
- **modelFiles:** The list of files used by the model as taken from the model text.
- **runtimeToken:** Lists any runtime token associated with this model
- **modelType:** Lists the *type* of model such as "origin", "version", "instance".
 - “origin” – the initially posted model
 - “version” – means that this model is a version of a previously POSTed or PUT model
 - “instance” – these are models created whenever a model is run.
- **modelKind:** Lists the *kind* of model such as "fitted", "excel" or "RASON".
 - “fitted” – fitted Data science model in PMML/JSON format
 - “excel” – models defined in Excel language
 - “RASON” – models defined in RASON language
- **isChampion:** True *only* if model has been specifically marked as the “Champion” using the REST API endpoint PATCH rason.net/api/model/{nameorid}.
- **parentModelId:** Identifies the model from which this model is derived. In the case of “Origin” models, this is always empty. Otherwise, this property identifies the model used to create this model. For example, if a user POST’s a model and then PUT’s another version of this model, "parentModelId" of the version will be the modelId of the originally posted model. Similar logic holds for posting versions of version and instances of version.

- **queryString:** The query string associated with the model when it was created. In the case of unnamed models, this will change throughout the model's lifecycle. A user may post an unnamed model with a particular query string but then solve the model with another. In the case of named models, this is the query string associated with the action used to create the model. A model could be posted with one query string, have several versions with different query strings and associated instances with yet another query string. If a particular version or instance doesn't have a query string, it will inherit the query string of the parent.

A model may be posted with raw body or form-data.

Posting with raw body:

- When a model is posted with the raw RASON script, the model acquires `ModelType=Origin` and `ModelKind=RASON` properties.
- When a data science JSON fitted model is posted, the model acquires `ModelType=Origin`, `ModelKind=Fitted` properties. The model is checked for validity (using Frontline's standards) and `modelName` is inferred from the model JSON text.
- When a data science PMML fitted model is posted (by specifying `Content-Type=application/xml`), the model acquires `ModelType=Origin`, `ModelKind=Fitted` properties. The XML (not PMML) is checked for validity and `modelName` is inferred from the model XML text.
- All other POSTS are considered as unrecognized `ModelKind`.

Posting with Content-Type=multipart/form-data:

- A RASON script can be sent as "Text" with content name "RASONModel". Additionally, one or more required (data) files may be attached.
- If there is a single component in form-data POST request and it is "File":
 - If Content-Type is one of the legal Excel media types (see list below); RASON assumes that the Excel model is being POSTed. The model acquires `ModelType=Origin`, `ModelKind=Excel`. The model name is inferred from the file name or from the request URL, ie. `POST rason.net/api/model/{modelname}`.
There is an optional query parameter `?worksheets=<name>` (where name equals the worksheet name) for the case when the workbook contains multiple worksheets. If omitted, the active sheet (or the entire workbook) is considered.
 - If the file is `Content-Type=application/xml`, RASON assumes that a DM PMML fitted model is being POSTed. The XLM file is *not* upload. Rather RASON reads the content of the file. The model acquires `ModelType=Origin`, `ModelKind=Fitted` properties. The XML (not the PMML) is checked for validity (using Frontline's standards) and the model name is inferred from the model XML text.
 - If the file is `Content-Type=application/json`, RASON assumes that the data science JSON fitted model is being POSTed. The JSON file is *not* uploaded. Rather the contents are read and the model acquires `ModelType=Origin` and `ModelKind=Fitted`. The model is checked for validity (using Frontline's standard) and the model name is inferred from the model JSON text.
 - All other POSTS are considered as unrecognized `ModelKind`.

Note: Click [here](#) to find common MIME types.

Excel Mime Types (case-insensitive):

- "application/vnd.ms-excel"; // .xls, .xlt, .xla
- "application/vnd.openxmlformats-officedocument.spreadsheetml.sheet"; // .xlsx
- "application/vnd.openxmlformats-officedocument.spreadsheetml.template"; // .xltx

- "application/vnd.ms-excel.sheet.macroEnabled.12"; // .xlsm
- "application/vnd.ms-excel.template.macroEnabled.12"; // .xltm
- "application/vnd.ms-excel.addin.macroEnabled.12"; // .xlam
- "application/vnd.ms-excel.sheet.binary.macroEnabled.12"; // .xlsb

- **URL**

<https://rason.net/api/model>

<https://rason.net/api/model/{myModelName}>

Example: <https://rason.net/api/model/TestModelName>

<https://rason.net/api/model/{myModelID}>

Example: <https://rason.net/api/model/123456789>

- **Method:**

POST

- **URL Params**

Required: None

Optional: If uploading an Excel optimization/simulation/decision table model use:
?worksheet=<name> where <name> is the name of the worksheet.

- **Headers**

Required: Authorization - Example: Authorization: bearer {your RASON token}

Optional: None

- **Data Params:**

An unnamed or named* RASON optimization, simulation, data science or decision table model, an unnamed or named data science fitted model, a data science PMML fitted model (with "xml" or "json" extension) or an Excel workbook. The model may be in "raw body" or "form-data" form. (See description above.)

*Note: If using the URLs <https://rason.net/api/model/{myModelName}> or <https://rason.net/api/model/{myModelID}>, modelName (if present) must be identical to {myModelName} or {myModelID}, whichever is being passed, otherwise a 400 (Bad Request) will be returned. Alternatively, modelName may be omitted from the RASON model.

```
{
  variables: {
    "d9:f9": {
      lower: 0, value: 0, finalValue: []
    }
  },
  data: {
    "d17:f17": {
      value: [[75, 50, 35]]
    }
  }
}
```

```

    "d11:f11": {
      value: [[1, 1, 0]]
    },
    "d12:f12": {
      value: [[1, 0, 0]]
    },
    "d13:f13": {
      value: [[2, 2, 1]]
    },
    "d14:f14": {
      value: [[1, 1, 0]]
    },
    "d15:f15": {
      value: [[2, 1, 1]]
    }
  },

  constraints: {
    c11: {
      formula: "sumproduct(d9:f9, d11:f11)",
      upper: 450
    },
    c12: {
      formula: "sumproduct(d9:f9, d12:f12)",
      upper: 250
    },
    c13: {
      formula: "sumproduct(d9:f9, d13:f13)",
      upper: 800
    },
    c14: {
      formula: "sumproduct(d9:f9, d14:f14)",
      upper: 450
    },
    c15: {
      formula: "sumproduct(d9:f9, d15:f15)",
      upper: 600
    }
  },

  objective: {
    d18: {
      type: "maximize",
      formula: "sumproduct(d9:f9, d17:f17)",
      finalValue: []
    }
  }
}

```

- **Success Response:**

Code: 201 (Created)

Response Example: (RASON data science model submitted as form-data as Text with content name "RASONModel" and two additional data files.)

```

{
  "ModelId": "2590+DecisionTreeClassification3+2020-06-01-21-20-00-698446",
  "ModelName": "DecisionTreeClassification3",
  "ModelDescr": "classification: decision tree",
  "ModelFiles": [
    {
      "fileName": "hald-small-binary-train.txt",
      "isOnServer": true
    },
    {
      "fileName": "hald-small-binary-valid.txt",
      "isOnServer": true
    }
  ]
}

```

```

        "fileName": "classification-decision-tree.xml",
        "isOnServer": false
    },
    {
        "fileName": "classification-decision-tree.json",
        "isOnServer": false
    }
],
"RuntimeToken": "",
"ModelType": "Origin",
"ModelKind": "RASON",
"IsChampion": false,
"ParentModelId": null,
"QueryString": ""
}

```

Response Example: (An Excel optimization model submitted as form-data as "File".)

```

{
    "ModelId": "2590+Blending (Opt)+2020-06-01-21-23-30-714655",
    "ModelName": "Blending (Opt) ",
    "ModelDescr": "",
    "ModelFiles": [],
    "RuntimeToken": "",
    "ModelType": "Origin",
    "ModelKind": "Excel",
    "IsChampion": false,
    "ParentModelId": null,
    "QueryString": ""
}

```

Response Example: (A data science fitted model submitted as raw body.)

```

{
    "ModelId": "2590+LogisticRegressionFitted1+2020-06-01-21-25-05-231970",
    "ModelName": "LogisticRegressionFitted1",
    "ModelDescr": "",
    "ModelFiles": [],
    "RuntimeToken": "",
    "ModelType": "Origin",
    "ModelKind": "Fitted",
    "IsChampion": false,
    "ParentModelId": null,
    "QueryString": ""
}

```

Response Example: (A portfolio optimization model submitted using <https://rasonrestapi2core.azurewebsites.net/api/model/PortfolioModel>. The property modelName was not present in the RASON model. The name of the model was obtained from the URL request, PortfolioModel.

```

{
    "ModelId": "2590+PortfolioModel+2024-01-04-15-01-29-496562",
    "ModelName": "PortfolioModel",
    "ModelDescr": "Quadratic RASON Model Example",
    "ModelFiles": [],
    "RuntimeToken": "",
    "ModelType": "Origin",
    "ModelKind": "RASON",
    "InstanceType": null,
    "IsChampion": false,
    "ParentModelId": null,
    "QueryString": "",
    "ModelContainer": null,
    "Permissions": null,
}

```

Response Example: (A portfolio optimization model submitted using <https://rasonrestapi2core.azurewebsites.net/api/model/987654321>. The property modelName was not present in the RASON model. The name of the model was obtained from the URL request, 987654321.

```

{

```



```

    "ModelId": "2590+987654321+2024-01-09-15-16-56-778314",
    "ModelName": "987654321",
    "ModelDescr": "Quadratic RASON Model Example",
    "ModelFiles": [],
    "RuntimeToken": "",
    "ModelType": "Origin",
    "ModelKind": "RASON",
    "InstanceType": null,
    "IsChampion": false,
    "ParentModelId": null,
    "QueryString": "",
    "ModelContainer": null,
    "Permissions": null,
  }

```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted. This error is returned when a non-unique model name is passed to "modelName", if it exists, within the RASON model.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

- **Notes:**

Attaching Data Files using POST: <https://rason.net/api/model>

The REST endpoint POST <https://rason.net/api/model> may also be used to submit one or more external data files, such as an Excel file, ms-access, mssql, odbc, or oracle database, OData endpoint, or a CSV, JSON, or XML file.

Below is an example that illustrates how to upload data files using JavaScript. The example code uses the POST rason.net/api/model endpoint to create a javascript 'FormData' object, set a special "RASONModel" field on the object and append the data files.

```

submitModelWithDataFiles: function () {
  var formData = new FormData(), me = this, request = new XMLHttpRequest();
  formData.append("RASONModel", this.RASONModel);
  for (var i = 0; i < this.modelDataFiles.length; ++i) {
    formData.append(this.modelDataFiles[i].name,
      this.modelDataFiles[i],
      this.modelDataFiles[i].name);
  }
  request.onreadystatechange = function () {
    if (this.readyState == 4) {
      if (this.status == 200 || this.status == 201) {
        // model has been posted to the RASON server. enqueue it
        // for processing
      }
    }
  }
  request.open("POST", "https://rason.net/api/model", true);
  request.send(formData);
}

```

```

        me.getModelId(this.response, this.statusText, this);
    } else {
        me.invokeFailed(this, this.statusText, null);
    }
}

};
//use the POST rason.net/api/model endpoint to create a
javascript FormData //object
request.open("POST", this.RASONServer + '/model');
request.setRequestHeader('Authorization',
this.ajaxOpts.headers['Authorization']);
request.send(formData);
}

```

See the RASON Reference Guide for information on how to access this data from within your RASON model via the `dataSources` section.

[Return to Asynchronous Endpoint List](#)

REST Endpoint: POST:
<https://rason.net/api/model/excel?connection-name=<NamedExcelConnection!Worksheet>&model-name=<MyExcelFlow>>

Executing this endpoint generates a single-stage decision flow with a stage invoking the reusable Excel model that is located on the user's OneDrive or SharePoint account using the named connection created on the user's www.RASON.com account. The resulting flow model is immediately POSTed on the server. See below for the response.

Note: ?connection-name query parameter is mandatory; “!Worksheet” is optional; model-name" query parameter is optional.

This endpoint assumes an existing named connection to the Excel Workbook on the OneDrive or SharePoint account, “MyExcelConnection”.

To view the actual flow (POSTed model), use GET rason.net/api/model/flow-MyExcelConnection.

Note the following:

1. If “!Worksheet” is omitted in the ?connection-name= query param, then the active sheet is considered
2. The generated flow acquires the flowName: “flow-”+”NamedExcelConnection” if "model-name" query parameter is omitted.
3. "model-name" is used to provide a flow name for the generated flow. If omitted, default "flow-" + <connection-name> is used.
4. The flowDescription property provides a comment on how this flow was generated and which data connection was used
5. Single stage is named “excelStage”
6. The property invokeModel uses the format: invokeModel: “Name=[NamedExcelConnection!Worksheet]”
7. POST rason.net/api/model/excel?connection-name=MyExcelConnection generates the updated flow and PUT rason.net/api/model/excel?connection-name=MyExcelConnection updates the flow for modelName “flow-MyExcelConnection”.

Once the Excel model is posted using this endpoint, users may refer to (“flow-MyExcelConnection”) in subsequent endpoint calls, i.e. POST rason.net/api/model/flow-MyExcelConnection/solve solves the model. The decision flow may be extended by adding parent/child stages.

invokeModel Format

The format of invokeModel: “Name=[NamedExcelConnection]” can be used in flows regardless of POST [rason.net/api/model/excel?connection-name=\[NamesExcelConnetion\]](https://rason.net/api/model/excel?connection-name=[NamesExcelConnetion]), which automatically generates the flow and POSTs the resulting model.

Note that RASON offers three formats of invokeModel:

1. Invoking RASON reusable models: invokeModel: “[modelName]”
2. Invoking Excel reusable models POSTed on the RASON server (i.e. direct solving without RASON script): invokeModel: “[modelName]:[workbook]![worksheet]”

3. Invoking Excel reusable models from OneDrive or SharePoint: invokeModel:
"Name=NamedExcelConnection!Worksheet"

- **URL**

`https://rason.net/api/model/excel?connection-name=<NamedExcelConnection!Worksheet>&model-name=<name>`

- **Method:**

POST

- **URL Params**

Required: ?connection-name

Optional:

- !Worksheet – If omitted, the active worksheet is considered
- model-name – Provides desired flow name for generated flow. If omitted, default "flow-" + <connection-name> is used.

Example: POST `rason.net/api/model/excel?connection-name=MyExcelConnection&model-name=MyExcelFlow`

- **Headers**

Required: Authorization - Example: `Authorization: bearer {your RASON token}`

Optional: None

- **Data Params:**

An existing named connection to the Excel Workbook on the user's account on www.RASON.com; OneDrive or SharePoint connection, "MyExcelConnection" under MyAccount. For more information on how to create a Named Connection on www.RASON.com, see the previous chapter, RASON Subscriptions.



- **Success Response:**

Code: 201 (Created)

Request Example: POST `rason.net/api/model/excel?connection-name=MyExcelConnection!ProductMixExample1`

Response Example:

```

{
  "ModelId": 227768+flow-MyExcelConnection+2020-06-15-00-28-
077645",
  "ModelName": "flow-MyExcelConnection",
  "ModelDescr": "This flow has been generated for an Excel model
at
  The location provided by data connection
  [MyExcelConnection]",
  "ModelFiles": [],
  "RunTimeToken": "",
  "ModelType": "Origin",
  "ModelKind": "RASON",
  "IsChampion": false,
  "ParentModelId": null,
  "QueryString": ""
}

```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

[Return to Asynchronous Endpoint List](#)

REST Endpoint: POST: <https://rason.net/api/stages>

POSTs a list of stages that exist in the decision flow, attached to the request body.

- **URL**

<https://rason.net/api/stages>

- **Method:**

POST

- **URL Params**

Required: None

Optional: None

- **Headers**

Required: Authorization - Example: Authorization: bearer {your RASON token}

Optional: None

- **Data Params**

A decision flow RASON Model. This decision flow includes two stages: optStage, which solves an optimization model, and simStage, which uses the final variable values from optStage as inputs to a simulation model.

```
{
  "flowName": "optSimWorkflow",
  "flowDescription": "This decision flow solves an optimization
    model that selects an optimal mix of capital improvement
    projects, then runs a simulation.",
  "optStage": {
    "comment": "Uncertain optimization by transformation",
    "modelName": "optModel",
    "modelSettings": {
      "transformStochastic": "robustCounterpart",
      "numTrials": 100,
      "randomSeed": 1
    },
  },
  "variables": {
    "x": {
      "dimensions": [8],
      "type": "binary",
      "initialValue": [],
      "finalValue": [],
      "dualValue": []
    }
  },
  "uncertainVariables": {
    "c": {
```

```

        "dimensions": [8]
    },
    "c[1]": {
        "formula": "PsiTriangular(400000, 500000, 900000)"
    },
    "c[2]": {
        "formula": "PsiTriangular(500000, 750000, 1250000)"
    },
    "c[3]": {
        "formula": "PsiTriangular(500000, 1000000, 1500000)"
    },
    "c[4]": {
        "formula": "PsiTriangular(400000, 600000, 900000)"
    },
    "c[5]": {
        "formula": "PsiTriangular(250000, 500000, 750000)"
    },
    "c[6]": {
        "formula": "PsiTriangular(300000, 500000, 600000)"
    },
    "c[7]": {
        "formula": "PsiTriangular(200000, 450000, 700000)"
    },
    "c[8]": {
        "formula": "PsiTriangular(400000, 500000, 700000)"
    },
    "d": {
        "dimensions": [8],
        "formula": "PsiBinomial(1, 0.9)"
    }
},
"formulas": {
    "f": {
        "value": [325000, 450000, 550000, 300000, 150000, 250000,
        150000, 325000]
    },
    "cash": {
        "formula": "sumproduct(c * d - f, x)"
    }
},
"constraints": {
    "invest": {
        "formula": "sumproduct(f, x)",
        "upper": 1500000
    }
},
"objective": {
    "totalObjective": {
        "type": "maximize",
        "formula": "cash",
        "chanceType": "ExpVal",
        "finalValue": []
    }
}
},
"simStage": {
    "comment": "Simulation at optimization result point",

```

```

"modelName": "simModel",
"inputParameters": {
  "x": {
    "value": "optStage.x.finalValue"
  }
},
"uncertainVariables": {
  "c": {
    "dimensions": [8]
  },
  "c[1]": {
    "formula": "PsiTriangular(400000, 500000, 900000)"
  },
  "c[2]": {
    "formula": "PsiTriangular(500000, 750000, 1250000)"
  },
  "c[3]": {
    "formula": "PsiTriangular(500000, 1000000, 1500000)"
  },
  "c[4]": {
    "formula": "PsiTriangular(400000, 600000, 900000)"
  },
  "c[5]": {
    "formula": "PsiTriangular(250000, 500000, 750000)"
  },
  "c[6]": {
    "formula": "PsiTriangular(300000, 500000, 600000)"
  },
  "c[7]": {
    "formula": "PsiTriangular(200000, 450000, 700000)"
  },
  "c[8]": {
    "formula": "PsiTriangular(400000, 500000, 700000)"
  },
  "d": {
    "dimensions": [8],
    "formula": "PsiBinomial(1, 0.9)",
    "mean": []
  }
},
"data": {
  "f": {
    "value": [325000, 450000, 550000, 300000, 150000, 250000,
              150000, 325000]
  }
},
"uncertainFunctions": {
  "cash": {
    "formula": "sumproduct(c * d - f, x)",
    "mean": []
  }
}
}

```

- **Example Request:** <https://rason.net/api/stages>

- **Success Response:**

Code: 200 (OK)

```
[
  "optStage",
  "simStage"
]
```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	REST API unable to map Client's URI to an unknown resource, i.e. passing an erroneous resource ID.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

[Return to Asynchronous Endpoint List](#)

REST Endpoint: POST: <https://rason.net/api/stages/{stage}>

POSTs information pertaining to the stage or stages that exist in the decision flow attached to the request body. {stage} refers to a specific stage name.

- **URL**

<https://rason.net/api/stages/{stage}>

- **Method:**

POST

- **URL Params**

Required: None

Optional: None

- **Headers**

Required: Authorization - Example: Authorization: bearer {your RASON token}

Optional: None

- **Data Params**

A decision flow RASON Model. This decision flow includes two stages: optStage, which solves an optimization model, and simStage, which uses the final variable values from optStage as inputs to a simulation model.

```
{
  "flowName": "optSimWorkflow",
  "flowDescription": "This decision flow solves an optimization
    model that selects an optimal mix of capital improvement
    projects, then runs a simulation.",
  "optStage": {
    "comment": "Uncertain optimization by transformation",
    "modelName": "optModel",
    "modelSettings": {
      "transformStochastic": "robustCounterpart",
      "numTrials": 100,
      "randomSeed": 1
    },
  },
  "variables": {
    "x": {
      "dimensions": [8],
      "type": "binary",
      "initialValue": [],
      "finalValue": [],
      "dualValue": []
    }
  },
}
```

```

    "uncertainVariables": {
      "c": {
        "dimensions": [8]
      },
      "c[1]": {
        "formula": "PsiTriangular(400000, 500000, 900000)"
      },
      "c[2]": {
        "formula": "PsiTriangular(500000, 750000, 1250000)"
      },
      "c[3]": {
        "formula": "PsiTriangular(500000, 1000000, 1500000)"
      },
      "c[4]": {
        "formula": "PsiTriangular(400000, 600000, 900000)"
      },
      "c[5]": {
        "formula": "PsiTriangular(250000, 500000, 750000)"
      },
      "c[6]": {
        "formula": "PsiTriangular(300000, 500000, 600000)"
      },
      "c[7]": {
        "formula": "PsiTriangular(200000, 450000, 700000)"
      },
      "c[8]": {
        "formula": "PsiTriangular(400000, 500000, 700000)"
      },
      "d": {
        "dimensions": [8],
        "formula": "PsiBinomial(1, 0.9)"
      }
    },
    "formulas": {
      "f": {
        "value": [325000, 450000, 550000, 300000, 150000, 250000,
          150000, 325000]
      },
      "cash": {
        "formula": "sumproduct(c * d - f, x)"
      }
    },
    "constraints": {
      "invest": {
        "formula": "sumproduct(f, x)",
        "upper": 1500000
      }
    },
    "objective": {
      "totalObjective": {
        "type": "maximize",
        "formula": "cash",
        "chanceType": "ExpVal",
        "finalValue": []
      }
    }
  },
},

```

```

"simStage": {
  "comment": "Simulation at optimization result point",
  "modelName": "simModel",
  "inputParameters": {
    "x": {
      "value": "optStage.x.finalValue"
    }
  },
  "uncertainVariables": {
    "c": {
      "dimensions": [8]
    },
    "c[1]": {
      "formula": "PsiTriangular(400000, 500000, 900000)"
    },
    "c[2]": {
      "formula": "PsiTriangular(500000, 750000, 1250000)"
    },
    "c[3]": {
      "formula": "PsiTriangular(500000, 1000000, 1500000)"
    },
    "c[4]": {
      "formula": "PsiTriangular(400000, 600000, 900000)"
    },
    "c[5]": {
      "formula": "PsiTriangular(250000, 500000, 750000)"
    },
    "c[6]": {
      "formula": "PsiTriangular(300000, 500000, 600000)"
    },
    "c[7]": {
      "formula": "PsiTriangular(200000, 450000, 700000)"
    },
    "c[8]": {
      "formula": "PsiTriangular(400000, 500000, 700000)"
    },
    "d": {
      "dimensions": [8],
      "formula": "PsiBinomial(1, 0.9)",
      "mean": []
    }
  },
  "data": {
    "f": {
      "value": [325000, 450000, 550000, 300000, 150000, 250000,
        150000, 325000]
    }
  },
  "uncertainFunctions": {
    "cash": {
      "formula": "sumproduct(c * d - f, x)",
      "mean": []
    }
  }
}
}

```

- **Example Request that POSTS the stage, optStage:** <https://rason.net/api/stages/optStage>

- **Success Response:**

Code: 200 (OK)

```
[
  {
    "name": "optStage",
    "type": "optimization",
    "reusableModel": "",
    "recency": "",
    "successors": [
      "simStage"
    ],
    "predecessors": [],
    "pipeline": [
      "optStage"
    ],
    "allResults": [
      "x.initialValue",
      "x.finalValue",
      "x.dualValue",
      "totalObjective.finalValue"
    ],
    "resultsForBindings": [
      "x.finalValue"
    ],
    "dataSources": [],
    "reusableModelInfo": {
      "name": "",
      "workbook": "",
      "worksheet": ""
    },
    "isTerminal": false,
    "fittedModel": {
      "name": "",
      "modelName": ""
    },
    "inputParameters": []
  }
]
```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	REST API unable to map Client's URI to an unknown resource, i.e. passing an erroneous resource ID.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.

Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.
--	--

[Return to Asynchronous Endpoint List](#)

REST Endpoint: POST:
<https://rason.net/api/model/{nameorid}/score>

The POST <https://rason.net/api/model/{nameorid}/score> endpoint produces the scores (prediction, transformation, forecast, etc.) for a fitted PMML/JSON model *POSTed to the RASON server* given the new data attached to the request as "form-data" consisting of one content part: "Data".

New data is expected to be either 1D (single record) or 2D (multiple records) JSON array compatible with the fitted model. (Note: Column headings are not supported for the new data.) The type of model and headers is inferred automatically. If the request is successful, the response will include the scores serialized in JSON array.

See the related API endpoint: POST <https://rason.net/api/score> which expects both the new data and the fitted model to be provided in the request body.

URL

<https://rason.net/api/model/{nameorid}/score>**Method:**

POST

- **URL Params**

Required: None

Optional: None

- **Headers**

Required: Authorization - Example: Authorization: bearer {your RASON token}

Optional: None

- **Data Params:** Form Data consisting of one content part, "Data".

Data

7 26 6 60 1

1 29 15 52 3

11 56 8 20 2

11 31 8 47 2

7 52 6 33 1

- **Success Response:**

Code: 200 (OK)

Example Request

POST <https://rason.net/api/model/LogisticRegression/score>

Note: LogisticRegression is the name of a fitted model already POSTed to the RASON server.

Example Response

The example response below contains the scored values.

```
[  
  "0",  
  "0",  
  "1",  
  "0",  
  "1"  
]
```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.

[Return to Asynchronous Endpoint List](#)

REST Endpoint: POST:

[https://rason.net/api/model/{nameorid}/{solvetype = datamine, decision, diagnose, optimize, simulation, or solve}](https://rason.net/api/model/{nameorid}/{solvetype=datamine,decision,diagnose,optimize,simulation,orsolve})

A REST API endpoint that binds model to data (optional), creates a new model instance, enqueues model to be solved (solvetype = datamine, decision, diagnose, optimize, solve or simulate) and returns a Location header with a new instance ID. The POST rason.net/api/model/{nameorid}/solve endpoint automatically creates an OData endpoint to allow for easy querying of the results. (Model must first be posted using POST <https://rason.net/api/model>.) A result with the following fields is returned.

- **modelId:** The unique identifier for the model
- **modelName:** The name of the model taken from the model text.
- **modelDescr:** The model description taken directly from the model text
- **modelFiles:** The list of files used by the model as taken from the model text.
- **runtimeToken:** Lists any runtime token associated with this model
- **modelType:** Lists the *type* of model such as "origin", "version", "instance".
 - “origin” – the initially posted model
 - “version” – means that this model is a version of a previously POSTed or PUT model
 - “instance” – these are models created whenever a model is run.
- **modelKind:** Lists the *kind* of model such as "fitted", "excel" or "RASON".
 - “fitted” – fitted Data science model in PMML/JSON format
 - “excel” – models defined in Excel language
 - “RASON” – models defined in RASON language
- **isChampion:** True if model is marked as the “Champion” using the REST API endpoint PATCH rason.net/api/model/{nameorid}.
- **parentModelId:** Identifies the model from which this model is derived. In the case of “Origin” models, this is always empty. Otherwise, this property identifies the model used to create this model. For example, if a user POST’s a model and then PUT’s another version of this model, “parentModelId” of the version will be the modelId of the originally posted model. Similar logic holds for posting versions of version and instances of version.
- **queryString:** The query string associated with the model when it was created. In the case of unnamed models, this will change throughout the model’s lifecycle. A user may post an unnamed model with a particular query string but then solve the model with another. In the case of named models, this is the query string associated with the action used to create the model. A model could be posted with one query string, have several versions with different query strings and associated instances with yet another query string. If a particular version or instance doesn’t have a query string, it will inherit the query string of the parent.

URL

<https://rason.net/api/model/{nameorid}/datamine>

<https://rason.net/api/model/{nameorid}/decision>

<https://rason.net/api/model/{nameorid}/diagnose>

`https://rason.net/api/model/{nameorid}/optimize`

`https://rason.net/api/model/{nameorid}/simulate`

`https://rason.net/api/model/{nameorid}/solve`

- **Method:**

POST

- **URL Params**

Required: None

- **Optional:** Any data component may be passed as a query parameter if a binding property exists for that parameter. For example, see the RASON example code snippet below.

```
data: {
  tktPrice: {
    value: [150],
    binding: 'get'
  }
}
```

To change a query parameter outside of the RASON model environment, use:

```
$.Post("https://rason.net/api/simulate?tkPrice=300", ...
```

RASON Server will map "tkPrice=?" with "tkPrice = 300". A query can return any number of rows that satisfy the filtering condition, from 0 to infinity.

To perform a query using multiple parameters (say custID, maritalStatus and age) outside of the RASON model environment, use the query parameter:

```
$.get(https://rason.net/api/decision?custID=c2&maritalStatus=s&age=40 .....
```

Or in general,

```
$.get(https://rason.net/api/decision?par1=val1&par2=val2.....
```

In a data science model, any *datasource* component may be passed as a query parameter if a binding property exists for that datasource.

```
"datasources": {
  "srcCustomers": {
    "type": "csv",
    "connection": "customers_dt.txt",
    "selection": "custID = $parCustID1 or custID = $parCustID2",
    "parameters": {
      "parCustID1": {
        "binding": "get",
        "value": "c1"
      },
      "parCustID2": {
        "binding": "get",
        "value": "c3"
      }
    }
  }
}
```

} ,

To query `custID=$parCustID1` or `custID=$parCustID2` outside of the RASON model environment, use:

```
$.get (https://rason.net/api/datamine?parCustID1=c1&parCustID2=c2...
```

Or in general,

```
$.get (https://rason.net/api/datamine?par1=val1&par2=val2.....
```

RASON Server will map "`custID=$parCustID1`" with "`parCustID1=c1`" and "`custID=$parCustID2`" with "`parCustID2=c3`". A query can return any number of rows that satisfy the filtering condition, from 0 to infinity.

This endpoint offers five optional parameters controlling the content of the final results and where intermediate results are stored: `stage`, `data-storage`, `keep-intermediate-results`, `simplify-final-results` and `response-format`.

- `Data-storage = DATABASE/JSON`

- This parameter controls whether the SQLite DATABASE or JSON file storage is used to store the results of *intermediate* stages. Database mode does not affect terminal stages. When database mode is selected, the in-memory SQLite database is used to store the results of intermediate stages and will spill on disk in out-of-memory conditions. The default setting is Database.

DATABASE mode can be orders of magnitude faster than JSON mode. In JSON mode, costs are incurred when reading/writing to/from disk. However, in DATABASE mode all in-memory database read/write operations are much more efficient. Intermediate results can still be stored and examined in DATABASE mode. These results may be queried with REST API or OData but they will *not* appear in the JSON response as serialized data frames.

Note: If `data-storage = DATABASE`, the results of intermediate stages (if kept) would be stored (and initially processed) only in SQLite DB. Therefore, even with `keep-intermediate-results=true`, you still will not see the actual evaluations for intermediate stages in the JSON response. You can only query these results using REST or OData APIs. (See below.) If results must be in JSON response, set `data-storage` to JSON. However, in any case with `keep-intermediate-results=true` the complete *status* object for each intermediate stage will be returned.

If `data-storage = database` and

- `keep-intermediate-results = False`, the database stays in memory and will be discarded after terminal stages are processed.
- `keep-intermediate-results = True`, an in-memory database is saved to disk to enable later querying.
- Example: POST: `https://rason.net/api/model/{nameorid}/solve?data-storage = json`
- Example: POST
`https://rason.net/api/model/{nameorid}/solve?data-storage=json&keep-intermediate-results=true`

- `Keep-intermediate-results = True/False`

- This parameter determines whether the RASON server stores the results of the *intermediate* stages in the decision flow, returns them with JSON response and makes them available for REST or OData querying later. Currently, this parameter not only affects the "pipeline solve" (i.e. when a single terminal stage is specified) but also the entire decision flow solve, which might have multiple terminal stages. The default setting is False.
 - Example: POST:
<https://rason.net/api/model/{nameorid}/solve?keep-intermediate-results=true>
- Response-format = STANDALONE/WORKFLOW
 - Use this parameter to switch between the default dataframe-based decision flow reporting format (for both single- and multi-stage decision flows) and the default reporting format (for single-stage decision flows only). When response-format= WORKFLOW and simplify-final-results = false, the formula outputs are reported as dataframes. The default setting is WORKFLOW.

 Note: If a formula refers to a decision table, the dataframe will be complete with column headers-as defined in the "outputs" section of the decision table definition, and property types – that will be available in both the JSON response and later in REST/OData queries. If a generic formula, RASON will use default column names and the best possible type for each result column.
 - Example: POST: <https://rason.net/api/model/{nameorid}/solve?response-format=standalone>
 - Example: POST: <https://rason.net/api/model/{nameorid}/solve?response-format=workflow>
- Schedule=interval/automatic
 - "interval" (ISO 8601 time or repeating time interval)- If time is specified, the RASON Server will run the model once on that day and time. If a repeating time interval is specified, the server will run the model at the starttime. After the model finishes and duration has elapsed, the server will run the model again for the specified number of repetitions. A new model instance is created each time the model is run. Applications will typically use a model name (rather than an ID) in API Calls to GET rason.net/api/model/{name}/status to refer to the most recent run.
 - "automatic" - This parameter behaves differently on standalone models versus decision flows. RASON Decision Services utilizes a new property, "modelRecency":"time-since-last-run", at the same level as the existing modelName, modelDescription and modelType properties.

 This new property informs the RASON server of how recently the model was run in order to determine if the results (in JSON or OData form) can be considered "current". The object "time-since-last run" must be in ISO 8601 time duration format (i.e. "PT12H" for 12 hours or "P1W" for 1 week). With this *property in place, the user may add "?schedule=automatic" to the existing REST API call POST rason.net/api/model/{name}/solve*. The RASON Server will consult past runs to determine the maximum time a solve usually takes to complete (max-time), and will arrange to run the model at intervals of time-since-last-run – max-time. If the "modelRecency" property is omitted, the

default value is infinity; '?schedule=automatic' will simply cause the model to run one time.

More importantly, when a workflow of multiple stages is run using POST `rason.net/api/model/{nameorid}/solve?schedule=automatic`, the RASON Server will take into account modelRecency information for each *stage* in the *workflow*. For example, suppose a workflow has a final stage dependent on two intermediate stages A and B: If both A and B have runs that satisfy their recency requirement, the RASON Server will just run the final stage, using the previous results from A and B. If A has model Recency=PT24H, takes 2 hours to run and last finished 23 hours ago, and B has modelRecency=Pt8H, takes 1 hour to run and last finished 6.5 hours ago, the server will re-run both A and B, and use the latest results (2 hours from now) to run the final stage. (The modelRecency property must be added to each stage in the decision flow.)

- Calling POST `rason.net/api/model/{name}/solve?keep-intermediate-results=true&schedule=automatic` solves the flow and persists the stage results on the server. These results may be accessed using the OData endpoint described in the next section.
 - To obtain the results from the last scheduled run, click the Editor tab at www.RASON.COM. The results will be listed under Results beneath the model name. In order to obtain results from an earlier scheduled run, click My Account, then Run Details on the Overview tab. Highlight and copy the Model ID of the desired model instance and use this Model ID while calling the REST API Endpoint, GET `rason.net/api/model/{nameorid}/result`.
- `Simplify-final-results = True/False`
 - When True, this parameter tells the engine to store and report the results of, only, terminal stages as simplest possible JSON object: a scalar for 1x1 dataframe, 1d - array for Nx1 dataframe and 2d - array for NxM dataframe, No headers or index columns are stored. This option does not affect the results of intermediate stages, these are always dataframes. These simple results are reported in JSON Response may be queried with a REST call. Currently OData does not recognize these objects. The default setting is False.
 - Example: POST: `https://rason.net/api/model/{nameorid}/solve?simplify-final-results = true`
 - `Stage=<stage-name>`
 - Use this parameter to retrieve information related to a specific stage of a decision flow. If this parameter is passed, the endpoint solves only the part of the decision flow required to solve the specified terminal stage, <stage-name>. The information produced by this optional parameter may be helpful during the creation or debugging phase. If a stage is not specified, a full Directed Acyclic Graph (DAG) solve is performed. To view all stages in a decision flow, click or use the REST endpoint GET: `https://rason.net/api/model/{nameorid}/stages/all`.
 - Example: POST: `https://rason.net/api/model/{nameorid}/solve?stage=imputation`

When submitted together, the two parameters, data-storage and keep-intermediate-results, create four cases.

Example: POST: <https://rason.net/api/model/{nameorid}/solve?data-storage=json&keep-intermediate-results=true>

- data-storage = JSON and keep-intermediate-results = True
 - SQLite database is not used to store results. Rather, JSON files are used for solving, storing, responses and querying. The results of intermediate and terminal stages are reported in JSON response as well as being available for REST/OData querying.
- data-storage = JSON and keep-intermediate-results = False
 - Same as when data-storage = JSON and keep-intermediate-results = True, except intermediate results are discarded after the decision flow solve. These results are not reported in JSON response and are not available for querying.
- data-storage = DATABASE and keep-intermediate-results = True
 - SQLite database is used for solving and storing the results of intermediate stages. Results are not returned within JSON response and are available for REST/OData querying only from the stored database. The results of terminal stages are initially stored in JSON files, reported within JSON response and available for REST/OData querying from stored JSON files.
- data-storage = DATABASE and keep-intermediate-results = False
 - Same as if data-storage = DATABASE and keep-intermediate-results = True except the database with intermediate results is discarded after the decision flow is solved. These results are not reported in JSON response and are not available for later querying.

Solving Excel Models in RASON

RASON Version 20.5 supports the solving of Excel models as standalone models or from within a stage in a decision flow. When solving an Excel model using RASON Services, most of the options and features of the POST rason.net/api/model/{nameorid}/solve endpoint is inherited. The unified structured response will be returned with indexed data frames available for OData querying, users can control *response-format=STANDALONE/WORKFLOW* to receive the old response format, *simplify-final-results=true/false* is to receive the JSON response with the objects coerced to the simplest possible type. Scheduling is also available. Note: When using this endpoint to post and solve an Excel model, the Excel model is uploaded and stored on the RASON Server.

When solving an Excel model, use *?worksheet=<worksheet-name>* to specify the name of the worksheet containing the model to be solved. If omitted, the active sheet is considered.

Note: Options such as *stage*, *keep-intermediate-results* and *data-storage* are not supported when solving an Excel model.

- **Headers**

Required: Authorization - Example: Authorization: bearer {your RASON token}

Optional: None

- **Data Params:** None

- **Success Response:**

Code: 202 (Accepted)

Response Example:

```
{
  "ModelId": "2590+dmMLRWorkflow+2020-06-09-02-44-22-635754",
  "ModelName": "dmMLRWorkflow",
  "ModelDescr": "",
  "ModelFiles": [],
  "RuntimeToken": "",
  "ModelType": "Instance",
  "ModelKind": "Excel",
  "IsChampion": false,
  "ParentModelId": "2590+dmMLRWorkflow+2020-06-09-00-18-07-773177",
  "QueryString": ""
}
```

Currently we have the following behavior in JSON response for the flow solves:

1. Overall flow “status” contains the id of the actual model instance that was created after calling `api/model/x/solve`
2. Stages “status” object has the id that corresponds to a model on the server:
 - a. For reusable stages, the id refers to origin/version resource id of the RASON/excel reusable model
 - b. For inline stages, the id refers to the resource id of the enclosing flow model instance. [Since there’s no way to tell whether the stage was inline or reusable based on the response, the only way to locate the inline stage model is to refer to the enclosing flow.]

However, this change has removed the information about when the particular stage was solved (previously it was a timestamp part of the generated resource id). Hence, I’ve added the additional field “solveTimestamp”:

1. Overall flow “status” shows “solveTimestamp” indicating the moment at time when the entire flow solve was done. This timestamp is different from the timestamp in the flow id – the latter indicates when the model instance was created and not when the flow was solved.
2. Stages “status” shows “solveTimestamp” indicating the moment at time when the stage was solved. Again, it’s different from the timestamp in the stage id – the latter indicates when the origin/version of the reusable model was created

Finally, given the “solveTimestamp” and “solveTime”, it’s easy to figure out the solve starting time for both the flow and each stage, if ever needed.

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.

Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.
--	--

[Return to Asynchronous Endpoint List](#)

REST Endpoint: POST:
https://rason.net/api/model/{nameorid}/stop

A REST API endpoint that stops a run for the model instance.

- **URL**

https://rason.net/api/model/{nameorid}/stop

- **Method:**

POST

- **URL Params**

Required: None

Optional: None

- **Headers**

Required: Authorization - Example: Authorization: bearer {your RASON token}

Optional: None

- **Data Params:** None

- **Success Response:**

Code: 200 (OK)

Response Example:

None

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

[Return to Asynchronous Endpoint List](#)

Rest Endpoint: PUT:https://rason.net/api/model/{nameorid}

A REST API endpoint that updates a previously posted named or unnamed RASON, Excel or JSON/PMML fitted models on the server, using the resource ID. When editing a previously posted model that requires a data upload, data **must** be re-uploaded with call to PUT <https://rason.net/api/model/{nameorid}>.

If updating a named model, a Location header is returned with a new resource id for the new version.

If updating an unnamed model, "Model has been updated" is returned in the body of the response; no new version is created, the model is simply updated.

- **URL**

<https://rason.net/api/model/{nameorid}>

- **Method:**

PUT

- **URL Params**

Required: None

Optional: None

- **Headers**

Required: Authorization - Example: Authorization: bearer {your RASON token}

Optional: None

- **Data Params:** An edited RASON Model

```
{
  modelName: "TestModel11",
  modelType: "optimization",
  variables: {
    "d9:f9": {
      lower: 0, value: 0, finalValue: []
    }
  },
  data: {
    "d17:f17": {
      value: [[75, 50, 35]]
    },
    "d11:f11": {
      value: [[1, 1, 0]]
    },
    "d12:f12": {
      value: [[1, 0, 0]]
    }
  },
}
```

```

    "d13:f13": {
      value: [[2, 2, 1]]
    },
    "d14:f14": {
      value: [[1, 1, 0]]
    },
    "d15:f15": {
      value: [[2, 1, 1]]
    }
  },
  constraints: {
    c11: {
      formula: "sumproduct(d9:f9, d11:f11)",
      upper: 450
    },
    c12: {
      formula: "sumproduct(d9:f9, d12:f12)",
      upper: 250
    },
    c13: {
      formula: "sumproduct(d9:f9, d13:f13)",
      upper: 800
    },
    c14: {
      formula: "sumproduct(d9:f9, d14:f14)",
      upper: 450
    },
    c15: {
      formula: "sumproduct(d9:f9, d15:f15)",
      upper: 600
    }
  },
  objective: {
    d18: {
      type: "maximize",
      formula: "sumproduct(d9:f9, d17:f17)",
      finalValue: []
    }
  }
}

```

- **Success & Response Examples:**

Named Model

Code: 201 (Created)

Location: <https://rason.net/api/model/model/2590+TestModel11+2019-11-22-21-54-04-233385>

Unnamed Model

Code: 200 (OK)

"Model has been updated"

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
------------------------------	---

Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

[Return to Asynchronous Endpoint List](#)

REST Endpoint: PUT:

<https://rason.net/api/model/excel?connection-name=<NamedExcelConnection!Worksheet>>

A REST API endpoint that updates a previously posted single-stage decision flow POSTed using POST rason.net/api/model/excel?connection-name=<NamedExcelConnection!Worksheet>.

- **URL**

<https://rason.net/api/model/excel?connection-name=<NamedExcelConnection!Worksheet>>

- **Method:**

PUT

- **URL Params**

Required: ?connection-name

Optional:

- !Worksheet – If omitted, the active worksheet is considered
- model-name – Provides desired flow name for generated flow. If omitted, default "flow-" + <connection-name> is used.

Example: PUT rason.net/api/model/excel?connection-name=MyExcelConnection&model-name=MyExcelFlow

- **Headers**

Required: Authorization - Example: Authorization: bearer {your RASON token}

Optional: None

- **Data Params:**

An existing named connection to the Excel Workbook on the user's account on [www.RASON.com](https://www.rason.com). OneDrive “MyExcelConnection” under MyAccount. For more information on how to create a Named Connection on [www.RASON.com](https://www.rason.com), see the previous chapter, RASON Subscriptions.



- **Success & Response Examples:**

Code: 201 (Created)

```
{
  "ModelId": "191227+Flow_July6+2020-07-06-05-01-11-945581",
  "ModelName": "Flow_July6",
  "ModelDescr": "This flow has been generated for an Excel mode
l at

the location provided by data connection [myExcelConnection2]",
  "ModelFiles": [],
  "RuntimeToken": "",
  "ModelType": "Version",
  "ModelKind": "RASON",
  "IsChampion": false,
  "ParentModelId": "191227+Flow_July6+2020-07-06-05-00-19-
996032",
  "QueryString": ""
}
```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

[Return to Asynchronous Endpoint List](#)

RASON Decision Flow Endpoints

The API endpoints described above for *PUT/POST api/model/excel?connection-name* generate and post decision flows invoking Excel models saved in a OneDrive or Sharepoint account. The following endpoints extend this functionality to all RASON and Excel models – residing either on the RASON Server or on an OneDrive or SharePoint account or attached to the body of the request.

GET/POST rason.net/api/flow

GET *rason.net/api/flow?connection-name={connection name}&model-name={model name}&worksheet={worksheet name}*

POST *rason.net/api/flow?connection-name={connection name}?model-name={model name}&worksheet={worksheet name}*

GET/POST/PUT rason.net/api/{nameorid}/flow

GET *rason.net/api/model/{nameorid}/flow?model-name={model name}&worksheet={worksheet name}*

POST *rason.net/api/model/{nameorid}/flow?model-name={model name}&worksheet={worksheet name}*

PUT *rason.net/api/model/{nameorid}/flow?model-name={model name}&worksheet={worksheet name}*

GET/POST/PUT rason.net/api/flow

GET *rason.net/api/model/flow?connection-name={connection name}&model-name={model name}*

POST *rason.net/api/model/flow?connection-name={connection name}&model-name={model name}*

PUT *rason.net/api/model/flow?connection-name={connection name}&model-name={model name}*

REST Endpoint: GET

https://rason.net/api/flow?connection-name=<connection_name>&model-name=<model_name>&worksheet=<worksheet_name>

A REST API endpoint that returns the generated flow in the response for the model stored on a OneDrive or SharePoint account or for the model attached in the body of the request.

- **URL**

`https://rason.net/api/flow?connection-name=<connection_name>&model-name=<model_name>&worksheet=<worksheet_name>`

- **Method:**

GET

- **URL Params**

Required:

- connection-name [string, required] - If model is stored on the user's OneDrive or Sharepoint account, this query parameter is required to specify the name of the data connection defined on the user's RASON account. If the model is not stored on the user's OneDrive or Sharepoint account, this query is not used.

Optional:

- model-name [string, optional] – Specifies the flowName for the generated flow. If omitted, default "flow-" + <model-name> is used.
- worksheet [string, optional, default=""] (active worksheet) – Selectes the specific worksheet for an Excel model. If omitted, the active worksheet is used.

Example: GET <https://rason.net/api/flow?connection-name=MyOneDriveConnection&model-name=AirpassWorkflow&worksheet=Data>

- **Headers**

Required: Authorization - Example: Authorization: bearer {your RASON token}

Optional: None

- **Data Params:**

An existing named connection to the Excel Workbook on the user's account, at www.RASON.com, a OneDrive or SharePoint “MyExcelConnection” under MyAccount.

For more information on how to create a Named Connection on www.RASON.com, see the previous chapter, RASON Subscriptions.

- **Success & Response Examples:**

Code: 200 (OK)

```
{
  "flow": "Airpass Workflow",
  "doubleExponential-1053": {
    "datasources": {
      "msExcelSrc": {
        "type": "excel",
        "connection": "name=MyOneDriveConnection!Data",
        "content": "time-series",
        "selection": "Data!$A$2:$B$145",
        "headerExists": true
      },
      "myNewSrc": {
        "type": "excel",
        "connection": "name=MyOneDriveConnection!Data",
        "selection": "Data!$A$2:$B$145",
        "headerExists": true
      }
    },
    "datasets": {
      "myData": {
        "binding": "msExcelSrc"
      },
      "myNewData": {
        "binding": "myNewSrc",
        "selectedCols": [
          "Passengers"
        ]
      }
    },
    "modelName": "doubleExponential-1053",
    "modelType": "datamining",
    "estimator": {
      "doubleExponential-1053": {
        "type": "timeSeries",
        "algorithm": "doubleExponential",
        "parameters": {
          "optimize": true
        }
      }
    },
    "actions": {
      "doubleExponential-1053Model": {
        "data": "myData",
        "estimator": "doubleExponential-1053",
        "action": "fit",
        "evaluations": [
          "fittedModelJson",
          "coefficientsInfo"
        ]
      },
      "fitted": {
        "data": "myData",
        "fittedModel": "doubleExponential-1053Model",
        "action": "transform",
        "evaluations": [
          "transformation",

```

```

        "residuals",
        "sse",
        "mse",
        "mape",
        "mad",
        "cfe",
        "mfe",
        "tse",
        "tsPlot"
    ]
},
"forecasted": {
    "data": "myData",
    "fittedModel": "doubleExponential-1053Model",
    "action": "forecast",
    "parameters": {
        "numForecasts": 1
    },
    "evaluations": [
        "forecast",
        "difference",
        "autocovariance",
        "autocorrelation",
        "partialAutocorrelation"
    ]
},
"ext-comp-1058": {
    "data": "myNewData",
    "fittedModel": "doubleExponential-1053Model",
    "action": "transform",
    "evaluations": [
        "transformation"
    ]
}
}
}

```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

[Return to Decision Flow Endpoints List](#)

REST Endpoint: POST

https://rason.net/api/flow?connection-name=<connection_name>&model-name=<model_name>&worksheet=<worksheet_name>

A REST API endpoint that returns the generated flow in the response for the RASON/Excel model attached in the request body.

- **URL**

`https://rason.net/api/flow?connection-name=<connection_name>&model-name=<model_name>&worksheet=<worksheet_name>`

- **Method:**

POST

- **URL Params**

Required:

- connection-name [string, optional] - If model is stored on the user's OneDrive or Sharepoint account, this query parameter is required to specify the name of the data connection defined on the user's RASON account. If model is not stored on a OneDrive/Sharepoint account, this query parameter is not used.

Optional:

- worksheet = <worksheet_name> – If omitted, the active worksheet is considered
- model-name = <model_name> – Provides desired flow name for generated flow. If omitted, default "flow-" + <model-name> is used.

Example: POST `rason.net/api/flow?model-name=MyExcelFlow&Worksheet=MyWorksheet1`

- **Headers**

Required: Authorization - Example: `Authorization: bearer {your RASON token}`

Optional: None

- **Data Params:**

A RASON or Excel model attached in the body of the request.

- **Success & Response Examples:**

Code: 201 (OK)

Example Response for Excel model attached to the body of the request.

```
{
  "flowName": "CollegeFundModel2",
  "flowDescription": "This flow has been generated for 'CollegeFundMode
```

```

    12' model",
    "CollegeFundGrowth2(Sim).xlsxStage": {
      "comment": "This interface has been generated by Psi fo

r an Excel mdl in the workbook CollegeFundGrowth2(Sim).xlsx",
      "invokeModel": "CollegeFundGrowth2(Sim).xlsx",
      "modelType": "simulation",
      "modelDescription": "",
      "outputResults": {
        "g40": {
          "evaluations": [
            "trials"
          ],
          "type": "array/number",
          "comment": "uncertain function"
        }
      }
    }
  }
}

```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

[Return to Decision Flow Endpoints List](#)

REST Endpoint: GET

[https://rason.net/api/model/{nameorid}/flow](https://rason.net/api/model/{nameorid}/flow?model-name=<model_name>&worksheet=<worksheet_name>)
[?model-name=<model_name>](https://rason.net/api/model/{nameorid}/flow?model-name=<model_name>&worksheet=<worksheet_name>)
[&worksheet=<worksheet_name>](https://rason.net/api/model/{nameorid}/flow?model-name=<model_name>&worksheet=<worksheet_name>)

A REST API endpoint that returns the generated flow for the RASON or Excel model posted on the user's account. If valid model name is passed, the champion or the most recent version of the model is used. If a valid model id is passed, the specific version identified by the id is used.

- **URL**

`https://rason.net/api/model/{name_or_id}/flow?model-name=<model_name>&worksheet=<worksheet_name>`

- **Method:**

GET

- **URL Params**

Required: none

Optional:

- model-name [string, optional] – Specifies the flowName for the generated flow. If missing, a default name is assigned.
- worksheet [string, optional, default=""] (active worksheet) – Selects the specific worksheet for the Excel model. If omitted, the active worksheet is used.

Example: GET

`rason.net/api/model/CollegeFundGrowth1(Sim)/flow?model-name=CollegeFund1&Worksheet=College Fund Growth`

- **Headers**

Required: Authorization - Example: Authorization: bearer {your RASON token}

Optional: None

- **Data Params:**

An existing RASON or Excel model posted to the user's RASON account.

Success & Response Examples:

Code: 200 (OK)

```
{
  "flowName": "CollegeFund1",
  "flowDescription": "This flow has been generated for 'College Fund1' model",
  "CollegeFundGrowth1(Sim) Stage": {
    "comment": "This interface has been generated by Psi for an Exc
```

```

    el model in the workbook CollegeFundGrowth1(Sim).xlsx",
    "invokeModel": "CollegeFundGrowth1(Sim)",
    "modelType": "simulation",
    "modelDescription": "",
    "outputResults": {
      "g41": {
        "evaluations": [
          "mean"
        ],
        "type": "number",
        "comment": "uncertain function"
      }
    }
  }
}

```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

[Return to Decision Flow Endpoints List](#)

REST Endpoint: POST

[https://rason.net/api/model/{nameorid}/flow
?model-name=<model_name>
&worksheet=<worksheet_name>](https://rason.net/api/model/{nameorid}/flow?model-name=<model_name>&worksheet=<worksheet_name>)

A REST API endpoint that generates and posts the generated flow for the RASON or Excel model on the user's account. If a valid model name is passed, the champion or the most recent version of the model is used. If a valid model id is passed, the specific version identified by the id is used.

- **URL**

`https://rason.net/api/model/{name_or_id}/flow?model-
name=<model_name>&worksheet=<worksheet_name>`

- **Method:**

POST

- **URL Params**

Required: none

Optional:

- `model-name`[string, optional] – Specifies the `flowName` for the generated flow. If missing, a default name is assigned.
- `worksheet` [string, optional, default=""] (active worksheet) – Selects the specific worksheet for the Excel model. If omitted, the active worksheet is used.

Example: POST

`rason.net/api/model/CollegeFundGrowth1(Sim)/flow?model-
name=CollegeFund1&Worksheet=College Fund Growth`

- **Headers**

Required: Authorization - Example: Authorization: bearer {your RASON token}

Optional: None

- **Data Params:**

An existing RASON or Excel model posted to the user's RASON account.

Success & Response Examples:

Code: 201 (Created)

```
{  
  "ModelId": "2590+flow-AirlineCrewScheduling(Opt)+2020-09-16-  
19-16-  
35-337048",  
  "ModelName": "flow-AirlineCrewScheduling(Opt)",  
  "ModelDescr": "This flow has been generated for 'AirlineCrewS  
chedul
```

```

    ing(Opt)' model",
    "ModelFiles": [],
    "RuntimeToken": "",
    "ModelType": "Origin",
    "ModelKind": "RASON",
    "IsChampion": false,
    "ParentModelId": null,
    "QueryString": ""
  }

```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

[Return to Decision Flow Endpoints List](#)

REST Endpoint: PUT

`https://rason.net/api/model/{nameorid}/flow?model-name=<model name>&worksheet=<worksheet name>`

A REST API endpoint that generates and modifies the generated flow for the RASON or Excel model posted on the user's account. If a valid model name is passed, the champion or the most recent version of the model is used. If a valid model id is passed, the specific version identified by the id is used.

- **URL**

`https://rason.net/api/model/{name_or_id}/flow?model-name=<model_name>&worksheet=<worksheet_name>`

- **Method:**

PUT

- **URL Params**

Required: none

Optional:

- model-name [string, optional] – Specify the flowName for the generated flow. If missing, a default name is assigned.
- worksheet [string, optional, default=""] (active worksheet) – Selects the specific worksheet for the Excel model. If omitted, the active worksheet is considered

Example: PUT

`rason.net/api/model/CollegeFundGrowth1(Sim)/flow?model-name=CollegeFund1&Worksheet=College Fund Growth`

- **Headers**

Required: Authorization - Example: Authorization: bearer {your RASON token}

Optional: None

- **Data Params:**

An existing RASON or Excel model posted to the user's RASON account.

Success & Response Examples:

Code: 201 (Created)

```
{
  "ModelId": "2590+flow-AirlineCrewScheduling (Opt)+2020-09-16-19-27-42-967965",

```

```

    "ModelName": "flow-AirlineCrewScheduling(Opt)",
    "ModelDescr": "This flow has been generated for 'AirlineCrewScheduling(Opt)' model",
    "ModelFiles": [],
    "RuntimeToken": "",
    "ModelType": "Version",
    "ModelKind": "RASON",
    "IsChampion": false,
    "ParentModelId": "2590+flow-AirlineCrewScheduling(Opt)+2020-09-16-19-16-35-337048",
    "QueryString": ""
  }

```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

[Return to Decision Flow Endpoints List](#)

REST Endpoint: GET

https://rason.net/api/model/flow?connection-name=<connection_name>&model-name=<model_name>

A REST API endpoint that composes the single-stage flow invoking a RASON/Excel model stored on the user's OneDrive/SharePoint account. GET returns the generated flow in the response.

- **URL**

`https://rason.net/api/model/flow?connection-name=<connection_name>&model-name=<model_name>`

- **Method:**

GET

- **URL Params**

Required:

- connection-name [string, required] – Name of data connection defined on the user's RASON account.

Optional:

- Model-name [string, optional] – Specifies the flowName for the generated flow. If missing, a default name is assigned.

Example: `GET rason.net/api/model/flow?connection-name=MyOneDriveConnection&model-name=Airpass Workflow`

- **Headers**

Required: Authorization - Example: `Authorization: bearer {your RASON token}`

Optional: None

- **Data Params:**

An existing RASON or Excel model posted to the user's RASON account.

Success & Response Examples:

Code: 200 (OK)

```
{
  "flow": "Airpass Workflow",
  "doubleExponential-1053": {
    "datasources": {
      "msExcelSrc": {
        "type": "excel",
        "connection": "name=MyOneDriveConnection",
        "content": "time-series",
```

```

        "selection": "Data!$A$2:$B$145",
        "headerExists": true
    },
    "myNewSrc": {
        "type": "excel",
        "connection": "name=MyOneDriveConnection",
        "selection": "Data!$A$2:$B$145",
        "headerExists": true
    }
},
"datasets": {
    "myData": {
        "binding": "msExcelSrc"
    },
    "myNewData": {
        "binding": "myNewSrc",
        "selectedCols": [
            "Passengers"
        ]
    }
},
"modelName": "doubleExponential-1053",
"modelType": "datamining",
"estimator": {
    "doubleExponential-1053": {
        "type": "timeSeries",
        "algorithm": "doubleExponential",
        "parameters": {
            "optimize": true
        }
    }
},
"actions": {
    "doubleExponential-1053Model": {
        "data": "myData",
        "estimator": "doubleExponential-1053",
        "action": "fit",
        "evaluations": [
            "fittedModelJson",
            "coefficientsInfo"
        ]
    },
    "fitted": {
        "data": "myData",
        "fittedModel": "doubleExponential-1053Model",
        "action": "transform",
        "evaluations": [

```

```

        "transformation",
        "residuals",
        "sse",
        "mse",
        "mape",
        "mad",
        "cfe",
        "mfe",
        "tse",
        "tsPlot"
    ]
},
"forecasted": {
    "data": "myData",
    "fittedModel": "doubleExponential-1053Model",
    "action": "forecast",
    "parameters": {
        "numForecasts": 1
    },
    "evaluations": [
        "forecast",
        "difference",
        "autocovariance",
        "autocorrelation",
        "partialAutocorrelation"
    ]
},
"ext-comp-1058": {
    "data": "myNewData",
    "fittedModel": "doubleExponential-1053Model",
    "action": "transform",
    "evaluations": [
        "transformation"
    ]
}
}
}
}

```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.

Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

[Return to Decision Flow Endpoints List](#)

REST Endpoint: POST

https://rason.net/api/model/flow?connection-name=<connection_name>&model-name=<model_name>

A REST API endpoint that composes the single-stage flow invoking a RASON/Excel model stored on the user's OneDrive/SharePoint account. POST generates and publishes the flow model to the RASON user account.

- **URL**

`https://rason.net/api/model/flow?connection-name=<connection_name>&model-name=<model_name>`

- **Method:**

POST

- **URL Params**

Required:

- connection-name [string, required] – Name of data connection defined on the user's RASON account.

Optional:

- Model-name [string, optional]– Specifies the flowName for the generated flow. If omitted, a default name is assigned.

Example: POST `rason.net/api/model/flow?connection-name=MyOneDriveConnection&model-name=Airpass Workflow`

- **Headers**

Required: Authorization - Example: `Authorization: bearer {your RASON token}`

Optional: None

- **Data Params:**

A RASON or Excel decision flow or standalone model.

Success & Response Examples:

Code: 201 (Created)

Example Request

POST `rason.net/api/model/flow?connection-name=MyOneDriveConnection&model-name=Airpass Workflow`

Example Response

```
{
  "ModelId": "2590+Airpass Workflow+2020-09-16-20-40-13-623361",
}
```

```

    "ModelName": "Airpass Workflow",
    "ModelDescr": "",
    "ModelFiles": [
      {
        "fileName": "name=MyOneDriveConnection",
        "isOnServer": false
      }
    ],
    "RuntimeToken": "",
    "ModelType": "Origin",
    "ModelKind": "RASON",
    "IsChampion": false,
    "ParentModelId": null,
    "QueryString": ""
  }
}

```

Example Request

<https://rason.net/api/model/flow?connection-name=MyOneDriveConnection&model-name=optSimWorkflow>

Example Response

```

{
  "ModelId": "2590+optSimWorkflow+2020-10-23-14-25-47-548990",
  "ModelName": "optSimWorkflow",
  "ModelDescr": "",
  "ModelFiles": [
    {
      "fileName": "name=MyOneDriveConnection",
      "isOnServer": false
    }
  ],
  "RuntimeToken": "",
  "ModelType": "Origin",
  "ModelKind": "RASON",
  "IsChampion": false,
  "ParentModelId": null,
  "QueryString": ""
}

```

Example Request

[https://rason.net/api/model/flow?connection-name=OneDriveConnection2&model-name=CollegeFundGrowth1 \(Sim\) Test2](https://rason.net/api/model/flow?connection-name=OneDriveConnection2&model-name=CollegeFundGrowth1 (Sim) Test2)

Example Response

```

{
  "ModelId": "2590+CollegeFundGrowth1 (Sim) Test2+2020-10-27-16-13-29-014896",
  "ModelName": "CollegeFundGrowth1 (Sim) Test2",
  "ModelDescr": "This flow has been generated for an Excel model at the location provided by data connection [OneDriveConnection2]",
  "ModelFiles": [],
  "RuntimeToken": "",

```



```

    "ModelType": "Origin",
    "ModelKind": "RASON",
    "IsChampion": false,
    "ParentModelId": null,
    "QueryString": ""
  }

```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

[Return to Decision Flow Endpoints List](#)

REST Endpoint: PUT

https://rason.net/api/model/flow?connection-name=<connection_name>&model-name=<model_name>

A REST API endpoint that composes the single-stage flow invoking a RASON/Excel model stored on the user's OneDrive/SharePoint account. PUT generates and modifies the flow model in the RASON user account.

- **URL**

`https://rason.net/api/model/flow?connection-name=<connection_name>&model-name=<model_name>`

- **Method:**

PUT

- **URL Params**

Required:

- connection-name [string, required] – Name of data connection defined on the user's RASON account.

Optional:

- model-name [string, optional] – Provides desired flow name for generated flow. If omitted, a default name is assigned.

Example: `PUT rason.net/api/model/flow?connection-name=MyOneDriveConnection&model-name=Airpass Workflow`

- **Headers**

Required: Authorization - Example: `Authorization: bearer {your RASON token}`

Optional: None

- **Data Params:**

An existing RASON or Excel model posted to the user's RASON account.

Success & Response Examples:

Code: 201 (Created)

```
{
  "ModelId": "2590+Airpass Workflow+2020-09-16-20-50-46-232611",
  "ModelName": "Airpass Workflow",
  "ModelDescr": "",
  "ModelFiles": [
    {
```

```

        "fileName": "name=MyOneDriveConnection",
        "isOnServer": false
    }
],
"RuntimeToken": "",
"ModelType": "Version",
"ModelKind": "RASON",
"IsChampion": false,
"ParentModelId": "2590+Airpass Workflow+2020-09-16-20-40-13-623361",
"QueryString": ""
}

```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

[Return to Decision Flow Endpoints List](#)

RASON OData Endpoints

When the REST API endpoint `POST rason.net/api/model/{nameorid}/solve` is used to solve a decision flow or model (of any type), an OData endpoint is automatically created. This new OData endpoint for the RASON Rest Server enables users to retrieve (read) the results of a given RASON Decision flow or model, persisted on Azure Storage, as an OData feed with query support. The OData Web API does not interact with the existing REST Web API on the server. The implementation relies on ASP.NET OData Web API for OData 4 standards and Entity Framework. Three data storage layers are supported for RASON Decision flows for OData endpoints: JSON, memory and file. Results and behavior are the same for all three types of storage.

RASON Decision Flows Model Response

The response of each RASON decision flow or model execution is a JSON object – a collection of requested evaluations represented as DataFrames. If the data storage layer is set to JSON, then each resulting dataframe is serialized into JSON format and provided with the response file. If data storage layer is memory (DB_Memory) or file (DB_FILE), then the JSON response file does not contain the actual data, but rather serves as the metadata that stores and provides only the SQLite table names, which are stored in the separate SQLite DB file.

Entity Data Models (EDM)

Navigating to the Service Root URL results in the following response, illustrating a simplified view of OData service metadata.

```
{
  "@odata.context": "https://rason.net/odata/$metadata",
  "value": [
    {
      "name": "Result",
      "kind": "EntitySet",
      "url": "Result"
    },
    {
      "name": "DataInstance",
      "kind": "EntitySet",
      "url": "DataInstance"
    }
  ]
}
```

This is the representation of a single RASON Decision flow or model result through the Entity Data Models.

Result *Entity Set* represents a single named results within the model response. It is a structured Entity Type with two elements and can be viewed as a named Dataframe.

- Name: STRING [key, required]
- Data: Collection<DataInstance>.[required]

DataInstance *Entity Set* represents a single entity within the result. It's an *Open Entity Type*, since it defines the dynamic properties that are undeclared and are not fixed in advance. It has two elements and can be viewed as a single row of an arbitrary Dataframe, defined by Result.Data.

- _ID: int [key, required]
- Properties: dynamic

Dynamic properties represent the named columns of the Dataframe and are not known in advance since the schema of RASON Decision flows results/evaluations are not fixed.

Note: Dataframes may not be accessed directly since the RASON server relies on the ASP.NET LINQ provider to implement the IQueryable interface which enables the translation and execution of the OData queries on Entity Data Models internally, using Entity Framework functionality.

Metadata Endpoint

Use the [https://rason.net/odata/\\$metadata](https://rason.net/odata/$metadata) endpoint to produce an EDMX document in XML format that contains a complete description of the feeds, types, properties and relationships exposed by the service in Entity Data Models.

Response:

```
<?xml version="1.0" encoding="utf-8"?>
<edmx:Edmx Version="4.0" xmlns:edmx="http://docs.oasis-open.org/odata/ns/edmx">
  <edmx:DataServices>
    <Schema Namespace="RASONOData" xmlns="http://docs.oasis-open.org/odata/ns/edm">
      <EntityType Name="Result">
        <Key>
          <PropertyRef Name="Name" />
        </Key>
        <Property Name="Name" Type="Edm.String" Nullable="false" />
        <NavigationProperty Name="Data" Type="Collection(RASONOData.DataInstance)" />
      </EntityType>
      <EntityType Name="DataInstance" OpenType="true">
        <Key>
          <PropertyRef Name="_ID" />
        </Key>
        <Property Name="_ID" Type="Edm.Int32" Nullable="false" />
      </EntityType>
      <EntityType Name="ODataMetrics">
        <Key>
          <PropertyRef Name="ID" />
        </Key>
        <Property Name="ID" Type="Edm.Int32" Nullable="false" />
        <Property Name="SolverUserID" Type="Edm.Int32" Nullable="false" />
        <Property Name="RequestStartTime" Type="Edm.DateTimeOffset" Nullable="false" />
        <Property Name="RequestDuration" Type="Edm.Int64" Nullable="false" />
        <Property Name="ResourceID" Type="Edm.String" />
        <Property Name="URL" Type="Edm.String" />
        <Property Name="Status" Type="Edm.String" />
        <Property Name="Error" Type="Edm.String" />
      </EntityType>
      <EntityType Name="UserMetrics">
        <Key>
          <PropertyRef Name="ID" />
        </Key>
        <Property Name="ID" Type="Edm.Int32" Nullable="false" />
        <Property Name="SolveTimestamp" Type="Edm.DateTimeOffset" Nullable="false" />
        <Property Name="SolverUserID" Type="Edm.Int32" Nullable="false" />
        <Property Name="RequestingURL" Type="Edm.String" />
        <Property Name="StorageMBs" Type="Edm.Double" Nullable="false" />
        <Property Name="SolveTimeMillis" Type="Edm.Int64" Nullable="false" />
        <Property Name="SolverResult" Type="Edm.String" />
        <Property Name="SolveTypeCode" Type="Edm.Int32" Nullable="false" />
        <Property Name="ModelName" Type="Edm.String" />
        <Property Name="ModelVersion" Type="Edm.String" />
        <Property Name="ResultID" Type="Edm.String" />
      </EntityType>
      <EntityType Name="Results">
        <Key>
```

```

        <PropertyRef Name="ResourceID" />
    </Key>
    <Property Name="ResourceID" Type="Edm.String" Nullable="false" />
    <Property Name="ModelName" Type="Edm.String" />
    <Property Name="ModelVersion" Type="Edm.String" />
    <Property Name="SolveTimeStamp" Type="Edm.DateTimeOffset" Nullable
="false" />
    <NavigationProperty Name="Evaluations" Type="Collection(RASONOData
.Evaluation)" />
</EntityType>
<EntityType Name="Evaluation">
    <Key>
        <PropertyRef Name="Name" />
    </Key>
    <Property Name="Name" Type="Edm.String" Nullable="false" />
    <NavigationProperty Name="Results" Type="RASONOData.Results" />
</EntityType>
<EntityContainer Name="RASONODataContainer">
    <EntitySet Name="Result" EntityType="RASONOData.Result">
        <NavigationPropertyBinding Path="Data" Target="DataInstance" /
>
    </EntitySet>
    <EntitySet Name="DataInstance" EntityType="RASONOData.DataInstance
" />

    <EntitySet Name="ODataLog" EntityType="RASONOData.ODataMetrics" />
    <EntitySet Name="SolveLog" EntityType="RASONOData.UserMetrics" />
    <EntitySet Name="Results" EntityType="RASONOData.Results">
        <NavigationPropertyBinding Path="Evaluations" Target="Evaluati
ons" />
    </EntitySet>
    <EntitySet Name="Evaluations" EntityType="RASONOData.Evaluation">
        <NavigationPropertyBinding Path="Results" Target="Results" />
    </EntitySet>
</EntityContainer>
</Schema>
</edmx:DataServices>
</edmx:Edmx>

```

REST Endpoint: GET: <https://rason.net/OData/odatalog>

A REST API endpoint that returns a log of odata requests for the requesting user.

Fields returned are:

- ID: Ordinal Table Key
- SolverUserID: Requesting user ID
- RequestStartTime: Time request was submitted
- RequestDuration: Time taken to fulfill request
- ResourceID: Resource ID submitted with endpoint
- URL: URL of ODATA request
- Status: Status of request.
- Error: Null if no error, otherwise the error will be reported here.

URL

<https://rason.net/OData/odatalog>

- **Method:**

GET

- **URL Params**

Required: None

Optional: It's possible to use a query param for any of the returned fields described above, for example:

GET [https://rason.net/odata/odatalog?\\$select=Status,Error](https://rason.net/odata/odatalog?$select=Status,Error)

GET [https://rason.net/odata/odatalog?\\$filter=year\(RequestStartTime\) eq 2020](https://rason.net/odata/odatalog?$filter=year(RequestStartTime) eq 2020)

- **Headers**

Required: Authorization - Example: Authorization: bearer {your RASON token}

Optional: None

- **Data Params:** None

- **Success Response:**

Code: 200 (OK)

Response Example:

```
{
  "@odata.context": "https://rason.net/odata/$metadata#ODataLog",
  "value": [
    {
      "ID": 953,
```

```

        "SolverUserID": 2590,
        "RequestStartTime": "2020-01-23T16:43:41.907Z",
        "RequestDuration": 1512,
        "ResourceID": "2590+Arima+2020-01-17-20-19-21-
674772",
        "URL": "https://rason.net:443/odata/result",
        "Status": "OK",
        "Error": null
    },
    {
        "ID": 954,
        "SolverUserID": 2590,
        "RequestStartTime": "2020-01-23T16:44:04.583Z",
        "RequestDuration": 31,
        "ResourceID": null,
        "URL": "https://rason.net:443/odata/result",
        "Status": "Forbidden",
        "Error": "Invalid or missing value for 'resourceID' h
eader"
    }
}

```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

REST Endpoint: GET: https://rason.net/OData/result

A REST API endpoint that retrieves the results, in OData format, from the model previously submitted to GET/POST `rason.net/api/model/{nameorid}/solve`. Given the structured nature of OData URL routing conventions, the resource ID along with the Bearer token must be submitted within the request header.

- **URL**

`https://rason.net/OData/result`

Method:

GET

- **URL Params**

Required: None

Optional: RASON Decision Services supports ODATA result queries, i.e.

```
https://rason.net/odata/result('customers_result_transformation')
/data?$select=Country,Age&$filter=Age le 40d and
startsWith(Country, 'Fr')&orderby=Age
```

- **Headers**

Required:

Authorization - Example: `Authorization: bearer {your RASON token}`

Resource ID – Example: `resourceID: {yourresourceID}`

Notes:

1. The Resource ID may also be passed as a query parameter, i.e.
`?resourceID=<Your_Resource_ID>`
2. The ResourceID could be a full qualified id of the completed model instance (with '+' characters encoded using '%2B') or it could be the model name. In the latter case RASON will create the OData feed for the last completed instance for that model name

Optional: None

- **Data Params:** None

- **Success Response:**

Code: 200 (OK)

Response Example:

```
{
  "@odata.context": "https://rason.net/odata/$metadata#Result(D
ata())
```

```

/$entity",
"Name": "customers_result_transformation",
"Data": [
  {
    "_ID": 0,
    "_Name": "Record 1",
    "CustomerID": "c1",
    "Country": "Germany",
    "Age": 30.0
  },
  {
    "_ID": 1,
    "_Name": "Record 2",
    "CustomerID": "c2",
    "Country": "France",
    "Age": 40.0
  }
]
}

```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.
Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.

REST Endpoint: GET: <https://rason.net/OData/results>

An ODATA endpoint that collects all RASON models on a user's account with persistent results (which are available for OData querying) and composes the OData feed with basic model info along with its evaluations. Model must have been solved with the asynchronous RASON API endpoint, POST rason.net/api/model/{nameorid}/solve, using the parameter `response-format = WORKFLOW` (the default setting for POST rason.net/api/model/{nameorid}/solve).

This endpoint allows full OData querying functionality – e.g. getting available results for a specific ResourceID or ModelName using

GET [rason.net/odata/results?\\$select=ModelName,Evaluations&\\$filter=ModelName eq 'opt-dm-model'](https://rason.net/odata/results?$select=ModelName,Evaluations&$filter=ModelName eq 'opt-dm-model')

which can be read as: "SELECT ModelName, Evaluations FROM results WHERE ModelName='opt-dm-model'".

It's also possible to use ResourceID in place of ModelName. To do so replace '+' with '%2B'.

- **URLs**

<https://rason.net/OData/results>

[https://rason.net/odata/results?\\$select=evaluations&\\$expand=evaluations](https://rason.net/odata/results?$select=evaluations&$expand=evaluations)

[https://rason.net/odata/results?\\$filter='{modelname}'](https://rason.net/odata/results?$filter='{modelname}')

Method:

GET

- **URL Params**

Required: None

Optional: None

- **Headers**

Required:

Authorization - Example: `Authorization: bearer {your RASON token}`

Optional: None

- **Data Params:** None

- **Success Response:**

Code: 200 (OK)

Request Example: GET <https://rason.net/OData/results>

Response Example:

{

```

"@odata.context": "https://rason.net/odata/$metadata#Results(
Evaluations())",
"value": [
  {
    "ResourceID": "227768+2020-01-27-22-01-16-520127",
    "ModelName": null,
    "ModelVersion": null,
    "SolveTimeStamp": "2020-01-27T22:01:19.9Z",
    "Evaluations": [
      {
        "Name": "rescaling.rescaleddata.transformatio
n"
      },
      {
        "Name": "rescaling.rescalermode
l.statistics"
      }
    ]
  },
  {
    "ResourceID": "227768+2020-01-27-22-03-01-489458",
    "ModelName": null,
    "ModelVersion": null,
    "SolveTimeStamp": "2020-01-27T22:03:04.873Z",
    "Evaluations": [
      {
        "Name": "optmodel.x.dualValue"
      },
      {
        "Name": "optmodel.x.finalValue"
      },
      {
        "Name": "optmodel.x.initialValue"
      },
      {
        "Name": "rescaling.rescaleddata.transformatio
n"
      },
      {
        "Name": "rescaling.rescalermode
l.statistics"
      }
    ]
  }
]

```

Request Example:

[https://rason.net/odata/results?\\$select=evaluations&\\$expand=evaluations](https://rason.net/odata/results?$select=evaluations&$expand=evaluations)

Response Example:

```

{
  "@odata.context": "https://rason.net/odata/$metadata#Results(
Evaluations,Evaluations())",
  "value": [
    {
      "Evaluations": [
        {
          "Name": "addHWMModel.coefficientsInfo"
        }
      ]
    }
  ]
}

```

```

        },
        {
            "Name": "addHWMModel.fittedModelJson"
        },
        {
            "Name": "fitted.cfe"
        },
    ],
},
{
    "Evaluations": [
        {
            "Name": "exponential-1022.exponential-1022Model.coefficientsInfo"
        },
        {
            "Name": "exponential-1022.exponential-1022Model.fittedModelJson"
        },
        {
            "Name": "exponential-1022.fitted.cfe"
        },
    ],
},
{
    "Evaluations": [
        {
            "Name": "obj.finalValue"
        },
        {
            "Name": "x.finalValue"
        },
        {
            "Name": "y.finalValue"
        },
        {
            "Name": "z.finalValue"
        },
    ],
},
]
}

```

Request Example:

ModelName: `https://rason.net/odata/results?$filter=ModelName eq 'productMixExample'`

Resource ID: `https://rason.net/odata/results?$filter=ResourceID eq '227768%2B2020-01-27-22-01-16-520127'`

Response Example:

```

{
    "@odata.context":
    "https://rason.net/odata/$metadata#Results(Evaluations())",
    "value": [

```

```

    {
      "ResourceID": "227768+productMixExample+2020-01-28-22-17-29-933835",
      "ModelName": "productMixExample",
      "ModelVersion": "227768+productMixExample+2020-01-28-22-17-03-362273",
      "SolveTimeStamp": "2020-01-28T22:17:32.213Z",
      "Evaluations": [
        {
          "Name": "d18.finalValue"
        },
        {
          "Name": "d9:f9.finalValue"
        }
      ]
    }
  ]
}

```

Querying Example:

```

https://rason.net/odata/results?$select=ModelName,Evaluations&$filter=ModelName eq 'productMixExample'

```

Response Example:

```

{
  "@odata.context": "https://rason.net/odata/$metadata#Results(ModelName,Evaluations,Evaluations())",
  "value": [
    {
      "ModelName": "productMixExample",
      "Evaluations": [
        {
          "Name": "d18.finalValue"
        },
        {
          "Name": "d9:f9.finalValue"
        }
      ]
    }
  ]
}

```

• Error Response:

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.

Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.
--	--

REST Endpoint: GET: <https://rason.net/OData/solveLog>

A REST API endpoint that returns the log of solve requests for the requesting user. Fields returned are listed below. *Note that a single model is a decision flow with one stage.*

- ResourceID: Resource ID of the model instance. The ResourceID is not updated in the case of scheduled models.
- SeqNumber: If the model is a scheduled model, this field allows multiple runs of the same model with the same ResourceID.
- SolverUserID: Requesting User ID
- Name: Name of submitter.
- Email: Email of submitter
- ModelName: Model name
- ModelVersion: Resource ID of the parent origin/version model, if any.
- SolveTimeStamp: Complete decision flow solve timestamp.
- SolveTime: The complete decision flow solve time in seconds.
- Origin: Submitted user IP address.
- SubmittedBy: Submitted application type.
- StatusCode*: Solver final result code for the decision flow.
- StatusMessage*: Solver final result code text message for the decision flow.
*See the description for GET rason.net/api/model/{nameorid}/result or the Solver Result Messages chapter within the RASON Reference Guide for detailed information on each possible status code and message.
- SolveType: Operation performed. See the list below for all solve types.
- NumStages: Total number of stages in the decision flow.
- NumAttemptedStages: The number of stages attempted, in the decision flow. This value will be equal to NumSucceededStages if solving the entire flow.
- NumSucceededStages: The number of stages that succeeded in the decision flow.
- ManagedModel: If True, this field indicates that the solved model is a managed model.
A managed model is a model stored on the RASON.com server and solved with an asynchronous endpoints. (Solves triggered by Analytic Solver Cloud are not included.)
- CommonSolveType: Endpoint used to solve the model: Unknown, Optimization (\optimize), Simulation (\simulate), Decision (\decision), Datamining (\datamine), Solve (\solve), IISFind (infeasibility report), Diagnosis (\diagnose) or Analysis.
- CommonStatusCode: Status of the decision flow: Success, Stopped, Invalid or Error.
- InstanceType: Displays the model instance type: Unknown, Optimization, Simulation, Calculation, Datamining, Flow.
- ModelKind: Filters on the ModelKind field: ModelKind = Excel (posted Excel model), RASON (RASON models) or LP/MPS models.
- Stages: Log returned for all stages in the decision flow. (A single model is a decision flow with one stage.)
 - SeqNumber: If running a scheduled model
 - Name: Standalone model or stage name.
 - StageID: Used to locate the stage model. If the decision flow is “in-line, this value will be the same as the enclosing flow. If a reusable model, this value will be the ResourceID of the reusable model.
 - Type: The model type: optimization, simulation, decision, or data science.
 - Kind: Filters on the ModelKind field: ModelKind = Excel (posted Excel model), RASON (RASON model) or LPMPS.
 - Attempted: True if a solve was attempted for this stage; false otherwise.
 - Succeeded: True if the solve of this stage was completed; false otherwise.

- Skipped: True if this stage was skipped in the decision flow solve. A skipped stage could be due to a failure or reusing previously available results based on Recency.
- StatusCode: Final status of solve. See the description for GET rason.net/api/model/{nameorid}/result for a list of possible result codes.
- StatusMessage: Final status of solve. See the Solver Result Messages chapter within the RASON Reference Guide for information on each result.
- SolveTime: The time (in seconds) to solve this stage.
- SolveTimestamp: The timestamp of the solve for this stage.
- Optimization: null, if the stage does not contain an optimization model
 - [
 - ResourceID: Resource ID of the model instance.
 - SeqNumber: If a scheduled model, this is the sequence number of the model instance.
 - NumVariables: The number of variables in the optimization stage model.
 - NumIntegerVariables: The number of integer variables in the optimization stage model.
 - NumFunctions: Number of constraints plus the objective in the optimization stage model.
 - Type: Optimization type: LP (linear problem), QP (quadratic problem), QCP (quadratically constrained problem), CVX (convex), NLP (nonlinear problem) or NSP (nonsmooth problem).
- Simulation: null, if the stage does not contain a simulation model
 - [
 - ResourceID: Resource ID of the model instance.
 - SeqNumber: If a scheduled model, this is the sequence number of the model instance.
 - NumUncertainVariables: Total number of uncertain variables in the simulation model.
 - NumUncertain Functions: Total number of uncertain functions in the simulation model.
- Calculation: null, if the stage does not contain a calculation, or decision, model
 - [
 - ResourceID: Resource ID of the model instance.
 - SeqNumber: If a scheduled model, this is the sequence number of the model instance.
 - NumObservations: Number of observations in the decision model.
- DataMining: null, if the stage does not contain a data science or forecasting model
 - [
 - ResourceID: Resource ID of the model instance.
 - SeqNumber: If a scheduled model, this is the sequence number of the model instance.
 - NumTrainingRows: Number of rows in the training data.
 - NumTrainingCols: Number of columns in the training data.
 - Type: Type of data science algorithm: classification, regression, transformation, featureSelection, timeSeries, clustering, textMining, bigdata, affinityAnalysis.

- Algorithm: Data science algorithm.
-]

SolveType Codes

Unknown = 0,
 Optimize = 1,
 Simulate = 2,
 IISFind = 3,
 Diagnose = 4,
 DataMine = 5,
 Calculate = 6,
 Decision = 7,
 Solve = 8,
 QuickOptimize = 101,
 QuickSimulate = 102,
 QuickIISFind = 103,
 QuickDiagnose = 104,
 QuickDataMine = 105,
 QuickCalculate = 106,
 QuickDecision = 107,
 QuickSolve = 108,
 ParameterAnalysis = 200,
 SenParameterAnalysis = 201,
 SimParameterAnalysis = 202,
 DataMineApp = 203,
 SolveForRecourseVariables = 1000,
 SolveWithoutIntegers = 1010

- **URL**

<https://rason.net/OData/solveolog>

- **Method:**

GET

- **URL Params**

Required: None

Optional: It's possible to use a query param for any of the returned fields described above, for example:

GET [https://rason.net/odata/solveolog?\\$select=SolveTimeStamp](https://rason.net/odata/solveolog?$select=SolveTimeStamp)

- **Headers**

Required:

Authorization - Example: Authorization: bearer {your RASON token}

Optional: None

- **Data Params:** None

- **Success Response:**

Code: 200 (OK)

Response Example:

```
"@odata.context":
"https://rason.net/odata/$metadata#SolveLog(Stages(Optimization()),Simulation(),Calculation(),DataMining())",
"value": [
{
  "ResourceID": "2590+2019-11-12-19-24-06-696423",
  "SeqNumber": 0,
  "SolverUserID": 2590,
  "Name": null,
  "Email": null,
  "ModelName": null,
  "ModelVersion": null,
  "SolveTimeStamp": "2019-11-12T19:24:07.353Z",
  "SolveTime": 656,
  "Origin": "98.181.251.23",
  "SubmittedBy": "Unknown",
  "StatusCode": -1,
  "StatusMessage": "Interpret_Err",
  "SolveType": "QuickDecision",
  "NumStages": 0,
  "NumAttemptedStages": 0,
  "NumSucceededStages": 0,
  "ManagedModel": false,
  "CommonSolveType": "Decision",
  "CommonStatusCode": "Error",
  "InstanceType": "Unknown",
  "ModelKind": "Unknown",
  "Stages": []
}, ...
```

- **Error Response:**

Code: 400 BAD REQUEST	Request sent to the RASON Server was incorrect or corrupted.
Code: 401 UNAUTHORIZED	Authorization header is corrupt, malformed, missing or includes an invalid RASON API authorization token.
Code: 403 FORBIDDEN	Request forbidden; user may be trying to access protected data.
Code: 404 NOT FOUND	Resource ID not found.
Code: 405 METHOD NOT ALLOWED	User is not authorized to access the specified URL, i.e. incorrectly using POST on a GET endpoint.

Code: 500 INTERNAL SERVER ERROR	Something has gone wrong on the RASON Server, contact technical support.
--	--

OData Service for RASON

Decision flows

Introduction

RASON Decision Services allows users to define multiple "stages" in a **single** RASON file, where a stage can perform an SQL operation, apply a data transformation, train a machine learning model, apply it to score new data, run a simulation, solve a mathematical optimization problem or evaluate one or more linked decision tables. Results are passed between stages in a rich, standard "Indexed Data Frame" form.

A dataframe, in XLMiner SDK; the workhorse of the RASON Server, is a collection of data organized into named columns of equal length and homogeneous type. RASON uses DataFrames to deliver input data to an algorithm and to deliver the results of the algorithm back to the user. DataFrames hold heterogeneous data across columns (variables): numeric, categorical, or textual. When solving a decision flow containing optimization or simulation models, the columns that are indexed over the same dimensions and that belong to the same entity are reported in a single dataframe with multiple columns rather than multiple dataframes, i.e. final, dual, initial, etc for optimization results and statistics for uncertain variables or functions in simulation models. RASON can still bind to the individual results such as `optModel.x.finalValue` but will also consider the possibility of the last segment being a dataframe column rather than a separate dataframe. As a result, JSON responses are concise which greatly simplifies OData representation and querying.

Example

Find the example decision flow below with two disconnected SQL Transformation stages. This example uses two tables, `customers_dt` and `orders` and returns them unmodified from `[customers]/[orders]`. The input file `customers.txt` is included in the RASON Examples Download file as explained within the section, [Uploading Data from an External Data File](#) within the chapter, Using the RASON Services Web IDE.

```
{
  "workflow": "sqlWorkflow",
  "customers": {
    "comment": "transformation: SQL",
    "datasources": {
      "srcCustomers": {
        "type": "csv",
        "connection": "customers_dt.txt"
      }
    }
  },
  "datasets": {
    "customers": {
      "binding": "srcCustomers",
      "colNames": [
```

```

        "CustomerID", "Country", "Age"
    ],
    "indexCols": ["CustomerID"]
}
},
"transformer": {
    "mySQLTransformer": {
        "type": "transformation",
        "algorithm": "sql",
        "parameters": {
            "query": "SELECT * FROM [customers];"
        }
    }
},
"actions": {
    "result": {
        "action": "transform",
        "evaluations": [
            "transformation"
        ]
    }
}
},
"orders": {
    "comment": "transformation: SQL",
    "datasources": {
        "srcOrders": {
            "type": "csv",
            "connection": "orders.txt"
        }
    }
},
"datasets": {
    "orders": {
        "binding": "srcOrders",
        "colNames": [
            "OrderID", "CustomerID", "Price", "Quantity"
        ],
        "indexCols": ["OrderID", "CustomerID"]
    }
},
"transformer": {
    "mySQLTransformer": {
        "type": "transformation",
        "algorithm": "sql",
        "parameters": {
            "query": "SELECT * FROM [orders];"
        }
    }
},
"actions": {
    "result": {
        "action": "transform",
        "evaluations": [
            "transformation"
        ]
    }
}
}

```

```
}
}
```

Solving the Decision Flow

To solve the decision flow, log on to www.RASON.com, click the Editor tab and paste the model into the Editor window. To initiate the solving process, use the following API endpoints to POST the model to the RASON Server, solve the decision flow and automatically create an OData endpoint, check the status of the solve and finally obtain the results. For more information on how to solve a decision flow/model using the RASON Services Web IDE, see the chapter Using the RASON Services Web IDE.

- POST `rason.net/api/model` to post the model to the RASON Server.
- POST `rason.net/api/model/{nameorid}/solve` to solve the decision flow and create and automatically create an OData endpoint. You'll see similar output. For more information on each labeled field, see the endpoint description for POST `rason.net/api/model/{nameorid}/solvetype` or POST `rason.net/api/model`.

```
{
  "ModelId": "2590+2020-01-28-20-18-46-971091",
  "ModelName": "",
  "ModelDescr": "",
  "ModelFiles": ["customers_dt.txt", "orders.txt"],
  "RuntimeToken": "",
  "ModelType": 2,
  "ModelTypeName": "Instance",
  "IsChampion": false,
  "ParentModelId": "2590+2020-01-28-20-18-37-110107",
  "QueryString": "",
  "ModelContainer": null
}
```

- GET `rason.net/api/model/{nameorid}/status` to check the status of the model
- GET `rason.net/api/model/{nameorid}/results` to obtain the results.

Using an API Development Platform or your own application, you can call the following REST API Endpoints to gain more information on the stages of the decision flow.

- Use GET `rason.net/api/model/{nameorid}/stages` to retrieve stage information from the decision flow. For more information, see the GET `rason.net/api/model/{nameorid}/stages` endpoint description above.

Example: GET `https://rason.net/api/model/2590+2020-01-28-20-18-46-971091/stages`

```
[
  "customers",
  "orders"
]
```

- Use GET `rason.net/api/model/{nameorid}/stages?type=ordered` to retrieve the stage information in the optimal order of execution. For more information, see the GET `rason.net/api/model/{nameorid}/stages` endpoint description above.

Example: GET `https://rason.net/api/model/2590+2020-01-28-20-18-46-971091/stages?type=ordered`

```
[
  "orders",
  "customers"
]
```

- Use GET rason.net/api/model/{nameorid}/stages?type=terminal to retrieve the terminal nodes in the decision flow. For more information, see the GET rason.net/api/model/{nameorid}/stages endpoint description above.

Example: GET <https://rason.net/api/model/2590+2020-01-28-20-18-46-971091/stages?type=terminal>

```
[
  "customers",
  "orders"
]
```

- Use GET rason.net/api/model/{nameorid}/stages/{stage-name} to retrieve information on a specified stage.

Example: GET <https://rason.net/api/model/2590+2020-01-28-20-18-46-971091/stages/orders>

```
[
  {
    "name": "orders",
    "type": "datamining",
    "pipeline": [
      "orders"
    ],
    "allResults": [
      "result.transformation"
    ],
    "resultsForBindings": [],
    "dataSources": [
      {
        "name": "srcorders",
        "type": "csv",
        "connection": "orders.txt",
        "selection": "",
        "content": "",
        "direction": "import",
        "isUsed": true,
        "isStageBinding": false,
        "isFittedDMMModel": false
      }
    ],
    "isTerminal": true
  }
]
```

- Use GET rason.net/api/model/{nameorid}/solve to attach the input files customers_.dt.txt and orders.txt and solve the decision flow.

```
{
  "status": {
    "id": "227768+2020-01-28-20-29-45-756097",
    "code": 0,
    "codeText": "Success"
  },
}
```



```

"results": {
  "customers_result_transformation": [],
  "orders_result_transformation": []
},
"orders": {
  "status": {
    "code": 0,
    "codeText": "Success"
  },
  "result": {
    "transformation": {
      "objectType": "dataFrame",
      "name": "DataFrameVector_Transformed",
      "order": "col",
      "rowNames": [
        "Record 1",
        "Record 2",
        "Record 3",
        "Record 4"
      ],
      "colNames": [
        "OrderID",
        "CustomerID",
        "Price",
        "Quantity"
      ],
      "colTypes": [
        "wstring",
        "wstring",
        "double",
        "double"
      ],
      "indexCols": null,
      "data": [
        [
          "o1",
          "o1",
          "o2",
          "o2"
        ],
        [
          "c1",
          "c2",
          "c1",
          "c2"
        ],
        [
          1,
          3,
          5,
          7
        ],
        [
          10,
          20,
          15,
          40
        ]
      ]
    }
  }
}

```

```

    ]
  ]
}
},
"customers": {
  "status": {
    "code": 0,
    "codeText": "Success"
  },
  "result": {
    "transformation": {
      "objectType": "dataFrame",
      "name": "DataFrameVector_Transformed",
      "order": "col",
      "rowNames": [
        "Record 1",
        "Record 2"
      ],
      "colNames": [
        "CustomerID",
        "Country",
        "Age"
      ],
      "colTypes": [
        "wstring",
        "wstring",
        "double"
      ],
      "indexCols": null,
      "data": [
        [
          "c1",
          "c2"
        ],
        [
          "Germany",
          "France"
        ],
        [
          30,
          40
        ]
      ]
    }
  }
}
}

```

Example Results

Two Dataframes, `orders.result.transformation` and `customers.result.transformation`, are the exact original tables from the two input files (`customers.txt` and `orders.txt`) and are stored in the SQLite DB file. Both Dataframes are represented in the form of JSON-serialized Entity Data Models described above and are navigable and querable.

The main OData endpoint to retrieve all results for a given resource ID is GET <https://rason.net/odata/result>.

Given the structured nature of OData URL routing conventions, the "resource ID" is not included within the URL request. Rather, the "resourceID" parameter must be submitted within the request header or provided as a query parameter. (For more information, see the GET <https://rason.net/odata/result> endpoint in the previous chapter.) Proper authorization, such as a Bearer token must also be submitted with the request.

OData is a flexible tool and offers many operators. Let's use a few to drill into the data.

- To retrieve a particular result or table by Name, use the OData endpoint:

GET <https://rason.net/odata/result/{tablename}> or

GET [https://rason.net/odata/result\('table'\)](https://rason.net/odata/result('table'))

Example: GET https://rason.net/odata/result/customers_result_transformation

```
{
  "@odata.context":
  "https://rason.net/odata/$metadata#Result(Data())/Entity",
  "Name": "customers_result_transformation",
  "Data": [
    {
      "_ID": 0,
      "_Name": "Record 1",
      "CustomerID": "c1",
      "Country": "Germany",
      "Age": 30.0
    },
    {
      "_ID": 1,
      "_Name": "Record 2",
      "CustomerID": "c2",
      "Country": "France",
      "Age": 40.0
    }
  ]
}
```

Note: Dots (.) in table names must be replaced with underscores (_).

- To retrieve the name of a given table, use the OData endpoint:

GET <https://rason.net/odata/result/{tablename}/name>

Example: GET https://rason.net/odata/result/customers_result_transformation/name

```
{
  "@odata.context":
  "https://rason.net/odata/$metadata#Result('customers_result_transformation')/Name",
  "value": "customers_result_transformation"
}
```

- To retrieve data from a given table, use the OData endpoint:

GET <https://rason.net/odata/result/{tablename}/data/0> or

GET [https://rason.net/odata/result\('table'\)/data\(0\)](https://rason.net/odata/result('table')/data(0))

Example: GET https://rason.net/odata/result/customers_result_transformation/data/0

```
{
  "@odata.context":
  "https://rason.net/odata/$metadata#DataInstance",
  "value": [
    {
      "_ID": 0,
      "_Name": "Record 1",
      "CustomerID": "c1",
      "Country": "Germany",
      "Age": 30.0
    },
    {
      "_ID": 1,
      "_Name": "Record 2",
      "CustomerID": "c2",
      "Country": "France",
      "Age": 40.0
    }
  ]
}
```

- To retrieve a particular instance or row from a given table using the ID, use the OData endpoint:

GET https://rason.net/odata/result/{tablename}/data/{rowID}

Example: GET https://rason.net/odata/result/customers_result_transformation/data/1

```
{
  "@odata.context":
  "https://rason.net/odata/$metadata#DataInstance/$entity",
  "_ID": 1,
  "_Name": "Record 2",
  "CustomerID": "c2",
  "Country": "France",
  "Age": 40.0
}
```

Querying

Most of the core querying capabilities of OData 4 are supported by ASP.NET OData. Our EDM structures enable all querying functionality. Below are a few examples.

Example: GET: https://rason.net/odata/result?{queryparams}

Example queries:

- Query on customers table to select Country and Age using the built-in query function "starts with" and order results by descending Age.

Example: GET

https://rason.net/odata/result('customers_result_transformation')
/data?select=Country, Age&\$filter=Age le 40d and startsWith(Country,
'Fr')&\$ordersby=Age

RASON Services supports integer, double and string types. For the fields of the double types, you must use "d" postfix in filters, arithmetic operations, etc. e.g. &\$filter=Age le 40d above where "d" = double.

```
{
  "@odata.context":
  "https://rason.net/odata/$metadata#DataInstance(Country, Age)",
  "value": [
    {
      "Country": "France",
      "Age": 40.0
    }
  ]
}
```

- Expand the data and then query on the customers table.

Notice that in the previous bullet, we initially navigated to the Data entity, practically excluding the Name property from the resulting set. Sometimes, however, you must operate on the complete entity. In this case, use the syntax \$expand – first expand the Data and then provide any query within the parentheses.

Example: GET:

[https://rason.net/odata/result\('customers_result_transformation'\)?\\$expand=Data\(\\$select=Country\)](https://rason.net/odata/result('customers_result_transformation')?$expand=Data($select=Country))

```
{
  "@odata.context": "https://rason.net/odata/$metadata#Result(Data(Country))/$entity",
  "Name": "customers_result_transformation",
  "Data": [
    {
      "Country": "Germany"
    },
    {
      "Country": "France"
    }
  ]
}
```

Notice that the entire entity has been retained with Name/Data entities, while querying the Data entity.

Advanced OData Settings

There are many settings available for fine turning the default behavior of OData. For more information, see the following link.

<https://docs.microsoft.com/en-us/dynamics365/fin-ops-core/dev-itpro/data-entities/odata>

Controlling the Amount of Information in Response

In all responses shown, the additional service field "@odata.context" is listed at the top. By default, the parameter controlling the presence/verbosity of metadata is set to minimal. To change to either "none" or "full" use: the Accept header.

Accept: application/json;odata.metadata=none

Paging

In order to avoid sending massive payloads with large amounts of data over the network all at once, the results can be paged. This can be done in two ways.

1. Client-driven paging:

- a. Clients can use \$Stop and \$Skip queries (and their combinations) to obtain partial results
- b. Clients can use odata.maxpagesize parameter in the Prefer header to control the paging, i.e.
`Prefer: odata.maxpagesize=3`

2. Server-driven paging:

- a. rason.net includes limit of PageSize = 1000.

For 1.b and 2.a, a partial response will be received, along with the nextLink-URL for the next page.

```
{
  "@odata.context":
  "https://rason.net/odata/$metadata#Result(Data())",
  "value": [
    {
      "Name":
      "customers_result_transformation",
      "Data": [
        {
          "_ID": 0,
          "_Name": "Record 1",
          "CustomerID": "c1",
          "Country": "Germany",
          "Age": 30.0
        },
        {
          "_ID": 1,
          "_Name": "Record 2",
          "CustomerID": "c2",
          "Country": "France",
          "Age": 40.0
        }
      ]
    }
  ],
  "@odata.nextLink":
  "https://rason.net/odata/result?$skip=1"
```

Using OData in Power BI

This simple example illustrates how one could use results saved in OData format to create a chart or graph in Power BI.

1. Open Power BI
2. On the Power BI ribbon, click Get Data – Blank Query; the Power Query Editor will appear.

3. Within the Power Query Editor, click View – Advanced Editor

Since Power BI does not currently provide a way to specify headers, M code must be used. Copy and paste the following M code into the Advanced Editor dialog.

```
let
    url="https://rason.net/odata/result",
    modelID="<your resource ID here>",
    token="Bearer <your token here>",
    Source = OData.Feed(url,null,[Headers=[Authorization=token,
    modelID=modelID], MoreColumns=true]),
in
    Source
```

URL: Any valid RASON OData endpoint may be substituted for the current URL.

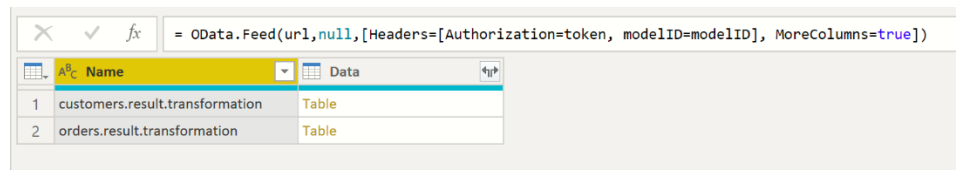
modelID: You can find the resource ID using the RASON Rest API endpoint: GET rason.net/pai/model/id.

token: You can find your token by logging on to RASON.com and clicking MyAccount. (See API Authorization on the MyAccount page.)



Note that "MoreColumns=True" is required for "Source" in order to see the related open type entities. By default, this setting is false. Setting this parameter to True is equivalent to checking "Include Open Type Columns" checkbox within the OData Feed dialog.

4. Click Done. Power BI will connect to the RASON Server's OData feed and pull the requested results.



- To view each table, click "Table" under the Data column.

Table 1 customers.result.transformation

<div> <div>✕</div> <div>✓</div> <div>fx</div> </div> <div>= Source{[Name="customers.result.transformation"]}[Data]</div>	
<div> <div>1²3</div> <div>_ID</div> </div> <div>More Columns</div>	
1	0 Record
2	1 Record

Table 2 orders.result.transformation

<div> <div>✕</div> <div>✓</div> <div>fx</div> </div> <div>= Source{[Name="orders.result.transformation"]}[Data]</div>	
<div> <div>1²3</div> <div>_ID</div> </div> <div>More Columns</div>	
1	0 Record
2	1 Record
3	2 Record
4	3 Record

- Click "Record" under More Columns (in either table), to see that instance of data.

<div> <div>✕</div> <div>✓</div> <div>fx</div> </div> <div>= #"orders result transformation"{[_ID=1]}[More Columns]</div>	
_Name	Record 2
OrderID	o1
CustomerID	c2
Price	3
Quantity	20

Click the Expand button to the right of More Columns to expand the fields.

<div> <div>✕</div> <div>✓</div> <div>fx</div> </div> <div>= Source{[Name="orders.result.transf</div>	
<div> <div>1²3</div> <div>_ID</div> </div> <div>More Columns</div>	<div> <div>⏏</div> </div>

Uncheck "Use original column name as prefix".

1²3

_ID

More Columns

Search Columns to Expand

✓

(Select All Columns)

✓

_Name

✓

OrderID

✓

CustomerID

✓

Price

✓

Quantity

☐ Use original column name as prefix

⚠

List may be incomplete.

Load more

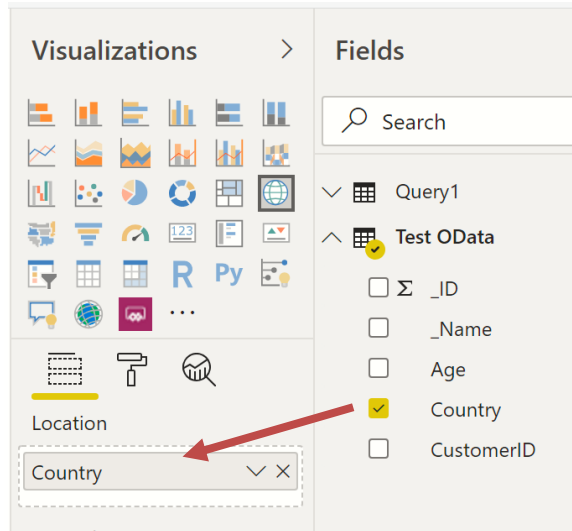
OK

Cancel

Then click OK.

Table.ExpandRecordColumn(#"orders result transformation", "More Columns", {"_Name", "OrderID", "CustomerID", "Price", "Quantity"}, ...)						
_ID	_Name	OrderID	CustomerID	Price	Quantity	
1	0 Record 1	o1	c1		1	
2	1 Record 2	o1	c2		3	
3	2 Record 3	c1			5	
4	3 Record 4	o2	c2		7	

- Click "Close and Apply" in the Power Query Editor. The data will now be available on the Power BI dashboard. For example, click Map under Visualizations and then drag "Country" under Fields on the right to "Location" on the left.



To produce the simple map below.



Creating Your Own App

Introduction

When using the RASON REST API, you have the ability to not only create, design and solve your optimization, simulation, simulation optimization, stochastic optimization, data science and decision table models on the Editor page on RASON.com, but you also have the ability to embed your model into your own application and solve it on the Web by calling our RASON Server.

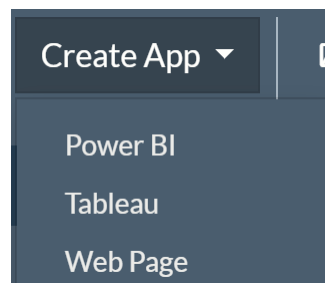
When you click Create App - Web Page on the Editor ribbon on RASON.com, your existing RASON model will be automatically converted to a mobile application written in JavaScript. Web App Developers will be able to immediately find how exceptionally easy it is to embed RASON models as JSON and solve them using Frontline's RASON server, which exposes a simple REST API that's scalable to handle very large, compute-intensive analytic models. Months of work, that would have previously been required, have been reduced to a single command button click!

Clicking Create App – Power BI, gives you the ability to turn your RASON model into a **Microsoft Power BI Custom Visual**. What you get isn't just a chart – it's your *full RASON model*, ready to accept Power BI data, **run on demand** on the web, and display visual results in Power BI! You simply need to drag and drop appropriate Power BI datasets into the “well” of inputs to match your model parameters. This is possible because Analytic Solver “wraps” a JavaScript-based Custom Visual around the RASON model. For more information on this feature, see “Creating Power BI Custom Visuals” below.

Clicking Create App - Tableau, allows you to turn your RASON model into a **Tableau Dashboard Extension**. In Tableau, you'll see the newly-created file under **Extensions** on the left side of the dashboard, where you can drag it onto your dashboard. You'll be prompted to match the parameters your model needs, with data in Tableau. Much like with Power BI, what you get isn't just a chart – it's your *full optimization or simulation model*, ready to accept Tableau data, **run on demand** (using our **RASON** server), and display visual results in Tableau! Note: This feature works (only) with Tableau version 2018.2 or later.

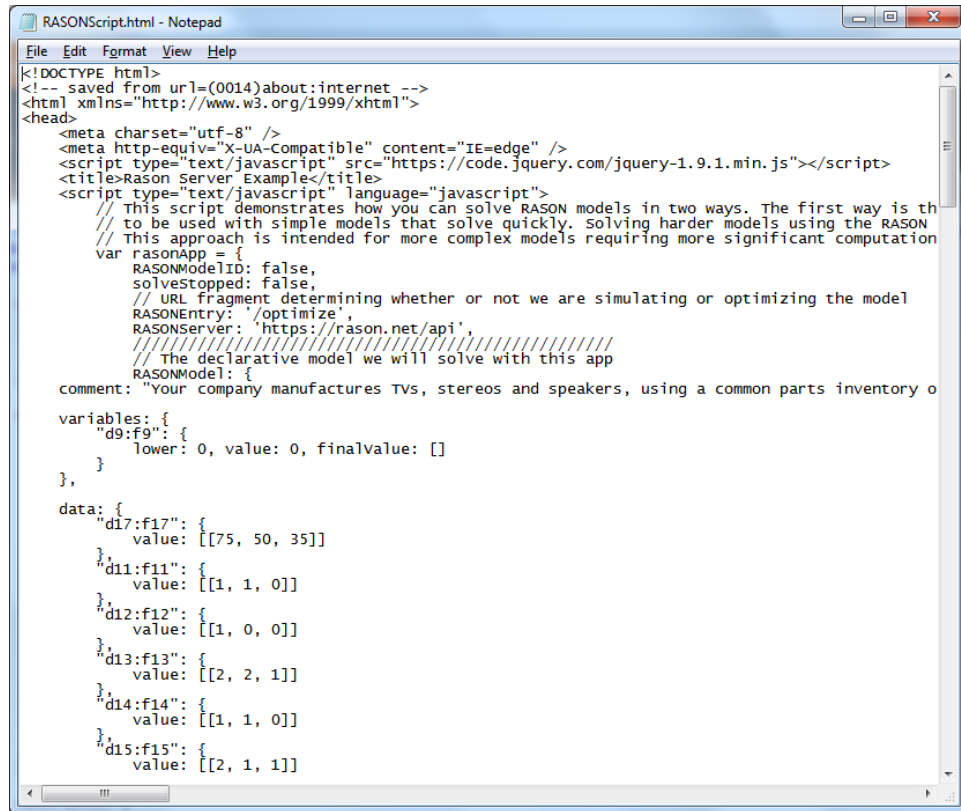
Creating Your Own Web Application

The quickest and easiest way to create your own optimization or simulation application is to use *Create App* on the RASON model editor ribbon. *Note: This feature is not supported when using Analytic Solver Data Science.*



When **Create App – Web Page** is clicked, a demo JavaScript application is created that solves the model within the RASON model editor. This file, RASONScript.html is downloaded locally. A snippet of the generated code is shown inside the Notepad application in the screenshot below.

A closer inspection of RASONScript.html will reveal the RASON code for the Product Mix example model, along with a call to the "Quick Solve" method to illustrate how to solve a simple model quickly, and also to the RASON REST API to demonstrate how to solve harder, more time consuming models. This sample application, written in HTML, Css and JavaScript, is just an example of how an application can be designed. For more information on the RASON REST API endpoints, see below.



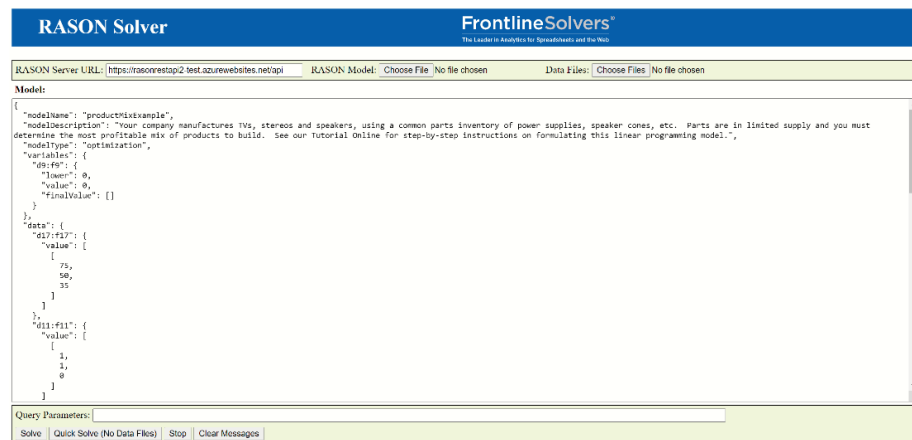
```

RASONScript.html - Notepad
File Edit Format View Help
<!DOCTYPE html>
<!-- saved from url=(0014)about:internet -->
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta charset="utf-8" />
<meta http-equiv="X-UA-Compatible" content="IE=edge" />
<script type="text/javascript" src="https://code.jquery.com/jquery-1.9.1.min.js"></script>
<title>Rason Server Example</title>
<script type="text/javascript" language="javascript">
// This script demonstrates how you can solve RASON models in two ways. The first way is th
// to be used with simple models that solve quickly. Solving harder models using the RASON
// This approach is intended for more complex models requiring more significant computation
var rasonApp = {
  RASONModelID: false,
  solveStopped: false,
  // URL fragment determining whether or not we are simulating or optimizing the model
  RASONentry: '/optimize',
  RASONServer: 'https://rason.net/api',
  // The declarative model we will solve with this app
  RASONModel: {
    comment: "Your company manufactures TVs, stereos and speakers, using a common parts inventory o
    variables: {
      "d9:f9": {
        lower: 0, value: 0, finalValue: []
      },
    },
    data: {
      "d17:f17": {
        value: [[75, 50, 35]]
      },
      "d11:f11": {
        value: [[1, 1, 0]]
      },
      "d12:f12": {
        value: [[1, 0, 0]]
      },
      "d13:f13": {
        value: [[2, 2, 1]]
      },
      "d14:f14": {
        value: [[1, 1, 0]]
      },
      "d15:f15": {
        value: [[2, 1, 1]]
      }
    }
  }
}

```

To run this application, open RASONScript.html in a browser of your choice.

The RASON example code is displayed, along with command buttons below that allow you to call the Quick Solve method (Quick Solve) along with a call to the RASON REST API (Solve).



Click **Quick Solve** to solve the model and retrieve the result. The Web application created by Create App – Web Page solves models of all types: decision flows, optimization, simulation, decision table and data science.

Rason Server Example

```
{
  "status": {
    "code": 0,
    "codeText": "Solver found a solution. All constraints and optimality
conditions are satisfied."
  },
  "variables": {
    "d9:f9": {
      "finalValue": [
        [
          200,
          200,
          0
        ]
      ]
    }
  },
  "objective": {
    "d18": {
      "finalValue": 25000
    }
  }
}
```

Refer to the previous chapter, Using the REST API, to find an in depth description of each RASON REST API endpoint.

Uploading your RASON Model to Power BI

RASON Decision Services allows you to turn your RASON-based optimization or simulation model into a **Microsoft Power BI Custom Visual**. (Neither data science nor decision table models are supported at this time.) Where others must learn JavaScript (or TypeScript) programming and a whole set of Web development tools to even begin to create a Custom Visual in Power BI, after reading this section, RASON users will be able to create one right away.

To summarize, users simply select rows or columns of data to serve as changeable parameters, then choose **Create App – Power BI** (on the Editor page ribbon), and save the file created by RASON. Afterwards, users click the *Import a Custom Visual* icon in Power BI and select the file just saved. What is produced isn't just a chart – it's a *full optimization or simulation model*, ready to accept Power BI data, **run on demand** on the web and display visual results in Power BI. Users simply need to drag and drop appropriate datasets into the “well” of inputs in Power BI to match the model parameters.

The RASON Server translates the Excel model into **RASON®** (RESTful Analytic Solver Object Notation, embedded in JSON), then “wraps” a JavaScript-based Custom Visual around the RASON model. See the previous chapter “Creating Your Own Application” for more information on RASON.

Installing Power BI

Power BI is Microsoft's desktop or cloud-based interactive data visualization business intelligence tool. In past versions of Analytic Solver, users have only been able to upload their model parameters (model data) to Power BI after solving an optimization or simulation model. RASON users can turn their RASON based optimization or simulation model into a Microsoft Power BI Custom Visual; *that*

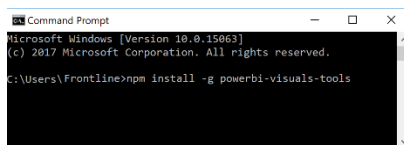
is the full optimization or simulation model, ready to accept Power BI data, run on demand on the web and display visual results in Power BI.

Note: Click this [link](#) to download Power BI for desktop.

In order to use this new feature, you must first open desktop Power BI or the free cloud based version of Power BI. (Power BI is part of the office 365 suite.) For more information on this business tool, see the following website: <https://powerbi.microsoft.com/en-us/documentation/powerbi-custom-visuals-getting-started-with-developer-tools/>

It is important to note, that in order to create a custom visual in Power BI, you will first need to install Power BI's Developer Tools which is comprised of NodeJS and the Power BI tools. Follow the steps below to install both required items.

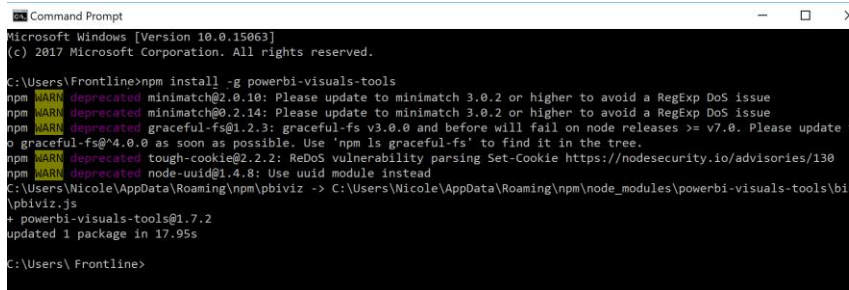
1. Click [here](#) to download NodeJS.
2. Click the Download LTS button to download NodeJS for 64-bit Windows.
3. Once the NodeJS installer is downloaded, run the installer and follow the directions on the installer dialogs to install NodeJS onto your machine.
4. Install the command line tools by opening a command prompt and typing: “npm install -g powerbi-visuals-tools” as shown in the screenshot below.



```
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\Frontline>npm install -g powerbi-visuals-tools
```

After clicking Enter, you'll see the following results in the command prompt window.



```
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\Frontline>npm install -g powerbi-visuals-tools
npm WARN deprecated minimatch@2.0.10: Please update to minimatch 3.0.2 or higher to avoid a RegExp DoS issue
npm WARN deprecated minimatch@0.2.14: Please update to minimatch 3.0.2 or higher to avoid a RegExp DoS issue
npm WARN deprecated graceful-fs@1.2.3: graceful-fs v3.0.0 and before will fail on node releases >= v7.0. Please update to
  graceful-fs@4.0.0 as soon as possible. Use 'npm ls graceful-fs' to find it in the tree.
npm WARN deprecated tough-cookie@2.2.2: ReDoS vulnerability parsing Set-Cookie https://nodesecurity.io/advisories/130
npm WARN deprecated node-uuid@1.4.8: Use uuid module instead
C:\Users\Nicole\AppData\Roaming\npm\pbiviz -> C:\Users\Nicole\AppData\Roaming\npm\node_modules\powerbi-visuals-tools\bin
\pbiviz.js
+ powerbi-visuals-tools@1.7.2
updated 1 package in 17.95s

C:\Users\Frontline>
```

Creating a Custom Visual from a RASON Model

Now that the Developer Tools are installed, we can move on to a small RASON example. In this section, we will create a custom visual using the Product Mix example model.

In this section, we will create a custom visual for Power BI using the Product Mix example model. Log on to www.RASON.com and then click on the Editor tab. Click the RASON Examples folder (on the top right), then click Optimization With Data Binding - ProductMixCsv4. Recall this example model determines the optimal mix of products that a company should produce in order to maximize profits.

In order to create a Custom Visual for Power BI, your RASON model must be formulated using an index set. The example code below creates two ordered sets, parts and prods. The parts set contains five items (in order as entered): chas, tube, cone, psup and elec while the prods set contains 3 items: tv, stereo and speakers. An indexSet is always defined as a JSON object{ }. For more information on index sets, see the RASON Reference Guide.

The example code below uses the dataSources section to create three datasources, parts_data, invent_data and profit_data; using three CSV files. Any supported file type may be used.

As discussed in previous sections, a `dataSource` allows data to be passed to the RASON Server using an external data file in a supported file type. By using an external data file, we can submit updated data to the RASON Server without having to edit our existing RASON model. For example, imagine that Purchasing was able to obtain a discount on Speakers and our inventory increased from 800 to 1,000. If we were not using an external data file, we would have to edit the RASON model directly to reflect this change.

```
{
  "modelName": "ProductMixCSV4",
  "modelType": "optimization",
  "comment": "Example of using CSV table binding to read params and save results",
  "indexSets": [{
    "name": "parts",
    "value": ["chas", "tube", "cone", "psup", "elec"]
  }, {
    "name": "prods",
    "value": ["tv", "stereo", "speaker"]
  }],
  "datasources": {
    "parts_data": {
      "type": "csv",
      "connection": "ProductMixParts.txt",
      "indexCols": ["parts", "prods"],
      "valueCols": ["qty"]
    },
    "invent_data": {
      "type": "csv",
      "connection": "ProductMixInventory.txt",
      "indexCols": ["parts"],
      "valueCols": ["inventory"]
    },
    "profit_data": {
      "type": "csv",
      "connection": "ProductMixProfits.txt",
      "indexCols": ["prods"],
      "valueCols": ["profits"]
    }
  },
  "data": [{
    "name": "parts2",
    "binding": "parts_data",
    "valueCol": "qty"
  }, {
    "name": "invent",
    "binding": "invent_data",
    "valueCol": "inventory"
  }, {
    "name": "profit",
    "binding": "profit_data",
    "valueCol": "profits"
  }],
  "formulas": [{
    "name": "piv_parts",
    "formula": "PIVOT(parts2, { 'prods' }, { 'parts' })"
  }],
  "variables": {
    "name": "x",
    "dimensions": ["prods"],
    "value": 0,
    "lower": 0,
    "finalValue": [],
    "indexValue": []
  },
  "constraints": [{
    "name": "c",
    "dimensions": ["parts"],
    "formula": "MMULT(piv_parts, x) - invent[]",
    "upper": 0,
    "finalValue": []
  }],
}
```

```

    "objective": {
      "name": "total",
      "formula": "sumproduct(x, profit[])",
      "type": "maximize"
    }
  }
}

```

Data Sources: ProductMixInventory.txt, ProductMixParts.txt and ProductMixProfits.txt

The image shows three screenshots of text files. The first, ProductMixInventory.txt, has columns 'parts' and 'inventory' with 6 rows of data. The second, ProductMixParts.txt, has columns 'parts', 'prods', and 'qty' with 12 rows of data. The third, ProductMixProfits.txt, has columns 'prods' and 'profits' with 4 rows of data.

parts	inventory
chas	450
tube	250
cone	800
psup	450
elec	600

parts	prods	qty
chas	tv	1
chas	stereo	1
tube	tv	1
cone	tv	2
cone	stereo	2
cone	speaker	1
psup	tv	1
psup	stereo	1
elec	tv	2
elec	stereo	1
elec	speaker	1

prods	profits
tv	75
stereo	50
speaker	35

DataSources

The first data source, parts_data, contains two index columns, parts and products, i.e., a LCD TV requires 2 Electronic components. The next two data sources, invent_data and profit_data, include one indexCol and one valueCol. These two properties create a dataframe. (While properties indexCols and valueCols create a RASON table.) The indexCol is the column containing the indices or the labels and the valueCol is the column containing the values for the indices.

```

"datasources": {
  "parts_data": {
    "type": "csv",
    "connection": "ProductMixParts.txt",
    "indexCols": [ "parts", "prods" ],
    "valueCols": [ "qty" ],
    "direction": "import"
  },
  "invent_data": {
    "type": "csv",
    "connection": "ProductMixInventory.txt",
    "indexCols": [ "parts" ],
    "valueCols": [ "inventory" ],
    "direction": "import"
  },
  "profit_data": {
    "type": "csv",
    "connection": "ProductMixProfits.txt",
    "indexCols": [ "prods" ],
    "valueCols": [ "profits" ],
    "direction": "import"
  }
}

```

Data

In the data section, "profit" is bound to the column "profits" within the data source, "profit_data", "parts2" is bound to the "qty" column within the "parts_data" datasource and "invent" is bound to the "inventory" column within the "invent_data" datasource.

```

"data": [
  {
    "name": "parts2",

```



```

    "binding": "parts_data",
    "valueCol": "qty"
  },

  {
    "name": "invent",
    "binding": "invent_data",
    "valueCol": "inventory"
  },

  {
    "name": "profit",
    "binding": "profit_data",
    "valueCol": "profits"
  }
],

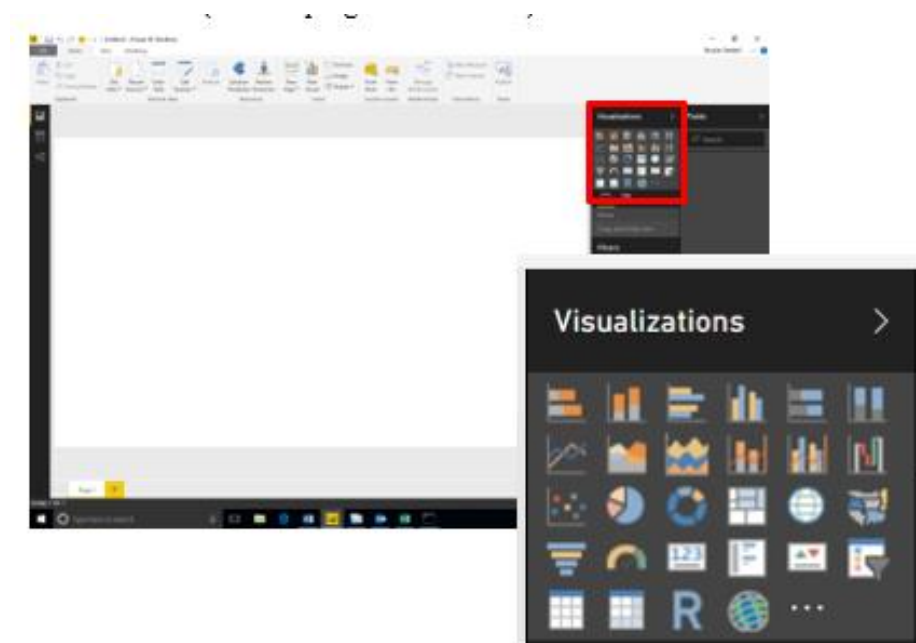
```

The three variables are specified within variables, the constraints are created within constraints (using a Pivot Table) and the objective is computed and maximized within objective. For more information on using a pivot table within RASON, see the *Using Tables* section that occurs in the chapter, **Using Array Formulas, For Loops and Tables in RASON**.

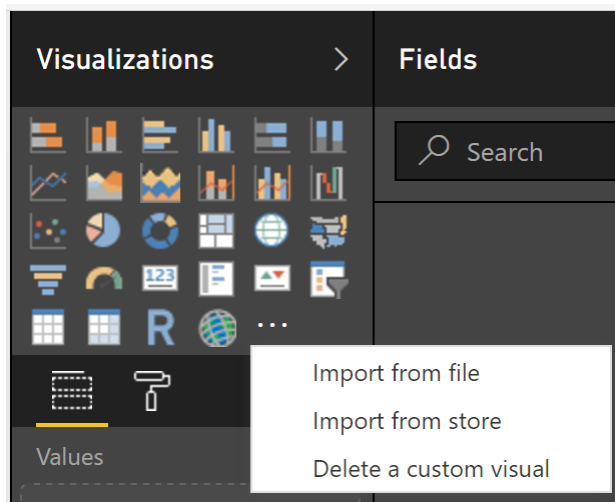
Click Create App – Power BI to create the Power BI custom visual. Save the custom visual in a location of your choice.



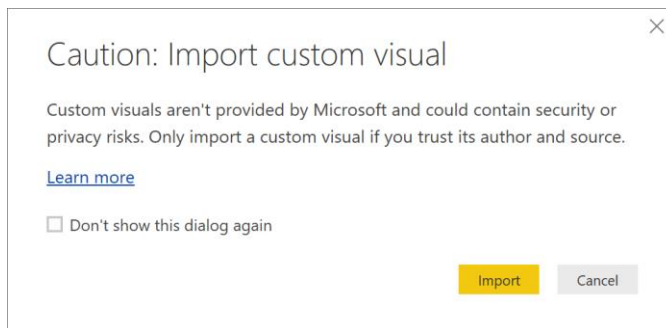
Open either desktop or cloud-based Power BI. The screenshot below depicts the opening screen of desktop Power BI. Click the icon containing three horizontal dots that appears at the bottom of Visualizations (in the top right-hand corner).



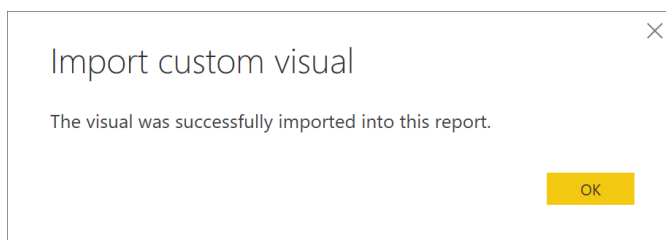
Then select *Import from file* from the menu.



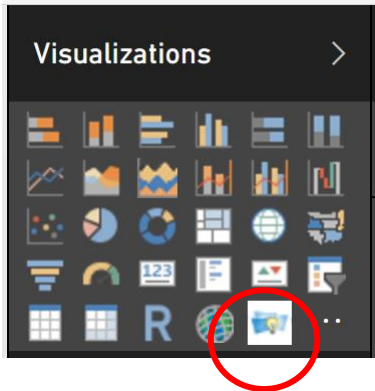
Click **Import** on the *Import custom visual* dialog that appears. (You can select *Don't show this dialog again* if you'd rather not see this dialog each time you import a custom visual.)



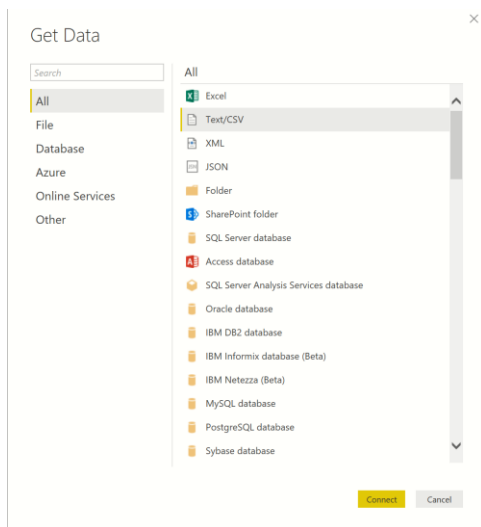
Navigate to the location of the saved Power BI custom visual, RASONAppPowerBI.pbiviz file and click Open. If the import was successful, you will see a message indicating as such. Click OK to clear this dialog.



A new icon, bearing the Frontline Solvers logo is added under Visualizations.



Now we are ready to upload our data to Power BI. Click Get Data – More – Text/CSV, then click Connect.



Navigate to the location of your data file(s), select the appropriate file(s), and then click Open. For simplicity's sake, the data for Power BI has been included within one file, ProductMixData.txt, however, all three data files could also have been used. ProductMixData.txt can be downloaded by clicking Download RASON Example Data on the Editor Ribbon.

Select the Data table on the Navigator window, then click Load. (Repeat these steps for each required data file.)

ProductMixData.txt

File Origin: 1252: Western European (Windows) | Delimiter: Comma | Data Type Detection: Based on first 200 rows

parts	prods	qty	profit	inventory
chas	tv	1	75	450
chas	stereo	1	50	450
chas	speakers	0	35	450
tube	tv	1	75	250
tube	stereo	0	50	250
tube	speakers	0	35	250
cone	tv	2	75	800
cone	stereo	2	50	800
cone	speakers	1	35	800
psup	tv	1	75	450
psup	stereo	1	50	450
psup	speakers	0	35	450
elec	tv	2	75	600
elec	stereo	1	50	600
elec	speakers	1	35	600

Load Edit Cancel

Multiple data fields are added beneath "Fields". These are our data fields.

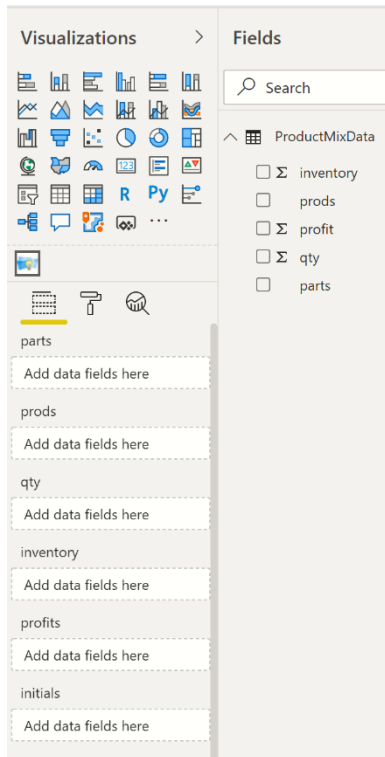
Fields

Search

ProductMixData

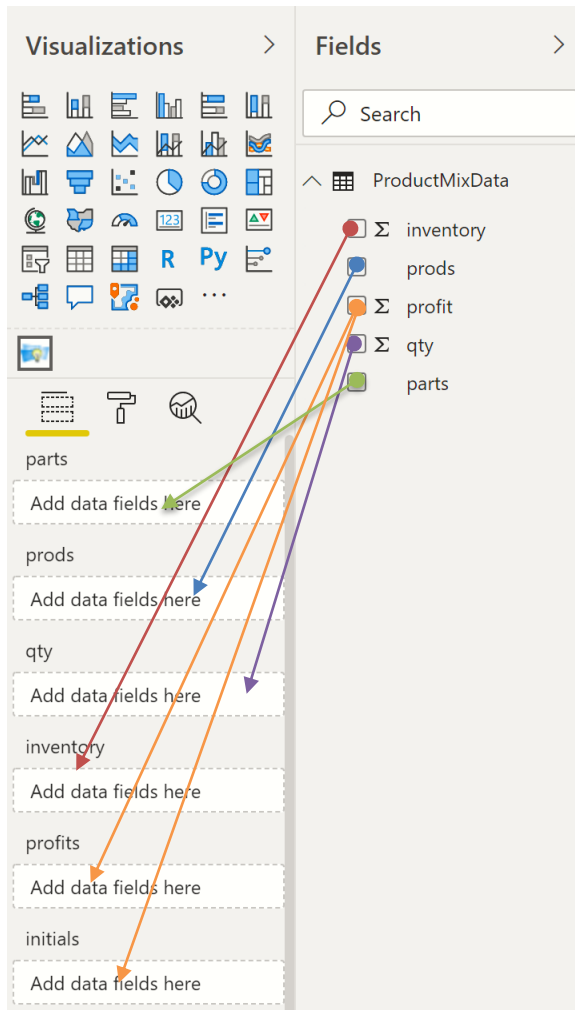
<input checked="" type="checkbox"/> Σ inventory
<input checked="" type="checkbox"/> prods
<input checked="" type="checkbox"/> Σ profit
<input checked="" type="checkbox"/> Σ qty
<input checked="" type="checkbox"/> parts

Select the Solver icon in Visualizations and notice new data "wells" appearing below. These "data wells" were created by the parts_data, profit_data and invent_data data sources.

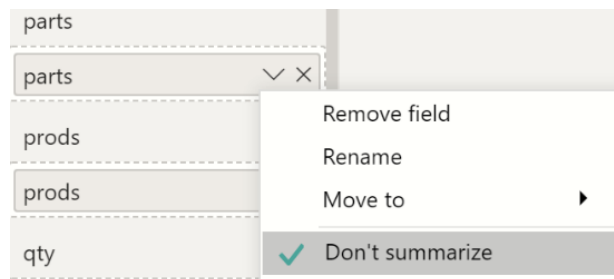
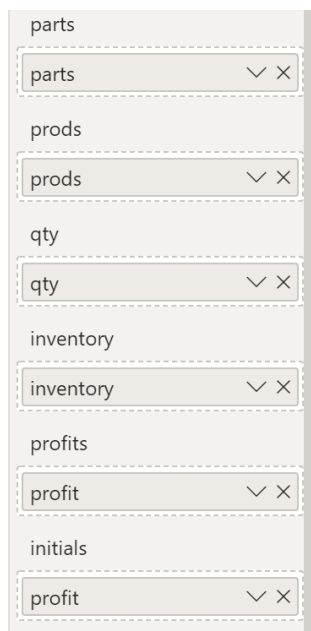


Drag the data fields into the data wells according to the following table.

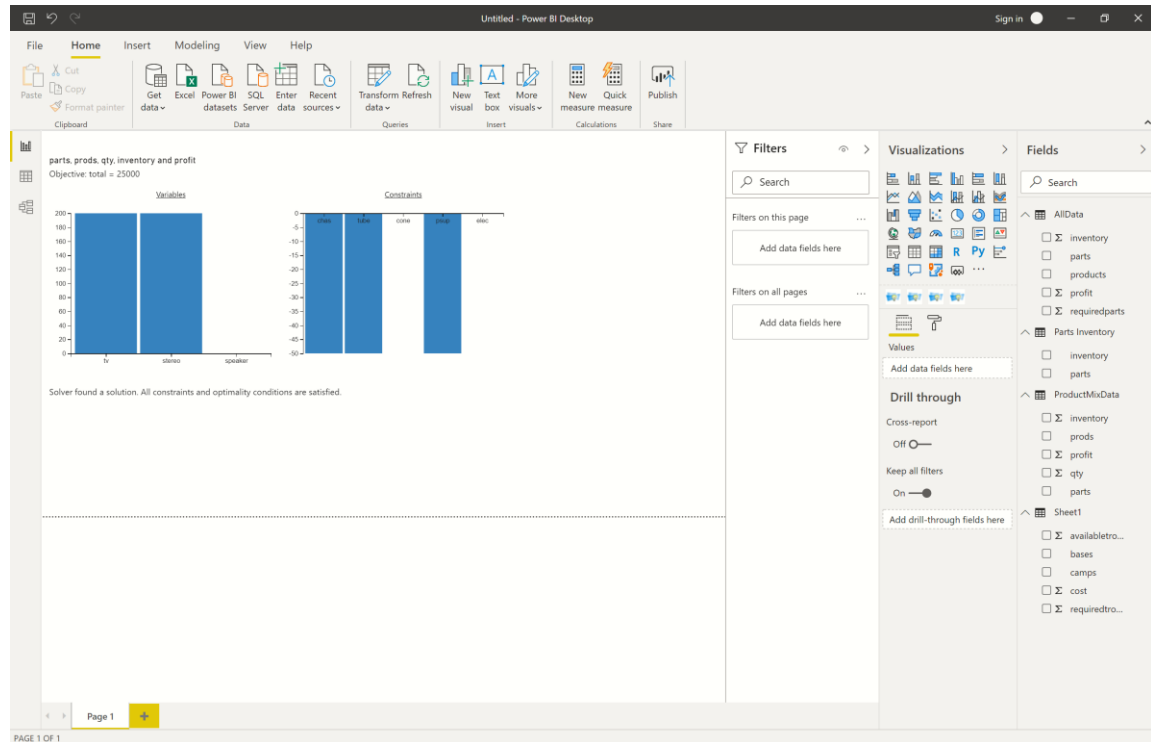
Data Field	TO	Data Well
inventory	TO	inventory
parts	TO	parts
prods	TO	prods
profit	TO	profits
profit	TO	initials
parts	TO	parts



Afterwards, your task pane should match the following screenshot. Be sure to click the down arrow next to each data field and select Don't Summarize.



Immediately, the RASON model is submitted to the RASON Server, the model is solved and the final variable values are imported back into Power BI.



In the chart, we see that final variable values are: Var1 (LCD TV) equal to 200, Var2 (Stereo) equal to 200 and Var3 (Speakers) equal to 0. These variable values result in an objective function value equal to \$25,000.

At the bottom of the custom visual, we find Solver's result message: Solver found a solution. All constraints and optimality conditions are satisfied. Recall that since we are solving a linear model, this message indicates that Solver has found the globally optimal solution: There is no other solution satisfying the constraints that has a better value for the objective. For more information on this Solver result, please see the chapter "Solver Results Messages" within the *Frontline Solvers Reference Guide*.

If our data were to change, you would simply need to update the data source, refresh the source within Power BI and watch as the Tableau Extension solved the model using the new data.

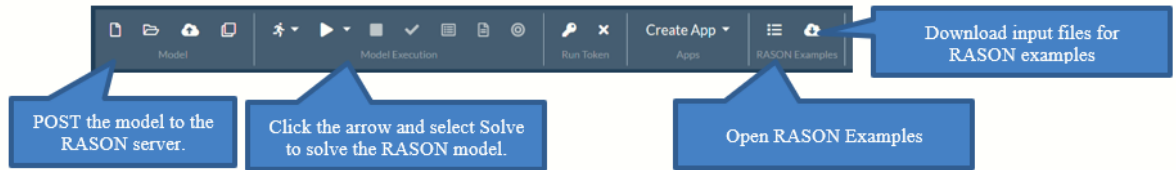
Scoring New Data within Power BI

RASON Decision Services includes the ability to score a data science or forecasting model within Power BI. As introduced in the previous section, What's New in Analytic Solver V2021.5, users can deploy and share data science and machine learning models, trained in Analytic Solver or RASON, to the Azure cloud, and use them directly for classification and prediction (without needing auxiliary "code" in R or Python, RASON or Excel). This section gives a step by step illustration of this new feature.

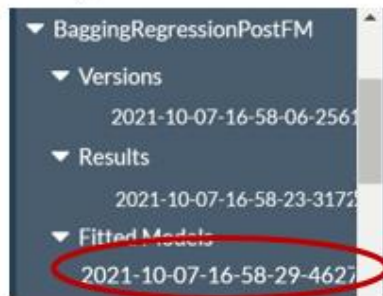
Open Bagging.json regression model by clicking the Editor tab on RASON.com, then Open RASON Example Model – Data Science – Regression – Bagging.json. This model fits a model to the hald-small-binary.txt dataset using the bagging ensemble regression method. The fitted model will be POSTed to the user's RASON account.

First attach the data file, hald-small.txt by clicking Choose Files on the Properties pane (on the right). This model can be downloaded by clicking Download RASON example data within the RASON Examples section of the ribbon. Next, post the model to the RASON server by clicking the POST

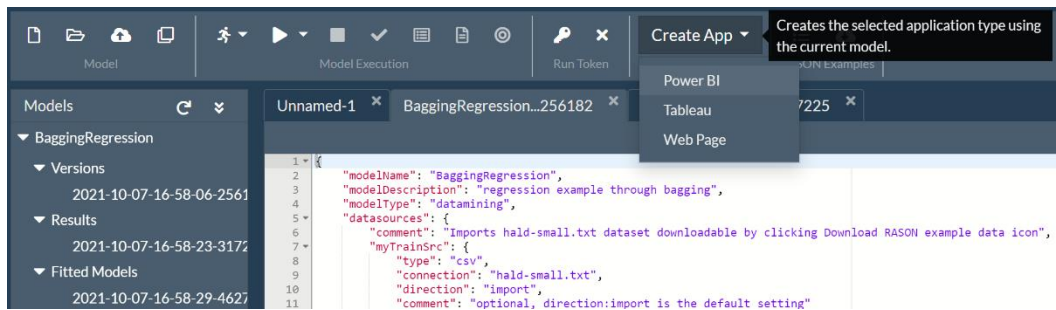
ration.net/api/model icon on the RASON ribbon. Then solve the model by clicking the down arrow next to the POST rason.net/api/model/id/solvetype icon and select Solve.



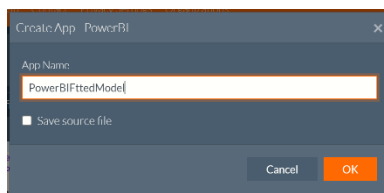
Notice on the Models pane that the RASON model, the results, and the fitted model are all now residing on the RASON server.



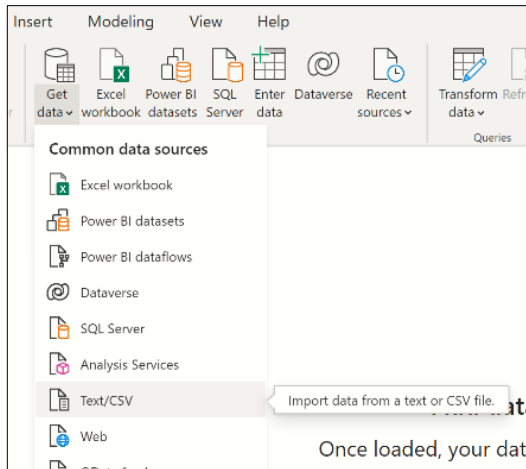
Select the Fitted Model, then click Create App – Power BI on the RASON ribbon to create a Power BI custom visual using the fitted model.



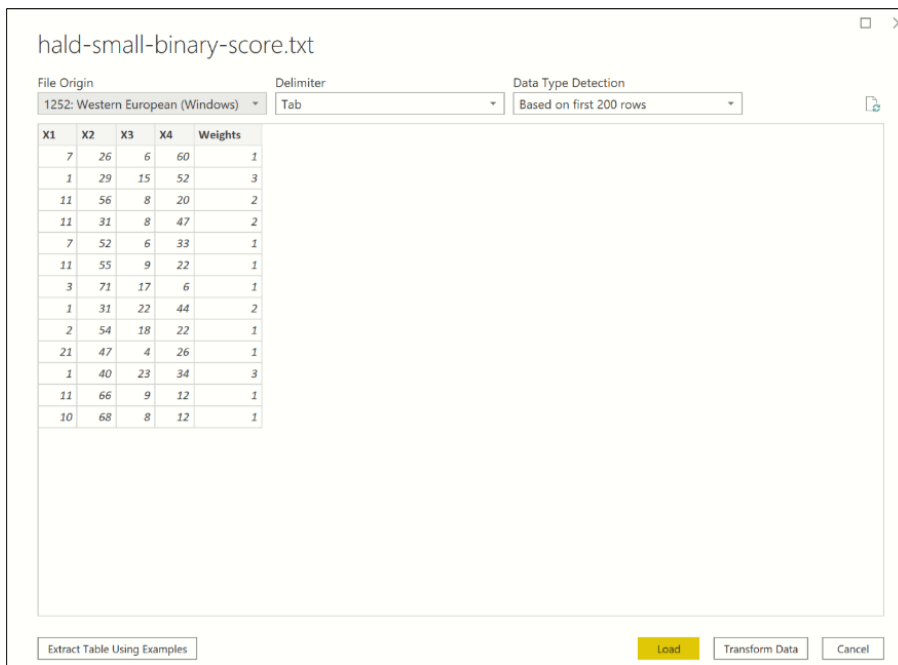
Give the Power BI Custom Visual a name, such as "PowerBIFittedModel", then click OK. Creating the custom visual could take up to 10 minutes. See below for an explanation of the "Save source file" checkbox.



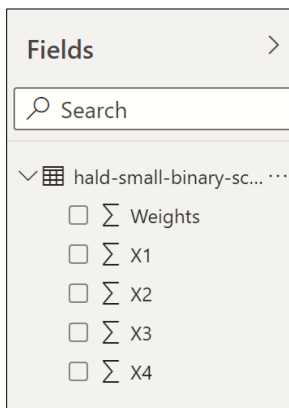
Once the custom visual has been created, open Power BI and click Get Data – Text/CSV to upload the data to be scored to Power BI.



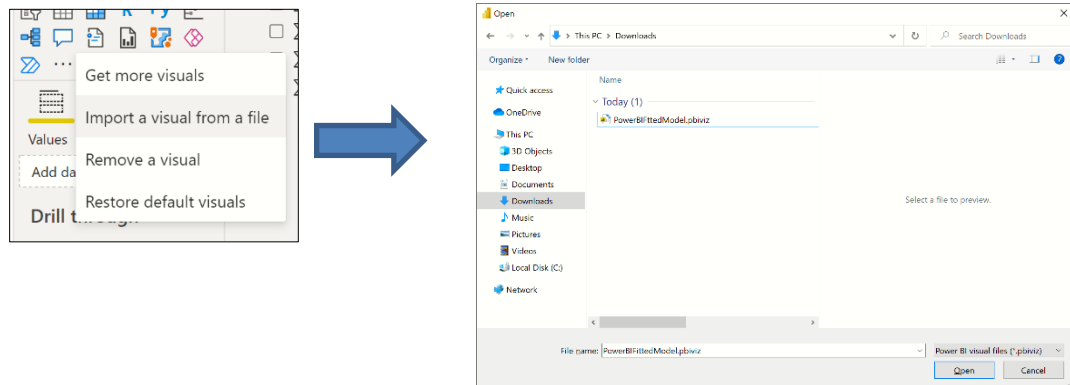
Select hald-small-binary-score.txt, also downloadable from Download RASON example data, then click Load to load the data into Power BI.



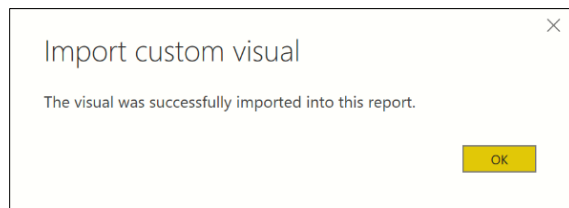
The dataset will appear in Power BI under Fields. Click the arrow to expand the file.



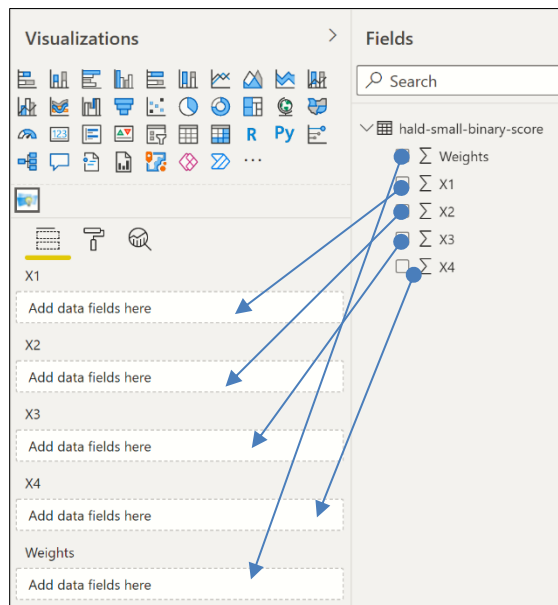
Click the three dots under Visualizations, then "Import a visual from a file", to open the newly created and downloaded Power BI custom visual, typically under Downloads.



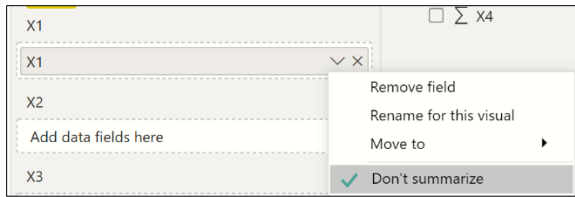
If the import is successful, the following message will be displayed. Click OK to clear this message.



Then click the newly imported Power BI custom visual to open the data wells.



Match the data columns from hald-small-binary-score to the custom visual data wells as shown below. Make sure to click the down arrow next to each data feature and select "Don't summarize".

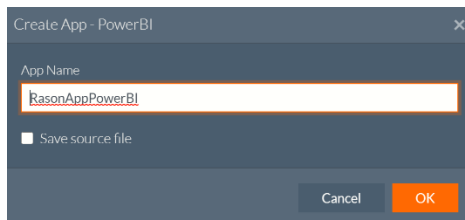


Immediately, the half-small-binary-score dataset is scored using the Power BI custom visual.

	X1	X2	X3	X4	Weights	Y
1	29	15	52	3		74.3
1	40	23	34	3		90.85
1	31	22	44	2		87.6
11	31	8	47	2		87.6
11	56	8	20	2		110.1
2	54	18	22	1		103.55000000000001
3	71	17	6	1		103.65
7	26	6	60	1		78.5
7	52	6	33	1		98.5
10	68	8	12	1		109.4
11	55	9	22	1		112.55000000000001
11	66	9	12	1		111.35
21	47	4	26	1		115.9

Advanced Settings

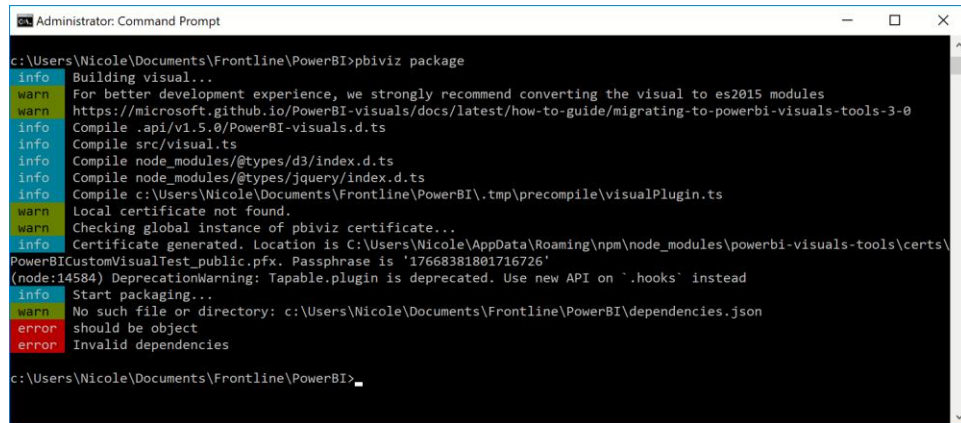
Recall from the examples above, once you click Create App - Power BI, a dialog appears containing a checkbox for "Save Source".



When this option is selected, a compressed file is created containing the pbviz file. Save the contents of this file to a desired location. This file will contain the source code for the custom visual.

You will find the type script file, visual.ts within the src folder. Users can manipulate this file to customize the look and feel of the custom visual, such as the face of the custom visual icon (within Visualizations), alter the RASON model, etc. For more information on customizing the visual.ts file, see the webpage: <https://powerbi.microsoft.com/en-us/documentation/>

Note: If the visual.ts file is altered, a new pbviz package must be created. To do so, open a command prompt, change the directory to the root directory of the custom visual and enter "pbviz package", as shown in the screenshot below.



```
Administrator: Command Prompt
c:\Users\Nicole\Documents\Frontline\PowerBI>pbiviz package
info Building visual...
warn For better development experience, we strongly recommend converting the visual to es2015 modules
https://microsoft.github.io/PowerBI-visuals/docs/latest/how-to-guide/migrating-to-powerbi-visuals-tools-3-0
info Compile .api/v1.5.0/PowerBI-visuals.d.ts
info Compile src/visual.ts
info Compile node_modules/@types/d3/index.d.ts
info Compile node_modules/@types/jquery/index.d.ts
info Compile c:\Users\Nicole\Documents\Frontline\PowerBI\.\tmp\precompile\visualPlugin.ts
warn Local certificate not found.
warn Checking global instance of pbiviz certificate...
info Certificate generated. Location is C:\Users\Nicole\AppData\Roaming\npm\node_modules\powerbi-visuals-tools\certs\
PowerBICustomVisualTest_public.pfx. Passphrase is '17668381801716726'
(node:14584) DeprecationWarning: Tapable.plugin is deprecated. Use new API on '.hooks' instead
info Start packaging...
warn No such file or directory: c:\Users\Nicole\Documents\Frontline\PowerBI\dependencies.json
error should be object
error Invalid dependencies
c:\Users\Nicole\Documents\Frontline\PowerBI>
```

The new custom visual file (.pbiviz) will be saved to the Dist folder.

Uploading your RASON Model to Tableau

The Editor on RASON.com includes the ability to turn your based optimization or simulation model into a **Tableau Dashboard Extension**, with just a few mouse clicks! (Neither data science nor decision table models are supported at this time.) Tableau is an industry leading desktop or cloud-based interactive business intelligence tool that centers on data visualization, dashboarding and data discovery.

To start, simply designate rows or columns of data to serve as changeable parameters, then choose **Create App – Tableau** on the Editor page on www.RASON.com, and save the file created by the RASON Server. In Tableau, you'll see the newly-created file under **Extensions** on the left side of the dashboard, where you can drag it onto your dashboard. You'll be prompted to match the parameters your model needs with data in Tableau. What you get isn't just a chart – it's your *full optimization or simulation model*, ready to accept Tableau data, **run on demand** (using our **RASON** server), and display visual results in Tableau!

In order to use this new feature in Analytic Solver, you must first *install Tableau V2018.2 or later* or open the free cloud based version of Tableau. For more information on this business intelligence tool, see the following website: <https://www.tableau.com/>.

Solving RASON Models in Tableau

In this section, we will create a custom Tableau Extension using the Product Mix example model. Log on to www.RASON.com and then click on the Editor tab. Click the RASON Examples folder (on the top right) and then Optimization With Data Binding - ProductMixCsv4. For a complete description of this model, see the Defining your Optimization Model chapter that appears previously in this guide.

In order to create an Extension in Tableau, your RASON model must be formulated using an index set. The example code below creates two ordered sets, part and prod. The part set contains five items (in order as entered): chas, tube, cone, psup, and elec while the prod set contains 3 items: tv, stereo, and speakers. An indexSet is always defined as a JSON object{ }. For more information on index sets, see the RASON Reference Guide.

The example code below uses the `dataSources` section to create three datasources, `parts_data`, `invent_data` and `profit_data`; using three CSV files. Any supported file type may be used. As discussed in previous sections, a `dataSource` allows data to be passed to the RASON Server using an external data file in a supported file type. By using an external data file, we can submit updated data to the RASON Server without having to edit our existing RASON model. For example, imagine that Purchasing was able to obtain a discount on Speakers and our inventory increased from 800 to 1,000.

If we were not using an external data file, we would have to edit the RASON model directly to reflect this change.

```
{
  "modelName": "ProductMixCSV4",
  "modelType": "optimization",
  "comment": "Example of using CSV table binding to read params and save results",
  "indexSets": [{
    "name": "parts",
    "value": ["chas", "tube", "cone", "psup", "elec"]
  }, {
    "name": "prods",
    "value": ["tv", "stereo", "speaker"]
  }],
  "datasources": {
    "parts_data": {
      "type": "csv",
      "connection": "ProductMixParts.txt",
      "indexCols": ["parts", "prods"],
      "valueCols": ["qty"]
    },
    "invent_data": {
      "type": "csv",
      "connection": "ProductMixInventory.txt",
      "indexCols": ["parts"],
      "valueCols": ["inventory"]
    },
    "profit_data": {
      "type": "csv",
      "connection": "ProductMixProfits.txt",
      "indexCols": ["prods"],
      "valueCols": ["profits"]
    }
  },
  "data": [{
    "name": "parts2",
    "binding": "parts_data",
    "valueCol": "qty"
  }, {
    "name": "invent",
    "binding": "invent_data",
    "valueCol": "inventory"
  }, {
    "name": "profit",
    "binding": "profit_data",
    "valueCol": "profits"
  }],
  "formulas": [{
    "name": "piv_parts",
    "formula": "PIVOT(parts2, { 'prods' }, { 'parts' })"
  }],
  "variables": {
    "name": "x",
    "dimensions": ["prods"],
    "value": 0,
    "lower": 0,
    "finalValue": [],
    "indexValue": []
  },
  "constraints": [{
    "name": "c",
    "dimensions": ["parts"],
    "formula": "MMULT(piv_parts, x) - invent[]",
    "upper": 0,
    "finalValue": []
  }],
  "objective": {
    "name": "total",
    "formula": "sumproduct(x, profit[])",
    "type": "maximize"
  }
}
```

}

Data Sources: ProductMixInventory.txt, ProductMixParts.txt and ProductMixProfits.txt

The image shows three separate CSV files side-by-side. The first file, ProductMixInventory.txt, has columns 'parts', 'inventory' and contains 6 rows of data. The second file, ProductMixParts.txt, has columns 'parts', 'prods', 'qty' and contains 12 rows of data. The third file, ProductMixProfits.txt, has columns 'prods', 'profits' and contains 4 rows of data.

parts	inventory
chas	450
tube	250
cone	800
psup	450
elec	600

parts	prods	qty
chas	tv	1
chas	stereo	1
tube	tv	1
cone	tv	2
cone	stereo	2
cone	speaker	1
psup	tv	1
psup	stereo	1
elec	tv	2
elec	stereo	1
elec	speaker	1

prods	profits
tv	75
stereo	50
speaker	35

DataSources

The last two data sources, `invent_data` and `profit_data`, include one `indexCol` and one `valueCol`. The `indexCol` is the column containing the indices or the labels and the `valueCol` is the column containing the values for the indices. The third data source, `parts_data`, contains two index columns, `parts` and `products`, i.e., a LCD TV requires 2 Electronic components. For more information on DataSources, see the RASON Reference Guide.

```
datasources: {
  parts_data: {
    type: "csv",
    connection: "ProductMixParts.txt; header",
    indexCols: ['parts', 'prods'],
    valueCols: ['qty'],
    direction: "import"
  },

  invent_data: {
    type: "csv",
    connection: "ProductMixInventory.txt; header",
    indexCols: ['parts'],
    valueCols: ['inventory'],
    direction: "import"
  },

  profit_data: {
    type: "csv",
    connection: "ProductMixProfits.txt; header",
    indexCols: ['prods'],
    valueCols: ['profits'],
    direction: "import"
  }
}
```

Data

In the data section, "profit" is bound to the column "profits" within the data source, "profit_data", "parts2" is bound to the "qty" column within the "parts_data" datasource and "invent" is bound to the "inventory" column within the "invent_data" datasource.

```
data: {
```

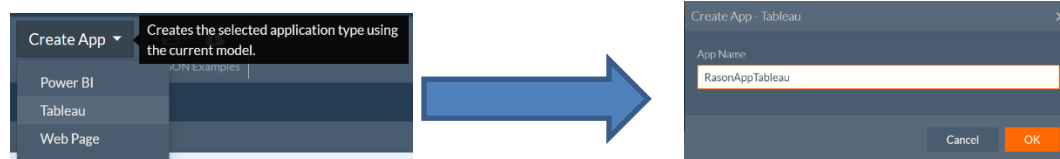
```

parts2: {
  binding: "parts_data", valueCol: 'qty'
},
invent: {
  binding: "invent_data", valueCol: 'inventory'
},
profit: {
  binding: "profit_data", valueCol: 'profits'
}
},

```

The three variables are specified within variables, the constraints are created within constraints (using a Pivot Table) and the objective is computed and maximized within objective. For more information on using a pivot table within RASON, see the *Using Tables* section that occurs in the chapter, **Using Array Formulas, For Loops and Tables in RASON**.

Click Create App – Tableau to create the Tableau Extension.

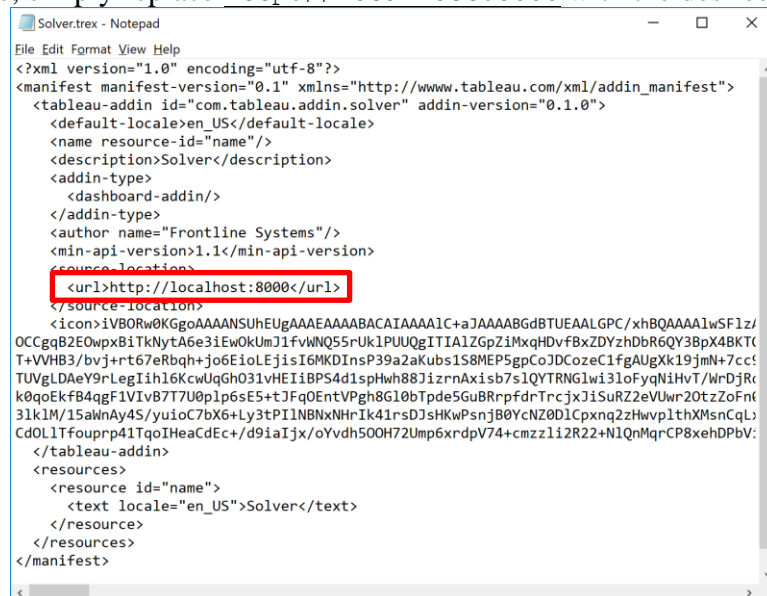


Confirm the name of the extension.

Once you click OK, the Tableau Extension is downloaded locally.

Viewing the Tableau Extension File

Open File Explorer and browse to the folder where your downloads are saved. Unzip the file, RASON App Tableau. Within the contents of the unzipped folder will be an Extensions folder. Click that folder and open the Solver.trex file using Notepad. This file contains information such as the extensions name (or how it will appear in Tableau) and the url where the extension is hosted. To publish the html file to a website, simply replace <http://localhost:8000> with the desired url.



Starting up the Server

Since Tableau extensions are simply web pages, we will first need to start up a web server to serve our content. For this example, we will serve up the webpage to the default location. To do so, open a command prompt, navigate to the root of the extensions repository and run “http-server -p 8000”.



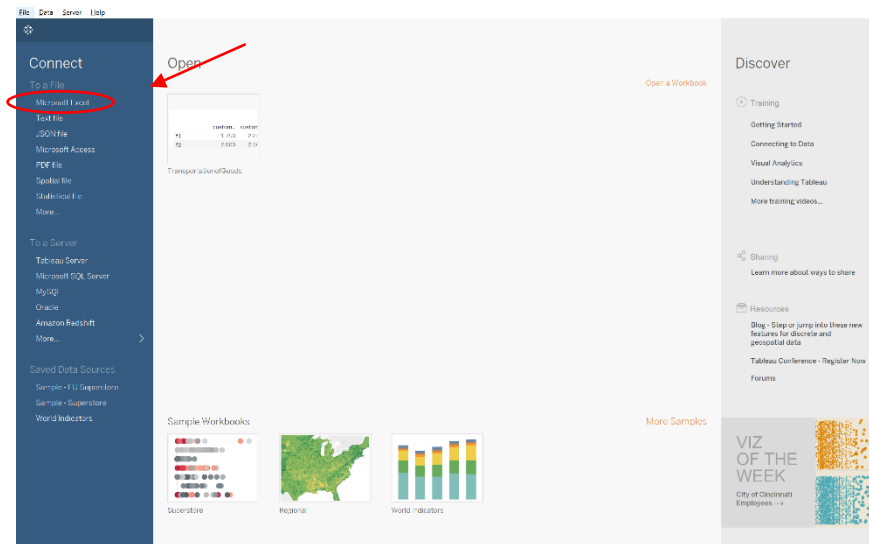
```
Command Prompt - http-server -p 8000

c:\FrontlineSystems\TableauExtensions>http-server -p 8000
Starting up http-server, serving ./
Available on:
  http://192.168.0.24:8000
  http://127.0.0.1:8000
Hit CTRL-C to stop the server
```

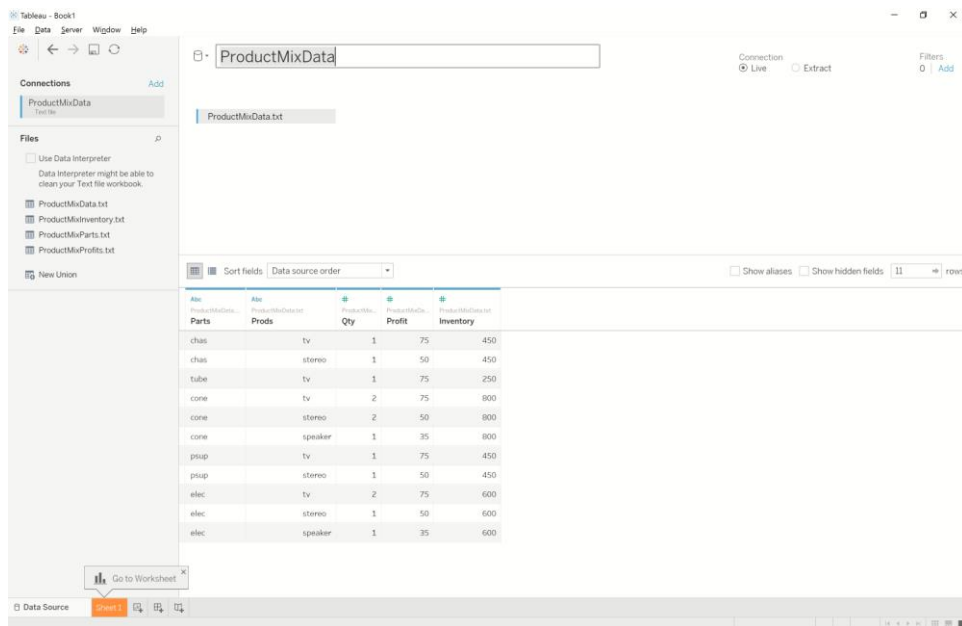
This command starts up a simple http server listening on port 8000.

Opening Extensions within Tableau

Open either desktop, server or cloud-based Tableau. The screenshot below depicts the opening screen of desktop Tableau. Click *Text File* under **Connect**.

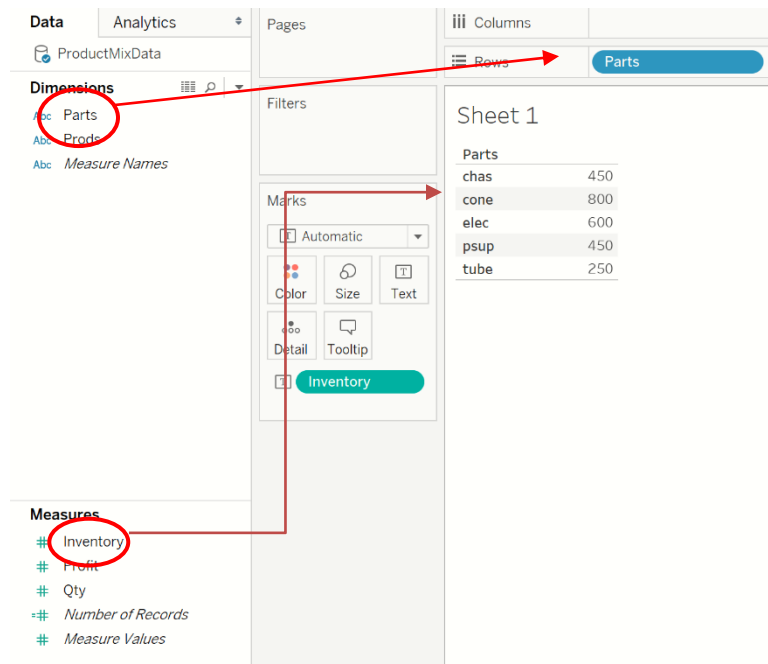


An Open dialog opens. Browse to the location of your data file(s) and click Open. For simplicity's sake, all data has been entered into a single text file for import into Tableau, ProductMixData.txt which can be downloaded by clicking the Download RASON Example Data on the Editor ribbon. A new worksheet, Sheet1, is automatically created. Click on the Sheet 1 tab.

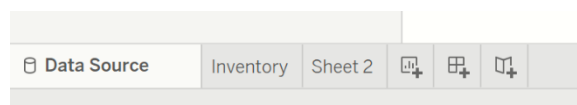


On Sheet 1, drag Parts (under Dimensions) to Sheet 1 rows and then drag Inventory (under Measures) to the values.


Important: Hover over , click the down arrow that appears to the right, then select Dimensions from the menu.



Right click the Sheet 1 tab and rename the worksheet to “Inventory”, then click the New Sheet icon at the bottom of Tableau to open a new worksheet.



On Sheet 2, drag Prods (under Dimensions) to Sheet 2 columns and then drag Profit (under Measures) to the

column values. Hover over , click the down arrow that appears to the right, then select Dimensions from the menu. Rename Sheet2 to Profit and open a new worksheet.

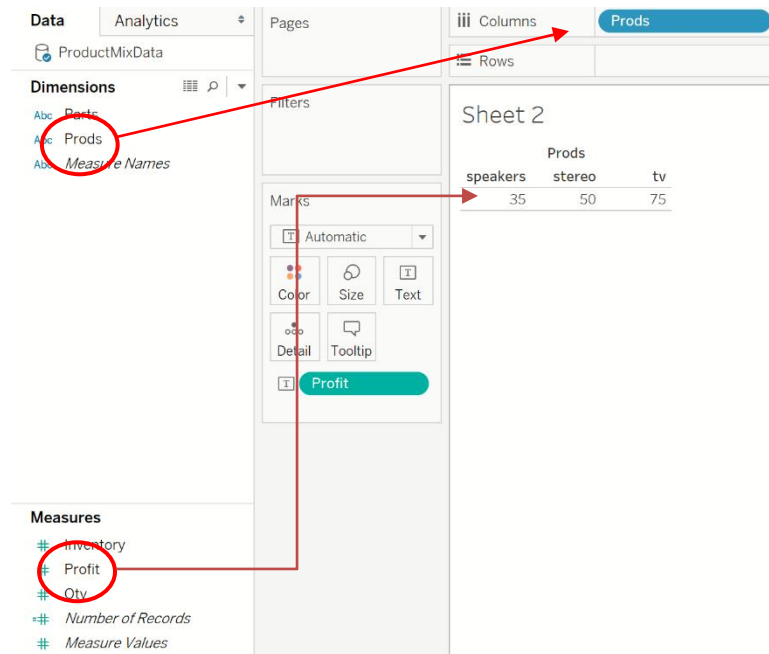


Tableau interface showing the setup for Sheet 2. The Dimensions shelf contains 'Prods' and 'Measure Names'. The Measures shelf contains 'Profit'. The Marks card is set to 'Automatic'. The Columns shelf is empty. The Rows shelf is empty. The view shows a table with columns 'speakers', 'stereo', and 'tv' and rows for 'Prods'.

Prods	speakers	stereo	tv
speakers	35	50	75

Add a new worksheet and drag “Parts”, under Dimensions, to Rows and “Products”, under Dimensions, to “Columns”. Then drag “RequiredParts”, under Measures, to the field elements.

Hover over Qty, click the down arrow that appears to the right, and select Dimensions from the menu.

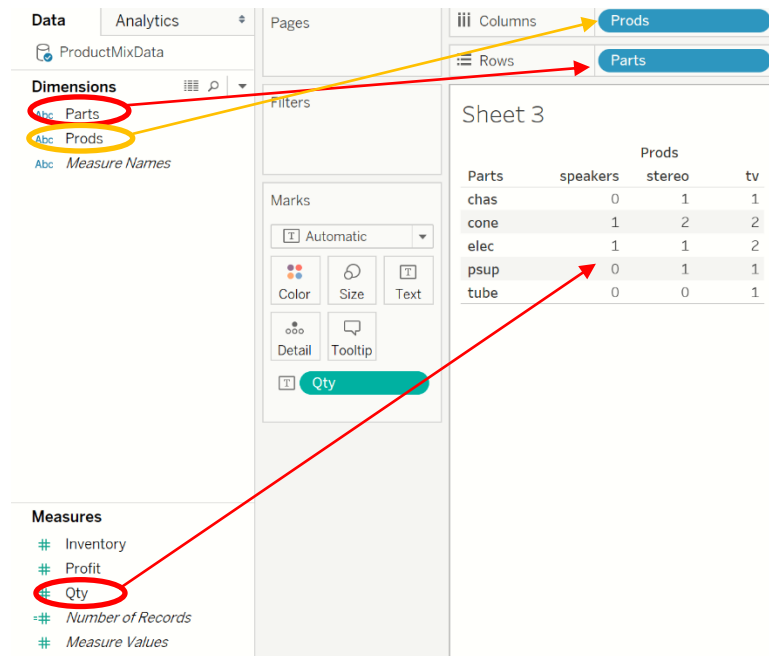


Tableau interface showing the setup for Sheet 3. The Dimensions shelf contains 'Parts' and 'Prods'. The Measures shelf contains 'Qty'. The Marks card is set to 'Automatic'. The Columns shelf is empty. The Rows shelf is empty. The view shows a table with columns 'speakers', 'stereo', and 'tv' and rows for 'Parts'.

Parts	speakers	stereo	tv
chas	0	1	1
cone	1	2	2
elec	1	1	2
psup	0	1	1
tube	0	0	1

Rename the sheet “Required Parts”. Then create a new dashboard.

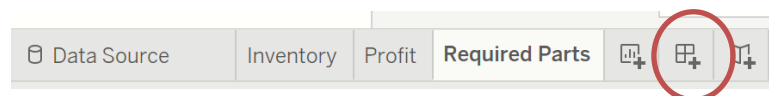
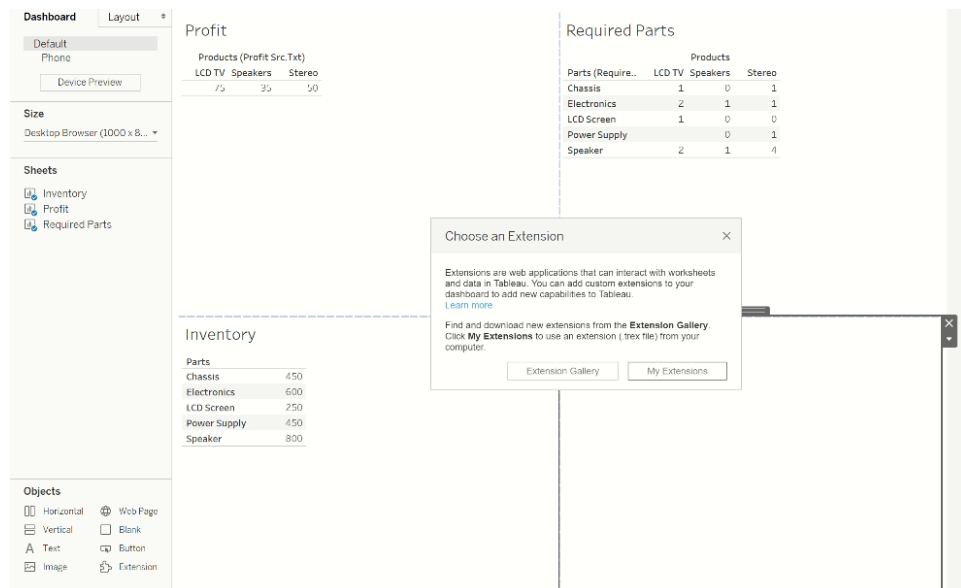
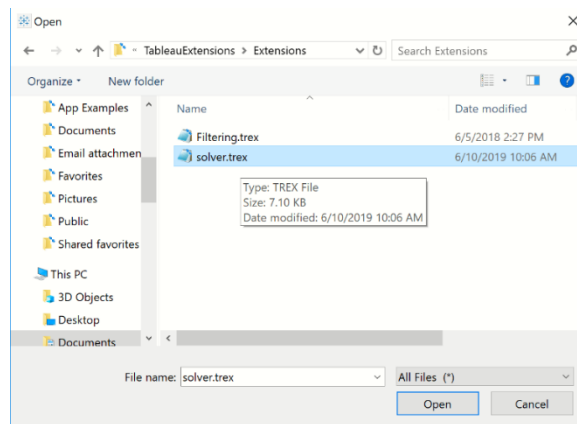


Tableau dashboard bar showing the 'Required Parts' sheet selected. The 'New Dashboard' button is circled in red.

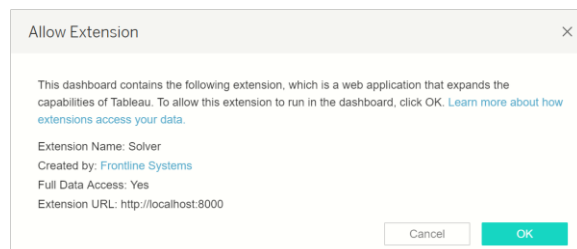
At this point we will connect the dimensions in the Tableau extension with the actual data. Drag all three sheets, Inventory, Required Parts and Profit and a new Extension (under Objects at bottom left) to the canvas.



Click My Extensions and browse to the location of the Tableau Extension file (Solver.trex). In this example, the file was saved to the folder, TableauExtensions. Select All Files, then double click the Extensions folder and select the Solver.trex file.



When asked to "Allow Extension", click OK.



At this point we will connect the dimensions with the actual data. For the first data source, parts, select "Inventory" for Sheet and "parts" for Column to match the "parts" dimension in the CSV file to the "parts" dimension in the Tableau Inventory sheet. Then click OK.

Select a data source for: parts

Sheet:

Column:

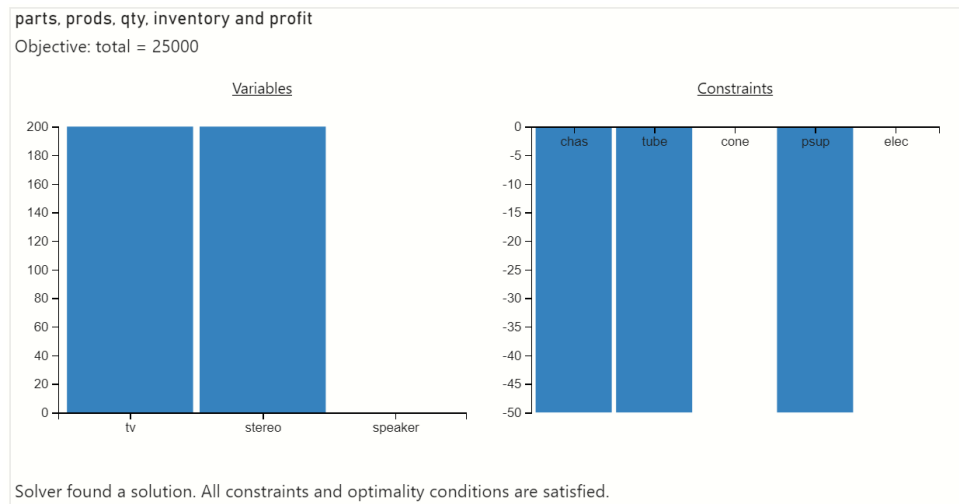
OK

Similar steps must be taken to connect the data with the correct dimension as shown in the table below.

Data Source	Worksheet	Column
parts	Inventory	parts
inventory	Inventory	inventory
prods	Required Parts	prods
profit	Profit	profit
qty	Required Parts	Qty

Immediately, the RASON model is submitted to the RASON Server, the model is solved with the LP/Quadratic engine and the final variable values are imported back into Tableau.

At the bottom of the extension, we find Solver's result message: Solver found a solution. All constraints and optimality conditions are satisfied. In the Variables chart, we see that the final variable values are: Var1 (LCD TV) equal to 233, Var2 (Stereo) equal to 133 and Var3 (Speakers) equal to 0. These variable values result in an objective function value equal to 30,833. In the chart to the right, we see the final constraint values are: Con1 (number of Chassis used) = 367, Con2 (number of LCD Screens used) = 233, Con3 (number of Speakers used) = 1,000, Con4 (number of Power Supplies used) = 367 and Con5 (number of Electronics used) = 600. Notice that with the addition of 2 more speakers required to manufacturing Speakers, the increased unit profit of Stereos to \$100 and the added Speaker inventory, our profit has increased by \$5,833.



In the chart, we see that final variable values are: Var1 (LCD TV) equal to 200, Var2 (Stereo) equal to 200 and Var3 (Speakers) equal to 0. These variable values result in an objective function value equal to \$25,000.

At the bottom of the custom visual, we find Solver's result message: Solver found a solution. All constraints and optimality conditions are satisfied. Recall that since we are solving a linear model, this message indicates that Solver has found the globally optimal solution: There is no other solution satisfying the constraints that has a better value for the objective. For more information on this Solver result, please see the chapter "Solver Results Messages" within the *Frontline Solvers Reference Guide*.

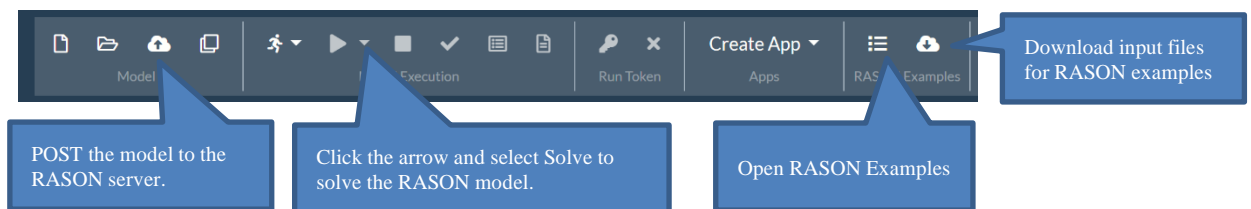
If our data were to change, you would simply need to update the data source, refresh the source within Tableau and watch as the Tableau Extension solved the model using the new data.

Scoring New Data within Tableau

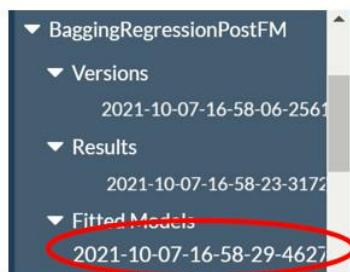
RASON Decision Services includes the ability to score a data science or forecasting model within Tableau. As introduced in the previous section, What's New in Analytic Solver V2021.5, users can deploy and share data science and machine learning models, trained in Analytic Solver or RASON, to the Azure cloud, and use them directly for classification and prediction (without needing auxiliary “code” in R or Python, RASON or Excel). This section gives a step by step illustration of this new feature.

Open Bagging.json regression model by clicking the Editor tab on www.RASON.com, then Open RASON Example Model (in the RASON Examples section of the ribbon) – Data Science – Regression – Bagging.json. This model fits a model to the hald-small.txt dataset using the bagging ensemble regression method. The fitted model will be POSTed to the user's RASON account.

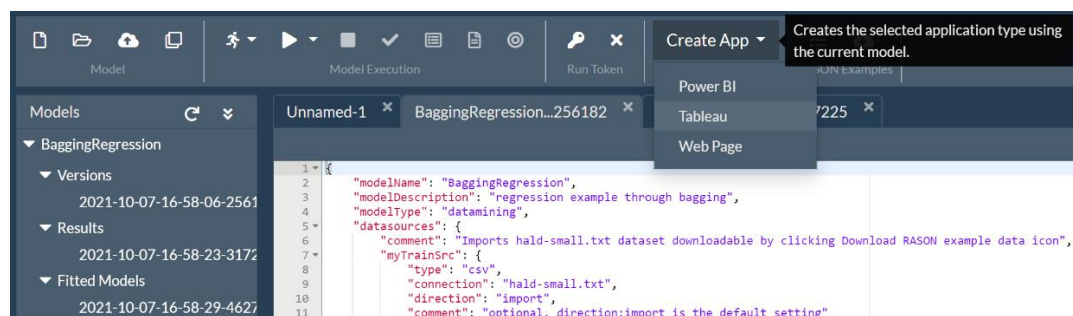
First attach the data file, hald-small-binary.txt by clicking Choose Files on the Properties pane (on the right). This model can be downloaded by clicking Download RASON example data within the RASON Examples section of the ribbon. Next, post the model to the RASON server by clicking the POST rason.net/api/model icon on the RASON ribbon. Then solve the model by clicking the down arrow next to the POST rason.net/api/model/id/solvetype icon and select Solve.



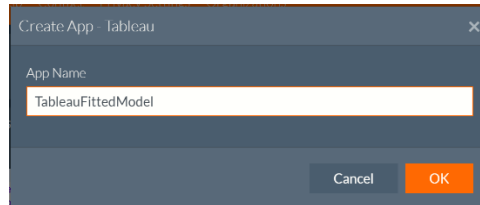
Notice on the Models pane that the RASON model, the results, and the fitted model are all now residing on the RASON server.



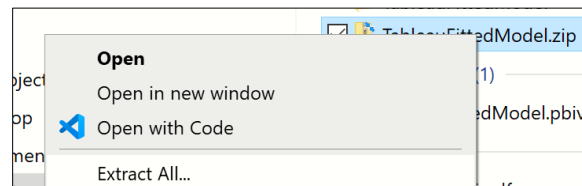
Select the Fitted Model, then click Create App – Tableau on the RASON ribbon to create a Tableau custom extension using the fitted model.



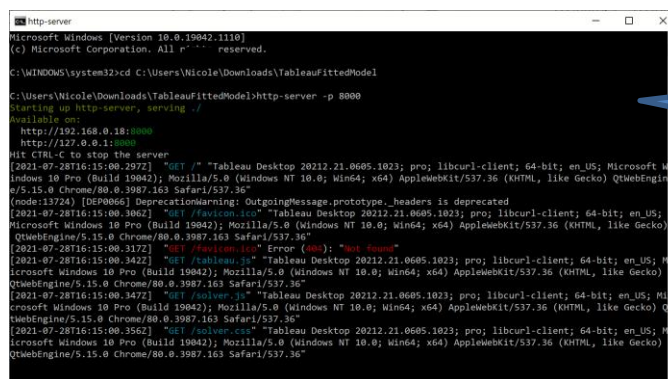
Give the Tableau Custom Extension a name, such as "TableauFittedModel", then click OK. Creating the custom extension could take up to 10 minutes. See below for an explanation of the "Save source file" checkbox.



Right click the downloaded file and unzip the contents. This example unzips the contents of TableauFittedModel.zip to the Downloads folder.

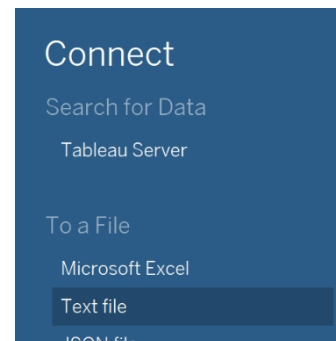


Recall that since Tableau extensions are simply web pages, we will first need to start up a web server to serve our content. For this example, we will serve up the webpage to the default location. To do so, open a command prompt, navigate to the root of the extensions repository and run "http-server -p 8000".



The contents of the downloaded model, TableauFittedModel.zip, were extracted to Downloads\TableauFittedModel.

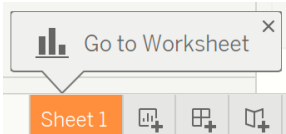
Once the custom extension has been created, open Tableau and click Connect – Text file to upload the data to be scored, to Tableau.



Select hald-small-binary-score.txt, also downloadable from Download RASON example data, then click Load to load the data into Power BI.

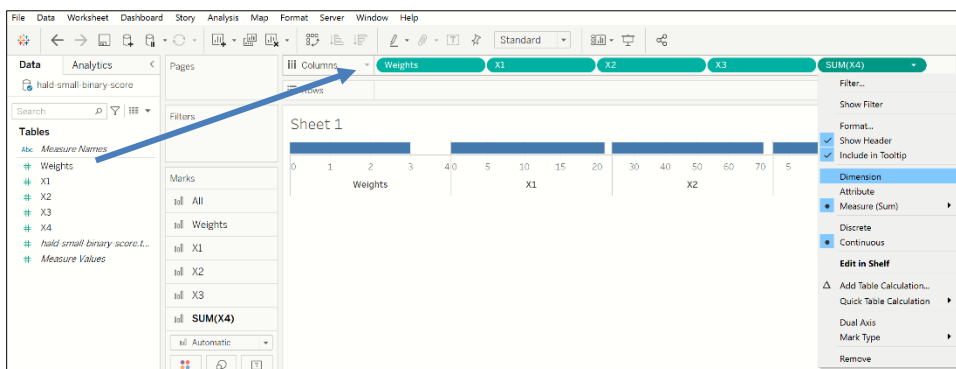
#	#	#	#	#
hold-small...	hold-small...	hold-small...	hold-small...	hold-small-binary...
X1	X2	X3	X4	Weights
7	26	6	60	1
1	29	15	52	3
11	56	8	20	2
11	31	8	47	2
7	52	6	33	1
11	55	9	22	1
3	71	17	6	1
1	31	22	44	2
2	54	18	22	1
21	47	4	26	1
1	40	23	34	3
11	66	9	12	1
10	68	8	12	1

Click Sheet 1 to open a Tableau worksheet.

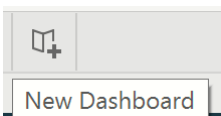


Drag the column headings from beneath Tables to Columns.

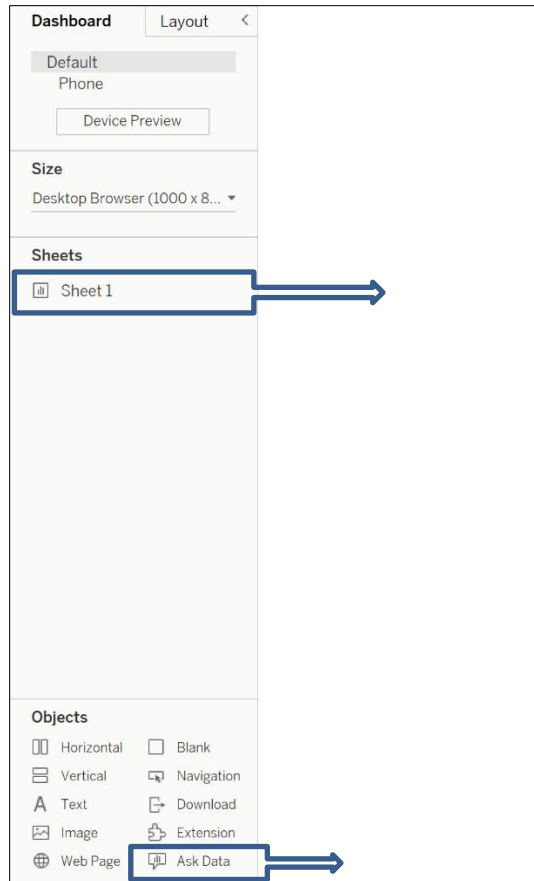
Click the down arrow next to each column heading and select "Dimension".



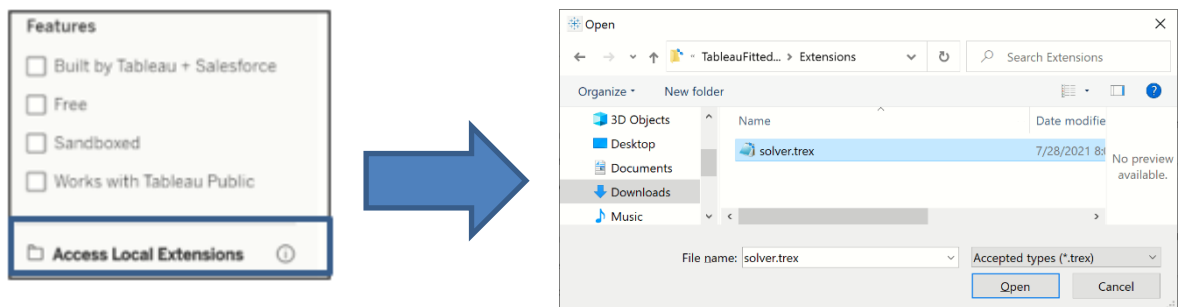
At the bottom, click New Dashboard to open a new Tableau dashboard.



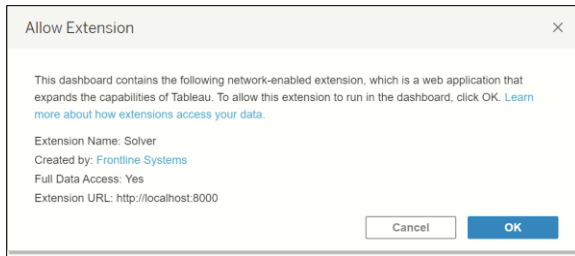
Once opened, drag Sheet 1, under Sheets, and Extension, under Objects to the dashboard.



The "Add an Extension" dialog opens. On this dialog, click **Access Local Extensions** on the bottom left and browse to the location of the Tableau .trex file.

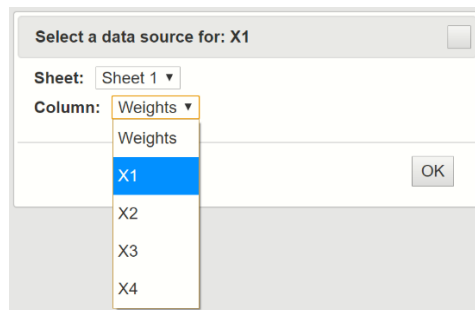


Click Open to open the Tableau extension. Then click OK to allow the extension to run the Tableau dashboard.

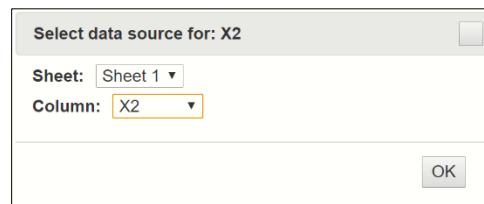


A series of dialogs will appear which allow the dimensions from the Tableau worksheet to be matched with the dimensions in the Solver custom Tableau extension (Solver.trex).

- Match the data source "X1" dimension with the worksheet "X1" dimension.



- Match the data source "X2" dimension with the worksheet "X2" dimension.



- Match the data source "X3" dimension with the worksheet "X3" dimension.



- Match the data source "X4" dimension with the worksheet "X4" dimension.



- Match the data source "Weights" dimension with the worksheet "Weights" dimension.

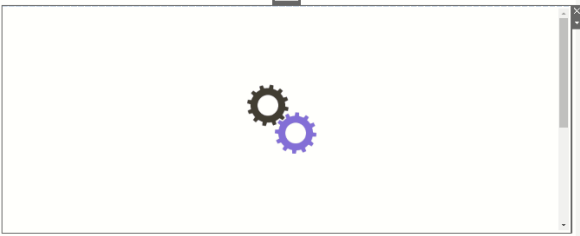
Select data source for: Weights

Sheet: Sheet 1

Column: Weights

OK

Once the last dimension is matched, Tableau immediately begins generating the custom extension.



The screenshot below displays the completed custom extension.

Output results contain the data to be scored plus the column containing the prediction of the output variable, Y.

Y scoring results					
X1	X2	X3	X4	Weights	Y
7	26	6	60	1	78.5
1	29	15	52	3	74.3
11	56	8	20	2	110.1
11	31	8	47	2	87.6
7	52	6	33	1	98.5
11	55	9	22	1	112.55000000000001
3	71	17	6	1	103.65
1	31	22	44	2	87.6
2	54	18	22	1	103.55000000000001
21	47	4	26	1	115.9
1	40	23	34	3	90.85
11	66	9	12	1	111.35
10	68	8	12	1	109.4

Using Array Formulas, For Loops and Tables in RASON

Introduction

In this section you'll learn about additional components that may be used within various sections of a RASON model to calculate the constraints and objective in an optimization or simulation optimization model and the uncertain functions in a simulation or simulation optimization model. Each of these components may be used in various sections such as the formula section to calculate intermediate results, the constraints section to define various constraints in an optimization model, the uncertainFunction section to define uncertain Functions in a simulation model, or anywhere else in the RASON model where the use of tables, array formulas and for-loops will create an efficient, easy-to-read and program model.

Array Formulas

This section discusses array formulas as implemented in the RASON modeling language. In RASON, if an array formula is defined in the *constraints* section of the model, multiple constraints will be generated. If an array formula evaluates to a scalar value, then it behaves as a standard non-array formula. In sections where the formula property exists (i.e. formulas, constraints, uncertainFunctions, etc), an index operator may be used. The index operator can be applied to **arrays** (limited to 2-D) or to **tables** (limited to 8-D). These are called indexed arrays and indexed tables, respectively. (For more information, see the Index Sets section that occurs earlier in this text.) However, the use of the index property does not necessarily qualify the formula as an array. For example, the following constraint definition is not an array since the result of the formula is a scalar, or single value.

```
c1 : { formula: "sumproduct(parts['chas'], x)", upper: 450 }
```

Thus, an indexed formula may not always qualify as an indexed array formula, but an indexed array formula will always be an indexed formula.

Three types of array formulas are supported in the RASON Modeling language

- Parallel Array Formulas
- Non-Parallel Array Formulas
- Indexed Array Formulas

Continue reading to learn about each type of array formulas supported in the RASON modeling language and how each type can be used within tables and arrays in a RASON model.

Parallel array formulas

```
constraint1 : { dimensions: [6], formula: "sqrt((x - dx)^2 + (y - dy)^2) - z", upper: 0 }
```

This type of array formula is interpreted (evaluated) only once. Every operation is executed in parallel for all components of its array operands. In the above example, x, y, z are scalars, while dx and dy are

1-D arrays containing 6 elements each. The operation $(x - dx)$ is executed for all 6 elements of dx resulting in a 1-D array of 6 elements.

The multi-dimensional result of the formula evaluation is assigned to an array variable with the appropriate dimensions. In the above example, the result is held by the 1-D vertical array *constraint1*, which has 6 elements. If the formula definition is in the *constraints*: section, one constraint will be generated for each component of the result.

The result of the evaluation is a 1-D horizontal/vertical array, or 2-D array. If a table slice is presented in the formula, it is always considered a vertical array in parallel operations. Currently, the RASON modeling language supports only 2-D arrays. The **biggest disadvantage** of this array formula type is the prevention of the reverse mode of evaluation which is much faster than the native forward evaluation.

Non-parallel array formulas

```
num_used : { dimensions: [5], formula: "MMULT(parts, x)", upper:
'inventory' }
```

This type of array formula is interpreted (evaluated) only once. The array result is *not* evaluated in parallel, rather it's implicated by the inner algorithms of participating functions. In the above example, MMULT is a function, which internally executes a matrix/vector multiplication resulting in an array of size 5.

The multi-dimensional result of this formula evaluation is assigned to the array variable *num_used* dimensioned as a 1-D vertical array of size 5. Again, if the formula definition is in the *constraints*: section, one constraint will be generated for each component of the result.

The result of the evaluation is a 1-D horizontal/vertical array, or 2-D array, even in the presence of table slices. The **biggest disadvantage** of this array formula type is the prevention of the reverse mode of evaluation which (as mentioned above) is much faster than the native forward evaluation.

Indexed array formulas

The goal of indexed array formulas is to allow more flexible definitions and evaluations. In the RASON modeling language, the *for()* definition is not treated as a loop (as in a typical programming language); rather, *for()* will be treated as a declarative statement to exclusively assign attributes to indexed array formulas. The **biggest advantage** of this array formula type is that it allows the reverse mode of evaluation. Let's discuss the basics of the indexed array formulas on the next example:

```
"for(p in 'part')" : {
  "constraint[p]":{ formula:"sumproduct(parts[p, ], x) -
inventory[p]", upper:0}
}
```

The result of this evaluation is a 1-D horizontal/vertical array, 2-D array, or m-D table.

The indexed array formula does not behave in the same way as the parallel array formula. The parallel formula is interpreted only once, and hence, provides values for all result components simultaneously. The indexed array formula is executed only for a **preset combination** of the participating indices. In the example above, there is only one index in the formula, but in general the number of indices is usually > 1 .

As mentioned above, the line of code *"for (p in 'part')"* in the RASON modeling language does not behave in the same way as a *"for"* statement in a typical programming language. Rather, *for()* in RASON simply describes the indices and their index sets as attributes to the array formula but it *does not* determine the current value of the indices. (Recall that in the RASON modeling language *for()* is not an iterator.) Though *"for()"* assumes the possible evaluation of the formula for all index combinations, the nature of the model may request only a few or even one combination/setting of indices to evaluate. Evaluation of the model is not sequential, but rather, tracks the dependencies of the function beginning with the top formula then tracking down to the variables. The **key** in an indexed

array formula evaluation is that the dependency is tracked not only by the nodes with formulas, but also by the desired combination of indices attached to that node.

Our example above is very simple; the indexed array formula is independent from any other formulas. Since this indexed array contains an upper bound, we can assume these three lines of code appear in the *constraints* section. We have defined multiple constraints here; one constraint for each p in 'part'. A *for()* definition may also appear in the *formulas*: section. In the above example, if we want to evaluate the constraint related to the "tube" element within the "part" index set, we set p = "tube" and execute the formula only for that particular index. The additional components in the array formula will be ignored.

Introducing indices in array formulas allows for more advanced evaluation algorithms. The following example demonstrates dependency of a given array component on its previous component.

```
formulas: {
  "for(t in 2..time)": {
    "coordX[t]": { dimensions: ['time'], value: 1.570796, formula:
    "coordX[t-1] + change[t]" }
  }
}
constraints: {
  cx: { formula: "abs(coordX[time] - TargetX)", upper: 10 }
}
```

In this example code, the *cx* constraint will be evaluated first. This constraint depends exclusively on the last component of *{coordX[10]}*. (In this example, time = 10.) As a result, RASON starts tracking the dependencies starting with *{coordX[10]}*. However during that evaluation we find another dependency on *{coordX[9]}*. Likewise when evaluating *{coordX[9]}*, we find a dependency on *{coordX[8]}*. This dependency tracking continues until *{coordX[1]}* is reached. Once *{coordX[1]}* is evaluated, we can evaluate *{coordX[2]}*, which allows us to evaluate *{coordX[3]}* and so on until we reach *{coordX[10]}* which ultimately results in the computation of the constraint *cx*.

Indexed array formulas (advanced)

In this section, more advanced constructions with indexed array formulas are introduced.

The SUM Aggregate Function

Currently, the RASON modeling language supports one aggregation statement which may be used to summarize a formula, the sum formula. Similar to *for()*, *sum()* is a statement that behaves not as a programming language loop, but rather as an attribute to its assigned formula. The *sum()* function computes a formula for a given set of indices.

```
sum(p in 'POT', g in 'GRADE'): {
  cost: { formula: "AMT[p, g] * Prem_Cost[g]" }
}
```

The formula for *cost* will be computed for each index and the sum of all resultant products will be calculated.

Note: The RASON language *sum()* statement described above should not be confused with the *sum()* function that may exist within a formula.

In this example code, AMT is an array, indexed by p and g and Prem_Cost is an array, indexed by g. If AMT[p,g] is a 2x2 matrix

```

1  1
2  2

```

and *Prem_Cost[g]* is a 2 x 1 vertical array

```

6
9

```

then *PremiumCost* will equal 45 ($1 * 6 + 1 * 9 + 2 * 6 + 2 * 9$).

Applying a Common *for()* Statement to Multiple Formulas

Another useful application of the *for()* definition is applying a common *for()* to multiple formulas. The following two examples are identical.

Ex1.

```

"for(t in 2..time)" : {
  "coordX[t]" : {dimensions: ['time'], formula: "coordX[t-1] +
Vx[t]" }
}
"for(t in 2..time)" : {
  "coordY[t]": {dimensions: ['time'], formula: "coordY[t-1] +
Vy[t]" }
}

```

Ex2.

```

"for(t in 2..time)" : {
  "coordX[t]": {dimensions: ['time'], formula: "coordX[t-1] +
Vx[t]" }
  "coordY[t]": {dimensions: ['time'], formula: "coordY[t-1] +
Vy[t]" }
}

```

Note: It is also possible to **bind** data to external sources using indexed array formulas. See the section on Binding for more information.

Arrays and tables in indexed array formulas

Currently the RASON modeling language supports arrays with up to 2 dimensions. If you require an array with more than 2 dimensions, then you must create a table. There are many advantages to using tables:

- Tables may hold more than 2 dimensions
- A table result may be used in an indexed array formula
- A table can be sparse while an array is dense.
- The evaluation of a table is less expensive (time consuming) than the evaluation of an array with more than 2 dimensions.

Below is an example of how to assign an indexed array formula to a table.

```

"for(p in 'POT', c in 'CRUCE', g in 'GRADE')" : {
  "Amount[p, c, g]": { formula: "SumTap[p, c, g] - AMT[p, c, g]",
equal: 0 }
}

```

```
}
```

The example code below illustrates how to assign an indexed array formula to an array. In this example, 'time' is the index set { 1, 2, 3, ..., 10 }.

```
"for(t in 'time')": {  
    "dist[t]": { dimensions: ['time'], formula: "sqrt(Vx[t]^2 +  
Vy[t]^2)" }  
}
```

Due to the presence of the dimensions property in the object above, the result, *dist[t]*, will be an array. This is *only* possible if the array has no more than 2 dimensions. Even if the index set is non-numeric, *dist[t]* will still result in an array as long as the *dimensions* property exists in the object definition.

Recall that we must refer to an array element by indices, while we refer to a table element using the names of index set components. For example, to refer to the 2nd element in the *dist[]* array, we would use *dist[2]*. To avoid confusion, it is permissible to use indexed array formulas with arrays that do not involve any index sets. The above example can be re-written as:

```
"for(t in 1..time)": {  
    "dist[t]": { dimensions: ['time'], formula: "sqrt(Vx[t]^2 +  
Vy[t]^2)" }  
}
```

where time is either the constant 10 or an index set/index column with size 10.

Examples with Indexed Array Formulas

Below you will find three examples of indexed array formulas. Each of these snippets of code may be found in the corresponding example model displayed, in brackets, beside or beneath each example definition. These example models can be opened on RASON.com on the "Editor" page by clicking the down arrow next to "Select One" (underneath "RASON Examples) or by opening the file from within RASONIDE.exe by clicking File/Open and browsing to C:\Program Files\Frontline Systems\Solver SDK Platform\Examples\RASON.

A simple 'for()' [Example Model: ProductMix5.json]

```
variables: {  
    x: {  
        lower: [0, 0, 0],  
        finalValue: []  
    },  
    data: {  
        profits: {  
            dimensions: [3],  
            value: [75, 50, 35],  
            binding: "get",  
            finalValue: []  
        },  
        parts: {  
            dimensions: [5, 3],  
            value: [  
                [1, 1, 0],  
                [1, 0, 0],  
                [2, 2, 1],
```

```

        [1, 1, 0],
        [2, 1, 1]
    ]
},

inventory: {
    value: [450, 250, 800, 450, 600]
}
},

constraints: {
    "for(p in 1..5)": {
        "cons[p]": { formula: "sumproduct(parts[p,], transpose(x))", upper:
'inventory' }
    }
}

```

In the above example we define 5 constraints in the vertical array *cons[]*. *Cons[]* is defined as an array (rather than a table), by the implicit vertical array upper bound *inventory*. Each constraint has its own unique index *p*. To evaluate the first constraint, set *p* = 1 and compute the formula for that *p*. (Note: This evaluation is not an array-formula evaluation.) Notice the transposition of *x* from a vertical matrix to a horizontal matrix in order to match the horizontal array slice *parts[p,]*.

A simple 'for()' with index set and array assignment

[Example Model: ProductMixTab3.json]

```

indexSets : {
    part: { value: ['chas','tube','cone','psup','elec'] },
    prod: { value: ['tv','stereo','speaker'] }
},
data : {
    parts : { indexCols: ['part', 'prod'],
        value: [['chas', 'tv', 1],
            ['elec', 'stereo', 1],
            ['tube', 'tv', 1],
            ['cone', 'tv', 2],
            ['cone', 'stereo', 2],
            ['chas', 'stereo', 1],
            ['cone', 'speaker', 1],
            ['psup', 'tv', 1],
            ['psup', 'stereo', 1],
            ['elec', 'tv', 2],
            ['elec', 'speaker', 1]]
    },
    inventory: { dimensions: ['part'], value: [450, 250, 800, 450,
600] }
},
constraints: {
    "for(p in 'part')": {
        "cons[p]": { formula: "sumproduct(parts[p,], x)", upper:
'inventory' }
    }
}

```



```
}
```

In the above example we again define 5 constraints in the vertical array *cons[5]*. As in the example above, *cons[5]* is defined as an array (rather than a table), by the implicit vertical array upper bound *inventory*. However, this time the index *p* belongs to an index set. Notice that *x* is not transposed here. This is because *parts* is a table, where slice *parts[p,]* is always a vertical array.

A simple 'for()' with index set and table assignment

[Example Model: ProductMixTab4.json]

```
indexSets : {
  part", value: ['chas','tube','cone','psup','elec'] },
  { name: "prod", value: ['tv','stereo','speaker'] }
},

data : {
  parts: { indexCols: ['part', 'prod'],

    value: [['chas', 'tv', 1],
             ['elec', 'stereo', 1],
             ['tube', 'tv', 1],
             ['cone', 'tv', 2],
             ['cone', 'stereo', 2],
             ['chas', 'stereo', 1],
             ['cone', 'speaker', 1],
             ['psup', 'tv', 1],
             ['psup', 'stereo', 1],
             ['elec', 'tv', 2],
             ['elec', 'speaker', 1]]

    },
  inventory: { indexCols: ['part'], value: [['chas',450],
['tube',250],

  ['cone',800], ['psup',450], ['elec',600]] }
},
constraints: {
  "for(p in `part`)": {

    "cons[p]": { formula: "sumproduct(parts[p,], x) -
inventory[p]",

    upper: 0 }
  }
}
```

In this example we define 5 constraints for each *p* in the index set '*part*', then assign the evaluation values to *cons[5]*. Within constraints, *cons[5]* is defined as a *table* since neither the *dimensions* property nor the *upper/lower/value* properties are used to implicitly define *cons[5]* as an array. The variable *p* belongs to an index set. To reference a table element, we would use, say, *cons['chas']* rather than *cons[1]* as the latter is a typical array reference. In practice, when defining model functions with index sets, the formula result identifier is likely a table.

Using For Loops

It is imperative that a user understand the difference between *for()* and *loop()* when using the RASON modeling language. *For()* is an attribute to each formula in its block. Only the determined formula (as a precedent in the dependency chain) and only a specific index combination is computed by *for()*.

Conversely, *loop()* behaves like a typical programming language loop. All formulas must execute in the listed sequence for a given index combination. Indices are iterated as they appear as index columns in a table. The rightmost (the last in the list) index changes more rapidly. The leftmost (the first in the list) index changes less rapidly. When using a *loop()* statement, many dependencies are resolved at parse time which results in an easier solving model.

In the example below, there is only one index *t*, which is iterated from 2 to 10. For each index value, all formulas in the loop block are evaluated in the sequence as they are entered.

```
formulas: {
  "loop(t in 2..time)" : {
    "direction[t]": { formula: "direction[t-1] + change[t]" },

    "Vx[t]": { formula: "Vx[t-1] + COS(direction[t])*blast[t]
+ x1grav[t-1] + x2grav[t-1]" },
    ...
  }
}
```

Examples using Loops

Below you will find two examples using loops . Each of these snippets of code may be found in the corresponding example model displayed, in brackets, beside or beneath each example definition. These example models can be opened on RASON.com on the "Editor" page by clicking the down arrow next to "Select One" (underneath "RASON Examples") or by opening the file from within RASONIDE.exe by clicking File/Open and browsing to C:\Program Files\Frontline Systems\Solver SDK Platform\Examples\RASON.

Note: Space1.json is not a completely scalable model. However, Space2.json, which reads data from an external source through data binding, is completely scalable.

A simple 'loop()' without index sets and array assignment

[Example Model: Space1.json]

```
data: {
  time: { value: 10 },
  direction: { dimensions: ['time'], value: 1.570796327 }
},
variables: {
  blast: { dimensions: ['time'], value: 20, lower: 0, upper: 30,
    finalValue: [] },

  change: { dimensions: ['time'], value: 5, lower: -6.28, upper:
6.28,
    finalValue: [] }
},
formulas: {
  "loop(t in 2..time)" : {
    "direction[t]": { formula: "direction[t-1] + change[t]" },
```

```

    "Vx[t]": { formula: "Vx[t-1] + COS(direction[t])*blast[t] +
xlgrav[t-1] + x2grav[t-1]" },
...
  }
}

```

In this example ‘time’ is the constant 10. We preliminarily define the formula assignments as arrays and initialize them with values. We cannot use the index ranges in ‘*loop()/for()*’ for dimensioning the result variables because the ranges determine the desired computation formulas only. For example, in this case there is no formula for $t = 1$, while `direction[1]` must exist in the model.

A fully scalable model with ‘loop()’

[Example Model: Space2.json; Space.txt must also be opened]

```

datasources: {
sun_data: { type: "csv", connection: "Space.txt", indexCols:
['time'], valueCols: ['sun1x', 'sun1y', 'sun2x', 'sun2y'],
direction: "import"
    }
},

formulas: {
"loop(t in 2..time)" : {
"direction[t]": { dimensions: ['time'], value: 1.570796327, formula:
"direction[t-1] + change[t]" },

"Vx[t]": { dimensions: ['time'], value: 0, formula: "Vx[t-1] +
COS(direction[t])*blast[t] + xlgrav[t-1] x2grav[t-1]" },
...
}
}

```

In this example, the dimensioning and initialization of arrays is performed inside the loop. The difference between this example and example 4 is that here an index set is determined through a data source binding. (All arrays with constants are put in an external source of type CSV and imported into the model through binding. For more information, see the *Data* section within the *RASON Reference Guide*.)

Notice that *loop()* is used in the *formulas* section and does not define any constraints. If, when evaluating a given constraint, the RASON Interpreter finds that a given part of a constraint formula depends on another formula, the whole loop must execute in sequence in order to determine the needed precedent.

Note: Do not nest a “for” definition because it is not a statement but rather an attribute to an identifier with a formula; use a new “loop” instead. Generally, one may nest all statements plus the aggregation definitions such as “sum”.

Using Statements

If-Else, If-Then-Else, While and Compound statements are also supported in the RASON modeling language. See below for syntax and useage examples of each.

If-Else and If-Then-Else Statement Examples

The else part in both IF statements above is optional. Notice that both If statements are the only place in RASON where two formulas may be assigned to the same identifier.

If – Else Example

```
"if (logic_exp)" : {  
    f: {...},  
    ...  
},  
"else": {  
    f: {...}  
    ...  
}
```

If – Then – Else Example

```
F: { if: "logic_exp", then: "formula1", else: "formula2" }
```

While Example

This example illustrates how to use a "while" statement in the RASON modeling language.

```
"while ( logic_exp)": {  
    ...  
}
```

Compound Example

The "body" in all statements is an implicit compound statement – a sequence of definitions and nested statements. The only case when we may need an explicit compound statement, is when initializing an identifier before using it in a next loop statement as shown below.

```
"compound" : {  
    i: {value: 1},  
    ...  
}
```

Using Tables

Recall from the earlier Array Formulas section that arrays in the RASON Modeling language are limited to 2-dimensions. If you require an array containing more than 2-dimensions, a table may be created. There are many advantages to using tables:

- Tables may hold more than 2 dimensions
- A table result may be used in an indexed array formula
- A table can be sparse while an array is dense.
- The evaluation of a table is less expensive (time consuming) than the evaluation of an array with more than 2 dimensions.

The following matrix of data displays the number of each part used in the manufacturing of three products: TVs, stereos and speakers.

Parts	TV	Stereo	Speaker
-------	----	--------	---------

Chassis	1	1	0
LCD Screens	1	0	0
Speaker Cones	2	2	1
Power Supply	1	1	0
Electronics	2	1	1

In the example code below, the matrix above has been specified "in line" (within the RASON model) as the *parts* table. This table contains 15 elements, the same as the matrix above, and is indexed using two dimensions: part and prod. (To open this full example, and any example referenced in this guide, click the down arrow under RASON examples on the Editor page of www.RASON.com or browse to C:\Program Files\Frontline Systems\Solver SDK Platform\Examples\RASON if using the RASON IDE.)

```
data: {
  parts: { indexCols: ['part', 'prod'],
    value: [
      ['chas', 'tv', 1], ['chas', 'stereo', 1], ['chas', 'speaker', 0],
      ['lcds', 'tv', 1], ['lcds', 'stereo', 0], ['lcds', 'speaker', 0],
      ['cone', 'tv', 2], ['cone', 'stereo', 2], ['cone', 'speaker', 1],
      ['psup', 'tv', 1], ['psup', 'stereo', 1], ['psup', 'speaker', 0],
      ['elec', 'tv', 2], ['elec', 'stereo', 1], ['elec', 'speaker', 1]]
    },
  variables : {
    x: {dimensions: ['prod'], value: 0, lower: 0, finalValue: [] },
  },
  constraints : {
    c1: {formula: "sumproduct(parts['chas'], x)", upper: 450 },
    c2: {formula: "sumproduct(parts['tube'], x)", upper: 250 },
    c3: {formula: "sumproduct(parts['cone'], x)", upper: 800 },
    c4: {formula: "sumproduct(parts['psup'], x)", upper: 450 },
    c5: {formula: "sumproduct(parts['elec'], x)", upper: 600
  }},
  objective : {
    total: {formula: "sumproduct(x, profits)", type: "maximize",
    finalValue: [] }
  }
}
```

Notice that even though the *parts* table is listed horizontally in the RASON model, the table elements are stored in a vertical array.

In the *variables* section, the x array, dimensioned using the index column *prod*, is an array of size 3 containing our decision values x[1], x[2] and x[3]. These are the number of each product to produce.

In the *constraints* section, five constraints are formulated by "slicing" the table to obtain the correct part data. For example, in the first constraint, `c1, parts['chas',]` slices the table across the part *chas* which results in the elements: 1, 1, 0. The second constraint, `c2`, slices the table across *tube* obtaining the elements 1, 0, 0. Constraints `c3`, `c4` and `c5` are calculated in the same fashion.

Since tables are sparse, as opposed to arrays which are dense, we can eliminate the table elements that are not non-zero. This leaves the following table that is still indexed with two dimensions, *part* and *prod*, but now only contains 11 elements. (See the example model `RGProductMixTab1.json` to view the complete model.)

Note: In the current release of the RASON Modeling Language, the overall internal structure of the table is dense, the missing records are considered 0's.

```
parts: {indexCols: ['part', 'prod'],
```

```
    value:
      [['chas', 'tv', 1],
       ['chas', 'stereo', 1],
       ['tube', 'tv', 1],
       ['cone', 'tv', 2],
       ['cone', 'stereo', 2],
       ['chas', 'stereo', 1],
       ['cone', 'speaker', 1],
       ['psup', 'tv', 1],
       ['psup', 'stereo', 1],
       ['elec', 'tv', 2],
       ['elec', 'stereo', 1],
       ['elec', 'speaker', 1]]
```

```
  },
```

Note that the order in which the table elements are entered is irrelevant since we are supplying both dimensions for each value. We could have also entered the table as:

```
parts: {indexCols: ['part', 'prod'],
```

```
    value:
      [['chas', 'tv', 1],
       ['tube', 'tv', 1],
       ['cone', 'tv', 2],
       ['psup', 'tv', 1],
       ['elec', 'tv', 2],
       ['chas', 'stereo', 1],
       ['cone', 'stereo', 2],
       ['psup', 'stereo', 1],
       ['elec', 'stereo', 1],
       ['cone', 'speaker', 1],
```

```

        ['elec', 'speaker', 1]]
    },

```

The ProductMixTab2.json example model illustrates how to use an index set to dimension a table. RASON uses index sets exclusively to dimension tables and arrays. Typical mathematical programming models include multiple tables and arrays indexed over various index sets. An index set should be created at the beginning of the model to establish a basis of order for each dimension appearing in a table or array. Otherwise, the user will be required to keep track of and maintain the correct order of elements in all arrays and tables present in the model. An index set is always a 1-dimensional array and *must* be defined within the *indexSets* section of the RASON model.

The example code below creates two ordered sets, *part* and *prod*. The *part* set contains five items (in order as entered): chas, tube, cone, psup and elec while the *prod* set contains 3 items: tv, stereo and speakers. For more information on index sets, see the *Index Sets* section within the *RASON Reference Guide*.

```

indexSets: {
    part: {
        value: ['chas', 'tube', 'cone', 'psup', 'elec']
    },
    prod: {
        value: ['tv', 'stereo', 'speaker']
    }
},

data: {
    parts: {
        indexCols: ['part', 'prod'],
        value: [

            ['chas', 'tv', 1],
            ['chas', 'stereo', 1],
            ['tube', 'tv', 1],
            ['cone', 'tv', 2],
            ['cone', 'stereo', 2],
            ['cone', 'speaker', 1],
            ['psup', 'tv', 1],
            ['psup', 'stereo', 1],
            ['elec', 'tv', 2],
            ['elec', 'stereo', 1],
            ['elec', 'speaker', 1]

        ]
    },
    profits: {

```

```

        dimensions: ['prod'],
        value: [75, 50, 35]
    },

    inventory: {dimensions: ['part'], value: [450, 250, 800, 450, 600],
    binding: "get", finalValue: [] }
}

formulas : {
    piv_parts: {formula: "PIVOT(parts, { 'prod' }, { 'part' })" }
},

constraints : {
    c: {dimensions: ['part'], formula: "MMULT(piv_parts, x) -
inventory", upper: 0 }
},

```

In the *data* section, an inline table object, *parts*, is created containing two index columns (*parts* and *prods*) and a value column. Since the index set *prod* exists, we can dimension the profits array according to this set in order to assign the correct profit values to the appropriate products. Likewise, we can dimension the inventory array according to the *part* set in order to assign the correct inventory level to each part.

Notice the use of the PIVOT function in the *piv_parts* formula. Recall that a table is stored as a vertical array. The PIVOT function returns a 2-dimensional table (or a 1- dimensional table) of values with rows and columns assigned by the index columns in the table passed as the first argument. (The first argument passes the data.) The second argument is the index column(s) to be assigned as column(s) in the PIVOT table. (These are the dimensions that will make up the columns of the table.) The third argument is the index column(s) to be assigned as row(s) in the PIVOT table. (These are the dimensions that will make up the rows of the table.) In this example, our table *parts* contains two index columns, *parts* and *prods*. The PIVOT function creates a 2-dimensional array of values by assigning the *prod* dimension (TV, Stereos and Speakers) as columns and the *part* dimension(chas, cone, elec, psup and tube) as rows. The result is a 5 x 3 matrix.

Since *piv_parts* is now a 2-dimensional table, we can perform matrix multiplication using the MMULT (inherited from Excel) function to multiply this table by the variable array, *x* (which is also a vertical array). The result of this operation is a vertical array of size 5 which must be less than the inventory array (enforced by the *upper* property of 0).

If the *sortIndexCols* (or *sort*) property is used, all *indexCols* will be sorted alphabetically. (Note: The properties *sort* and *sortIndexCols* perform the same function.) Otherwise, the table will be sorted as entered. In the example below, the order for the *prod* indexCol will be: speaker, stereo, tv. While the order for the *part* indexCol will be: chas, cone, elec, psup, tube.

```

.
data : {
    parts: {indexCols: ['part', 'prod'], sortIndexCols: true,
        value: [
            ['chas', 'tv', 1], ['chas', 'stereo', 1], ['chas', 'speaker', 0],
            ['tube', 'tv', 1], ['tube', 'stereo', 0], ['tube', 'speaker', 0],
            ['cone', 'tv', 2], ['cone', 'stereo', 2], ['cone', 'speaker', 1],
            ['psup', 'tv', 1], ['psup', 'stereo', 1], ['psup', 'speaker', 0],
            ['elec', 'tv', 2], ['elec', 'stereo', 1], ['elec', 'speaker', 1]]
    }
}

```



```
},
```

The table below gives a recap on how table elements may be referenced. See the list below for syntax in obtaining coefficients for dimensions and dimension slices in a table created using an index set.

Formula Syntax	Action
<code>parts['cone',]</code>	Retrieves coefficients for all products containing the part, cone. The result is a vertical array.
<code>parts[, 'tv']</code>	Retrieves coefficients for all parts used to build a TV. The result is a vertical array.
<code>['cone', 'stereo']</code>	Retrieves the coefficient for part = 'cone' and prod = 'stereo'.

Creating a Sparse Table

Most large LP models are *sparse* in nature: While they may include thousands of decision variables and constraints, the typical constraint will depend upon only a few of the variables. In RASON, this sparsity can be exploited to save memory and gain speed when solving by setting "sparse" : True within the modelSettings section of the RASON model.

Likewise, most large cubes are *sparse* in nature. While they may contain thousands of elements, in practice, not all combinations of dimension elements are possible; therefore, not all will define a model function during the Psi Interpreter's evaluation of the problem. This means that most cubes will provoke output results as sparse cubes (with missing constraints). Such sparsity in a cube (also known as structural sparsity) can be exploited in RASON to save memory and gain speed by setting "sparseCubes": True in the modelSettings section of the RASON model.

RASON also includes the ability to exploit sparsity in a table, i.e. if a constraint formula depends on a few records within a table (rather than the full cube). This sparsity will be exploited without the need to set a model setting option.

The example RASON model below is dense transportation optimization model. In this example, a company produces 3 *products* in 2 factories and must ship to 4 warehouses and/or 5 customers. The company can ship products from the factories to the warehouses, the factories to the customers, or from the warehouses to the customers. The number of products received by a warehouse should be the same as the number of products shipped from the warehouse. The objective is to minimize the total shipping cost while meeting demand and not exceeding supply. All dimension elements and record values for all combinations have been entered in each Table.

```
{
comment: "Transport3 model(dense)",

engineSettings: { engine: "LP/Quadratic" },

indexSets : {
  factories: { value: ['f1','f2'] },
  products: { value: ['p1','p2','p3'] },
  warehouses: { value: ['w1','w2','w3','w4'] },
  customers: { value: ['c1','c2','c3','c4','c5'] }
},

variables: {
  f_w_num: { indexCols: ['warehouses','products','factories'],
value: 0, lower: 0, finalValue: [] },
```

```

    f_c_num: { indexCols: ['customers','products','factories'],
value: 0, lower: 0, finalValue: [] },
    w_c_num: { indexCols: ['customers','products','warehouses'],
value: 0, lower: 0, finalValue: [] }
},

data: {
    demand: { indexCols: ['customers','products'],
value: [30000, 23000, 15000, 32000, 16000,
20000, 15000, 22000, 12000, 18000,
25000, 22000, 16000, 20000, 25000] },
    cap_f: { indexCols: ['products','factories'],
value: [90000, 100000, 80000,
75000, 65000, 90000] },
    cap_w: { indexCols: ['warehouses','products'],
value: [35000, 20000, 30000, 15000,
30000, 25000, 15000, 24000,
20000, 20000, 25000, 20000] },
    f_w_cost: { indexCols: ['warehouses','products','factories'],
value: [0.50, 0.50, 1.00, 0.20,
1.00, 0.75, 1.25, 1.25,
0.75, 1.25, 1.00, 0.80,
1.50, 0.30, 0.50, 0.20,
1.25, 0.80, 1.00, 0.75,
1.40, 0.90, 0.95, 1.10] },
    f_c_cost: { indexCols: ['customers','products','factories'],
value: [2.75, 3.50, 2.50, 3.00, 2.50,
2.50, 3.00, 2.00, 2.75, 2.60,
2.90, 3.00, 2.25, 2.80, 2.35,
3.00, 3.50, 3.50, 2.50, 2.00,
2.25, 2.95, 2.20, 2.50, 2.10,
2.45, 2.75, 2.35, 2.85, 2.45] },
    w_c_cost: { indexCols: ['customers','products','warehouses'],
value: [1.50, 0.80, 0.50, 1.50, 3.00,
1.00, 0.90, 1.20, 1.30, 2.10,
1.25, 0.70, 1.10, 0.80, 1.60,
1.00, 0.50, 0.50, 1.00, 0.50,
1.25, 1.00, 1.00, 0.90, 1.50,
1.10, 1.10, 0.90, 1.40, 1.75,
1.00, 1.50, 2.00, 2.00, 0.50,
0.90, 1.35, 1.45, 1.80, 1.00,
1.25, 1.20, 1.75, 1.70, 0.85,
2.50, 1.50, 0.60, 1.50, 0.50,
1.75, 1.30, 0.70, 1.25, 1.10,
1.50, 1.10, 1.50, 1.10, 0.90] }
},

formulas: {

    "sum(w in warehouses, p in products, f in factories)": {
        s1: { formula: "f_w_cost[w,p,f] * f_w_num[w,p,f]", finalValue:
[] }
    },

    "sum(c in customers, p in products, f in factories)": {

```

```

        s2: { formula: "f_c_cost[c,p,f] * f_c_num[c,p,f]", finalValue:
[] }
    },

    "sum(c in customers, p in products, w in warehouses)": {
        s3: { formula: "w_c_cost[c,p,w] * w_c_num[c,p,w]", finalValue:
[] }
    },

    s11: { formula: "sumproduct(f_w_cost[, ,], f_w_num[, ,])",
finalValue:
[] },

    s22: { formula: "sumproduct(f_c_cost[, ,], f_c_num[, ,])",
finalValue:
[] },

    s33: { formula: "sumproduct(w_c_cost[, ,], w_c_num[, ,])",
finalValue:
[] }
},

constraints: {
    "for(c in customers, p in products)": {
        d_con: { formula: "sum(w_c_num[c,p,]) + sum(f_c_num[c,p,]) -
demand[c,p]", lower: 0, finalValue: [] }
    },

    "for(p in products, f in factories)": {
        cf_con: { formula: "sum(f_w_num[,p,f]) + sum(f_c_num[,p,f]) -
cap_f[p,f]", upper: 0, finalValue: [] }
    },

    "for(w in warehouses, p in products)": {
        cw_con: { formula: "sum(f_w_num[w,p,]) - cap_w[w,p]", upper:
0,
        finalValue: [] }
    },

    "for(w in warehouses, p in products)": {
        w_con: { formula: "sum(f_w_num[w,p,]) - sum(w_c_num[,p,w])",
equal:
0, finalValue: [] }
    }
},

objective: {
    total: { type: "min", formula: "s11 + s22 + s3" }
}

```

Constraints ensure that demand is met and capacity is not exceeded

Objective minimizes shipping costs

In the example code below, you'll find the changed variables, data and constraints sections. Notice that only a few combinations of index elements are present in all tables. For example within the data section, `cap_f` and `cap_w` demonstrate that Factory 1 exclusively produces product 1 and delivers it to warehouse 1 ([`'p1'`, `'f1'`, 90000] & [`'w1'`, `'p1'`, 35000]). Factory 2 produces only products 2 and 3 and delivers them only to Warehouse 1 ([`'p2'`, `'f1'`, 65000] & [`'w2'`, `'p2'`, 25000]).

Constraints are defined using a "for" construction. The second part of this construction uses the `exist (indexed_table) RASON` function. This function evaluates to a Boolean value. If no record exists in the `indexed_table` for a given index, the "for()" definition will not generate a constraint.

Note: This function is able to filter constraints through any logic (not only the existence of records). For example, a logical expression may be written to consider only array elements \geq a given constraint. This logic conditional operator is allowed only within "for()" definitions in the `constraints` or `uncertainFunctions` sections.

```
...
data: {
  demand: { indexCols: ['customers','products'],
    value: [['c1','p1',30000],
      ['c2','p2',15000],
      ['c3','p3',16000],
      ['c4','p3',20000],
      ['c5','p3',25000]]
  },
  cap_f: { indexCols: ['products','factories'],
    value: [['p1','f1',90000],
      ['p2','f2',65000],
      ['p3','f2',90000]]
  },
  cap_w: { indexCols: ['warehouses','products'],
    value: [['w1','p1',35000],
      ['w2','p2',25000],
      ['w3','p3',25000],
      ['w4','p3',20000]]
  },
  f_w_cost: { indexCols: ['warehouses','products','factories'],
    value: [['w1','p1','f1',0.50],
      ['w2','p2','f2',0.80],
      ['w3','p3','f2',0.95],
      ['w4','p3','f2',1.10]]
  },
  f_c_cost: { indexCols: ['customers','products','factories'],
    value: [['c1','p1','f1',2.75],
      ['c2','p2','f2',2.95],
      ['c3','p3','f2',2.35],
      ['c4','p3','f2',2.85],
      ['c5','p3','f2',2.45]]
  },
  w_c_cost: { indexCols: ['customers','products','warehouses'],
    value: [['c1','p1','w1',1.50],
      ['c2','p2','w2',1.00],
      ['c3','p3','w3',1.75],
      ['c4','p3','w4',1.10],
      ['c5','p3','w4',0.90]]
  }
},
...
constraints: {
  "for(c in customers, p in products | exist(demand[c,p]))":
  {
    d_con: { formula: "sum(w_c_num[c,p,]) + sum(f_c_num[c,p,]) -
      demand[c,p]", lower: 0, finalValue: [] }
  },

```

```

    "for(p in products, f in factories | exist(cap_f[p,f]))":
    {
      cf_con: { formula: "sum(f_w_num[,p,f]) + sum(f_c_num[,p,f]) -
        cap_f[p,f]", upper: 0, finalValue: [] }
    },
    "for(w in warehouses, p in products | exist(cap_w[w,p]))":
    {
      cw_con: { formula: "sum(f_w_num[w,p,]) - cap_w[w,p]", upper:
0,
      finalValue: [] }
    },
    "for(w in warehouses, p in products)":
    {
      w_con: { formula: "sum(f_w_num[w,p,]) - sum(w_c_num[,p,w])",
equal:
      0, finalValue: [] }
    }
  },
  ...

```

This model can be improved further by separating the data in an external data source as shown in the example below in datasources.

```

{
  comment: "Transport3 model (sparse)",
  engineSettings: { engine: "LP/Quadratic" },
  indexSets : {
    factories: { value: ['f1','f2'] },
    products: { value: ['p1','p2','p3'] },
    warehouses: { value: ['w1','w2','w3','w4'] },
    customers: { value: ['c1','c2','c3','c4','c5'] }
  },
  datasources :
  {
    f_w_data: {
      type: "excel",
      connection: "Logistic3.xlsx",
      selection: "Transport3!A2:E5",
      indexCols: ['warehouses','products','factories'],
      valueCols: ['f_w_cost','f_w_num'],
      direction: "import"
    },
    f_c_data: {
      type: "excel",
      connection: "Logistic3.xlsx",
      selection: "Transport3!A9:E13",
      indexCols: ['customers','products','factories'],
      valueCols: ['f_c_cost','f_c_num']
      direction: "import"
    },
    w_c_data: {
      type: "excel",
      connection: "Logistic3.xlsx",
      selection: "Transport3!A17:E21",
      indexCols: ['customers','products','warehouses'],
      valueCols: ['w_c_cost','w_c_num']
      direction: "import"
    }
  }
}

```

```

    },
    demand_data: {
      type: "excel",
      connection: "Logistic3.xlsx",
      selection: "Transport3!H17:J21",
      indexCols: ['customers', 'products'],
      valueCols: ['demand']
      direction: "import"
    },
    cap_f_data: {
      type: "excel",
      connection: "Logistic3.xlsx",
      selection: "Transport3!H9:J11",
      indexCols: ['products', 'factories'],
      valueCols: ['cap_f']
      direction: "import"
    },
    cap_w_data: {
      type: "excel",
      connection: "Logistic3.xlsx",
      selection: "Transport3!H2:J5",
      indexCols: ['warehouses', 'products'],
      valueCols: ['cap_w']
      direction: "import"
    },
    obj_data: {
      type: "excel",
      connection: "Logistic3.xlsx",
      selection: "Transport3!L2"
      direction: "import"
    }
  },
  variables: {
    f_w_num: { binding: "f_w_data", valueCol: 'f_w_num', lower: 0,
    finalValue: [] },
    f_c_num: { binding: "f_c_data", valueCol: 'f_c_num', lower: 0,
    finalValue: [] },
    w_c_num: { binding: "w_c_data", valueCol: 'w_c_num', lower: 0,
    finalValue: [] }
  },
  data: {
    demand: { binding: "demand_data" },
    cap_f: { binding: "cap_f_data" },
    cap_w: { binding: "cap_w_data" },
    f_w_cost: { binding: "f_w_data", valueCol: "f_w_cost" },
    f_c_cost: { binding: "f_c_data", valueCol: "f_c_cost" },
    w_c_cost: { binding: "w_c_data", valueCol: "w_c_cost" }
  },
  formulas: {
    s1: { formula: "sumproduct(f_w_cost[,], f_w_num[,])" },
    s2: { formula: "sumproduct(f_c_cost[,], f_c_num[,])" },
    s3: { formula: "sumproduct(w_c_cost[,], w_c_num[,])" }
  },
  constraints: {
    "for(c in customers, p in products | exist(demand[c,p]))":
    {

```

```

    d_con: { formula: "sum(w_c_num[c,p,]) + sum(f_c_num[c,p,]) -
demand[c,p]", lower: 0, finalValue: [] }
  },

  "for(p in products, f in factories | exist(cap_f[p,f]))":
  {
    cf_con: { formula: "sum(f_w_num[,p,f]) + sum(f_c_num[,p,f]) -
cap_f[p,f]", upper: 0, finalValue: [] }
  },
  "for(w in warehouses, p in products | exist(cap_w[w,p]))":
  {
    cw_con: { formula: "sum(f_w_num[w,p,]) - cap_w[w,p]", upper:
0,
    finalValue: [] }
  },
  "for(w in warehouses, p in products)":
  {
    w_con: { formula: "sum(f_w_num[w,p,]) - sum(w_c_num[,p,w])",
equal:
    0, finalValue: [] }
  }
},
objective: {
  total: { type: "min", formula: "s1 + s2 + s3", binding:
"obj_data" }
}
}

```

Final results for constraints and variables in sparse tables are only returned for the participating records.

```

{
  "status": {
    "code": 0,
    "codeText": "Solver found a solution. All constraints and
optimality conditions are satisfied."
  },
  "variables": {
    "f_w_num": {
      "indexCols": ['warehouses', 'products', 'factories'],
      "finalvalue": [
        ['w1', 'p1', 'f1', 30000],
        ['w2', 'p2', 'f2', 15000],
        ['w3', 'p3', 'f2', 0],
        ['w4', 'p3', 'f2', 20000]
      ]
    },
    "f_c_num": {
      "indexCols": ['customers', 'products', 'factories'],
      "finalvalue": [
        ['c1', 'p1', 'f1', 0],
        ['c2', 'p2', 'f2', 0],
        ['c3', 'p3', 'f2', 16000],
        ['c4', 'p3', 'f2', 0],
        ['c5', 'p3', 'f2', 25000]
      ]
    },
    "w_c_num": {
      "indexCols": ['customers', 'products', 'warehouses'],

```

```

        "finalvalue": [
            ['c1', 'p1', 'w1', 30000],
            ['c2', 'p2', 'w2', 15000],
            ['c3', 'p3', 'w3', 0],
            ['c4', 'p3', 'w4', 20000],
            ['c5', 'p3', 'w4', 0]
        ]
    },
    },
    "objective": {
        "total": {
            "finalValue": 229850
        }
    },
    "constraints": {
        "d_con": {
            "indexCols": ['customers', 'products'],
            "finalvalue": [
                ['c1', 'p1', 0],
                ['c2', 'p2', 0],
                ['c3', 'p3', 0],
                ['c4', 'p3', 0],
                ['c5', 'p3', 0]
            ]
        },
        "cf_con": {
            "indexCols": ['products', 'factories'],
            "finalvalue": [
                ['p1', 'f1', -60000],
                ['p2', 'f2', -50000],
                ['p3', 'f2', -29000]
            ]
        },
        "cw_con": {
            "indexCols": ['warehouses', 'products'],
            "finalvalue": [
                ['w1', 'p1', -5000],
                ['w2', 'p2', -10000],
                ['w3', 'p3', -25000],
                ['w4', 'p3', 0]
            ]
        },
        "w_con": {
            "indexCols": ['warehouses', 'products'],
            "finalvalue": [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
        }
    }
}

```


RASON Power Apps Tutorial

Calling RASON Services from Power Apps

This tutorial will walk you through how to create a Power Apps application that solves a model using RASON Services and displays the results.

Airline Crew Scheduling Excel Model

Click Help – Example Models – Optimization Examples, scroll down to click the Airline Crew Scheduling link to open the Airline Crew Scheduling example model.

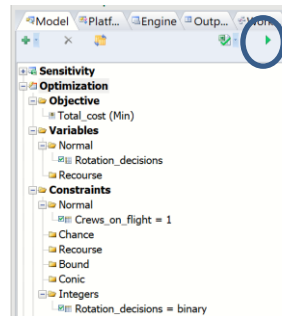
Problem: A small airline maintains a schedule of two daily flights between Salt Lake City, Dallas, and Chicago. A crew that leaves a city in the morning must return to the same city at night but can return on another airline, if needed, on an 8 pm flight. There are 6 airplanes in use. When a crew is flying, the cost is \$200 per hour. When a crew is waiting or being flown back, the cost of the crew is \$75 per hour. How should the company schedule the crews to minimize cost?

	A	B	C	D	E	F	G	H	I
11									
12		✈ = working • = riding		Flying	Other				
13		Routes	Hours	Hours	Cost	Decisions		Number of Crews	
14		Salt Lake ✈ Dallas ✈ Salt Lake	6	2	\$1,350	0	Salt Lake to Dallas Depart	0	
15		Salt Lake ✈ Dallas • Salt Lake	3	11	\$1,425	0	Salt Lake to Dallas Return	0	
16		Salt Lake ✈ Dallas ✈ Chicago • Salt Lake	5	10	\$1,750	0	Salt Lake to Chicago Depart	0	
17		Salt Lake ✈ Chicago • Salt Lake	4	10	\$1,550	0	Salt Lake to Chicago Return	0	
18		Salt Lake ✈ Chicago ✈ Dallas • Salt Lake	6	5	\$1,575	0	Dallas to Salt Lake Depart	0	
19		Dallas ✈ Salt Lake ✈ Dallas	6	3	\$1,425	0	Dallas to Salt Lake Return	0	
20		Dallas ✈ Salt Lake • Dallas	3	12	\$1,500	0	Dallas to Chicago Depart	0	
21		Dallas ✈ Salt Lake ✈ Chicago • Dallas	7	7	\$1,925	0	Dallas to Chicago Return	0	
22		Dallas ✈ Chicago ✈ Salt Lake • Dallas	6	5	\$1,575	0	Chicago to Salt Lake Depart	0	
23		Dallas ✈ Chicago ✈ Dallas	4	5	\$1,175	0	Chicago to Salt Lake Return	0	
24		Chicago ✈ Salt Lake ✈ Dallas • Chicago	7	7	\$1,925	0	Chicago to Dallas Depart	0	
25		Chicago ✈ Salt Lake ✈ Chicago	8	3	\$1,825	0	Chicago to Dallas Return	0	
26		Chicago ✈ Dallas ✈ Chicago	4	3	\$1,025	0			
27		Chicago ✈ Dallas ✈ Salt Lake • Chicago	7	9	\$2,075	0			
28				Total Cost:	\$0				

- Column B lists the 14 different flight schedules that the airline could provide.
- Column C contains the flight hours, column D contains "other" hours spent waiting for flights or commuting.
- Column E contains the cost for each flight schedule.
- The F column holds the decision variables. These are the cells that Solver will change. If a flight is selected, Solver will insert a 1 into the cell next to the corresponding flight. Solver will select the flight schedules that satisfy the constraints in column I while minimizing the total cost to the airline.
- Column I ensures that only 1 crew is assigned per flight. For example, cell I14 ($=F14+F15+F16$) ensures that any flights leaving from Salt Lake and flying to Dallas are manned by 1, and only 1, crew. Likewise, cell I15 ($=F19+F24$) ensures that any flights returning from Salt Lake to Dallas are manned by 1, and only 1, crew.

The Solver Task Pane on the right displays the objective function, cell F28, the decision variables (F14:F27) and the constraints that cells I14:I25 must be equal to 1. Notice that the objective function has been given the defined name of "Total_cost", the decision variables have been given the defined name of "Rotation_decisions" and the constraints have been given the defined name of "Crews_on_flight". The variables are specified as "binary", 0 or 1.

Figure 1: Solver Task Pane



To solve the example, click the green arrow on the Task Pane.

	A	B	C	D	E	F	G	H	I
11									
12		✈ = working • = riding		Flying	Other				
13		Routes	Hours	Hours	Cost	Decisions			Number of Crews
14		Salt Lake ✈ Dallas ✈ Salt Lake	6	2	\$1,350	1		Salt Lake to Dallas Depart	1
15		Salt Lake ✈ Dallas • Salt Lake	3	11	\$1,425	0		Salt Lake to Dallas Return	1
16		Salt Lake ✈ Dallas ✈ Chicago • Salt Lake	5	10	\$1,750	0		Salt Lake to Chicago Depart	1
17		Salt Lake ✈ Chicago • Salt Lake	4	10	\$1,550	0		Salt Lake to Chicago Return	1
18		Salt Lake ✈ Chicago ✈ Dallas • Salt Lake	6	5	\$1,575	1		Dallas to Salt Lake Depart	1
19		Dallas ✈ Salt Lake ✈ Dallas	6	3	\$1,425	1		Dallas to Salt Lake Return	1
20		Dallas ✈ Salt Lake • Dallas	3	12	\$1,500	0		Dallas to Chicago Depart	1
21		Dallas ✈ Salt Lake ✈ Chicago • Dallas	7	7	\$1,925	0		Dallas to Chicago Return	1
22		Dallas ✈ Chicago ✈ Salt Lake • Dallas	6	5	\$1,575	1		Chicago to Salt Lake Depart	1
23		Dallas ✈ Chicago ✈ Dallas	4	5	\$1,175	0		Chicago to Salt Lake Return	1
24		Chicago ✈ Salt Lake ✈ Dallas • Chicago	7	7	\$1,925	0		Chicago to Dallas Depart	1
25		Chicago ✈ Salt Lake ✈ Chicago	8	3	\$1,825	1		Chicago to Dallas Return	1
26		Chicago ✈ Dallas ✈ Chicago	4	3	\$1,025	1			
27		Chicago ✈ Dallas ✈ Salt Lake • Chicago	7	9	\$2,075	0			
28					Total Cost:	\$8,775			

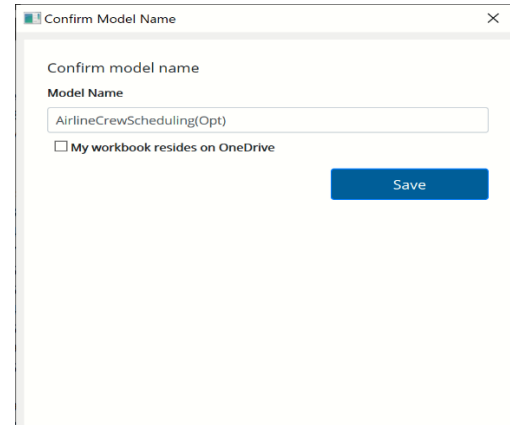
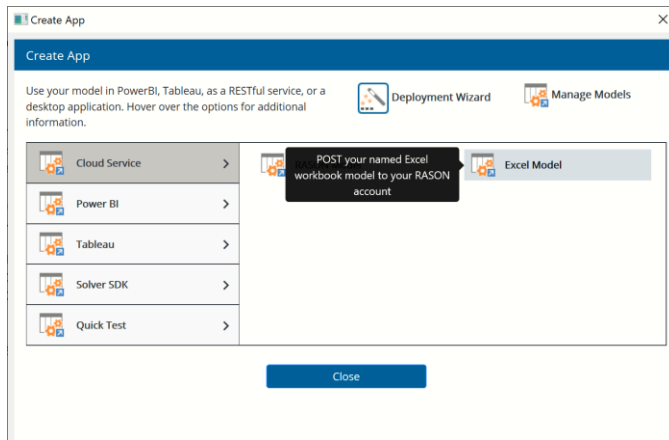
Solver was able to find a solution that satisfies all the constraints. The final solution is to schedule 6 flights: F14 (Salt Lake to Dallas to Salt Lake), F18 (Salt Lake to Chicago, to Dallas to Salt Lake), F19 (Dallas to Salt Lake to Dallas), F22 (Dallas to Chicago to Salt Lake to Dallas), F25 (Chicago to Salt Lake to Chicago) and F26 (Chicago to Dallas to Chicago). The constraints in column I indicate that all required flight legs are manned by 1, and only 1, crew.

Deploying the Excel Model to Rason Cloud Services

Starting with Analytic Solver 2020, Analytic Solver users can now deploy their models onto RASON Cloud Services which allows them to run their model or decision flow remotely.

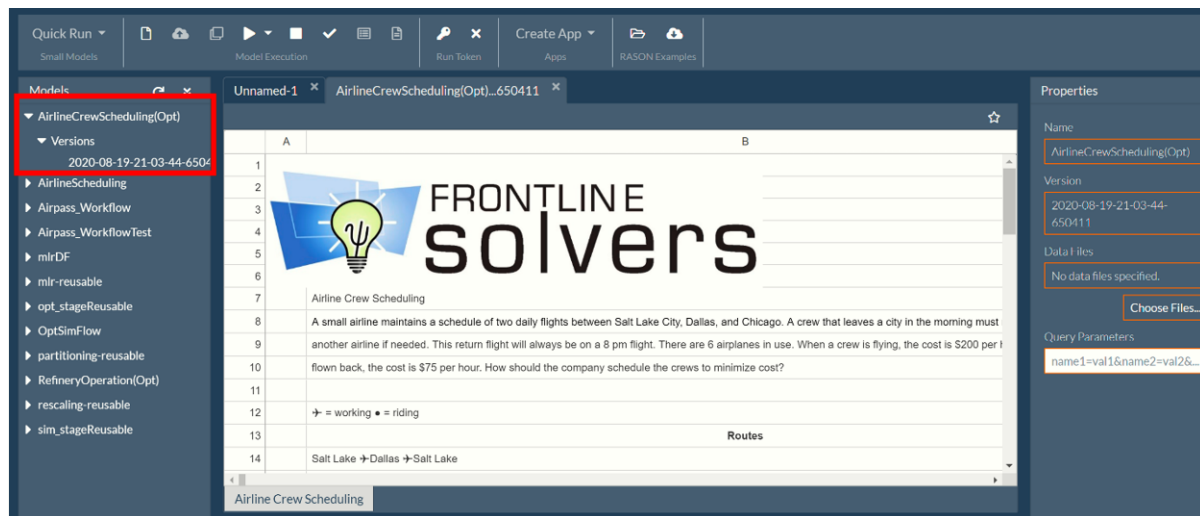
To deploy your model on RASON Decision Services, click Create App – Cloud Service – RASON Model to translate your model into Frontline's high-level RASON modeling language. To deploy the Excel model itself to the RASON Server, click Create App – Cloud Service – Excel Model. To keep it simple, we will deploy the Excel model to the RASON Server by clicking Create App – Cloud Service – Excel Model.

A dialog will open in Excel asking you to confirm the model name. If the model was saved to a OneDrive account, you would check the "My workbook resides on OneDrive" checkbox. Since this model is saved to C:\ProgramData\Frontline Systems\Examples, we can ignore this checkbox and click Save to accept the model name as "AirlineCrewScheduling(Opt)".



Immediately, Frontline's RASON portal, www.rason.com opens displaying the model in the Model Editor window. Notice "AirlineCrewScheduling(Opt)" now appears at the top of the Models list, on the left. Note that if your model were to change in any way, say to add another route to the schedule, or to change the cost of a flight, you would need to change the model in Excel, then use Create App – Cloud Service – Excel Model to deploy the amended model.

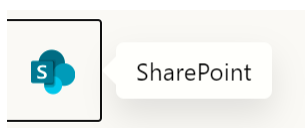
Note: If we would have used Create App – Cloud Service – RASON Model to convert the model to the RASON modeling language, we would have the option of amending the model on www.rason.com without the need to return to Excel.



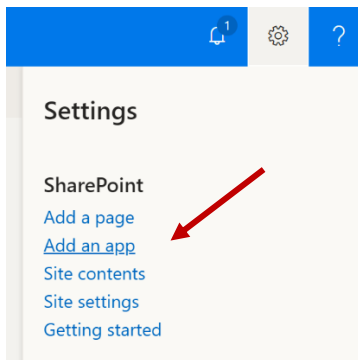
Now that our model is deployed to the Rason Server, we are ready to create our app.

Entering Data in SharePoint

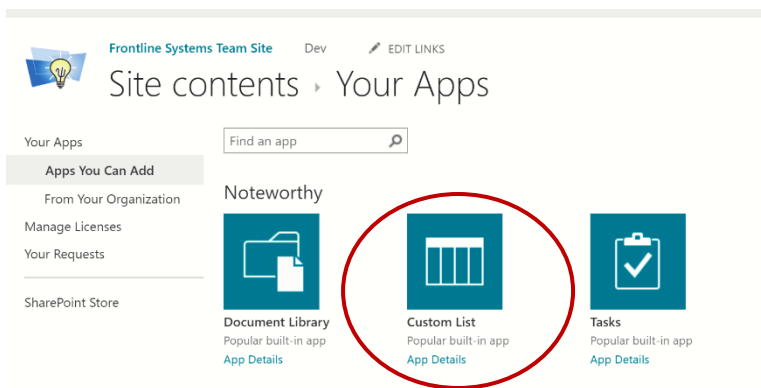
For this example, we will create a SharePoint list to hold our data. Log onto your Work or School account at office.com/signin, then click the SharePoint icon from the list of Microsoft products, on the left.



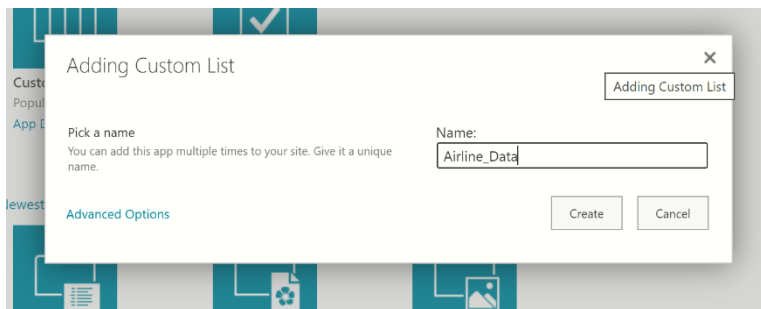
Click the Settings gear icon on the top right of the screen and select Add an app.



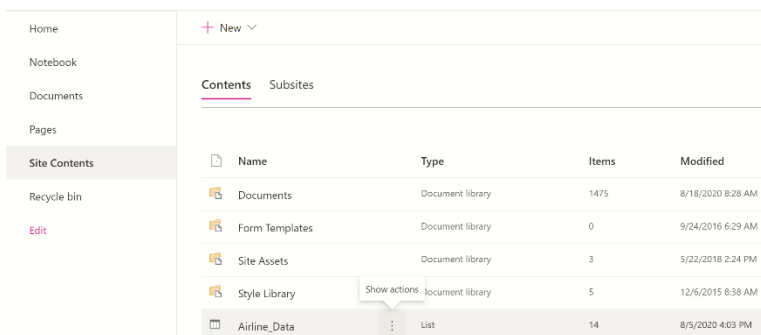
Then select Custom List.



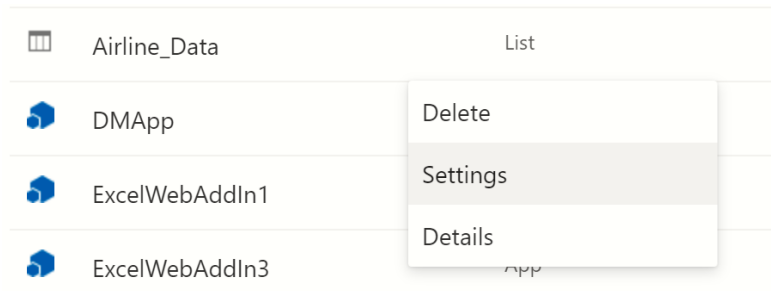
Enter a name for the list, such as "Airline_Data". Then click Create.



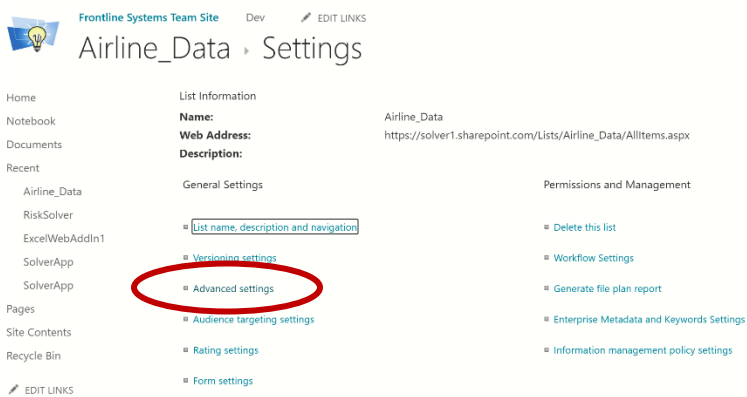
The Site Contents page will open displaying the Airline_Data list under Contents.



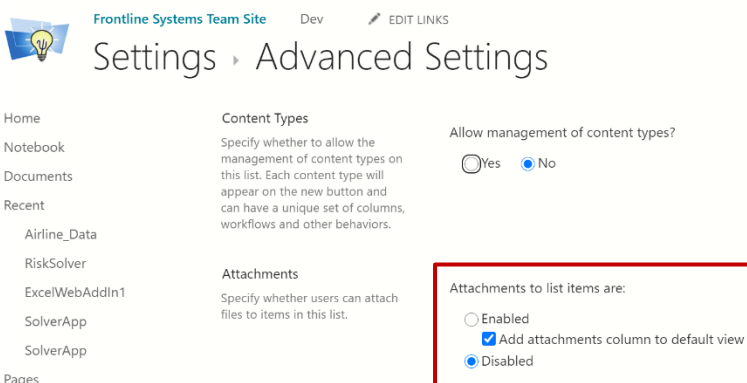
Click the three dots next to Airline_Data and select Settings.



Click "Advanced settings" ...



... and disable Attachments.



Click the Back arrow to return to the Settings page and scroll down till you see the "Columns" heading.


Columns

A column stores information about each item in the list. The following columns are currently available in this list:

Column (click to edit)	Type	Required
Title	Single line of text	✓
Modified	Date and Time	
Created	Date and Time	
Created By	Person or Group	
Modified By	Person or Group	

- [Create column](#)
- [Add from existing site columns](#)
- [Column ordering](#)
- [Indexed columns](#)

Click on "Title" beneath "Column (click to edit)" to edit this column. Set the Column name to FlightName and be sure to set "Require that this column contains information" to "No". Then scroll to the very bottom of the screen and click OK.

 Frontline Systems Team Site Dev [EDIT LINKS](#)

Settings ▸ Edit Column ⓘ

[Home](#)
[Notebook](#)
[Documents](#)
[Recent](#)
Airline_Data
RiskSolver
ExcelWebAddIn1
SolverApp
SolverApp
[Pages](#)
[TEST](#)
[Site Contents](#)
[Recycle Bin](#)
[EDIT LINKS](#)

Name and Type

Type a name for this column.

Column name:

The type of information in this column is:

Single line of text

Additional Column Settings

Specify detailed options for the type of information you selected.

Description:

Require that this column contains information:

☐ Yes ☒ No

Enforce unique values:

☐ Yes ☒ No

Maximum number of characters:

Default value:

☒ Text ☐ Calculated Value

Confirm that you now see FlightName as a Column. Note that now there is no checkmark beneath Required. This simply means that when entering a record, this field is not required.

Columns

A column stores information about each item in the list. The following columns are currently available in this list:

Column (click to edit)	Type	Required
FlightName	Single line of text	

Now we are ready to enter the rest of our column names by clicking "Create column".

Created By

Modified By

- [Create column](#)
- [Add from existing site columns](#)
- [Column ordering](#)
- [Indexed columns](#)

Enter name as "FlyingHours", set the "Type" to "Number" and set "Default value" to "Number". Then scroll to the bottom of the page and click OK.

Name and Type
Type a name for this column.

Column name:

The type of information in this column is:

☐ Single line of text
☐ Multiple lines of text
☐ Choice (menu to choose from)
☒ Number (1, 1.0, 100)
☐ Currency (\$, ¥, etc)
☐ Yes/No (check box)

Additional Column Settings
Specify detailed options for the type of information you selected.

Description:

Require that this column contains information:
☐ Yes ☒ No

Enforce unique values:
☐ Yes ☒ No

You can specify a minimum and maximum allowed value:
 Min: Max:

Number of decimal places:

Default value:
☒ Number ☐ Calculated Value

Create 3 more columns: OtherHours, Cost and Decisions using the settings shown in the chart.

Column name	Type	Default Value
OtherHours	Number	Number
Cost	Currency	Currency
Decisions	Number	Number

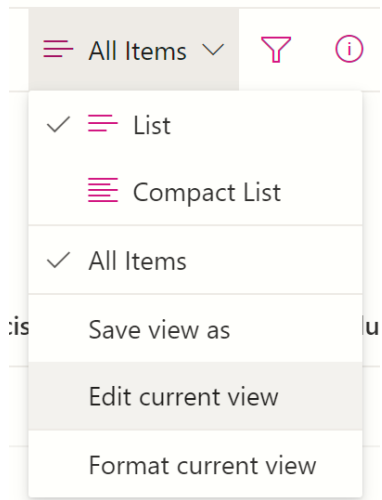
After clicking OK on the last column heading entry, your Columns should look like the following.

Columns

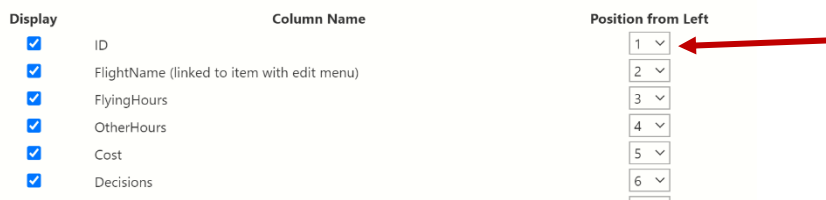
A column stores information about each item in the list. The following columns are currently available in this list:

Column (click to edit)	Type	Required
FlightName	Single line of text	
FlyingHours	Number	
OtherHours	Number	
Cost	Currency	
Decisions	Number	
Modified	Date and Time	
Created	Date and Time	
Created By	Person or Group	
Modified By	Person or Group	

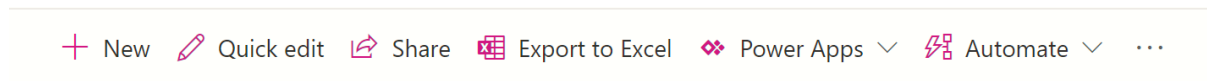
Click the Site Contents link in the left pane to return to Site Contents, then click Airline_Data to enter the data into the list. First, click All Items on the top right, and select Edit current view from the drop down menu.



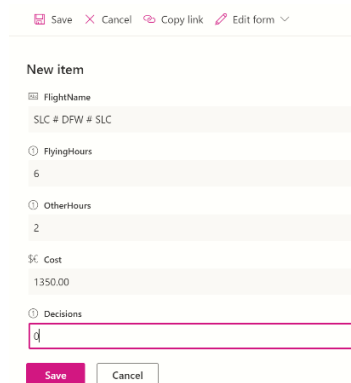
You'll be navigated to the Settings – Edit view page. Scroll down, select ID and make this column first by setting "Position from Left" to 1. Then scroll down to the bottom of the page and click OK.



Click New...











...then enter the data from the Airline Crew Scheduling Excel example (just enter 0 for Decisions) for the first record and click Save.



Repeat these steps to enter all 14 records. Note that "#" has replaced the airplane icon and "-" has replaced the "." icon in the FlightName column.

Once you have entered all the data, your list should be similar to the one in the screenshot below.

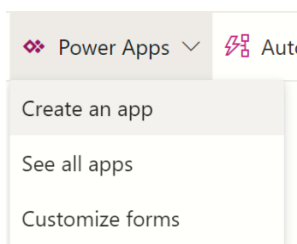
 New	 Quick edit	 Share	 Export to Excel	 Power Apps	 Automate		 All
Airline_Data							
ID	FlightName	FlyingHours	OtherHours	Cost	Decisions		
1	SLC # DFW # SLC	6	2	\$1,350.00	0		
2	SLT # DFW - SLC	3	11	\$1,425.00	0		
3	SLT # DFW # ORD - SLC	5	10	\$1,750.00	0		
4	SLC # ORD - SLC	4	10	\$1,550.00	0		
5	SLT # ORD # DFW - SLC	6	5	\$1,575.00	0		
6	DFW # SLC # DFW	6	3	\$1,425.00	0		
7	DFW # SLC - DFW	3	12	\$1,500.00	0		
8	DFW # SLC # ORD - DFW	7	7	\$1,925.00	0		
9	DFW # ORD # SLC - DFW	6	5	\$1,575.00	0		
10	DFW # ORD # DFW	4	5	\$1,175.00	0		
11	ORD # SLC # DFW - ORD	7	7	\$1,925.00	0		
12	ORD # SLC # ORD	0	0	\$1,000.00	0		

Now we are ready to create our Power App.

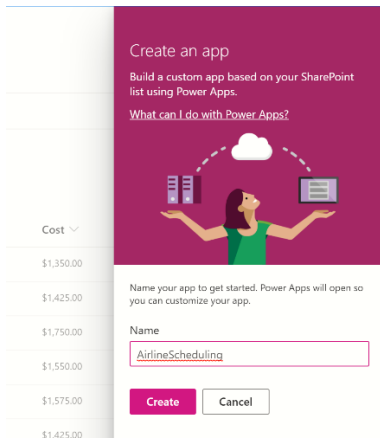
Creating the Power App

In this section we will create the application that can be called from a smartphone to solve the Airline Crew Scheduling optimization model.

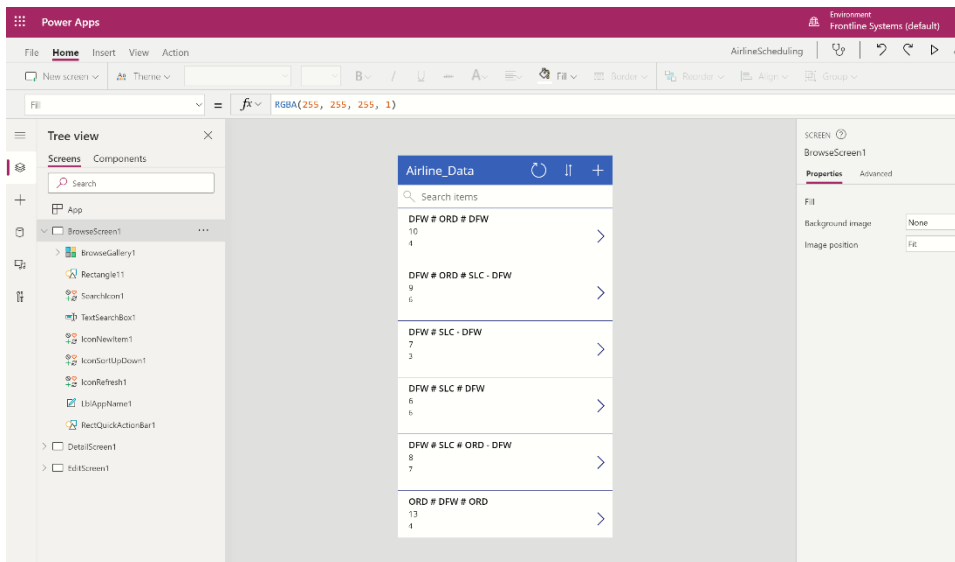
To create the Power App, click the down arrow next to Power Apps and select Create an app from the list.



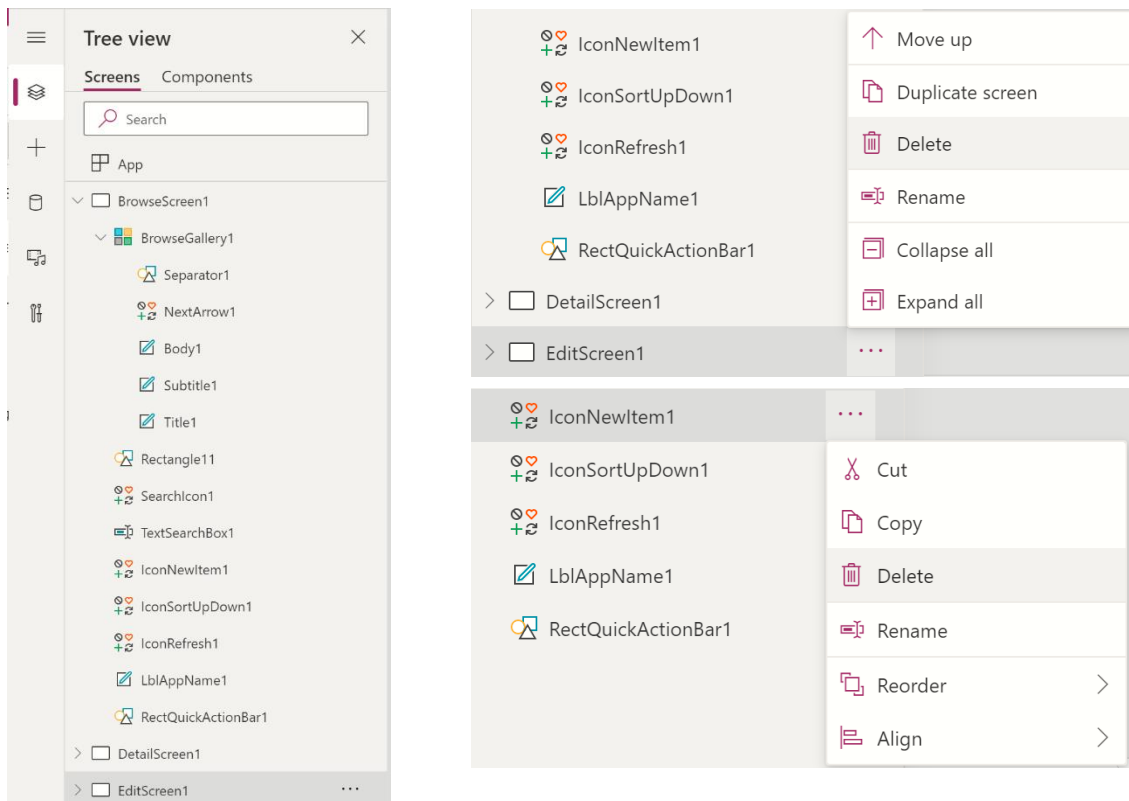
A pane opens on the right. Enter a name for the app, such as "AirlineScheduling", then click Create.



Power Apps will open and display most of our data automatically! We will just need to perform a few housekeeping duties to "pretty up" the application.



Since our app will not allow new schedules to be added, we can delete EditScreen1 and to keep it simple, we will also delete DetailScreen1. To delete, click the three dots (...) to the right of each screen in the Tree view and select Delete from the menu. Since we are deleting EditScreen1, we can also delete IconNewItem1.



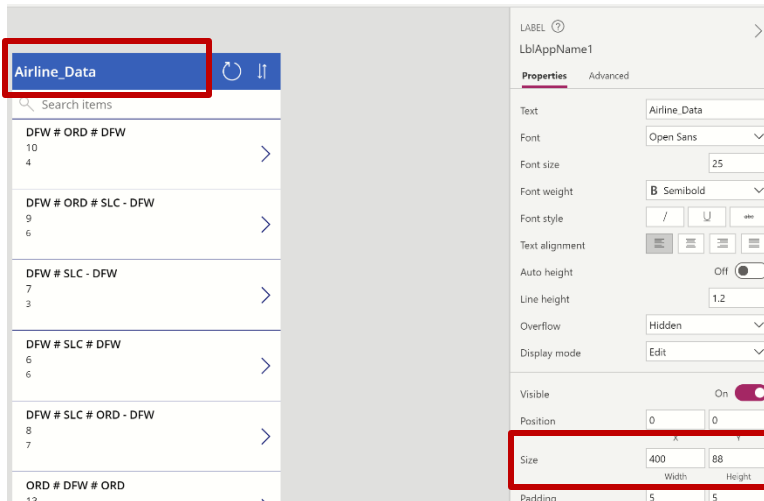
After you delete `IconNewItem1`, you'll notice that the Sort and Refresh icons are both moved to the far left of the screen.



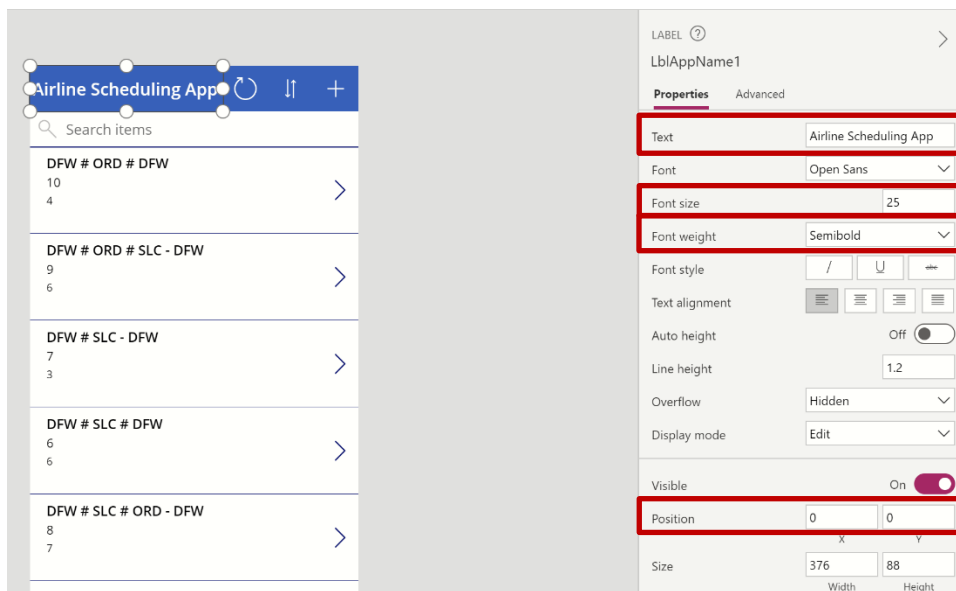
Use your mouse to select and move each icon as shown in the screenshot below.



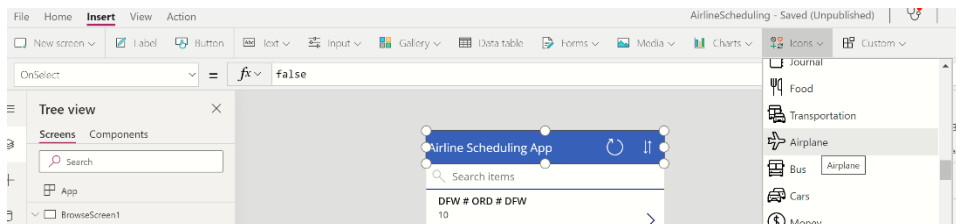
Notice that the title of the app (`Airline_Data`) has disappeared! Where did it go? Actually, it's still there, if you click on `LblAppName1` under Tree View (on the left) you'll notice three dots appear on the title bar. If you look at the properties pane for the label (on the right) you'll notice that the Width Size is 0. Simply enter a larger value than 0, say 400, to enlarge the field and display our current app title.



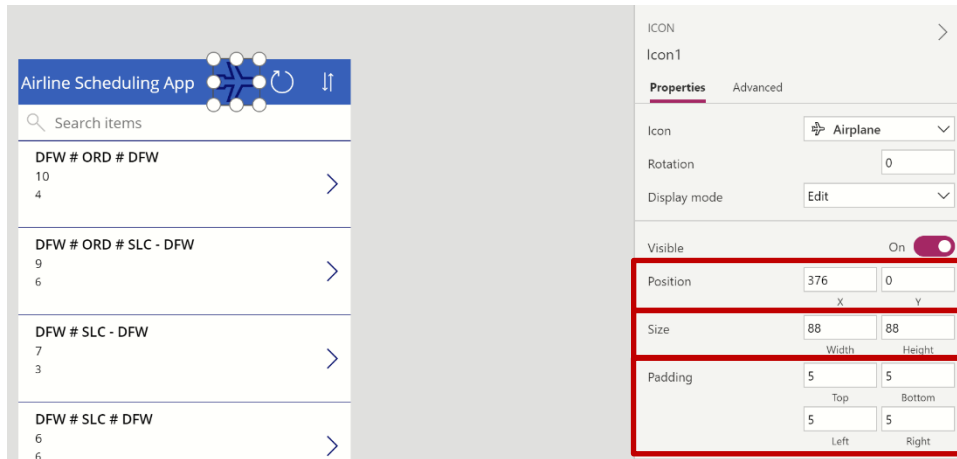
Let's change the title, at the very top of the application, from "Airline_Data" to "Airline Scheduling App" to make it a little more professional. Click "Airline_Data" on the title bar of the app. The Label pane opens on the right where you can enter "Airline Scheduling App" for **Text**, change the **Font size** to 25, change the **Font weight** to Semibold, and change the **X** and **Y** Positions to 0, 0.



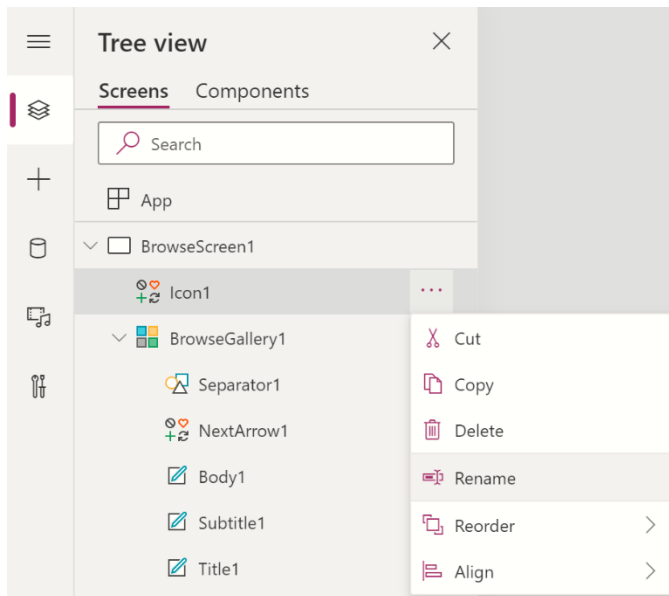
Now let's add a third icon, which is what we will click when we are ready to solve the model. Click the Insert – Icons – Airplane to place the Airplane icon onto the title bar.



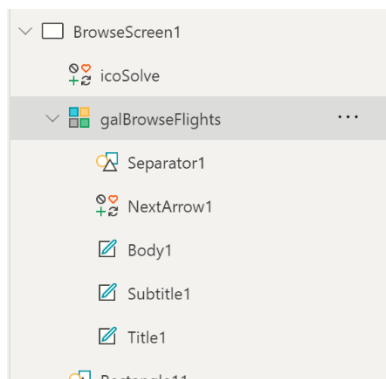
Change the Position of the Airplane icon to X = 376, Y = 0, Size to Width = Height = 88 and Padding to Top = Bottom = Left = Right = 5 as shown in the screenshot below.



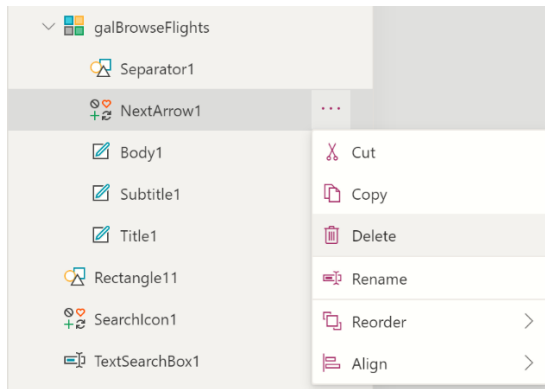
Then right click the newly added icon under Tree view (on the left) and rename to "icoSolve".



While we are here, right click BrowseGallery1 and again select Rename and rename to "galBrowseFlights". Then click the down arrow next to galBrowseFlights to expand the gallery. From here we will delete NextArrow1 and give more meaningful names to Body1, Subtitle1 and Title1.

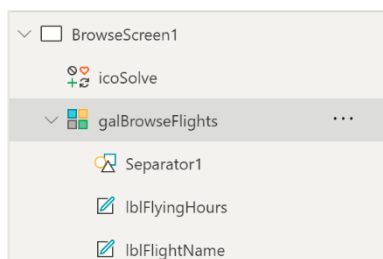


1. Click the three dots next to NextArrow1 and select Delete.

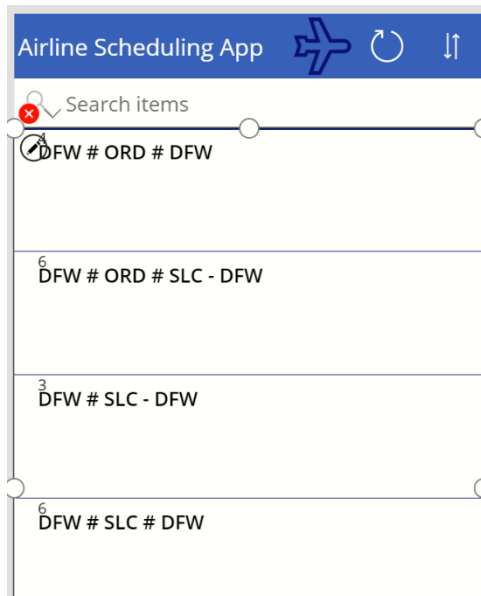


2. To give both Body1 and Title1 more meaningful names, click the three dots next to each and select Rename. Rename Body1 to "lblFlyingHours" and Title1 to "lblFlightName".
3. Click the three dots next to Subtitle1 and select Delete to delete the ID number from the app as this does not give us any relevant information.

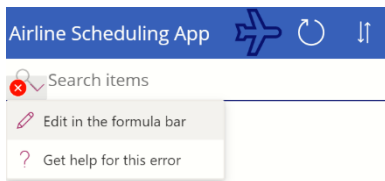
Afterwards, your Tree view should look similar to the screenshot below.



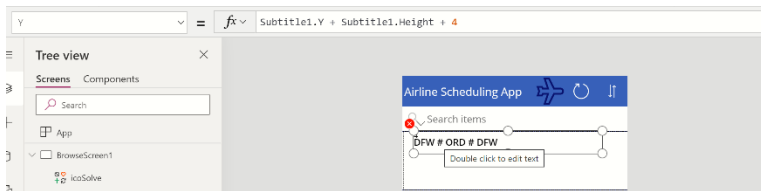
Now we are ready to customize our gallery. Notice that your app now shows a red circle with an X in the center. Our first error!



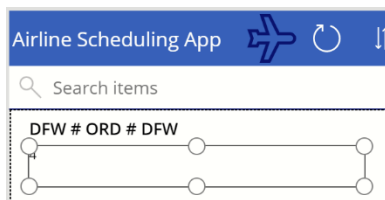
Click the down arrow next to the red dot and select "Edit in the formula bar".



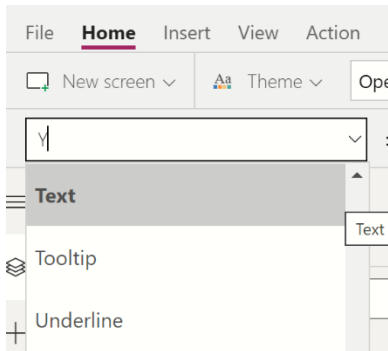
Immediately, the formula bar for the Y attribute for lblFlyingHours appears. You'll notice that the formula includes "Subtitle1" which we just deleted. This is what is causing the error. Subtitle1 no longer exists! To fix this error simply type "60" for "Y".



You'll notice that the error resolves and the field now appears beneath lblFlightName. Much better!



Click the down arrow next to Y (at the top beneath New screen) and select Text.



You'll notice that "ThisItem.FlyingHours" already appears for this attribute. Enter the following at the very beginning to display "Flying Hours: x", where x is the actual number of flying hours for the flight.



Now we can customize this field by using the lblFlyingHours label pane on the right. Set Padding Top, Bottom, Left and Right to 5. Feel free to let your creative juices flow and perform your own set of customizations! We will wait for you to finish..... All set? Let's move on!

LABEL ⓘ

IblFlyingHours

Properties Advanced

Text: Flying Hours: 4

Font: Open Sans

Font size: 16

Font weight: Normal

Font style: / U abc

Text alignment: [Left] [Center] [Right] [Justify]

Auto height: Off

Line height: 1.2

Overflow: Hidden

Display mode: Edit

Visible: On

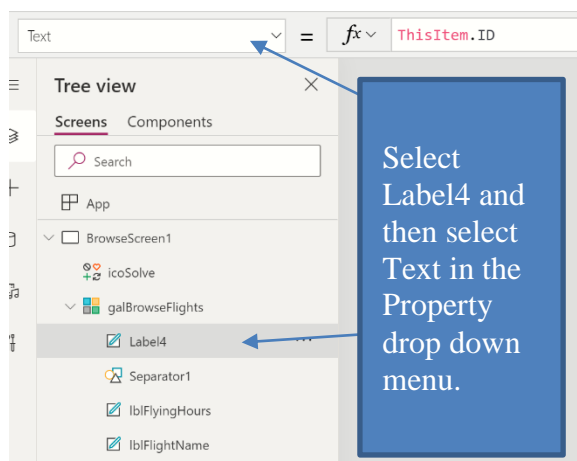
Position: X: 32 Y: 60

Size: Width: 224 Height: 44

Padding: Top: 5 Bottom: 5 Left: 5 Right: 5

Now we are ready to add 3 more labels in order to display data for Other Hours, Cost and finally the Decisions. Ready? Let's go!

1. Click Insert – Label to insert a new label onto the gallery. You'll notice that a new label is inserted into galBrowseFlights containing the number 10, which, if you inspect the Text attribute, you'll find is the ID number for this record. Since we really aren't interested in the ID number, we are going to change the Text of this attribute to display "Other Hours" in Step 2.
2. Make sure Label4 is selected under galBrowseFlights and select Text from the Property drop down menu.



3. Enter "Other Hours: " & ThisItem.OtherHours in the Text formula bar.

Text: "Other Hours: " & ThisItem.OtherHours

- On the right in the Label4 pane, you can customize to your heart's content. Here's what we did.

Label4

Properties Advanced

Text: Other Hours: 5

Font: Open Sans

Font size: 16

Font weight: Normal

Font style: / U abc

Text alignment: [Left] [Center] [Right] [Justify]

Auto height: Off

Line height: 1.2

Overflow: Hidden

Display: Edit

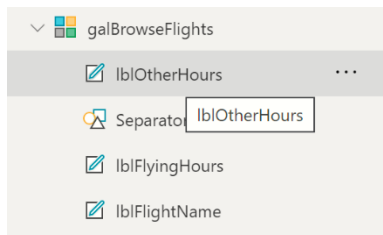
Visible: On

Position: X: 32 Y: 93

Size: Width: 240 Height: 74

Padding: Top: 5 Bottom: 5 Left: 5 Right: 5

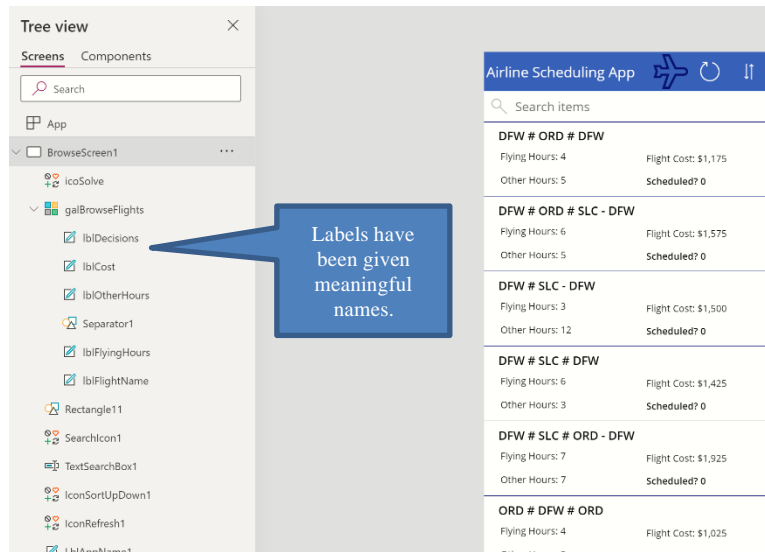
- Finally, let's give Label4 a more meaningful name by renaming this property, or attribute, to lblOtherHours.



- Now insert two more labels to display Cost and Decisions so that your application ends up similar to what is shown in the screenshot below. Remember, this is your time to shine when it comes to customizations. The world is your oyster!

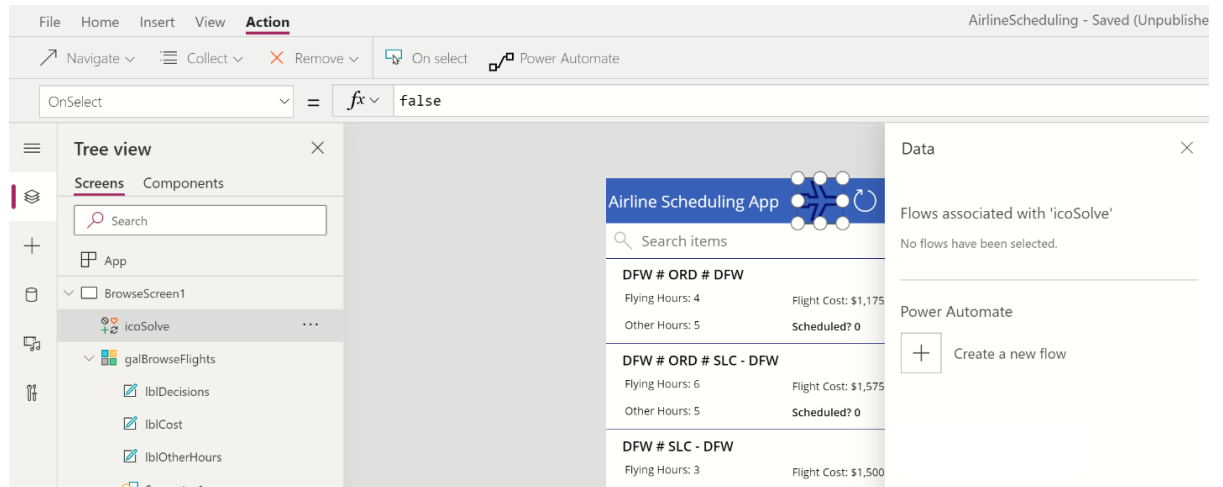
Text: Flight Cost: " & Text(ThisItem.Cost, "\$-en-US)\$#,###") - Note formatting for "Flight Cost".

Text: Scheduled? " & ThisItem.Decisions

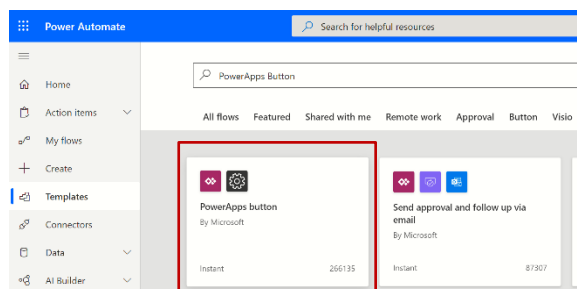


Creating the Decision Flow

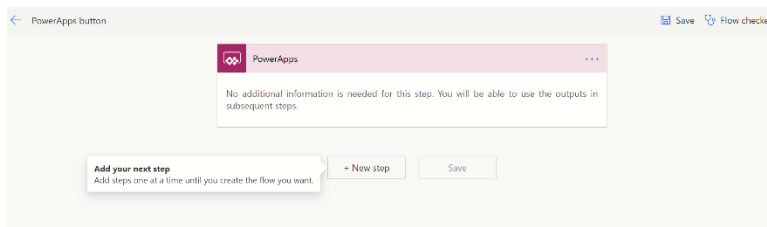
Now that our app is presentable, we are now ready to solve the example optimization model and report the results back to our application. We will use the Airplane icon to initiate the solve. Click the icon and then go to Action, the Property menu will display "OnSelect". Then click Power Automate to open the Data tab and click Create a new flow.



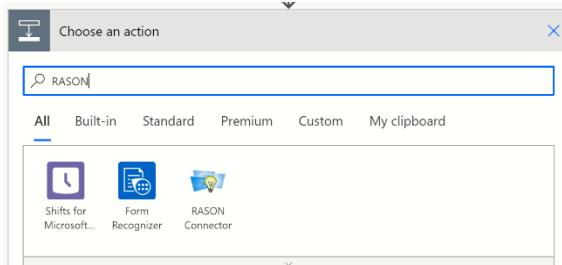
You will be immediately directed to Power Automate. Select PowerApps button to initiate the flow.



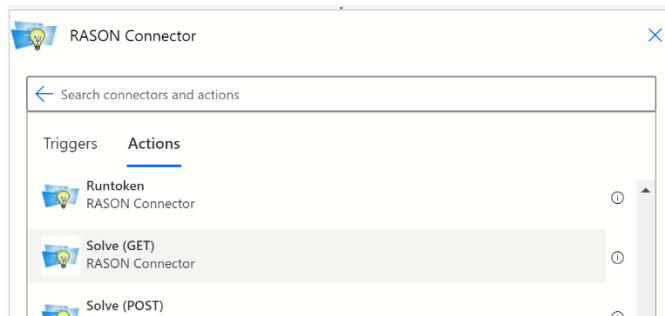
Click +New step to add our first step to the decision flow.



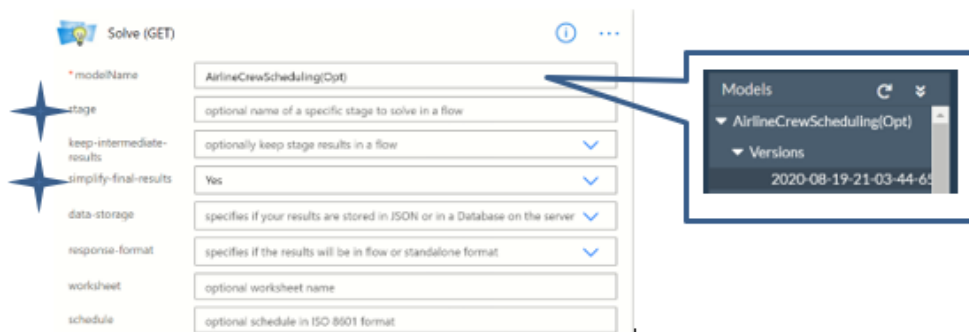
Enter RASON into the search box and then select "RASON Connector" from the list.



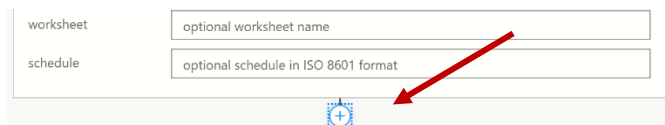
Then select Solve (GET) from the list.



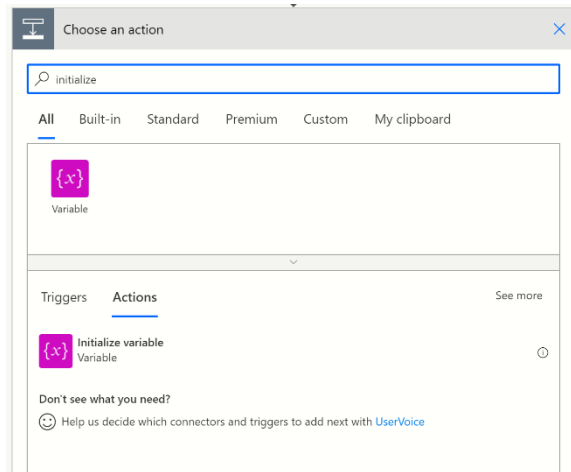
Enter "AirlineCrewScheduling(Opt)" for modelName since this is the name of the model we uploaded to the RASON Server.



Click +New Step button to create a new step.

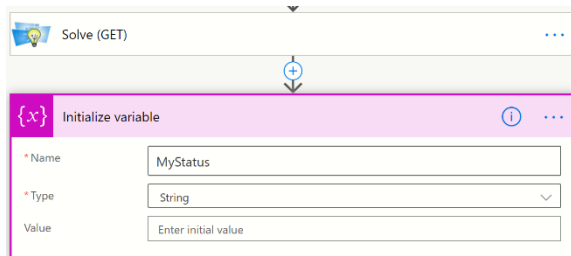


This time search for "initialize" and select "Initialize variable" from the list.

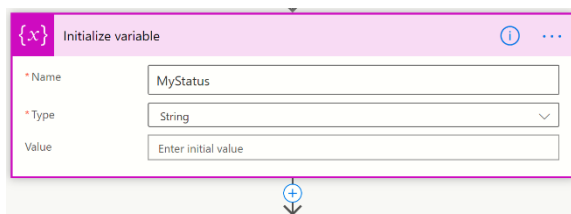


In this step we will initialize a new variable to "null".

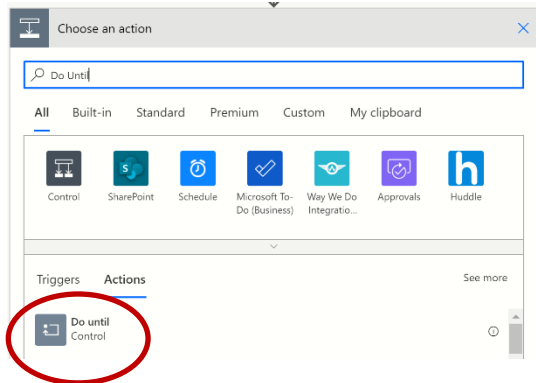
- Enter a name for the variable, such as MyStatus
- Select String for Type.
- Leave Value blank.



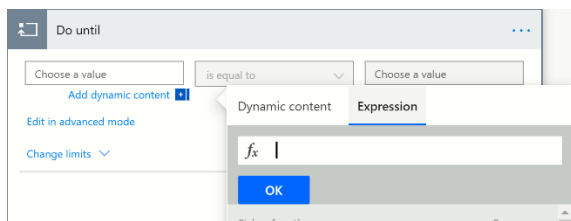
Click the plus sign on the down arrow to add a new step.



Enter "Do until" into the search field and select "Do until" from the list. This step will repeat until a criteria has been met. In our case, the criteria is that the status of the solve is listed as "Complete".



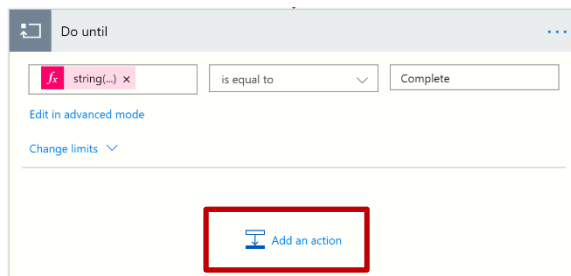
Click Choose a value on the left to open the Dynamic content window, then select Expression.



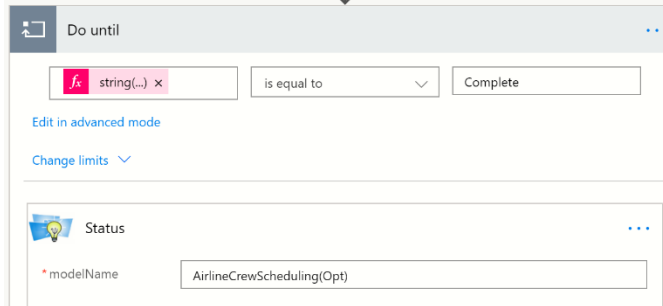
Enter the following into the function field: `string(variables('MyStatus'))`. Then click the OK button.

This expression sets our newly added variable, MyStatus to a string variable. I know, I know...we did this already when we initialized the variable. Unfortunately, that does not appear to be "enough" so, we will use this expression to tell Power Automate a 2nd time that MyStatus is a string variable.

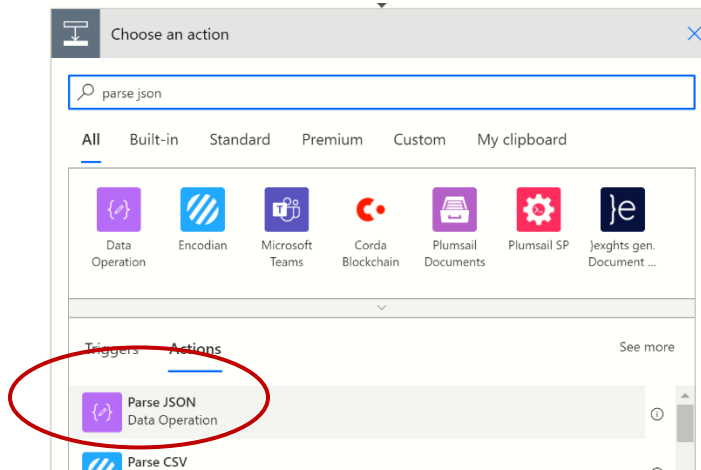
Within our "Do until" we want to perform this loop that we are creating until the GET Status RASON API Endpoint returns Status = Complete, which signifies that the solve has finished. To do this we first need to add a 2nd action within the Do until that retrieves the status of the solve. So, click "Add an action" at the bottom of the Do until step.



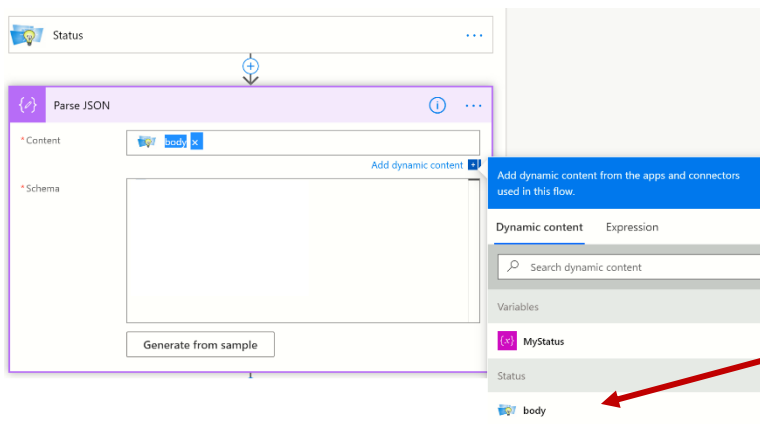
Type RASON in the search field and choose Rason Connector, then select Status from the list of REST API Endpoints. Enter "AirlineCrewScheduling(Opt)" for modelName. (Recall this is the name given to our model that now resides on the RASON Server.) This step retrieves the Status of the model solve so that we can check if the solve has been completed.



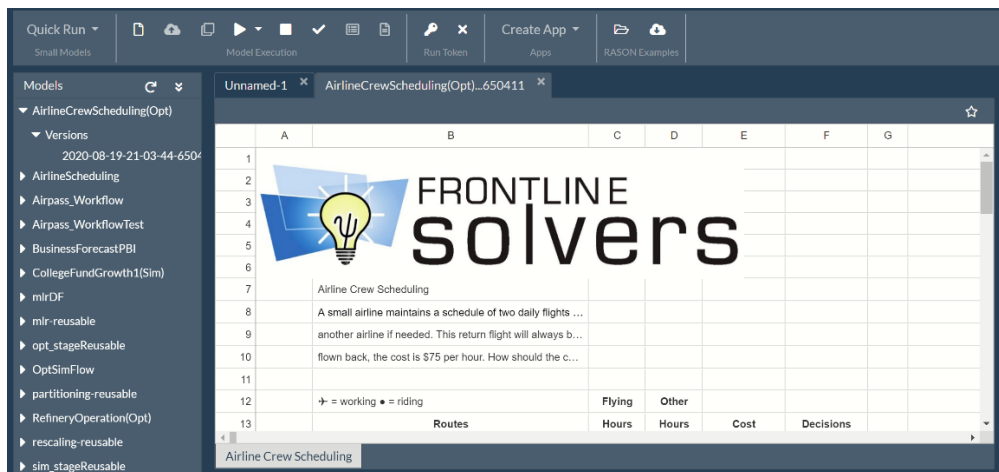
Now that we have the status, we want to know if solving is "Complete" or "Incomplete". Click Add an action, within the Do until, and search for "Parse Json", then select the "Parse JSON" action icon. This action will "read" the JSON response from the GET Status API Endpoint.



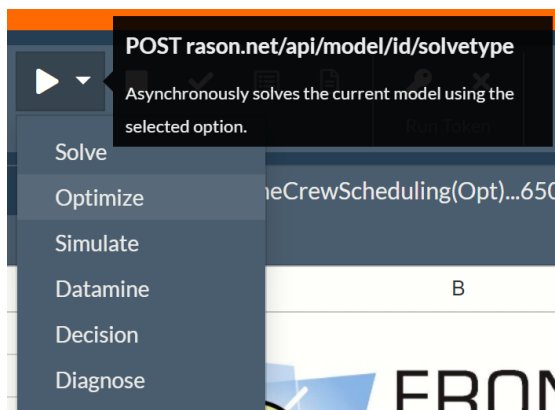
Select the Content field and then select "body" under Status.



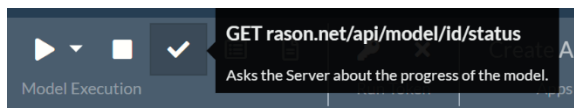
Now we need to supply an example of a response from this endpoint. To do so, we will go back to www.Rason.com and solve AirlineCrewScheduling(Opt).



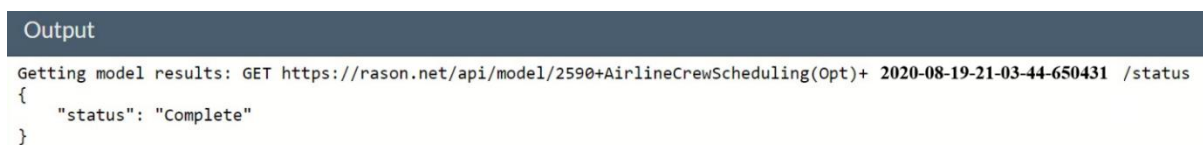
Click the down arrow beneath the Play button icon and select Optimize from the list.



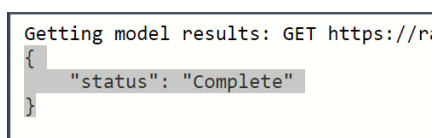
Then click the checkbox to call GET rason.net/api/model/id/status to check the status of the solve.



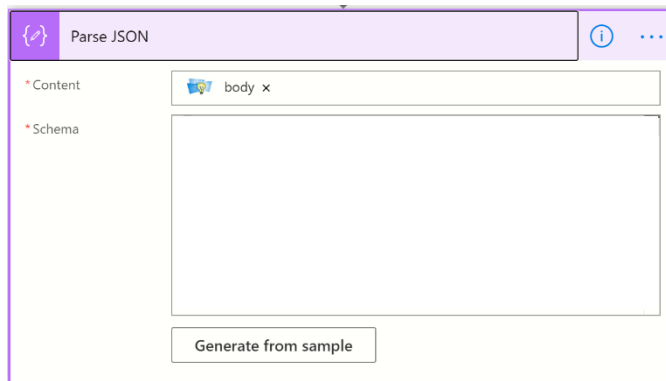
In the Output window (at the bottom of the screen) you'll find the body of the output from GET rason.net/api/model/id/status.



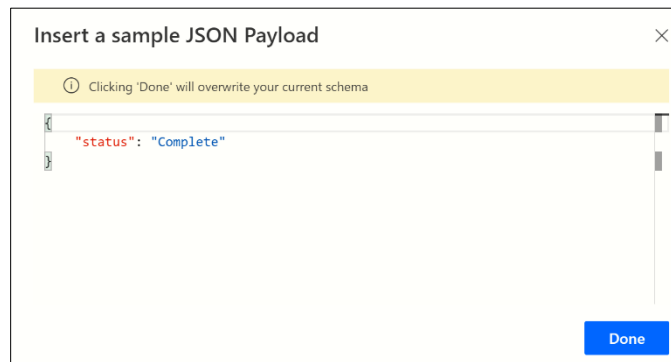
Highlight and copy (CTRL + C) { "status": "Complete"}...



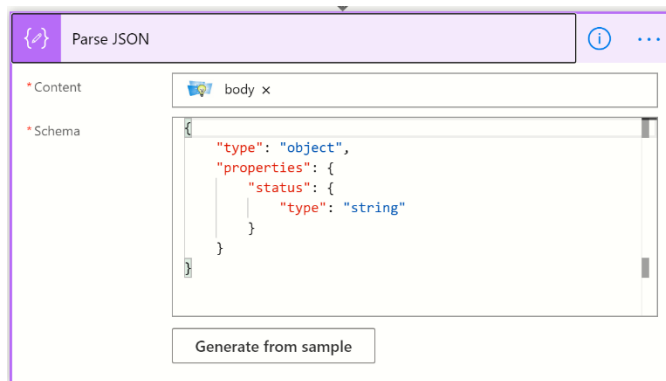
... and then go back to Power Automate, click "Generate from sample"...



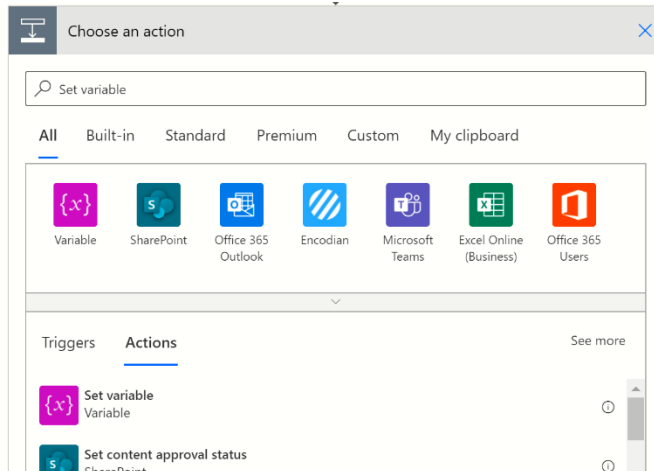
... paste the body of the API call into the "Insert a sample JSON Payload" and click Done.



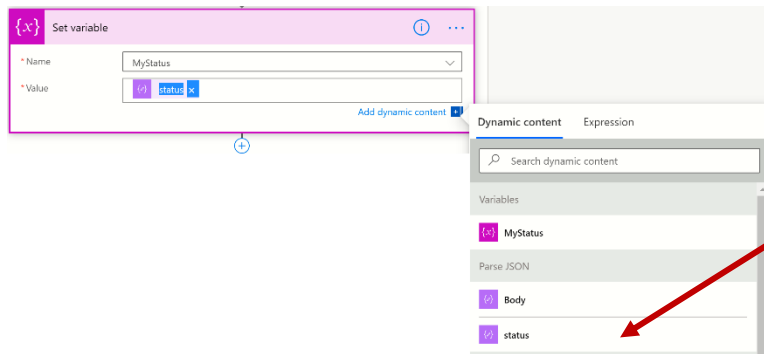
Notice that the required result schema from GET Status has been translated from the sample JSON Payload, { "status": "Complete" }.



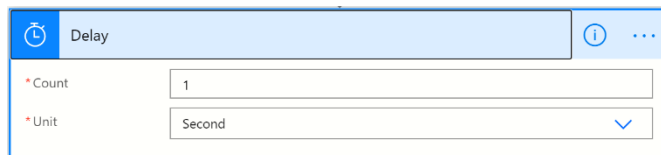
Next we need to set the MyStatus variable (initialized to null in the initial Do until step) to "status". Add a new step, search for "Set variable" and then select the Set variable action.



Select MyStatus for Name and then choose "status" under Parse JSON within the Dynamic content window. (Remember that the Dynamic content window appears automatically when you select certain fields such as "Value".)



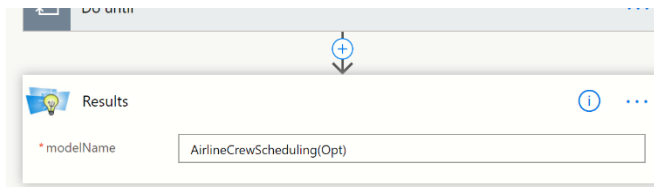
Finally, we have come to our last step, click Add a new step, search for "Delay", then select the "Delay" action. We need to add a delay in order to slow down the calls to GET status so that the loop gives the model time to execute. Set the Count to "1" and the Unit to "Second".



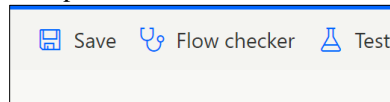
Now our "Do until" is complete. This step will be performed until the "Status": "Complete" is returned from the REST API call, GET rason.net/api/model/id/status.

Okay, we are almost there! We started the solve, we obtained the status of the solve, and now we need to retrieve the results.

To obtain the results from our model, we must call the REST API endpoint, GET rason.net/api/model/id/result. To do so, click +New Step, search for RASON, select RASON Connector and then select Results from the list. Enter the RASON model name for modelName, AirlineCrewScheduling(Optional).

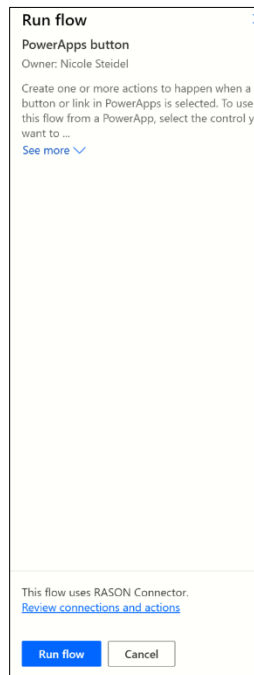


Let's pause here to test our Flow. At the top right click Save and then Test.



Select "I'll perform the trigger action" and then click **Test**.

Then click Run flow from the next pane.



Click the Flow Runs Page link on the Run flow pane.

Run flow



Your flow run successfully started. To monitor it, go to the [Flow Runs Page](#).

PowerApps button > Run history

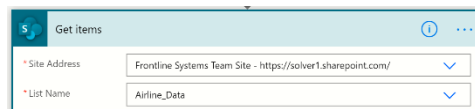
Start time	Duration	Status
Aug 27, 12:14 PM (4 sec ago)	00:00:03	Test succeeded

Click the date which will take you back to your flow. If you click on the final step of the flow, Results, you can observe exactly what was returned.

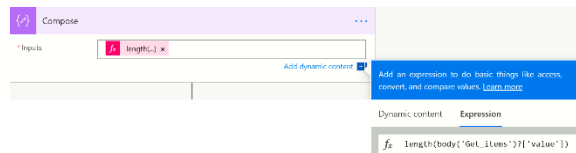


The next task that we'll tackle is to obtain the number of records in our SharePoint list. Currently our list is static, meaning that there are 14 records in the list and none will be added or deleted within this introductory application. However, to make our app more dynamic, we will utilize the "Get items" SharePoint endpoint to get all the items in our list, then use the Compose action to construct an object to check the length of our list and then initialize a new variable with the returned value.

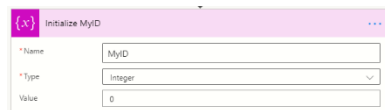
1. Use the "Get items" SharePoint action to get the contents of our list.
 - a. Click +New Step and search for "Get items", then select the "Get items" SharePoint action from the list.
 - b. Click the down arrow next to Site Address and select your Microsoft Site Address.
 - c. Click the down arrow next to List Name and select our SharePoint list, Airline_Data. Leave remaining fields at their defaults.



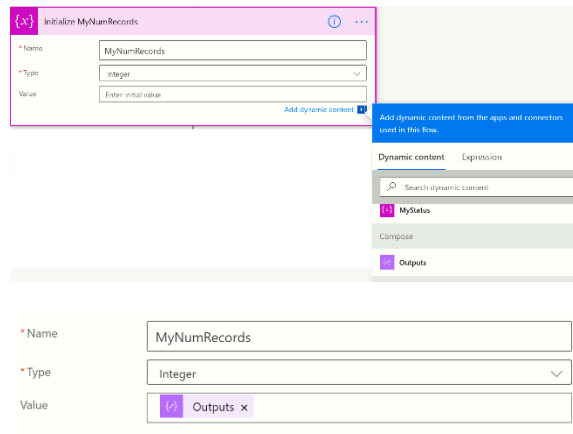
2. Use "Compose" to construct an object that checks the length of Airline_Data.
 - a. Click +New Step and search for "Compose", then select the "Compose" data operation from the list.
 - b. Click "Add dynamic content", click the Expression tab and enter the following formula: `length(body('Get_items')['value'])`, then click the OK button. This expression returns the value from the body of the "Get items" step. In this case, the value is 14 since there are 14 entries in our list.



3. Initialize the variable MyID and set this variable equal to 0.
 - a. Click +New Step and search for "Initialize new variable", then select "Initialize variable" from the list.
 - b. Enter "MyID" for Name and select Integer for Type.
 - c. Enter 0 for Value.

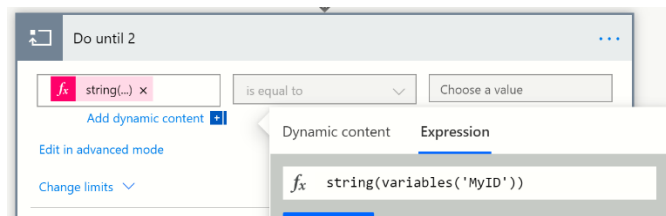


4. Initialize the variable MyNumRecords and set this variable equal to the value returned from Compose.
 - a. Click +New Step and search for "Initialize new variable", then select "Initialize variable" from the list.
 - b. Enter "MyNumRecords" for Name and select Integer for Type.
 - c. Click the Value field and select Outputs from the list.

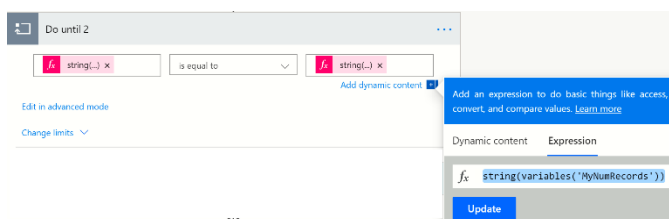


In the last task we will ask for the final variable values in order to report them back to our application.

Click +New Step, search for "Do until", then select the "Do until" control from the list. Within this step, we will update the SharePoint Airline_Data list with our final variable values. Select the field on the left and click the Expression tab on the Dynamic content pane, then enter: `string(variables('MyID'))` which simply converts the variable myID into a string. Then click the OK button.



Select "is equal to" for the middle field. Then click the field on the right, click Expression and enter `string(variables('MyNumRecords'))`.



Click Add an action, search for Update item and select the Update item SharePoint Action. Click the down arrow next to Site Address and select your Site Address. Click the down arrow next to List Name and select, Airline_Data. Then click the field next to ID and select MyID from the list.

Click Decisions, which is the column where our final values will be updated to, select Expressions and enter: `body('Results')['f14:f27'][variables('MyID')]`, then click the OK button. Now what is going on here, you may ask? Well `body('Results')` is referring to the body of the response from Results, `['f14:f27']` is the name of the variable block from the `AirlineCrewScheduling(Opt)` model on www.RASON.com. Finally, `[variables('MyID')]` refers to the newly created MyID variable that we just initialized to 0.

Now we need to loop through all of our variables so click "Add an action", search for "Increment variable" and select the "Increment variable" action from the list. Click the down arrow next to Name and select "MyID" from the list. Enter 1 for Value.

Variable block from RASON model.

```

"variables": {
  "f14:f27": {
    "value": 0,
    "type": "bin",
    "lower": 0,
    "upper": 1,
    "comment": "Rotation_decisions",
    "finalValue": []
  }
},

```

That's it! We did it! We've finished our flow! Let's Save and Test to see if it completes.

Select "I'll perform the trigger trigger action" and then click **Test**.

Then click "Run flow" from the next pane.

Click the Flow Runs Page link on the Run flow pane.

Run flow

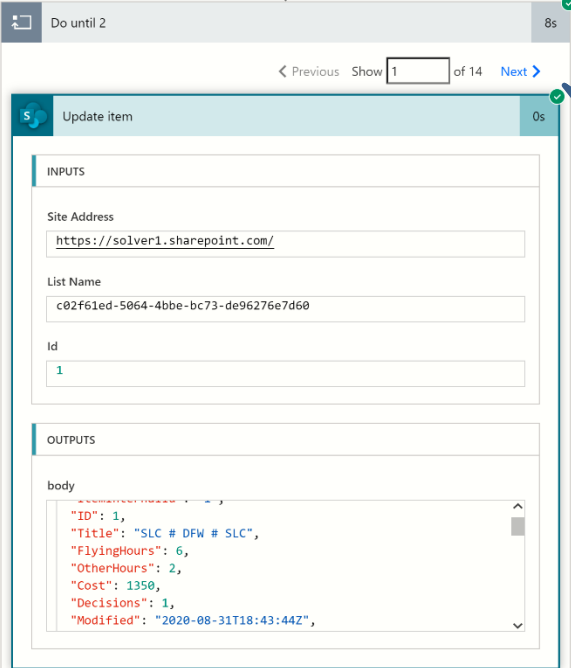


Your flow run successfully started. To monitor it, go to the [Flow Runs Page](#).

PowerApps button > Run history

Start time	Duration	Status
 Aug 21, 11:43 AM (2 min ago)	00:00:13	Test succeeded

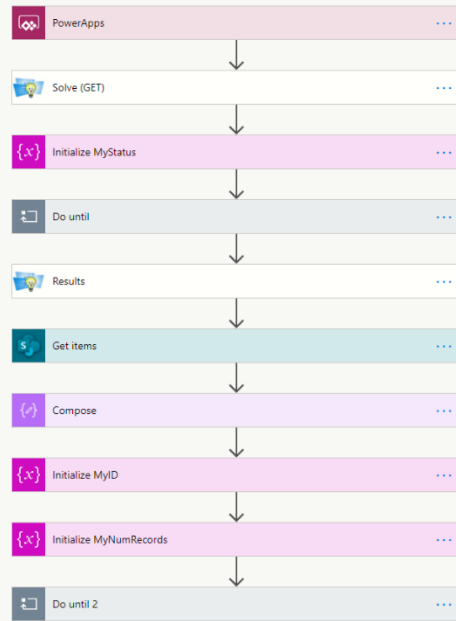
Click the date at the top of the list to return to your flow. If you click on the final step of the flow, Update item within Do until 2, you can observe exactly what was returned.



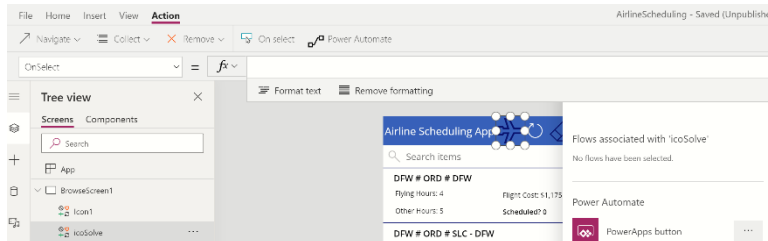
The screenshot shows the 'Do until 2' step in a PowerApps flow. The 'Update item' step is selected, showing its configuration and output. The 'Inputs' section includes 'Site Address' (https://solver1.sharepoint.com/), 'List Name' (c02f61ed-5064-4bbe-bc73-de96276e7d60), and 'Id' (1). The 'Outputs' section shows the 'body' of the response, which is a JSON object containing details about the updated item, including its ID, title, flying hours, other hours, cost, decisions, and modified date.

Click Next to increment through the 14 records in the Airline_Data SharePoint list.

Here is the entire Flow. Notice that I have renamed some of the steps to more meaningful titles, for example "Initialize Variable" has been renamed to "Initialize MyStatus".



Okay, now we are ready to import the results back into our app. Click back to Power Apps and to the AirlineScheduling app. Recall that our last step in completing this app was to finish calling the Power Automate flow. Click the airplane icon and then click Action – Power Automate and select our newly finished flow, PowerApps button.



The text "PowerAppsbutton.Run(" is inserted into the OnSelect field. To close the call to our flow, just enter ")". The full command is: `PowerAppsbutton.Run()` ;

Now we will insert a few more lines of code just to finish up this call (press Enter + Shift to move down a line):

`Refresh([@Airline_Data]);` - Refreshes the Airline_Data SharePoint list with the new data.
`Notify("Fetching schedule..... Click Refresh after this window closes to view schedule.", NotificationType.Warning, 5000);` - Displays a warning with instructions.

Note: The Notify line of code is needed due to a bug within Power Apps where the Refresh command does not execute correctly.

```

PowerAppsbutton.Run();
Refresh([@Airline_Data]);
Notify("Fetching schedule..... Click Refresh after this window closes to view schedule.", NotificationType.Warning, 5000);
  
```

Before we test our app, let's add a new button, a "reset" button that resets the decision variable values in the Airline_Data SharePoint list to 0. That way we will be able to tell if our app has actually completed successfully.


Search items

DFW # ORD # DFW

Flying Hours: 4

Flight Cost: \$1,175

Other Hours: 5

Scheduled? 0

DFW # ORD # SLC - DFW

Flying Hours: 6

Flight Cost: \$1,575

Other Hours: 5

Scheduled? 1

DFW # SLC - DFW

Flying Hours: 3

Flight Cost: \$1,500

Other Hours: 12

Scheduled? 0

DFW # SLC # DFW

Flying Hours: 6

Flight Cost: \$1,425

Other Hours: 3

Scheduled? 1

DFW # SLC # ORD - DFW

Flying Hours: 7

Flight Cost: \$1,925

Other Hours: 7

Scheduled? 0

ORD # DFW # ORD

Flying Hours: 4

Flight Cost: \$1,025

Other Hours: 3

Scheduled? 1

Glossary of RASON Property Names

RASON Property Names Glossary

o = optimization, s = simulation, c = calculation (decision tables), d = data science, w = decision flow

Defined Name	Function	Application Use	Description
action	object string property	d	Defines an "action" to be performed in a data science model such as fit, predict, transform, or forecast.
actions	section type	d	The section in a RASON data science model where an action (fit, predict, transform or forecast) is defined.
activeSheet	modelSettings property	o/s/c/d	Used (only) during automatic conversion from Analytic Solver (desktop or online) to the RASON modeling language.
aggregationType	object string property	d	This option provides 5 statistics that can be inferred from a Big Data dataset: sum, average, standard deviation, minimum and maximum.
algorithm	object string property	d	Use this property to specify the algorithm to be used to perform a data science transformation.
asynch	object string property	d	Submits Big Data job asynchronously.
autoRegressiveOrder	object string property	d	Sets the non-seasonal Autoregressive parameter (p) in an ARIMA forecasting model.
ave	statement keyword	o/s/c	Keyword that averages a set of values.
average	result property	o/s	Creates an empty array to hold the mean (or average) of the uncertain variable in a simulation, simulation optimization or stochastic optimization model.
awsS3	object bool property	d	If Big data source is Amazon S3 (AWS S3), set this option to true.
awsS3AccessKey	object bool property	d	For use with Big Data. Passes the Amazon S3 access key.

awsS3SecretKey	object bool property	d	For use with Big Data. Passes the Amazon S3 secret key.
baseIndex	object numeric property	d	Factorization converts a variable into a new numeric, categorical variable. Use this parameter to specify the number with which to begin the categorization.
bootstrapSeed	object numeric property	d	When performing a classification or regression task, this value specifies the seed for random resampling of the training data for each weak learner.
binding	data property	o/s/c/d	<p>Used within data section to bind dataset to a data source.</p> <p>Used within dataSources section to allow data to be edited outside of the RASON model environment, i.e. from a URL .</p> <p>May also be used within other sections (such as variables, constraints, objective, uncertainFunctions, uncertainVariables, etc.) in conjunction with the valueCols property to bind imported values from a readable data source.</p> <p>Used in RASON Data Science to bind fitted model results to a writeable data source or to bind a previously fit model to a "fittedModel".</p>
binningTypeFeatures	object string property	d	When using the Univariate for Transformation DM algorithm, this parameter transforms continuous input variables into categorical variables.
binningTypeTarget	object string property	d	When using the Univariate for Transformation DM algorithm, this parameter transforms a continuous output variable into a categorical variable.
binValueOption	object array property	d	Binning: Determines how the variables are to be binned.
categoricalFeaturesNames	object array property	d	In the Univariate for Transformation DM algorithm, enter categorical variables by name using this parameter.
categoricalTarget	object bool property	d	In the Univariate for Transformation DM algorithm, setting this option to "true" denotes that the Output Variable is a categorical variable while setting this option to "false" denotes that the Output Variable is a continuous variable.
categories	object array property	d	

sensorType	distribution property	o/s	Sets global default bounds on the probability distributions of all uncertain variables.
cutoffType	distribution property	o/s	Sets global default bounds on the probability distributions of all uncertain variables.
chanceConstraintNorm	modelSettings property	o	Determines the norm (distance measure) used to constrain the size of uncertainty sets in the Robust Counterpart model.
chanceProbability	constraint property	o	Defines the percentile for use with VaR, CVaR and USet objective types.
chanceType	constraint property	o	Defines the constraint or constraint block as a chance constraint(s). Constraint or constraint block <i>must</i> contain uncertainties. The property chanceProbability property must exist if chanceType exists.
colIndex	datasource property	o/s/c/d	Use this property to create an implicit index set consisting of integer numbers from 1 to the number of columns.
colNames	object array property	d	When manipulating data within the data section of a DM model, this property assigns column names. (Array property)
comment	common property	o/s/c/d	Use a comment to describe the data.
compound	statement keyword	o/s/c	Use "compound" when the formulas in a loop() do not need to be repeated, but still executed in order. See the Using Statements section above for more information.
computeConceptImportance	object bool property	d	Text Mining: if true, the Concept Importance table is computed.
computeGroupCounts	object bool property	d	Big Data: Use this option when 1 or more Grouping Variables exist.
computeTermImportance	object bool property	d	Text Mining: If true, the Term Importance table is computed.
confidenceLevel	object numeric property	d	Smoothing methods, Logistic Regression or Time Series: Use this property to set the confidenceLevel.
connection	datasource property	o/s/c/d	Use this property to pass the filename, path and/or credentials for the data source.
constraints	section	o	This optional section is used for defining normal, recourse or chance constraints in optimization, stochastic optimization or simulation optimization models.
content	datasource property	d	Use this property to read data in some specific manner such as: corpus – text corpus, itemset – item list, json-model – Model in JSON format, pmml-model – Model in PMML format, table – table or time-series – time series dataset.

correction	object numeric property	d	Rescaler: Sets the "correction" option when technique = "NORMALIZATION" or "ADJUSTED_NORMALIZATION". See "technique" explanation below.
costFunction	object numeric property	d	Neural Networks for Classification or Regression: Sets the cost function. The cost function measures "how well" a neural network performed with respect to a given training dataset and the expected output.
data	section	o/s/c/d	Data arrays may be defined and calculated in this optional section to be used later in a data science method. If you are pulling data from an external source, use this section to "bind" the data to an array or table.
dataCol	object string property	d	Data section: Selects input variables. (String property)
dataForErrorComputation	object string property	d	Neural Networks: Specifies the data partition to be used to estimate the error after each training epoch.
dataFormat	object string property	d	Big Data Sampling: If data is in Apache Parquet format, use "PARQUET" for this option. If your data is in Delimited Text format, USE "CSV".
dataFrame	result property	d	A collection of data organized into named columns of equal length and homogeneous type. RASON DM uses DataFrames to deliver input data to an algorithm and to deliver the results of the algorithm back to the user. DataFrames hold heterogeneous data across columns (variables): numeric, categorical, or textual.
datasets	section	d	In a RASON DM model, where the external dataset is "bound" to a RASON dataset.
dataSources	section	o/s/c/d	External data sources are defined in this section.
decisionTables	section	o/s/c	Defines a decision table which contains a set of rules specifying actions to perform based on specific conditions.
deterministicEquivalent	modelSettings property	o	Use this option to specify that an optimization model with uncertainty will be solved using deterministic equivalent form.
difference	modelSettings property	o	Time Series: Sets the non-seasonal Difference parameter (d). Lag Analysis: Returns the differenced data.
Dimensions	object property		Defines a n – dimensional vertical array with according to the number of elements passed. All arrays are 1 – based. If missing, variable array shape will be implicitly defined by the shape of the <i>lower</i> , <i>upper</i> , or <i>value</i> properties, however, for readability of the code, the use of the

			<i>dimensions</i> property is recommended.
Direction	datasource property	o/s/c/d	Determines if an external data will be imported or exported.
Dissimilarity	object string property	d	Hierarchical clustering uses the Euclidean Distance as the similarity measure for working on raw numeric data. When the data is binary, the remaining two options, Jaccard's coefficients and Matching coefficient are available.
dualValue	result property	o	Creates an empty array to hold the reduced cost for the variable or variable block.
dualLower	result property	o	Creates an empty array to hold the Allowable Decrease for the constraint or constraint block.
dualUpper	result property	o	Creates an empty array to hold the Allowable Increase for the constraint or constraint block.
Else	statement keyword	o/s/c/d	Executed if the result of a previous test condition evaluates to false
emailToken	object string property	d	Text Mining: If NORMALIZE_EMAIL is passed for "processing", then this term may be used to replace email addresses appearing in the document collection with the term entered here.
endPhrase	object string property	d	Text Mining: Text appearing before the first occurrence of the Start Phrase (if used) will be disregarded and similarly, text appearing after End Phrase (if used) will be disregarded.
engine	object property	o	Identifies the engine to be used during the optimization or stochastic optimization solve
engineSettings	section	o/s/c	In this optional section, you'll specify the engine to be used to solve the optimization, simulation optimization or stochastic optimization model and/or set any engine options.
Equal	object property	o	Defines an equality constraint in an optimization or stochastic optimization model.
errorTolerance	object numeric property	d	Neurel Networks: Use this option to set the error tolerance. The error in a particular iteration is backpropagated only if it is greater than the value specified for this option.
estimator	section	d	This section is where the "estimator" is defined and is applicable only to algorithms that "fit" a model.
evaluations	result property	d	RASON DM: Requested results in a DM model. The selection for this property depends on the "model" or "estimator" selected.
excludedCols	object array property	d	Data section: Excludes specified columns from the data science method. (Array property)

exclusionTerms	object array property	d	Text Mining: If used, terms entered for "exclusionTerms" will be removed from the document collection during pre-processing.
exclusiveInclusionTerms	object array property	d	Text Mining: If used, terms entered for "exclusiveInclusionTerms" will be removed from the document collection during pre-processing.
expectation	result property	o/s	Creates an empty array to hold the mean (or average) of the uncertain function.
featureSelectionSeed	object numeric property	d	This value sets the feature selection random number seed.
fileLocation	object string property	d	Big Data: Enter the location of the Big Data file here.
fIn	object numeric property	d	Linear/Logistic Wrapping: Used when method = FORWARD_SELECTION or STEPWISE_SELECTION A statistic is calculated when variables are added or eliminated.
finalValue	result property	o/s/c	Creates an empty array to hold the final constraint value for the constraint or constraint block in an optimization or stochastic optimization model.
fitIntercept	object bool property	d	Linear Regression: When this option is set to true, the default setting, the linear regression intercept will be fit.
fittedModel	section	d	This section is similar to "datasets" but rather than refining imported data, this section defines a model that you can bind to when performing an "action" such as "forecast", "predict", "fit" or "transform". Used when scoring a model, this property is used to reference the model generated inside of the "model" object.
flow/flowName	top property	w	Assigns a name to the decision flow. Unnamed decision flows are not supported.
For	statement keyword	o/s/c/d	for-loop (or simply for loop) is a control flow statement for specifying iteration , which allows code to be executed repeatedly.
formula	object property		Enter a formula to calculate a result or array which will be used in a constraint, uncertain function or in the objective function.
Formulas	section	o/s/c	This optional section can be used to perform calculations on data arrays or constant values which will be used in a constraint, objective function or uncertain function definition.

fOut	object numeric property	d	For use when method = BACKWARD_ELIMINATION OR STEPWISE_SELECTION. A statistic is calculated when variables are eliminated.
groupingVariables	object array property	d	Big Data: Use this option to specify a grouping variable(s).
header headerExists	datasource property	o/s/c/d	Indicates whether the columns contain titles, or headers, or not and is set to True by default.
hiddenLayerActivation	object string property	d	Neural Networks: Determines how the weighted sum is adjusted during the data science process.
identLeaves	sensitivity property	o/s/c	Use this property to create a parameter sensitivity plot.
identParams	sensitivity property	o/s/c	equivalent of ident leaves in Excel sensitivity
If	statement keyword	o/s/c/d	The IF statement evaluates the test expression inside the parenthesis ().
ignoreLock	modelSettings property	o/s	If true, all PsiLock(val) properties in all formulas are ignored.
imputation	object array property	d	In Estimator section: Set transform: imputation to transform missing values. For Imputation For Transformation algorithm: Determines the strategy of the selected variables.
imputationStrategy	object array property	d	Determines the strategy of the selected variables.
In	statement keyword	o/s/c/d	Use in a for loop(), i.e. for (p in 1..5). See the RASON Example, UGProductMix5. (RASON Examples – <i>Example models discussed in RASON User Guide – UGProductMix5.json</i>)
includeTies	object bool property	d	kNearestNeighbors Classification or Regression: If includeTies = True, all points with distance equal to kth nearest neighbor are included in the result. If includeTies = False, exactly k nearest neighbors are returned.
indexCols	object property	o/s/c/d	DataSources: Used in conjunction with valueCols. Use this property to index by dimension(s).
indexSets	section	o/s/c	RASON uses index sets exclusively to dimension tables and arrays.
indexValue	result property	o/s/c	Creates an empty array to hold the indexValue of the constraint, variable,

			uncertainFunction or uncertainVariable
initialValue	result property	o	Creates an empty array to hold the initial value for the objective function, variable or constraint.
inputDataType	object string property	d	The Hierarchical clustering method can be used on raw data as well as the data in Distance Matrix format. Pass the appropriate option to fit your dataset.
inputParameters	section	o/s/c/d	Section heading in a workflow used for specifying input parameters required for the reusable model.
Interval	object array property	d	Moving Average Smoothing: Use this parameter to enter the window width for the moving average smoothing method. Binning: Use this parameter to indicate whether the interval for each variable is CLOSED [], RIGHT_CLOSED [], or LEFT_CLOSED [].
invokeModel	top property	w	The name of the reusable model in a decision flow.
jobID	object string property	d	BigData Enter the ID of the previously submitted async job.
kurtosis (kurt)	result property	o/s	Creates an empty array to hold the kurtosis of the uncertain variable or uncertain function.
laplaceSmoothing	object bool property	d	Naïve Bayes: This parameter allows users to specify a small correction value, known as a pseudocount, so that no probability estimate is ever set to 0.
learningOrder	object string property	d	Neural Networks: This option sets the order in which the records in the training dataset are processed.
learningOrderSeed	object numeric property	d	Neural Networks: This option specifies the seed for shuffling the training records.
learningRate	object numeric property	d	Neural Networks: This option sets the multiplying factor for the error correction during backpropagation; it is roughly equivalent to the learning rate for the neural network.
levelParam	object numeric property	d	Smoothing Methods: Use this parameter to enter the smoothing parameter for exponential, double exponential and holt winters smoothing methods.
linkage	object string property	d	One of the simplest agglomerative hierarchical clustering methods is single linkage, also known as the nearest neighbor technique.

loop	statement keyword	o/s/c/d	a sequence of instructions that is continually repeated until a certain condition is reached.
lower (lowerBound)	object property	o	Specifies the lower bound of the variable or constraint.
lowerCensor	distribution property	o/s	Use this parameter to set a lower "censor" bound for values sampled from the probability for a specific Uncertain Variable.
lowerCutoff	distribution property	o/s	Use this parameter to set a lower cutoff for values sampled from the probability for a specific Uncertain Variable.
mapping	object array property	d	Category Reduction: Assigns a specific category number to single or multiple categories.
matrixMethod	object string property	d	Use this option to specify the method used to calculate the transformation matrix when computing Principal Components.
max	statement keyword	o/s/c/d	Creates an empty array to hold the maximum value of the uncertain variable or uncertain function.
maxDocumentFrequency	object numeric property	d	Text Miner will remove terms that appear in more than the percentage of documents specified.
maximum (max)	result property	o/s	Creates an empty array to hold the maximum value of the uncertain variable or uncertain function.
maxIterations	object numeric property	d	ARIMA, Logistic Regression, kMeans Clustering: Sets the maximum number of iterations.
maxLag	object numeric property	d	Lag Analysis: Sets the maximum number of lags.
maxNumConcepts	object numeric property	d	LSA: Use this parameter to specify the maximum number of concepts in the Scree Plot.
maxNumEpochsWithNoImprovement	object numeric property	d	Neural Networks: The algorithm will stop after this number of epochs has been completed and no improvement has been realized.
maxNumLevels	object numeric property	d	This option specifies the maximum number of levels in the decision tree.
maxNumLevelsTreeDiagram	object numeric property	d	This option specifies the maximum number of levels in the decision tree to be included in the output.
maxNumNodes	object numeric property	d	This option specifies the maximum number of nodes in the decision tree.

maxNumSplits	object numeric property	d	This option specifies the maximum number of splits in the decision tree.
maxNumSubsetsExhaustive	object numeric property	d	Linear/Logistic Wrapping: For use when method = EXHAUSTIVE_SEARCH.
maxSubsetSize	object numeric property	d	Linear/Logistic Wrapping: Enter an integer from 1 up to N where N is the number of variables (features) in the model.
maxTermLength	object numeric property	d	Text Miner will remove terms that contain this number of characters.
maxTime	engineSettings property	o/s	The value for Max Time determines the maximum time in seconds that the Solver Engine will run before it stops.
maxTrainingTimeSeconds	object numeric property	d	Neural Networks: The algorithm will stop once this time (in seconds) has been exceeded.
maxVocabulary	object numeric property	d	This parameter reduces the number of terms in the final vocabulary of the Text Mining model to the most frequently occurring in the collection.
Mean	result property	o/s	Creates an empty array to hold the mean (or average) of the uncertain variable or uncertain function.
Median	result property	o/s	Creates an empty array to hold the median of the uncertain function or uncertain variable.
Method	object array property	d	Estimator or WeakLearner: Use this property to set parameter values or turn parameters on or off using "true" or "false".
Metric	object string property	d	Use this parameter to compute the desired metric for Feature Selection.
Min	statement keyword	o/s/c/d	Creates an empty array to the minimum value of the uncertain variable or uncertain function.
minConfidence	object numeric property	d	Association Rules: A value entered for this option specifies the minimum confidence threshold for rule generation.
minDocumentFrequency	object numeric property	d	Text Miner will remove terms that appear in less than the percentage of documents specified.
minimum (min)	result property	o/s	Creates an empty array to the minimum value of the uncertain variable or uncertain function.
minLag	object numeric property	d	Lag Analysis: Sets the minimum number of lags.

minNumRecordsInLeaves	object numeric property	d	This option specifies the minimum , number of records allowed in terminal nodes, or leaves of the decision tree.
minPercentExplained	object numeric property	d	Text Mining: Identifies the concepts with singular values that, when taken together, sum to the minimum percentage explained, 90% is the default.
minRelativeErrorChange	object numeric property	d	Text Mining: If the relative change in error is less than this value, the algorithm will stop.
minRelativeErrorChangeComparedToNullModel	object numeric property	d	Text Mining: If the relative change in error compared to the Null Model is less than this value, the algorithm will stop.
minStemmedTermLength	object numeric property	d	Text Mining: If stemming reduced a term's length to 2 or less characters, Text Miner will disregard the term.
minSupport	object numeric property	d	Specify the minimum number of transactions in which a particular item-set must appear for this set to qualify for inclusion in an association rule here.
Mode	result property	o/s	Creates an empty array to hold the mode of the uncertain variable or uncertain function.
modelDescription	top property	o/s/c/d/w	Section header used to add a text string containing the description of the model (optional).
modelName	top property	o/s/c/d	Section heading; Use modelName to assign a name to the RASON model, optional.
modelRecency	top property	o/s/c/d	Property specifying how recent the model has been run.
modelSettings	section	o/s/c	In this optional section, you may specify options relevant to the solve such as the number of optimizations or simulations to run, the number of trials to perform in a simulation model, what Stochastic Transformation method to use when solving a stochastic optimization model, if a nonsmooth model should be transformed, etc.
modelType	top property	o/s/c/d/w	The model type: optimization, simulation, calculation, data science or decision flow.
moneyToken	object string property	d	Text Mining: If NORMALIZE_MONEY is passed for "preprocessing", numbers appearing in the document collection will be replaced with the term, "numbertoken".
movingAverageOrder	object numeric property	d	ARIMA: Sets the non-seasonal Moving Average parameter (d).

name	object property		Use this property to enter a name for a decision variable, constraint, objective, uncertain variable or uncertain function.
nonSeasonalLag	object numeric property	d	Sets the nonseasonal lag for Lag Analysis.
normType	object string property	d	If technique = "UNIT_NORMALIZATION", use "normType" to set the normalization type. "L1" normalizes the data using the Manhattan Distance (L1-norm) while "L2" uses the Euclidean Length (L2-norm).
numberToken	object string property	d	Text Mining: If NORMALIZE_NUMBER is passed for "processing", then this term may be used to replace numbers appearing in the document collection with the term entered here.
numBins	object array property	d	Enter the number of desired bins for each selected variable using this option.
numBinsFeatures	object numeric property	d	Univariate for Transformation: Sets the maximum number of bins for the input variables.
numBinsTarget	object numeric property	d	Univariate for Transformation: Sets the maximum number of bins for the target (output) variable.
numCategories	object array property	d	Category Reduction: Use this property to reduce the number of total categories in a data set.
numClusters	object numeric property	d	Hierarchical Clustering: This option lets you stop the process at a given number of clusters.
numEpochs	object numeric property	d	Returns the number of epochs performed in a Neural Network.
numForecasts	object numeric property	d	Smoothing Methods and Time Series: Sets the number of forecasts.
numIterations	object numeric property	d	Logistic Regression: Sets the number of iterations. ARIMA: Returns the number of iterations completed.
numNeighbors	object numeric property	d	Classification and Regression; This is the parameter k in the k-Nearest Neighbor algorithm.
numNeurons	object array property	d	Use this open to specify the number of neurons in the Neural Network Architecture i.e. the number of neurons in the hidden layer(s).
numOptimizations	modelSettings property	o	Use this property to set the number of optimizations to run.

numPrincipalComponents	object numeric property	d	PCA for Transformation: Use either numPrincipalComponents to select the number of principal components displayed in the output or varianceCutoff, but not both.
numSelectedFeatures	object numeric property	d	The Random Trees ensemble method works by training multiple “weak” classification trees using a fixed number of randomly selected features then taking the mode of each class to create a “strong” classifier.
numSimulations	modelSettings property	s	The specific simulation that will be performed if multiple simulations are defined.
numStarts	object numeric property	d	K Means Clustering; Enter the number of desired starting points for the clustering algorithm.
numThreads	modelSettings property	o/s	Determines the number of virtual cores to be used to solve the model.
numTopFeatures	object numeric property	d	Linear/Logistic Wrapping: Enter a value ranging from 1 to the number of features in the model.
numTrials	modelSettings property	o/s	Use this property to set the number of Monte Carlo trials to run in each simulation
numWeakLearners	object numeric property	d	This option controls the number of “weak” classification/regression models that will be created.
objective	section	o	This optional section is used for defining a normal, expected, or chance objective function in an optimization, stochastic optimization or simulation optimization model.
optimize	object bool property	d	Select this option to select the Alpha parameter for the Exponential smoothing method, the Alpha and Beta parameters for the Double Exponential Smoothing method, and the Alpha, Beta and Gamma parameters for the Holt Winter Smoothing method to minimize the residual mean squared errors in the training and validation sets.
outputLayerActivation	object string property	d	Neural Networks: The output layer is also computed using the same transfer function as described for setHiddenLayerActivation
outputResults	section	w	Results from a decision flow.
parameters (params)	section	o/s/c	In WeakLearner and Estimator sections, use this property to set parameter values or turn parameters on or off using "true" or "false".
partition	object string property	d	Specifies the partition to transform.
partitionMethod	object string property	d	Use this option when partitioning the dataset using a partition variable to specify random or sequential.

partitionVariable	object array property	d	When "partitionMethod" = "Manual", the partition variable specified is used to partition the dataset which serves as a flag for writing each observation to the appropriate partition(s).
percentiles	result property	o/s	Creates an empty array to hold the percentiles of the uncertain variable or uncertain function.
phraseReplacement	object array property	d	Use this parameter to combine words into phrases that indicate a singular meaning such as "station wagon" which refers to a specific type of car rather than two distinct tokens – station and wagon.
placeholder	object array property	d	Text Mining: Sets the custom placeholder for missing values in a column.
plotParams	sensitivity property	o/s/c	Equivalent of param plot in Excel sensitivity.
preprocessing	object array property	o/d	Text Mining: Use this property to replace or remove nonsensical terms such as HTML tags, URLs, Email addresses, etc. from the document collection. Optimization: When this parameter is set, the LP/Quadratic performs preprocessing on constraints involving integer variables, to simplify the problem and speed up the solution process.
preProcessor	section	o/s/c/d	This optional section may be used for preliminarily data preparation or to compute values of some properties, which are passed later, at parse-time, to the RASON DM engine.
priorProb	object array property	d	For classification models only. Specifies the desired class and probability values
priorProbMethod	object string property	d	For classification models only. Use EMPIRICAL when the probability of encountering a particular class in the dataset is the same as the frequency with which it occurs in the training data. Use UNIFORM when all classes occur with equal probability. Use MANUAL to enter the desired class and probability. See the example to the left.
prunedTreeType	object string property	d	Use this option to select the tree used to score the validation dataset: FULL_GROWN, BEST_PRUNED, MIN_ERROR or MANUAL.
prunedTreeNumDecisionNodes	object numeric property	d	Decision Trees: Used in conjunction with "prunedTree Type" : "MANUAL". Use this option to specify the number of decision nodes in the pruned tree.

query	object string property	d	SQL Transformation: Use this property to specify a query string. See the Example section within the OData Service for RASON Decision Flows chapter.
randomGenerator	engineSettings property	o/s	Equivalent to using the Excel interpreter when solving a model in Microsoft Excel.
randomSeed	engineSettings property	o/s/d	<p>k Means Clustering: This option initializes the random number generator that is used to assign the initial cluster centroids.</p> <p>Big Data Sampling: Sets the desired sorting seed here.</p> <p>Simulation: Setting the random number seed to a nonzero value (any number of your choice is OK) ensures that the <i>same</i> sequence of random numbers is used for each simulation.</p>
randomStreams	engineSettings property	o/s	Equivalent to using the Excel interpreter when solving a model in Microsoft Excel.
rank	object array property	d	<p>Binning: This parameter is available when binValueOption is set to "RANK". Use the "rank" parameter to specify the <i>Start</i> value of the first bin and the <i>Interval</i> of each bin. Subsequent bin values will be calculated as the previous bin + interval value.</p>
ratios	object array property	d	Partitioning: Specify the percentages for the training set, validation set and test sets.
recourse	variable property	o	Defines the variable or variable block as recourse variable(s) in a stochastic optimization model.
replaceOption	object bool property	d	Sampling/Stratified Sampling: Set this option to "true" to sample with replacement.
resamplingSeed	object numeric property	d	For Classification models only. This value specifies the seed for random resampling of the training data for each weak learner.
responseCorrection	object numeric property	d	This option specifies the value applied to the Normalization rescaling formula, if the output layer activation is Sigmoid (or Softmax in Classification) or Adjusted Normalization, if the output layer activation is Hyperbolic Tangent.
robustCounterpart	modelSettings property	o	Use this option to determine how an optimization model with uncertainty will be solved.
rowIndex	datasource property	o/s/c/d	Use this property to create an implicit index set consisting of integer numbers from 1 to the number of rows.
rowNames	object array property	d	Assigns row names.
runSpecificSim	modelSettings property	s	When running multiple simulations, use this parameter to select a specific simulation to be solved.

sampleFraction	object numeric property	d	This is the expected size of the sample as a fraction of the dataset's size.
sampleSize	object numeric property	d	Sampling for Big Data: Sets the desired sample size here.
samplingType	object string property	d	Sampling for Big Data: Determines how the size of the sample will be determined; select Approximate or Exact.
seasonalAutoRegressiveOrder	object numeric property	d	ARIMA: Sets the seasonal Autoregressive parameter (P).
seasonalDifference	object numeric property	d	ARIMA: Sets the seasonal Difference parameter (D).
seasonalityParam	object numeric property	d	The Holt Winters Smoothing technique utilizes this parameter to manage the presence of seasonality in the data.
seasonalLag	object numeric property	d	ARIMA: Sets the seasonal lag.
seasonalMovingAverageOrder	object numeric property	d	ARIMA: Sets the seasonal Moving Average parameter (Q).
seed	object numeric property	d	Oversampling: Random partitioning uses the system clock as a default to initialize the random number seed.
selectedCols	object array property	d	Selects specified columns. (Array property)
selectedVariables	object array property	d	Big Data: Variables passed to this parameter will be included in the sample.
selection	datasource property	o/s/c/d	Use this property to select the columns/fields to import.
sensitivityPoints	modelSettings property	d	Use this property to specify the axis points for the first listed PsiOptParam, PsiSenParam or PsiSimParam when varying two parameters independently or simultaneously.
sensitivityPoints2	modelSettings property	d	Use this property to specify the axis points for the second listed PsiOptParam, PsiSenParam or PsiSimParam when varying two parameters independently.
simulationOptimization	modelSettings property	o	Use this option to solve an optimization model with uncertainty using simulation optimization.

skewness (skew)	result property	o/s	Creates an empty array to hold the skewness of the uncertain function or uncertain variable.
slackValue	result property	o	Creates an empty array to hold the slack value for each optimization constraint.
smoothingAlpha	object numeric property	d	Naïve Bayes Smoothing and Classification: Setting this option to zero is equivalent to no smoothing.
sortIndexCols (sort)	datasource property	o/s/c	Use this property to sort the columns alphabetically.
sortIndexes	object bool property	d	When this option is set to "true", the data is sorted using the simple random sampling technique, taking into account the additional parameter settings.
sparkServer	object string property	d	Big Data: Use this option to enter the endpoint for the Apache Spark cluster.
stable	object bool property	d	K Nearest Neighbors Classification and Regression: If stable = true, the tied neighbors (up to kth neighbor) remain in the original order. If stable = false, the tied neighbors (up to kth neighbor) are in pseudo-random order.
startPhrase	object string property	d	Text Mining: Text appearing before the first occurrence of the Start Phrase (if used) will be disregarded and similarly, text appearing after End Phrase (if used) will be disregarded.
stdev	result property	o/s	Creates an empty array to hold the standard deviation of the uncertain function or uncertain variable.
stepSize	object numeric property	d	For Regression models only. The Adaboost algorithm minimizes a loss function using the gradient descent method. The Step size parameter is used to ensure that the algorithm does not descend too far when moving to the next step.
sterr	result property	o/s	Creates an empty array to hold the standard error of the uncertain function or uncertain variable.
stopwordsExtraTerms	object array property	d	When used, over 300 commonly used words/terms (such as a, to, the and, etc.) will be removed from the document collection during preprocessing.
strataCol	object string property	d	For use with Datasets section; Selects the stratum variable when performing stratified random sampling. (String property)
stratificationMethod	object string property	d	Determine the method used (proportional or equal size) when performing sampling.
stratumSampleSize	object numeric property	d	Sampling: Sets desired stratum sample size.

stringZero	modelSettings property	o/s/c/d	If True, strings in arithmetic operations are considered to be "0".
successClass	object string property	d	For classification models only. Select the success value for the output variable here (i.e. 0 or 1 or "yes" or "no").
successProbability	object numeric property	d	For classification models only. Enter a value between 0 and 1 for this option to denote the cutoff probability for success.
successRatioInTraining	object numeric property	d	Sets the percentage of successes to be assigned to the training set in a data science model.
sum	statement keyword	o/s/c/d	Keyword adds a series of values.
synonyms	object property	d	Use this parameter to replace synonyms such as "car", "automobile", "convertible", "vehicle", "sedan", "coupe", "subcompact", and "jeep" with "auto". During pre-processing, Text Miner will replace the terms "car", "automobile", "convertible", "vehicle", "sedan", "coupe", "subcompact" and "jeep" with the term "auto".
targetCol	object string property	d	Data Science: Use this property to pass the name of the output column. (String property)
technique	object string property	d	Rescaler: Select either standardization or normalization for feature scaling.
testRatioFromValidation	object numeric property	d	Partitioning: If a test set is desired, specify the percentage of the validation set that should be allocated to the test set using this option.
then	statement keyword	o/s/c/d	If the condition is true, the statements following the then are executed. Otherwise, the execution continues in the following branch – either in the else block or if there is no else branch, then after the end If.
timeVariable	object string property	d	Time Series: Selects the Time variable in a time series dataset. (String property)
trackRowID	object bool property	d	If this option is set to "true", data records in the resulting sample will carry the ordinal IDs corresponding to the original data records.
trainData	object string property	d	This property may be used interchangeably with the property, "data". In some algorithms, it is possible to provide both "trainData" and "validData" i.e. for classification and regression algorithms.
transformer	section	d	Data Science - The "transformer" section is used to differentiate the algorithms that do not have a model, i.e. they do not implement the "fit"

			interface or extract any type of model. Rather, these algorithms implement the "transform" interface (only) by operating directly on the data to produce transformed data which can serve as input to other data science methods.
transformNonSmooth	modelSettings property	o	Use this option to choose whether Solver will attempt to transform constraints in your model that are <i>non-smooth</i> functions of the decision variables into equivalent linear constraints that depend on newly introduced binary integer and continuous decision variables.
transformStochastic	modelSettings property	o	Use this option to determine how an optimization model with uncertainty will be solved. Your optimization includes uncertainty if the formula for the objective, or any constraint, depends (directly or indirectly) on an uncertain variable cell, where you've used a PSI distribution function (such as PsiNormal). Stochastic Transformation works only with <i>linear</i> models that include uncertainty; it uses either stochastic programming or robust optimization methods to solve the problem (see the Transformation options for further information).
trendParam	object numeric property	d	The Double Exponential and Holt Winters smoothing techniques include this parameter to contend with trends in the data.
trials	result property	o/s	Creates an empty array to hold the trial values of the uncertain variable and uncertain function.
type	object property		Define the variable type, i.e. integer, binary, alldiff, real, diff or sem.
uncertainFunctions	section	s	This section, required in a simulation model, is where the uncertain functions are defined.
uncertainVariables	section	o/s	The "uncertain variables" section is required in a simulation, simulation optimization or stochastic optimization model. Here you will define an uncertain variable or block of uncertain variables using a PSI distribution such as PSILogNormal or PSITriangular.
upper (upperBound)	object property	o	Specifies the upper bound of the variable (variable block) or constraint (constraint block).
upperCensor	distribution property	o/s	Use this parameter to set an upper "censor" bound for values sampled from the probability for a specific Uncertain Variable.
upperCutoff	distribution property	o/s	Use this parameter to set an upper cutoff for values sampled from the probability for a specific Uncertain Variable.
urlToken	object string property	d	If NORMALIZE_URL is passed for "preprocessing", URLs appearing in

			the document collection will be replaced with the term, "urlToken".
useCorrelations	modelSettings property	o/s	Set this property to True in order to use a correlation matrix in your simulation, simulation optimization or stochastic optimization model.
usePvalueForSelection	object bool property	d	If "True", then the variables will be ranked from smallest to largest using the P-value of the measure or statistic selected.
validData	object string property	d	This property may be used interchangeably with the property, "data". In some algorithms, it is possible to provide both "trainData" and "validData" i.e. for classification and regression algorithms.
value	object property	o/s/c/d	Sets the values of a table or array.
valueCol	data property	o/s/c/d	Used with binding property to bind imported values from a readable data source.
valueCols	object property	o/s/c/d	Used in conjunction with indexCols. Use this property to import columns/fields containing values
variables	section	o	A required section within an optimization model, where the decision and recourse variables (if used) are defined.
variance (var)	result property	o/s	Creates an empty array to hold the standard deviation of the uncertain function or uncertain variable.
varianceCutoff	object numeric property	d	PCA: Use either numPrincipalComponents to select the number of principal components displayed in the output or varianceCutoff, but not both.
weakLearner	section	d	This section is used (only) to specify a weak learner in a bagging or boosting estimator algorithm.
weightCol	object string property	d	Logistic Regression: Selects a weight variable which allows the user to allocate a weight to each record.
weightDecay	object numeric property	d	Neural Networks: Use this option to prevent over-fitting of the network on the training data.
weightingScheme	object string property	d	k Nearest Neighbors Classification or Regression: Use this option to select the weighting scheme: equal or inverse
weightingSchemeDocument	object string property	d	Using these three options, users can select their own choices for local weighting, global weighting and normalization.
weightingSchemeNormalization	object string property	d	
weightingSchemeTerm	object string property	d	

weightInitSeed	object numeric property	d	Neural Networks: Use this option to initialize the Random Seed for Weights Initialization.
weightMomentum	object numeric property	d	This option controls the weight change momentum in the neural network algorithm.
weights	object array property	d	Logistic Regression: Provides a weight variable allowing the user to allocate a weight to each record.
while	statement keyword	o/s/c/d	A statement that allows code to be executed repeatedly based on a given condition.
withReplacement	object bool property	d	Sampling: If this option is set to "true" the data will be sampled with replacement.
worksheets	modelSettings property	o/s/c/d	Use this URL parameter to specify the worksheet of the desired Excel model.

RASON Organization Accounts

Introduction

RASON Organization Accounts

RASON Organization Accounts are useful for organizations desiring the highest degree of isolation and security for their models and data, and for organizations with multiple model builders who need to collaborate on a variety of models. Organization Accounts take advantage of the Microsoft Identity Platform, Azure Storage Accounts, and Role-Based Access Control (RBAC) to enable IT staff and central analytics groups to better manage model development, deployment for production use, and monitoring.

RASON without Organization Accounts

In “regular” use of RASON (without an Organization Account), each user account is independent of, and isolated from every other user account; this independence is enforced by the RASON Server, which acts as the “master identity” using Azure resources whenever a model is created, updated, solved or deleted. Since user accounts are isolated, users cannot see other user’s models, or collaborate on the same model. Users log in to RASON.com with an email address and password, or using their “Microsoft accounts”.

All user models are persisted in Frontline’s Azure Storage Account, which Microsoft bills to Frontline Systems; each user is given a unique Azure “container” which holds all of his/her models and data. Models are solved (and other operations are performed) using Frontline’s Azure virtual CPUs or “App Services”, which Microsoft bills to Frontline Systems. The cluster of CPUs is “auto-scaled”, increasing or decreasing based on demand from all RASON users that varies over time.

RASON with Organization Accounts

When a company uses a RASON Organization Account, each RASON user within the company “belongs” to the Organization Account; users must login to RASON.com using the Microsoft Identity Platform (they simply click a button to do this) and are authenticated using the company’s own Azure Active Directory, i.e. their “work or school account”.

With an Organization Account, the RASON Server still provides the virtual CPUs or “App Services”, which Microsoft bills to Frontline Systems. But the company’s models and data are maintained privately in the company’s own Azure Storage Account (so Microsoft bills the company for storage use), isolated from all other RASON users. Within this Azure Storage Account, each user will have a unique “container” for their personal models and data (automatically created by RASON), but the

company's IT staff or central analytics group can create additional Azure containers for models and data that should be shared among users for collaboration purposes, treated as "development" or "production", etc.

Further, using standard Azure Role-Based Access Control (RBAC), the company can assign "roles" to different users that give them selective access to these containers and their models and data. Users will then be able to "see" the containers they can access (read-only or read/write) in the RASON Model Editor, and easily create, update or delete models within those containers in accordance with their roles. Roles may be assigned using Azure Portal, Azure Powershell, or even Azure CLI and REST endpoints.

Role-Based Access Control is enforced every time a RASON model is created, updated, solved or deleted. For example, when models are solved, or results are accessed from a running application, the request includes an Authorization header with an API token, created in RASON.com by a properly authenticated user. On each new REST API request, the RASON Server will access the chosen Azure Storage Account, using the Azure identity (and limited access rights) of the user and role used to create the API token.

Instructions for Administrators

When using a RASON Organizational Account, users will login using their Microsoft account and RASON will access the organization's Azure Storage on behalf of the user. As a result, user's must be given the appropriate permissions. Click [here](#) to access Azure help.

Note: The RASON models and data are stored on your Azure storage account. Frontline can read and write to any of those models provided that 1. the user is logged in and 2. user has been given permissions to do so. However, once the user has been granted permission, Frontline does maintain an access token so the user is not prompted to log in and grant permissions every time. This token is encrypted at rest and in motion.

Preparing Azure Storage for use with your RASON Organization Account

Prerequisites:

You'll need to either use an existing storage account that is already part of a resource group, or create new Azure Storage Account and add it to a new or existing Azure Resource Group containing a storage account

- Azure storage Account. If you need to create the storage account, please see <https://docs.microsoft.com/en-us/azure/storage/common/storage-account-create?toc=%2Fazure%2Fstorage%2Fblobs%2Ftoc.json&tabs=azure-portal>
- Azure Resource Group containing the storage account. To create a resource group and add resources, see <https://docs.microsoft.com/en-us/azure/azure-resource-manager/management/manage-resource-groups-portal>

Required User Permissions

Because RASON organizational account users access your organizations storage on behalf of the user, those users will require certain permissions based on the needs of the user. All access is controlled using Azure RBAC, groups roles and permissions. How these permissions and roles are assigned is dictated by the security practices of the organization.

All users will need the following permissions:

```
"actions": [
```



```

    "Microsoft.Authorization/roleAssignments/read",
    "Microsoft.Authorization/roleDefinitions/read"
  ],

```

Rason uses these permissions to determine the users access level to containers and blobs in the storage account. From here, permissions can be fine tuned and edited to meet the security needs of your organization. For example, users who require the ability to read/write/update/delete models as well as move models to other storage blob containers would require the following permissions:

```

  "actions": [
    "Microsoft.Storage/storageAccounts/blobServices/containers/read",
    "Microsoft.Storage/storageAccounts/blobServices/containers/write",
    "Microsoft.Storage/storageAccounts/blobServices/containers/delete"
  ],
  "dataActions": [
    "Microsoft.Storage/storageAccounts/blobServices/containers/blobs/delete",
    "Microsoft.Storage/storageAccounts/blobServices/containers/blobs/read",
    "Microsoft.Storage/storageAccounts/blobServices/containers/blobs/write",
    "Microsoft.Storage/storageAccounts/blobServices/containers/blobs/add/action"
  ],

```

And users requiring only read access to the RASON models would require the following permissions:

```

  "actions": [
    "Microsoft.Storage/storageAccounts/blobServices/containers/read"
  ],
  "dataActions": [
    "Microsoft.Storage/storageAccounts/blobServices/containers/blobs/read"
  ],

```

For example. To set up a new azure storage account and grant the necessary permissions one could perform the following actions:

1. in the Azure portal:
 - a. Create a resource group named RASON Resource Group.
 - b. Create a storage account named rason-storage.
 - c. Create the following roles with permissions.
 - i. RASON Model Role Reader


```

{
  "id": "/subscriptions/<your subscription id>/providers/Microsoft.Authorization/roleDefinitions/<role id>",
  "properties": {
    "roleName": "RASON Model Role Reader",
    "description": "Read resource group role assignments and role definitions",
    "assignableScopes": [
      "/subscriptions/<your subscription id>/resourceGroups/<your resource group name>"
    ],
    "permissions": [
      {
        "actions": [
          "Microsoft.Authorization/roleAssignments/read",
          "Microsoft.Authorization/roleDefinitions/read"
        ],
        "notActions": [],

```

```

        "dataActions": [],
        "notDataActions": []
    }
}
}
}
ii. RASON Model Contributor
{
    "id": "/subscriptions/<your subscription id>/
    providers/Microsoft.Authorization/roleDefinitions/
    <role id>",
    "properties": {
        "roleName": "RASON Model Contributor",
        "description": "Allow read/write/delete actions on the
        control plane. Allow read/write/add/delete on the data
        plane",
        "assignableScopes": [
            "/subscriptions/<your subscription id>/
            resourceGroups/<your resource group name>/
            providers/Microsoft.Storage/storageAccounts/
            <your storage name>"
        ],
        "permissions": [
            {
                "actions": [
                    "Microsoft.Storage/storageAccounts/
                    blobServices/containers/read",
                    "Microsoft.Storage/storageAccounts/
                    blobServices/containers/write",
                    "Microsoft.Storage/storageAccounts/
                    blobServices/containers/delete"
                ],
                "notActions": [],
                "dataActions": [
                    "Microsoft.Storage/storageAccounts/
                    blobServices/containers/blobs/delete",
                    "Microsoft.Storage/storageAccounts/
                    blobServices/containers/blobs/read",
                    "Microsoft.Storage/storageAccounts/
                    blobServices/containers/blobs/write",
                    "Microsoft.Storage/storageAccounts/
                    blobServices/containers/blobs/add/action"
                ],
                "notDataActions": []
            }
        ]
    }
}
iii. RASON Model Reader
{
    "id": "/subscriptions/<your subscription id>/
    providers/Microsoft.Authorization/roleDefinitions/
    <role id>",
    "properties": {
        "roleName": "RASON Model Reader",
        "description": "",

```

```

"assignableScopes": [
  "/subscriptions/<your subscription id>/
  resourceGroups/ResourceGroup1/providers/
  Microsoft.Storage/storageAccounts/
  rasonstoragehotmail"
],
"permissions": [
  {
    "actions": [
      "Microsoft.Storage/storageAccounts/
      blobServices/containers/read"
    ],
    "notActions": [],
    "dataActions": [
      "Microsoft.Storage/storageAccounts/
      blobServices/containers/blobs/read"
    ],
    "notDataActions": []
  }
]
}

```

- d. Create the following Groups
 - i. RASON Role Readers Group
 - ii. RASON Model Contributor Group
 - iii. RASON Model Readers Group
- e. Add both the RASON Model Contributor Group and RASON Model Readers Group to the RASON Role Readers Group
- f. Add users to either the RASON Model Contributor Group or the RASON Model Readers Group, depending on their needs.
- g. Navigate to the RASON Resource Group and add the “RASON Model Role Reader” to the role assignments of the RASON Resource Group. When adding the assignment, add “RASON Role Readers Group” as a member.
- h. Navigate to the rason-storage storage account and add both the RASON Model Contributor and RASON Model Reader roles to the account role assignments. When adding the assignments, add “RASON Model Contributor Group” and “RASON Model Readers Group” as members, respectively.

Click [here](#) to access Azure help.

Assigned roles now affect the Models and Editor tab functionality. For example, if a User has only “read” access, then POST `rason.net/api/model` and PUT `rason.net/api/model/id` will be disabled on the Editor tab and error 403 will be returned if calling either endpoint directly.

See the list below to see how permissions will affect the Editor page functionality and the models appearing in the Models lists (on the Models tab).

Editor Tab

- Read Only User
 - Does not have access to the “my models” container
 - POST and PUT menu items are disabled.
- Container without Write Permission
 - POST to {container name} and PUT to {container name} menu items disabled.
- Container without Delete Permission
 - Models list container “Delete All...” and “Delete” context menu items disabled.
- Current model in container without Write permission

- Async solve menu items disabled.
- Score button disabled.
- Create and Delete run token buttons disabled.
- Create App menu disable.

Models Tab

- List Context Menu
 - “Open Model” is disabled if container does not have Read permission.
 - “Model Details” is disabled if container does not have Read permission.
- Detail Context Menu
 - “Open” is disabled if container does not have Read permission.

Note: If the query parameter "?container=<container_name>" is not passed to the API endpoints POST rason.net/api/model or POST rason.net/api/model/id/solvetype, RASON will create a personal container for each user. When the query parameter "?container=<container_name>" is passed to the API end point(s), RASON will attempt to POST the model to the specified container, <container_name>. Permissions to read/write to any container can be controlled in the portal. For more information on POSTing models to specific containers or to the user's personal container, see Instructions for End Users below.

A Note on Groups

Alternatively, an administrator can create a RASON “group” where roles are assigned to a group and members are then added to this group. This might be a more maintainable exercise than assigning a role to many different users. For example:

RASON Model Readers Group – This group might be a “read-only” group for users that would monitor model usage. The main focus for this group would most likely be the Models tab.

RASON Modelers Group – Model developers could be assigned to this group which would allow members to create, edit, delete and test models. Their main focus for this group would most likely be the Editor tab.

Any group with access to RASON Decision services must be assigned the RASON Model Role Reader role in order to determine the effective permissions. Using the example groups above, the RASON Model Readers Group might be assigned the RASON Model Reader role. Members belonging to this group would only have read access to all models. In contrast, the RASON Modelers Group could be assigned the RASON Model Contributor role since users of this group would need full access to models (read, write, add and delete).

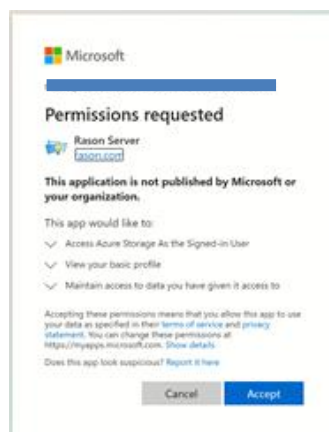
Administrators can refine these group permissions further to meet the security requirements of their organization. An example of one such refinement might be a RASON Model Executor, or a user who can read and create models but not delete and edit. (Create permission is necessary since executing a model creates a version of that model.)

Registering on www.RASON.com

1. If a new user account was created, visit www.RASON.com and register each new user. All storage account users must have an account on www.RASON.com.
2. Email Support (Support@solver.com) the email address for each user and the Storage account name so that all users may be added to the Organization Account. Please wait for confirmation from Support that all Users have been added to the Organization Account before continuing to Step 6.
3. RASON accesses the storage account on behalf of the user; therefore, each user must grant permission to RASON Services in order to do so. Therefore, end users must sign in using the "Microsoft Login" button on www.RASON.com in order to **Accept** the permission request.



Clicking Accept ensures that the RASON server is able to obtain a refresh token for the user's Azure account. This refresh token is then used to gain an access token. This only needs to be done one time. (However, if the user does not access the storage account for longer than six months, the user will need to login using the Microsoft button again.)



4. Once the user has been logged onto www.RASON.com click the Editor tab to begin posting models to the Azure storage containers.

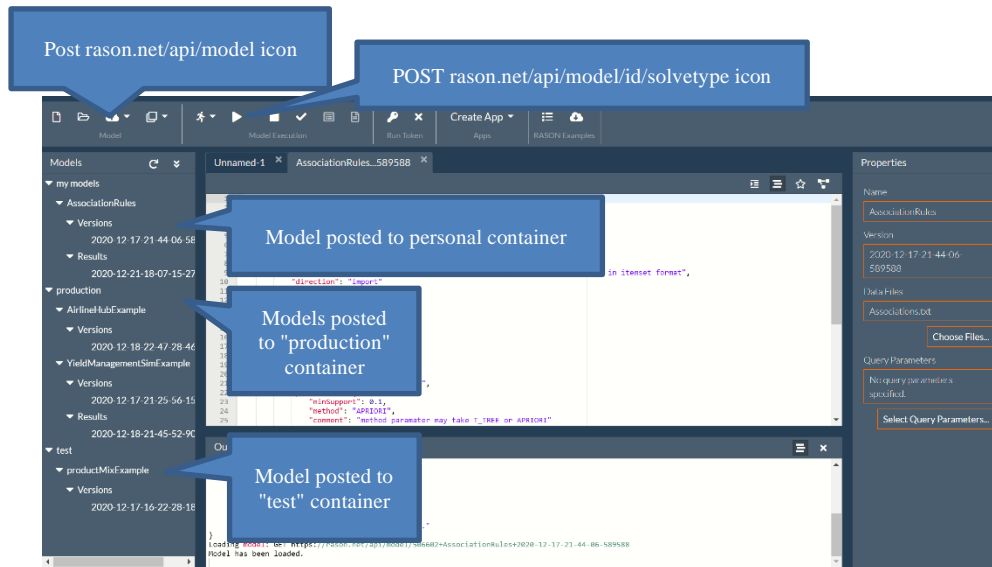
Instructions for End Users

Instructions for the end users of an organization account are provided below.

Using a RASON Organization Account

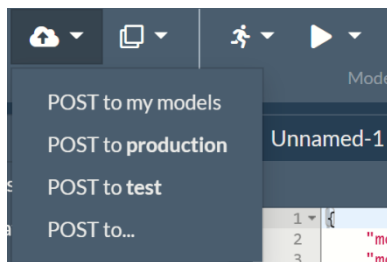
Click the Editor tab to easily Post new or modify existing models in your organization's Azure storage container(s). Storage containers, *containing RASON model(s) that the currently-logged-in user has permission to access*, are listed on the Models pane (on the left). Models posted to a specific container are listed under that container. Models posted to the user's "personal" storage container are listed under "my models". **See the list documented above to discover how specific permissions affect the functionality of the Editor tab.**

Note: Containers that do not contain a RASON model will NOT be displayed in the Models pane on the Editor tab.



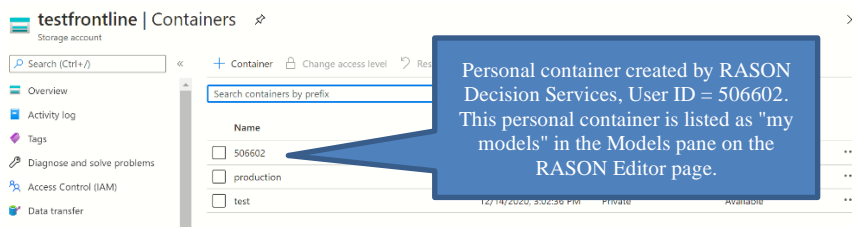
Posting a Model

To post a model to an Azure storage account, click the down arrow next to the POST icon and select the desired container where the model should be saved (POSTed). (A model may also be POSTed and solved by clicking the down arrow next to the POST rason.net/api/model/id/solvetype icon.) When a container is specified, RASON will attempt to use that container. Permissions to read/write to any container can be controlled in the portal. *See the list documented above to discover how specific permissions affect the functionality of the Editor tab.*



- Posting to "my models" will POST the model to the user's personal container. RASON will create a personal container for each user.

Organizations can change the permissions on the storage account to prohibit the creation of this new container, if desired, thus forcing all user models to be stored in the existing containers. (This must be done within the Azure portal.)



- Posting to <container_name> will save (POST) the model to the selected container. Organizations can limit access to containers within the Azure portal.

- Post to... allows users to spontaneously create a new container within the storage account or to POST a RASON model to an existing container. (Users must have permissions to create a new container or write to an existing container.) Simply click Post to... and then enter a name for the new or existing container.

All other REST API calls made on the Editor tab will remain the same. Solving the model is as simple as clicking POST `rason.net/api/model/{nameorid}/solve`, results can be obtained by clicking GET `rason.net/api/model/{nameorid}/result`, etc.

If calling the REST API from outside the Editor tab on www.RASON.com, the query parameter `"?container=<container_name>"` must be passed to all calls. See Working Outside the RASON Editor below for more information.

Working with Containers

Most tasks work exactly the same no matter if a user is part of an organization account or not. However, there are a few features within the RASON Editor that are accessible only when logged in as an organization account user.

Copying Models to Containers

To copy a model from one container to another container, simply open the model in its current position, i.e., click the model version to open in the Editor, then the down arrow next to POST `rason.net/api/model` and select the appropriate container or enter a new/existing container name.

Note: Container_name may be an existing container or a new container. Recall that if a new container name is provided for `<container_name>` a new container will be created on the user's storage account *as long as* the currently-logged-in user has permission to create a new container in the Azure storage account.

Properties Pane

The properties pane has an added field when logged in as an organization account user, Container. This field displays the current container where the model has been POSTed.

Properties

Container
archive

Name
PortfolioOptExample

Version
2021-01-06-23:39:09-371856

Data Files
No data files specified.
Choose Files...

Query Parameters
No query parameters specified.
Select Query Parameters...

Container where model is POSTed (saved).

Working with Decision Flows

When dragging a new Business Knowledge Model to an existing decision flow, a drop down menu will appear on the Properties pane allowing users to select a reusable model previously saved to the same container (as the decision flow).

Note: The decision flow and all reusable models used within the flow must be POSTed (saved) to the same container.

The screenshot displays the RASON Editor interface. On the left, the 'Models' pane shows a tree structure with 'production' selected, containing 'AirlineHubExample', 'OptSimReusable', and 'sim_stageReusable'. A red box highlights the 'production' container. A callout points to this box: 'Models/Decision flows contained within the "production" container.' In the center, the canvas shows a decision flow diagram with nodes 'sim_stageReusable', 'opt_stageReusable', and 'Business Knowledge Model-1'. A callout points to the 'production' container in the Models pane: 'Decision flow contained within the "production" container.' On the right, the 'Properties' pane shows the 'Invoke Model' dropdown menu with 'sim_stageReusable' selected. A callout points to this dropdown: 'Models/Decision flows contained within the "production" container.' Below the main screenshot, another screenshot shows the 'my models' container selected in the Models pane, with a callout: 'Models/Decision flows contained within the "my models" personal container.' The Properties pane dropdown now shows 'AssociationRules'. A third callout points to the 'my models' container: 'Models/Decision flows contained within the "my models" personal container.'

Working Outside of RASON Editor

When calling the REST API calls outside of the Editor tab on www.RASON.com, you must pass the "container" query parameter. See the examples below which POST a model to the "test" container.

Example 1

POST <https://rason.net/api/model?container=test>

Example 2

POST <https://rason.net/api/model/{nameorid}/solve?container=test>

POST <https://rason.net/api/model/YieldManagementSimExample/solve?container=test>

Example 3

GET <https://rason.net/api/model/{nameorid}/simulate?container=test>

GET <https://rason.net/api/model/YieldManagementSimExample/simulate?container=test>

Example 4

GET <https://rason.net/api/model/{nameorid}/result?container=test>

GET <https://rason.net/api/model/YieldManagementSimExample/result?container=test>

Appendix 1:

Insurance Auto Policy Model in CL3

Introduction

This appendix walks users through creating the DMN/FEEL Insurance Auto Policy Model for Conformance Level 3 in RASON Decision Services.

In practice, this model could exist as a desktop application or as website where a customer may enter information pertaining to him/her, his/her car(s) and co-drivers. This information might flow to a backend server running the RASON model, shown below.

Problem Definition

This example model is taken from the book DMN Method & Style: The Practitioners Guide to Decision Modeling with Business Rules. (Cody-Cassidy Press; 2nd edition (March 3, 2019))

1. This example first determines if a driver and their automobile qualifies for auto insurance based on a set of inputs. If so, a premium is calculated.
2. Additional drivers and automobiles may be added to a policy.
3. A driver is eligible for an auto policy if and only if all drivers and cars are eligible.
4. If a driver is eligible, all premiums of all participating drivers and cars are added together to form one total premium.

Building the RASON Model

This tutorial builds the model incrementally.

Define custom type definitions for the input data within the typeDefs section of the RASON model.

Component custom types are defined as custom types with components such as names, types and, optionally, allowed values (i.e. domain). If a component type is attached to a variable, then that variable defines values in its array structure according to the component type description. The variable may be referenced component-wise using the '.' operator.

This example includes 4 custom type definitions: tCar, tCarList, tDriver, and tDriverList.

The tCar type definition defines the input data type for a candidate vehicle. This type definition defines various properties relating to a vehicle.

```
tCar: {
  components: {
    makeModel: { typeRef: 'string' },
    modelYear: { typeRef: 'number' },
    style: {
      typeRef: 'string',
```

```

    allowedValues: ['compact','sedan','luxury']
  },
  price:      { typeRef: 'number' },
  isConvertible: { typeRef: 'boolean' },
  hasDriverAirbags: { typeRef: 'boolean' },
  hasPassenAirbags: { typeRef: 'boolean' },
  hasSideAirbags: { typeRef: 'boolean' },
  hasRollbar: { typeRef: 'boolean' },
  hasAlarm: { typeRef: 'boolean' },
  uninsMotorist: { typeRef: 'boolean' },
  medicalPolicy: { typeRef: 'boolean' }},
  language: 'Feel'
},

```

- Make and Model of the vehicle (makeModel) This is a string value such as "Subaru Output".
- Vehicle year (modelYear) This is a year such as "2013".
- Style of vehicle : This component allows three values: compact, sedan or luxury. (style)
- Price of the vehicle (price) This is a number, such as "25000".
- If the automobile is a convertible or not (isConvertible) Boolean
- If the vehicle has driver-side airbags or not (hasDriverAirbags) Boolean
- If the vehicle has passenger-side airbags or not (hasPassenAirbags) Boolean
- If the vehicle has side airbags or not (sideAirbags) Boolean
- If the vehicle has a roll bar (hasRollbar) Boolean
- If the vehicle has a security alarm (has Alarm) Boolean
- If the driver is uninsured (uninsMotorist) Boolean
- If the driver holds a medical policy (medicalPolicy) Boolean

The second custom type definition is tCarList.

```
tCarList: { typeRef: 'tCar', isCollection: true, language: 'Feel' },
```

While tCar models a single record for one driver, tCarList defines a collection of tCar.

The third custom type definition is tDriver. This type definition defines various components that relate to a prospective insured driver such as nae, age, gender, etc.

```

tDriver: {
  components: {
    name:      { typeRef: 'string' },
    age:       { typeRef: 'number' },
    gender:    { typeRef: 'string', allowedValues: ['M','F'] },
    isMarried: { typeRef: 'boolean' },
    stateCode: { typeRef: 'string' },
    trainCertType: { typeRef: 'string', allowedValues: ['school','Certified
trainer','Senior refresher','none or other'] },
    hasDUI:    { typeRef: 'boolean' },
    violationCount: { typeRef: 'number' },
    accidentCount: { typeRef: 'number' },
    language: 'Feel'
  },

```

- Name of prospective insured driver (name)
- Age (age)
- Gender: Only two values are allowed, M (male) or F (female) (gender)
- Is the driver married (isMarried)
- State where the driver resides (stateCode)
- What certification does the driver hold? Only r values are allowed here: school, certified trainer, senior refresher, or none. (trainCertType)
- Has the driver ever had a DUI (hasDUI)

- The number of moving violations on the driver's record. (violationCount)
- The number of cited accidents the driver has on his record. (accidentCount)

The fourth type definition is tDriverList.

```
tDriverList: { typeRef: 'tDriver', isCollection: true, language: 'Feel'
},
```

While tDriver models a single record for one driver, tDriverList defines a collection of tDriver.

Step 2: Define the decision tables. Six decision tables are used to compute the probability that a particular car make and model will be stolen (tblCarTheftCategory), vehicle safety rating (tblCarInjuryCategory), driver risk (tblDriverRiskCategory, tblDriverAgeCategory and tblDriverScore), driver eligibility (tblDriverEligible) and to compute the cost of a policy (tblDriverPremium).

Define all data in the data: { } section.

Lists - For this model we have two lists for cars that are routinely stolen (highTheftList) and for driver certifications (trainCertList).

Honda Civics and Toyota Corollas are both cars that are routinely subject to theft and are contained in the list, highTheftList.

```
highTheftList: {
  value: ['Honda Civic', 'Toyota Corolla']
}
```

Driver certifications can be "school", "certified trainer" and "senior refresher". All three certificates are contained in the list, trainCertList.

```
trainCertList:
  value: ['school', 'Certified trainer', 'Senior refresher']
}
```

Scalar Inputs - Two scalar inputs are also included in the data section: isLongTermClient and appDate.

The variable, isLongTermClient, is True if the driver is an existing client and false if not.

The variable, appDate, is the date of the application.

Note the "binding": "get" property for both variables. This property allows the value for isLongTermClient to be captured as a customer input parameter.

2D Arrays – Two 2-D arrays, "cars" and "drivers", contain information used *for testing and visualization purposes, only*. In practice, driver and car records will be entered through an online form.

In practice, all information may be passed from the customer input form to the RASON model using the

```
binding: 'get' technology.cars: {
  type: 'tCarList',
  value: [
    ['Honda HRV', 2017, 'luxury', 21000, false, true, true, true, false,
     false, false, false]
  ],
  binding: 'get'
},
drivers: {
  type: 'tDriverList',
  value: [['John', 57, 'M', false, 'CA', 'school', false, 0, 1]],
  binding: 'get'
}
```

The "cars" 2-D array, used only for testing purposes, contains information related to a specific Honda HRV. This record is a component structure defined by the tCar type def definition (explained above). Recall the tCar typeDef definition:

tCar Components
makeModel: Honda HRV
modelYear: 2017
style: luxury
price: 21000
isConvertible: false
hasDriverAirbags: true
hasPassenAirbags: true
hasSideAirbags: true
hasRollbar: false
has Alarm: false
uninsMotorist: false
medicalPolicy: false

The "drivers" 2-D array, again used only for testing purposes, contains information related to a 57 year old driver, John, who lives in California. Each record is a component structure defined by the tDriver typedef (explained above). Recall the tDriver typeDef definition:

tDriver Components
name: John
age: 57
gender: M
isMarried: false
stateCode: CA
trainCertType: school
hasDUI: false
violationCount: 0
accidentCount: 1

Decision Tables Used in this Example

Six decision tables are utilized in this example.

tblCarTheftCategory – Determines the vehicle theft risk (high, moderate, or low) based on 1. If the vehicle is listed on the onTheftList, the price, and if the vehicle is a convertible.

tblCarInjuryCategory – Determines the risk of injury for a specific vehicle (extremely high, high, moderate or low) based on the safety features of the vehicle (airbags, if the car is a convertible, or has a rollbar).

tblDriverRiskCategory – Determines the risk of a particular driver based on if the driver has ever had a DUI, the number of accidents the driver has been cited for and the number of violations on the driver's record. The response is either true (risky driver) or false.

tblDriverAgeCategory – Determines the age category of a driver: young (<25) , typical (25-70) or senior (71+).

tblDriverEligible – Determines if a driver has any special certifications which may qualify him/her for a premium discount. The response is true (driver is eligible) or false.

tblDriverScore - Assigns a score to each driver based on

tblDriverPremium

Decision Table 1 – tblCarTheftCategory

This decision table accepts three inputs, onTheftList (Boolean), carPrice (number) and isConvertible (Boolean).

- The input onTheftList is whether the vehicle appears in the highTheftList defined within the data section (above).
- The input carPrice is simply the price of the car as captured from the customer.

The output is a string, theftCategory, which can be high, moderate or low.

The hit policy for this table is "U" or "Unique" which means that a unique rule evaluating to a unique result will succeed. If multiple rules are "hit", an error will be returned.

```
tblCarTheftCategory: {
  inputs: ['onTheftList', 'carPrice', 'isConvertible'],
  outputs: ['theftCategory'],
  inputTypes: ['boolean', 'number', 'boolean'],
  outputTypes: ['string'],
  outputValues: [['high'], ['moderate'], ['low']],
  rules: [
    [true, '-', '-', 'high'],
    [false, '>45000', '-', 'high'],
    [false, '<=45000', true, 'high'],
    [false, '[20000..45000]', false, 'moderate'],
    [false, '<20000', false, 'low']
  ],
  hitPolicy: 'U'
},
```

According to the rules of the table,

- Rule 1 assigns a category of "high", if the vehicle is on the highTheftList, regardless of price or if the car is a convertible.
- Rule 2 assigns a category of "high" if the vehicle is not on the highTheftList but has a purchase price of over \$45,000, regardless if the vehicle is a convertible or not.
- Rule 3 assigns a category of "high" if the vehicle is not on the highTheftList but has a purchase price less than or equal to \$45,000 and is a convertible.
- Rule 4 assigns a category of "moderate" if the vehicle is not on the highTheftList, has a purchase price of \$20,000 to \$45,000 and is not a convertible.
- Rule 5 assigns a category of "low" if the car is not on the highTheftList, has a purchase price less than \$20,000 and the car is not a convertible.

This table is called from within the funAutoEligible box function, described below, to determine if a vehicle is eligible for coverage.

Decision Table 2 – tblCarInjuryCategory

This decision table accepts five Boolean inputs, hasDriverAirbags, hasPassenAirbags, hasSideAirbags, hasRollbar, isConvertible and returns one output, isHighRisk, a string. All five inputs are captured from the customer input. The output variable, isHighRisk, can return one of 4 strings: "extremely high", "high", "moderate" or "low".

The hit policy for this table is "P" or Priority. If multiple rules are "hit" and multiple results collected, only one result is returned, the result with the highest priority. Priorities are defined by the order of the output values.

```
"tblCarInjuryCategory": {
  "inputs": ["hasDriverAirbags", "hasPassenAirbags", "hasSideAirbags",
    "hasRollbar", "isConvertible"],
  "outputs": ["injuryCategory"],
  "inputTypes": ["boolean", "boolean", "boolean", "boolean", "boolean"],
  "outputTypes": ["string"],
  "outputValues": [
```

```

    ["extremely high"],
    ["high"],
    ["moderate"],
    ["low"]
  ],
  "rules": [
    [false, false, false, "-", "-", "extremely high"],
    ["-", "-", "-", false, true, "extremely high"],
    [true, false, false, "-", "-", "high"],
    [true, true, false, "-", "-", "moderate"],
    [true, true, true, "-", "-", "low"]
  ],
  "hitPolicy": "P"
}

```

According to the rules of the table:

- The first rule assigns an "extremely high" rating when a car has no airbags, regardless of whether the vehicle has a rollbar or is a convertible.
- The second rule assigns an "extremely high" rating when a car is a convertible with no rollbar, regardless of the status of the airbags.
- The third rule assigns a "high" rating when a car has driver side airbags but not passenger or side airbags.
- The fourth rule assigns a "moderate" rating when a car has driver and passenger side airbags but not side airbags.
- The fifth rule assigns a "low" rating when a car has driver, passenger, and side airbags.

Note: Since the "hit policy" is set to Priority, the result with the highest priority will be returned. Priority is given by the order of outputVariables. If a vehicle has driver, passenger and side airbags but is also a convertible two rules will succeed, rule 2 and rule 5. However, a rating of "extremely high" will be assigned since that rating has a higher priority than a "low" rating.

Decision Table 3 – tblDriverRiskCategory

This decision table accepts three inputs, hasDUI (Boolean), accidentCount (number) and violationCount (number). All inputs are captured from the customer input. The output variable, isHighRisk, returns a Boolean.

The hit policy for this table is "U" or "Unique" which, as discussed above, means that a unique rule evaluating to a unique result will succeed. If multiple rules are "hit", an error will be returned.

```

"tblDriverRiskCategory": {
  "inputs": ["hasDUI", "accidentCount", "violationCount"],
  "outputs": ["isHighRisk"],
  "inputTypes": ["boolean", "number", "number"],
  "outputTypes": ["boolean"],
  "rules": [
    [true, "-", "-", true],
    [false, ">2", "-", true],
    [false, "<=2", ">3", true],
    [false, "<=2", "<=3", false]
  ],
  "hitPolicy": "U"
}

```

According to the rules of the table:

- The first rule returns "true" if a driver has had a DUI, regardless of the number of accidents and violations is on the driver's record.

- If the driver has no DUI but more than 2 accidents, Rule #2 returns "true".
- If the driver has no DUI, 2 accidents or less and greater than 3 moving violations, Rule #3 returns "true".
- If the driver has no DUI, 2 accidents or less and 3 moving violations or less, Rule #4 returns "false".

Decision Table 4 – tblDriverAgeCategory

This decision table accepts two inputs, gender and age. Both inputs are captured from the customer input. The output variable, ageCategory, returns one of three strings: young, typical and senior.

The hit policy for this table is "U" or "Unique" which, as discussed above, means that a unique rule evaluating to a unique result will succeed. If multiple rules are "hit", an error will be returned.

```
"tblDriverAgeCategory": {
  "inputs": ["gender", "age"],
  "outputs": ["ageCategory"],
  "inputTypes": ["string", "number"],
  "outputTypes": ["string"],
  "inputValues": [
    ["M", ""],
    ["F", ""]
  ],
  "outputValues": [
    ["young"],
    ["senior"],
    ["typical"]
  ],
  "rules": [
    ["M", "<25", "young"],
    ["M", "[25..70]", "typical"],
    ["F", "<20", "young"],
    ["F", "[20..70]", "typical"],
    ["-", ">70", "senior"]
  ],
  "hitPolicy": "U"
}
```

According to the rules of the table:

- The first rule returns "young" if the driver is a male under 25 years old.
- The second rule returns "typical" if the driver is male aged 25 to 70 years old.
- The third rule returns "young" if the driver is a female under 25 years old.
- The fourth rule returns "typical" if the driver is a female aged 25 to 70 years old.
- The fifth rule returns "senior" if the driver is over 70, regardless of gender.

Decision Table 5 – tblDriverEligible

This decision table accepts two inputs, ageCategory and hasTrainCert, and returns one output.

- The ageCategory input parameter can be one of three values: young, typical or senior.
- The hasTrainCert input parameter is captured

The output, isEligible, is Boolean.

The hit policy for this table is "U" or "Unique" which, as discussed above, means that a unique rule evaluating to a unique result will succeed. If multiple rules are "hit", an error will be returned.

```
"tblDriverEligible": {
```



```

"inputs": ["ageCategory", "hasTrainCert"],
"outputs": ["isEligible"],
"inputTypes": ["string", "boolean"],
"outputTypes": ["boolean"],
"inputValues": [
  ["young", ""],
  ["senior", ""],
  ["typical", ""]
],
"rules": [
  ["young", true, true],
  ["senior", true, true],
  ["typical", "-", true],
  ["senior", false, false],
  ["young", false, false]
],
"hitPolicy": "U"
}

```

According to the rules of the table:

- The first rule returns "true" if the driver is "young" and has a training certificate.
- The second rule returns "true" if the driver is "senior" and has a training certificate.
- The third rule returns "true" if the driver is "typical", regardless of training certificate status.
- The fourth rule returns "false" if the driver is "senior" and has no training certificate.
- The fifth rule returns "false" if the driver is "young" and has no training certificate.

Decision Table 5 – tblDriverScore

This decision table accepts three inputs, isDriverHighRisk (Boolean), isDriverEligible (boolean) and driverAgeCategory (string), and returns one output, a number.

- The driverAgeCategory input accepts three strings: young, senior or typical. Anything else will produce an error.

The hit policy for this table is "C+" which means that if multiple rules are successful, the collection of results (in this case numbers) will be added together.

```

"tblDriverScore": {
  "inputs": ["isDriverHighRisk", "isDriverEligible", "driverAgeCategory"],
  "outputs": ["eligibleScore"],
  "inputTypes": ["boolean", "boolean", "string"],
  "outputTypes": ["number"],
  "inputValues": [
    ["", "", "young"],
    ["", "", "senior"],
    ["", "", "typical"]
  ],
  "outputValues": [
    ["[0..1000]"]
  ],
  "rules": [
    [true, "-", "-", 100],
    ["-", false, "young", 30],
    ["-", false, "senior", 20],
    ["-", true, "-", 0]
  ],
  "hitPolicy": "C+"
}

```

}

According to the rules of the table:

- The first rule returns the number 100 if the driver is determined to be high risk regardless if they are eligible to be covered or their age category.
- The second rule returns the number 30 if the driver is eligible and classified as a "young" driver, regardless of the driver's risk rating.
- The third rule returns the number 20 if the driver is eligible and classified as a "senior" driver, regardless of the driver's risk rating.
- The third rule returns 0 if the driver is eligible, regardless of the driver's risk rating and age category.

Note: Since the C+ hit policy totals all results returned a "young", "high risk" "eligible" driver would be scored at 130 (100 from rule 1 and 30 from rule 2).

Decision Table 6 – tblDriverPremium

The last decision table in the example computes the premium for the insurance policy.

This decision table takes 5 inputs: ageCat, state, isMarried, isHighRisk and accidentCount.

- The first input, ageCat, is determined by the funDriverScore custom function, explained below. This input can be one of three strings: young, senior, or typical. Any other string will invoke an error.
- The second input, state, is taken directly from the customer input.
- The third input, isMarried, is taken directly from the customer input.
- The fourth input, isHighRisk, is determined by the funDriverScore custom function, explained below.
- The fifth input, accidentCount is taken directly from the customer input.

This table returns one output, the premium for the insurance policy.

The hit policy for this table is "C+" which, as discussed above, means that if multiple rules are successful, the collection of results (in this case numbers) will be added together.

```
"tblDriverPremium": {
  "inputs": ["ageCat", "state", "isMarried", "isHighRisk",
    "accidentCount"],
  "outputs": ["premium"],
  "inputTypes": ["string", "string", "boolean", "boolean", "number"],
  "outputTypes": ["number"],
  "inputValues": [
    ["young", "", "", "", ""],
    ["senior", "", "", "", ""],
    ["typical", "", "", "", ""]
  ],
},
"rules": [
  ["young", "CA,NY,VA", true, "-", "-", 700],
  ["young", "CA,NY,VA", false, "-", "-", 720],
  ["young", "not (CA,NY,VA)", "-", "-", "-", 300],
  ["senior", "CA,NY,VA", "-", "-", "-", 500],
  ["senior", "not (CA,NY,VA)", "-", "-", "-", 200],
  ["typical", "-", "-", "-", "-", 0],
  ["-", "-", "-", true, "-", 1000],
  ["-", "-", "-", "-", "-", "accidentCount*150"]
],
"hitPolicy": "C+"
}
```

According to the rules of the table:

- The first rule returns \$700 if the driver is classified as "young", lives in CA, NY or VA and is married, regardless of the driver's risk rating and number of accidents.
- The second rule returns \$720 if the driver is classified as "young", lives in CA, NY or VA and is single, regardless of the driver's risk rating and number of accidents.
- The third rule returns \$300 if the driver is classified as "young" and does not live in CA, NY, or VA, regardless of marital status, if the driver is high risk, or the number of accidents.
- The fourth rule returns \$500 if the driver is classified as "senior" and lives in CA, NY, or VA, regardless of marital status, if the driver is high risk or the number of accidents.
- The fifth rule returns \$200 if the driver is classified as "senior" and does not live in CA, NY, VA, regardless of marital status, if the driver is high risk or the number of accidents.
- The sixth rule returns \$0 if the driver is classified as "typical", regardless of where the driver resides, marital status, risk status, or number of violations.
- The seventh rule returns \$1000 if the driver is high risk, regardless of where the driver resides, marital status, risk status, or number of violations.
- The eighth rule multiplies the number of accidents * 150, no matter the driver's age category, where the driver resides or driver's risk status.

Note: Since the C+ hit policy totals all results returned, the premium for a young, high risk, single driver, living in CA with no accidents would pay \$1,720 for an insurance policy (\$720 from Rule 2 and \$1000 from Rule 7).

Custom Box Functions and Formulas

If the final application were to only test for a single car and driver, there would be no need for the custom box functions, the code could have proceeded directly to the formulas section by using Feel formulas exclusively. However, in order to process multiple cars and drivers at once, reusable logic, which can apply to all cars and drivers, is required. This is where the box functions come into play. There are six custom box functions used in this example. Three are applicable to drivers and three are applicable to autos. Custom box functions for cars and for drivers are independent of each other.

Each box function invokes a decision table, described above. Recall from the Custom Box Function chapter that the intermediate formulas inside a box function may reference the decision tables, other box functions or be a unique Feel expression. Note that the result formula computes to the same type mentioned in the resultType: "" property.

In order to test each custom box function, a formula will be entered in the "formulas" section of the RASON model. This formula will call the custom box function in order to test the functionality of the function. Notice that the first formulas, autoEligible, autoScore and autoPremium, apply only to 1 auto while the last three custom functions, driverEligible, driverScore and driverPremium, apply only to 1 driver. Note that these values are merely intermediate and are not required. These formulas are provided for testing purposes and visualization purposes, only. The reusable logic is passed using the first record of the cars and drivers array, cars[1] and drivers[1], respectively.

1st Box Function: funAutoEligible

This box function determines if a car is eligible to be covered by a policy or not.

This box function takes one input, car, of type tCar. The output is a "string": not eligible, provisional or eligible.

The body of the function calculates the theft category (theftCat) and injury category (injuryCat).

- theftCat invokes the tblCarTheftCategory decision table using the inputs "car.makeModel in highTheftList", car.price and car.isConvertible. Recall that the decision table, tblCarTheftCategory can return one of three strings: high, moderate or low.
- injuryCat invokes the tblCarInjuryCategory decision table using the inputs car.hasDriverAirbags, car.hasPassengerAirbags, car.hasSideAirbags, car.hasRollbar and car.isConvertible. Recall that the decision table, tblCarInjuryCategory returns one of four strings: extremely high, high, moderate or low.

```
"funAutoEligible": {
  "inputs": ["car"],
  "inputTypes": ["tCar"],
  "language": "FEEL",
  "resultType": "string",
  "body": {
    "theftCat": {
      "formula": "tblCarTheftCategory(, car.makeModel in highTheftList,
        car.price, car.isConvertible)"
    },
    "injuryCat": {
      "formula": "tblCarInjuryCategory(, car.hasDriverAirbags,
        car.hasPassengerAirbags, car.hasSideAirbags, car.hasRollbar,
        car.isConvertible)"
    }
  },
  "result": "if injuryCat = 'extremely high' then 'not eligible' else if
injuryCat = 'high' then 'provisional' else if theftCat = 'high' then
'provisional' else 'eligible' "
},
```

Function Result

If the injury category (injuryCat) is "extremely high", then funAutoEligible will assign a category of "not eligible".

ELSE

If the injury category (injuryCat) is "high" then funAutoEligible will assign a category of "provisional".

ELSE

If the theft category (theftCat) is "high", then funAutoEligible will return "provisional".

Otherwise, "eligible" is returned.

- Therefore, if injuryCat is moderate or low and theftCat is moderate or low, funAutoEligible will return "eligible".

Frontline Systems recommends that each box function be tested. In order to test this box function, we must invoke the function within the formulas section. To test this box function, a new formula, autoEligible, is added to the "formulas" section, as shown below.

```
"formulas": {
  "autoEligible": {
    "freeFormula": "funAutoEligible(cars[1])",
    "finalValue": []
  }
}
```

Recall the "cars" array from the "data" section. Recall that this array is entered as a testing tool only. In production, the data from the cars array will be captured from customer input.

The result from the autoEligible formula is below. The final value for the autoEligible formula is "eligible".

```
{
  "status": {
    "id": "2590+AutoInsurancePolicyExample+2022-03-14-18-14-04-032487",
    "code": 0,
    "codeText": "Calculation executed.",
    "solveTimestamp": "2022-03-14-18-14-04-969026",
    "solveTime": 5
  },
  "results": {
    "autoEligible.finalValue": []
  },
  "autoEligible" {
    "finalValue": {
      "objectType": "dataFrame",
      "name": "autoEligible",
      "order": "col",
      "colNames": ["Col1"],
      "colTypes": ["wstring"],
      "indexCols": null,
      "data": [
        ["eligible"]
      ]
    }
  }
}
```

2nd Box Function: funAutoScore

This box function assigns a score for each vehicle.

This box function takes one input, car, of type tCar. The result of this function is a number: 100, 50, or 0

The body of this function, elig, assigns a score to each vehicle by invoking the funAutoEligible box function (described above).

- If funAutoEligible returns "not eligible", then the function will return a score of 100.
- If funAutoEligible returns "provisional" then the function will return a score of 50
- Else 0.

```
"funAutoScore": {
  "inputs": ["car"],
  "inputTypes": ["tCar"],
  "language": "FEEL",
  "resultType": "number",
  "body": {
    "elig": {
      "formula": "funAutoEligible(car)"
    }
  },
  "result": "if elig = 'not eligible' then 100 else if elig =
'provisional' then 50 else 0"
},
```

To test this box function, a new formula, autoScore, is added to the "formulas" section, as shown below.

```
"autoScore": {
  "freeFormula": "funAutoScore(cars[1])",
  "finalValue": []
}
```

As mentioned above, the "cars" array exists within the "data" section as a testing tool only. In production, the data from the cars array will be captured from customer input.

The result from the autoScore formula is below. The final value for the autoScore formula is 0.

```
...
"autoScore": {
  "finalValue": {
    "objectType": "dataFrame",
    "name": "autoScore",
    "order": "col",
    "colNames": ["Col1"],
    "colTypes": ["double"],
    "indexCols": null,
    "data": [
      [0]
    ]
  }
}
```

3rd Box Function: funAutoPremium

This box function determines the premium for one driver and one vehicle.

This function takes one input, car, of type tCar. The result of this function is a number.

The body of this function calculates the total insurance premium based on the customer's input data for "car".

base: This is the base price for the insurance policy.

- If the car is "compact" then the value of 250 is returned for base,
- If the car is "sedan", then 400 is returned for "base"
- If the car style is "luxury" then 500 is returned for base.

Recall that the tCar custom type definition allows three values for style: compact, sedan, or luxury.

- age: This formula calculates the vehicle age by subtracting the year given in appDate by the model year.
- ageCat: This formula assigns a category to the age of a car. If age is less than 0, then "new" is assigned; if the car age is between 0 and 5, then "recent" is assigned, and if the car age is between 5 and 10, then "older" is assigned, and if the car is older than 10 years, then a category of "really old" is assigned.
- agePrem: This formula determines how the premium should be increased due to the age of the vehicle.
 - If ageCat = "new", then \$400 is added to the base premium.
 - If ageCat="recent", then \$300 is added to the base premium.
 - If ageCat = "older", then \$250 is added to the base premium.
 - If the ageCat="really old", then nothing is added to the base premium.
- uninsPrem: This formula adds \$300 to the base premium if car.uninsMotorist is True.
- medicPrem: This formula adds \$600 to the base premium if car.medicPrem is true.

- **injuryCat:** This formula invokes the `tblCarInjuryCategory` decision table using the car input parameters: `car.hasDriverAirbags`, `car.hasPassenAirbags`, `car.hasSideAirbags`, `car.hasRollbar`, `car.isConvertible`.
- **injuryPremium:** This formula adds a value to the base premium, according to the injury category.
 - If `injuryCat = extremely high`, then \$1,000 is added to the base premium.
 - If `injuryCat = high`, then 500 is added to the base premium.
 - Otherwise nothing is added to the base premium.
- **theftCat:** This formula invokes the `tblCarTheftCategory` decision table using the car input parameters: `car.makeModel` in the `highTheftList`, `carPrice` and `car.isConvertible`.
- **theftPremium:** This formula adds a value to the base premium, according to the theft category.
 - If `theftCat = high`, then \$500 is added to the base premium.
- **airbagDiscount:** A discount is applied if a vehicle has driver, passenger or side airbags.
 - If a car has only driver side airbags (`car.hasDriverAirbags = true`), then return a 12% discount.
 - If a car has both driver and passenger side airbags (`car.hasDriverAirbags` and `car.hasPassenAirbags = true`), then return and 15% discount.
 - If a car has all three types of airbags, driver, passenger and side, then return a 18% discount.
 - If a car has no airbags, a discount is not returned.
- **alarmDiscount:** A 10% discount is applied if a high theft vehicle has an alarm system.

funAutoPremium Custom Box Function

```
"funAutoPremium": {
  "inputs": ["car"],
  "inputTypes": ["tCar"],
  "language": "FEEL",
  "resultType": "number",
  "body": {
    "base": {
      "formula": "if car.style = 'compact' then 250 else if car.style =
        'sedan' then 400 else if car.style = 'luxury' then 500 else 0"
    },
    "age": {
      "formula": "date(appDate).year - car.modelYear"
    },
    "agePrem": {
      "formula": "if ageCat = 'new' then 400 else if ageCat = 'recent'
        then 300 else if ageCat = 'older' then 250 else 0"
    },
    "uninsPrem": {
      "formula": "if car.uninsMotorist then 300 else 0"
    },
    "medicPrem": {
      "formula": "if car.medicalPolicy then 600 else 0"
    },
    "injuryCat": {
      "formula": "tblCarInjuryCategory(, , car.hasDriverAirbags,
        car.hasPassenAirbags, car.hasSideAirbags, car.hasRollbar,
        car.isConvertible)"
    },
  },
}
```

```

"injuryPrem": {
  "formula": "if injuryCat = 'extremely high' then 1000 else if
injuryCat = 'high' then 500 else 0"
},
"theftCat": {
  "formula": "tblCarTheftCategory(, car.makeModel in highTheftList,
car.price, car.isConvertible)"
},
"theftPrem": {
  "formula": "if theftCat = 'high' then 500 else 0"
},
"airbagDiscount": {
  "formula": "if (car.hasDriverAirbags and car.hasPassenAirbags =
false and car.hasSideAirbags = false) then 0.12 else if
(car.hasDriverAirbags and car.hasPassenAirbags and car.hasSideAirbags
= false) then 0.15 else if car.hasDriverAirbags and
car.hasPassenAirbags and car.hasSideAirbags) then 0.18 else 0"
}
},
"result": "(base + agePrem + uninsPrem + medicPrem + injuryPrem +
theftPrem) * (1-airbagDiscount)"
}
},

```

The result adds all premiums to the base and subtracts the air bag and alarm discounts.

To test this box function, a new formula, autoPremium, is added to the "formulas" section, as shown below.

```

"autoPremium": {
  "feelFormula": "funAutoPremium(cars[1])",
  "finalValue": []
}

```

The result from the autoPremium formula is below. The final value for the autoPremium formula is 410.

```

...
"autoPremium": {
  "finalValue": {
    "objectType": "dataFrame",
    "name": "autoPremium",
    "order": "col",
    "colNames": ["Col1"],
    "colTypes": ["double"],
    "indexCols": null,
    "data": [
      [410.00000000000006]
    ]
  }
}

```

4th Box Function: funDriverEligible

This box function determines if a driver is eligible to be covered by a policy or not.

This function takes one input, driver, of type tDriver. The result of this function is a Boolean.

The body of this function invokes the decision table, tblDriverAgeCategory. Recall that this decision table takes two inputs: gender and age. The box function passes driver.gender for the gender input and driver.age for the age input.


```

"funDriverEligible": {
  "inputs": ["driver"],
  "inputType": ["tDriver"],
  "language": "FEEL",
  "resultType": "boolean",
  "body": {
    "driverAgeCat": {
      "formula": "tblDriverAgeCategory(, driver.gender, driver.age)"
    }
  },
  "result": "tblDriverEligible(, driverAgeCat, driver.trainCertType in
trainCertList)"
},

```

The result invokes the `tblDriverEligible` decision table with two inputs, `driverAgeCat` and `driver.trainCertType` in `trainCertList`. (Recall that the decision table, `tblDriverEligible`, takes two inputs `ageCategory` and `hasTrainCert` and one output, `isEligible`.)

To test this box function, a new formula, `driverEligible`, is added to the "formulas" section, as shown below.

```

"driverEligible": {
  "feelFormula": "funDriverEligible(drivers[1])",
  "finalValue": []
}

```

The result from the `driverEligible` formula is below. The final value for the `driverEligible` formula is true.

```

...
  "driverEligible": {
    "finalValue": {
      "objectType": "dataFrame",
      "name": "driverEligible",
      "order": "col",
      "colNames": ["Col1"],
      "colTypes": ["wstring"],
      "indexCols": null,
      "data": [
        ["true"]
      ]
    }
  }
}

```

5th Box Function: `funDriverScore`

This box function assigns a score to a driver based on several factors such as if the driver is high risk, the driver's age, and if the driver is eligible to be covered by a policy.

This function takes one input, `driver`, of type `tDriver`. The result of this function is a score, or a number.

The body of this function invokes three decision tables: `tblDriverRiskCategory`, `tblDriverAgeCategory` and `tblDriverEligible`.

- `driverHighRisk` invokes the decision table `tblDriverRiskCategory` to determine if a driver is high risk or not. Recall that this decision table accepts three inputs: `hasDUI`, `accidentCount` and `violationCount`. This function passes `drivers.hasDUI` for the first input (`hasDUI`), `driver.accidentCount` for the second input (`accidentCount`) and `driver.violationCount` for the third input (`violationCount`).
- `driverAgeCat` invokes the decision table, `tblDriverAgeCategory`, to determine if a driver can be categorized as young, typical or senior. Recall that this decision table accepts two inputs, `gender` and `age`, and has one output, `ageCategory`. The function passes `driver.gender` for the `gender` input parameter and `driver.age` for the `age` input parameter.

- driverEligible invokes the decision table tblDriverEligible to determine if a driver is eligible to be covered by a policy or not. Recall that this decision table accepts two inputs, ageCategory and hasTrainCert, and one output, isEligible, a boolean.

```
"funDriverScore": {
  "inputs": ["driver"],
  "inputTypes": ["tDriver"],
  "language": "FEEL",
  "resultType": "number",
  "body": {
    "driverHighRisk": {
      "formula": "tblDriverRiskCategory(, driver.hasDUI,
        driver.accidentCount, driver.violationCount)"
    },
    "driverAgeCat": {
      "formula": "tblDriverAgeCategory(, driver.gender, driver.age)"
    },
    "driverEligible": {
      "formula": "tblDriverEligible(, driverAgeCat,
        driver.trainCertType in trainCertList)"
    }
  },
  "result": "tblDriverScore(, driverHighRisk, driverEligible,
    driverAgeCat)"
},
```

The result invokes the tblDriverScore decision table with three inputs, driverHighRisk, driverEligible and driverAgeCat. (Recall that the decision table, tblDriverScore, takes three inputs: isDriverHighRisk, isDriverEligible and driverAgeCategory.)

To test this box function, a new formula, driverScore is added to the "formulas" section, as shown below.

```
"driverScore": {
  "feelFormula": "funDriverScore(drivers[1])",
  "finalValue": []
}
```

The result from the driverScore formula is below. The final value for the driverScore formula is 0.

```
...
  "driverScore": {
    "finalValue": {
      "objectType": "dataFrame",
      "name": "driverScore",
      "order": "col",
      "colNames": ["Col1"],
      "colTypes": ["double"],
      "indexCols": null,
      "data": [
        [0]
      ]
    }
  }
}
```

6th Box Function: funDriverPremium

This box function assigns a premium to a driver based on several factors such as if the driver is high risk and their age.

This function takes one input, driver, of type tDriver. The result of this function is the premium or price of the policy.

The body of this function invokes two decision tables: tblDriverRiskCategory and tblDriverAgeCategory.

- driverHighRisk invokes the decision table tblDriverRiskCategory to determine if a driver is high risk or not. Recall that this decision table accepts three inputs: hasDUI, accidentCount and violationCount. This function passes drivers.hasDUI for the first input (hasDUI), driver.accidentCount for the second input (accidentCount) and driver.violationCount for the third input (violationCount).
- driverAgeCat invokes the decision table, tblDriverAgeCategory, to determine if a driver can be categorized as young, typical or senior. Recall that this decision table accepts two inputs, gender and age, and has one output, ageCategory. The function passes driver.gender for the gender input parameter and driver.age for the age input parameter.

```
funDriverPremium: {
  inputs: ['driver'],
  inputTypes: ['tDriver'],
  language: "FEEL",
  resultType: "number",
  body: {
    driverHighRisk: {
      formula: "tblDriverRiskCategory(,, driver.hasDUI,
        driver.accidentCount, driver.violationCount) "
    },
    driverAgeCat: {
      formula: "tblDriverAgeCategory(,, driver.gender,
        driver.age) "
    }
  },
  result: "tblDriverPremium(,, driverAgeCat, driver.stateCode,
    driver.isMarried, driverHighRisk, driver.accidentCount) "
}
```

The result invokes the tblDriverPremium decision table with five inputs, driverAgeCat, driver.StateCode, driver.IsMarried, driverHighRisk, driver.accidentCount. (Recall that the decision table, tblDriverPremium, takes five inputs: ageCat, state, isMarried, isHighRisk, accidentCount.)

To test this box function, a new formula, driverPremium is added to the "formulas" section, as shown below.

```
"driverPremium": {
  "feelFormula": "funDriverPremium(drivers[1])",
  "finalValue": []
}
```

The result from the driverPremium formula is below. The final value for the driverPremium formula is 150.

```
...
"driverPremium": {
  "finalValue": {
    "objectType": "dataFrame",
    "name": "driverPremium",
    "order": "col",
    "colNames": ["Col1"],
    "colTypes": ["double"],
    "indexCols": null,
    "data": [
      [150]
    ]
  }
}
```

```

    }
}

```

Computing the remaining formulas

At this point we have created all required decision tables and custom box functions. Next, we compute the eligibility and the premium for all cars and drivers using the Feel iteration features **for** and **every**. We sum the individual results at each iteration. These formulas are also intermediate and for testing purposes only.

totalScore

This formula computes both the driver and auto scores for all drivers and autos in the cars and drivers arrays.

```

"totalScore": {
  "feelFormula": "sum(for c in cars return funAutoScore(c)) +
    sum(for d in drivers return funDriverScore(d))",
  "finalValue": []
}

```

Recall that the funAutoScore custom function assigns a score of 100 to ineligible cars, a score of 50 to provisional cars and a score of 0 to all remaining cars. This function calls the funAutoEligible custom function to determine the eligibility rating.

Recall that the funAutoEligible custom function calls two decision tables, tblCarTheftCategory and tblCarInjuryCategory, to determine both the injury category and theft category of a vehicle.

Using our test car and driver, both autoScore and driverScore return 0 and thus totalScore returns 0 as well.

```

...
  "totalScore": {
    "finalValue": {
      "objectType": "dataFrame",
      "name": "totalScore",
      "order": "col",
      "colNames": ["Col1"],
      "colTypes": ["double"],
      "indexCols": null,
      "data": [
        [0]
      ]
    }
  }
}

```

totalPremium

This formula computes the total premium for all drivers and cars being considered for an insurance policy.

```

"totalPremium": {
  "feelFormula": "sum(for c in cars return funAutoPremium(c)) +
    sum(for d in drivers return funDriverPremium(d))",
  "finalValue": []
}

```

Recall that the funAutoPremium custom function determines a premium based on various properties of the vehicle such as the type and age of vehicle. The funDriverPremium function determines a premium based on various properties of a driver.

Using our test car and driver, autoPremium returns 410 and driverPremium returns 150, thus totalPremium returns 560 (410 + 150).

```
...
    "totalPremium": {
      "finalValue": {
        "objectType": "dataFrame",
        "name": "totalPremium",
        "order": "col",
        "colNames": ["Col1"],
        "colTypes": ["double"],
        "indexCols": null,
        "data": [
          [560]
        ]
      }
    }
  }
}
```

allEligible, allEnelibile, isEligible

The next three formulas determine what cars and drivers are eligible to be included on the policy.

- allEligible: Calls funAutoEligible and funDriverEligible to determine what cars and drivers are eligible to be included in the policy

```
"allEligible": {
  "feelFormula": "(every c in cars satisfies
    funAutoEligible(c) = 'eligible') and (every d in drivers
    satisfies funDriverEligible(d) = true)",
  "finalValue": []
}
```

- allIneligible: Calls funAutoEligible and funDriverEligible to determine what cars and drivers are *not* eligible to be included in the policy.

```
"allIneligible": {
  "feelFormula": "(every c in cars satisfies
    funAutoEligible(c) = 'not eligible') and (every d in
    drivers satisfies funDriverEligible(d) = false)",
  "finalValue": []
}
```

- isEligible: If the driver is a long term client *or* if the driver and car are eligible to be included on the policy *or* totalscore is less than 100, the client is determined to be "eligible".

```
"isEligible": {
  "feelFormula": "(isLongTermClient or allEligible or
    totalScore < 100",
  "finalValue": []
}
```

Using our test car and driver arrays,

- allEligible returns true meaning that a car and driver were found that are eligible to be included on the policy.
- allIneligible returns false, meaning that no cars or drivers were found to be "ineligible" to be included on the policy.
- isEligible returns true meaning that the driver and car are eligible to be included on the policy.

```
"allEligible": {
```

```

        "finalValue": {
            "objectType": "dataFrame",
            "name": "allEligible",
            "order": "col",
            "colNames": ["Col1"],
            "colTypes": ["wstring"],
            "indexCols": null,
            "data": [
                ["true"]
            ]
        }
    },
    "allIneligible": {
        "finalValue": {
            "objectType": "dataFrame",
            "name": "allIneligible",
            "order": "col",
            "colNames": ["Col1"],
            "colTypes": ["wstring"],
            "indexCols": null,
            "data": [
                ["false"]
            ]
        }
    },
    "isEligible": {
        "finalValue": {
            "objectType": "dataFrame",
            "name": "isEligible",
            "order": "col",
            "colNames": ["Col1"],
            "colTypes": ["wstring"],
            "indexCols": null,
            "data": [
                ["true"]
            ]
        }
    }
}

```

autoPolicy

The final required result is if the potential client, the driver, qualifies to be covered by an insurance policy and the cost of the policy. If the driver is not eligible, null is returned, otherwise, the premium is returned.

Given the test data in the cars and drivers arrays, the premium returns is \$560.

```

{
    "status": {
        "id": "2590+AutoInsurancePolicyExample+2022-03-17-19-35-19-148143",
        "code": 0,
        "codeText": "Calculation executed.",
        "solveTimestamp": "2022-03-17-19-35-20-151074",
        "solveTime": 12
    },
    "results": {
        "allEligible.finalValue": [],
        "allIneligible.finalValue": [],
    }
}

```

```

    "autoEligible.finalValue": [],
    "autoPolicy.finalValue": [],
    "autoPremium.finalValue": [],
    "autoScore.finalValue": [],
    "driverEligible.finalValue": [],
    "driverPremium.finalValue": [],
    "driverScore.finalValue": [],
    "isEligible.finalValue": [],
    "totalPremium.finalValue": [],
    "totalScore.finalValue": []
  },
  "autoEligible": {
    "finalValue": {
      "objectType": "dataFrame",
      "name": "autoEligible",
      "order": "col",
      "colNames": ["Col1"],
      "colTypes": ["wstring"],
      "indexCols": null,
      "data": [
        ["eligible"]
      ]
    }
  },
  "autoScore": {
    "finalValue": {
      "objectType": "dataFrame",
      "name": "autoScore",
      "order": "col",
      "colNames": ["Col1"],
      "colTypes": ["double"],
      "indexCols": null,
      "data": [
        [0]
      ]
    }
  },
  "autoPremium": {
    "finalValue": {
      "objectType": "dataFrame",
      "name": "autoPremium",
      "order": "col",
      "colNames": ["Col1"],
      "colTypes": ["double"],
      "indexCols": null,
      "data": [
        [410.000000000000006]
      ]
    }
  },
  "driverEligible": {
    "finalValue": {
      "objectType": "dataFrame",
      "name": "driverEligible",
      "order": "col",
      "colNames": ["Col1"],
      "colTypes": ["wstring"],

```

```

        "indexCols": null,
        "data": [
            ["true"]
        ]
    },
    "driverScore": {
        "finalValue": {
            "objectType": "dataFrame",
            "name": "driverScore",
            "order": "col",
            "colNames": ["Col1"],
            "colTypes": ["double"],
            "indexCols": null,
            "data": [
                [0]
            ]
        }
    },
    "driverPremium": {
        "finalValue": {
            "objectType": "dataFrame",
            "name": "driverPremium",
            "order": "col",
            "colNames": ["Col1"],
            "colTypes": ["double"],
            "indexCols": null,
            "data": [
                [150]
            ]
        }
    },
    "totalScore": {
        "finalValue": {
            "objectType": "dataFrame",
            "name": "totalScore",
            "order": "col",
            "colNames": ["Col1"],
            "colTypes": ["double"],
            "indexCols": null,
            "data": [
                [0]
            ]
        }
    },
    "totalPremium": {
        "finalValue": {
            "objectType": "dataFrame",
            "name": "totalPremium",
            "order": "col",
            "colNames": ["Col1"],
            "colTypes": ["double"],
            "indexCols": null,
            "data": [
                [560]
            ]
        }
    }
}

```



```

    },
    "allEligible": {
      "finalValue": {
        "objectType": "dataFrame",
        "name": "allEligible",
        "order": "col",
        "colNames": ["Col1"],
        "colTypes": ["wstring"],
        "indexCols": null,
        "data": [
          ["true"]
        ]
      }
    },
    "allIneligible": {
      "finalValue": {
        "objectType": "dataFrame",
        "name": "allIneligible",
        "order": "col",
        "colNames": ["Col1"],
        "colTypes": ["wstring"],
        "indexCols": null,
        "data": [
          ["false"]
        ]
      }
    },
    "isEligible": {
      "finalValue": {
        "objectType": "dataFrame",
        "name": "isEligible",
        "order": "col",
        "colNames": ["Col1"],
        "colTypes": ["wstring"],
        "indexCols": null,
        "data": [
          ["true"]
        ]
      }
    },
    "autoPolicy": {
      "finalValue": {
        "objectType": "dataFrame",
        "name": "autoPolicy",
        "order": "col",
        "colNames": ["Col1"],
        "colTypes": ["double"],
        "indexCols": null,
        "data": [
          [560]
        ]
      }
    }
  }
}

```

And there you have it! The final model would only need to print the final goal, the value of the premium. This can be achieved quickly by simply removing "finalValue" from each intermediate formula.