Progress®

Corticon.js Integration

Progress® Corticon®

Copyright

© 2021 Progress Software Corporation and/or its subsidiaries or affiliates. All rights reserved.

These materials and all Progress® software products are copyrighted and all rights are reserved by Progress Software Corporation. The information in these materials is subject to change without notice, and Progress Software Corporation assumes no responsibility for any errors that may appear therein. The references in these materials to specific platforms supported are subject to change.

DataDirect Cloud, DataDirect Connect, DataDirect Connect64, DataDirect XML Converters, DataDirect XQuery, DataRPM, Defrag This, Deliver More Than Expected, DevReach (and design), Icenium, Inspec, Ipswitch, iMacros, Kendo UI, Kinvey, MessageWay, MOVEit, NativeChat, NativeScript, OpenEdge, Powered by Chef, Powered by Progress, Progress, Progress Software Developers Network, SequeLink, Sitefinity (and Design), Sitefinity, Sitefinity (and design), SpeedScript, Stylus Studio, Stylized Design (Arrow/3D Box logo), Styleized Design (C Chef logo), Stylized Design of Samurai, TeamPulse, Telerik, Telerik (and design), Test Studio, WebSpeed, WhatsConfigured, WhatsConnected, WhatsUp, and WS_FTP are registered trademarks of Progress Software Corporation or one of its affiliates or subsidiaries in the U.S. and/or other countries.

Analytics360, AppServer, BusinessEdge, Chef Automate, Chef Compliance, Chef Desktop, Chef Habitat, Chef WorkStation, Corticon.js, Corticon Rules, Data Access, DataDirect Autonomous REST Connector, DataDirect Spy, DevCraft, Fiddler, Fiddler Everywhere, FiddlerCap, FiddlerCore, FiddlerScript, Hybrid Data Pipeline, iMail, JustAssembly, JustDecompile, JustMock, KendoReact, NativeScript Sidekick, OpenAccess, PASOE, Pro2, ProDataSet, Progress Results, Progress Software, ProVision, PSE Pro, Push Jobs, SafeSpaceVR, Sitefinity Cloud, Sitefinity CMS, Sitefinity Digital Experience Cloud, Sitefinity Feather, Sitefinity Insight, Sitefinity Thunder, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, Supermarket, SupportLink, Unite UX, and WebClient are trademarks or service marks of Progress Software Corporation and/or its subsidiaries or affiliates in the U.S. and other countries. Java is a registered trademark of Oracle and/or its affiliates. Any other marks contained herein may be trademarks of their respective owners.

Last updated with new content: Corticon.js 1.2

Updated: 2021/06/04

Table of Contents

About Corticon.js integration.....7

Node.js platform.....11
AWS Lambda platform.....12
Azure Functions platform.....12
Google Cloud Functions platform.....12
Browser platform.....13

JSON payloads and results in Corticon.js.....17

Customized data access operators.....27

Rule statements and Rule messages in Corticon.js.....33

How to use the Corticon.js utilities.....37

Config options.....39

How to test deployed rules.....41

How to use logging and diagnostics.....43

1

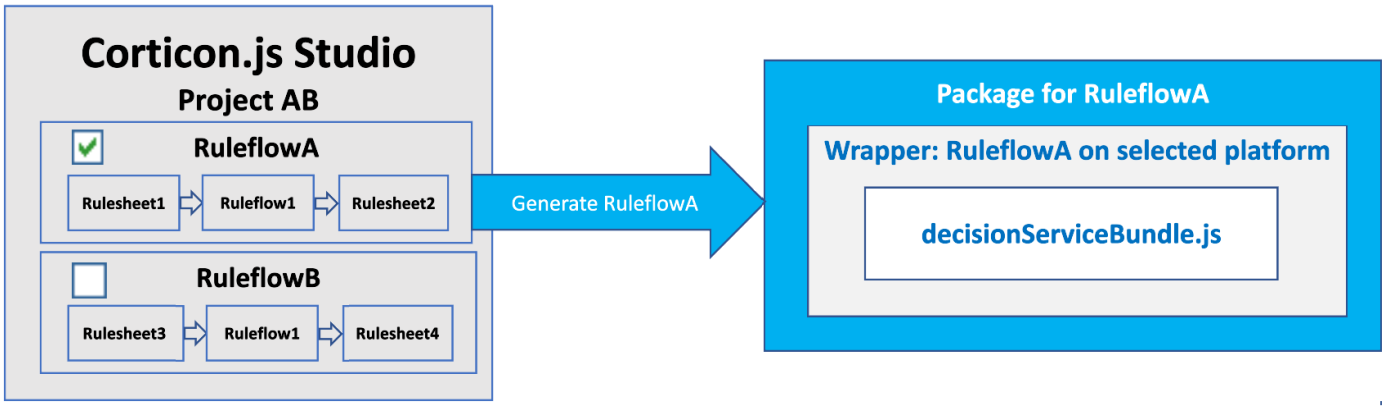
About Corticon.js integration

Corticon.js enables you to deploy rules on any JavaScript platform:

- Rules deployed as serverless functions on AWS Lambda or Microsoft Azure Functions
- Rules integrated into a cloud work flow such as AWS Step Functions or Microsoft Flow
- Rules run on your own back end as part of your Node.js platform
- Rules bundled in a mobile app created with Xamarin, React, Vue.js, or other toolkits
- Rules executed in a browser as part of a web application

After you have created and tested your rules in the Corticon.js Studio Ruletests, you are ready to package them for deployment. When you package your rules, Corticon generates a **self-contained JavaScript bundle**:

- **Self-contained** means that it has no dependencies on other Corticon components at runtime.
- **JavaScript** means that it will be compatible with any JavaScript platform.
- **Bundle** means that the complete set of rules are produced in the scope of the Ruleflow to include all its Rulesheets and enclosed Ruleflows in a single JavaScript file.



Each **bundle** sets the top-level Ruleflow as the entry point to your Decision Service, tailored for the selected platform, with a wrapper that you can customize, and the rules in an obfuscated JavaScript file.

How to package rules for JavaScript deployment

When you package rules for deployment, you select the **target platform**. Corticon will generate a JavaScript bundle with your rules and a wrapper specific to the target platform. The available target platforms are:

- **AWS Lambda**
- **Azure Functions**
- **Google Cloud Functions**
- **Browser**
- **Node.js**

Selecting AWS Lambda, Azure Functions, or Google Cloud Functions as the target platform generates the rules and a wrapper ready

for deployment as a serverless function. After the rules are deployed, you can use them on your cloud platform—for example, to expose the decision service through a REST endpoint, to respond to a database update, or to integrate into a cloud vendor workflow system such as AWS step functions.

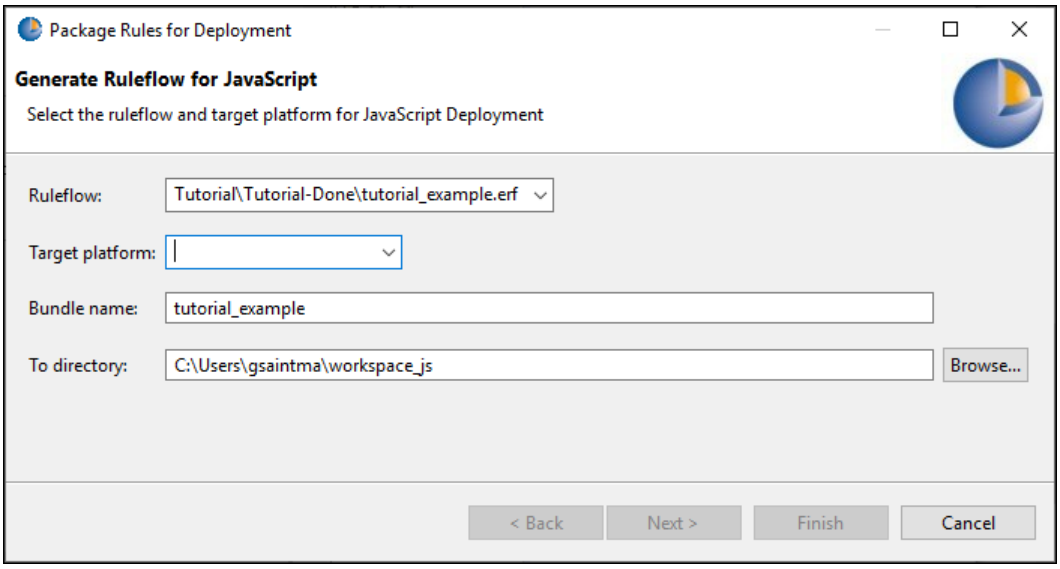
The Node.js target can be used for running decision services in Node server and as well to run them in Mobile applications created with NativeScript and ReactNative.

Selecting Browser generates the decision service and simple example code that demonstrates how to integrate the rules into your application.

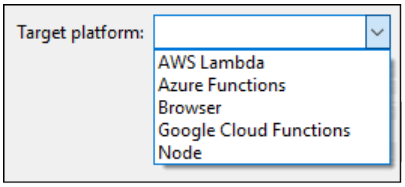
Note: See the Corticon.js Supported Platforms Matrix to review the supported JavaScript and web platforms, and mobile apps. You are not limited to these JavaScript platforms. The API for integrating rules in your JavaScript application allows you to develop your own wrapper or integration code that calls rules.

To create a Corticon JavaScript package:

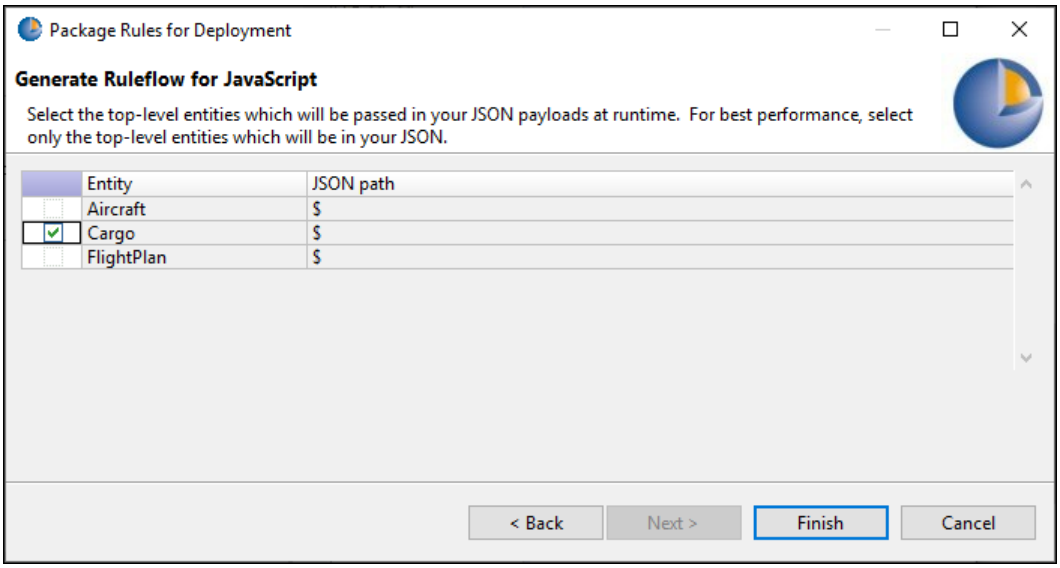
1. In Corticon.js Studio, select a project, and then select **Project > Package Rules for Deployment**. The **Package Rules for Deployment** dialog box opens:



2. Select the **Ruleflow** for your application.
3. Choose the **Target Platform**:



4. Enter a preferred **Bundle** name so that you produce your package in a distinctly named folder. Each Ruleflow selected generates a separate package.
5. Enter a preferred **to Directory** where your package will be placed.
6. Click **Next**. The second panel of the dialog opens:



7. Choose the top-level entities for this package. In our example, only the Cargo entity is referenced so just that entity is selected.
8. Click **Finish** to generate the packaged JavaScript as a Decision Service plus everything you need to deploy it. The packaging process generates a single JavaScript bundle that is the entry point for the rules, which contain all the decision logic in its enclosed components for the selected Ruleflow. The generated bundle is compressed and obfuscated. It is valid, executable JavaScript, but it is not readable and cannot be edited.

How to use the JavaScript rules API

All five platform wrappers have a lot in common. Their fundamental difference is the flow of the operations on each platform. Each of the `sample.js` wrappers is self-documented.

The integration of the bundle with your JavaScript application is a simple API, not much more than a call to execute against the bundle. The following sample code is taken from the `node.sample.js` file:

1. Include the generated rule bundle:

```
const decisionService = require('./decisionServiceBundle');
```

2. Get the JSON payload from your application:

```
const payload = readPayload(payloadFileName);
```

3. Set the configuration:

```
const configuration = { logLevel: 0 };
```

4. Execute the rules on the payload:

```
const result = decisionService.execute(payload, configuration);
```

Customizing the wrapper

The wrappers generated by Corticon.js are sample code with extensive comments. You are free to modify them to meet your needs, or use them as reference for integrating Corticon.js rules with other code.

At their core, the wrappers all do the same things:

- Get the JSON payload to be processed.
- Execute rules on the payload.
- Do something with the result JSON payload.

In your custom code you could:

- Transform the payload, for example to make it suitable to pass to Corticon.
- Enrich the payload, for example with data you read from a database
- Store the result payload, saving the output of rule processing
- Process the result payload, for example passing it to REST service or another step in a workflow

What you do is up to you.

Each of the wrapper samples provides the same logging and error handling template. You can also define a preferred logger, and set logging to debug mode. See How to use logging and diagnostics on page 43 for details.

Service contracts

A service contract for a decision service shows the inputs you need to pass to the decision service as part of the input payload, and the output you can expect to receive as part of the result JSON payload.

For details, see the following topics:

- Node.js platform
- AWS Lambda platform
- Azure Functions platform
- Google Cloud Functions platform
- Browser platform

Node.js platform

When packaging for Node deployment, Corticon.js will generate the files:

- `decisionServiceBundle.js`: Your obfuscated rules
- `node.sample.js`: Sample Node.js wrapper

The wrapper, `node.sample.js`, is pure sample code demonstrating how to embed rules in a Node.js application. How you invoke rules from your own Node.js application depends on your needs.

You need to define a file named `payload.json` that contains the decision service request. You can export the data from Corticon rule tester. To do so, open the Ruletest file (`.ert`) file, and then choose the menu option: **Rulesheet->RuleTest -> Data -> Input -> Export Request JSON**

The sample wrapper reads the file `payload.json`, passes it to your rules, and writes the result to `result.json`.

See the Node.js documentation for details on deploying serverless functions.

AWS Lambda platform

When packaging for AWS Lambda deployment, Corticon.js generates the files:

- `decisionServiceBundle.js`: Your rules, obfuscated
- `index.js`: Sample AWS Lambda wrapper
- `lambda.zip`: Zip file with both `.js` files to simplify deployment.

The wrapper, index.js, implements the AWS Lambda API to be invoked as a serverless function and returns the results of its execution.

When invoked, the AWS Lambda function is passed a JSON payload. Examples of how this could be triggered include using a REST API gateway to call your function, and using the function as a step in an AWS Step Functions workflow. Once deployed as a serverless function, your rules are available to the AWS ecosystem.

The wrapper is a complete, ready-to-deploy, Lambda function but you can also modify it to meet the specific needs of your application.

The `lambda.zip` file is generated as a convenience. AWS makes it easy to deploy a Lambda function in one step using a `.zip` file.

See the AWS Lambda documentation for details on deploying serverless functions.

Azure Functions platform

When packaging for AWS Functions deployment, Corticon.js generates the files:

- `decisionServiceBundle.js`:Your rules, obfuscated
- `azure.sample.js`: Sample Azure Functions wrapper

The wrapper, `azure.sample.js`, implements the Azure Functions API for responding to HTTP triggers.

When invoked the Azure function is passed a JSON payload. This is most likely triggered by a REST call to invoke the function.

The wrapper is a complete, ready-to-run, Azure function, yet you can also modify it to meet the specific needs of your application. The Azure functions API has different signatures for different triggers. You can modify the wrapper to meet the needs of different triggers.

See the Azure Functions documentation for details on deploying serverless functions.

Google Cloud Functions platform

When packaging for Google Cloud Functions deployment, Corticon.js generates the files:

- `decisionServiceBundle.js`:Your rules, obfuscated
- `index.js`: Sample Google Cloud Functions wrapper

The wrapper, `index.js`, implements the Google Cloud Functions API for responding to HTTP triggers.

When invoked, the Google Cloud function is passed a JSON payload. This is most likely triggered by a REST call to invoke the function.

The wrapper is a complete, ready-to-run, Google Cloud function, yet you can also modify it to meet the specific needs of your application. The Google Cloud functions API has different signatures for different triggers. You can modify the wrapper to meet the needs of different triggers.

See the Google Cloud Functions documentation for details on deploying serverless functions.

Browser platform

When packaging for browser deployment, Corticon.js generates the files:

- `decisionServiceBundle.js`: Your obfuscated rules
- `browser.sample.js`: Sample code demonstrating calling Corticon.js rules
- `browser.sample.html`: Sample HTML page for testing browser deployment

The HTML page, `browser.sample.html`, present a simple form where you can provide a JSON payload, invoke your rules, and see the resulting JSON.

The wrapper, `browser.sample.js`, simply invokes the rules with the payload input provided in the form.

The html page and wrapper are pure sample code. They just demonstrate how Corticon.js rules can be embedded into a JavaScript application running in a browser.

The `browser.sample.html` is ready to run a simple page in a browser.

Paste in the JSON you exported from Studio tester into the left panel, and then click **Run Decision Service**. The results in the right panel as shown:

Sample Calling Corticon JavaScript Decision Service

Enter the payload to pass to the decision service:


```
{
    "orderId": 494748,
    "customer": "Acme Industries",
    "customerStatus": "elite",
    "governmentAgency": false,
    "shippingAddress": {
        "address1": "1234 Industrial Lane",
        "address2": null,
        "city": "Boston",
        "state": "MA",
        "zip": "01234"
    },
    "shippingDetails": {
        "expedite": true,
        "mode": "ground"
    },
    "products": [
        {
            "sku": "XYZ-BB-43",
            "unitPrice": 2300.00,
            "quantity": 2,
            "tags": [
                "industrial",
                "compressor"
            ]
        }
    ],
    "discount": 0.0
}
```

```
{
  "orderId": 494748,
  "customer": "Acme Industries",
  "customerStatus": "elite",
  "governmentAgency": false,
  "shippingAddress": {
    "address1": "1234 Industrial Lane",
    "address2": null,
    "city": "Boston",
    "state": "MA",
    "zip": "01234"
  },
  "shippingDetails": {
    "expedite": true,
    "mode": "express"
  },
  "products": [
    {
      "sku": "XYZ-BB-43",
      "unitPrice": "2300",
      "quantity": 2,
      "tags": [
        "industrial",
        "compressor"
      ]
    }
  ],
  "discount": "0",
  "corticon": {
    "messages": {
      "message": [
        {
          "severity": "Info",
          "text": "Upper SKU items of more tha
          "ruleSheet": "OrderEntry",
          "rule": "1"
        }
      ]
    },
    "timestamp": "2021-03-29T14:55:31.944Z",
    "status": "success"
  }
}
```


Use the link below to run the decision service.

Run Decision Service

In the preceding image, note that the rules replaced the default shipping mode and added a message.

You can tailor `browser.sample.html` to build your UI around the Decision Service that you have now validated.

Multiple decision services on a browser page

An integrator can put more than one Decision Services on a single HTML page so that a set of Corticon engine execution functions are created with one execute function per included Decision Service. In the browser bundle, the file `browser.sample.multipleDS.html` provides the format for multiple decision services. In it, you see the array property **corticonEngines**. Each entry in the array contains an object literal with the **execute** function. The script inclusion order determines where in the array a specific decision is located. The first included Decision Service is available at index 0 `window.corticonEngines[0].execute`):

```
<script type="text/javascript" src="decisionServiceBundle.js" </script>
<script type="text/javascript" src="decisionServiceBundle2.js" </script>
const result1 = window.corticonEngines[0].execute(payload1, configuration);
const result2 = window.corticonEngines[1].execute(payload2, configuration);
```

You could specify a different configuration for each of the services.

To set up multiple Decision Services to run in one browser instance

1. In Studio, generate the first package as a bundle.
2. Generate the second package as a bundle with a different bundle name.
3. In the second bundle's folder, rename `decisionServiceBundle.js` to `decisionServiceBundle2.js`, and then copy it to the browser folder of the first bundle.
4. Opening `browser.sample.multipleDS.html` opens with two input and output areas for the Decision Services.

Note: If you want to add additional Decision Services, follow the pattern of steps two and three, and then revise `browser.sample.multipleDS.html` to specify each of the added Decision Services.

When you include only one decision service, you can access the execute function with either of the following code excerpts:

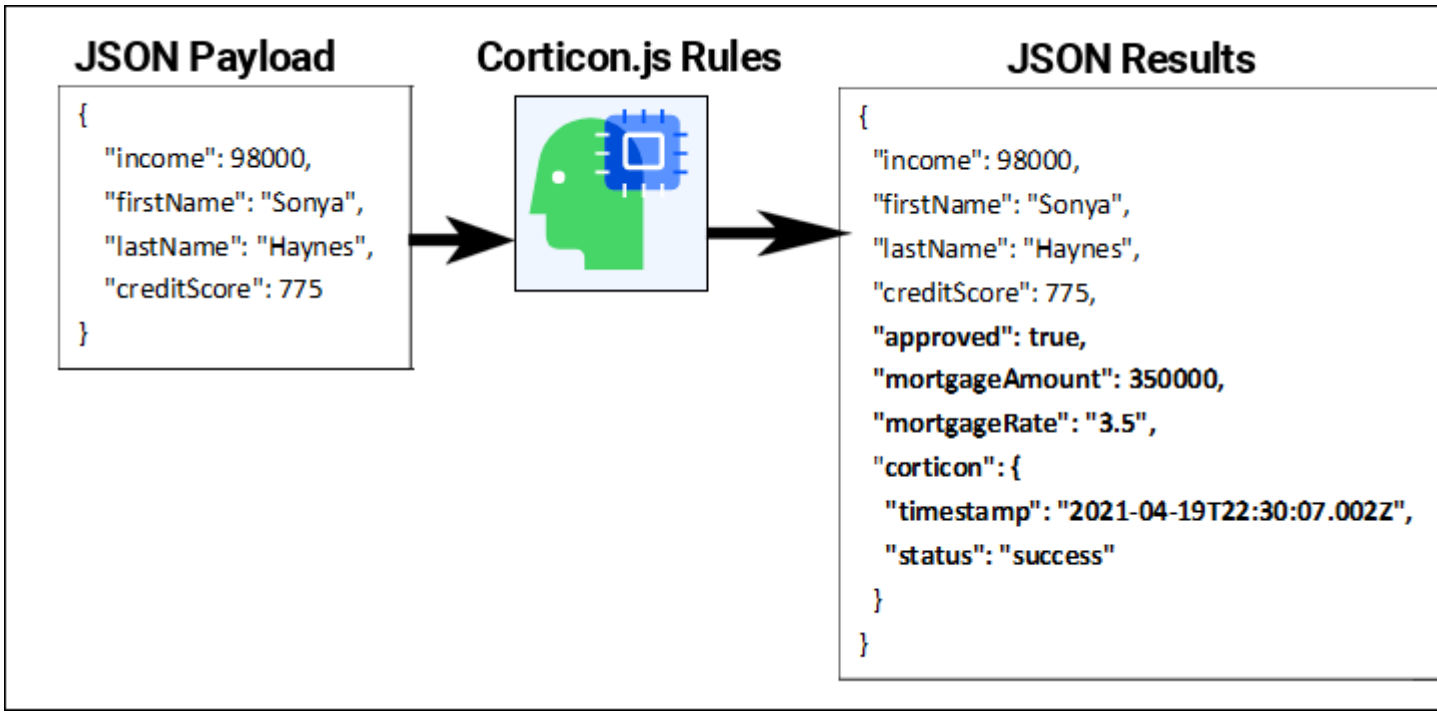
```
// Using the first array element
const result = window.corticonEngines[0].execute(payload, configuration);
```

```
// Using the last included one
const result = window.corticonEngine.execute(payload, configuration)
```

2

JSON payloads and results in Corticon.js

When deployed or integrated into your application your Corticon.js decision services execute by accepting a JSON payload and returning a JSON result. The JSON payload contains the item or items your rules are to execute on—for example, a mortgage application to be evaluated or a set of sales leads to be routed. When your rules execute, they will update or add to the input payload—for example, adding a determination for a mortgage application or assigning the sales rep for each sales lead. The JSON result returned from the decision service will contain the state of the payload after rule execution.



Corticon.js accepts most well-formatted JSON as the input payload. The result JSON will reflect the input payload plus any changes made by the rules. In addition, it will contain information about the rule execution, such as status indicating if rule execution was successful.

As a very simple example, consider a mortgage approval application where the JSON payload to your decision service is as follows:

```
{
  "income": 98000,
  "firstName": "Sonya",
  "lastName": "Haynes",
  "creditScore": 775
}
```

The rules in the decision service look at the mortgage application information to determine if the applicant should be approved, how much they are approved for, and at what rate. The rules in the decision service set the attributes `approved`, `mortgageRate` and `mortgageAmount` on the application. The result payload would be similar to:

```
{
  "income": 98000,
  "firstName": "Sonya",
  "lastName": "Haynes",
  "creditScore": 775,
  "approved": true,
  "mortgageAmount": 350000,
  "mortgageRate": "3.5",
  "corticon": {
    "timestamp": "2021-04-19T22:30:07.002Z",
    "status": "success"
  }
}
```

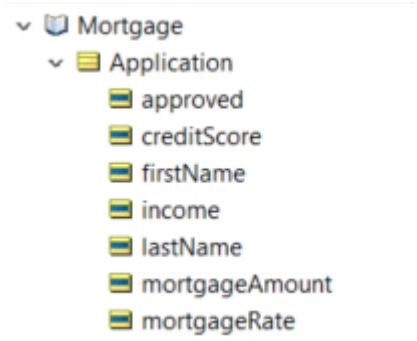

The result payload has the same data as was in the input payload plus the attributes set by the rules. In this case the applicant has been approved for $350,000 at 3.5%.

The result payload also contains a `corticon` object that has the `timestamp` of when the rules were run plus the status, `success`, of the execution. The `corticon` object contains additional information about the execution of your rules.

JSON to vocabulary mapping

When your decision service is invoked, Corticon maps the JSON payload to the internal data model of your Corticon vocabulary to enable the rules to execute on it. To perform this mapping, Corticon must first examine the JSON payload to identify the top-level objects in the JSON and the vocabulary entities they correspond to.

In our simple mortgage application example, the vocabulary could be defined as:

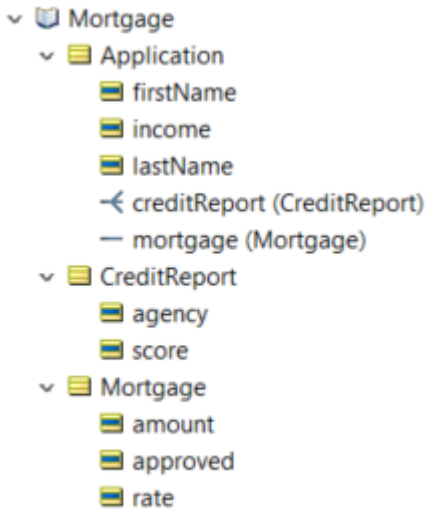


This is a very simple vocabulary with a single entity, `Application`. Our JSON payload is equally simple:

```
{
  "income": 98000,
  "firstName": "Sonya",
  "lastName": "Haynes",
  "creditScore": 775
}
```

Here the JSON payload represents a single application and Corticon would map it to the `Application` entity. In a real application your Corticon vocabulary and JSON payload are likely to be much more complex.

Once the top-level objects in the JSON payload are mapped to the vocabulary model, Corticon can map any nested objects in the JSON to associations in the vocabulary. Let's expand our mortgage sample to have a slightly more complex vocabulary:



Here the mortgage application was defined to have an `Application` entity with a one-to-many association to a `CreditReport` entity and a one-to-one association to a `Mortgage` entity. This is more reflective of a real Corticon vocabulary. A real vocabulary is likely to have many entities and associations. A JSON payload for this vocabulary may look like:

```
{
  "income": 98000,
  "firstName": "Sonya",
  "lastName": "Haynes",
  "creditReport": {
    "score": 775,
    "agency": "AccuCredit"
  }
}
```

Once Corticon has mapped the root object in the JSON to the `Application` entity, it can use the knowledge that the `Application` entity has an association to a `CreditReport` entity to map the nested `creditReport` in the JSON. Corticon will set the association on the `Application` entity to this `CreditReport` entity.

JSON Path and JSON Element Name

Corticon employs two strategies to map a JSON payload to the vocabulary. First, it will use the knowledge it has of the vocabulary to identify how to perform the mapping. When this information is insufficient, Corticon will use a "best match" strategy to perform the mapping.

In your Corticon vocabulary, you can define properties on entities, attributes, and associations to provide guidance for Corticon when mapping a JSON payload to the vocabulary. If fully utilized, Corticon may need to perform no "best match" mapping.

In the Corticon Studio vocabulary, you can set the **JSON Path** property on entities and the **JSON Element Name** property on attributes and associations.

The **JSON Path** property defines the path to the entity in the JSON payload. When JSON payload is passed to a decision service, Corticon can use this information to map entities in the payload to the vocabulary.

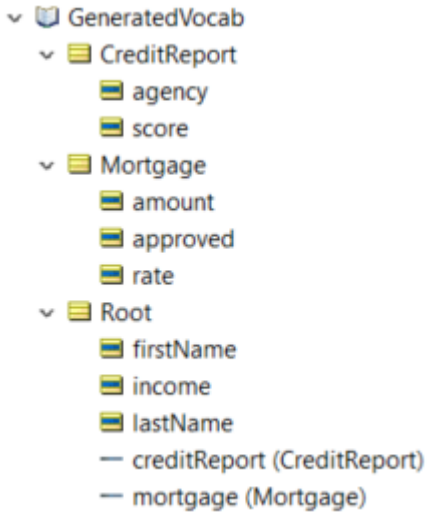
The **JSON Element Name** property defines the attribute a field in the JSON payload is mapped to or the association a nested object in the payload is mapped to.

Both **JSON Path** and **JSON Element name** can be set in the Corticon.js vocabulary editor but the easiest, and least error prone, way to set them in your vocabulary is to generate your vocabulary using the Corticon.js **Populate Vocabulary form JSON** wizard. See *"Generate a Vocabulary" in the Corticon.js Rule Modeling Guide*.

Given this JSON:

```
{
  "income": 98000,
  "firstName": "Sonya",
  "lastName": "Haynes",
  "creditReport": {
    "score": 775,
    "agency": "AccuCredit"
  },
  "mortgage": {
    "amount": 999999,
    "approved": false,
    "rate": 5.0
  }
}
```

Corticon would generate this vocabulary:



This is similar to the previous vocabulary, the only differences are that the application entity is named `Root` and the `creditReport` association is one-to-one. Where the root entity in JSON doesn't have a name, Corticon names it `Root`. It is common to rename this to something more meaningful to rule modelers. The `creditReport` association is one-to-one because there was only one credit report in the JSON.

Let's examine how **JSON Path** would be set in this vocabulary:

Entity	JSON Path
Application (aka Root)	`$`
CreditReport	`$[*].creditReport`
Mortgage	`$[*].mortgage`

JSON Path is a robust syntax for defining the path to elements in a JSON file. In our example the **JSON Path** element `$` indicates the root of the document. Where `Application` is the root entity in the JSON, its JSON Path is simply `$`. A period, `.`, Is used to identify a nested entity.

The **JSON Path** syntax can be complex but fortunately Corticon.js can set it for you when populating a vocabulary from a JSON file. In most cases, all you need set is the **JSON Path** for the root entities in the JSON.

Now let's examine how **JSON Element Name** would be set in this vocabulary:

Entity	JSON Element Name
Application.firstName	firstName
Application.lastName	lastName
Application.income	income
CreditReport.agency	agency
CreditReport.score	score
Mortgage.amount	amount
Mortgage.rate	rate
Mortgage.approved	approved

Association	JSON Element Name
Application.creditReport	creditReport
Application.mortgage	mortgage

The JSON Element Name is the name of the field or nested object in the JSON. You can rename attributes and associations in your vocabulary to be a name more meaningful to a rule modeler. If specified, the JSON Element Name must match the JSON payload.

Best Match Mapping

You don't have to specify **JSON Path** or **JSON Element Name** in your vocabulary. When not specified, Corticon.js will use a "best match" algorithm to map JSON payload to your vocabulary. This algorithm is dependent on the object and field names in your JSON closely matching the entity, attribute, and association names in your vocabulary.

You also don't have to specify the top-level objects in the JSON payload. When not specified, Corticon.js will examine the attributes on the top-level objects in the JSON payload and use them to map to the best matching entity.

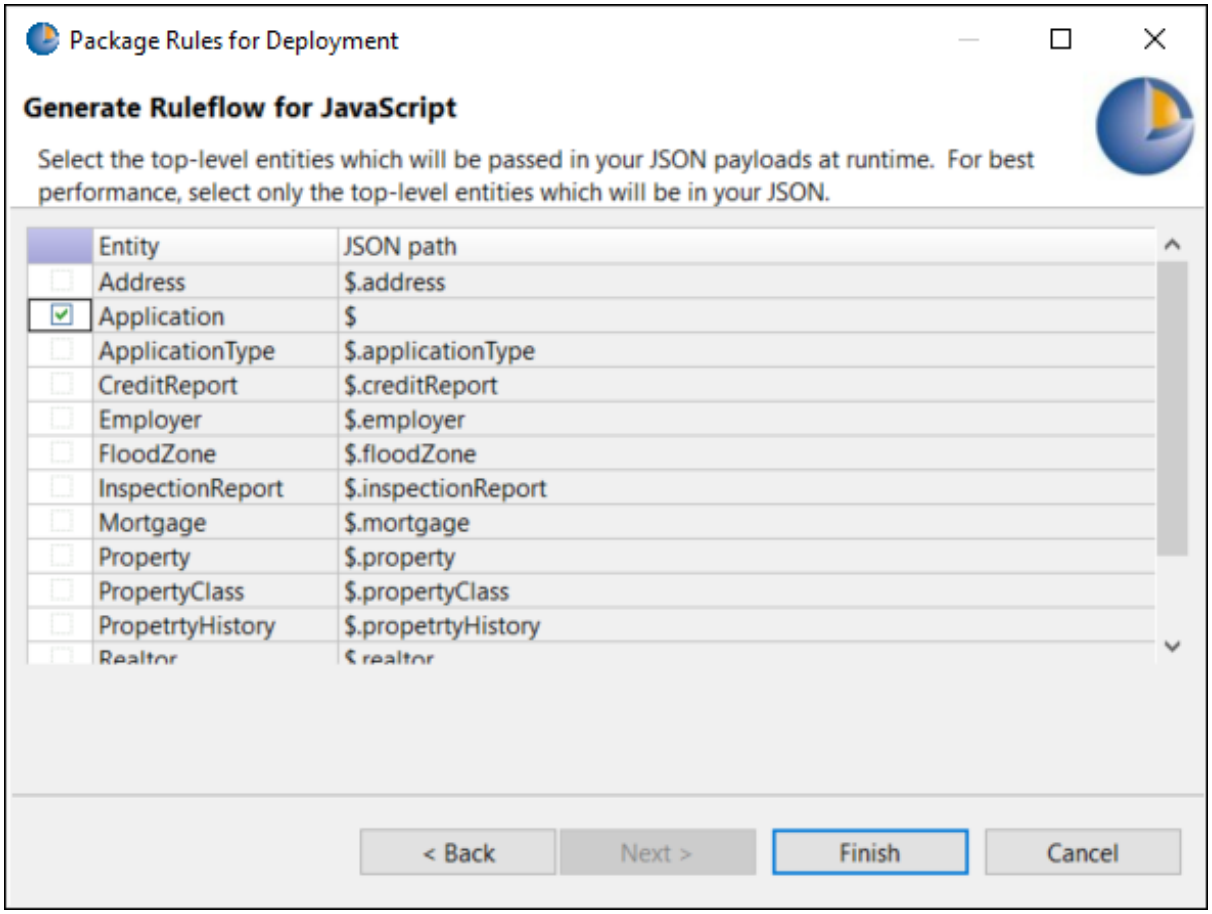
Corticon.js "best matching" is very effective in most use cases. The negative aspect is that the mapping is slower. For performance critical applications or those processing large JSON payloads, this performance impact may be significant.

At minimum, you should have **JSON Path** specified for the top-level entities in your JSON payload and identify these as the top-level entities when packaging your rules for deployment.

Identifying Top-Level Entities

When you package your rules for deployment, you can specify the top-level entities of interest in the JSON payload that will be processed when integrated with your application. This information can then be used by Corticon.js to optimize the mapping of JSON payloads to your vocabulary. It is not uncommon for a Corticon vocabulary to have hundreds of entities. If only one of these entities will be at the top-level, identifying the entity allows Corticon.js to map payloads to the vocabulary much more efficiently.

In Corticon.js Studio, the **Package Rules for Deployment** wizard provides a dialog where you can select the top-level entities.



In this example, the vocabulary was expanded to have many more entities as would be typical in a real application. Here, `Application` is selected to indicate that `Application` is the top-level entity in the JSON payloads the decision service will process.

Top-level entities do not need to be at the root of your JSON payload. Often the objects of interest to your rules may be nested in a larger JSON document.

```
{
  "vendor": "mortgageFlex",
  "vendorId": "MA3626",
  "FDO": {
    "stage": "CP3",
    "lta": "GO",
    "application": {
      "income": 98000,
      "firstName": "Sonya",
      "lastName": "Haynes",
      "creditReport": {
        "score": 775,
        "agency": "AccuCredit"
      }
    }
  }
}
```

Here the mortgage application is nested in a JSON document where the other information is not needed by the rules. Setting JSON Path for the Application entity to `$.FDO.application` will let Corticon know where it is within the JSON document. It and its nested objects are the only portions of the JSON that need to be mapped to the vocabulary.

Corticon.js allows the top-level entities to be identified when a Ruleflow is packaged for deployment. The integration API allows you to override this within your application. See *"Config options" in this guide*.

Unmapped JSON Payload

Any objects or fields in the JSON payload that are not mapped to your vocabulary are preserved and included in the result payload.

3

Customized data access operators

Corticon.js provides data access operators in a mechanism that will execute custom JavaScript functions. The custom functions are accessed as a set of built-in operators available from Studio's Rule operator tree under **General > Functions** as standalone operators, one for each Corticon.js data type:

- `getBoolean`
- `getDateTime`
- `getDecimal`
- `getInteger`
- `getString`

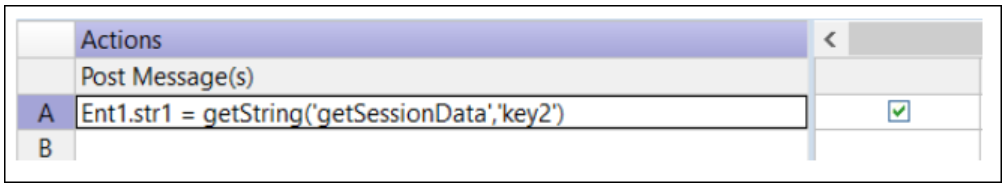
These operators all work the same way, only the returned data type is different.

How custom function work in a Rulesheet

A custom function takes two parameters:

- The name of the custom function to call
- A string parameter, which, for multiple parameters, can be a JSON string encapsulating all that needs to be passed to the custom function.

For example, to call the custom function `getSessionData` with parameter `key2` using the `getString` operator, use the following action. The result is stored in the string attribute, `Ent1.str1`.

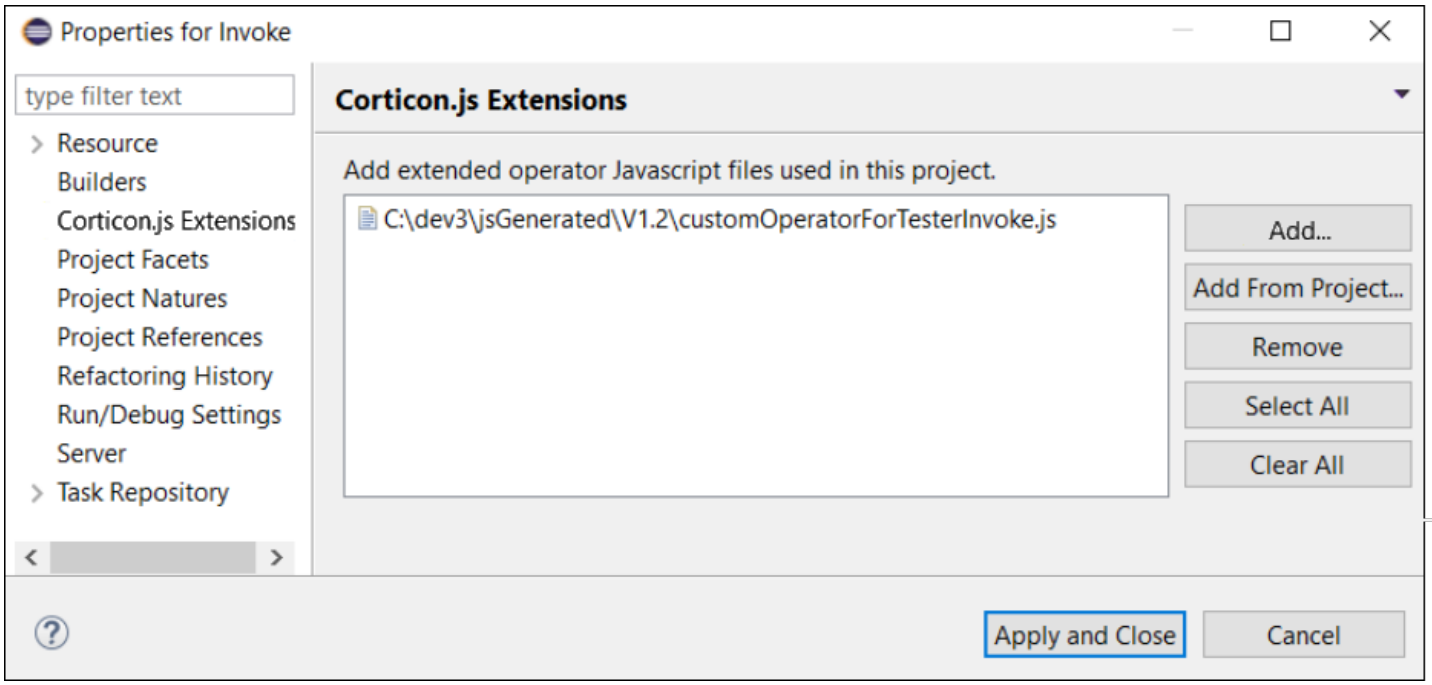


How custom functions work in a Ruletest

A Studio Ruletest runs Ruleflows and Rulesheets in the context of a local Node.js process. However, it is possible that the production environment data is not available, or the production context is not available. For example, in studio tester, the browser session context or the Serverless function context are not available.

To specify a mock Implementation in Studio Tester:

1. In Corticon.js Studio, open your project.
2. Click on your project, and then select **Properties**.
3. Select **Corticon.js Extensions**.
4. Click **Add**, and then select your .js implementation file, as illustrated:



Note: The functions from this code are not added to a production bundle that is generated by **Package Rules for Deployment**.

How to Write a Mock Implementation

The implementation file contains the custom functions. Export them using the following syntax:

`module.exports={`*`name of function as used in rulesheet`*`:`*`reference to custom function`*`};`

For example:

```
module.exports={getSessionData:getSessionData};
```

That can be abbreviated to `module.exports ={getSessionData};`

The following code shows a more complete example with two custom functions:

```
const sessionData = new Map();
sessionData.set('key1', true);
sessionData.set('key2', 'my session string');
sessionData.set('key3', 12);

function getSessionData ( helper, name ) {
 if ( !sessionData.has(name) )
  return 'ERROR - no session data for ' + name;
 else
  return sessionData.get(name);
}

function getMoreData ( helper, name ) {
 return name;
}

 module.exports = { getSessionData, "from session data": getMoreData };
```

These definitions enable you to use `getSessionData` `from session data` as custom function names in any of the simplified extended operators.

For example:

L	Ent1.int1=getInteger('getSessionData','key3')
M	Ent1.str2=getString('from session data','key2')

For information on implementation, see *"How to customize the data access operators" in the Corticon.js Integration Guide*.

How to specify implementation of custom functions for run time

Custom functions are specified at run time in the configuration object. This allows for complete separation of the custom functions from the rulesheets and its bundle.

In particular, the custom functions can easily be:

- Shared across several decision services.
- Developed and managed with your own build pipeline.

The custom functions are specified in the configuration object using the attribute "**customFunctions**".

This attribute points to an array of object literals. Each object literal contains the information for one custom function.

The syntax of the object literal is:

{name of function as used in rulesheet:reference to custom function};

For example, the following is a configuration for the custom function example:

```
const configuration = {
 logLevel: 0,
 customFunctions: [  {"getSessionData": getSessionData},
       {"from data store":getMoreData}
       ]
};
```

Function signature

A custom function must have the following signature:

`function <functionName> ( helper, name )`

The name parameter contains the second argument string as entered in the Rulesheet editor.

For example, in the Rulesheet editor:

```
Ent1.str1 = getString('getSessionData','key2')
```

The `getSessionData` custom function name parameter will contain the string `key2`.

The helper function is an object literal containing references to Decimal and DateTime operators.

It has the following syntax:

```
helper={'decimal': <all decimal operators>,
        'dateTime': <all dateTime operators> }
```

How to write the implementation of custom functions for run time

You can implement the body of a custom function with whatever logic your use case requires.

The only constraints are:

- The custom function has the proper signature.
- You return the proper object type. A `getDecimal` call needs to return a Decimal, likewise a `getDateTime` needs to return a DateTime.
- You use the helper object to create or operate on Decimal and DateTime objects.

Note: The list of operators is the same as the ones listed in the studio operator tree. See *"DateTime"* and *"Decimal" in the Corticon.js Rule Language Guide*.

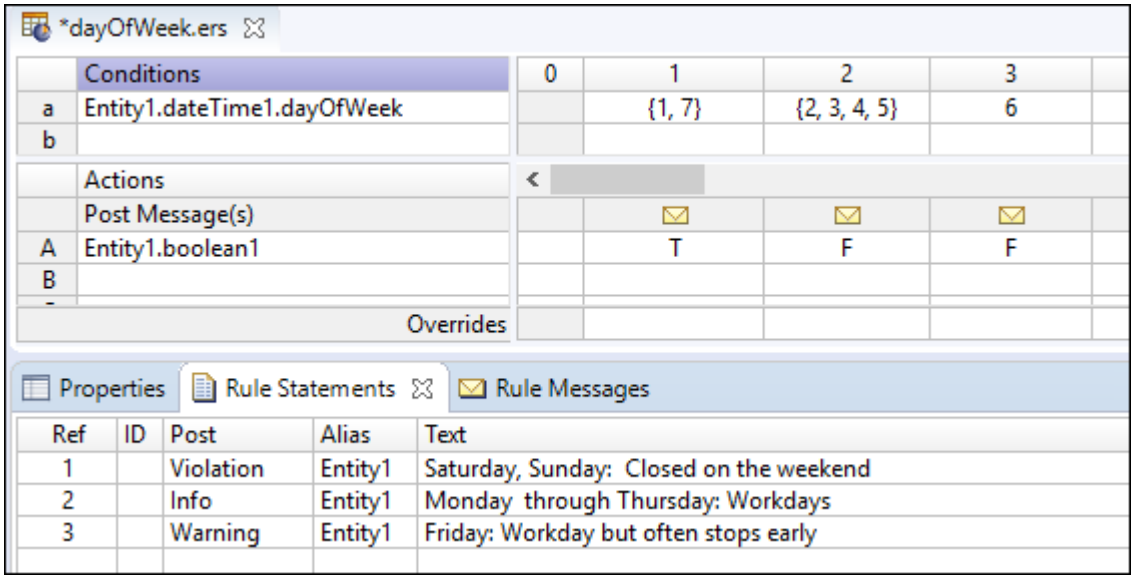
The following code shows how to construct and return a decimal and a dateTime:

```
const sessionData = new Map();
sessionData.set('key1', true);
sessionData.set('key2', 'my session string');
sessionData.set('key3', 12);

function getSessionData ( helper, name ) {
    if ( name === 'key4' ) {  // example showing how to construct a DateTime
        return helper.dateTime.constructDateTime('2021-02-10T00:00:00.000Z');
    }
    else if ( name === 'key5' ) { // example showing how to construct a Decimal
        return helper.decimal.constructDecimal('10.23');
    }
    else if ( name === 'key6' ) { // example showing how to use additional built-in
operators like now and addDays
        const dt = helper.dateTime.now();
        const dt2 = helper.dateTime.addDays(dt, 7);
        return dt2;
    }
    else if ( !sessionData.has(name) )
        return 'ERROR - no session data for ' + name;
    else
        return sessionData.get(name);
}

function getMoreData ( helper, name ) {
    return name;
}
module.exports = { getSessionData, "from session data": getMoreData };
```

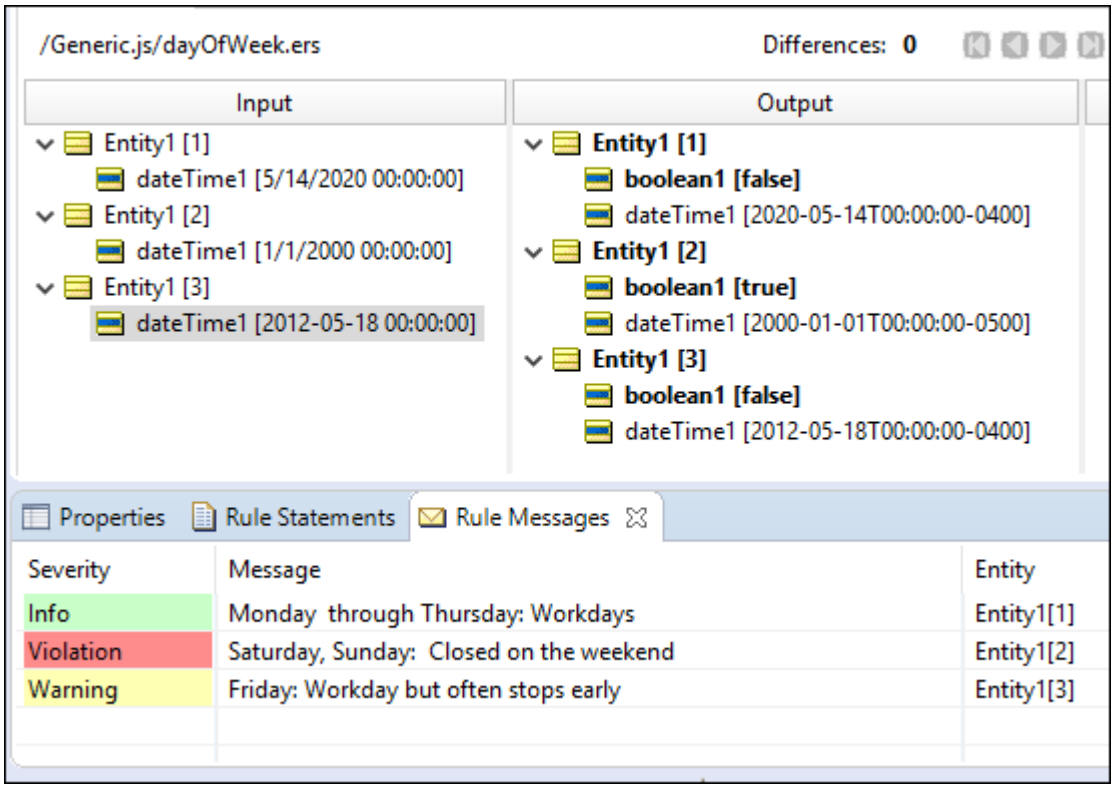
4

Rule statements and Rule messages in Corticon.js

In Studio you can add rule statements to rules. Rule statements allow you to add a message to individual rules that will be returned with the result payload from a call to the decision service. Rule statements allow you to capture "why" a decision was made. They are often used to debug rules or to capture an audit trail. Example:



When these rules execute, they produce the corresponding types of rule messages:



Configuration

Each of the target platform wrappers provide the information for implementing and managing rule messaging, as shown:

```
/*
 *******************************************************
 Configuration Properties for Rule Messages
 *******************************************************
 */
 const configuration = {
  logLevel: 0,
  ruleMessages: {
    executionProperties: {
    restrictInfoRuleMessages: true,
    // If true Restricts Info Rule Messages
    restrictWarningRuleMessages: true,
    // If true Restricts Warning Rule Messages
    restrictViolationRuleMessages: true,
    // If true Restricts Violation Rule Messages

    restrictResponseToRuleMessagesOnly: true,
    // If true the response returned has only rule messages
   },
  },
 };*/
```

Setting a configuration

The syntax of a configuration might look like this when we want to suppress violation messages, and put messages in the log:

```
const configuration = {logLevel:0,ruleMessages:
{restrictViolationRuleMessages:true,logRuleMessages:true} };
```

5

How to use the Corticon.js utilities

You can script continuous integration and continuous delivery (CI/CD) when you use the Corticon.js utilities to make changes to software rapidly and iteratively. You can use the `corticonjs` command line utility with the following options and sub-options:

- Package rules for deployment
- Generate rule reports
- Run rule tests

The utility is bundled with the Corticon JavaScript Studio installation in its `Utilities` directory.

corticonJS.bat options

```
usage: corticonJS
-c,--compile             compile a ruleflow into a decision service
-h,--help                print this message
-r,--report              generate the report for a Corticon asset
-t,--test                execute tests for a set of ruleflows or rulesheets
```

--compile options

```
usage: compile
-b,--bundle <name>     the name of the decision service bundle
-h,--help              print this message
-i,--input <file>      ruleflow (.erf file) to compile
-o,--output <folder>   folder to place the decision service Javascript bundle in
-p,--platform <target platform> target platform (Azure, Browser, Google, Lambda, Node)
-t,--top <entities>   space separated list of entity names
                                  that are expected at the root of the payload
```

--report options

```
usage: report
-c,--css <CSS file>         CSS file to use in report
-h,--help                   print this message
-i,--input <file>           Corticon asset (.ecore, .ers, .erf, .ert) to report
-if,--image <image folder>  image folder to copy to output folder
-o,--output <folder>        folder to place the report files in
-p,--project <project name> Corticon project name to use in report
-x,--xslt <XSLT file>       XSLT file to use in report
```

The files for generating reports are in your Corticon Work directory's `Reports` folder.

--test options

```
usage: test
-a,--all                          run all test sheets in the ruletest
-dj,--dependentjs <dependentJS>   comma separated list of dependent javascript files
-h,--help                         print this message
-i,--input <file>                 comma separated list of input (.ert) files to run,
                                        wildcards allowed
-o,--output <file>                file to contain output of test run
-s,--sheet <Sheet names>          comma separated list of test sheets to run
```

Note: The `test` option requires a valid JavaScript enabled license file (`CcLicense.jar`) in the `Utilities/lib` folder.

6

Config options

A config option you can use at run time defines the `topLevelEntities`, a JSON Object.

```
const config ={
 topLevelEntities:
   { "JSON PATH" : "Entities @ the JSON PATH specified"
 },
};
```

For example, where the `Patient` Entity is at root:

```
topLevelEntities: { "$" : "Patient" }
```

In another example, there are three top-level entities, so you use a comma-separated array:

```
topLevelEntities: { "$" : ["Cargo", "FlightPlan", "Aircraft"]}
```

7

How to test deployed rules

Corticon.js Studio lets you create tests that will exercise your rulesheets and ruleflows. The best practice is to create ruletests as part of the development of your rule projects.

To help you test your rules once they are deployed, Corticon.js provides a way to export the input data from your ruletests as JSON that you can then use as input to test on your deployed platform. In Corticon.js Studio, open a ruletest, and then right-click to choose **Export JSON** (or **Export JSON to Clipboard**) to export the JSON data for testing.

Your rules should execute the same way on your JavaScript deployment as they do in the Corticon.js Studio tester.

8

How to use logging and diagnostics

Default logging

Corticon.js decisions services will, by default, log messages to the default output of the platform, which is, for most JavaScript platforms, the JavaScript console. On other platforms, it is a context or similar object that the platform provides for directing logging.

Custom logger configuration

You can set the logger configuration to control what, if anything, gets logged. See the options in the sample's comments:

```
 // logLevel: 0: only error gets logged (default), 1: debugging info gets logged
 // logIsOn: when false, do not log. True by default, you can override dynamically
    to log data only for certain calls (for example by checking for a specific payload)

 // logPerfData: when true, will log performance data
 // logRulesStatements: when true, will write rule statements to the log
 // logFunction: Used to implement your own logger. When defined this function
    is called with a string message to log and an error level.

const configuration = { logLevel: 0 };
```

Custom logger

Each sample includes a simple logger where 1: log error and 2: log debug data. Your might want to log to a preferred destination, such as into a database. To do that, you specify a custom logger:

```
function myLogger(msg, logLevel) {
    if ( logLevel === 0 )
        console.error(`**CUSTOM ERROR LOGGER: ${msg}`);
    else
        console.info(`**CUSTOM DEBUG LOGGER: ${msg}`);
}
```

Then you declare that you want to use a custom logger with the `logFunction` setting:

```
/*
const configuration = logFunction; { logLevel: 1 }
```

Basic logging

Each wrapper includes a sample configuration for logger that is a function you can override dynamically when to log data. It is useful for tracing only certain calls (for example by checking for a specific payload) This function is optional. When you pass a simple configuration without the `logIsOn` property you do not need to define this function.

Logging has just two levels: 0: only errors get logged (default), 1: debugging info gets logged.

Filtered logging

Log entries can produce a lot of data, especially in debug mode. You can dynamically override when to log data, such as tracing only certain calls or payload attribute values. When you pass a simple configuration without the `logIsOn` property you don't need to define this property.

```
/*
function isLogOnForThisPayload(payload) {
 let flag;
 try {
  if ( payload.Objects[0]['int1'] === 1 )
   flag = true;
  else
   flag = false;
 }
 catch ( e ) {
  console.log (`Error in isLogOnForThisPayload: ${e}\n`);
  flag = true;
 }
 console.log(`isLogOnForThisPayload: ${flag}\n`);
 return flag;
}
```