Corticon.js Integration

Copyright

Last updated with new content: Corticon.js 1.1

Updated: 2021/03/15

Table of Contents

1

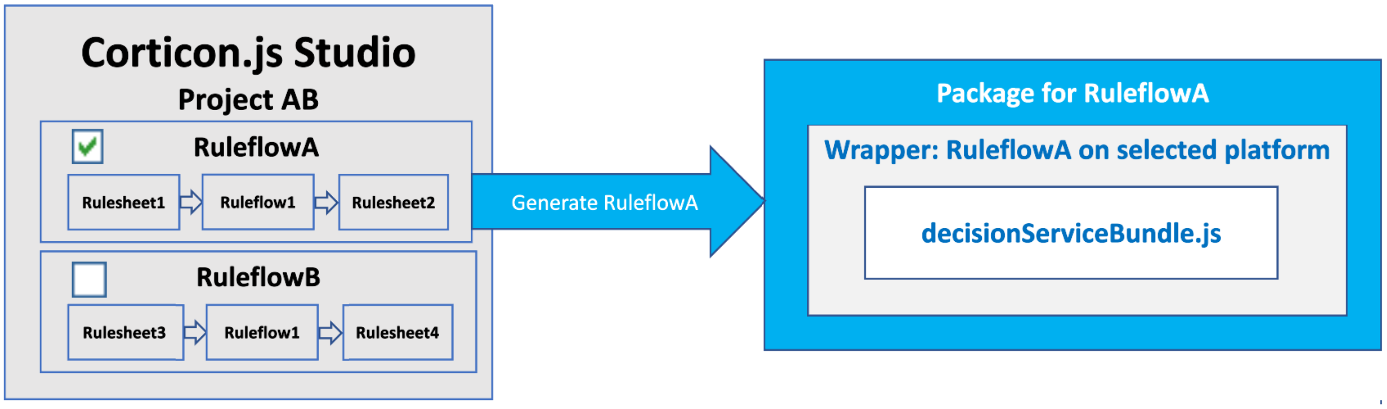
About Corticon.js integration

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**
- **Browser**
- **Node.js**

Selecting AWS Lambda or Azure 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.

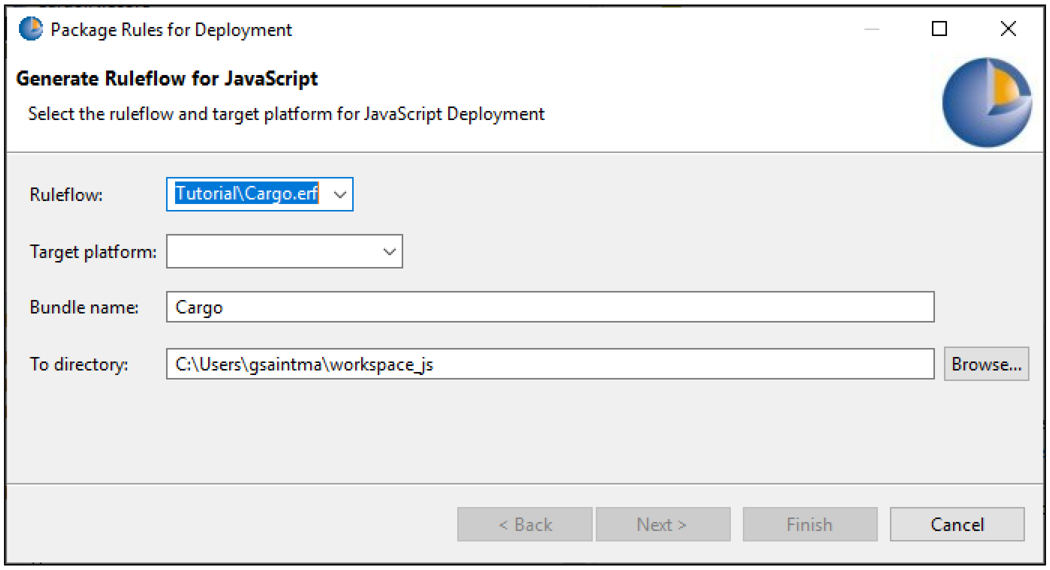
The Node.js target can be used for running decision services in Node server and as well to run them in Mobile applications created with NativeScript and ReactNative.

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 1.2 - 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 **Project > Package Rules for Deployment**.
2. The **Package Rules for Deployment** dialog box opens:



Package Rules for Deployment

Generate Ruleflow for JavaScript

Select the ruleflow and target platform for JavaScript Deployment

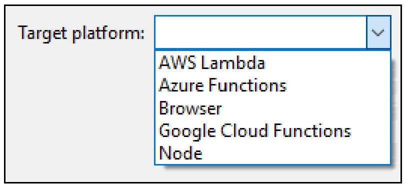
Ruleflow: Tutorial\Cargo.erf

Target platform:

Bundle name: Cargo

To directory: C:\Users\gsaintma\workspace_js

3. Choose the **Target Platform**:



Target platform:

- AWS Lambda
- Azure Functions
- Browser
- Google Cloud Functions
- Node

4. Select the Ruleflows for your application. Note that the **JavaScript File Name** can be edited so that you produce your package a distinctly named folder. Each Ruleflow selected generates a separate package.
5. Click **Finish** to generate the packaged JavaScript as Decision Services plus everything you need to deploy each of them. 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 each 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 four 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 31 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:

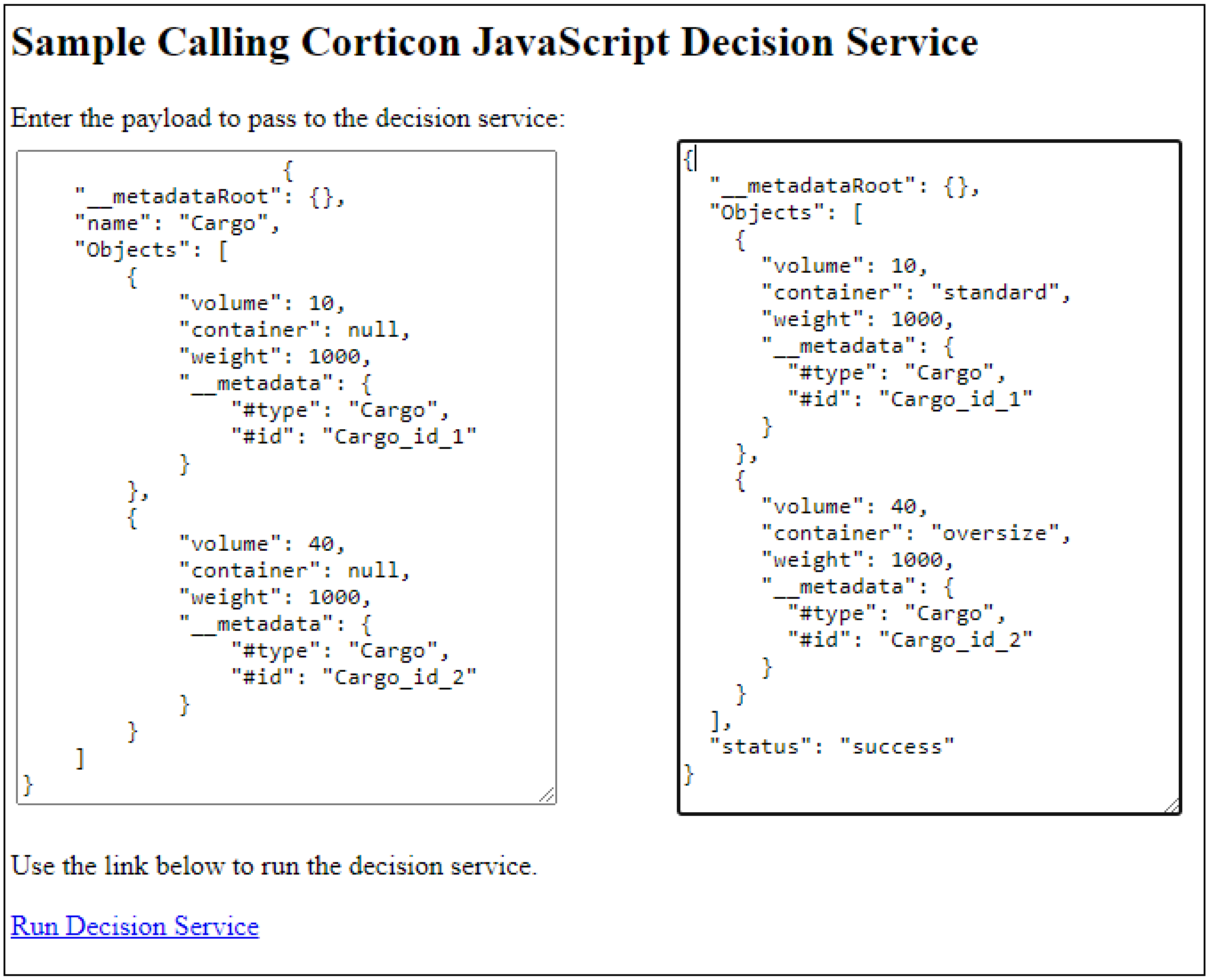
- `decisionServiceBundle.js`: Your obfuscated rules
- `browser.sample.js`: Sample code demonstrating calling Corticon.js rules
- `browser.sample.html`: Sample HTML page for testing browser deployment

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:



Sample Calling Corticon JavaScript Decision Service

Enter the payload to pass to the decision service:

```
                {
    "__metadataRoot": {},
    "name": "Cargo",
    "Objects": [
        {
            "volume": 10,
            "container": null,
            "weight": 1000,
            "__metadata": {
                "#type": "Cargo",
                "#id": "Cargo_id_1"
            }
        },
        {
            "volume": 40,
            "container": null,
            "weight": 1000,
            "__metadata": {
                "#type": "Cargo",
                "#id": "Cargo_id_2"
            }
        }
    ]
}
```

```
{
  "__metadataRoot": {},
  "Objects": [
    {
      "volume": 10,
      "container": "standard",
      "weight": 1000,
      "__metadata": {
        "#type": "Cargo",
        "#id": "Cargo_id_1"
      }
    },
    {
      "volume": 40,
      "container": "oversize",
      "weight": 1000,
      "__metadata": {
        "#type": "Cargo",
        "#id": "Cargo_id_2"
      }
    }
  ],
  "status": "success"
}
```

Use the link below to run the decision service.

Run Decision Service

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 in the illustration. You can tailor `browser.sample.html` to build your UI around the decision service that you have now validated.

2

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)
-r,--root <entities>   space separated list of entity names 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
-ll,--loglevel <Level>            level of detail to write to log file
-lp,--logpath <Path>              directory path for log file
-o,--output <file>                file to contain output of test run
-s,--sheet <Sheet names>          comma separated list of test sheets to run
```

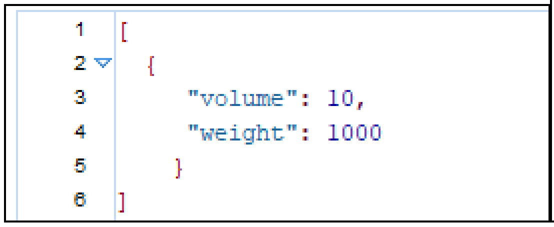
Note: The `test` option requires a valid JavaScript enabled license file (`CcLicense.jar`) in the `Utilities/lib` folder.

3

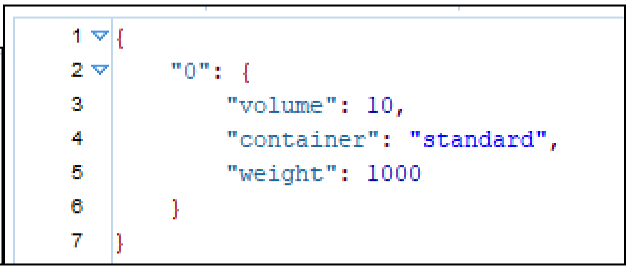
JSON processing in Corticon.js

Introduction

The processing of rules generated from a Corticon.js project is very straightforward. You pass well-formatted JSON in to a Decision Service, and you get well-formatted JSON back that includes the results of rule execution. This example shows a sample where the input on the left (`volume` and `weight`) was evaluated by rules to the provide the response (`container`) on the right:

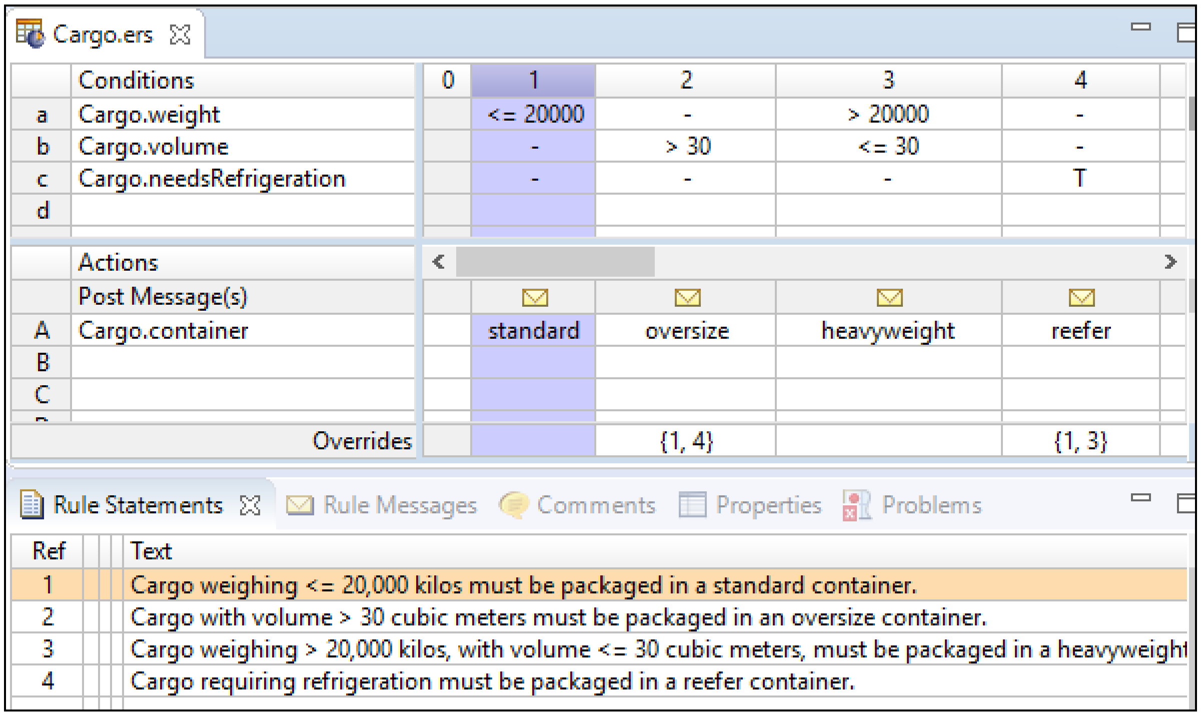


```
[
  {
     "volume": 10,
     "weight": 1000
  }
]
```



```
{
    "0": {
        "volume": 10,
        "container": "standard",
        "weight": 1000
    }
}
```

The decision service that was called has one set of rules:



Cargo.ers

	Conditions	0	1	2	3	4
a	Cargo.weight		<= 20000	-	> 20000	-
b	Cargo.volume		-	> 30	<= 30	-
c	Cargo.needsRefrigeration		-	-	-	T
d						

	Actions					
	Post Message(s)		✉	✉	✉	✉
A	Cargo.container		standard	oversize	heavyweight	reefer
B						
C						
	Overrides			{1, 4}		{1, 3}

Rule Statements | Rule Messages | Comments | Properties | Problems

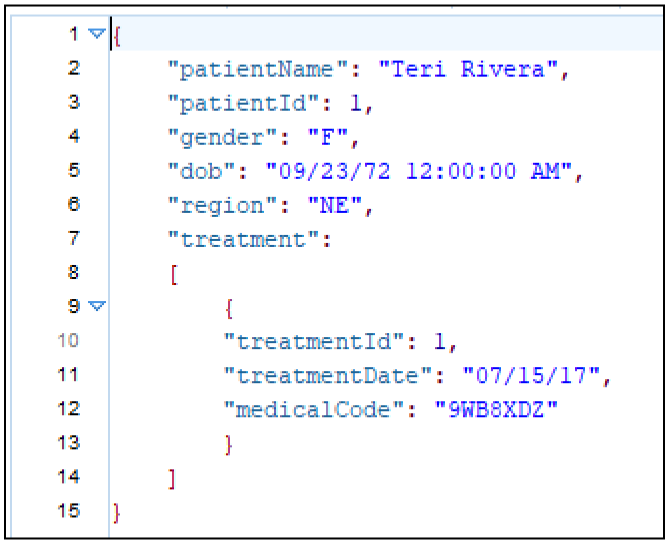
Ref	Text
1	Cargo weighing <= 20,000 kilos must be packaged in a standard container.
2	Cargo with volume > 30 cubic meters must be packaged in an oversize container.
3	Cargo weighing > 20,000 kilos, with volume <= 30 cubic meters, must be packaged in a heavyweight
4	Cargo requiring refrigeration must be packaged in a reefer container.

When the rules ran, they applied changes to the input payload, and then returned the result payload with the changes. The conditions in the request were met in rule column 1 so the action assigned the standard container. Other of rules could determine the shipping cost for the container, the flight plan and assigned airplane, a rebate amount for a customers, and insurance coverage for the container.

Create a Vocabulary from a JSON response

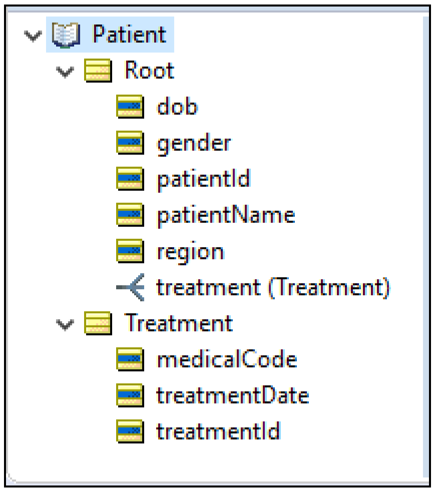
If you have a schema, you can get a rich description of how to define a Vocabulary. However, just having a JSON response can be interpreted by Corticon.js to infer the entities, attributes, and associations that comprise a Vocabulary.

For example, here is a JSON response that can be the foundation of a Corticon.js vocabulary: :



```
{
    "patientName": "Teri Rivera",
    "patientId": 1,
    "gender": "F",
    "dob": "09/23/72 12:00:00 AM",
    "region": "NE",
    "treatment":
    [
        {
        "treatmentId": 1,
        "treatmentDate": "07/15/17",
        "medicalCode": "9WB8XDZ"
        }
    ]
}
```

When that response populates a Corticon.js vocabulary, the vocabulary looks like this:

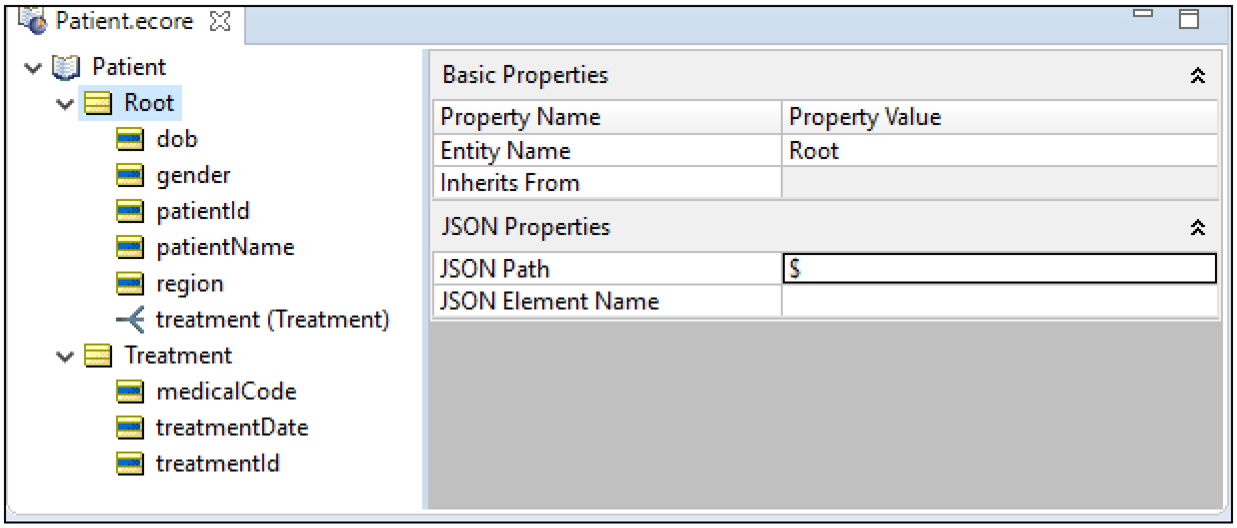


From just the minimal JSON tags, the Vocabulary was interpreted as:

- An entity of an unspecified name, which defaulted to the name `Root`
- The several attributes of the `Root` entity
- Another entity named `treatment` that is an association from `Root`
- The several attributes of the `treatment` entity
- The the one-to-many association between `Root` and `treatment`

The notion of `Root` was the default name for two reasons:

1. The JSONObject that is at the base level of the source file had no name, so it was given the name `Root`.
2. The base level of the source file is maintained in the Vocabulary as the **JSON Path** value $.



Basic Properties

Property Name	Property Value
Entity Name	Root
Inherits From	

JSON Properties

JSON Path	$
JSON Element Name	

Corticon.js will map input payloads into that internal data model defined by the vocabulary for processing. Only entities and attributes that can be mapped to the vocabulary are available to the rules.

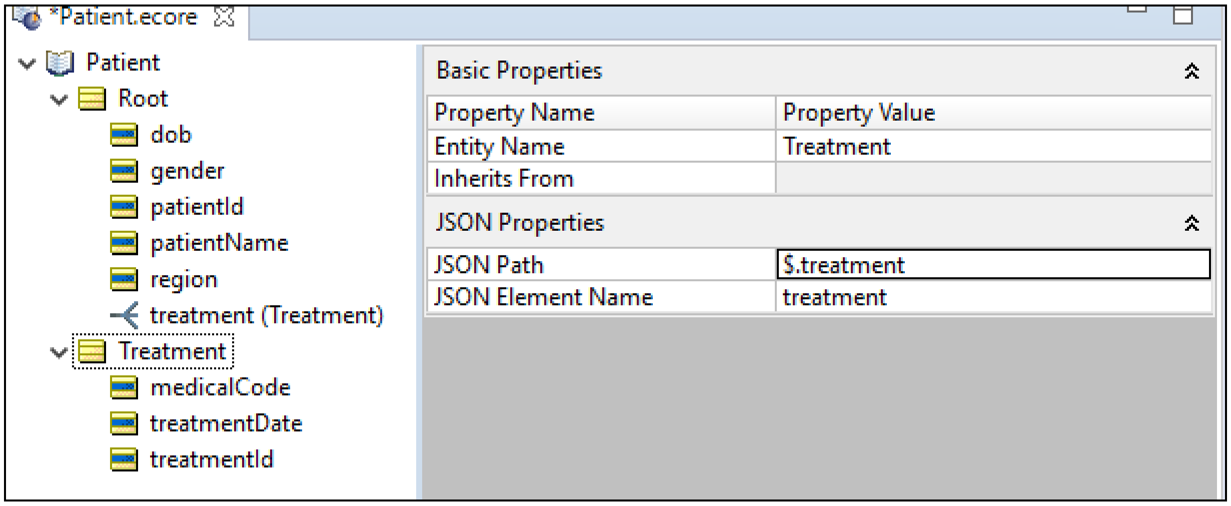
Corticon.js accepts your JSON as-is. There is no need to annotate the JSON. This is important because it allows you to connect your decisions with other services without any transforms. That's useful, especially when wiring together cloud services or calling decisions from a trigger in a SaaS application.

Because you generated your vocabulary from a JSON source, the JSON paths for the entities are stored in the vocabulary and automatically set when you package for deployment.

Note: When defining a vocabulary manually, you need not define `Root` and **JSON path**. See TBD section for more information on `Root` and **JSON path** .

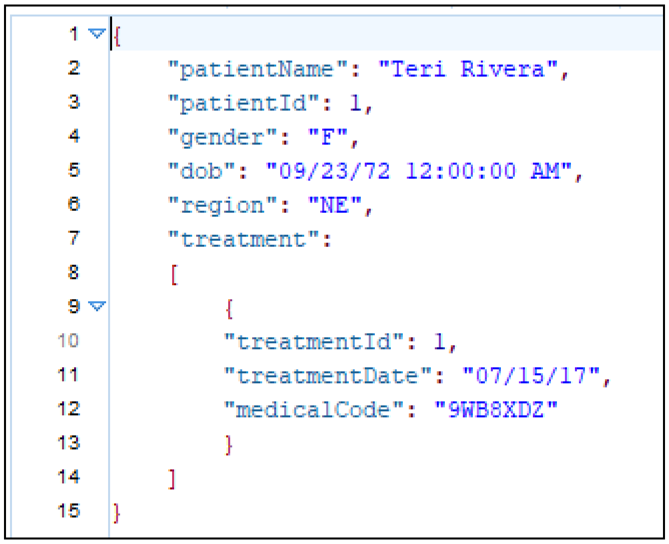
Associations

After the top-level entities are defined, Corticon can map child objects in the JSON by the associations defined on the entities in the vocabulary. Related entities are hierarchical relationships that will have a **JSON Path** value relative to `Root`, like this:



Property Name	Property Value
Entity Name	Treatment
Inherits From	
JSON Path	$.treatment
JSON Element Name	treatment

The source file had these notations that used `[ {` to distinguish the hierarchical subset `treatment`:



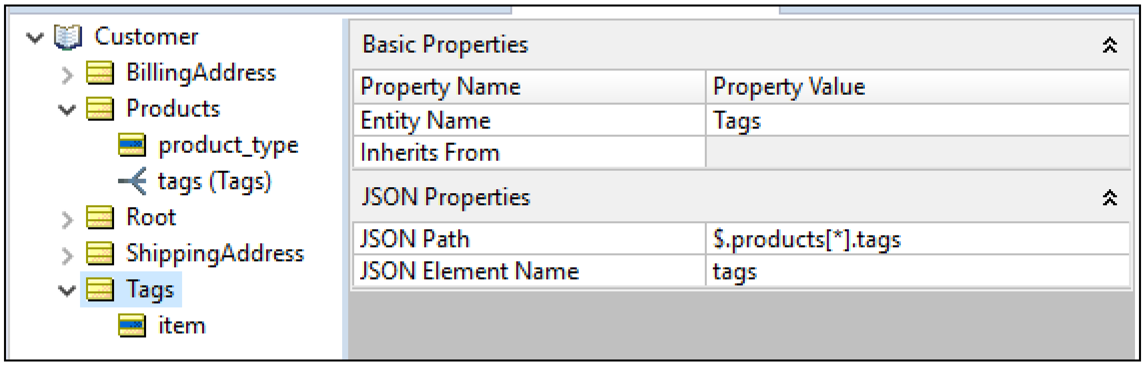
```
{
    "patientName": "Teri Rivera",
    "patientId": 1,
    "gender": "F",
    "dob": "09/23/72 12:00:00 AM",
    "region": "NE",
    "treatment":
    [
        {
        "treatmentId": 1,
        "treatmentDate": "07/15/17",
        "medicalCode": "9WB8XDZ"
        }
    ]
}
```

Arrays

Object arrays (A->[B]->[C]

• Show sample JSON

After populating the Vocabulary, the array is represented an association between "products": [{ "product_type": "Shirts", "tags": ["mensware", "tshirts"] }]`products` and one or more `tags`:



Property Name	Property Value
Entity Name	Tags
Inherits From	

JSON Path	$.products[*].tags
JSON Element Name	tags

How data types are mapped

- For a JSON schema where a data type is specified, use that data type.
- For a JSON instance:
  - For a number that can be successfully converted to a relevant Java Date, set its data type as `DateTime`.
  - "products": [{For a number with a decimal point, set its data type as `Decimal`.
  - For a number without a decimal point, set its data type as `Integer`.
  - For a string that is an ISO 8601 value, set its data type as `DateTime`, else it is a `String`.
  - For an attribute with a data type of null, it is a `String`.
  - For an empty array, it is a `String` array.

o How arrays of scalar values are mapped

This is arrays of strings, numbers, …

Show sample JSON

Show how it's associations in the vocab

o Root entity

Top level entity in JSON

Show sample JSON

May or may not be in the vocabulary

- JSON path

o This section would give specific on the JSON path syntax (not the full syntax, just what we care about)

o Back with examples showing JSON and the path specification for different elements

Cortion.js API (Appendix)

- Suggest adding this to have a home for all the details of the api

- This will allow other sections to stay high level

4

How to implement custom functions at run time

How to Specify Implementation of Custom Functions for Runtime

The custom functions are specified at runtime in the configuration object. This allows for complete separation of the custom functions from the rulesheet(s) 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 Runtime

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.

The list of operators is the same as the one listed in the studio operator tree.

The following code shows to return (construct) decimals or dateTimes:

```
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;
}
```

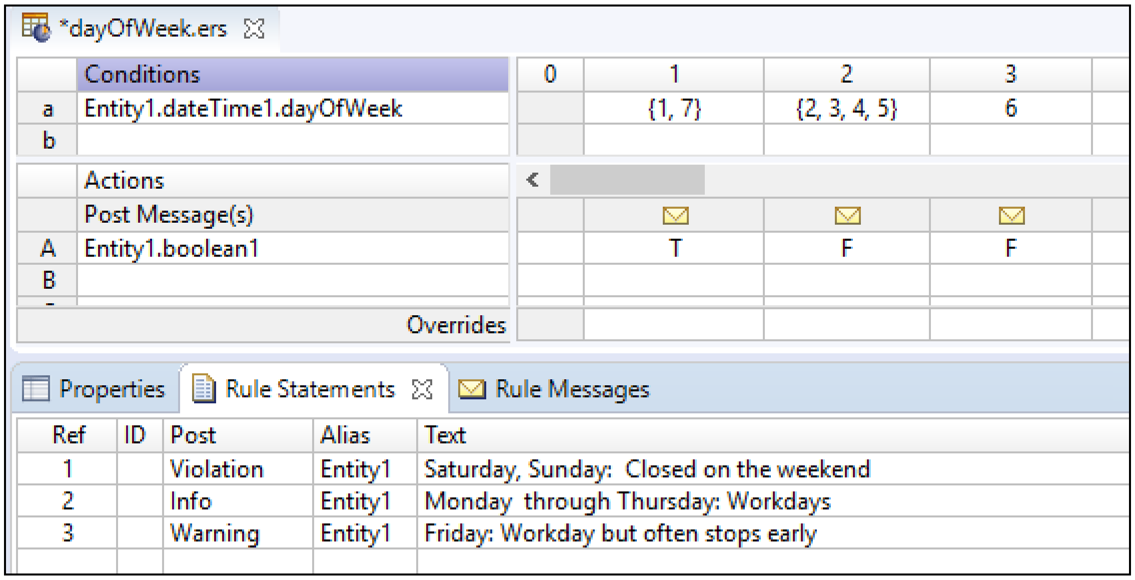
For information on creation of custom functions, see

"How to create extensions as Corticon.js language operators" in the Corticon.js Rule Modeling Guide.

5

Rule statements and Rule messages in Corticon.js

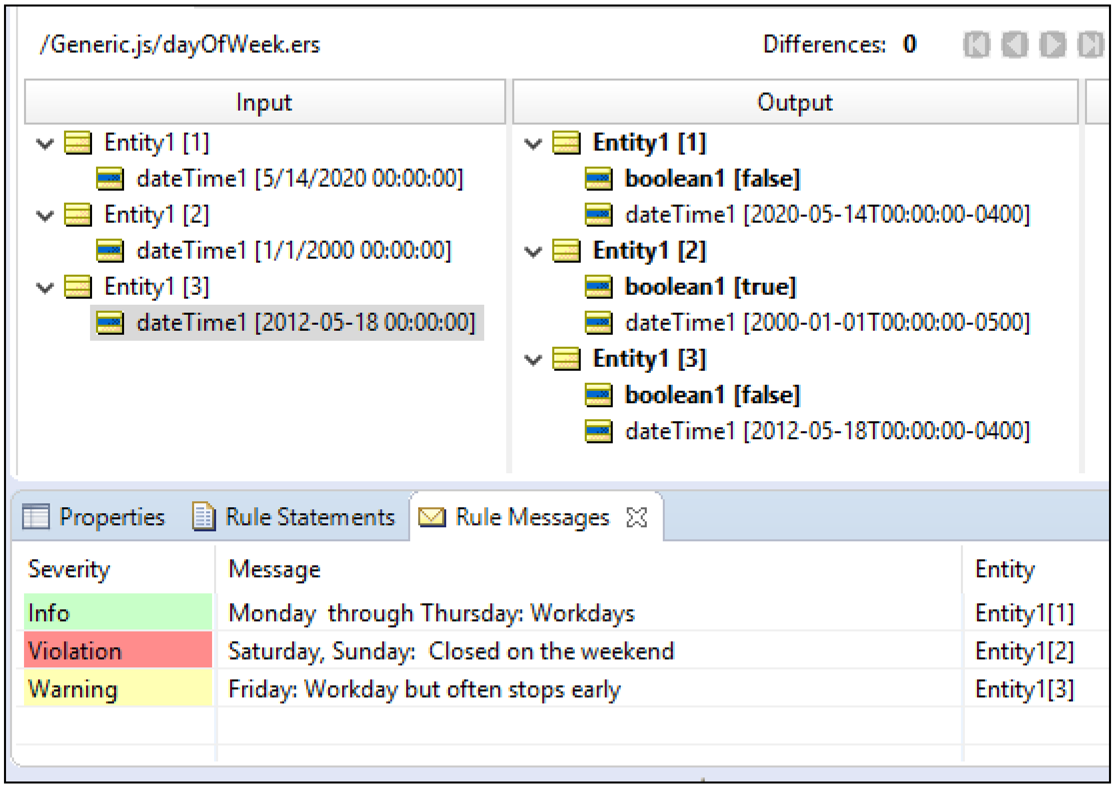
In Studio you can add rule statements to rules. Rule statements allow you to add a message to individual rules that will be returned with the result payload from a call to the decision service. Rule statements allow you to capture "why" a decision was made. They are often used to debug rules or to capture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



	Conditions	0	1	2	3
a	Entity1.dateTime1.dayOfWeek		{1, 7}	{2, 3, 4, 5}	6
b					
	Actions				
	Post Message(s)		✉	✉	✉
A	Entity1.boolean1		T	F	F
B					
	Overrides				

Ref	ID	Post	Alias	Text
1		Violation	Entity1	Saturday, Sunday: Closed on the weekend
2		Info	Entity1	Monday through Thursday: Workdays
3		Warning	Entity1	Friday: Workday but often stops early

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


/Generic.js/dayOfWeek.ers

Differences: 0

Input

- Entity1 [1]
  - dateTime1 [5/14/2020 00:00:00]
- Entity1 [2]
  - dateTime1 [1/1/2000 00:00:00]
- Entity1 [3]
  - dateTime1 [2012-05-18 00:00:00]

Output

- **Entity1 [1]**
  - **boolean1 [false]**
  - dateTime1 [2020-05-14T00:00:00-0400]
- **Entity1 [2]**
  - **boolean1 [true]**
  - dateTime1 [2000-01-01T00:00:00-0500]
- **Entity1 [3]**
  - **boolean1 [false]**
  - dateTime1 [2012-05-18T00:00:00-0400]

Severity	Message	Entity
Info	Monday through Thursday: Workdays	Entity1[1]
Violation	Saturday, Sunday: Closed on the weekend	Entity1[2]
Warning	Friday: Workday but often stops early	Entity1[3]

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} };
```

6

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.

7

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;
}
```