Progress®
Corticon®

Corticon.js Extensions

Copyright

Last updated with new content: Corticon.js 2.3

Updated: 2025/04/23

Table of Contents

1

About Corticon.js extensions for service callouts

Many rule applications need to access external data within the context of rules. The service callout feature in Corticon.js provides users with ways to enrich their JSON payload from external data. The most typical case is for payload enrichment where a fragment of the payload to be processed is passed to a decision service and the rules determine what additional data is needed.

You determine the implementation of the callouts in Ruleflow nodes. You can provide an implementation to access external data sources as well as one to access the payload being processed; thereby, enabling you to:

- Execute custom JavaScript code that can change the payload by doing various computations and data manipulations, or by accessing external data.
- Enrich the payload with data retrieved from an external data source.
- Persist the payload, or a portion of it, in an external data source.

Note: corticon.js-samples repository—The samples referenced in this document are derived from GitHub samples in the repository for Corticon.js, Corticon.js Samples Repository. You could export the entire zip at the root level and import the .zip subset **ServiceCallout** into a workspace to ensure that the assets use the correct extensions. Then run **Upgrade rule assets** to get the files updated. Be sure to add the `.js` files to the project's **Properties** tab **Corticon.js Extensions**.

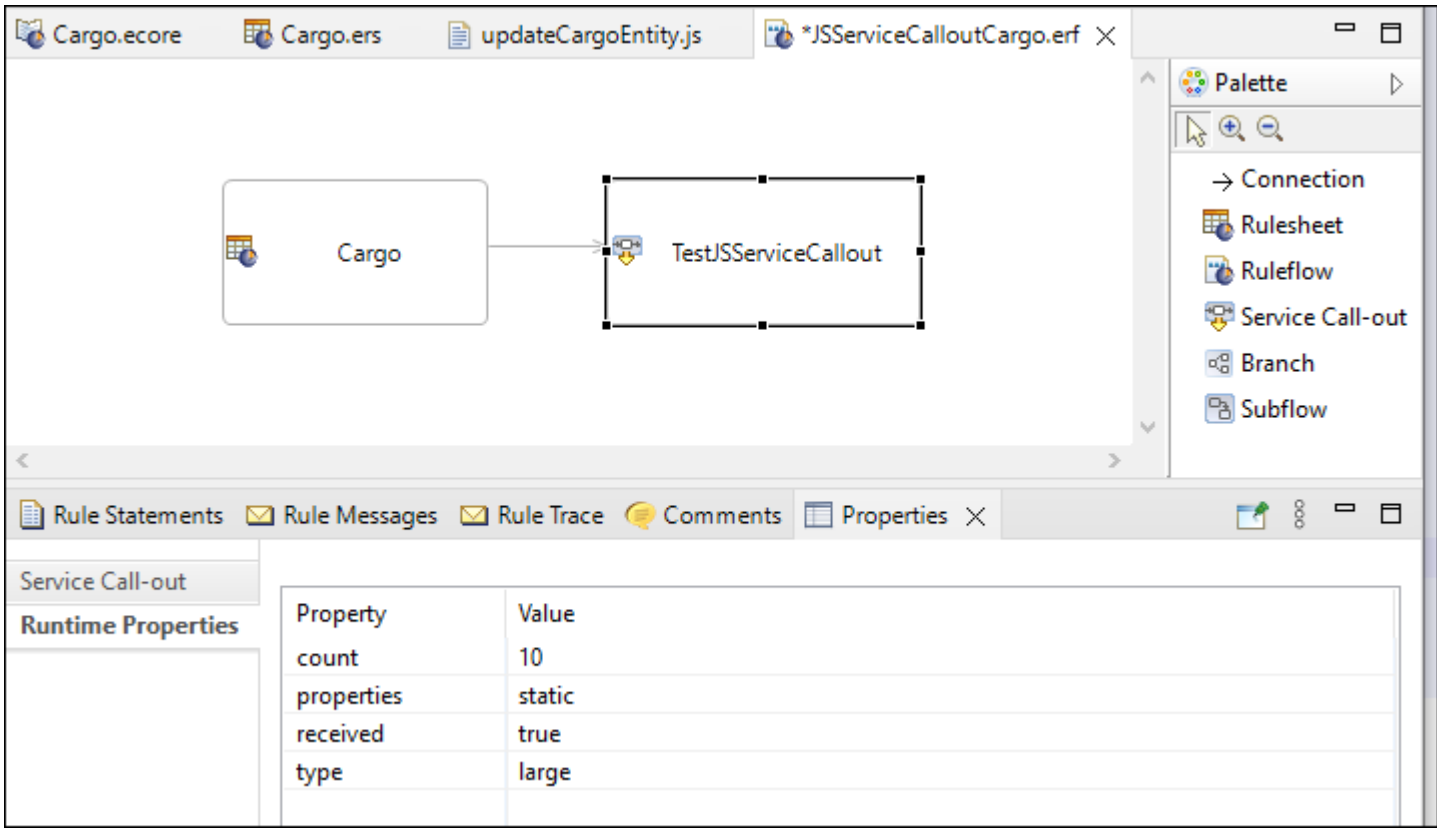
2

Add static properties to a Service Callout

You can pass static properties to your service callouts that will be passed in as a JSON object to the service callout. These properties are added in the Studio Ruleflow editor. The properties will be embedded in the decision service bundle and cannot be modified at runtime.

Note: IMPORTANT: These properties will be clear text in the Decision Service. As such, you should not add user name and password properties to the Service Callout, or any secret and confidential data here or in the payload.

These can be added from the Studio in the Ruleflow editor's **Properties** pane as shown:



How to use the API to access properties

To access the static runtime properties in service callouts, change the method signature of your API to have a second parameter, `serviceCalloutProperties`. For example:

```
function getAllEntitiesImpl(corticonDataManager, serviceCalloutProperties) {
//Your service callout function
}
```

When you access the `serviceCalloutProperties` Object, the properties are name value pairs. For example:

```
{"count":"10", "properties":"static", "received":"true", "type":"large"}
```

You can access any static runtime properties defined in the UI by using its name. For example:

```
serviceCalloutProperties['count']
```

3

Define a Corticon.js Service Callout

To define a Corticon.js service callout in JavaScript, you must follow guidelines for its four parts:

1. Metadata definition
2. Service callout function implementation
3. Configure runtime properties
4. Export actions

Metadata definition

The metadata definition of the service callout is what gets populated and displayed in the Corticon.js Studio Ruleflow Editor. Here is an example of the metadata definition of a Service Callout:

```
const updateData = {
    func: 'updateCargoEntity',
    type: 'ServiceCallout',
    description: {'en_US': 'updates the entity'},
    extensionType: 'SERVICE_CALLOUT',
    name: { 'en_US': 'updateData' }
};
```

The metadata definition has the following fields:

- **func** (required) - The actual functional definition of the service callout code that gets executed at runtime by the Corticon engine.
- **type** (required) - `ServiceCallout` A string value that defines the type required by a Corticon.js Studio Service Callout.

- **description** - Object, where the keys are locales (as strings) and the values are localized strings used for the description of the service callouts. For example, an entry to the US English localized name: `{'en_US': 'updates the entity'}` The description gets displayed in Corticon.js Studio.
- **extensionType** (required) - `SERVICE_CALLOUT` - The extension type for a service callout in case sensitive text `SERVICE_CALLOUT` . If the extension type is not defined, the Corticon.js Studio will not see it as a service callout.
- **name**(required) - Object, where the keys are locales (as strings) and the values are localized strings used for the name For example, the US English localized name. `{ 'en_US': 'updateData' }`

Service Callout function implementation

To implement the service callout function, the function name must match the name defined in the service callout metadata definition. For example in the above metadata definition, the service callout function name is `updateCargoEntity`, so the function should also have that name. Here is an example of the service callout function that gets entities by type `Cargo`, and then updates the volume of the `Cargo` that has the container `oversize`:

```
function updateCargoEntity(corticonDataManager) {
const entities = corticonDataManager.getEntitiesByType('Cargo');
entities.forEach(entity => {
    if(entity['container'] === 'oversize') {
        //Update the attribute value to 50
        entity['volume'] = 50;
    }
});
}
```

Configure runtime properties

You can add properties to the service callout so that, for example, when a REST service needs authentication, you can pass in credentials as part of the configuration that just sits in the wrapper, and then propagates the configuration to the service callout. These properties are outside the bundle, in the wrapper. That means that you do not need to redeploy the decision service every time you want to, say, change the log level from 0 to 1. These are overrides to the properties in a Ruleflow that are static service callout settings.

As configuration runtime properties, you can have your own encryption-decryption mechanisms. You can pass in your secret. You will need to have your decryption logic in your service callout function that can read these properties, and then use them as needed in your service callout function. Configuration runtime properties are not embedded in the decision service bundle JavaScript file.

Export actions

Specifies the type of export, and then an implementation you have defined. For example:

```
exports.updateCargoEntity = updateCargoEntity;
```

The completed definition

The completed definition here is:

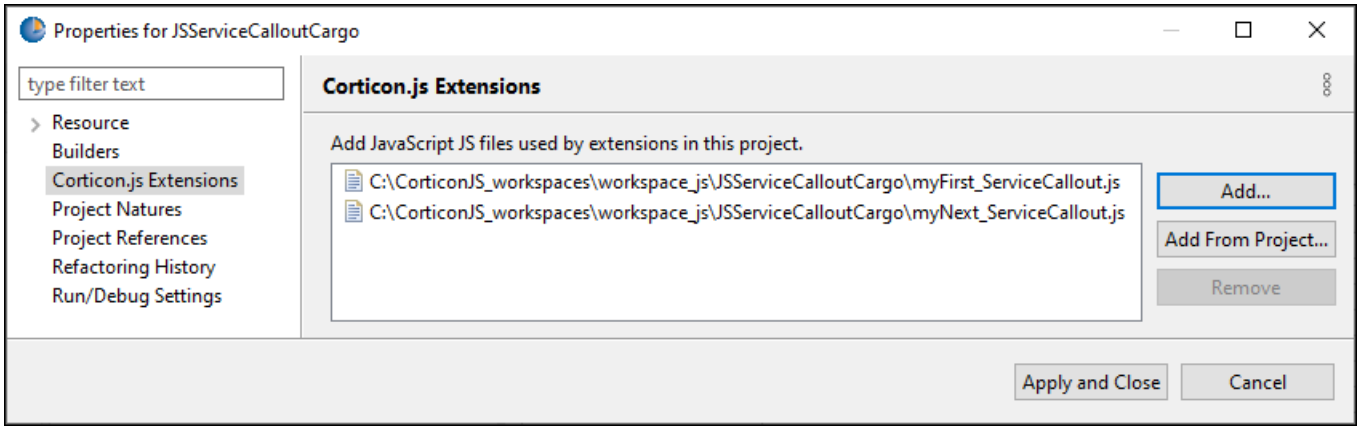
```
//Metadata
const updateData = {
    func: 'updateCargoEntity',
    type: 'ServiceCallout',
    description: {'en_US': 'updates the entity'},
    extensionType: 'SERVICE_CALLOUT',
    name: { 'en_US': 'updateData' }
};

//Service Callout Function Implementation
function updateCargoEntity(corticonDataManager) {
const entities = corticonDataManager.getEntitiesByType('Cargo');
entities.forEach(entity => {
    if(entity['container'] === 'oversize') {
        //Update the attribute value to 50
        entity['volume'] = 50;
    }
});
}
//Config
const configuration = { logLevel: 0, "serviceCallout": { "prop1": "abc" , "prop2": 12
} };

//Export Actions
exports.updateCargoEntity = updateCargoEntity;
```

4

Add your Service Callouts to a Corticon.js project

Corticon.js Service Callout functions must be defined in a JavaScript file with the extension `.js` file. The naming convention of each `.js` file must end in `ServiceCallout.js` to be recognized in Corticon.js Studio as a service callout definition.

The one or more service callout files are added to the Corticon.js Project's **Preferences** using the **Corticon Extensions** wizard, as shown:

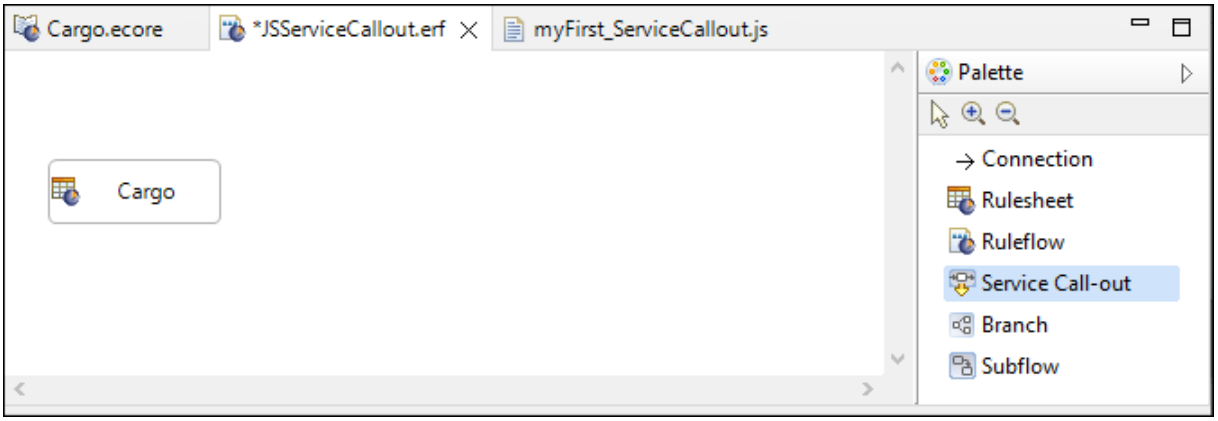


Dependent JavaScript files used by the service callouts must be added for the project to compile. For example, if you have a service callout that has a dependency on a third-party JavaScript library, the dependent JS library needs to be added to the Corticon extensions as well so that the service callouts and the third-party dependent libraries are compiled, and are included in the JavaScript bundle.

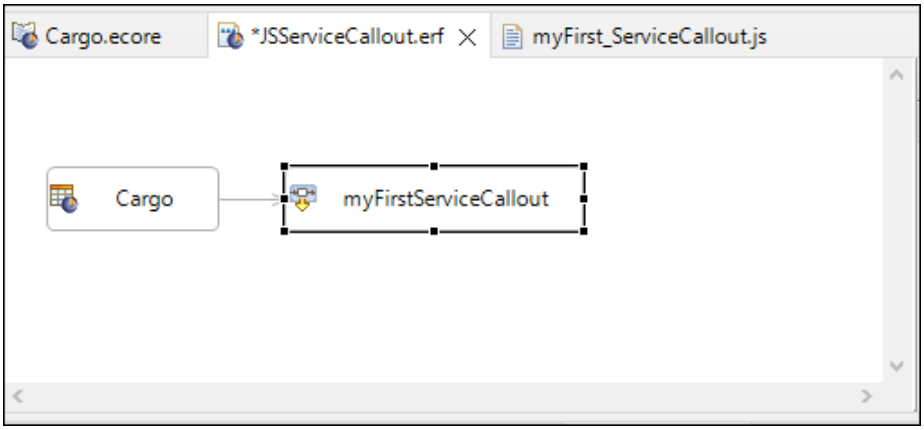
Add Service Callout Nodes to a project Ruleflow

After the service callout files have been added to the Corticon Extensions, you can use the service callouts in the project.

1. In a Ruleflow editor where the Rulesheet has been dragged and dropped, double-click **Service Call-out** in the Palette:



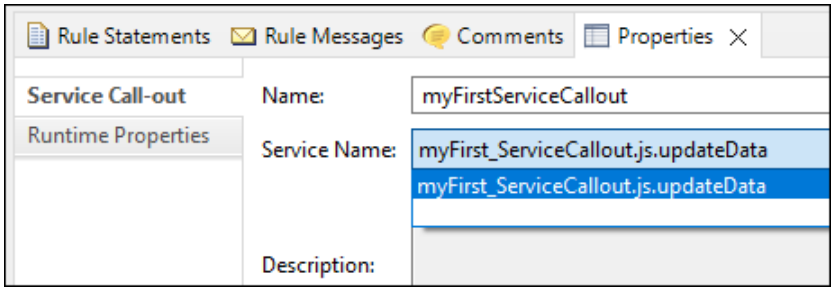
2. In the Object that appears on the canvas, name it. Here it is `myFirstService Callout`.



3. Create a connection to the Rulesheet.

Define the service callout

Select the Service Call-out on the canvas. In the **Properties** panel's **Service Call-out** section, click the dropdown menu for **Service Name**:



Service callout runtime properties are where you select the service callout function you want. A service callout definition can include multiple functions. The drop-down for the service callout name populates with the functions defined in your JS service callout file. The selected service callout is validated, and added to the JS bundle.

5

Access configuration properties

Configuration properties can be accessed by the caller of the service callout.

CODE: AccessConfigurationServiceCallout.js

```
const accessConfiguration = {
    func: 'accessConfigurationFct',
    type: 'ServiceCallout',
    description: {'en_US': 'This function shows how to access properties from the
decision service configuration.'},
    extensionType: 'SERVICE_CALLOUT',
    name: {'en_US': 'accessConfiguration'}
};

let logger;

/*
 For example, for the following configuration:

 const configuration = { logLevel: 0, "serviceCallout": { "prop1": "abc", "prop2": 12
} };

 corticonDataManager.getConfiguration() will return the object literal: { "prop1":
"abc", "prop2": 12 }
*/
function accessConfigurationFct(corticonDataManager) {
 logger = corticonDataManager.getLogger();
    logger.logDebug('Start SCO accessConfigurationFct');

    // This is how you access the list of properties
    const configurationProperties = corticonDataManager.getConfiguration();

    // Just output all properties in one string.
    let propsString = JSON.stringify(configurationProperties);
  logger.logDebug(propsString);

 // Set the message attribute to the list of properties
 const helloEntities = corticonDataManager.getEntitiesByType('Hello');
    helloEntities.forEach(hello => {
     hello['message'] = propsString;
    });
}

exports.accessConfigurationFct = accessConfigurationFct;
```

When that JavaScript runs in a browser, you get the following output:

Notice that the quotes-within-quotes were handled in the "stringify" operation by escaping the inner quotes.

6

Use functions defined in other JavaScript files

You can use a library or functions defined in supporting JavaScript files by creating a file that defines the function, and then using the function from a `require` statement.

Example of a utility function

CODE: CallToSeparateLibrary/Util.js

This example determines the appropriate greeting based on the time of day.

Table 1: Utility file to evaluate `getPeriodOfDayFromCurrentTime`

```
function getPeriodOfDayFromCurrentTime() {
 const hour = new Date().getHours();
 let postfix = 'night';
 if ( hour < 12 )
    postfix = "morning";
 else if ( hour < 18 )
   postfix = "afternoon";
 else if ( hour < 21 )
   postfix = "evening";

 return postfix;
}

function notExposedPublicly() {
  ...
}

module.exports = { getPeriodOfDayFromCurrentTime };
```

The last line exposes the function publicly.

Example of referencing the utility function

With the function defined, the `require` call brings the function into the service callout, and then exports the result, as follows:

CODE: HelloWorldServiceCallout.js

```
const utility = require('./Util');


const hello = {
    func: 'sayHelloFct',
    type: 'ServiceCallout',
    description: {'en_US': 'This function writes a greeting message into the Hello
entity'},
    extensionType: 'SERVICE_CALLOUT',
    name: {'en_US': 'hello'}
};

function sayHelloFct(corticonDataManager) {
 const postfix = utility.getPeriodOfDayFromCurrentTime();
    const entities = corticonDataManager.getEntitiesByType('Hello');
    entities.forEach(entity => {
     entity.message = 'Hello World from SCO.  I wish you a good ' + postfix;
    });
}

exports.sayHelloFct = sayHelloFct;
```

Note: Only Node module usage is supported where you can use `require` and `module.exports`. Modules written with ES6 import/export are not supported.

7

APIs for Corticon.js Service Callouts

getLogger()	Returns a handle to the logger. It provides a mechanism to log trace or errors into the logfile based on the decision service configuration
getConfiguration()	Returns a reference to the service callout properties passed in the decision service overall configuration
getAllEntities()	Returns all the entities in the working memory.
getEntityTypes()	Returns all the entity types in the working memory.
getEntitiesByType()	Takes the Entity type as a parameter, and then returns all the entities of the type in the working memory.
createEntity(<entityType>)	Takes a String defining the entity vocabulary type. Creates an entity with the defined entity type at the root and a unique identifier. Returns the newly created Entity.
createEntity(<entityType>,*{initializers}*)	Takes a String defining the entity vocabulary type. The function also accepts an initializer JSON object to set the attributes of the entity. Creates a new entity with the defined entity type at the root. Returns the newly created Entity.
removeEntity(<entityObject>)	Removes the entity from the working memory.

getOperator()	Accesses a decimal, dateTime or date operator. See Use APIs to update attributes on page 24 sections: • Decimal • Date • DateTime
addAssociation(entityObject1, entityObject2, roleName)	Adds an association of the element Entity2 with Entity1, provided such an association is allowed by the Vocabulary.
setAssociation(entityObject1, entityObject2, roleName)	Replaces any existing elements of Entity1.roleName.
removeAssociation(entityObject1, entityObject2, roleName)	Removes the element Entity2 from Entity1.roleName, provided such element is currently associated.
addEntitiesAndAssociations(entityType, payload)	Adds entities and associations as top level entities.
addAssociationsToEntity(parentEntity, rolename, payload)	Adds nested associations to existing entities in the working memory.

EXAMPLES

getLogger() example

```
let logger;

function <serviceCalloutFunctionName> (corticonDataManager) {
  logger = corticonDataManager.getLogger();
  logger.logDebug('some text') // Will only be logged when configuration = {
logLevel: 1 };
  logger.logError('We encountered error xyz') // Will be logged independently of
logLevel.
}
```

getConfiguration() example

```
const configuration = { logLevel: 0,
 "serviceCallout":
  { "prop1": "abc"
  , "prop2": 12
  }
 };
```

The SCO will be passed the object literal: `{ "prop1": "abc", "prop2": 12 }`

Note: These runtime properties are not embedded in a decision service bundle JavaScript file.

getAllEntities() example

```
function <serviceCalloutFunctionName> (corticonDataManager) {
  const entities = corticonDataManager.getAllEntities();
  //You can then use the entities returned to either modify the data/update the
attributes
}
```

getEntityTypes() example

```
function <serviceCalloutFunctionName> (corticonDataManager) {
  const entityTypes = corticonDataManager.getEntityTypes();
}
```

You can then use the entity types to get all the entities of a particular type and then update or modify the data as needed.

getEntitiesByType (<entityType>) example

Takes the Entity type as a parameter, and then returns all the entities of the type in the working memory. For example:

```
function <serviceCalloutFunctionName> (corticonDataManager) {
  const cargoEntities = corticonDataManager.getEntitiesByType('Cargo');
  //This will return all the entities of type Cargo in the working memory
}
```

createEntity(<entityType>) example

Another way to create an Entity is to just specify the entityType. This will create an entity with the defined entity type and a unique identifier.

```
function <serviceCalloutFunctionName> (corticonDataManager) {
 const entityAircraft = corticonDataManager.createEntity('Aircraft');
   //Creates a new entity aircraft, with a unique identifier.
   //This new entity is  also added to the root of the JSON.
   }
```

You can then set the attributes of this new entity:

```
function <serviceCalloutFunctionName> (corticonDataManager) {
 const entityAircraft = corticonDataManager.createEntity('Aircraft');
  //Creates a new entity aircraft, with a unique identifier.
  //This new entity is also added to the root of the JSON.
 entityAircraft['aircraftType'] = '737Boeing';
 }
```

For more information, see **Use APIs to update attributes**

createEntity(<entityType>, {initializers}) example

Create an new entity at the root. Takes a String defining the entity vocabulary type. The function creates the entity in the JSON document with a unique identifier. The function also accepts an intitalizer JSON object to set the attributes of the entity. The function returns the newly created Entity.

For example, to initialize an Entity of type `Aircraft`, and also set the attribute `aircraftType` to `'737Boeing'`

```
function <serviceCalloutFunctionName> (corticonDataManager) {
 const entityAircraft =
corticonDataManager.createEntity('Aircraft',{'aircraftType':"737Boeing"});
  // Creates a new entity aircraft, with a unique identifier and sets the attribute
aircraftType
  // This new entity is also added to the root of the JSON
}
```

removeEntity(<entityObject>) example

Removes the entity from the working memory. For example:

```
function <serviceCalloutFunctionName> (corticonDataManager) {

  const entities = corticonDataManager.getEntitiesByType('Cargo');
  let entityToRemove;
  entities.forEach(entity => {
      if(entity['container'] === 'oversize') {
        entityToRemove = entity;
      }
  });
  corticonDataManager.removeEntity(entityToRemove);
}
```

For details, see the following topics:

- Use APIs to update attributes
- Use APIs to update entities and associations
- Use APIs to have the payload update entities and associations
- Use APIs to access entities and associations

Use APIs to update attributes

To update a particular attribute of an entity, you must get the entity to update, and then access the attribute. To get the entity, use either `getAllEntities()` or `getEntitiesByType(<stringEntityType>>)`.

Note: You need to use `getOperator()` for decimal, date, and dateTime attributes.

The Corticon.js Service Callout APIs to update attributes of entities are:

- Integer
- Boolean
- String
- Decimal
- Date
- DateTime

Get the Entity to update

To update a particular attribute or association of an entity, you have to get the entity you want to update, and then access the attribute. To get the entity, you can use either `getAllEntities()` or `getEntitiesByType(<stringEntityType>)` For example, to update an Integer attribute of an Entity:

```
function updateCargoEntity(corticonDataManager) {
const entities = corticonDataManager.getEntitiesByType('Cargo');
  entities.forEach(entity => {
      if(entity['container'] === 'oversize') {
          //Update the attribute value to something else
          entity['volume'] = 50;
      }
  });
}
```

INTEGER

To update an integer attribute of an Entity:

```
function updateCargoEntity(corticonDataManager) {
const entities = corticonDataManager.getEntitiesByType('Cargo');
  entities.forEach(entity => {
      if(entity['container'] === 'oversize') {
          //Update the attribute value to something else
          entity['dispatched'] = false;
      }
  });
}
```

BOOLEAN

To update a Boolean attribute of an Entity:

```
function updateCargoEntity(corticonDataManager) {
const entities = corticonDataManager.getEntitiesByType('Cargo');
  entities.forEach(entity => {
      if(entity['container'] === 'oversize') {
          //Update the attribute value to something else
          entity['dispatched'] = false;
      }
  });
}
```

STRING

To update a String attribute of an entity, access the entity, and then update the particular attribute of the entity to the preferred value:

```
function updateCargoEntity(corticonDataManager) {
const entities = corticonDataManager.getEntitiesByType('Cargo');
  entities.forEach(entity => {
      if(entity['container'] === 'oversize') {
          //Update the attribute value to something else
          entity['manifestNumber'] = '737FA0987';
      }
  });
}
```

DECIMAL

To update Decimal Attributes of an Entity, use the `getOperator()` API to access the Decimal operator helper functions. The decimal operator is accessed as shown:

```
corticonDataManager.getOperator().decimal
```

You need to access the Decimal operator in Corticon.js, and then use its helper functions. Once you have access to the decimal operator, you can use its functions to update your attribute. For example, If you want to create a decimal value, you can use the `constructDecimal` operator:

```
corticonDataManager.getOperator().decimal.constructDecimal('<value>');
For example,
corticonDataManager.getOperator().decimal.constructDecimal('54.678');
//This will create a new decimal Object of value 54.678.
//If you want to assign this to an attribute you can simply do as below
aircraftEntity['maxCargoWeight'] =
corticonDataManager.getOperator().decimal.constructDecimal('54.678');
//The above code will create a decimal object and set the value of 'maxCargoWeight' to
 the 54.678
```

To use other operators on a decimal data type, you can access the operators available using `corticonDataManager.getOperator().decimal`.

To use the decimal `add` operator:

```
aircraftEntity['maxCargoWeight'] =
corticonDataManager.getOperator().decimal.add(aircraftEntity['maxCargoWeight'],
'100.89');
//Adds 100.89 to the existing value of maxCargoWeight.
```

To use the decimal `lessThan` operator:

```
corticonDataManager.getOperator().decimal.lessThan(aircraftEntity['maxCargoWeight'],
'100.89');
//Will compare the if the first parameter is less than the second and return a boolean.

//Returns true if the first parameter is lessThan second, else false.
```

Note: To see all the decimal helper functions, link to *"Customized data access operators" in the Corticon.js Integration Guide*.

DATE

To update Date Attributes of an Entity, you use the `getOperator()` API to get access to the Date operator helper functions. The `date` operator can be accessed as shown:

```
corticonDataManager.getOperator().date
```

For example, to update an attribute of type Date, you access the `date` operator available in Corticon, and then use the helper functions available. Once you have access to the `date` operator you can use its helper functions to update your attribute. For example, to create a new `date` value, use the `constructDate` operator:

```
corticonDataManager.getOperator().date.constructDate('<value>');
//It accepts either a string which has to be an
//ISO-8601 representation of a Date or a long value which represents the time since
epoch in milliseconds.
For example,
corticonDataManager.getOperator().date.constructDate('2021-11-11');
//This will create a new date Object with the above date .
//If you want to assign this to an attribute you can simply do as below
aircraftEntity['manufactureDate'] =
corticonDataManager.getOperator().date.constructDate('2021-11-11');
//The above code will create a date object and set the 'manufactureDate'
```

To use other operators on a Date, you can access the operators available using `corticonDataManager.getOperator().date` For example, to use the `addDays` operator:

```
aircraftEntity['manufactureDate'] =
corticonDataManager.getOperator().date.addDays('2021-11-11', 4);
//The first argument is a date dataType and the second argument is a number.
//The function will add 4 days to the date object and return the date object.
```

This example wants to add 20 days to a date:

```
aircraftEntity['manufactureDate'] =
corticonDataManager.getOperator().date.addDays(aircraftEntity['inspectionDate'], 20);
//The first argument is a date dataType and the second argument is a number.
//The function will add 20 days to the inspectionDate and set the manufactureDate value
 to inspectionDate + 20 days.
```

To use the date `lessThan` operator:

```
corticonDataManager.getOperator().date.lessThan(aircraftEntity['inspectionDate'],
aircraftEntity['manufactureDate']);
//Will compare the if the first parameter is less than the second and return a boolean.

//Returns true if the first parameter is lessThan second, else false.
```

To use the date `getMilliseconds` operator:

```
corticonDataManager.getOperator().date.getMilliseconds(aircraftEntity['inspectionDate']);
//Will return the current date as milliseconds since epoch (Jan 1, 1970)
```

Note: To see all the date helper functions, link to *"Customized data access operators" in the Corticon.js Integration Guide*.

DATETIME

To update DateTime Attributes of an Entity, you use the `getOperator()` API to get access to the DateTime operator helper functions. The `dateTime` operator can be accessed as shown:

```
corticonDataManager.getOperator().dateTime
```

For example, to update an attribute of type DateTime, you access the `dateTime` operator available in Corticon, and then use the helper functions available. Once you have access to the `dateTime` operator you can use its helper functions to update your attribute. For example, to create a new `dateTime` value, use the `constructDateTime` operator:

```
corticonDataManager.getOperator().dateTime.constructDateTime('<value>');
//It accepts either a string which has to be an
//ISO-8601 representation of a DateTime or a long value which represents the time since
 epoch in milliseconds.
For example,
corticonDataManager.getOperator().dateTime.constructDateTime('2021-11-11T19:42:08+0000');
//This will create a new datetime Object with the above date .
//If you want to assign this to an attribute you can simply do as below
aircraftEntity['manufactureDate'] =
corticonDataManager.getOperator().dateTime.constructDateTime('2021-11-11T19:42:08+0000');
//The above code will create a datetime object and set the 'manufactureDate'
```

To use other operators on a DateTime, you can access the operators available using `corticonDataManager.getOperator().dateTime` For example, to use the `addDays` operator:

```
aircraftEntity['manufactureDate'] =
corticonDataManager.getOperator().dateTime.addDays('2021-11-11T19:42:08+0000', 4);
//The first argument is a datetime dataType and the second argument is a number.
//The function will add 4 days to the dateTime object and return the dateTime object.
```

This example wants to add 20 days to a dateTime:

```
aircraftEntity['manufactureDate'] =
corticonDataManager.getOperator().dateTime.addDays(aircraftEntity['inspectionDate'],
20);
//The first argument is a datetime dataType and the second argument is a number.
//The function will add 20 days to the inspectionDate and set the manufactureDate value
 to inspectionDate + 20 days.
```

To use the dateTime `lessThan` operator:

```
corticonDataManager.getOperator().dateTime.lessThan(aircraftEntity['inspectionDate'],
aircraftEntity['manufactureDate']);
//Will compare the if the first parameter is less than the second and return a boolean.

//Returns true if the first parameter is lessThan second, else false.
```

To use the dateTime `getMilliseconds` operator:

```
corticonDataManager.getOperator().dateTime.getMilliseconds(aircraftEntity['inspectionDate']);
//Will return the current date and time as milliseconds since epoch (Jan 1, 1970)
```

Note: To see all the dateTime helper functions, link to *"Customized data access operators" in the Corticon.js Integration Guide*.

ASSOCIATIONS

See also Use APIs to update entities and associations.

Use APIs to update entities and associations

Corticon.js Service Callout APIs let you add, set, and remove associations between entities. The parameters are:

- `entityParent` - The Entity for the association
- `roleName` - The association roleName
- `entityChildren` - The Entity's association

addAssociation

`addAssociation(entityParent, entityChild, roleName)`

```
function addFlightPlanToCargo(corticonDataManager) {
 let childEntity;
 let parentEntity;
 let aircraftEntity;
 const entities = corticonDataManager.getEntitiesByType('Cargo');
    entities.forEach(entity => {
        if((entity['container'] === 'reefer') && entity['volume'] === 10 &&
entity['weight'] === 1000) {
            //Update the attribute value to something else
            parentEntity = entity;
        }
    });

 const entitiesChild = corticonDataManager.getEntitiesByType('FlightPlan');
    entitiesChild.forEach(entity => {
        if(entity['flightNumber'] === 737) {
            childEntity = entity;
        }
    });

 //Add Association
 corticonDataManager.addAssociation(parentEntity, childEntity, 'flightPlan');

}
```

setAssociation

`setAssociation(entityParent, entityChild, roleName)` - Updates an association between a parent and child entity for the rolename:

```
const updateFlightPlan = {
    func: 'updateFlightPlanToCargo',
    type: 'ServiceCallout',
    description: {'en_US': 'associates FlightPlan to Cargo'},
    extensionType: 'SERVICE_CALLOUT',
    ret: 'Void',
    name: { 'en_US': 'updateFlightPlan' }
};
```

```
function updateFlightPlanToCargo(corticonDataManager) {
    let childEntity;
    let parentEntity;
    const entities = corticonDataManager.getEntitiesByType('Cargo');
    entities.forEach(entity => {
        if((entity['container'] === 'reefer') && entity['volume'] === 10 &&
entity['weight'] === 1000) {
            //Get the parent Entity
            parentEntity = entity;
        }
    });

    const entitiesChild = corticonDataManager.getEntitiesByType('FlightPlan');
    entitiesChild.forEach(entity => {
        if(entity['flightNumber'] === 737) {
            //Get the child Entity
            childEntity = entity;
        }
    });

    //Set Association
    corticonDataManager.setAssociation(parentEntity, childEntity, 'flightPlan');

}
```

removeAssociation

`removeAssociation(entityParent, entityChild, roleName)` - Deletes an association between a parent and child entity for the rolename:

```
const removeFlightPlan = {
    builtin: false,
    func: 'removeFlightPlanImpl',
    type: 'ServiceCallout',
    description: {'en_US': 'removes FlightPlan associated with Cargo'},
    extensionType: 'SERVICE_CALLOUT',
    ret: 'Void',
    name: { 'en_US': 'removeFlightPlan' }
};
```

```
function removeFlightPlanImpl(corticonDataManager) {
    let childEntity;
    let parentEntity;
    const entities = corticonDataManager.getEntitiesByType('Cargo');
    entities.forEach(entity => {
        if((entity['container'] === 'reefer') && entity['volume'] === 10 &&
entity['weight'] === 1000) {
            //Get the parent Entity
            parentEntity = entity;
        }
    });

    const entitiesChild = corticonDataManager.getEntitiesByType('FlightPlan');
    entitiesChild.forEach(entity => {
        if(entity['flightNumber'] === 737) {
            //Get the child Entity
            childEntity = entity;
        }
    });

    //Remove Association
    corticonDataManager.removeAssociation(parentEntity, childEntity, 'flightPlan');

}
```

Note: Example of adding an association

Create an entity and associate it with another entity, as shown:

Code to Create Association: UpdateProductServiceCallout.js

UpdateProductServiceCallout.js

```
/*
    This function demonstrates how to create an entity (a provider) and how to associate it to the product
    on order, effectively creating the association.
*/
function updateProductWithProviders (productToBuy, productFromStore, corticonDataManager) {
    const allProviders = productFromStore["providers"];
    for ( let i=0; i<allProviders.length; i++ ) {
        const oneProvider = allProviders[i];
        productToBuy['_debug'] += " - Provider: " + oneProvider["name"];
        const providerEntity = corticonDataManager.createEntity('Provider', oneProvider);
        logger.logDebug('one Provider entity created for '+ oneProvider["name"]);
        corticonDataManager.addAssociation(productToBuy, providerEntity, 'provider');
        logger.logDebug('one Provider association added for ' + productToBuy["sku"]);
    }
}
```

When run in a browser deployment, the payload has the following output:

```
[
    {
        "quantityRequested": 2,
        "sku": "A-123"
    },
    {
        "quantityRequested": 15,
        "sku": "B-123"
    },
    {
        "quantityRequested": 15,
        "sku": "XXXXX"
    }
]
```

```
    },
    {
      "quantityRequested": 15,
      "sku": "B-123",
      "quantityAvailable": 58,
      "price": "79.39",
      "expirationDate": "2025-12-01T00:00:00.000Z",
      "_debug": "Ran at 2023-03-07T17:04:06.909Z - Provider: Provider3",
      "provider": [
        {
          "name": "Provider3",
          "email": "abc@prov3.com",
          "availableTillDate": "2024-02-14T00:00:00.000Z",
          "deliveryCost": "11.7"
        }
      ],
      "totalCost": "1202.55"
    },
    {
      "quantityRequested": 15,
      "sku": "XXXXX",
      "quantityAvailable": 0,
      "_debug": "Requested product not found for sku: XXXXX",
      "totalCost": "0"
    }
  ],
  "corticon": {
    "messages": {
      "message": []
    },
    "timestamp": "2023-03-07T17:04:06.919Z",
    "status": "success"
  }
```

Here is an example with two providers:

You can see that the total cost of the first item, A-123, is ((2*11.22)+15.33+21.10)=58.87, the sum of both provider deliveries.

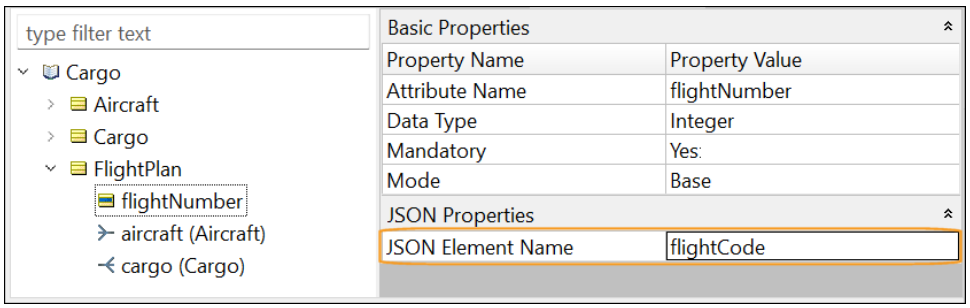
addEntitiesAndAssociations

```
addEntitiesAndAssociations(entityType,payload)
```

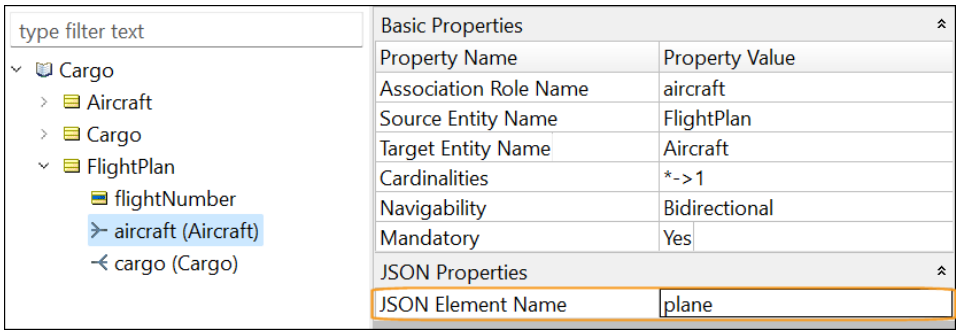
To enrich the data and the working memory, you can add entities and associations using this API. The entities will be added as top level entities. This api takes two parameters, the `entityType` and the `payload`. The `entityType` is the type of entity (defined in the vocabulary) at the root of the payload. The payload itself can either be a **JSON Object** or a **JSONArray**. If passing in a **JSONObject**, the `entityType` should match the `entityType` at the root of the **JSONObject**.

```
Sample using Cargo
Say you want to add another Cargo to the payload for your rules to execute. You can
create
a service callout and append that cargo to the working memory.
For example:
{
        "volume": 10,
        "container": null,
        "flightPlan": {
            "aircraft": {
                "aircraftType": "Airbus",
                "maxCargoWeight": null,
                "tailNumber": null,
                "maxCargoVolume": "30.000000"
            },
            "flightNumber": 380
        },
        "weight": 1000
}
The above JSON payload has a Cargo at root and its associations nested in the JSON.
You can use the API to add this Cargo to the working memory.
//Sample Code for using the API as below
//Metadata for the Service Callout
const asyncAddCargoEntity = {
    func: 'asyncAddCargoEntityFct',
    type: 'ServiceCallout',
    description: {'en_US': 'This function is used to add some data to the working
memory.'},
    extensionType: 'SERVICE_CALLOUT',
    name: {'en_US': 'asyncAddCargoEntity'}
};
//Service Callout Function
async function asyncAddCargoEntityFct(corticonDataManager) {
 const logger = corticonDataManager.getLogger();
 //Define the payload i.e your JSON Object
 const payload = {
        "volume": 10,
        "container": null,
        "flightPlan": {
            "aircraft": {
                "aircraftType": "Airbus",
                "maxCargoWeight": null,
                "tailNumber": null,
                "maxCargoVolume": "30.000000"
            },
            "flightNumber": 380
        },
        "weight": 1000
    };
try {
 //Api to add the Cargo to the root. Takes the entity Type and the payload.
 const returnCode = corticonDataManager.addEntitiesAndAssociations('Cargo', payload);
  if(returnCode === success)
   logger.logDebug(`*** Cargo has been Added`);
 }
 catch ( e ) {
  logger.logError(`*** Error adding entity: ${e}`);
 }
}
exports.asyncAddCargoEntityFct = asyncAddCargoEntityFct;
In the above example the API will add a Cargo at root and
also add the nested association FlightPlan and the association Aircraft to the working
 memory.
```

Note: Check your vocabulary before you pass in the payloads, or use the role name. Make sure the payload has the JSON overrides specified in each **JSON Element Name**, as illustrated for an attribute where the `flightNumber` is the full `flightCode` (carrier+flight number):



Similarly, an association might have a preferred **JSON Element name** as illustrated where the `aircraft` association is mapped as `plane`:



An Entity's **JSON Path** has no use in this context.

Multiple Cargos

You can also add multiple Cargos to the working memory. In that case, you can pass in a `JSONArray` of Cargo instead of a `JSONObject`.

```
To pass a JSONArray
//Metadata for the Service Callout
const asyncAddCargoEntity = {
    func: 'asyncAddCargoEntityFct',
    type: 'ServiceCallout',
    description: {'en_US': 'This function is used to add some data to the working
memory.'},
    extensionType: 'SERVICE_CALLOUT',
    name: {'en_US': 'asyncAddCargoEntity'}
};
//Service Callout Function
async function asyncAddCargoEntityFct(corticonDataManager) {
 const logger = corticonDataManager.getLogger();
 const payload = [
    {
        "volume": 10,
        "container": null,
        "flightPlan": {
            "aircraft": {
                "aircraftType": "Airbus",
                "maxCargoWeight": null,
                "tailNumber": null,
                "maxCargoVolume": "30.000000"
            },
            "flightNumber": 380
        },
        "weight": 1000
    },
    {
        "volume": 40,
        "container": null,
        "weight": 1000,
  "flightPlan": {
            "aircraft": {
                "aircraftType": "Boeing",
                "maxCargoWeight": null,
                "tailNumber": null,
                "maxCargoVolume": "50.000000"
            },
            "flightNumber": 737
        }
    },
    {
        "volume": 20,
        "container": null,
        "weight": 30000
    },
    {
        "volume": 10,
        "container": null,
        "needsRefrigeration": true,
        "weight": 1000
    }
];
try {
 //API to add Cargo's to the working memory
 //In the above example, 4 Cargo entities will be added to working memory
 const returnCode = corticonDataManager.addEntitiesAndAssociations('Cargo', payload);
  if(returnCode === success)
   logger.logDebug(`*** Cargos have been Added`);
 }
 catch ( e ) {
  logger.logError(`*** Error adding entity: ${e}`);
 }
}
//Exports
exports.asyncAddCargoEntityFct = asyncAddCargoEntityFct;
```

addAssociationsToEntity

```
addAssociationsToEntity(parentEntity,rolename,payload)
```

To enrich the data and the working memory, you can add nested associations to existing entities using this API. The entity to which you want to add associations should be in the working memory. The entity can be at any level in the input tree. This api takes three parameters, the `parentEntity` the `rolename` and the `payload`. The `parentEntity` is the entity in the working memory to which you want to add the association. The `rolename` is the association you want to add (defined in the vocabulary). The payload itself can either be a **JSON Object** or a **JSONArray**.

```
To add a flightPlan association to Cargo, the JSON payload might look like this:

{
            "aircraft": {
                "aircraftType": "Airbus",
                "maxCargoWeight": null,
                "tailNumber": null,
                "maxCargoVolume": "30.000000"
            },
            "flightNumber": 380
}

The above JSON has FlightPlan and nested Aircraft.
You can add this association to the existing Cargo using the sample below -
//Metadata for the Service Callout
const asyncAddFlightPlanAssociationEntity = {
    func: 'asyncAddFlightPlanEntityFct',
    type: 'ServiceCallout',
    description: {'en_US': 'This function is used to add some flight plan to the cargo
 entity.'},
    extensionType: 'SERVICE_CALLOUT',
    name: {'en_US': 'asyncAddFlightPlanAssociationEntity'}
};
//Service Callout Function
async function asyncAddFlightPlanEntityFct(corticonDataManager) {
 const logger = corticonDataManager.getLogger();
 const payload = {
            "aircraft": {
                "aircraftType": "Airbus",
                "maxCargoWeight": null,
                "tailNumber": null,
                "maxCargoVolume": "30.000000"
            },
            "flightNumber": 380
        };
try {
  //This API gets the Cargo entities in working memory
  const entities = corticonDataManager.getEntitiesByType('Cargo');
  /* Logic to determine which Cargo you want to associate this
     Flight Plan with.
     In this example we are using volume and if the volume of the Cargo
     equals 10 then we are going to attach the flight plan to the Cargo
     The logic determines the parent entity and that parent entity in passed
     to the API below
  */
  let entityParent;
  for (let entity of entities) {
   if(entity['volume'] === 10) {
    entityParent = entity; //Determines the parent Entity to be used.
    break;
   }
  }
  /* This call then passes in the parent entity, the role name FlightPlan as defined
  in the vocabulary and the above payload. If there are no errors, success is returned
 from the API
  and the association will be attached to the Cargo */
 const success = corticonDataManager.addAssociationsToEntity(entityParent, "FlightPlan",
 payload);
  logger.logDebug(`*** Data Added`);
 }
 catch ( e ) {
  logger.logError(`*** Error adding entity: ${e}`);
 }
}
//Exports
exports.asyncAddFlightPlanEntityFct = asyncAddFlightPlanEntityFct;
```

Multiple nested associations

You can also add multiple nested associations to an entity. If you want to add multiple associations to an entity, it must be supported in the vocabulary. To add multiple associations you must pass in a **JSONArray**.

```
const asyncAddFlightPlanAssociationEntity = {
    func: 'asyncAddFlightPlanEntityFct',
    type: 'ServiceCallout',
    description: {'en_US': 'This function is used to add some flight plan to the cargo
 entity.'},
    extensionType: 'SERVICE_CALLOUT',
    name: {'en_US': 'asyncAddFlightPlanAssociationEntity'}
};
async function asyncAddFlightPlanEntityFct(corticonDataManager) {
 const logger = corticonDataManager.getLogger();
 const payload = [{
  "aircraft": {
   "aircraftType": "Airbus",
   "maxCargoWeight": null,
   "tailNumber": null,
   "maxCargoVolume": "30.000000"
  },
  "flightNumber": 380
 },
 {
  "aircraft": {
   "aircraftType": "Boeing",
   "maxCargoWeight": null,
   "tailNumber": null,
   "maxCargoVolume": "30.000000"
  },
  "flightNumber": 737
 },
 {
  "aircraft": {
   "aircraftType": "Airbus",
   "maxCargoWeight": null,
   "tailNumber": null,
   "maxCargoVolume": "30.000000"
  },
  "flightNumber": 350
 },
 {
  "aircraft": {
   "aircraftType": "Boeing",
   "maxCargoWeight": null,
   "tailNumber": null,
   "maxCargoVolume": "30.000000"
  },
  "flightNumber": 747
 }
 ];
try {
  //This API gets the Cargo entities in working memory
  const entities = corticonDataManager.getEntitiesByType('Cargo');
  /* Logic to determine which Cargo you want to associate this
     Flight Plan with.
     In this example we are using volume and if the volume of the Cargo
     equals 10 then we are going to attach the flight plan to the Cargo
     The logic determines the parent entity and that parent entity in passed
     to the API below
  */
  let entityParent;
  for (let entity of entities) {
   if(entity['volume'] === 10) {
    entityParent = entity; //Determines the parent Entity to be used.
    break;
   }
  }
  /* This call then passes in the parent entity, the role name  FlightPlan as defined

  in the vocabulary and the above payload. If there are no errors, success is returned
 from the API
```

```
  and the association will be attached to the Cargo. It will add 4 Flight Plan to the
Cargo  */
  const success = corticonDataManager.addAssociationsToEntity(entityParent, "FlightPlan",
 payload);
  logger.logDebug(`*** Data Added`);
 }
 catch ( e ) {
  logger.logError(`*** Error adding entity: ${e}`);
 }
}
exports.asyncAddFlightPlanEntityFct = asyncAddFlightPlanEntityFct;
```

Use APIs to have the payload update entities and associations

Corticon.js Service Callout APIs let you use payloads to add, set, and remove associations between entities. The parameters are:

- `entityParent` - The Entity for the association
- `roleName` - The association roleName
- `entityChildren` - The Entity's association
- `payload` - The payload as a JSONObject or JSONArray

add Entities and Associations from a payload

`addEntitiesAndAssociations(entityType, payload)`

To enrich the data and the working memory you can add entities and associations using this API. The entities will be added as top level entities. This api takes two parameters, the <entityType> and the <payload>. The entityType is the type of entity (defined in the vocabulary) at the root of the payload. The payload itself can either be a **JSON Object** or a **JSONArray**. If passing in a **JSONObject**, the entityType should match the entityType at the root of the JSONObject.

```
Use Cargo as the Sample
Let's say you want to add another Cargo to the payload for your rules to execute. You
can create
a service callout and append that cargo to the working memory.
For example:.
{
        "volume": 10,
        "container": null,
        "flightPlan": {
            "aircraft": {
                "aircraftType": "Airbus",
                "maxCargoWeight": null,
                "tailNumber": null,
                "maxCargoVolume": "30.000000"
            },
            "flightNumber": 380
        },
        "weight": 1000
}
The above JSON payload has a Cargo at root and its associations nested in the JSON.
You can use the API to add this Cargo to the working memory.
//Sample Code for using the API as below
//Metadata for the Service Callout
const asyncAddCargoEntity = {
    func: 'asyncAddCargoEntityFct',
    type: 'ServiceCallout',
    description: {'en_US': 'This function is used to add some data to the working
memory.'},
    extensionType: 'SERVICE_CALLOUT',
    name: {'en_US': 'asyncAddCargoEntity'}
};
//Service Callout Function
async function asyncAddCargoEntityFct(corticonDataManager) {
 const logger = corticonDataManager.getLogger();
 //Define the payload i.e your JSON Object
 const payload = {
        "volume": 10,
        "container": null,
        "flightPlan": {
            "aircraft": {
                "aircraftType": "Airbus",
                "maxCargoWeight": null,
                "tailNumber": null,
                "maxCargoVolume": "30.000000"
            },
            "flightNumber": 380
        },
        "weight": 1000
    };
try {
 //Api to add the Cargo to the root. Takes the entity Type and the payload.
 const returnCode = corticonDataManager.addEntitiesAndAssociations('Cargo', payload);
  if(returnCode === success)
   logger.logDebug(`*** Cargo has been Added`);
 }
 catch ( e ) {
  logger.logError(`*** Error adding entity: ${e}`);
 }
}
exports.asyncAddCargoEntityFct = asyncAddCargoEntityFct;
In the above example the API will add a Cargo at root and
also add the nested association FlightPlan and the association Aircraft to the working
 memory.
```

You can also add multiple Cargo's to the working memory. In that case you can pass in a JSON Array of Cargo's instead of a JSONObject:

```
eg. To pass JSONArray
//Metadata for the Service Callout
const asyncAddCargoEntity = {
    func: 'asyncAddCargoEntityFct',
    type: 'ServiceCallout',
    description: {'en_US': 'This function is used to add some data to the working
memory.'},
    extensionType: 'SERVICE_CALLOUT',
    name: {'en_US': 'asyncAddCargoEntity'}
};
//Service Callout Function
async function asyncAddCargoEntityFct(corticonDataManager) {
 const logger = corticonDataManager.getLogger();
 const payload = [
    {
        "volume": 10,
        "container": null,
        "flightPlan": {
            "aircraft": {
                "aircraftType": "Airbus",
                "maxCargoWeight": null,
                "tailNumber": null,
                "maxCargoVolume": "30.000000"
            },
            "flightNumber": 380
        },
        "weight": 1000
    },
    {
        "volume": 40,
        "container": null,
        "weight": 1000,
  "flightPlan": {
            "aircraft": {
                "aircraftType": "Boeing",
                "maxCargoWeight": null,
                "tailNumber": null,
                "maxCargoVolume": "50.000000"
            },
            "flightNumber": 737
        }
    },
    {
        "volume": 20,
        "container": null,
        "weight": 30000
    },
    {
        "volume": 10,
        "container": null,
        "needsRefrigeration": true,
        "weight": 1000
    }
];
try {
 //API to add Cargo's to the working memory
 //In the above example, 4 Cargo entities will be added to working memory
 const returnCode = corticonDataManager.addEntitiesAndAssociations('Cargo', payload);
  if(returnCode === success)
   logger.logDebug(`*** Cargos have been Added`);
 }
 catch ( e ) {
  logger.logError(`*** Error adding entity: ${e}`);
 }
}
//Exports
exports.asyncAddCargoEntityFct = asyncAddCargoEntityFct;
```

Note: Make sure the payload has the JSON Overrides value . Refer to the JSON Overrides sections here.

addAssociationsToEntity

`addAssociationsToEntity(parentEntity, rolename, payload)` - To enrich the data and the working memory you can add nested associations to existing entities using this API. The entity to which you want to add associations should be in the working memory . The entity can be at any level in the input tree. This api takes three parameters, the <parentEntity> the <rolename> and the <payload>. The parentEntity is the entity in the working memory to which you want to add the association. The rolename is the association you want to add (defined in the vocabulary). The payload itself can either be a **JSON Object** or a **JSONArray**.
:

```
Let say we want to add a flightPlan association to Cargo.
The JSON payload looks like this -
{
            "aircraft": {
                "aircraftType": "Airbus",
                "maxCargoWeight": null,
                "tailNumber": null,
                "maxCargoVolume": "30.000000"
            },
            "flightNumber": 380
}
The above JSON has FlightPlan and nested Aircraft.
You can add this asscoaition to the existing Cargo using the sample below -
//Metadata for the Service Callout
const asyncAddFlightPlanAssociationEntity = {
    func: 'asyncAddFlightPlanEntityFct',
    type: 'ServiceCallout',
    description: {'en_US': 'This function is used to add some flight plan to the cargo
 entity.'},
    extensionType: 'SERVICE_CALLOUT',
    name: {'en_US': 'asyncAddFlightPlanAssociationEntity'}
};
//Service Callout Function
async function asyncAddFlightPlanEntityFct(corticonDataManager) {
 const logger = corticonDataManager.getLogger();
 const payload = {
            "aircraft": {
                "aircraftType": "Airbus",
                "maxCargoWeight": null,
                "tailNumber": null,
                "maxCargoVolume": "30.000000"
            },
            "flightNumber": 380
        };
try {
  //This API gets the Cargo entities in working memory
  const entities = corticonDataManager.getEntitiesByType('Cargo');
  /* Logic to determine which Cargo you want to associate this
     Flight Plan with.
     In this example we are using volume and if the volume of the Cargo
     equals 10 then we are going to attach the flight plan to the Cargo
     The logic determines the parent entity and that parent entity in passed
     to the API below
  */
  let entityParent;
  for (let entity of entities) {
   if(entity['volume'] === 10) {
    entityParent = entity; //Determines the parent Entity to be used.
    break;
   }
  }
  /* This call then passes in the parent entity, the role name < FlightPlan> as defined

  in the vocabulary and the above payload. If there are no errors, success is returned
 from the API
  and the association will be attached to the Cargo */
 const success = corticonDataManager.addAssociationsToEntity(entityParent, "FlightPlan",
 payload);
  logger.logDebug(`*** Data Added`);
 }
 catch ( e ) {
  logger.logError(`*** Error adding entity: ${e}`);
 }
}
//Exports
exports.asyncAddFlightPlanEntityFct = asyncAddFlightPlanEntityFct;
```

You can also add multiple nested associations to an entity. If you want to add multiple associations to an entity, it must be supported in the vocabulary. To add multiple associations one must pass in a JSONArray:

```
const asyncAddFlightPlanAssociationEntity = {
    func: 'asyncAddFlightPlanEntityFct',
    type: 'ServiceCallout',
    description: {'en_US': 'This function is used to add some flight plan to the cargo
 entity.'},
    extensionType: 'SERVICE_CALLOUT',
    name: {'en_US': 'asyncAddFlightPlanAssociationEntity'}
};
async function asyncAddFlightPlanEntityFct(corticonDataManager) {
 const logger = corticonDataManager.getLogger();
 const payload = [{
  "aircraft": {
   "aircraftType": "Airbus",
   "maxCargoWeight": null,
   "tailNumber": null,
   "maxCargoVolume": "30.000000"
  },
  "flightNumber": 380
 },
 {
  "aircraft": {
   "aircraftType": "Boeing",
   "maxCargoWeight": null,
   "tailNumber": null,
   "maxCargoVolume": "30.000000"
  },
  "flightNumber": 737
 },
 {
  "aircraft": {
   "aircraftType": "Airbus",
   "maxCargoWeight": null,
   "tailNumber": null,
   "maxCargoVolume": "30.000000"
  },
  "flightNumber": 350
 },
 {
  "aircraft": {
   "aircraftType": "Boeing",
   "maxCargoWeight": null,
   "tailNumber": null,
   "maxCargoVolume": "30.000000"
  },
  "flightNumber": 747
 }
 ];
try {
  //This API gets the Cargo entities in working memory
  const entities = corticonDataManager.getEntitiesByType('Cargo');
  /* Logic to determine which Cargo you want to associate this
     Flight Plan with.
     In this example we are using volume and if the volume of the Cargo
     equals 10 then we are going to attach the flight plan to the Cargo
     The logic determines the parent entity and that parent entity in passed
     to the API below
  */
  let entityParent;
  for (let entity of entities) {
   if(entity['volume'] === 10) {
    entityParent = entity; //Determines the parent Entity to be used.
    break;
   }
  }
  /* This call then passes in the parent entity, the role name < FlightPlan> as defined

  in the vocabulary and the above payload. If there are no errors, success is returned
 from the API
```

```
  and the association will be attached to the Cargo. It will add 4 Flight Plan to the
Cargo  */
  const success = corticonDataManager.addAssociationsToEntity(entityParent, "FlightPlan",
 payload);
  logger.logDebug(`*** Data Added`);
 }
 catch ( e ) {
  logger.logError(`*** Error adding entity: ${e}`);
 }
}
exports.asyncAddFlightPlanEntityFct = asyncAddFlightPlanEntityFct;
```

Use APIs to access entities and associations

`getAssociationsForEntity(parentEntity, rolename)`

To access the data in the working memory you can use this API. The entity can be at any level in the input tree. This api takes two parameters, the `parentEntity` and the `rolename`. The `parentEntity` is the entity in the working memory that you want to query. The `rolename` is the association you want to fetch (as defined in the vocabulary). The API returns a **JSON Array**. It will bring all the associations for the rolename and also nested associations.

```
Let say we want to get a flightPlan association to a particular Cargo.
//Metadata for the Service Callout
const getFlightPlanAssociationCargo = {
    func: 'getFlightPlanAssociationCargoFct',
    type: 'ServiceCallout',
    description: {'en_US': 'This function is used to get flight plans for the cargo'},

    extensionType: 'SERVICE_CALLOUT',
    name: {'en_US': 'getFlightPlanAssociationCargo'}
};
//Service Callout Function
async function getFlightPlanAssociationCargoFct(corticonDataManager) {
 const logger = corticonDataManager.getLogger();
try {
  //This API gets the Cargo entities in working memory
  const entities = corticonDataManager.getEntitiesByType('Cargo');
  /* Logic to determine the Cargo, the parent entity which will be used.
     In this example we are using volume and if the volume of the Cargo
     equals 10 then we are going to get the flight plan
     The logic determines the parent entity and that parent entity in passed
     to the API below
  */
  let entityParent;
  for (let entity of entities) {
   if(entity['volume'] === 10) {
    entityParent = entity; //Determines the parent Entity to be used.
    break;
   }
  }
  /* This call then passes in the parent entity, the role name < FlightPlan> as defined

  in the vocabulary. If there are no errors, json is returned from the API
  In this case it will have all the flight Plans which are associated with the Cargo.
  If the FlightPlan has another association attached to it it will be returned as well.
 */
  const jsonReturned = corticonDataManager.getAssociationsForEntity(entityParent,
"FlightPlan");
  logger.logDebug(`*** Data retrieved`+JSON.stringify(jsonReturned));
  //Write the JSON to a file
  const fs = require('fs');
  fs.writeFileSync('Output.json', JSON.stringify(jsonReturned, null, 2));
 }
 catch ( e ) {
  logger.logError(`*** Error retrieving data : ${e}`);
 }
}
//Exports
exports.getFlightPlanAssociationCargoFct = getFlightPlanAssociationCargoFct;
```