Configuring an API Callout

Last Modified on 27/01/2021 3:07 pm AEDT

To configure an individual API Callout endpoint, the following information is required:

- The HTTP Method
- The endpoint address
- Information about the data being sent (optional)

HTTP Method

The HTTP method is a part of how HTTP requests work. It will typically be one of “GET”, “POST” “PUT”, or “DELETE” and should be identified in the documentation for the API being called.

If the method is not specified by the documentation then, as a rule of thumb:

- Assume that the method is `GET` if there is no data (XML or JSON) being sent as part of the request
- Assume that the method is `POST` if there is data (XML or JSON) being sent as part of the request

Endpoint address

The endpoint address is the web address of the individual third-party API being called. For ReadiNow API Callouts, a base address can be configured as part of the API Callout Library, such as `http://thirdparty.com/.` Individual endpoints then specify their address relative to this base address (enter this information into the Relative URL field). For example: `/someapi`

Before calling the API, the base address is prepended to the relative address (e.g. `http://thirdparty.com/someapi`).

It is common for web APIs to receive certain information as part of their address. This information may take the form of part of the path, or as part of the parameter. You may include static parameters as part of the address by directly typing those values.

For example: `/someapi?settings=somevalue`. Information that is provided in this way must be correctly encoded as per normal URL encoding rules so that it will form a valid URL.

You may also include values that are provided by the calling workflow by embedding them as parameters. For example, Relative URL may be: `/api/department/{[Department Code]}/staff.`

If an API Callout activity is added to a workflow, then the above URL will automatically case a "Department Code" string input to be provided, and information received from the workflow activity will be included into the URL. The system will automatically perform the relevant URL encoding in this case. See below for more details.

Request Body

If the HTTP Method is one of `POST`, `PUT`, or `DELETE`, then an additional "Request Template" input is shown. This is where XML, or JSON, or other data may be defined that will be sent to the server. The third-party API documentation will describe the format of data that it is expecting.

Text Templates

As described above, a web request consists of various components, including:

- A web address - that is, a URL
- A request body
- HTTP Headers (additional metadata)

For each of these, the data is text. To allow complete flexibility over what text is offered, ReadiNow APIs use a text template system, which is very similar to ReadiNow document templates.

You enter what the request message needs to look like, and then embed additional macros that can be used to:

- Insert data values
- Conditionally include/exclude certain parts of the message
- Generate repeated content in the message

The macros to do this are very similar to those for document generation.

Simple example

A JSON request URL and body might look like this:

URL : http://hostname.com/api/department/sales/staff

```
{
   "firstName":"Jack",
   "lastName":"Smith",
   "status":"NewHire",
   "hoursPerWeek":38,
   "startDate":"2017-01-09"
}
```

The corresponding templates for the request URL and body could look like this:

URL: http://hostname.com/api/department/{[Department Code]}/staff

```
{
  "firstName": { [Firstname] },
  "lastName": { [Lastname] },
  "status": "NewHire",
  "hoursPerWeek": { input(int, [Hoursperweek]) },
  "startDate": { dateadd(day, 14, getdate()) }
}
```

In the above example, the API Callout would detect 4 inputs that can be passed in from a workflow:

- Department Code (Text)
- First Name (Text)

- Last Name (Text)
- Hours per week (Integer)

It would also calculate the date 14 days into the future and specify that as the start date. The system will automatically perform URL and JSON string encoding as necessary.

Encoding inputs

As illustrated above, any unrecognised identifier is treated as an input. By default, these are treated as string inputs but by using the "input" function, it is possible to also specify the type of input. For example, `input(currency, [Balance])` will expose an input called "Balance" of type currency.

Records can also be passed as inputs by using the object name as the input type. For example, `input([Person], [Owner])` will create an input called "Owner" that receives a Person record. This can then be used to access individual fields in the same manner as other calculations.

Note that the type of an input only needs to be specified the first time that an input is used in the document. For example:

```
{
  "ownerName": { input([Person], [Owner].[Firstname])+' '+[Owner].[Lastname] },
  "department": { [Owner].[Department] }
}
```

Repeated Content

Repeated content can be written in API requests in the same way that it can be done for document generation. For example:

```
{
  "ownerName": {
    input([
      Person
    ],
    [
      Owner
    ].[
      Firstname
    ])+' '+[
      Owner
    ].[
      Lastname
    ])
  },
  "department": {
    [
      Owner
    ].[
      Department
    ]
  }
}
```

The `{sep}` instruction can also be used to create separation lists:

```
{with all(Student)} {[First Name]} {sep},{end}
```

Which yields the result: "Adam, Belinda, Charles, Diana"

For JSON data, the API requests can also use the special `array` keyword as follows (it automatically handles the square brackets and commas):

```

{
  "students": {
    arrayall(Student)
  }{
    "firstName": {
      [
        Firstname
      ]
    },
    "lastName": {
      [
        Lastname
      ]
    },
    "subjects": {
      array[
        Subjects
      ]
    }{
      [
        Name
      ]
    }{
      end
    }
  }{
    end
  }
}
```

Which yields the following result:

```
{
  "students": [
    {
      "firstName": "Alex",
      "lastName": "Zeo",
      "subjects": [
        "Accounting",
        "Economics"
      ]
    },
    {
      "firstName": "Belinda",
      "lastName": "Yang",
      "subjects": [
        "Linguistics",
        "Maths"
      ]
    }
  ]
}
```

The example above generates an array of student as objects. For each student it also follows the "subjects" relationship and creates an array of subject names as strings for that student.

Output encoding

When building XML / JSON templates using calculation, the system will automatically attempt to encode values as required; this is achieved by setting the expected result type to URL, XML or JSON respectively. When building URLs, the system will also attempt encode values as required.

Calculations will implicitly convert strings, and other values, to these encodings by applying the standard escaping rules for each format.

Additionally, for JSON strings and dates, the encoding will add enclosing double-quote marks. This is necessary so that JSON encoding can appropriately encode nulls.

For example:

- Before: `I have one" double quote`
- After: `"I have one\" double quote"`

Note the addition of the surrounding quote marks.

When dates, times, and date-times are converted (explicitly or implicitly) to URL, XML or JSON encodings, they will automatically assume an ISO8601 format:

- **Dates**: 2012-07-31
- **Times**: 23:30:00
- **Date-times**: 2012-07-31T23:30:00

In all the above cases, automatic encoding can be bypassed by using the explicit `donotencode` encoding. For example, `convert(donotencode, getdate())`

Escape sequences for braces

In the case of JSON, curly braces define both the macros, and the object syntax. Curly braces have been used as they are a common delimiter for documentation in how APIs describe URLs, and they are also used in Word for document generation macros.

The API callout will generally be able to decide the meaning of the curly brace from context, but it can be explicitly controlled using the following rules:

- If the first non-whitespace character after an opening curly brace is a double quote, or if the enclosed sequence contains a colon, then the text is assumed to be a JSON object (otherwise it is assumed to be a template macro)
- The syntax `${..}` can be used to explicitly enforce the template syntax.
- The syntax `${{` can be used to mark a literal '{' within plain text
- The syntax `$}}` can be used to mark a literal '}' within a macro string
- The syntax `$$` can be used to mark a literal '$' in either context

The goal of these rules is that explicit escape sequences are seldom used but are available in case some obscure combination of text is required.

Responses

An API Callout makes a web request to some remote third-party service. In most cases the service will reply with some form of response, either to indicate the success/failure of the call, or to return some form of requested data (REST APIs typically return their results in either JSON or XML format).

API Callouts are accessed via a workflow activity and this activity makes response information available through output parameters.

Activity Output Parameters

API Callout activities return the following output parameters, which can then be used in calculations for other activities:

Parameter Name	Type	Description
Response	Dynamic type	A record that represents the root JSON or XML item that can be used to directly access properties and arrays from the parsed response. The platform uses the response template to work out what type of field should be made available in the response.
HTTP Status Code	Integer	The HTTP Response code, such as 200 for OK. See list of typical response codes.

HTTP Response Body	String	The raw unprocessed text response (e.g. raw JSON or XML) from the server.
HTTP Request URL	String	The web address that was called. This can be useful when troubleshooting, to see how parameters were encoded into the URL.
HTTP Request Body	String	The content that was sent as part of the web request. This can be useful when troubleshooting, to see how parameters were encoded into the request template.

Setting a Response Template

The system uses the `Message format` setting on the API Callout to determine what type of response to expect. Options are:

- JSON
- XML

When an API Callout endpoint is created, a "typical" expected response (the response template) must be provided.

The template provides the platform with an example of what a response will look like. The platform uses the template to discover what types of fields and relationships it should expose via the `Response` parameter.

For example, a response template may be:

```
{
  "fullName": "Whatever",
  "birthday": "2000-01-01T00:00:00",
  "shirtSize": 1,
  "address": {
    "line1": "whatever",
    "line2": "whatever",
    "suburb": "whatever"
  },
  "hobbies": [
    "a",
    "b"
  ],
  "pets": [
    {
      "name": "whatever",
      "age": 1
    }
  ]
}
```

From the above information, the platform will deduce the following:

- There is a string field called `fullName`
- There is a date-time field called `birthday`
- There is an integer field called `shirtSize`
- There is a lookup called `address`
  - The related data has string fields called `line1, line2` and `suburb`
- There is a relationship called `hobbies`
  - The related data has a string field called `Value`
- There is a relationship called `pets`
  - The related field has a string field called `name`, and an integer field called `age`

The general idea is that a response template can be filled in by copying and pasting the "example response" from the third-party service's API documentation. In practice, example documentation often contains non-standard notation that must be removed or adjusted to make the message valid. The sample response entered into the ReadiNow response template must be valid JSON or XML and match the message format.

The actual data values specified in the template are used to determine the field types and are then replaced with the actual response data when the API itself is called.

Accessing Response Information

A workflow activity calculation can access response data via the `Response` output parameter. For example, if there is an API Callout activity called "Api Callout1", then response data for the above example template can be accessed using any of the following expressions

- `[API Callout1.Response].[fullName]`
- `[API Callout1.Response].[birthday]`

- `[API Callout1.Response].[shirtSize]`
- `[API Callout1.Response].[address]`
- `[API Callout1.Response].[address].[line1]`
- `[API Callout1.Response].[address].[line2]`
- `[API Callout1.Response].[address].[suburb]`
- `[API Callout1.Response].[hobbies]`
- `[API Callout1.Response].[pets]`

This information can be used in the same way that normal fields and relationships can in calculations. For example:

`let r = [API Callout1.Response]`

`select r.[fullName] + (r.[shirtSize]*2) + count(r.[hobbies])`

Note that, as with all calculations, the square brackets are optional if the field name only contains letters/numbers, and starts with a letter.

These dynamic types cannot be assigned to workflow variables.

Accessing response arrays using the For Each activity

The response data can be used inside **For Each** activities.

For example, a **For Each** activity could be created in an example workflow with its **List** input parameter set to: `[API Callout1.Response].pets` and would then iterate over the items in the JSON response, assigning each in turn to the
`[For Each.Record]` output parameter. Activities within the **For Each** loop can then further access JSON properties on each individual entry. For example by using `[For Each.Record].[age].`

If the array is an array of strings, numbers or some other simple data type, then the information is made available via a special 'Value' property. It can be accessed, for example, by using ` [For Each.Record].[Value]`.

If the JSON template refers to an embedded object, or an array, such as this example does for `address`, `hobbies`, and `pets`, then the value acts like a record that has a dynamic type.

If it is a single embedded object (like address in this example), then it behaves like a lookup to a single related record. If it is an array, then it acts like a relationship to many related records.

Rules for Interpretation of Field Types

The platform uses the following rules when interpreting the response template to infer field types.

For JSON

- `true`, `false` (without quotes) → yes/no field
- numbers (without quotes) that do not contain decimal points → `Number` (integer) field
- numbers (without quotes) that do contain decimal points → `Decimal` field

- quoted strings where the example data contains a datetime → `DateTime` field (the date must be in one of the following formats):
  - `2014-12-31T00:00:00`
  - `2014-12-31 00:00:00`
  - `2014-12-31T00:00:00.123`
  - `2014-12-31T00:00:00.123Z`
- quoted string where the example data contains a date only → `Date` field
  - The date must be in following format: 2014-07-31
- embedded objects using { curly braces } → treated as lookups
- embedded arrays using [square brackets] → treated as relationships
- embedded arrays of simple types:
  - If an embedded array contains only strings, then the related object will have a single string field called "Value".
  - If an embedded array contains only one other type of data, as per the rules above, then the related object will have a single field called "Value" of that type.
  - If an array contains a mixture of types, then the platform will attempt to find the most general type. (numbers and decimals mixed will result in array of decimals; most things mixed with strings will result in array of strings).
- Embedded arrays of objects:
- If the sample template has multiple example records, then their contents are effectively “unioned”.Or, if there are conflicting types, then the most general type is selected
- Root level arrays
  - Return an object that contains a single relationship called values
    - e.g. for a root object array: `[ {"myfield":1}, {"myfield":2} ]`, you can access individual objects via the special 'Values' relationship.
      - e.g. For each: `[Values]`
      - Or sum individual items using `sum([Values].[myfield])`
    - e.g. for root scalar array: [1,2,3,4] can access the individual values via `[Values].[Value]`
      - e.g. `sum([Values].[Value])`
- `null` is generally ignored
  - If the example response template contains a `null`, then the user should change the example to show what type of data might be expected
  - A `null` in arrays will merge with other types in the array without affecting them
- A special `[JSON]` string field can be used on any dynamic JSON object to get its JSON representation as text

For XML

- XML attributes appear as fields
    - Same detection rules as for XML, namely:
    - true, false → yes/no field
    - Whole numbers → Number (integer) field
    - Decimal numbers → Decimal field
    - Same date and date-time formats as for JSON
- XML elements appear as relationships
- The text content of an element can be accessed via a [Text] string field.
    - E.g. `[API Callout.Response].rootElement.childElement.text`
- If an element appears once, then it behaves like a to-one lookup
- If an element appears more than once in the same parent, then it behaves like a to-many relationship. It may be convenient to use the shorthand notation to specify the second instance.
    - For example:
      In this instance, the books can be accessed via `[API Callout.Response].books.book`, then the titles accessed via `[For Each.Record].title.text`
- A special [XML] string field can be used on any dynamic XML element to get its XML representation as text

- XML attributes appear as fields
    - Same detection rules as for XML, namely:
    - true, false → yes/no field
    - Whole numbers → Number (integer) field
    - Decimal numbers → Decimal field
    - Same date and date-time formats as for JSON
- XML elements appear as relationships
- The text content of an element can be accessed via a [Text] string field.
    - E.g. `[API Callout.Response].rootElement.childElement.text`
- If an element appears once, then it behaves like a to-one lookup
- If an element appears more than once in the same parent, then it behaves like a to-many relationship. It may be convenient to use the shorthand notation to specify the second instance.
    - For example:
      In this instance, the books can be accessed via `[API Callout.Response].books.book`, then the titles accessed via `[For Each.Record].title.text`
- A special [XML] string field can be used on any dynamic XML element to get its XML representation as text

```
• XML attributes appear as fields
    ∘ Same detection rules as for XML, namely:
    ∘ true, false → yes/no field
    ∘ Whole numbers → Number (integer) field
    ∘ Decimal numbers → Decimal field
    ∘ Same date and date-time formats as for JSON
• XML elements appear as relationships
• The text content of an element can be accessed via a [Text] string field.
    ∘ E.g. [API Callout.Response].rootElement.childElement.text
• If an element appears once, then it behaves like a to-one lookup
• If an element appears more than once in the same parent, then it behaves like a to-many relati
  onship. It may be convenient to use the  shorthand notation to specify the second instance.
```

- For example:
  dads
  In this instance, the books can be accessed via `[API Callout.Response].books.book`, then the titles a ccessed via `[For Each.Record].title.text`

```
• A special [XML] string field can be used on any dynamic XML element to get its XML representat
  ion as text
```

- XML attributes appear as fields
  - Same detection rules as for XML, namely:
  - `true`, `false` → `yes` / `no` field
  - Whole numbers → `Number` (integer) field
  - Decimal numbers → `Decimal` field
  - Same date and date-time formats as for JSON
- XML elements appear as relationships
- The text content of an element can be accessed via a `[Text]` string field.
  - E.g. `[API Callout.Response].rootElement.childElement.text`
- If an element appears once, then it behaves like a to-one lookup
- If an element appears more than once in the same parent, then it behaves like a to-many relationship. It may be convenient to use the shorthand notation to specify the second instance.

  - For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<books>
   <book id="123">
      <title>Harry Potter</title>
   </book>
   <book />
   <!-- second element to indicate repetition -->
</books>
```

In this instance, the books can be accessed via `[API Callout.Response].books.book`, then the titles accessed via `[For Each.Record].title.text`

- A special `[XML]` string field can be used on any dynamic XML element to get its XML representation as text