5: Setup Hooks

Last Modified on 02/01/2021 4:20 pm EST

1
Info

>

2
Properties

>

3
Authentication

>

4
Config & Parameters

>

5
Hooks

>

6
Events

>

7
Resources

Hooks enable you to execute custom JavaScript before an API request (pre-request hook) and after the API provider sends a response (post-response hook). You can use two types of hooks when you create a connector: global hooks and resource hooks. Global hooks happen on every request or response, while resource hooks happen only on requests to and responses from specific endpoints.

Use hooks to manipulate any part of a request or response or to operate on a configuration. You might need a hook due to the authentication expected by the endpoint. You might need to send a value to an endpoint, but it requires a different data type than what SAP Cloud Platform Open Connectors supports. You might also need to manipulate headers to extract an ID to include in a response. See Examples for more use cases.

Definitions

global hook
A hook that applies to all requests or responses configured for a connector.
resource hook
A hook that applies to requests or responses configured for a specific endpoint.
pre-request hook
A script that executes prior to sending API requests. For example, use a pre-request hook to manipulate or add query parameters, headers, the path, the body, or any connector configuration.
post-response hook
A script that executes after recieving a response from an API provider. For example, use a post-response hook to manipulate response headers, the body, or any connector configuration based on the response.

Add Hooks

You can create pre-request and post-response hooks as part of the whole connector configuration, in events, and for individual resources.

- To add global hooks: on the Setup page open the Hooks section, click the **Add hook** button for the type of hook, and then write the script.
- To add resource hooks: add or edit a resource. In the Hooks section of the endpoint, click the **Add hook** button for the type of hook, and then write the script.
- To add event hooks: configure events, click **Add an event hook**, and then write the script.

JavaScript for Hooks

Use JavaScript to write your global or resource hooks. The function signature for all JS when building connectors looks like:

```
/**
* @param  {Object}   request_body                          The incoming request HTTP body
* @param  {Object}   request_body_map                      The incoming request HTTP body, converte
d to a MAP for easy script access
* @param  {Object}   request_headers                       The incoming request headers for this AP
I call
* @param  {Object}   request_path                          The incoming request path for this API c
all
* @param  {Object}   request_parameters                    The incoming query parameters for this A
PI request
* @param  {Object}   request_vendor_parameters             The incoming vendor query parameters for
this API request
* @param  {Object}   request_method                        The request HTTP method
* @param  {Object}   request_vendor_method                 The request vendor HTTP method
* @param  {Object}   request_vendor_path                   The request path
* @param  {Object}   request_vendor_headers                The request vendor HTTP headers
* @param  {Object}   request_vendor_body                   The request vendor HTTP body
* @param  {Object}   request_vendor_body_map               The request vendor HTTP body, converted
to a MAP for easy script access
* @param  {Object}   request_vendor_url                    The request vendor URL
* @param  {Object}   request_expression                    The OCNQL where parameter of the resourc
e, converted to List of Map containing "value, operator, attribute\" to construct the search oper
ation the endpoint needs
* @param  {Object}  request_previous_response              If the endpoint is part of a response ch
ain, a previous request response value. This value can be used to construct the final response fr
om chained API calls.
* @param  {Object}  request_previous_response_headers      If the endpoint is part of a response ch
ain, a previous request response header. This value can be used to construct the final response f
rom chained API calls.
* @param  {Object}   meta_data                             Metadata about the object
* @param  {Object}   configuration                         The Connector's configuration object
* @param  {Object}   vendor_object_name                    This will be the same as object_name unl
ess a vendor object name has been set on the object already. Represented as a string
* @param  {Function} metadata_merge                        Applicable for the endpoint /objects/{ob
jectName}/metadata. If set to true when vendors support metadata and post hook returns CE format
of vendor metadata, it merges the vendor’s metadata with CE model metadata.
* @param  {Function} done                                  The callback function that you will nee
d to call at the end of your JS
*/
function(request_body, request_body_map, request_headers, request_path, request_parameters, reque
st_vendor_parameters, request_method, request_vendor_method, request_vendor_path, request_vendor_
headers, request_vendor_body, request_vendor_body_map, request_vendor_url, request_expression, re
quest_previous_response, request_previous_response_headers, meta_data, configuration, metadata_me
rge, done) {
// your JavaScript goes here
}
```

Note the following when writing Javascript in formulas:

- For all scripts, JavaScript `strict` mode is enforced.
- You can use `console.log` to log data to the JavaScript console to help debug your formula.
- You can use `notify.email` to send an email notification.
- ES6 is supported; see Mozilla's documentation for additional information.
- The function parameters are immutable, meaning they cannot be assigned to directly. To change an object or value passed into the function, first copy it to your own local variable and then make the necessary changes.
- Body variables (request_body) are applicable only to methods that pass a JSON body like POST, PATCH, and PUT. Body variables are undefined/null if there is no JSON body sent.

Note: when adding a `\n` (new line) character to a string, the Javascript editor in Connector Builder will parse the character and add a new line. In order to add a new line character, you must escape the `\n` by adding an additional slash.

For example, the `\n` character would be entered as `\\n`.

The done Function

The `done` function is a callback function that should be called to end the function. It can pass a `continue` object, indicating that the API request should continue to be processed and any new objects that should overwrite the existing incoming objects to this function. An example might be:

```
done({
    "continue": true,
    "request_vendor_parameters": new_request_vendor_parameters
});
```

Send `false` as the `continue` value in a pre-hook, to stop the execution at this point and returns the response. If the request has a postHook, then it will execute that before returning. This can be used to further customize a response.

In the above example, the `request_vendor_parameters` that are returned will overwrite the request vendor parameters that need to be sent to the endpoint.

Libraries

- CE: Our custom library that provides some common functionality. You do not need to `require` this library because it is available by default.
    - `CE.randomString()`: Generate a random string (approx. 10 characters long).
    - `CE.randomEmail()`: Generate a random email address.
    - `CE.md5(str)`: Create an MD5 hash from a string value. Takes a `string` as a parameter. Returns a `string`.
    - `CE.b64(str)`: Encode a string in base64. Takes a `string` as a parameter. Returns a `string`.
    - `CE.decode64(str)`: Decode a string from base64, using UTF-8 encoding. Takes a `string` as a parameter. Returns a `string`.
    - `CE.hmac(algo)(enc)(secret, str)`: HMAC hash a string (*str*) using the provided secret (*secret*), algorithm (*algo*), and encoding (*enc*). See https://nodejs.org/api/crypto.html#crypto_class_hmac for more information about the algorithm and encoding parameters.
    - `CE.hmac[algo][enc](secret, str)`: This is a set of convenience functions that allow HMAC hashing using some common algorithms and encodings. For example, `CE.hmacSha1Hex(secret, str)` will create an HMAC SHA1 hash of the provided string, using the provided secret, and return a hex string. You can replace *algo* and *enc* with the following values: *algo*: `Sha1`, `Sha256`, `Md5` *enc*: `Hex`, `base64`
- Lodash: The popular `lodash` library. To use this library, simply `require` it in your script. It is possible to use the library modules, as well, such as `lodash/fp`.
- Util: The standard Node `util` library. To use, `require` it in your script.

Examples

This section presents some possible use cases for hooks. Because you can write JavaScript, the possibilities available are limited only to your needs and imagination.

- Global Pre-Request Hook for All Delete Methods
- Pre-Request Hook Using Connector Configuration
- Post-Response Hook Reading Response Headers
- Reading Event Webhooks
- Removing Headers

- HTTP and HTTPS Library Examples

Global Pre-Request Hook for All Delete Methods

The hook below applies to all delete method requests. If the request is `delete`, then override or create object with that key.

```
if(request_vendor_method === 'DELETE') {
    request_vendor_headers["Content-Type"] = "*/*";
    done( {
        "request_vendor_headers": request_vendor_headers
    } );
}
```

Pre-Request Hook Using Connector Configuration

This hook is an example of reading a value from the configuration of your connector, then manipulating the data that has been posted to an endpoint.

```
var body = JSON.parse(request_vendor_body);
var contactEmailUpsert = configuration["contact.emailupsert"];
if(contactEmailUpsert === false) {
    done();
}
//Updating the body field contact object with upsert=true
body["contact"] = {
    "upsert": true
};

//Converting the object to string and returning
done( {
    "request_vendor_body": JSON.stringify(body),
    "continue": true
} );
```

Post-Response Hook Reading Response Headers

This hook is an example of reading the response headers to retrieve a value, then extracting that value as an ID and sending it as a response.

The script only executes if the response behaves as expected.

```
if(response_headers === null
    || !(response_status_code === 201
        || response_status_code === 200)) {
    done();
}

//Get the location string from headers
var location = response_headers["location"];
if(location === null) {
    done();
}
//Extract just the id part from the location string
location = location.replace("https://someurl/v1/contacts/","")
location = location.replace(".json","");

//Construct the response body
var response = {
    "id": location
};

//return the response body
done( {
    "response_body": response
} );
```

Reading Event Webhooks

This hook is an example of reading the event webhook types and formatting them into what SAP Cloud Platform Open Connectors expects.

```
var formattedEvents = getArray();
var eventObj = {};
eventObj.event_date = events["modifiedAt"];
eventObj.event_object_id = events["id"];

var webhook_types = events.eventHeaders["x-event"];

if(webhook_types === 'convo.created') {
    eventObj.event_type = 'CREATED';
    eventObj.event_object_type = 'incidents';
} else if(webhook_types === 'convo.updated') {
    eventObj.event_type = 'UPDATED';
    eventObj.event_object_type = 'incidents';
} else if(webhook_types === "convo.deleted"){
    eventObj.event_type = 'DELETED';
    eventObj.event_object_type = 'incidents';
} else if(webhook_types === 'customer.created'){
    eventObj.event_type = 'CREATED';
    eventObj.event_object_type = 'users';
} else if(webhook_types === 'customer.updated'){
    eventObj.event_type = 'UPDATED';
    eventObj.event_object_type = 'users';
}

formattedEvents.add(eventObj);
done( {
    "events" : formattedEvents
} );
```

Note that when the value is not a valid EventType, the `eventObj.event_type` field is set by default to `UNKNOWN` .

Removing Headers

By default, we send `Accept: "application/json"` and `Content-Type: "application/json"` in the headers. If the service provider cannot handle `Accept` or `Content-Type` headers, you can remove them from the request.

In this example, we remove the `Content-Type` header.

```
let headers = {
     "Content-Type": null
};

done({
     "request_vendor_headers": headers
...
});
```

HTTP and HTTPS Library Examples

Use the HTTP and HTTPs libraries to make requests from a hook to any HTTP or HTTPS endpoint.

```
const https = require('https');
//Get SFDC connector from CE and return the results
https.get('https://api.openconnectors.us2.ext.hana.ondemand.com/elements/api-v2/elements/sfdc', (
res) => {
  console.log('after response');
    let rawData = '';
    res.on('data', (chunk) => rawData += chunk);
    res.on('end', () => {
      try {
        let parsedData = JSON.parse(rawData);
        done({ "response_body": parsedData });
      } catch (e) {
        console.log(e.message);
        done({ "response_error": e.message});
      }
    });
});
```

Example using http:

```
const http = require('http');

//Call Swagger petstore
let options = {
  hostname: 'petstore.swagger.io',
  path: '/v2/store/inventory',
  headers: {
    'Accept': 'application/json'
  }
};

const apiCall = http.request(options);
apiCall.on('response', res => {
     console.log('after response');
     let rawData = '';
     res.on('data', (chunk) => rawData += chunk);
     res.on('end', () => {
       try {
         let parsedData = JSON.parse(rawData);
         console.log('Parsed response');
         done({ "response_body": parsedData });
       } catch (e) {
         done({ "response_error": e.message});
       }
     });
 });

apiCall.on('error', err => {
  done({ "response_error": err.message});
});

apiCall.end();
```