Unica Link V12.1
Creating Link Connectors Tutorial

Contents

Introduction 3
Approach 3
Scenario 4
Requirements 4
Validation of Technical Requirements 4
Webapp Simulator for testCONN 6
Task 1 – Setup 6
Step 1 – Install the webapp 6
Step 2 – Install the test application 7
Task 2 – Create a Service Definition 8
Step 1 – Define the REST service 8
Step 2 – Define a Test Connection endpoint (optional) 11
Step 3 – Define the Send Email endpoint 13
Step 4 – Define the Get Templates endpoint 15
Task 3 – Create the Connector Project 16
Step 1 – Create (or Import) the Project 16
Step 2 – Create Link Schema for JSON Templates for API Endpoints 19
Step 3 – Create Schema for Input and Output CSV 22
Step 4 – Create Request Map 26
Step 5 – Create Response Map 35
Step 6 – Create Error Response Map 37
Step 7 – Create First Flow 39
Step 8 – Running the Flow 47
Step 9 – Enhance Flow to Process Large Data 50
Step 10 – Extend the Request Map 54
Step 11 – Updating and Running the Bulk Flow 57
Task 4 – Create the Properties Descriptor 60
Task 5 – Package the Connector 62
Task 6 – Install the Connector 64
Task 7 – Test the Connector 65
Step 1 – Define a Server 65
Step 2 – Create a Connection 66
Running the Action 69

© Copyright HCL Ltd. 2021 All rights reserved

Next Steps 72

© Copyright HCL Ltd. 2021 All rights reserved

Introduction

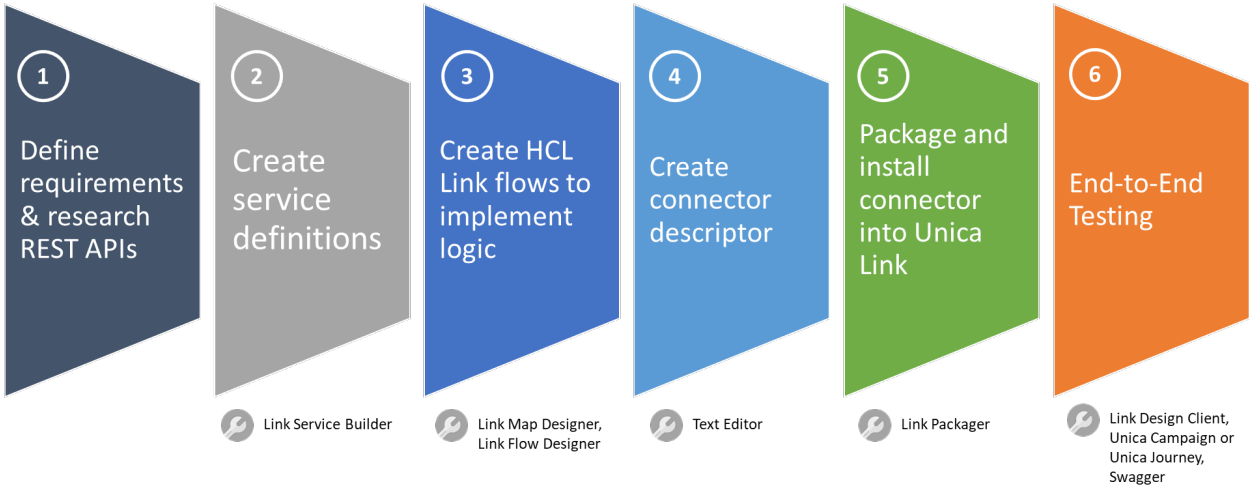
This document provides a walkthrough of the steps required to create, test, and deploy a Link connector. The document Creating Link Connectors – Reference provides more details about the concepts of Link connectors and should be read in parallel with this document.

This tutorial provides step-by-step instructions for creating a connector, using the hypothetical example 'testconn' provided in the development package. The tutorial is aimed at those who are inexperienced with Link, but for experienced users it provides a useful guide for the logical steps in creating a connector.

The tutorial describes a step-by-step approach where one first builds a simple solution, then incrementally expands upon it to produce the complete connector.

Approach

Creating a Unica Link custom connector consists of completing the tasks shown below.



1. The first task is to **define the requirements** for the custom connector and then to **verify that appropriate REST APIs or available integration methods are provided** by the third-party application to be able to meet those requirements. This is frequently the most difficult and time-consuming task, but it is also the most important. As with any other development project, if you do not know what you're trying to accomplish, you will never accomplish anything. Once the requirements are documented and the REST APIs identified, you are ready to begin creating the connector.

2. Next, you **use the Link Service Builder to create service definitions** for the REST APIs and endpoints that will be used to connect to the third-party application/service.

3. After creating the service definitions, you **create HCL Link artifacts** – integration flows, transformation maps and schema definitions – **that implement the connector logic**.

Copyright © 2021 HCL Software

4. Once the connector is developed, you **create a simple connector descriptor**, which is a JSON text file, that identifies the connector in Unica Link and displays the Unica Link user interface within Unica Campaign and Unica Journey.

5. Now, you are ready to **package and install the connector** using the Link Packager utility provided in the development kit.

6. Finally, it is time to do **end-to-end testing**, which can be done from the HCL Link Design Client, Unica Campaign, Unica Journey and Swagger.

Scenario

For the purposes of this tutorial, you want to connect Campaign or Journey to a service provided by fictitious testCONN, which sends personalized emails to the audience list you provide.

Requirements

As described above, the first step is to define what you want the connector to do. The desired functionality for this tutorial connector is fairly simple and straight-forward. Ideally, the connector shall:

- Work in both Campaign and Journey.
- Enable the Marketer who is creating the flowchart or journey to specify the template to be used to send the personalized email.
- Invoke public REST APIs provided by testCONN to send an email to a designated audience member.
- Process the response(s) from testCONN to provide those results back to Campaign and/or Journey.
- Periodically poll (every 4 hours for 30 days) for additional results responses from testCONN to provide those results back to Campaign and/or Journey.
- Correlate the result responses received with the emails sent.

Validation of Technical Requirements

The next step is to see if the APIs provided by the testCONN service enable the desired functionality. Taking the requirements one-by-one, here is what we have discovered:

☑ Work in both Unica Campaign and Unica Journey.
  - Unica Link is designed in such a way that you can create a single connector that works with both Unica Campaign and Unica Journey.
  - Unica Campaign and Unica Journey interface with Unica Link by delivering an audience list via a defined channel (a file for Campaign, a message on a Kafka topic for Journey), which is delivered to Unica Link as CSV formatted data.

☑ Enable the Marketer who is creating the flowchart or journey to specify the template to be used to send the personalized email.
  - Unica Link provides a mechanism via a connector descriptor to give users the ability to customize the settings for their connector.

Copyright © 2021 HCL Software

- The testCONN REST API provides an API (template) that will get a list of templates that can be used

☑ Invoke public REST APIs provided by testCONN to send an email to a designated audience member.
- testCONN is a hypothetical email service with a simple REST API that mimics an email delivery engine.

```
POST /sender/immediate
```

- This REST API takes a JSON-formatted request, sends the email, and then returns one of two JSON-formatted responses – one if the email is sent successfully and a different one if sending the email resulted in an error.

☑ Process the response(s) from the public REST API(s) provided by testCONN to provide those results back to Unica Campaign and/or Unica Journey.
- Unica Link has a defined standard response history CSV format, which the two JSON-formatted responses from the testCONN REST API can be mapped to produce.
- Campaign and Journey will consume this response data and update response history accordingly.

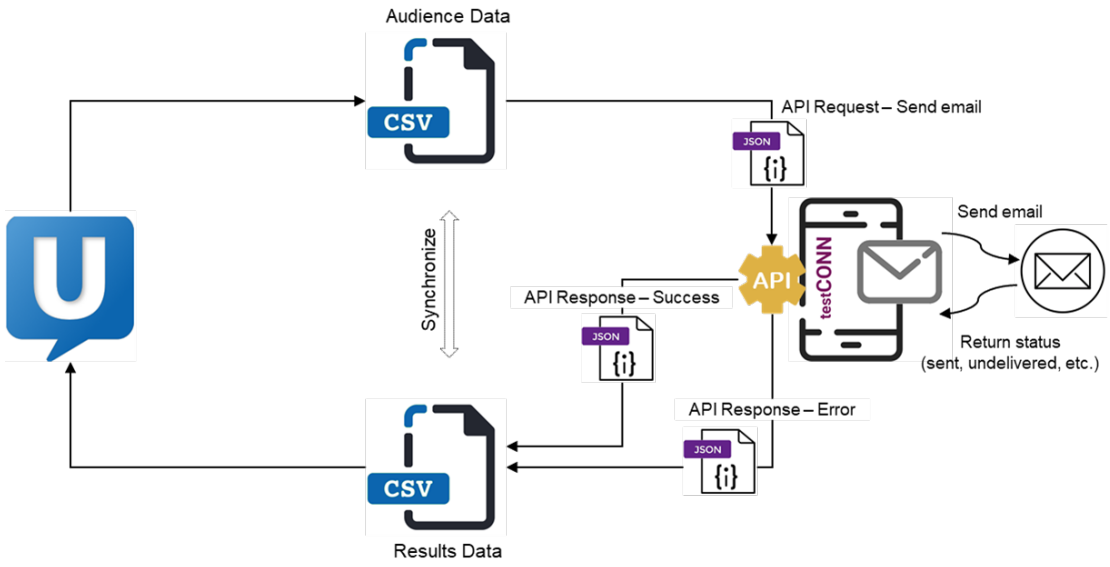
☐ **(NOT SUPPORTED)** Periodically poll (every 4 hours for 30 days) for additional results responses from testCONN to provide those results back to Campaign and/or Journey.
- testCONN does not provide an API to be able to poll for results after the email is sent. Status of the send is returned at the time the email is sent as part of the immediate response.

☑ Correlate the result responses received with the emails sent.
- Campaign and Journey can pass context information for each audience member that allows the contact to be identified uniquely.
- Testconn can receive context information in the Send email request API and return it with its status response.
- Therefore, this context information can be returned to Campaign or Journey with the contacts results.

As you can see, the desired functionality is limited by what the third-party application supports. In our tutorial, polling is not supported, so it will not be possible to support that in the testCONN connector.

Copyright © 2021 HCL Software

The following diagram illustrates the data flow of this simple custom connector starting at the Unica logo at the left and going clockwise.



Webapp Simulator for testCONN

Included in the development package with this tutorial is a simple webapp (called testapp) that implements this API. The main API is:

```
POST /sender/immediate
```

where the body is a JSON object that has the email address and other attributes that would be used in the email body if this service actually sent an email. This API returns a status that indicates whether the email was sent, undelivered or bounced. It returns these statuses randomly. It also validates that the given email address is validly formed, i.e., has an '@' and a '.' character, and returns a 400 status if the format is invalid.

Task 1 – Setup

Perform the following steps to set up the development environment for this tutorial.

Step 1 – Install the webapp

The first step is to install the testapp.war file in the Link runtime server. This needs to be copied from the testapp directory in the toolkit to the /usr/local/tomcat/webapps directory in the server.

1. If using the Docker version of Link, issue this command running as the user that was used to install Link:

   ```
   docker cp testapp.war hip-rest:/usr/local/tomcat/webapps
   ```

   If using a native (non-Docker) install, the war file can be directly copied to the /usr/local/tomcat/webapps directory in the Link install. Once copied, it will be automatically installed by Tomcat.

2. You can verify that it was installed by invoking the 'test connection' API. The webapp uses Basic authentication, but simply checks that the password value is "good". To test that the webapp is installed invoke:

   ```
   curl -u test:good http://localhost:8080/testapp/
   ```

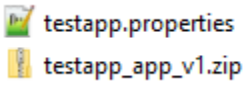
   Adjust the scheme, host, and port accordingly. If the webapp is installed correctly it will return:

   ```
   {"status":"OK"}
   ```

Step 2 – Install the test application

As well as implementing the 'send email' service, the test application also mimics the application that Link is integrated with. This application is invoked from the field mapping UI when configuring an action.

1. To install the test app copy these files from the testapp directory to the modules directory on the server (which is /opt/hipmodules by default):

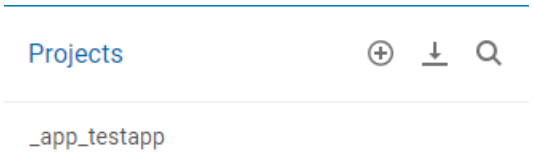
   

2. Then edit the testapp.properties file and specify the base_url to point to the server where testapp.war was installed. This needs to specify the IP address of the server: "http://<ip-address>:8080/testapp". Specifying 'localhost' will not work because that is not resolved when invoking the Link runtime container from the Link server container.

3. Restart the hip-server server to install the testapp app:

   ```
   docker restart hip-server
   ```

4. After the server starts refresh the project list and you should see a project named _app_testapp:

Now that setup is complete, the following steps should be followed in sequence to create the necessary artifacts for the connector.

Task 2 – Create a Service Definition

To meet the requirements described in the scenario earlier in this document, you need to define the Testapp service and three APIs in Link's Service Builder:

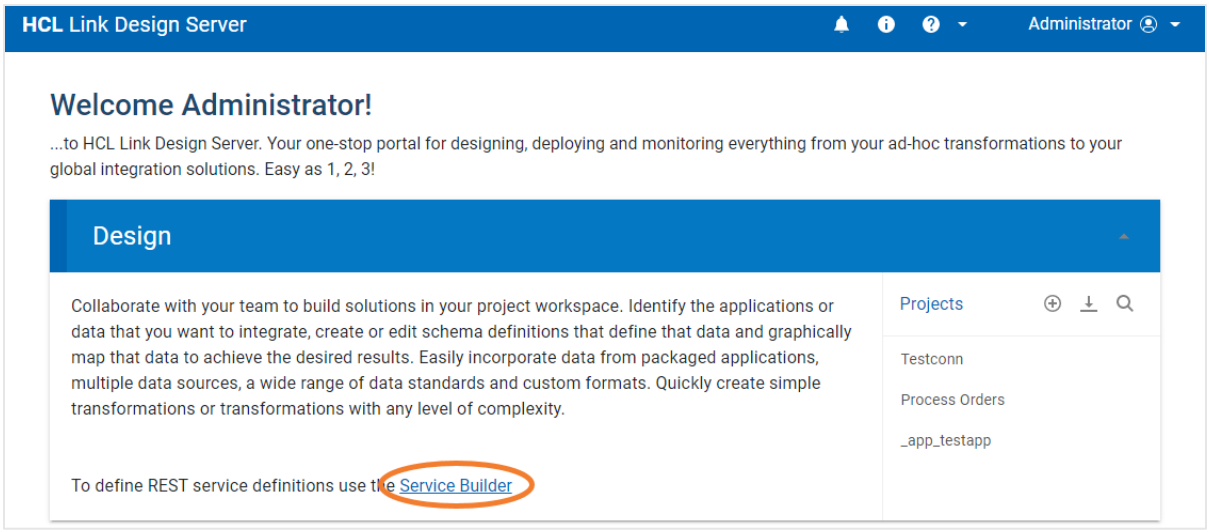
- Test connection: GET {$base_url}/sender/test
- Send email: POST {$base_url}/sender/immediate
- Get a list of templates: GET {$base_url}/templates

Step 1 – Define the REST service

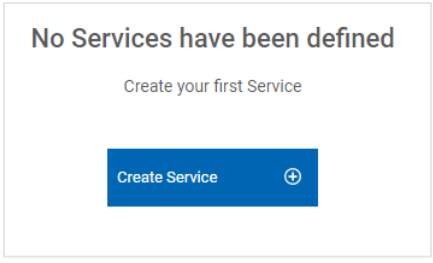
Testapp uses basic authentication to authenticate the user and the only accepted password is 'good'.

To define a REST service for testCONN:

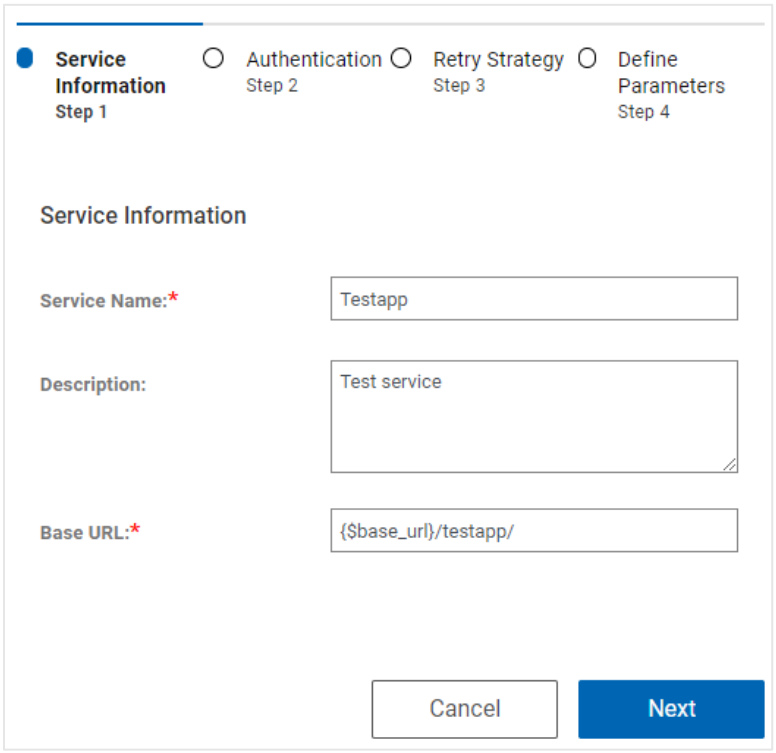
1. Launch Link in your browser and login on the main screen.
2. Click on **Service Builder** on the home page.



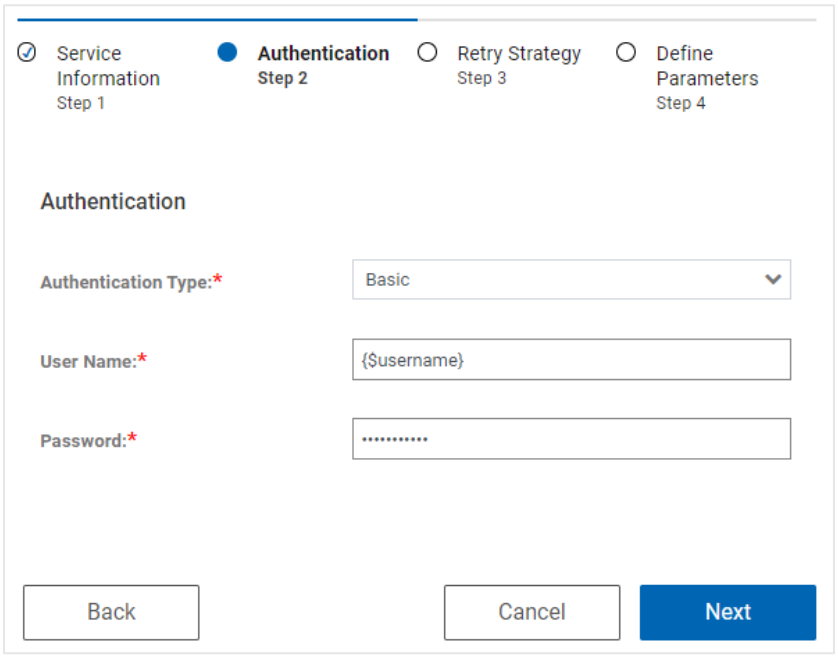
3. Create a new service by clicking on the 'Create Service' button.



4. Name the service and specify the URL. Use a parameter value for the server address, so it can be provided at runtime. Parameters are specified by using notation "{$variable-name}".

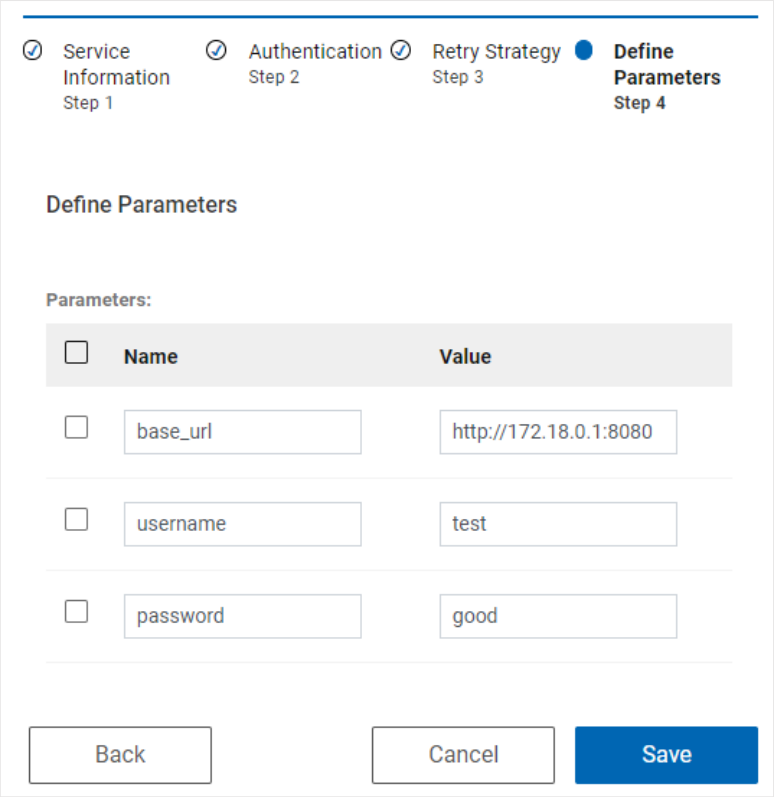


5. Specify basic authentication in the next page and specify the username and password as parameters.

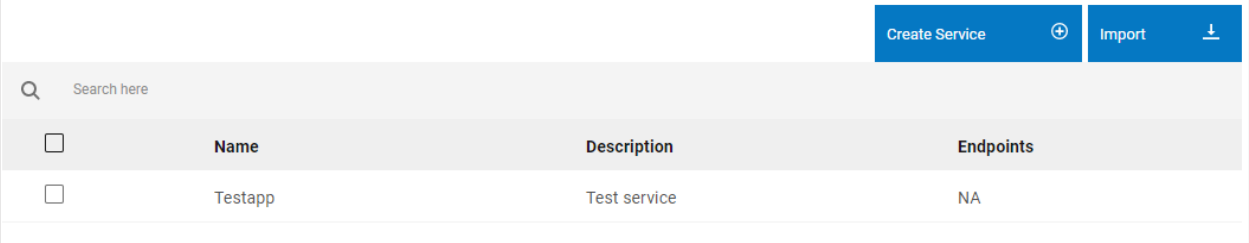


6. Advance to the last step where the parameters table will be displayed. Populate the values with default values that will be used when invoking endpoints in the editor. These parameter values can be set to different values when we subsequently invoke the endpoints.

   The base_url should include the IP address of the host server. If Link was installed as Docker containers, specifying 'localhost' will not work because that is not resolved from the server docker container.



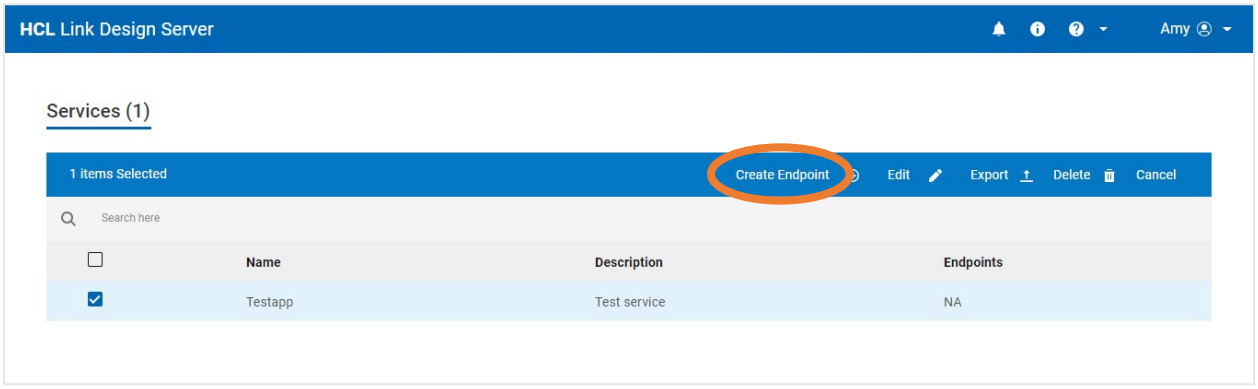
7. Click Save to save the service definition.



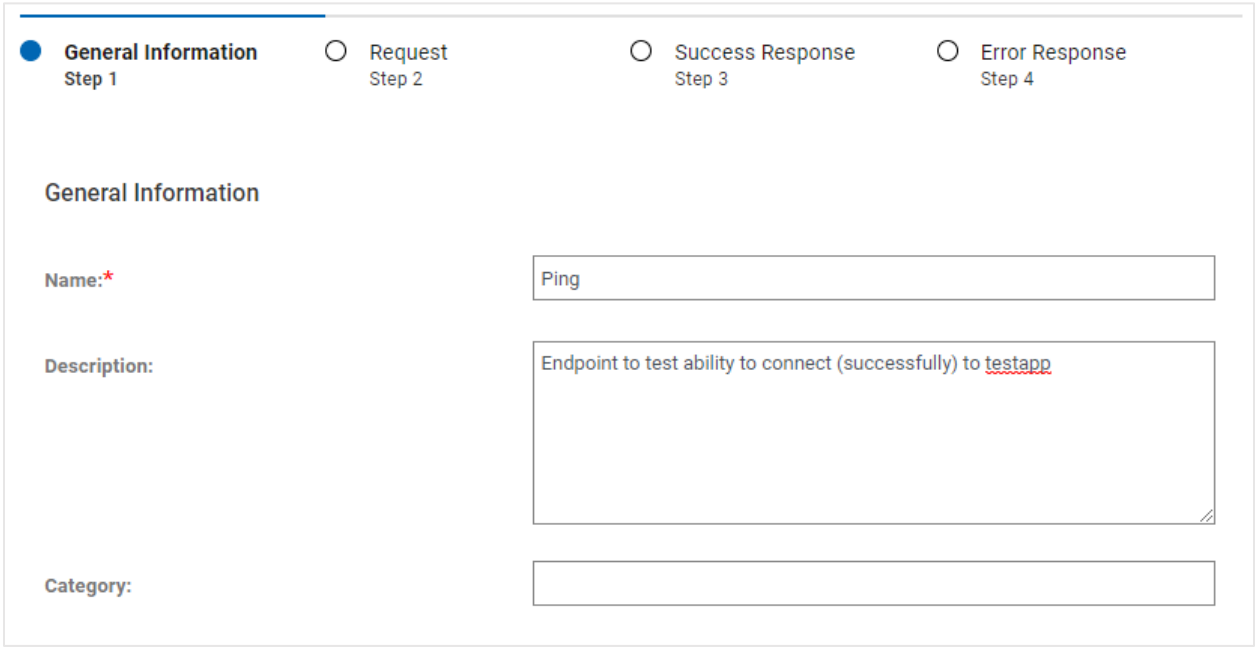
Step 2 – Define a Test Connection endpoint (optional)

The next step is to create an endpoint for the 'test connection' API. This is optional but is a good idea to ensure that the configuration is correct. The test connection endpoint will be invoked when a user creates a connection to the resource.

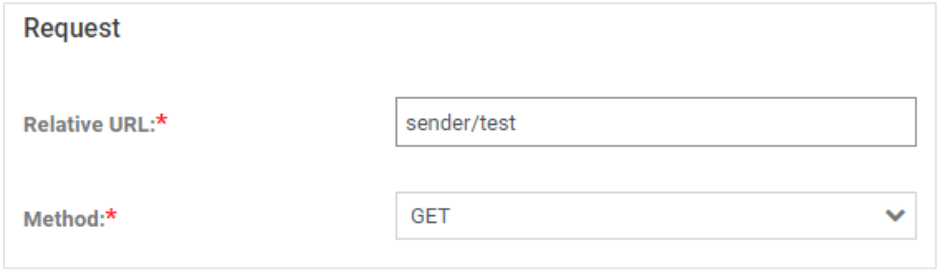
1. Select the **Testapp** service you just created and then click on **Create Endpoint**



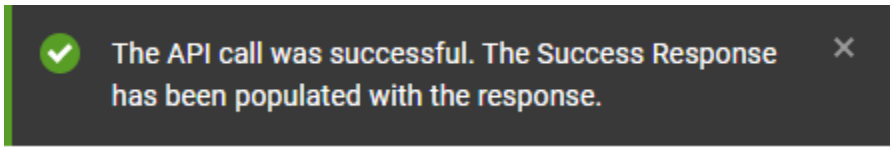
2. Name the endpoint **Ping** and click **Next**.



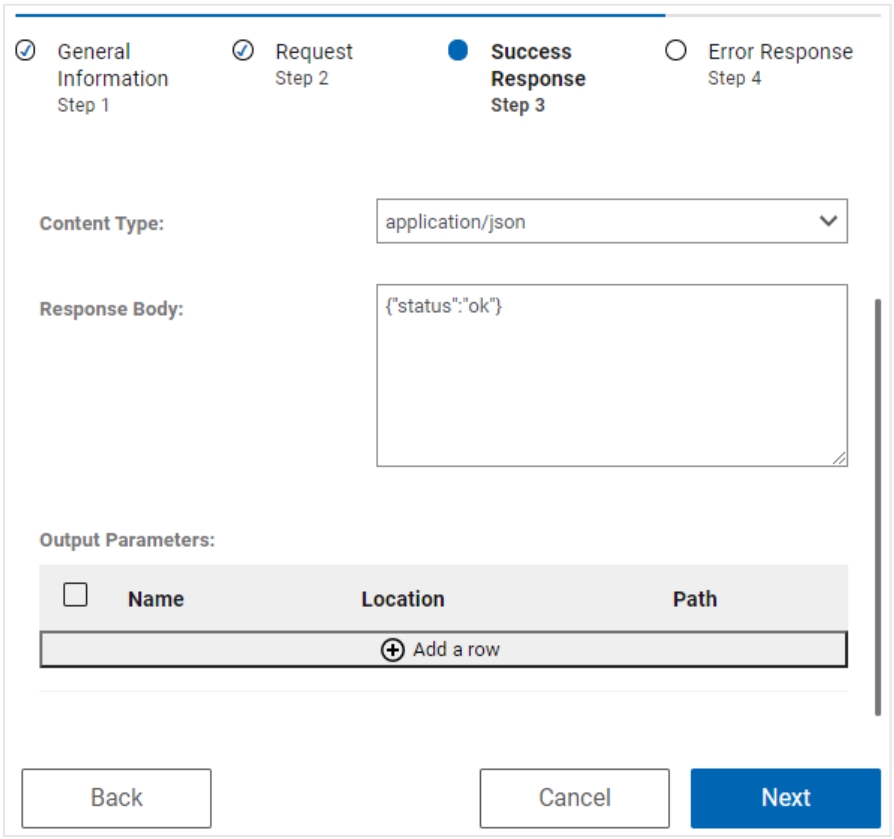
3. In the **Request** page, specify the **Relative URL** and **Method**. The relative API is the path added to the base URL that is defined for the service. Alternatively, you can put an absolute path starting with http/https here.



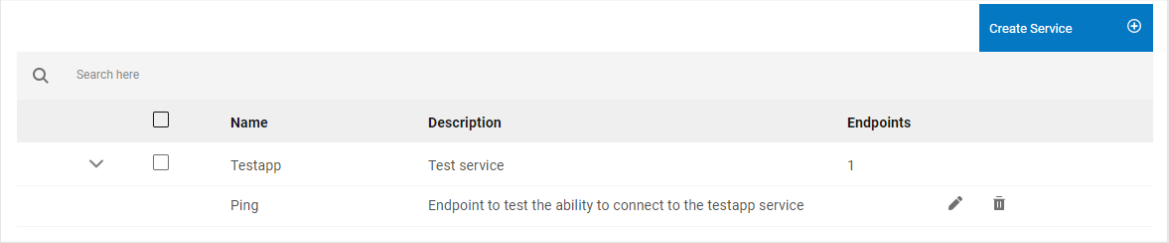
4. Click **Call API** at the bottom of the Request page to invoke the API. If the call is successful, it will display this message and the Success Response page will be populated.



5. Click **Next** to view the successful response:



6. Click **Next** and then **Save** to save the endpoint.

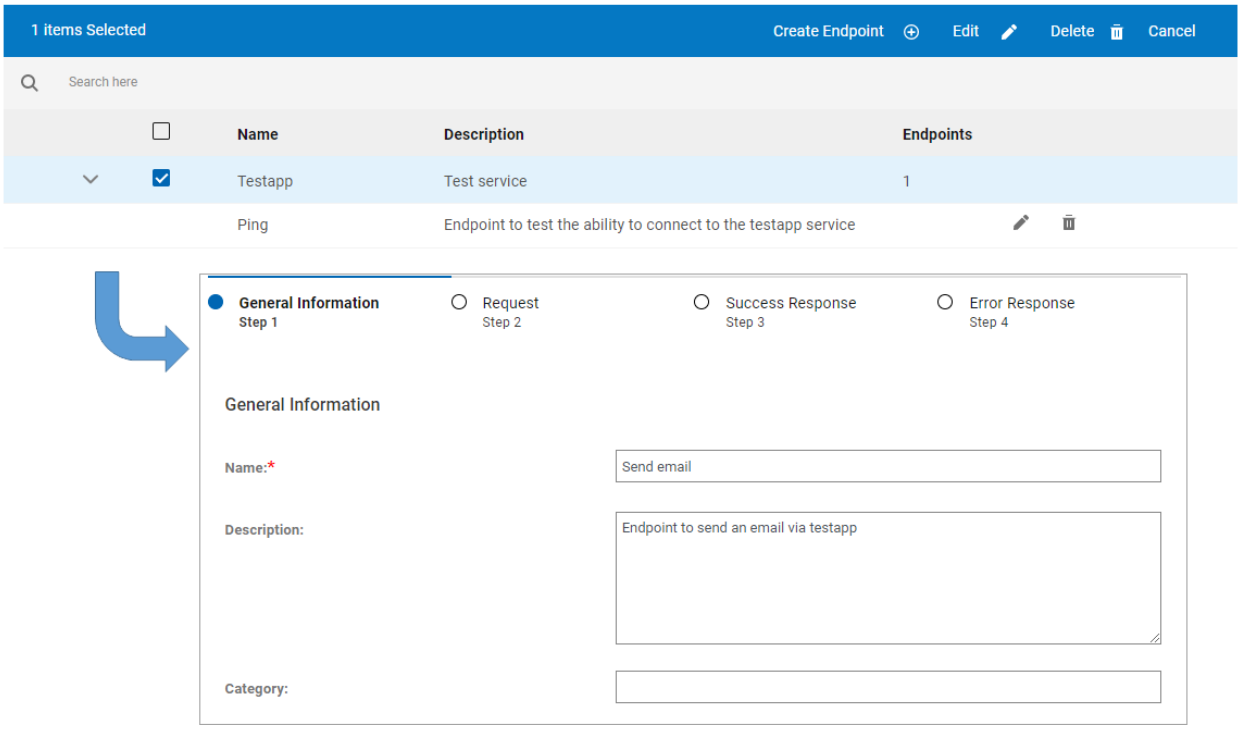


Step 3 – Define the Send Email endpoint

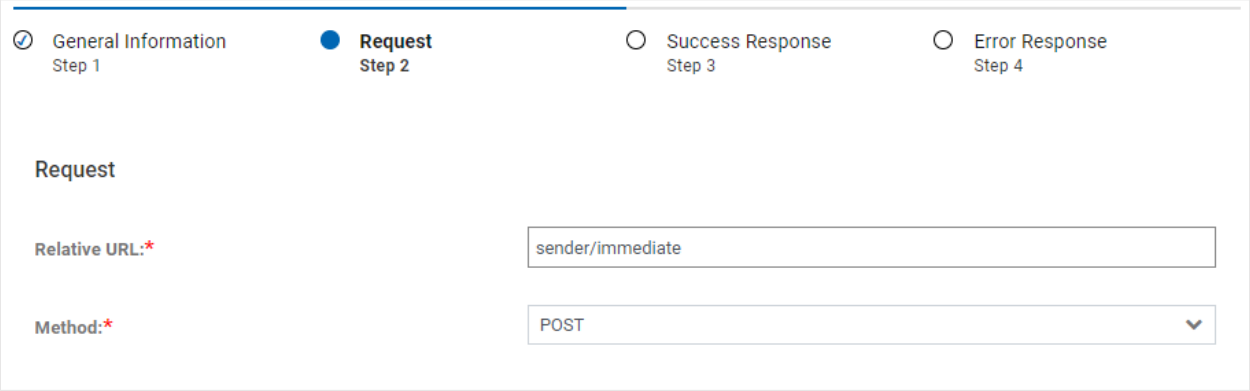
The 'Send email' endpoint requires a JSON request body that looks like this:

```
{
    "email": "fred@test.com",
    "first_name": "Fred",
    "last_name": "Smith",
    "template": 12345
}
```

1. Create a new endpoint for Testapp named 'Send email' and select Next.

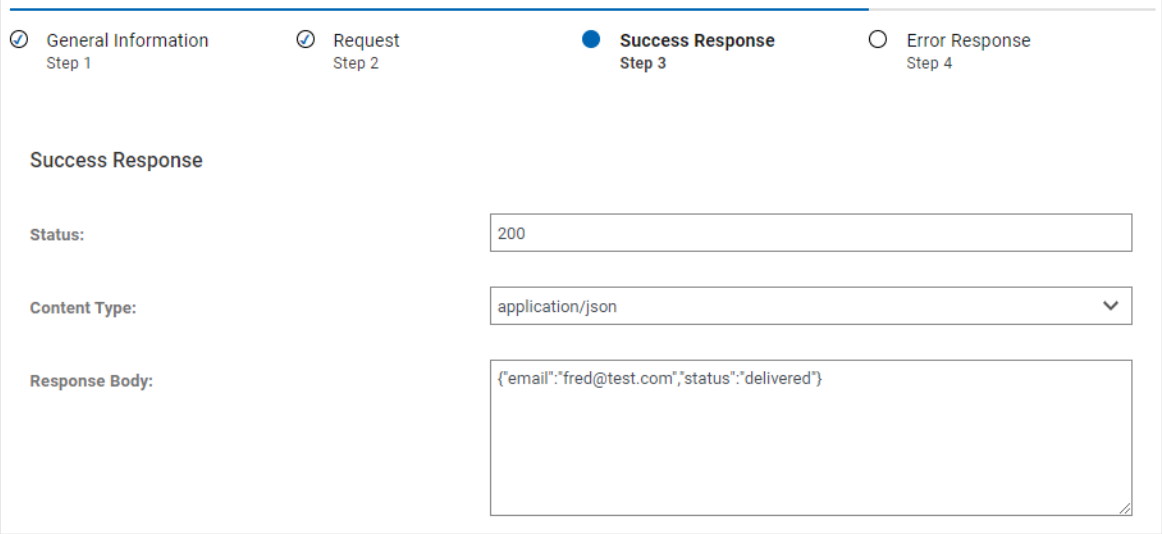


2. Enter the relative URL and method properties on the Request page.

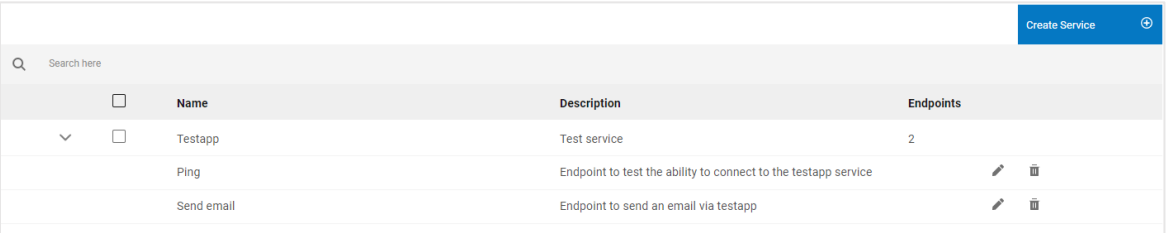


3. Then, scroll down on the Request page and specify the content type and body, as shown.

4. Click Call API. The API call should be successful and the Success Response populated.



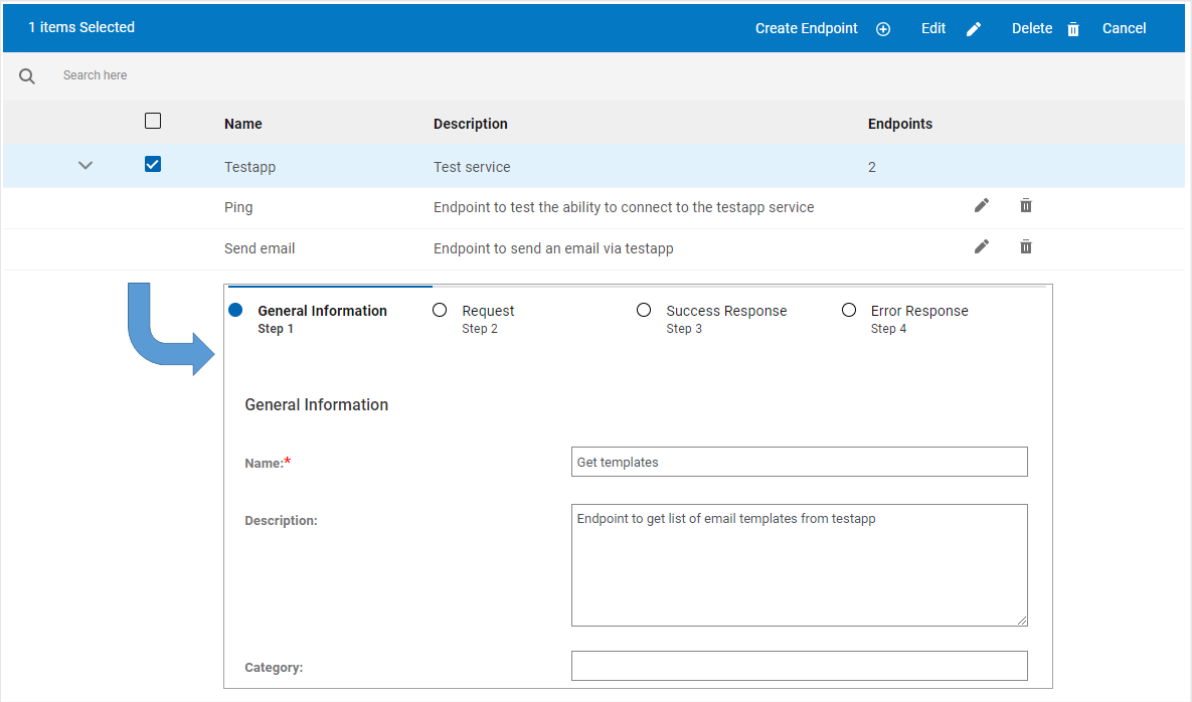
5. Click Next and Save to save the endpoint definition.



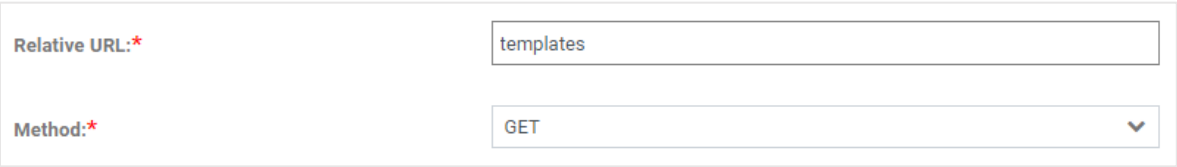
Step 4 – Define the Get Templates endpoint

Finally, repeat the same steps to create an endpoint for invoking the 'get templates' API.

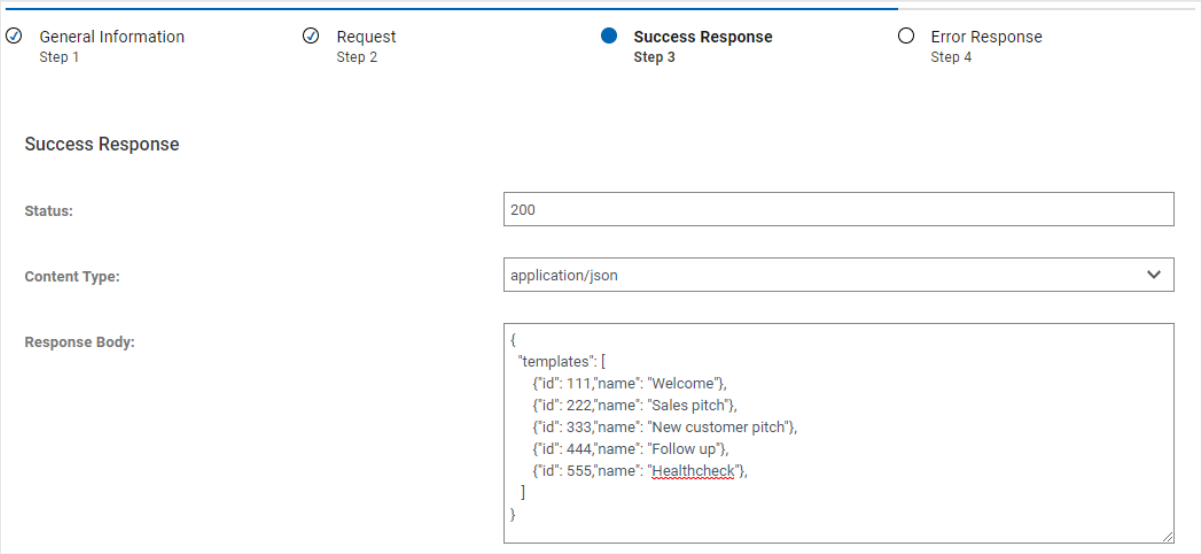
1. Create a new endpoint for Testapp named “Get templates” and select Next.



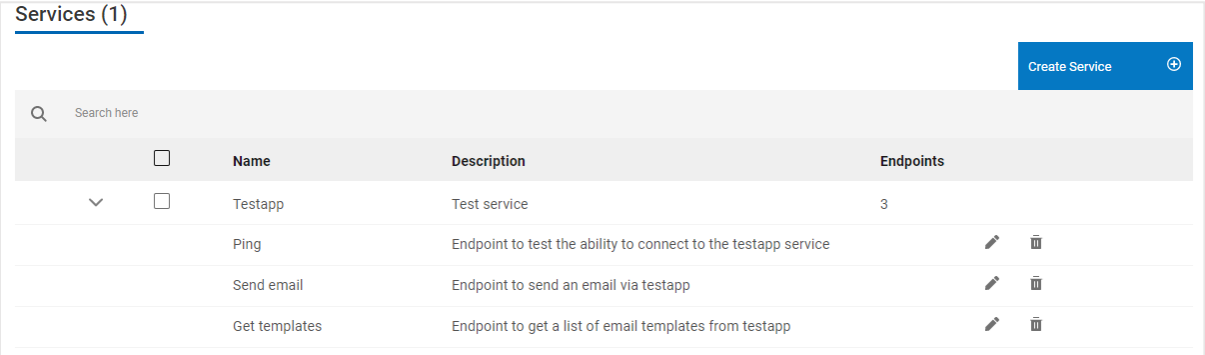
2. Enter the relative URL and method properties on the Request page.



3. Click Call API. The API call should be successful and the Success Response should be populated:



4. Click Next and Save to save the endpoint definition. Having defined the testapp service and the three endpoints, the Service Builder should show these entries.



We can now reference the service in the project.

Task 3 – Create the Connector Project

This section walks through the steps of creating a project for the connector implementation and then creating the various artifacts within the project.

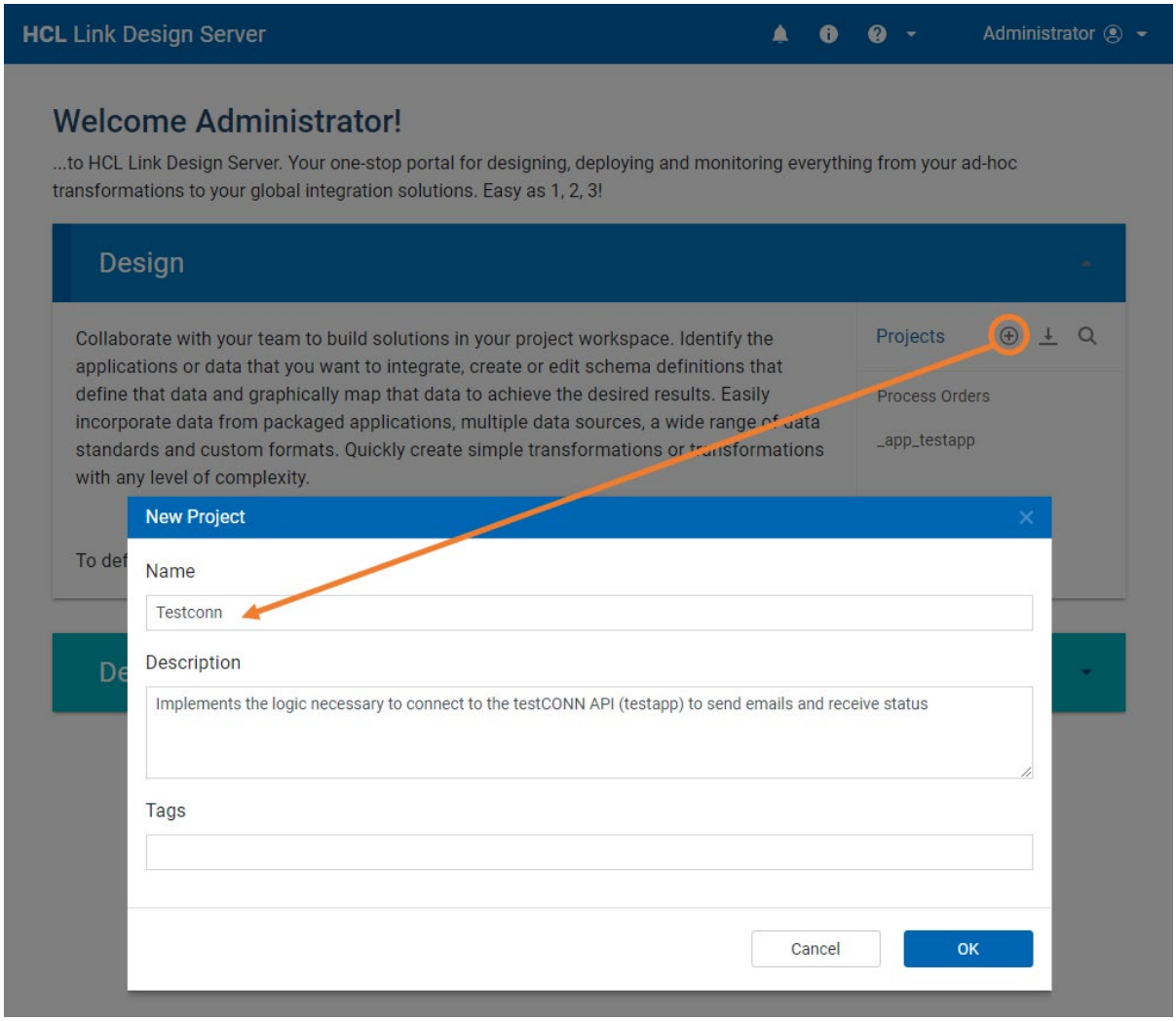
Step 1 – Create (or Import) the Project

The next step is to create a Link project for the new connector. In this tutorial, the project and artifacts are already created, so you can simply import the existing project. If creating a new connector, one would create a new project and populate it with artifacts.

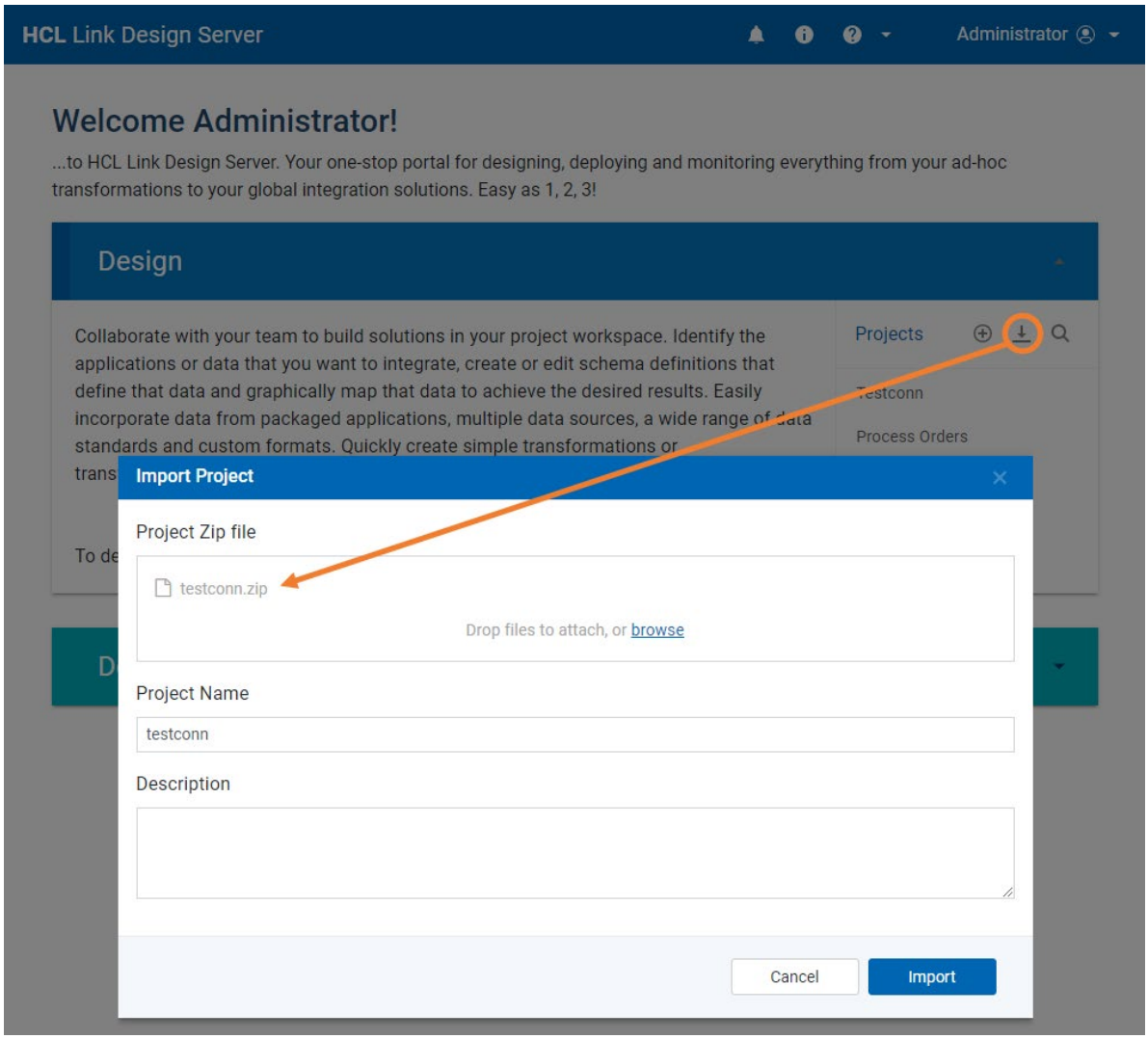
1. Go to the Link home screen. (Note: From the Service Builder, click on **HCL Link Design Server** in the upper left blue title bar.)

To continue the tutorial, either create the project (step 2) or import the project (step 3) before continuing with step 4.

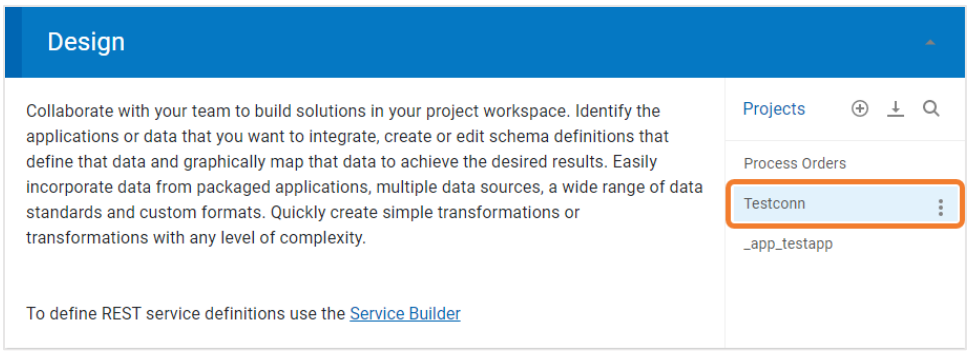
2. **To create a new project**, plus sign in a circle icon above the project list and give your project a meaningful name.



3. **To import the existing project**, click on the ⬇ icon and select the testconn.zip project file.



4. Open the project by clicking on its name from the list of projects.



Step 2 – Create Link Schema for JSON Templates for API Endpoints

The testCONN API request and response messages are JSON formatted. You need to define Link schemas that describe the format of these JSON formatted messages. Link can automatically generate those schemas based on a “sample JSON document” – also called a template. There are three API formats for which we need to import templates – the request, the response, and the error response. These schemas will be used with the REST node when we create the Link flow that implements the connector logic.

We will be creating a flow that uses the REST Client node in Multiple Request mode. In this mode, a JSON array of objects is passed in and the REST API is invoked for each object in the array. This mode also allows for a context object to be passed in, which is then copied to the response, allowing for context for an individual request to be passed through the flow.

Request

This file is saved in the schemas directory of the tutorial as file `sendEmail_request.json`.

```
[
    {
        "_request_": {
            "email": "fred@test.com",
            "first_name": "Fred",
            "last_name": "Smith",
            "template": 123
        },
        "_context_": {
            "identity": "xxx"
        }
    }
]
```

Response

This file is saved in the schemas directory of the tutorial as file `sendEmail_response.json`:

```
[
    {
        "_response_": {
            "email": "fred@test.com",
            "status": "undelivered"
        },
        "_context_": {
            "identity": "xxx"
        }
    }
]
```

Copyright © 2021 HCL Software

Error Response

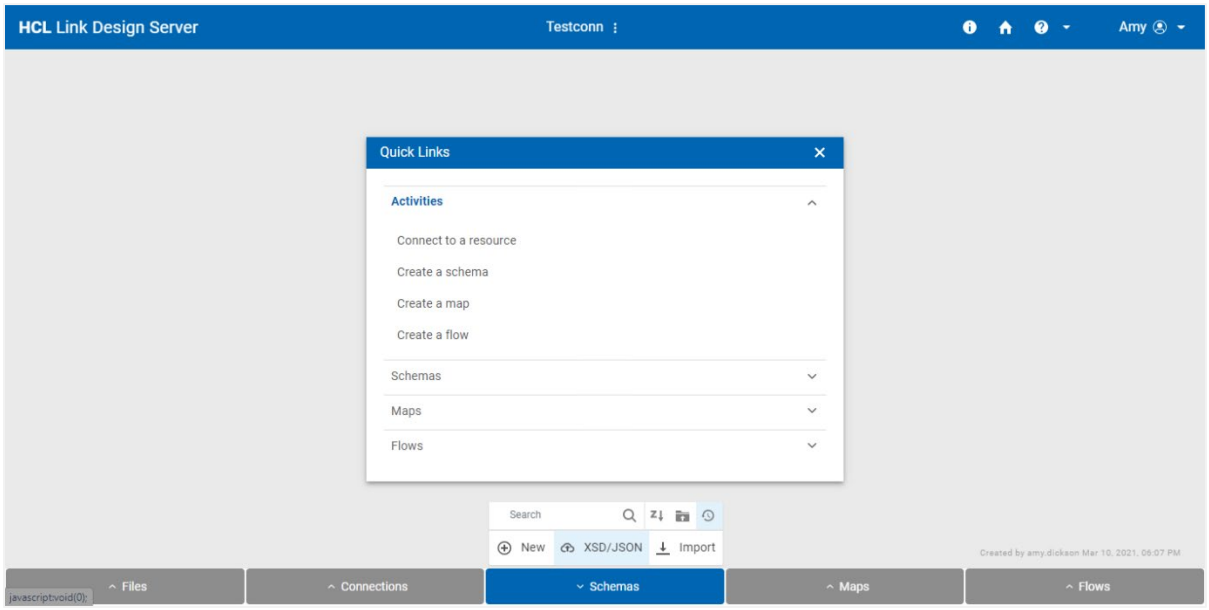
The API produces a different format for an error response. This file is saved in the schemas directory of the tutorial as file `sendEmail_error_response.json`.

```
[
    {
        "_context_": {
            "identity": "002"
        },
        "_response_": {
            "status": 400,
            "message": "",
            "response": {
                "email": "bad-email.com",
                "error": "email is not valid"
            }
        }
    }
]
```

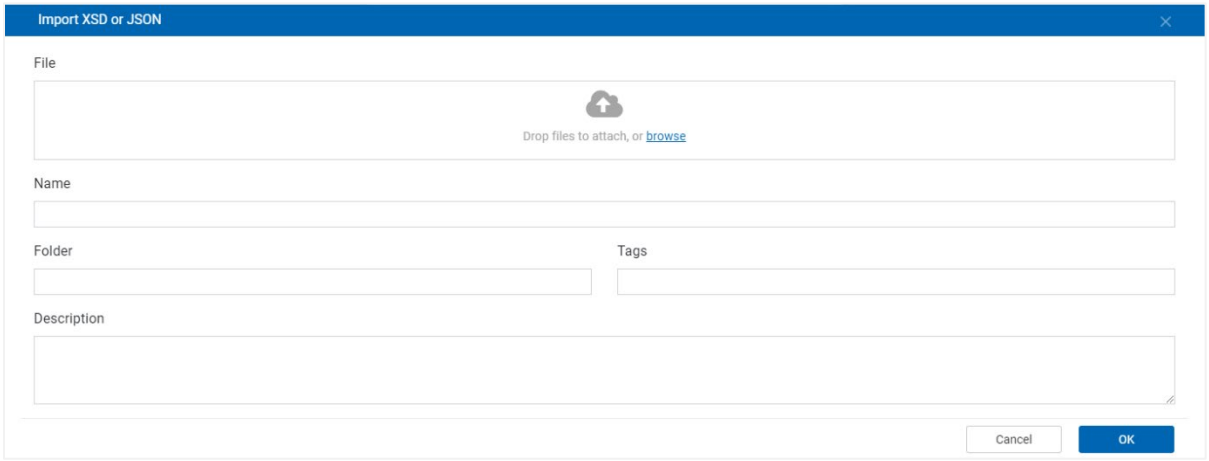
All three files need to be imported into the project by clicking on the Schemas list and selecting XSD/JSON. Select each file in turn and import into the project.

Creating the Schemas

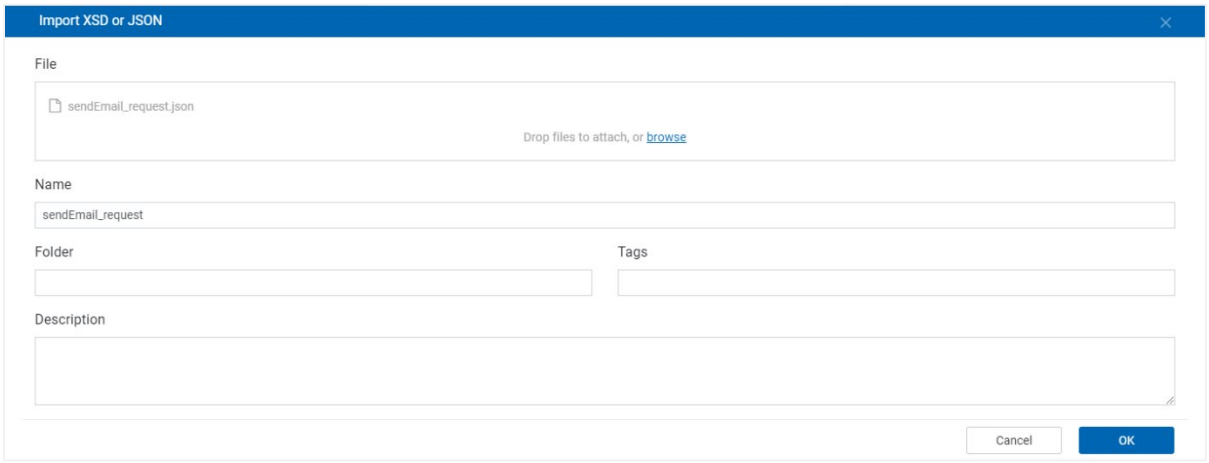
1. From within the _app_testapp project, click on **Schemas** from the bottom navigation bar and click on **XSD/JSON** from the popup menu.



2. Drag `sendEmail_request.json` to the Link window or browse to locate and select the file from the <devkit>\testconn\schemas directory.

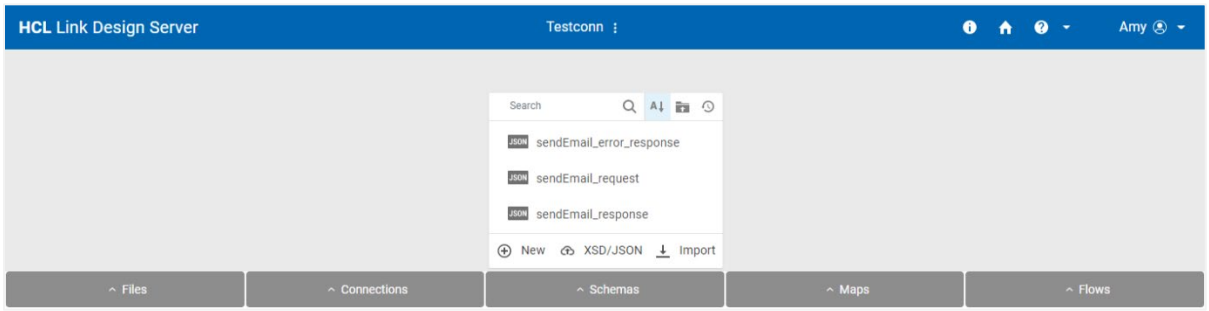


3. Confirm the name for the schema and add a description (optional), then click OK.



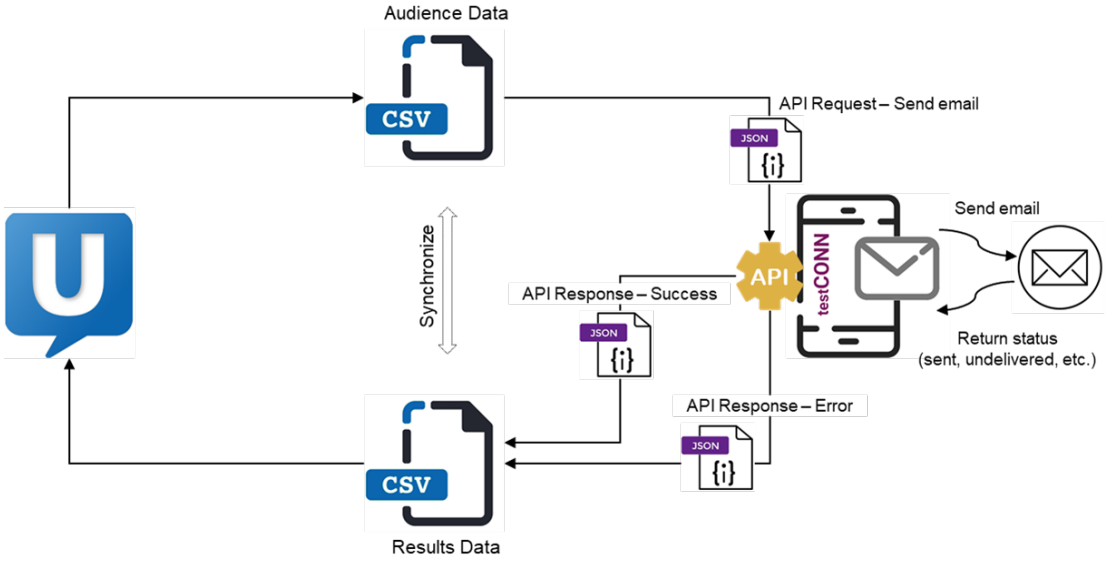
4. Repeat steps 1-3 for the response and error response schemas.

5. Click on **Schemas** from the bottom navigation bar and you should see the three JSON schemas that you just created.



Now we have the input and output definitions for the REST calls ready.

Step 3 – Create Schema for Input and Output CSV



As shown above (and previously), the connector's flow will receive a CSV file (containing audience data from Campaign or Journey), map from that CSV to the JSON request, and invoke the REST API with each JSON object. It will then map the results returned by the REST API to a CSV results file that is returned from the connector to Campaign or Journey.

To develop our map and flow we need to create a CSV data file and its corresponding schema, which we can do using the Link CSV import capability.

The file `<devkit>/testconn/schemas/input5.csv` is a small CSV file containing a header row and five rows of data that represent the audience data from Campaign or Journey identifying the emails to be sent via testCONN:

```
email,first,last,identity
name1@test.com,First1,Last1,0001
name2@test.com,First2,Last2,0002
name3@test.com,First3,Last3,0003
name4@test.com,First4,Last4,0004
bad-email.com,First5,Last5,0005
```

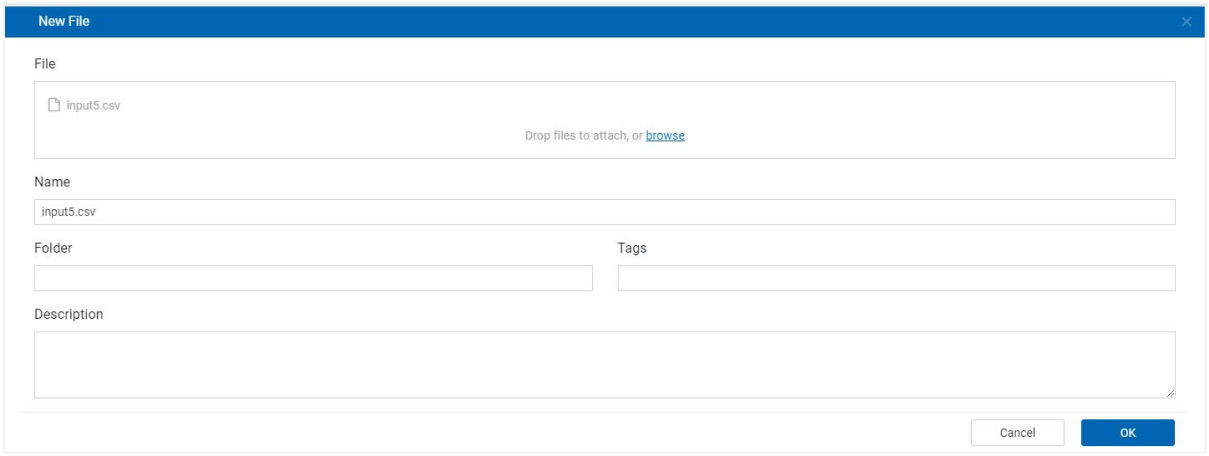
The file `<devkit>/testconn/schemas/response.csv` is a small CSV file containing a header row and one row of data that represents the response results data to be sent to Campaign or Journey:

```
email,status,errormsg,timestamp,identity
fred@test.com,sent,,2020-07-20T13:01:19,001
```

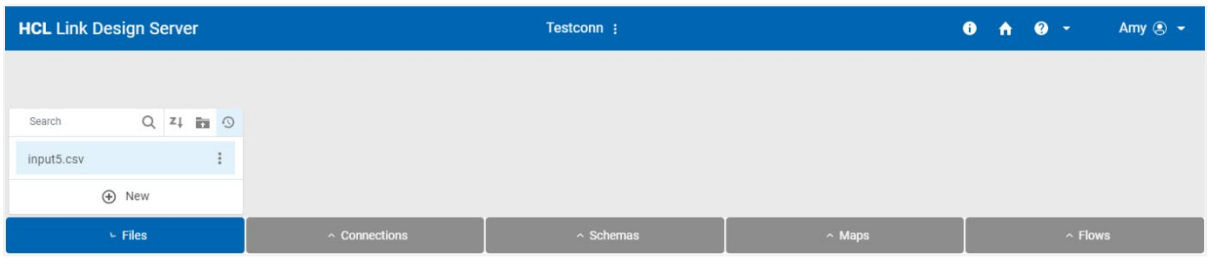
Add the CSV Files to the Project

1. From within the _app_testapp project, click on **Files** from the bottom navigation bar and click on **New** from the popup menu.

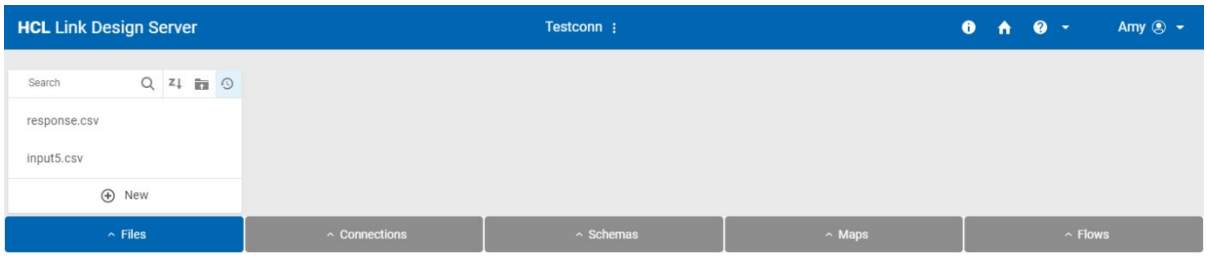
2. Drag `input5.csv` to the Link window or browse to locate and select the file from the <devkit>\testconn\schemas directory. Click **OK**.



3. Click on Files from the bottom navigation bar to view the file you just added.



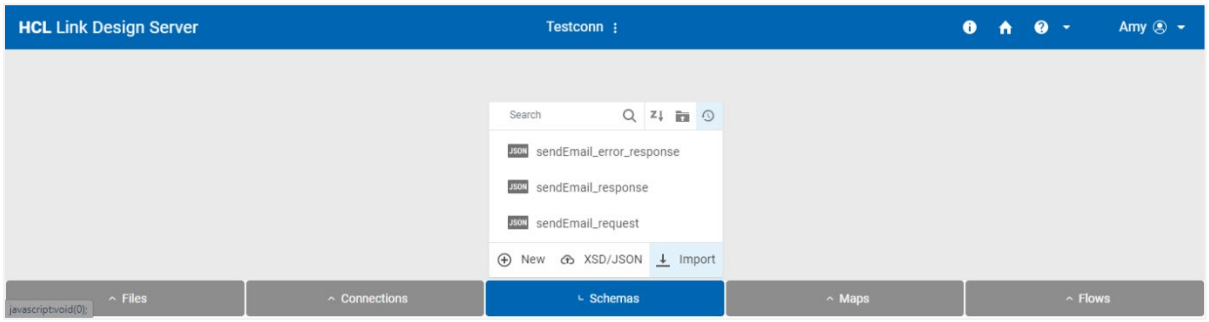
4. Repeat steps 1-3 for <`devkit>/testconn/schemas/response.csv`.



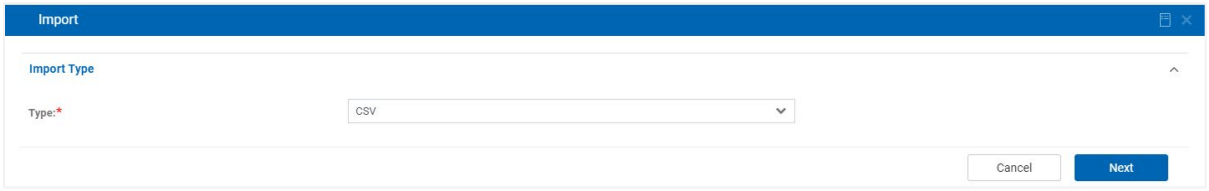
Copyright © 2021 HCL Software

Create the Schema for the Input CSV

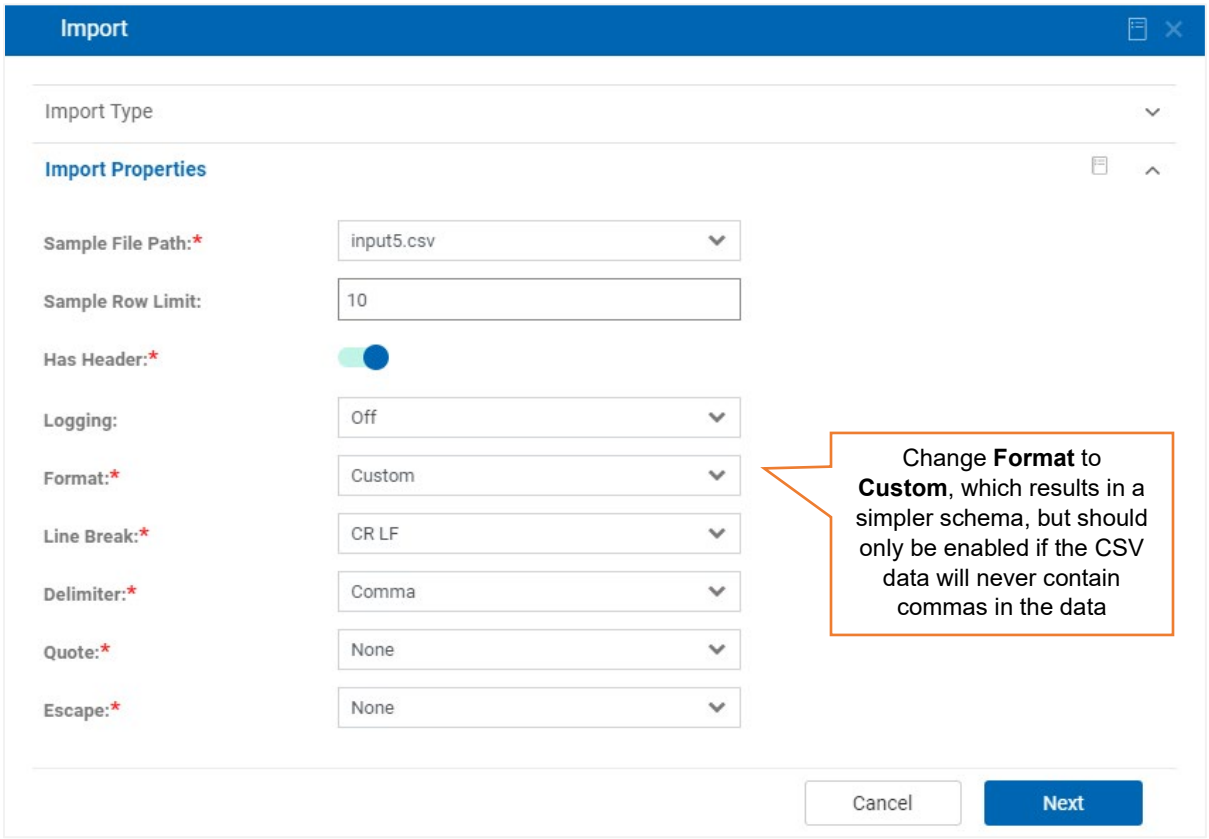
1. Click on **Schemas** from the bottom navigation bar and click on **Import** from the popup menu.



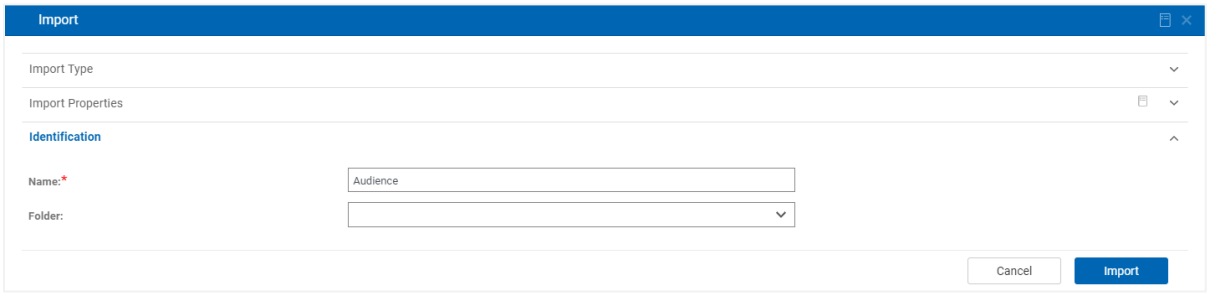
2. Select CSV as the **Import Type** and click **Next**.



3. Enter the **Import Properties** as shown below.

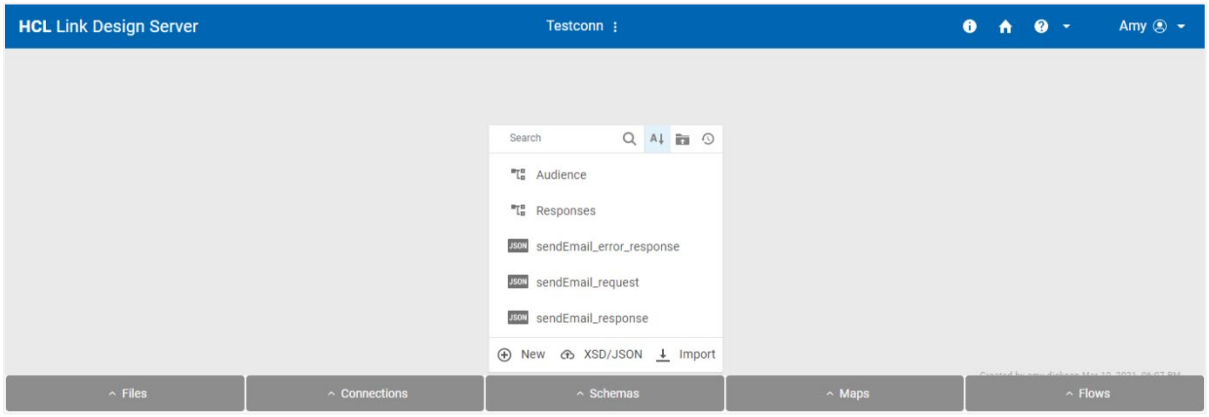


4. Enter a name for the schema and click **Import** to create the schema.



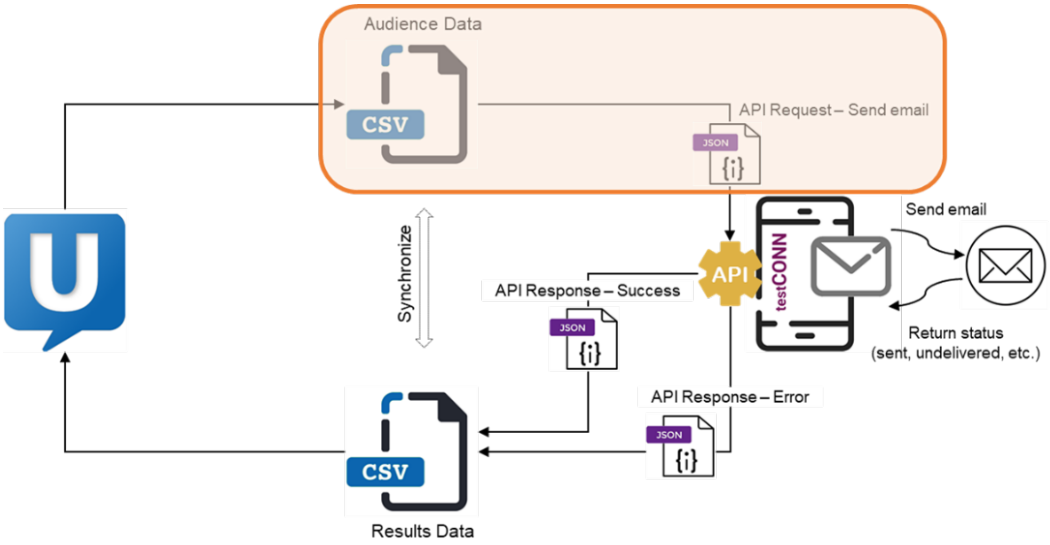
5. Repeat steps 1-4 for <`devkit`>`/testconn/schemas/response.csv`.

6. Click on **Schemas** from the bottom navigation bar to view your two new schemas.



Step 4 – Create Request Map

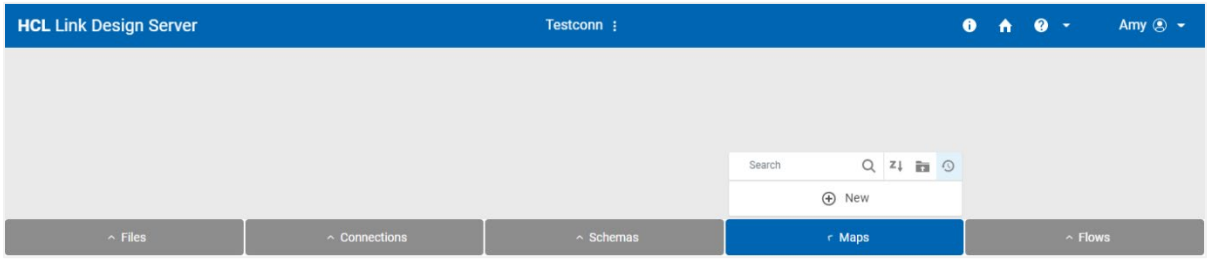
Now we have the input, request and response schemas defined in the project we can create a map that maps from the audience (input) CSV to the JSON array Send Email request we created earlier.



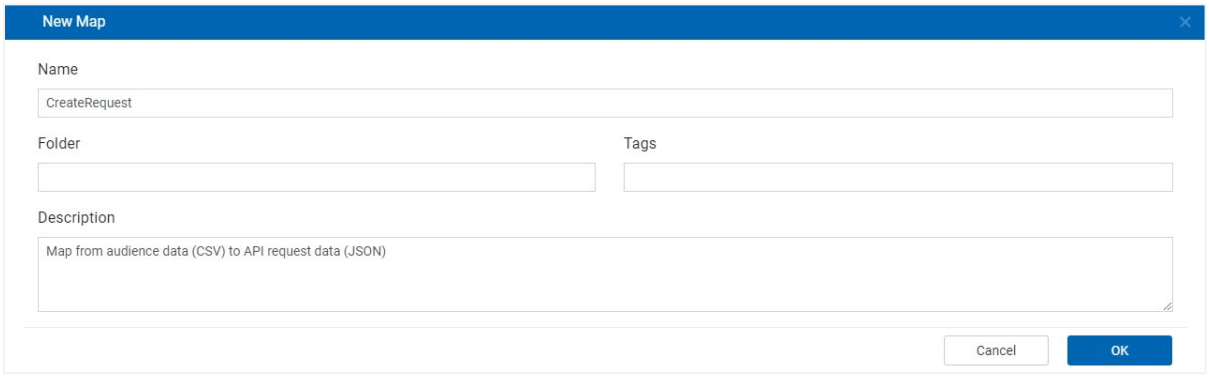
Note: Initially, we will create a map that uses a fixed set of fields for a specific case. At the end of the tutorial, we will replace this map with an auto-generated one. For the purposes of development, it is necessary to start with a specific use case, so that schemas and maps can be generated.

The map `CreateRequest` is an example of such a map. To create this map, follow these steps:

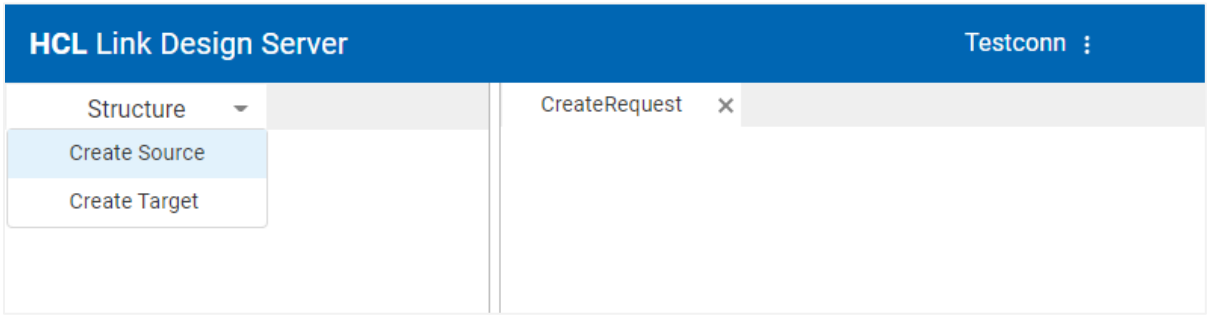
1. Click on **Maps** from the bottom navigation bar and click on **New** from the popup menu.



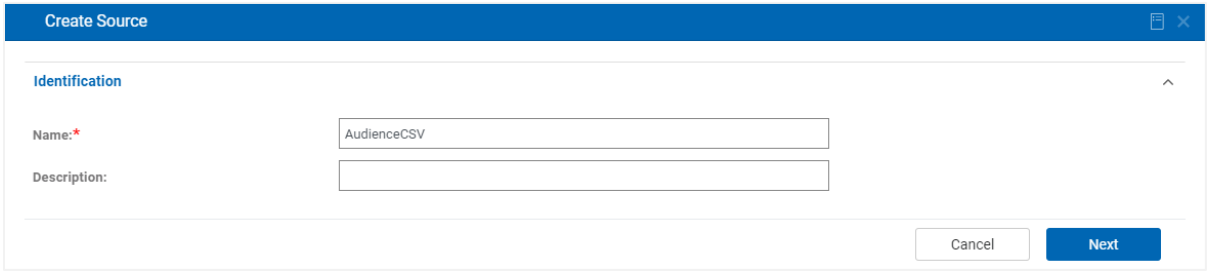
2. Enter a name and description (optional) and click **OK.**



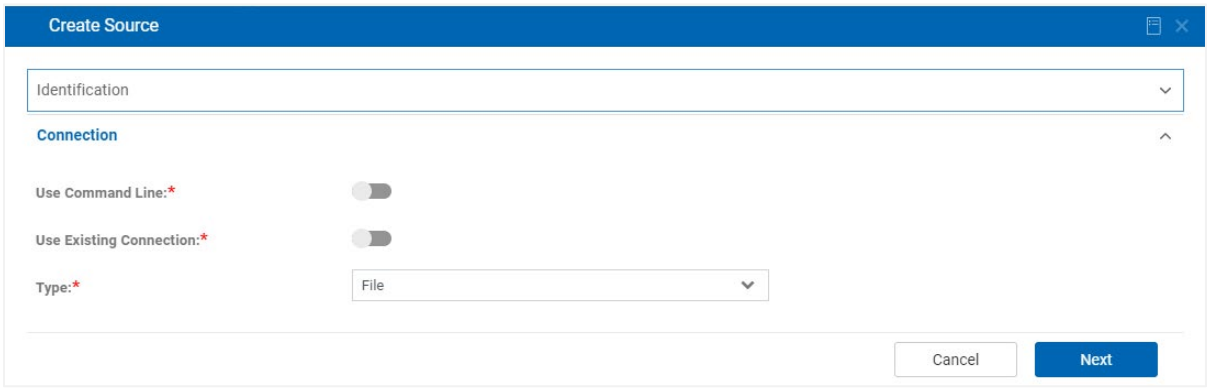
3. Click on the drop-down menu next to **Structure** in the upper left and select **Create Source**.



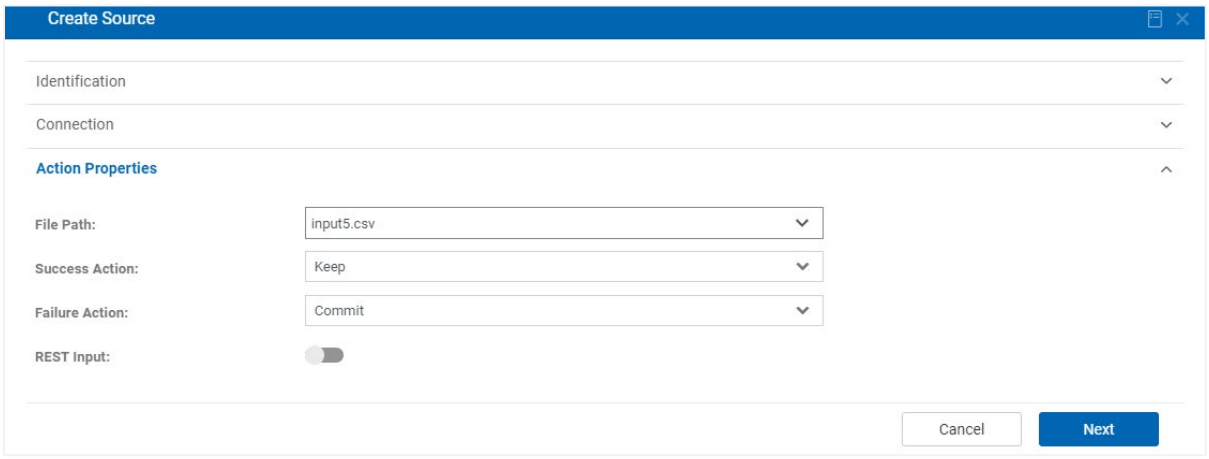
4. Provide a meaningful name for the source, such as AudienceCSV, and a description (optional) and click **Next**.



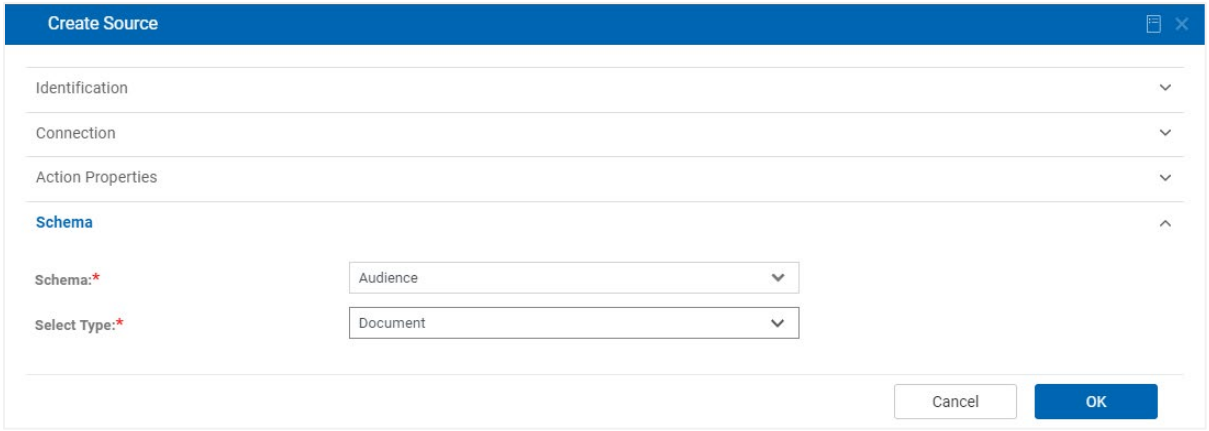
5. Disable **Use Existing Connection** and select **File** from the **Type** drop-down list. Click **Next.**



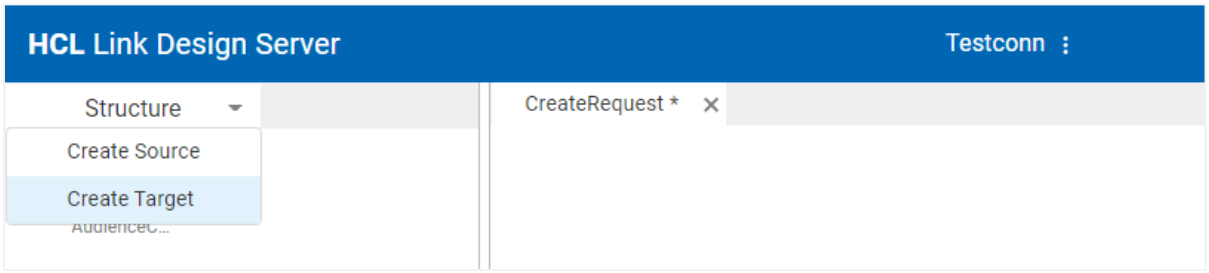
6. Select input5.csv for **File Path** and click **Next.**



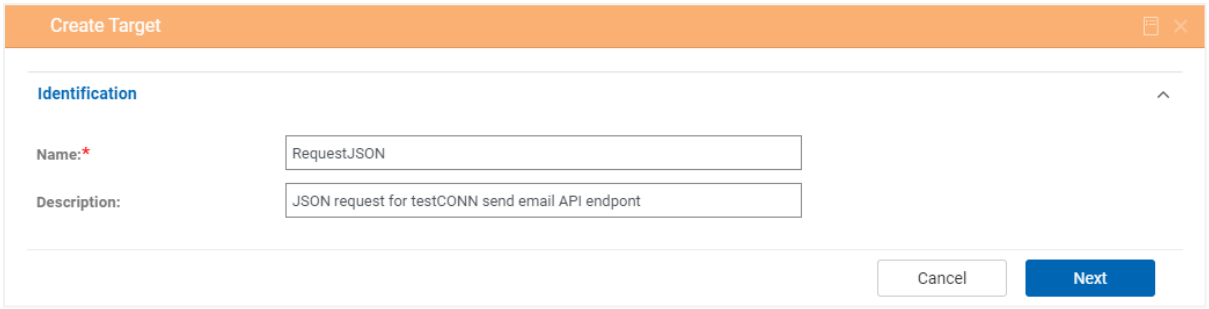
7. Select the CSV audience schema you created in the earlier step for **Schema** and select the **Document** for **Select Type**, then click **OK.**



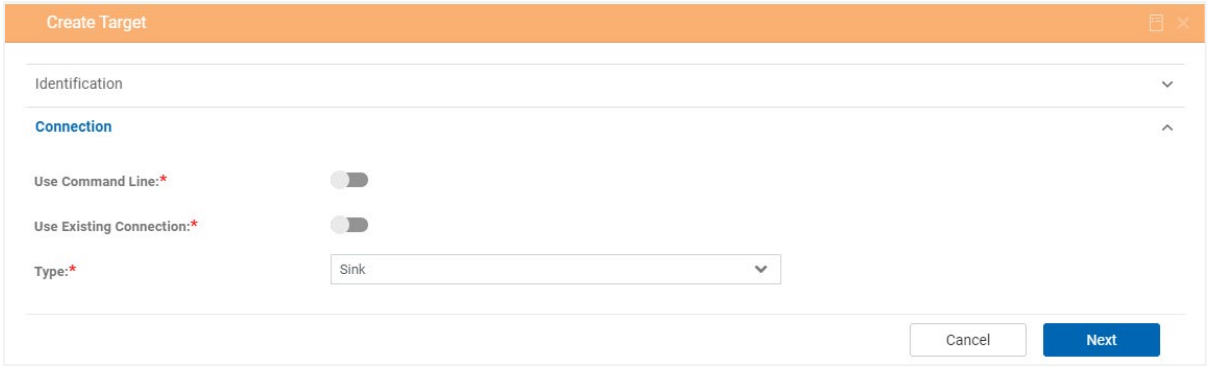
8. Click on the drop-down menu next to **Structure** in the upper left and select **Create Target**.



9. Provide a meaningful name for the target, such as RequestJSON, a description (optional) and click **Next**.

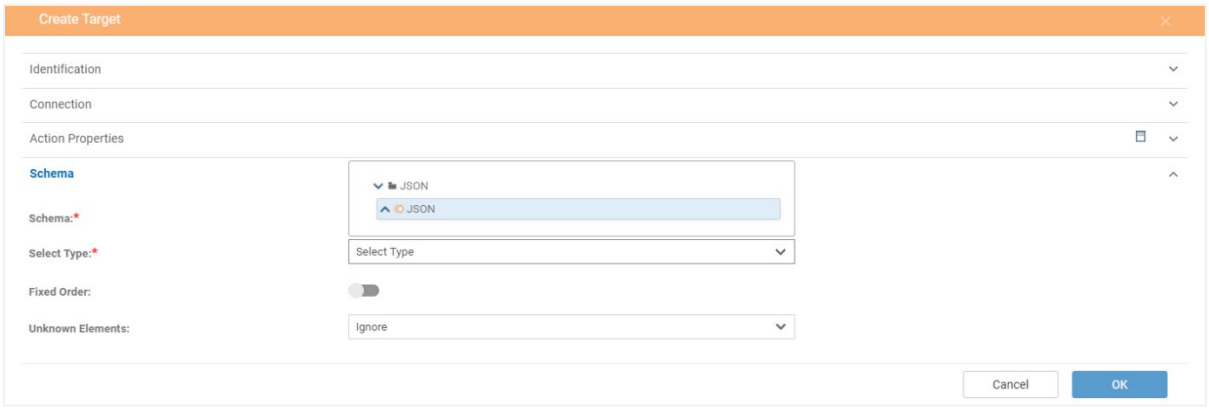


10. Disable **Use Existing Connection** and select **Sink** from the **Type** drop-down list. Click **Next.** Sink means that the output will not be written to a file or other resource, but the data sent to this output can be viewed in the Map Designer and will be passed to a downstream node in a flow.

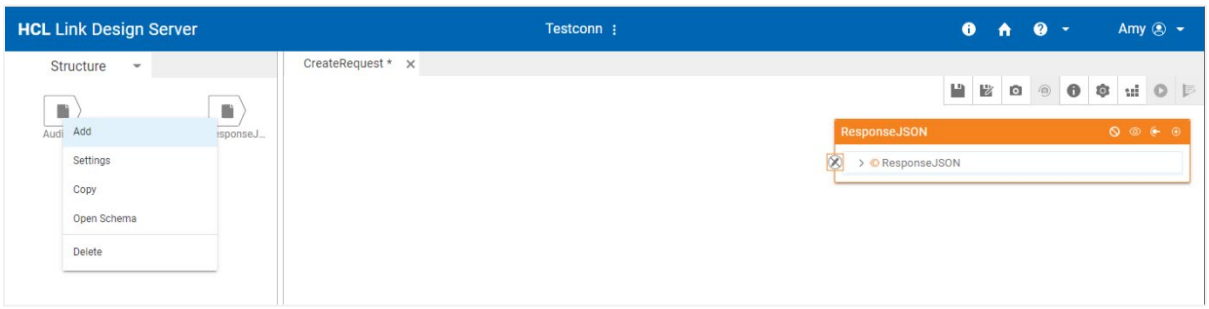


11. Click **Next**.

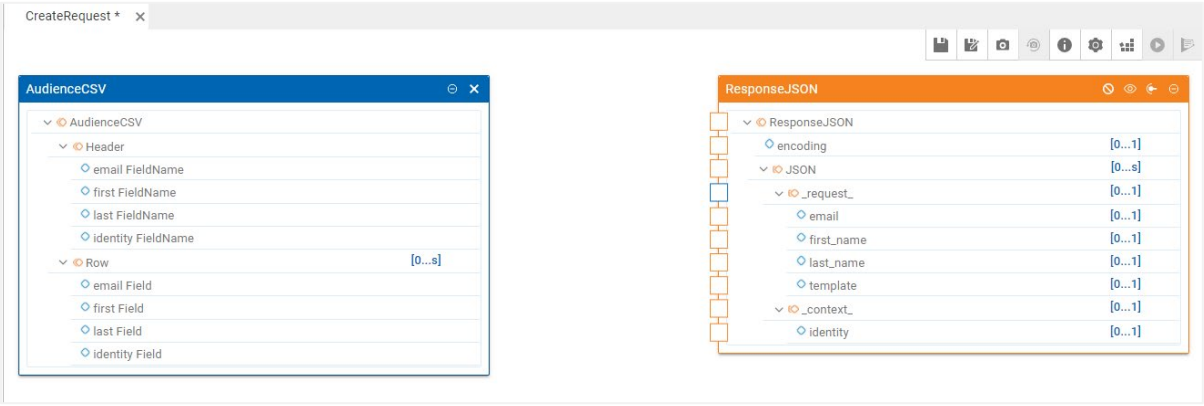
12. Select **sendEmail_request** for Schema and specify the schema properties, as shown below, and click **OK**.



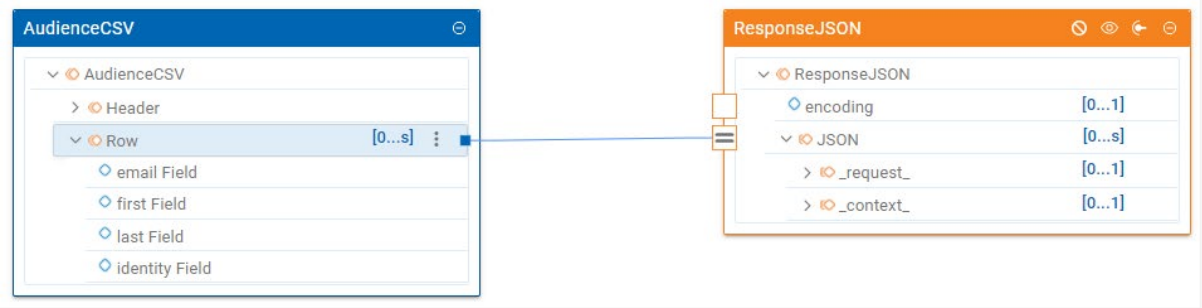
13. In the Structure panel, click over your AudienceCSV source and click on **Add** from the context menu. This adds AudienceCSV to the mapping workspace to the right.



14. Now you are ready to map! (If your sources and targets are collapsed, click on the arrows next to AudienceCSV and ResponseJSON to expand the schema structure.)



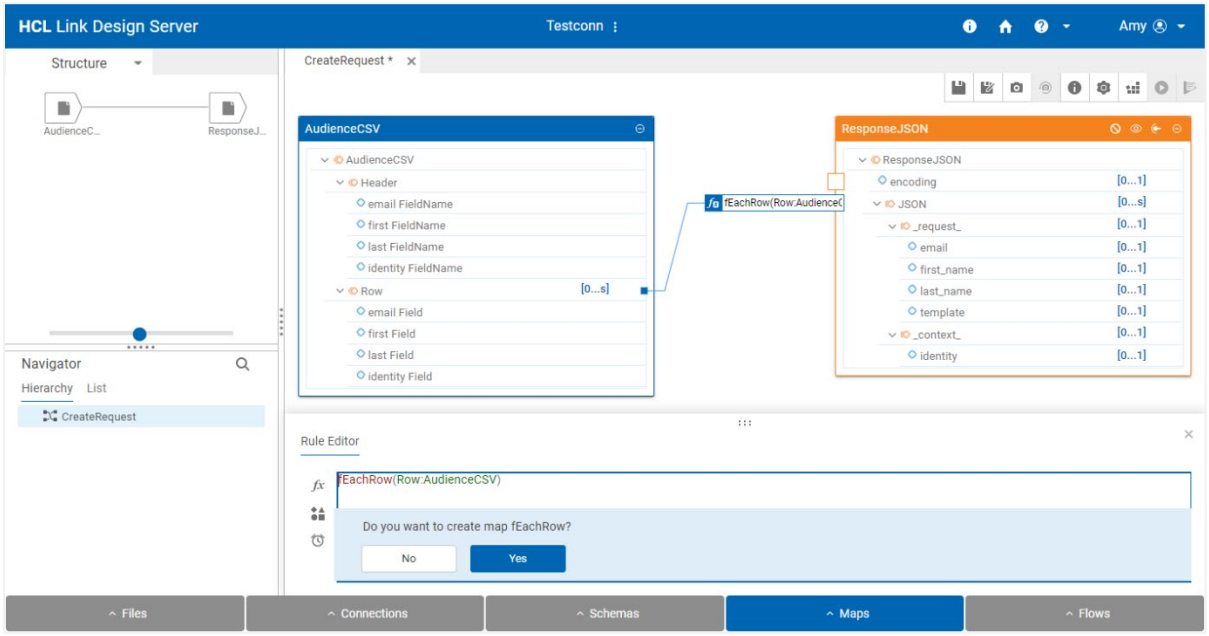
15. Map the input to the output by dragging the **Row** object in the AudienceCSV to the **JSON** object in the ResponseJSON.



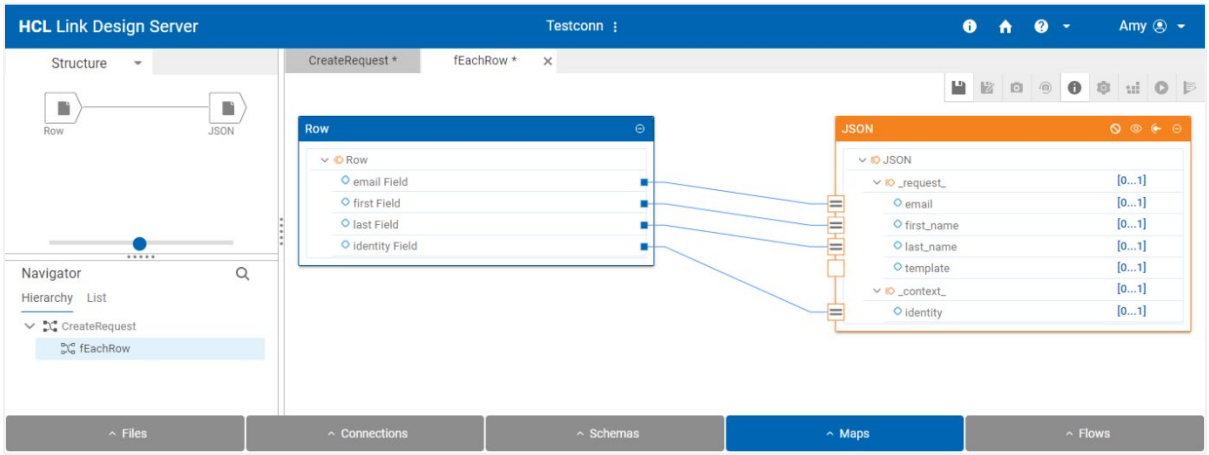
Notice that Row in AudienceCSV and JSON in ResponseJSON can occur a variable number of times (0…s means 0 to unlimited). To be able to handle these repeating elements, you need to create a sub-map, called a functional map, to map each Row in AudienceCVS to each JSON object in ResponseJSON.

Copyright © 2021 HCL Software

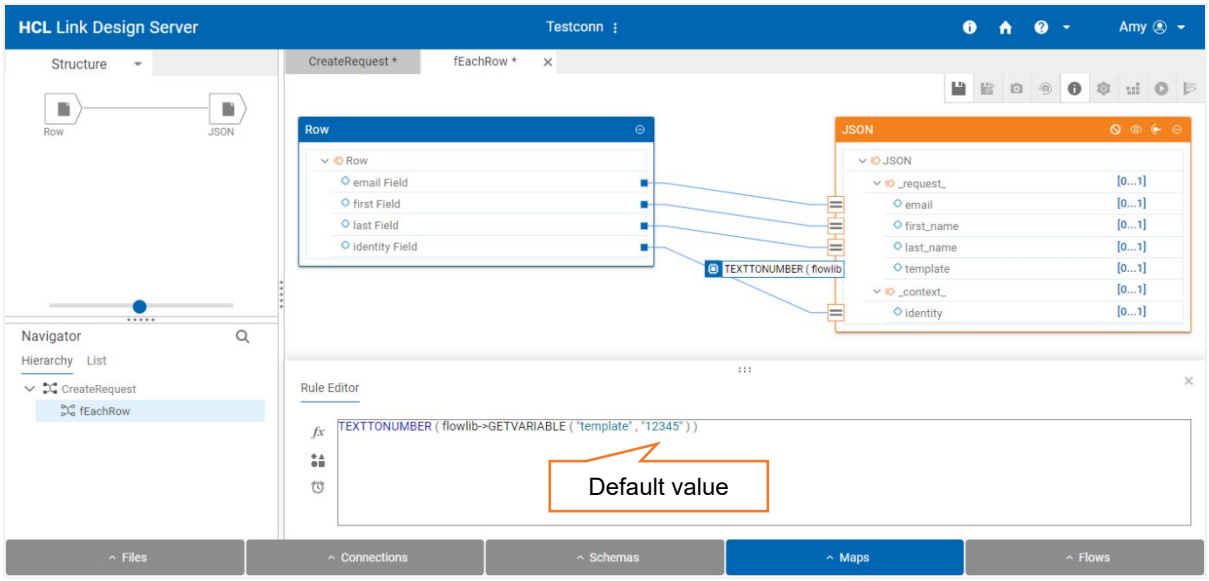
16. To create a functional map, click on the JSON rule to open the Rule Editor. Add **fEachRow()** around the Row object that is already present in the rule. Press **Enter.** A dialog appears at the bottom of the window asking if you want to create the map fEachRow. Click **Yes.**



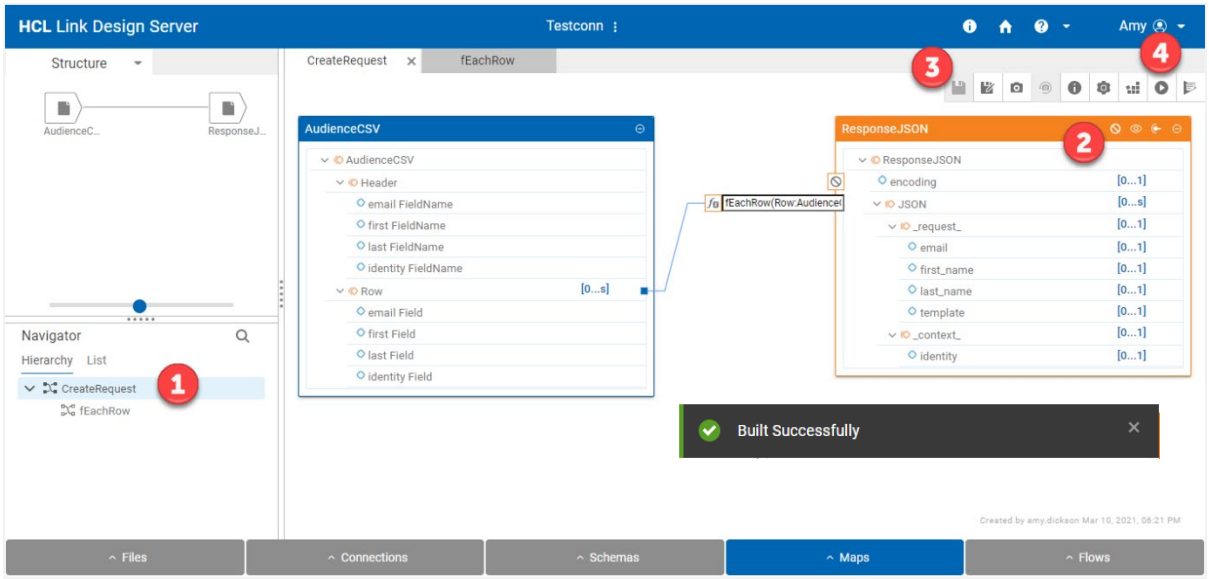
17. The new functional map, **fEachRow**, will be displayed. Drag the inputs from Row to the corresponding outputs in JSON.



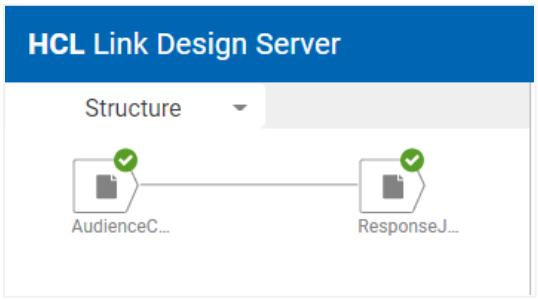
18. The value for template will be coming from properties that the end user specifies when configuring the connector in Campaign or Journey. These are passed to the map as **flow variables**. The rule for template will use the **flowlib->GETVARIABLE function** to get the value of these variables/properties. The **TEXTTONUMBER function** converts the flow variable (which is text) to the template ID (which is a numeric value).



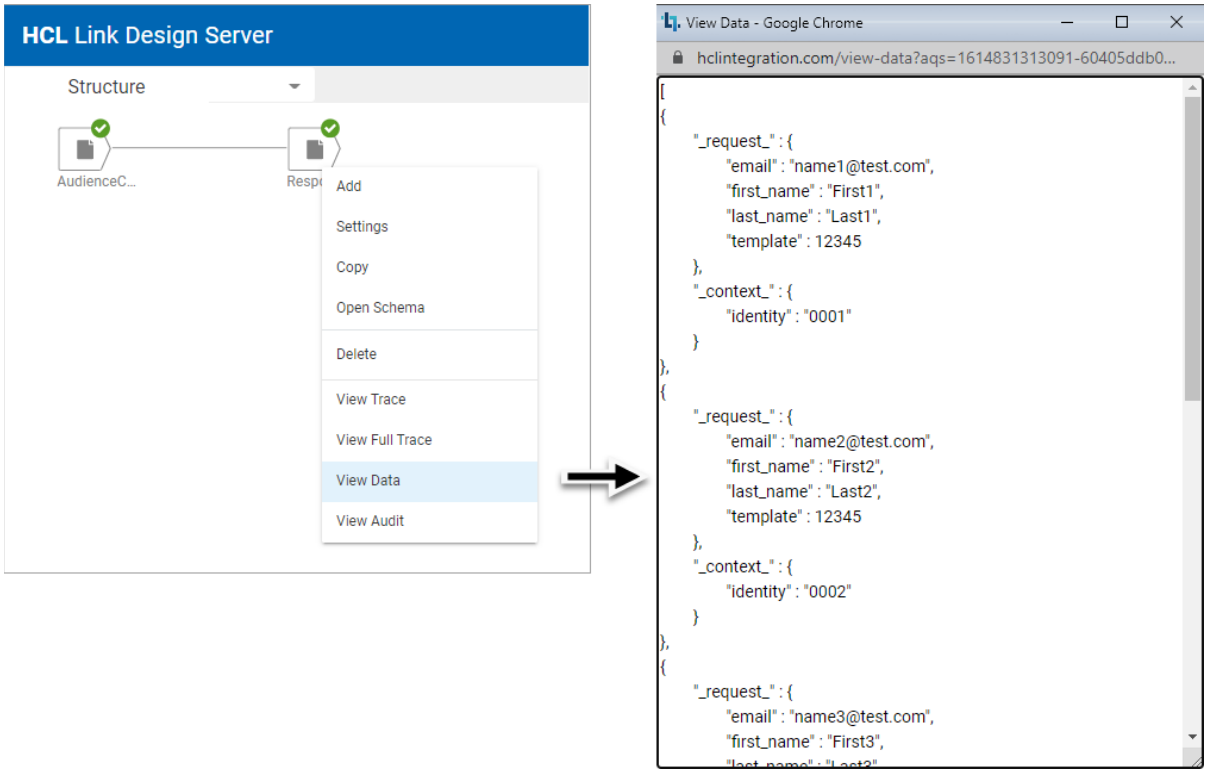
19. Return to the main CreateRequest map by clicking on its name in the Navigator. Cllick **Insert NONE** on ResponseJSON and then click **Save** and **Build** on the toolbar. If there are no errors in the map, a message will indicate that the build was successful.



20. The map can now be run directly in the Map Designer by clicking the Run icon. After running, the Structure panel shows the state of the inputs and outputs.

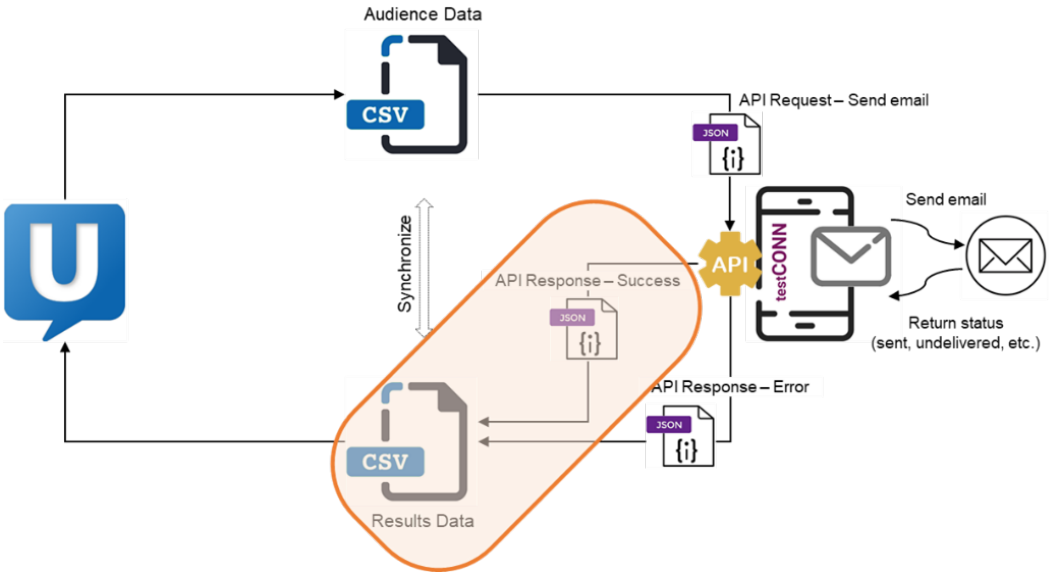


21. The input data, as well as the data produced by the output can be viewed by right-clicking on the appropriate node in the Structure panel and selecting View Data. This data will be passed to the REST Client node in a flow in a subsequent step.



Step 5 – Create Response Map

Now we need to create a map that maps the JSON formatted success responses returned by testCONN's API to the response CSV file expected by Campaign and Journey, as shown in the diagram.



Follow these steps to create a map to map the success JSON array to the results CSV (refer to the steps in the Step 4 – Create Request Map, as needed):

1. Click on **Maps** from the bottom navigation bar and click on **New** from the popup menu.
2. Enter **ProcessResponse** as the name and a description (optional) and click **OK.**
3. Click on the drop-down menu next to **Structure** in the upper left and select **Create Source**.
4. Provide a meaningful name for the source, such as ResponseJSON, and a description (optional) and click **Next**.
5. Disable **Use Existing Connection** and select **File** from the **Type** drop-down list. Click **Next.**
6. Select **--- Update File ---** for **File Path**, upload the sendEmail_response.json file, click **OK** and then click **Next.**
7. Select the sendEmail_response JSON schema you created in the earlier step for **Schema** and select **JSON** for **Select Type**, then click **OK.**
8. Click on the drop-down menu next to **Structure** in the upper left and select **Create Target**.
9. Provide a meaningful name for the target, such as ResultsCSV, a description (optional) and click **Next**.

Copyright © 2021 HCL Software

10. Disable **Use Existing Connection** and select **Sink** from the **Type** drop-down list. Click **Next** and click **Next** again**.**

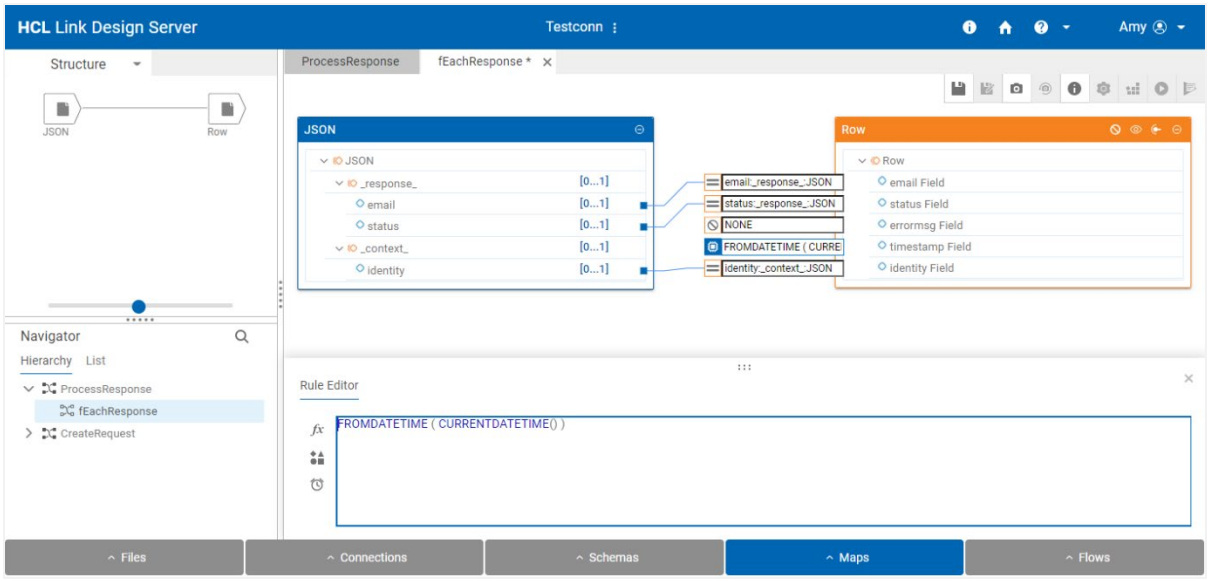
11. Select the **Responses** schema and the **Document** type, then click **OK**.

12. In the Structure panel, click over your ResponseJSON source and click on **Add** from the context menu. This adds ResponseJSON to the mapping workspace to the right.

13. Map the input to the output by dragging the **JSON** object in ResponseJSON to the **Row** object in ResultsCSV.

14. To create a functional map to map each incoming JSON array to a corresponding row in the CSV file, click on the JSON rule to open the Rule Editor. Add **fEachResponse()** around the Row object that is already present in the rule. Press **Enter.** A dialog appears at the bottom of the window asking if you want to create the map fEachResponse. Click **Yes.**

15. The new functional map, **fEachResponse**, will be displayed. Drag the inputs from Row to the corresponding outputs in JSON, type in the rule shown below for timestamp (using the FROMDATETIME and CURRENTDATETIME functions) and click on the Insert NONE icon in the Row title bar to map errormsg Field.



16. Return to the main ProcessResponse map by clicking on its name in the Navigator. Cllick **Insert NONE** on ResultsCSV and then click **Save** and **Build** on the toolbar. If there are no errors in the map, a message will indicate that the build was successful.

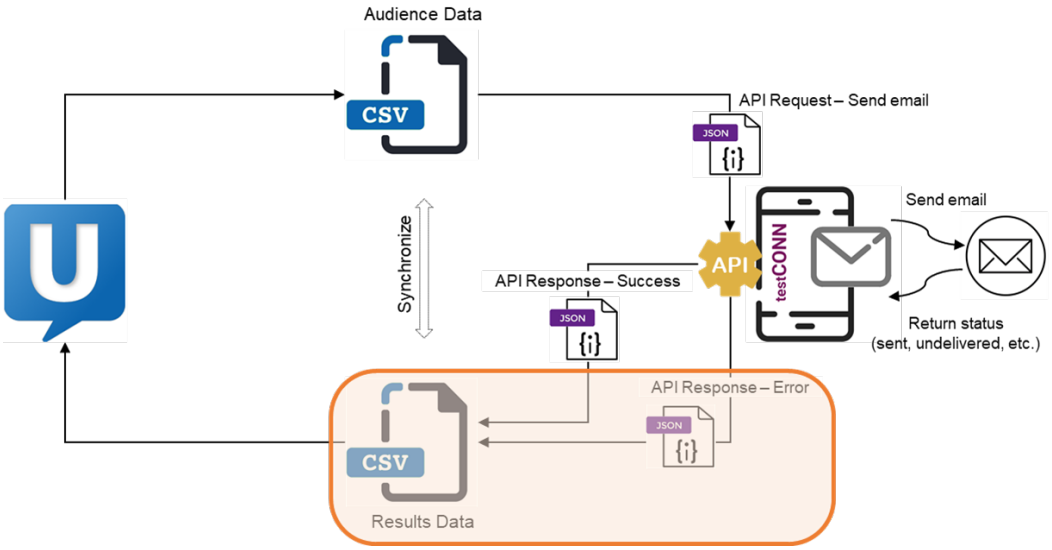
17. The map can now be run directly in the Map Designer by clicking the Run icon. After running, the Structure panel shows the state of the inputs and outputs.

18. The input data, as well as the data produced by the output can be viewed by right-clicking on the appropriate node in the Structure panel and selecting View Data.

Copyright © 2021 HCL Software

Step 6 – Create Error Response Map

Now we need to create a map that maps the JSON formatted success responses returned by testCONN's API to the response CSV file expected by Campaign and Journey, as shown in the diagram.



Follow these steps to create a map to map the JSON array of error responses to the results CSV (refer to the steps in the Step 4 – Create Request Map, as needed):

1. Click on **Maps** from the bottom navigation bar and click on **New** from the popup menu.
2. Enter **ProcessErrorResponse** as the name and a description (optional) and click **OK.**
3. Click on the drop-down menu next to **Structure** in the upper left and select **Create Source**.
4. Provide a meaningful name for the source, such as ErrorResponseJSON, and a description (optional) and click **Next**.
5. Disable **Use Existing Connection** and select **File** from the **Type** drop-down list. Click **Next.**
6. Select **--- Update File ---** for **File Path**, upload the sendEmail_error_response.json file, click **OK** and then click **Next.**
7. Select the sendEmail_error_response JSON schema you created in the earlier step for **Schema** and select **JSON** for **Select Type**, then click **OK.**
8. Click on the drop-down menu next to **Structure** in the upper left and select **Create Target**.
9. Provide a meaningful name for the target, such as ResultsCSV, a description (optional) and click **Next**.
10. Disable **Use Existing Connection** and select **Sink** from the **Type** drop-down list. Click **Next** and click **Next** again**.**

Copyright © 2021 HCL Software

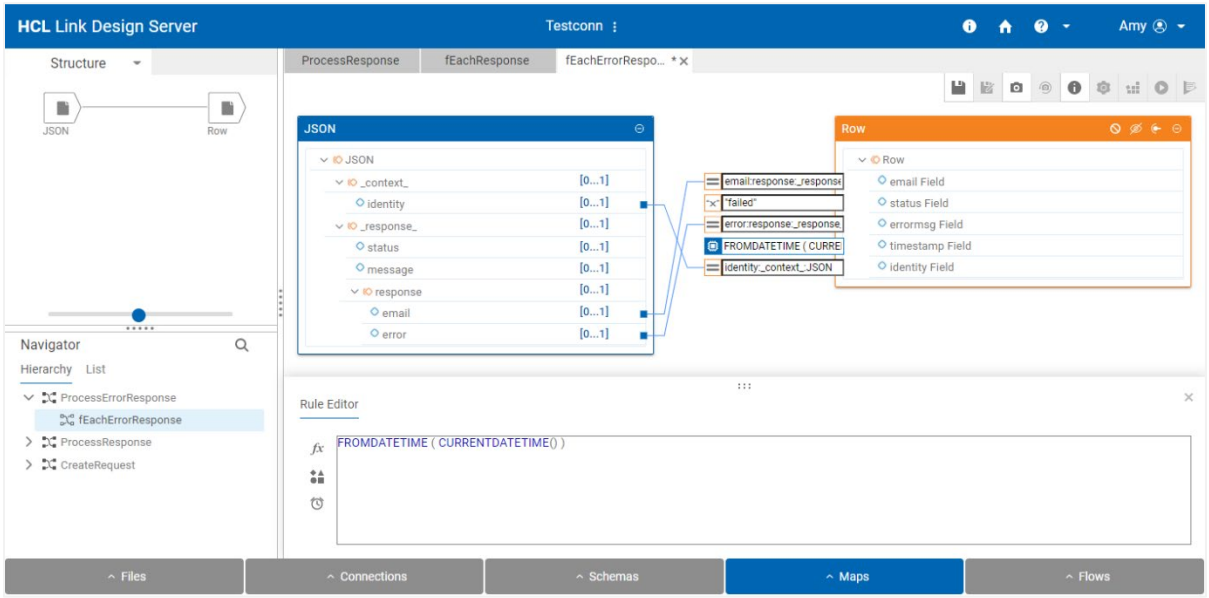
11. Select the **Responses** schema and the **Document** type, then click **OK**.

12. In the Structure panel, click over your ErrorResponseJSON source and click on **Add** from the context menu. This adds ResponseJSON to the mapping workspace to the right.

13. Map the input to the output by dragging the **JSON** object in ErrorResponseJSON to the **Row** object in ResultsCSV.

14. To create a functional map to map each incoming JSON array to a corresponding row in the CSV file, click on the JSON rule to open the Rule Editor. Add **fEachErrorResponse()** around the Row object that is already present in the rule. Press **Enter.** A dialog appears at the bottom of the window asking if you want to create the map fEachResponse. Click **Yes.**

15. The new functional map, **fEachErrorResponse**, will be displayed. Drag the inputs from Row to the corresponding outputs in JSON, type in the rule shown below for timestamp (using the FROMDATETIME and CURRENTDATETIME functions) and click on the Insert NONE icon in the Row title bar to map errormsg Field.



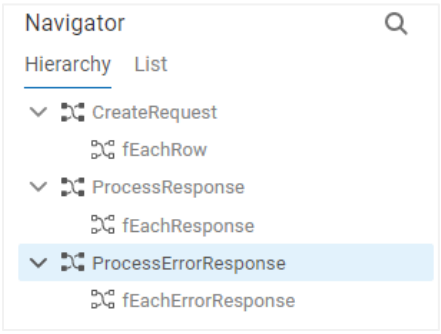
16. Return to the main ProcessErrorResponse map by clicking on its name in the Navigator. Click **Insert NONE** on ResultsCSV and then click **Save** and **Build** on the toolbar. If there are no errors in the map, a message will indicate that the build was successful.

17. The map can now be run directly in the Map Designer by clicking the Run icon. After running, the Structure panel shows the state of the inputs and outputs.

18. The input data, as well as the data produced by the output can be viewed by right-clicking on the appropriate node in the Structure panel and selecting View Data.

Copyright © 2021 HCL Software

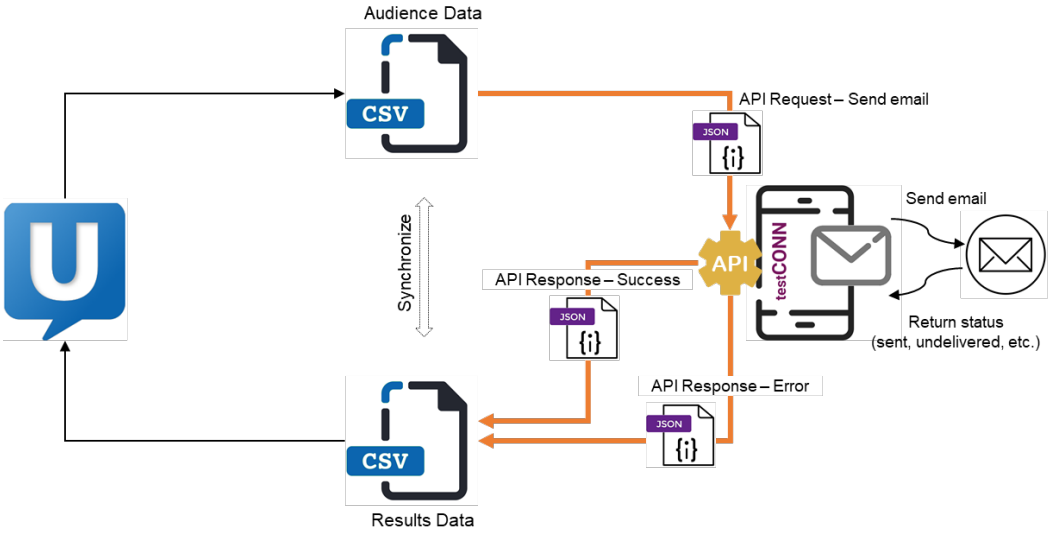
Now we have created the necessary three maps, each with its own functional map:



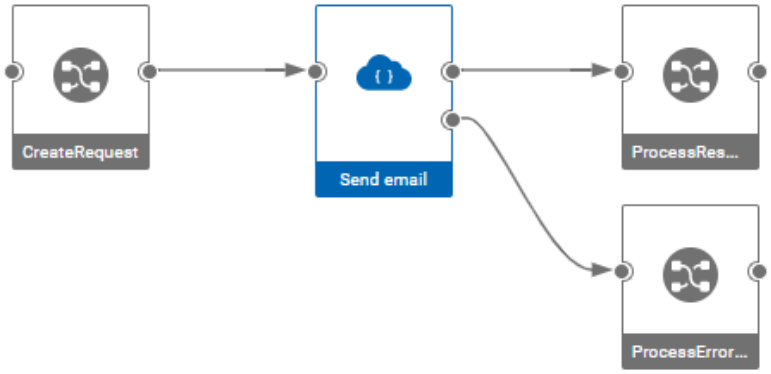
Step 7 – Create First Flow

The next step is to create a simple flow that will leverage the other artifacts created so far in this tutorial – services, endpoints and maps – to build out the logic for the connector, including:

- Map incoming CSV to a JSON array
- Invoke the Send Email API for each object in the array
- Map the JSON arrays for successful and error responses to a CSV result

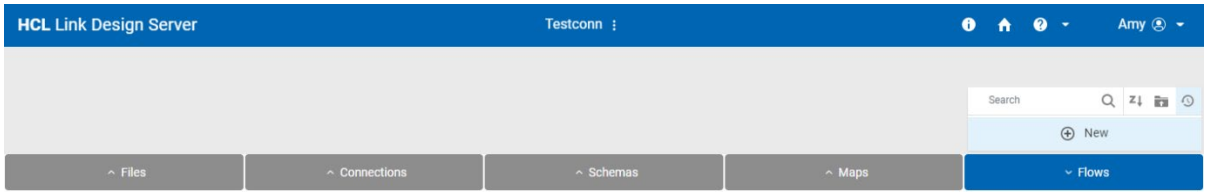


The final flow is provided in the project as SendEmails which looks like this:

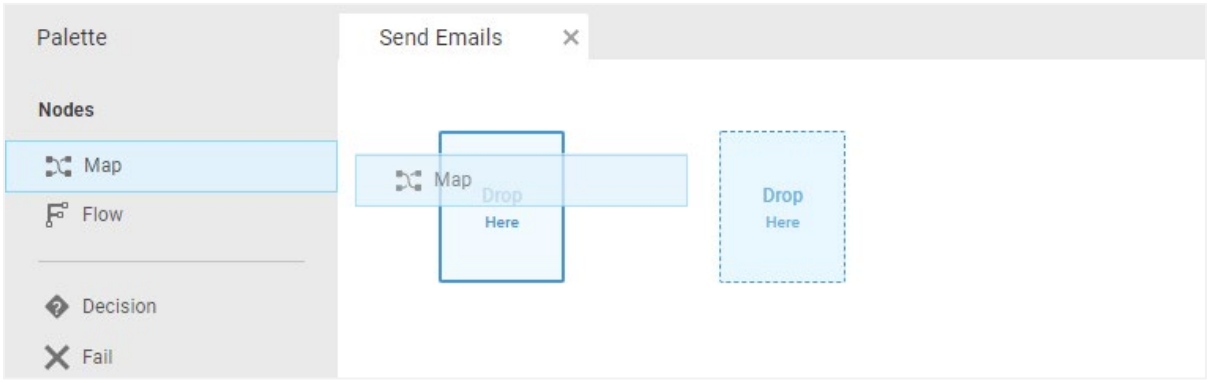


To create this flow:

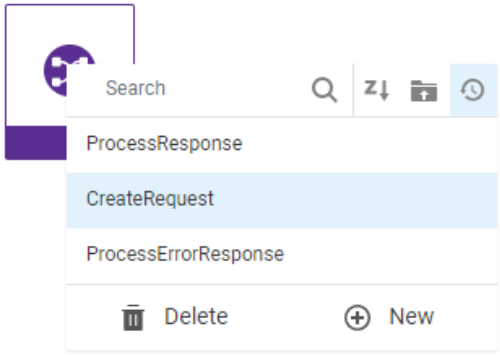
1. Click on **Flows** from the bottom navigation bar and click on **New** from the popup menu.



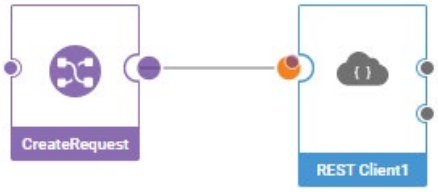
2. Drag a **Map node** from the Palette to the canvas:



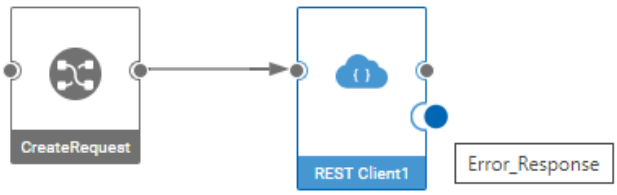
3. Right-click on the new map node and select the **CreateRequest** map from the list.



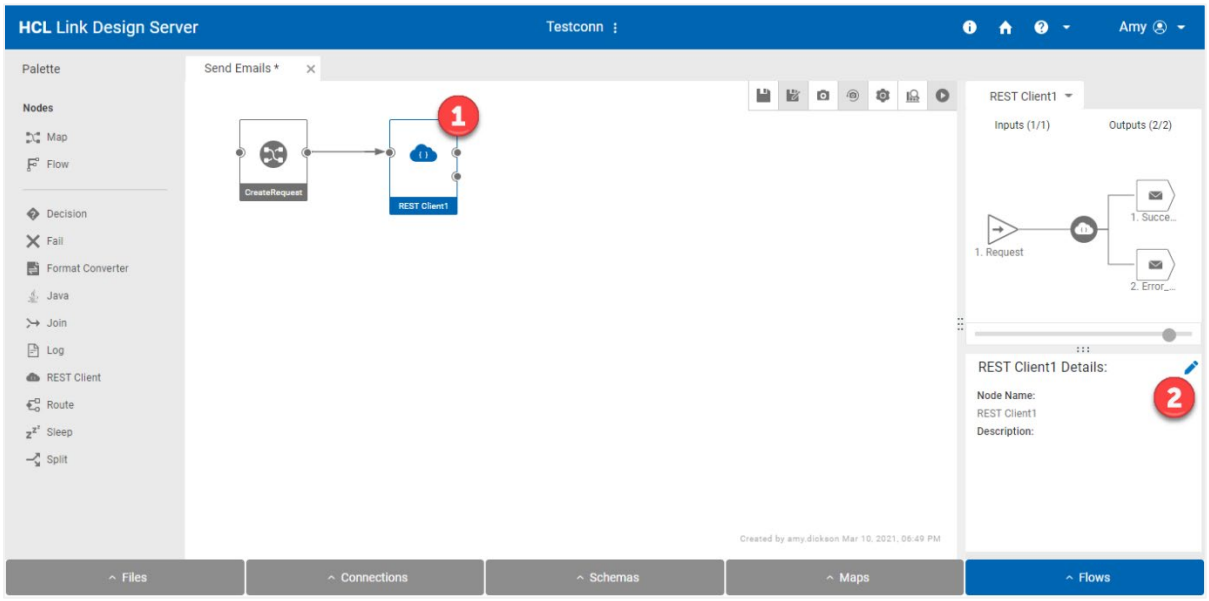
4. Drag a **REST Client node** from the palette, and then connect the output of the CreateRequest map to the input of the REST Client node to connect them.



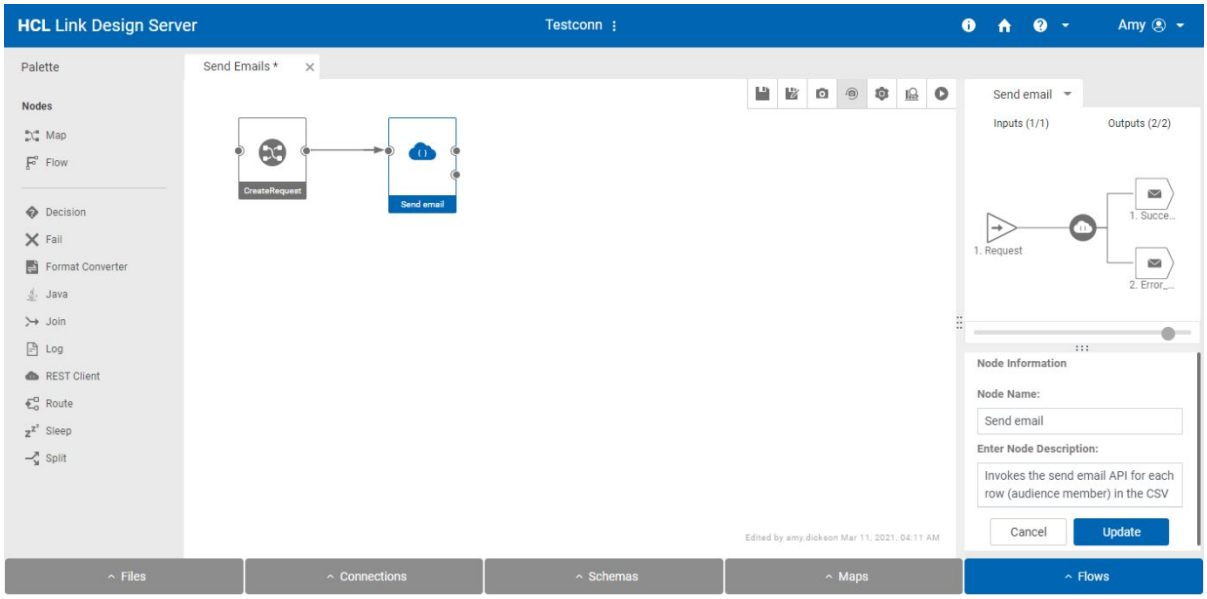
Note that the Rest Client node has two output terminals. The top one is for the success response and the bottom one is for the error response. You can see information on each output by hovering over the terminal, as shown below.



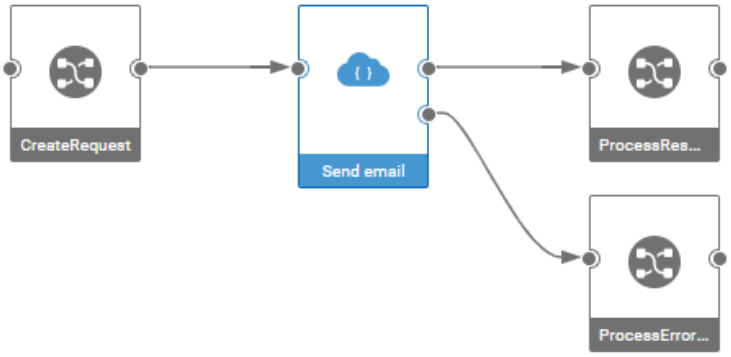
5. Click on the **Rest Client1** node and then click on the edit icon in the information panel.



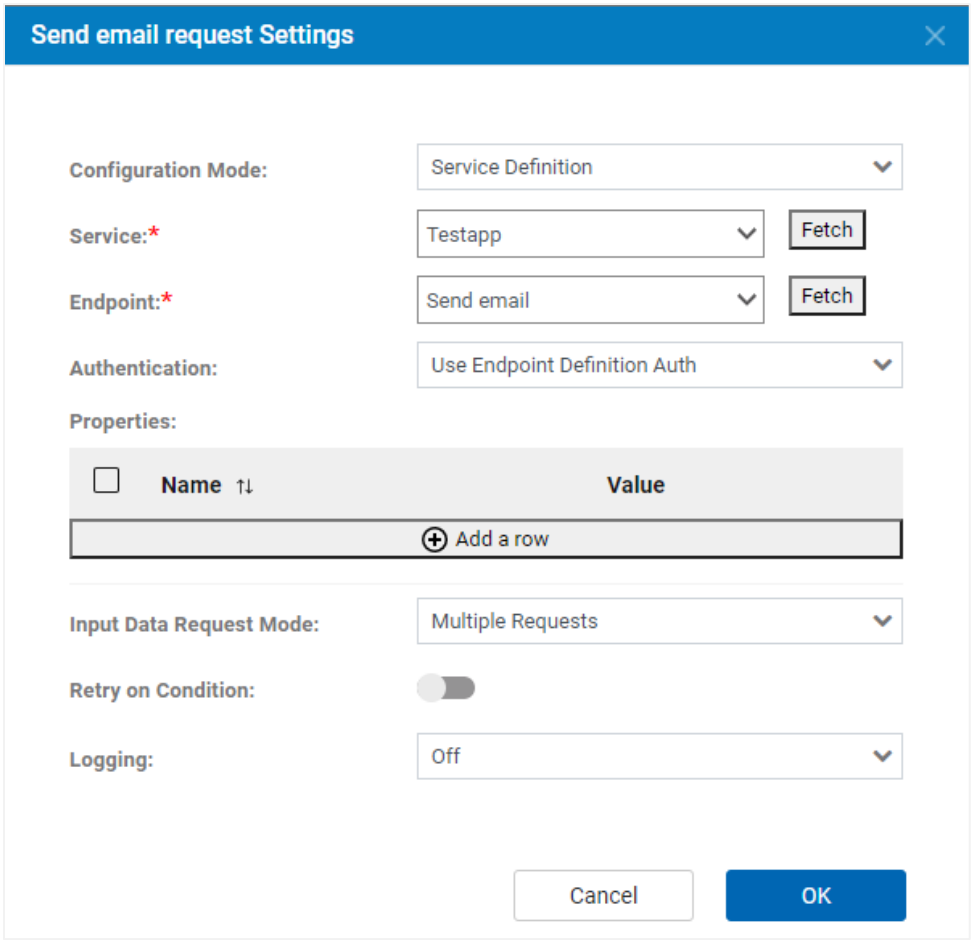
6. Enter a meaningful name and description in the information panel in the lower right for the REST endpoint node that will send the email request and click **Update**.



7. Repeat steps 2 and 3 to add the **ProcessResponses** and **ProcessErrorResponse** maps to the flow. Then connect the two maps to the appropriate terminals (top is success response, bottom is error response) from the Send email REST node.

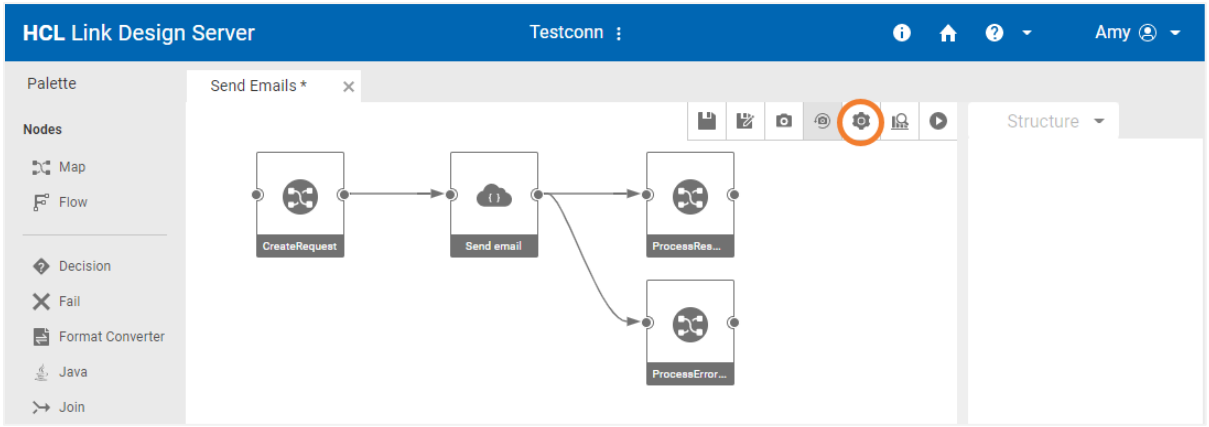


8. Now we need to configure the **Send email** REST Client node. Right-click on the node and select **Settings**. Configure the settings as shown below and described on the following page, then click **OK.**

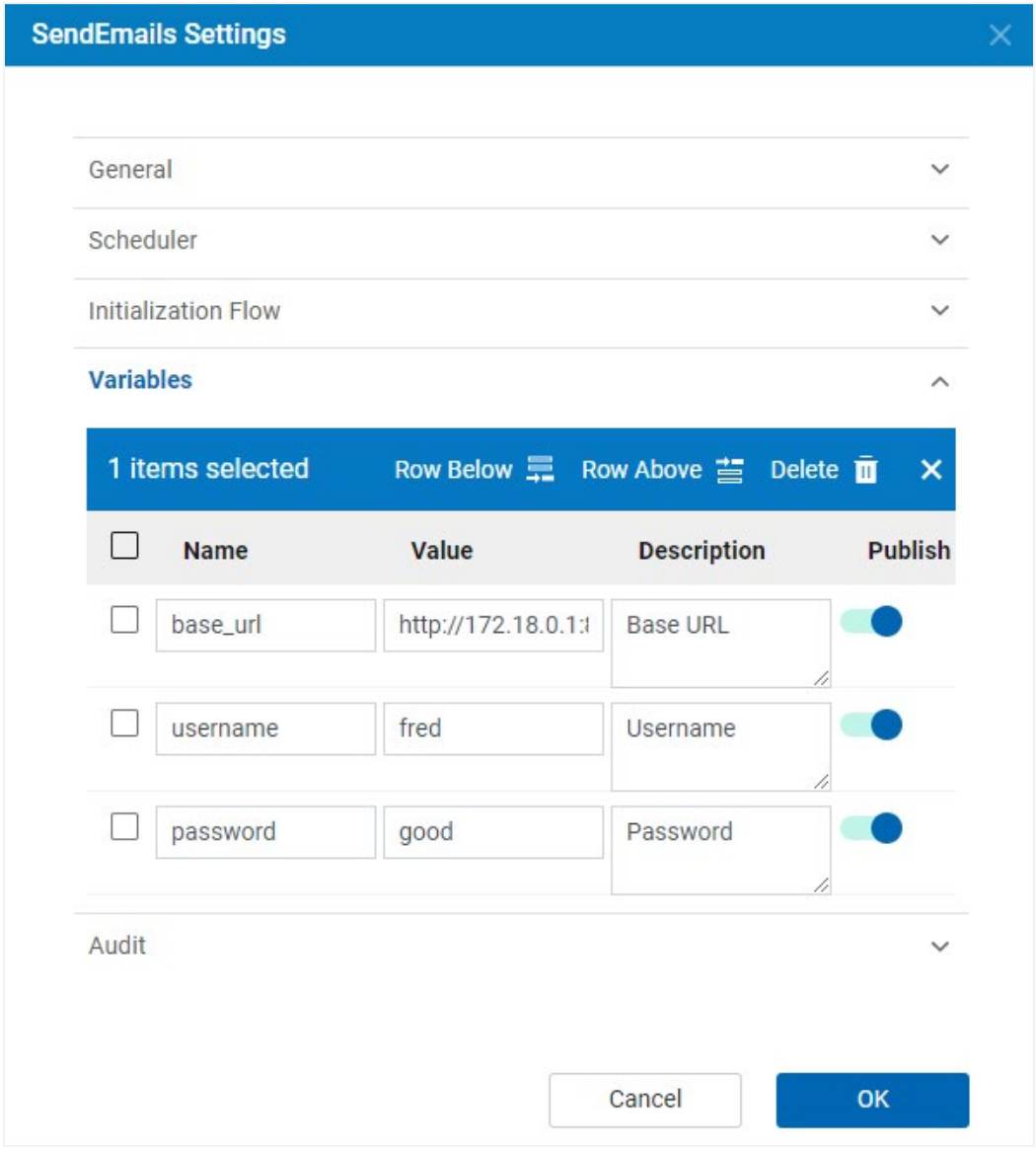


- **Configuration Mode**: Select **Service Definition**. This instructs the node to get the API information from a service created in the REST Service Editor.
- **Service:** Click on **Fetch** to get a list of services and select **Testapp**.
- **Endpoint:** Click on **Fetch** to get a list of the endpoints defined in service **Testapp** and select **Send email**.
- **Authentication:** Select **Use Endpoint Definition Auth**. This instructs the node to use the authentication setting defined in the service (this was defined earlier as basic authentication).
- **Input Data Request Mode:** Select **Multiple Requests**. In this mode the node expects an array of JSON objects, which contain the request data to send to the Send Email API.
- **Logging**: Optionally, specify Logging properties. The log path is a file on the design server.

9. Click on the **Settings** icon in the toolbar to add flow variables to the flow properties.

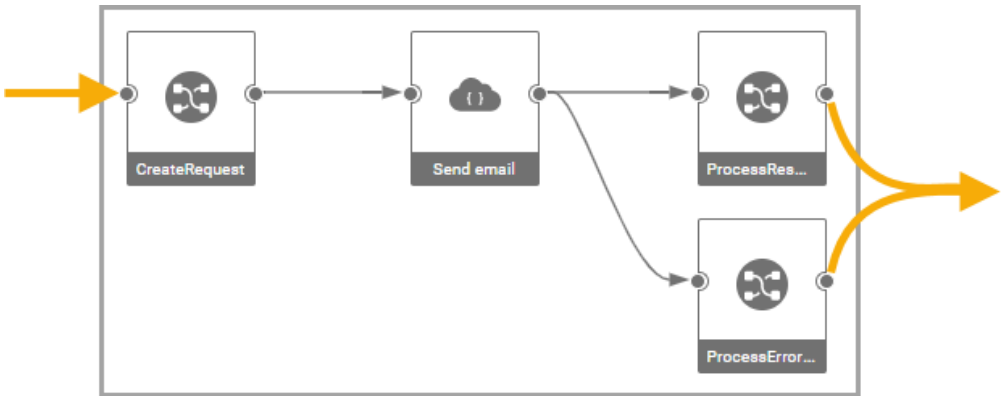


10. Configure the flow properties, as shown below. The value for each property is a default value that can be overridden when the flow is run. Enabling **Publish** will make these variables be documented in the Swagger user interface when the flow is deployed.

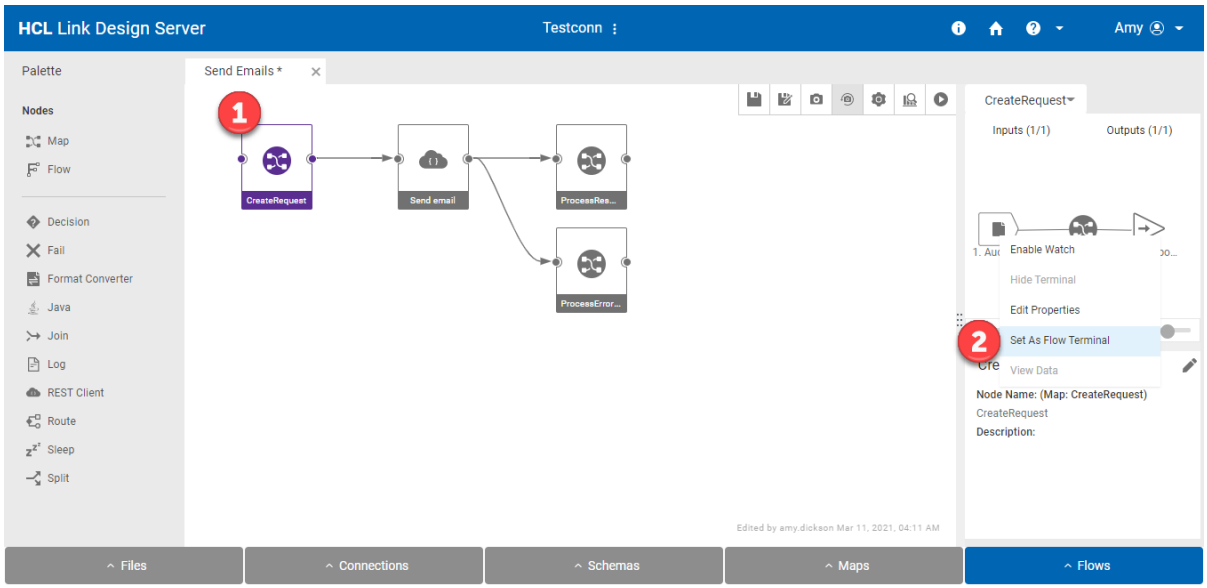


Finally, we need to specify what are the inputs and outputs for the flow. When the flow is invoked via a REST API call, this determines where the HTTP request data is sent, and which output terminal(s) are providing the HTTP response data. We want the request to be passed to

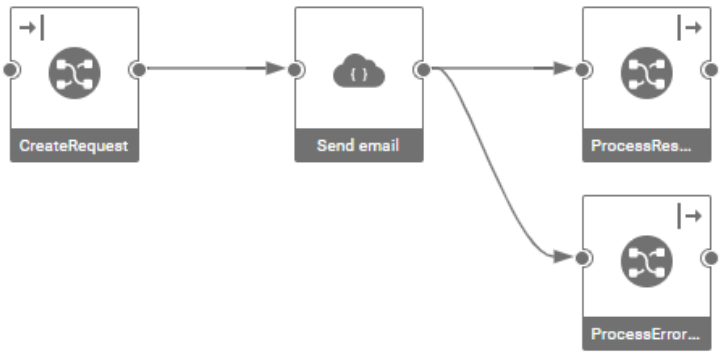
the input terminal of the **CreateRequest** map and want the outputs of the **ProcessResponse** and **ProcessErrorResponse** maps to be returned (we will combine these two outputs later).



11. Click on the **CreateRequest** node in the flow diagram, and then in the structure diagram right-click on the input terminal and select **Set as Flow Terminal**.



12. Repeat step 12 for the two response maps, selecting **Set as Flow Terminal** for the **output** of each in the Structure panel.



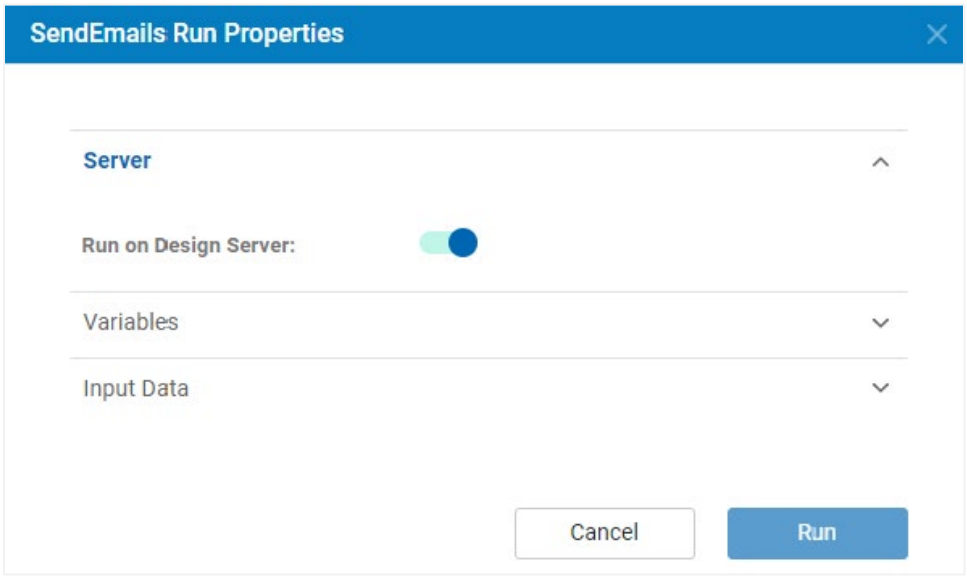
13. Click on **Save** on the toolbar to save the flow.

Step 8 – Running the Flow

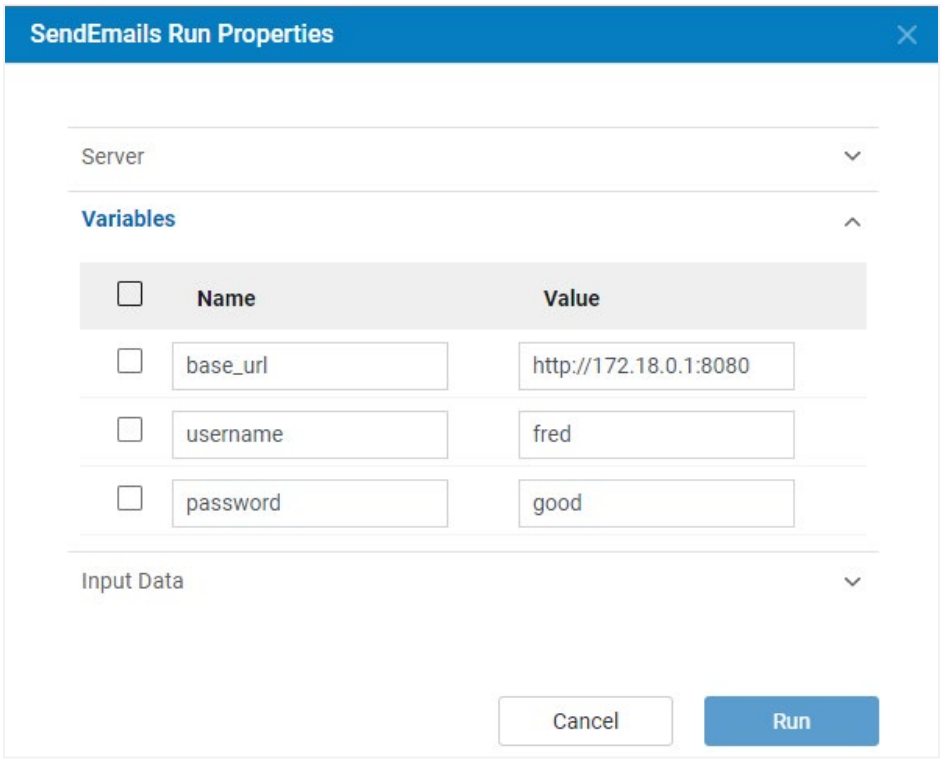
Having created the **Send Emails** flow, it can be run directly in the designer. To run it, follow these steps:

1. Click on the **Run** icon on the toolbar.

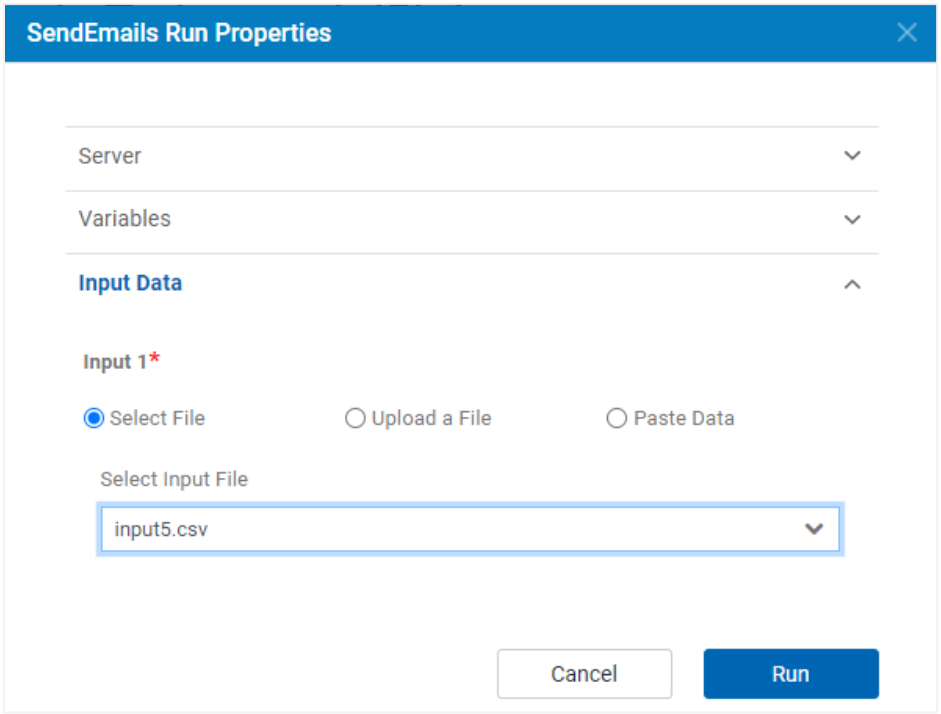
2. Under **Server**, select **Run on Design Server**. This will run the flow directly in the designer environment.



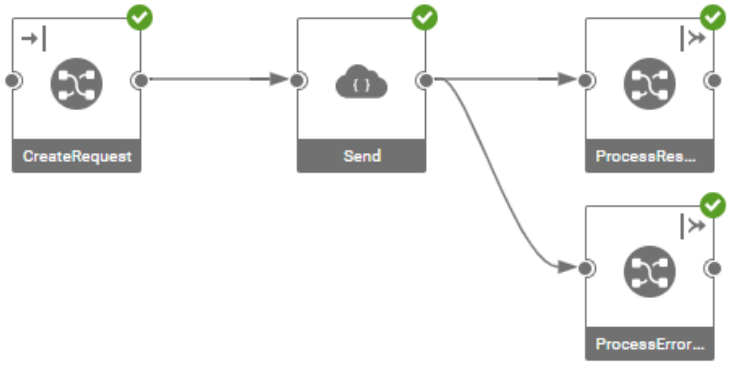
3. Expand **Variables** and specify the values for the variables that will be used when the flow is run. It will default to the values defined earlier in the flow settings.



4. Expand **Input Data** and select the input5.csv file added to the project earlier.



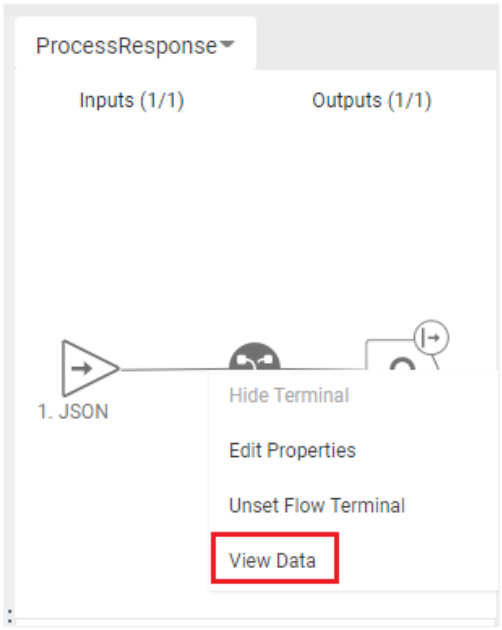
5. Click **Run** to run the flow. After a few seconds, the run will complete and the status of each flow node will have a graphical indicator showing whether the node succeeded, failed, or was not run.



6. To view more information about each node run, right click on the node and select **View Log**.

7. To view the data that was sent down a link, right click on the link and select **View Data**.

8. To view the data that was returned from the flow, click on the final nodes (ProcessResponse and ProcessError) and from the structure panel, right click the output terminal:



This will open a new browser window displaying the output data:

```
name1@test.com,undelivered,,2020-11-12T11:15:33,0001
name2@test.com,undelivered,,2020-11-12T11:15:33,0002
name3@test.com,sent,,2020-11-12T11:15:33,0003
name4@test.com,undelivered,,2020-11-12T11:15:33,0004
```

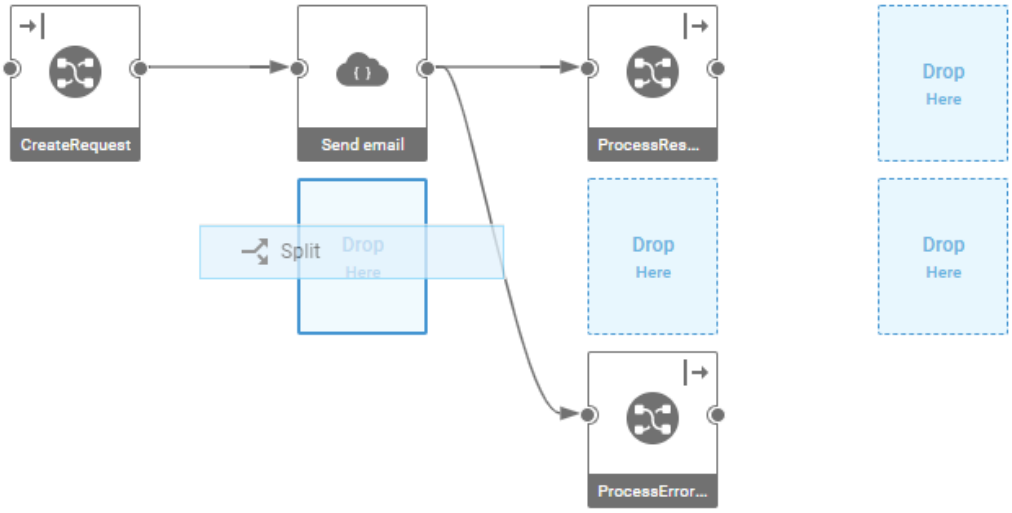
Step 9 – Enhance Flow to Process Large Data

The next step is to expand the flow – by adding **Split** and **Join** nodes – so that large data can be processed efficiently in parallel. The flow created above is fine if there are a small number of records to be processed, but if there are many rows, then we need to process requests in parallel to achieve adequate performance.

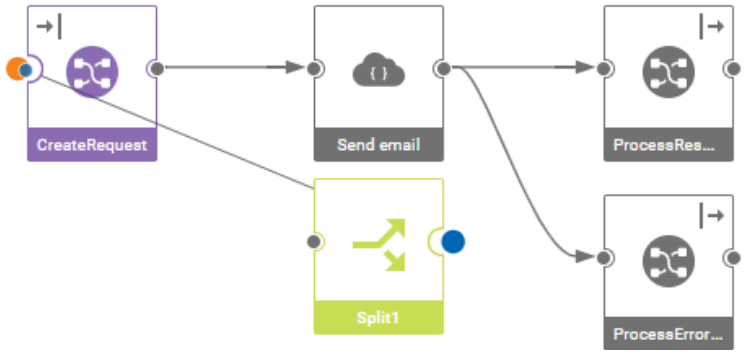
The **Split** node allows a large data set to be split into smaller batches, and each batch to be processed in parallel in separate threads of execution. The **Join** node then consolidates the results of the split flow, concatenating the results into a single output.

To create this bulk processing flow:

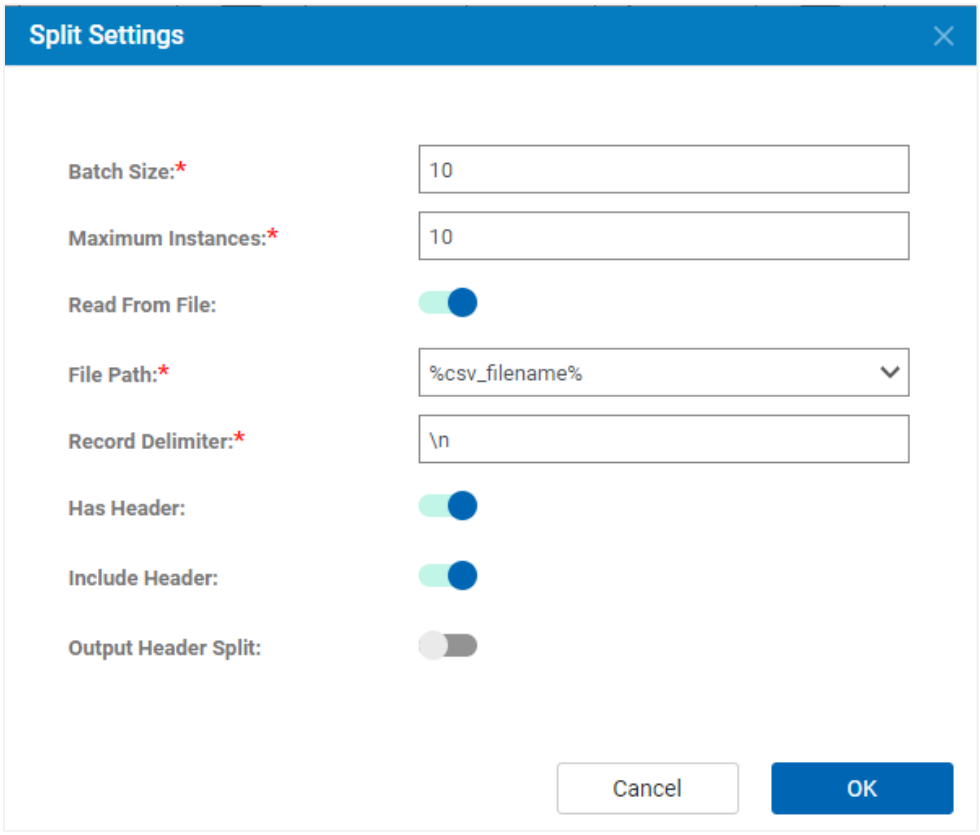
1. Open the **Send Emails** flow and click **Save As** ( ) from the toolbar.
2. Enter a name for the new flow, such as **SendBulkEmails**, and click **OK**.
3. Drag the **Split** node from the Palette and place close to the first node.



4. Connect the output terminal of the **Split1** node to the input terminal of the **CreateRequest** node:



5. Right click on the **Split** node to edit its properties, as shown and described below.



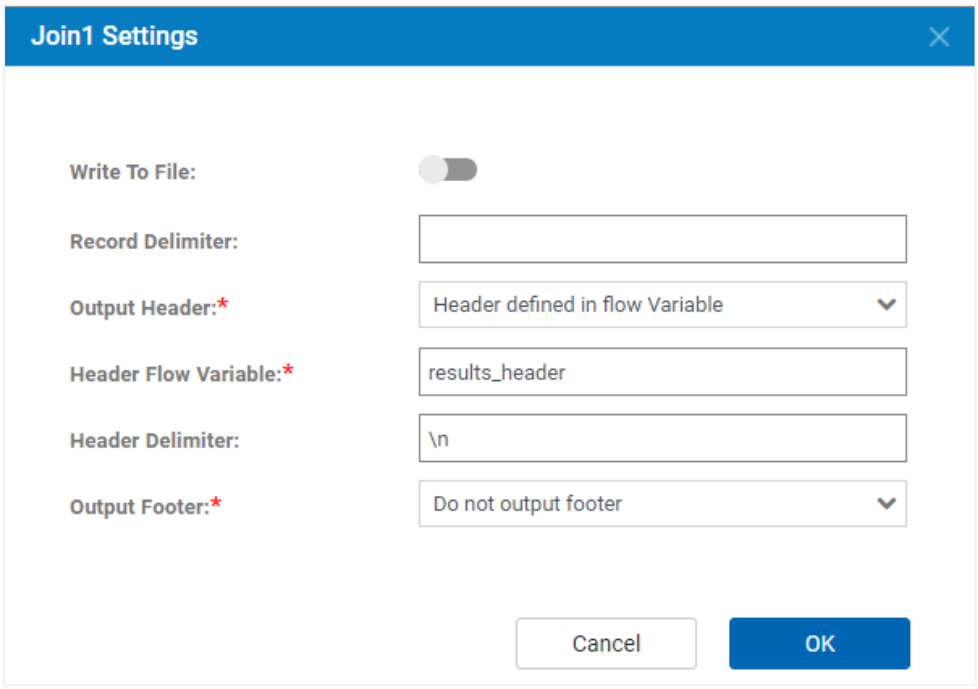
- **Batch Size:** Specifies how many rows will be contained per batch.
- **Maximum Instances:** Specifies how many parallel batches will be executed at one time.
- **File Path:** Specified as a flow variable **csv_filename**.
- **Record Delimiter:** Specified as **\n**, which means a newline character.
- **Has Header:** Specifies that the incoming CSV data has a header row.
- **Include Header**: Specifies that the header row should be provided in each batch sent to the output terminal.

6. Add a **Join node** after the **ProcessResponse** node.

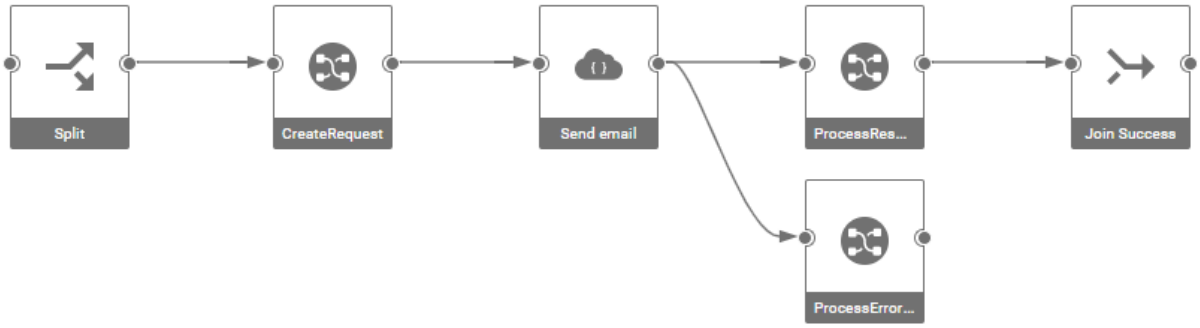
7. Connect the output terminal of **ProcessResponse** to the new **Join node**.

Copyright © 2021 HCL Software

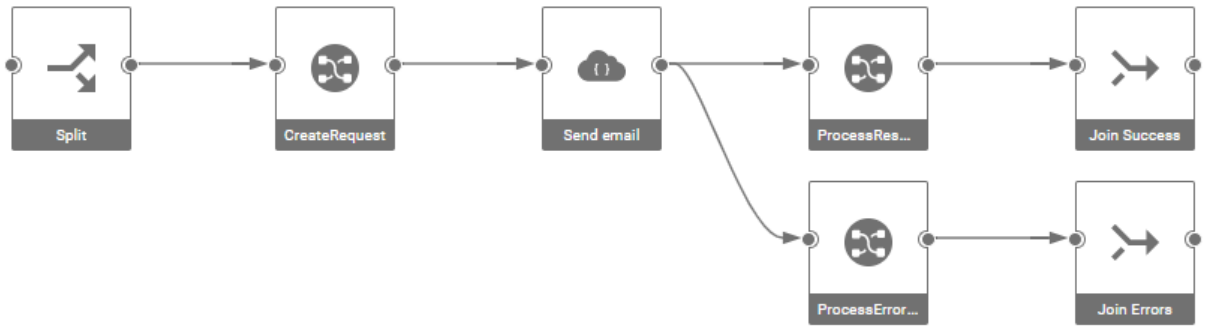
8. Right click over the new Join node and specify the Join Settings, as shown. Specify that the Output Header is coming from a flow variable named **results_header**. We will set up this flow variable in a subsequent step.



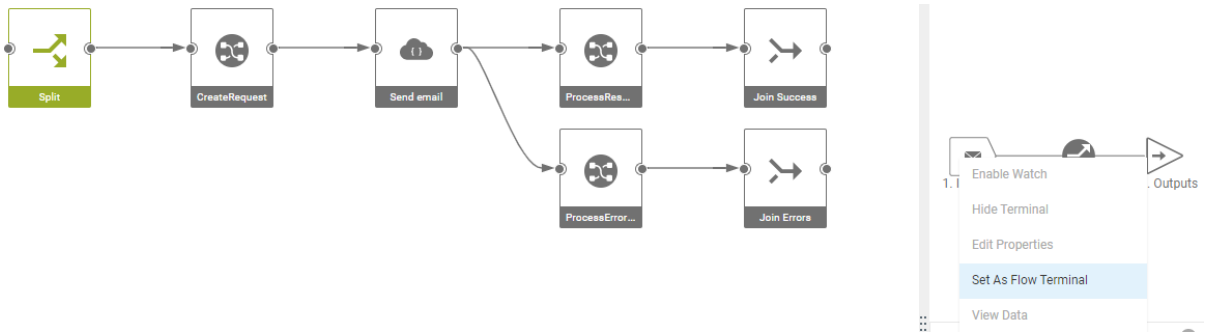
9. Click on the new **Join node** and add a meaningful name in the Information panel in the lower right. Click **Update.**



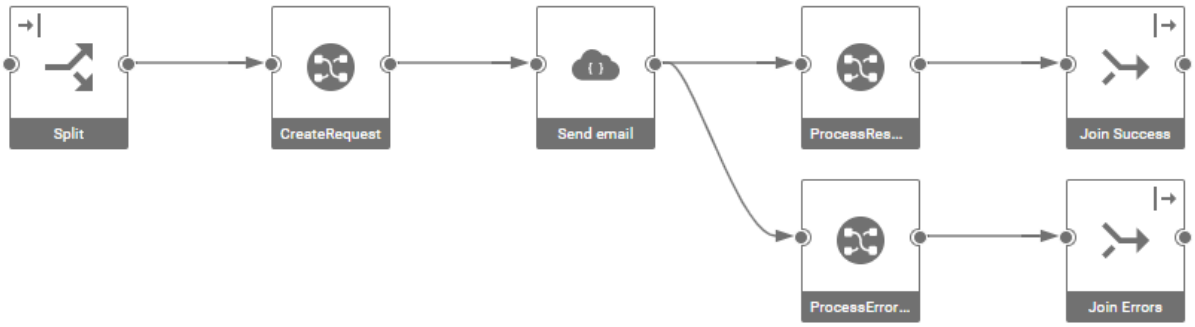
10. Repeat steps 6-9 to add a **Join node** following the **ProcessErrorResponse** map node.



11. After adding the Split and Join nodes, the flow terminals need to be set on the Split and Join source and target terminals, as before. Select the node in the flow, then right-click on the source (Split node) or target (Join Success and Join Errors nodes) to define the flow terminals.

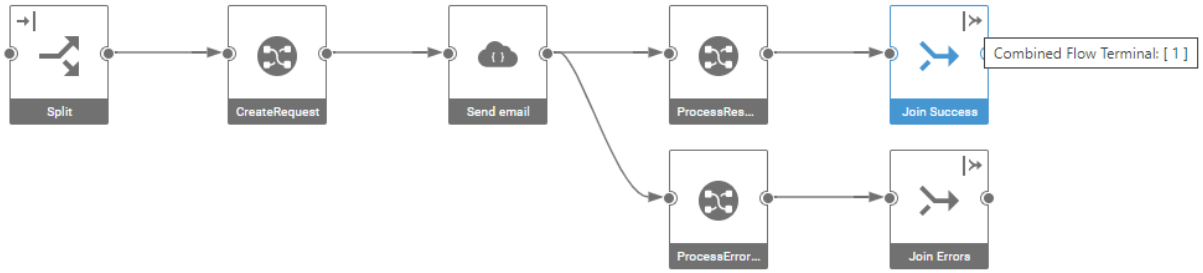


Your flow should now look like this.



12. Finally, we want the flow to combine the results from the success and error paths into a single CSV file. To do so, click on the Settings icon on the toolbar to bring up the Flow Settings and enable the **Combine Output Terminals** property. Click **OK.**

13. After this enabled, the icon on the Join nodes changes, and the tooltip indicates that the terminals are combined.



14. Click on Save on the toolbar to save your changes to the flow.

Step 10 – Extend the Request Map

The Join node will be setting the CSV header row to the value of the flow variable **results_header**, so this variable needs to be set in the **CreateRequest** map.

The output results file has a prescribed format necessary to be successfully updated in Campaign or Journey. It consists of fields from the audience data

Remember from earlier that the file <`devkit>/testconn/schemas/response.csv` is a small CSV file containing a header row and one row of data that represents the response results data to be sent to Campaign or Journey:

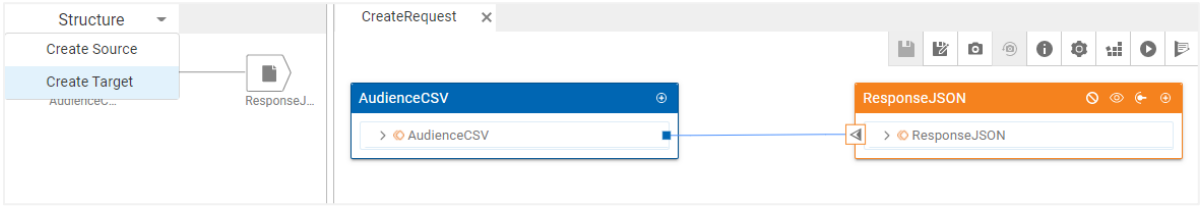
```
email,status,errormsg,timestamp,identity
fred@test.com,sent,,2020-07-20T13:01:19,001
```

The first four fields – email, status, errormsg and timestamp – in this CSV are fixed. However, the last field contains the identity context information that is sent from Campaign/Journey, passed to testConn in the Send email request, returned as the results from the Send email request and then returned to Campaign/Journey. The name of this field can vary from campaign-to-campaign or journey-to-journey. Therefore, we are going to get the name of this identity context field directly from the audience CSV input header and capture it in the **results_header** flow variable to be used when producing the output CSV.

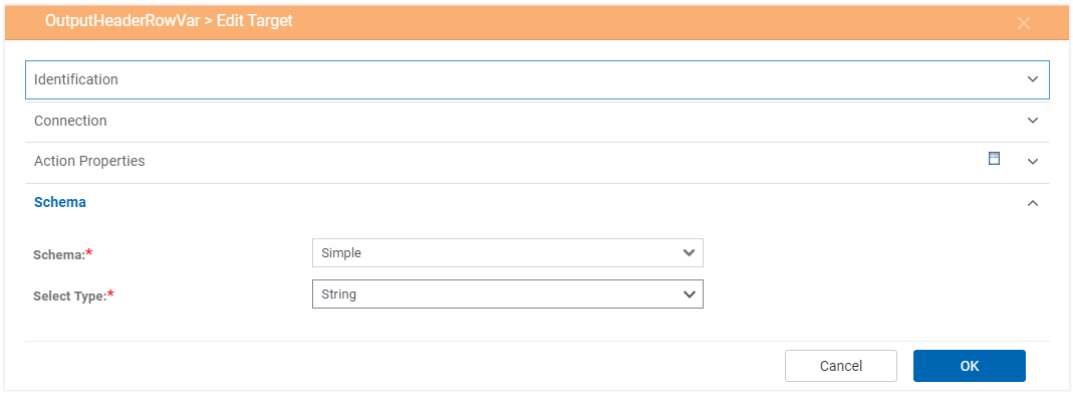
Copyright © 2021 HCL Software

To do this:

1. Open the **CreateRequest** map.
2. Click on **Create Target** to add this new output.



3. Give it a meaningful name, such as **Properties** or **OutputHeaderRowVar**. Click **Next**.
4. For **Connection**, specify the output as **Sink** and click **Next.** Click **Next** again.
5. Select a schema that has a single text object. One is provided in the tutorial project named **Simple**. Select **String** from the Simple schema for **Type.**

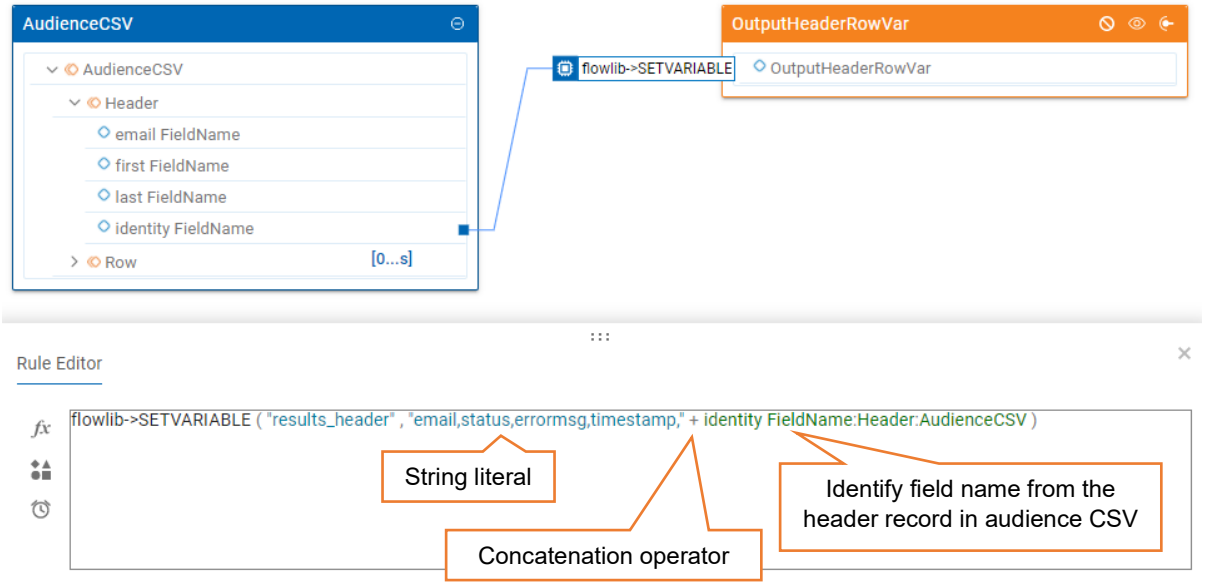


6. In the Structure panel, click on the new target to make it the current card in the mapping canvas.

7. Right-click on **AudienceCSV** in the Structure panel and select **Add** to add it to the mapping canvas.

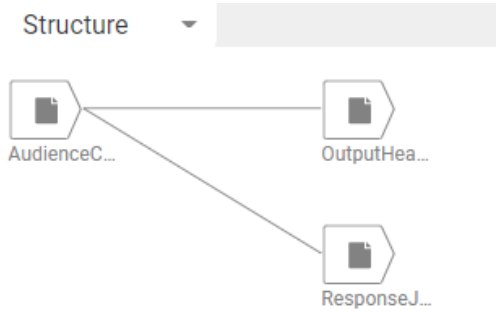
8. Use the flowlib->SETVARIABLE() function to set **results_header** to the value:

```
flowlib->SETVARIABLE("results_header", "email,status,errormsg,timestamp,"
+ identity FieldName:Header:AudienceCSV)
```



9. In the Structure panel in the upper left, switch the order of the outputs so that the new output is first. Simply drag the new target to be above the ResponseJSON target.

   This is very important and necessary, so that it is executed (and the output header flow variable value assigned) before data produced from the output link in the flow is sent to downstream nodes.

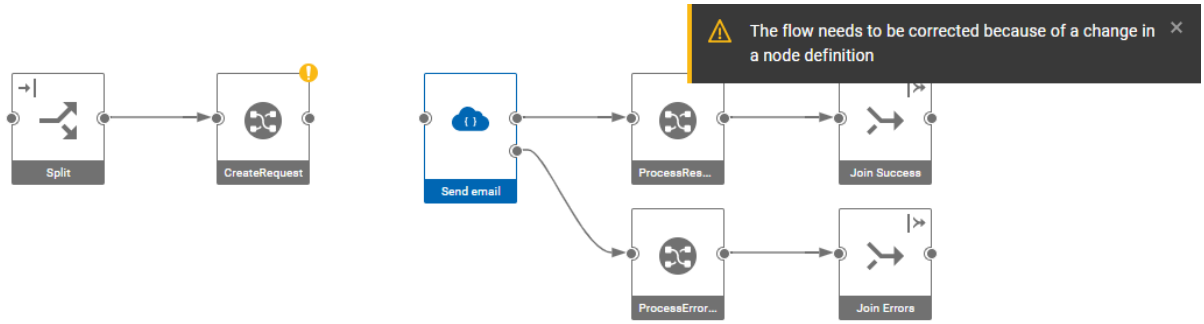


10. On the toolbar, click **Save** and then click **Build**.

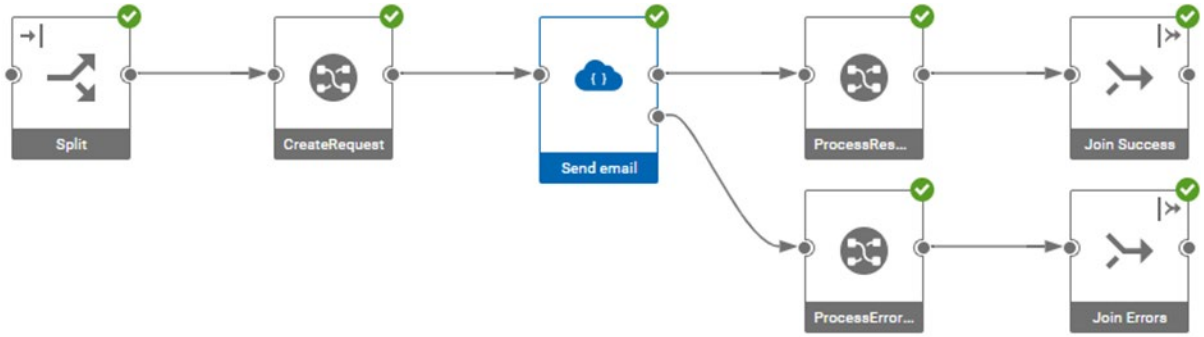
Step 11 – Updating and Running the Bulk Flow

After modifying the map, the flow needs to be refreshed to pick up the new map definition.

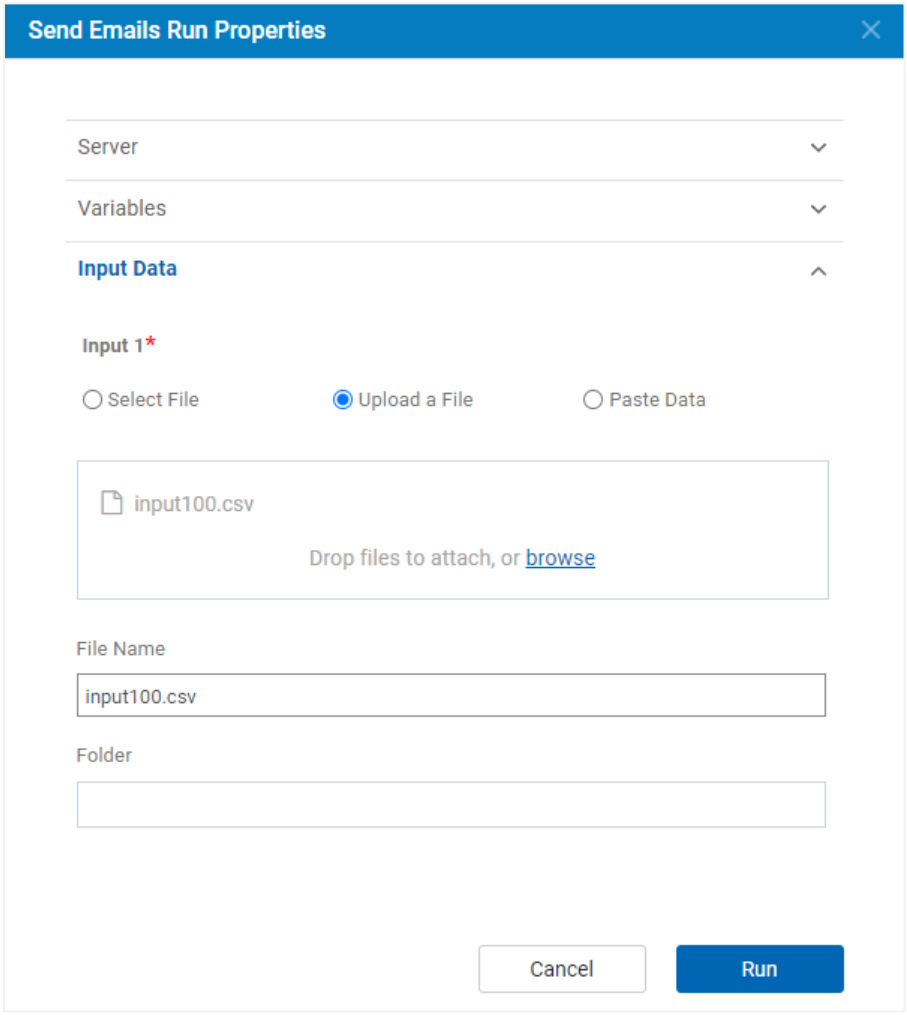
1. Open the **Send Emails in Bulk** flow.
2. A warning message will be displayed and the **CreateRequest** map node will have a warning icon.



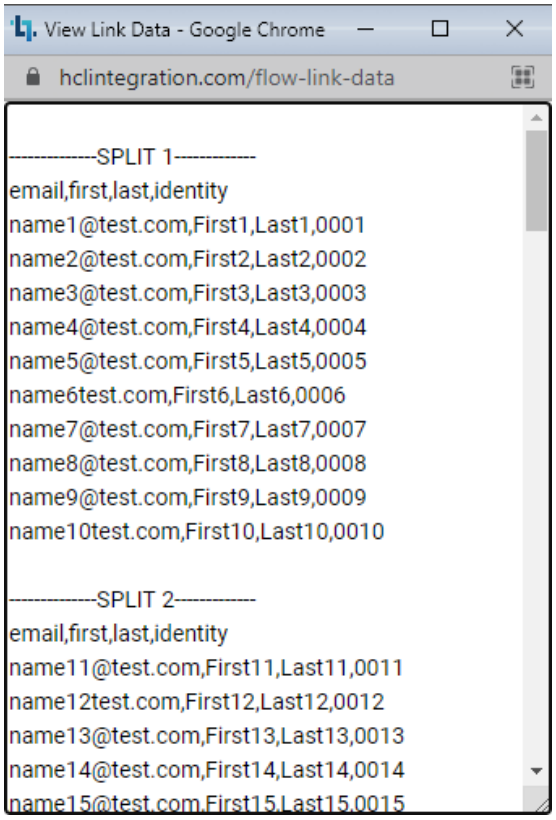
3. Drag the target terminal from **CreateRequest** to the source terminal of **Send email**.
4. On the toolbar, click **Save** and then click **Analyze**.



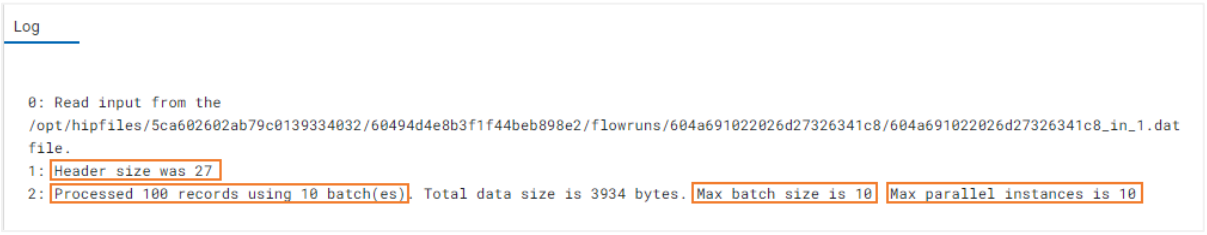
5. Run the flow as before, this time using the larger input data set, **input100.csv**, which can be found in `<devkit>/testconn/schemas/`.



6. Since the split node had **Batch Size** set to 10 and 100 records were provided in the test data, the flow will have run 10 threads in parallel. You can see this if you right-click on a link to see the data that is sent to the link. For example, click on the first link. The data will be displayed as:



7. Clicking View Log on the Split node confirms that 10 batches were created:



With this, we are done creating the connector flows and maps.

Task 4 – Create the Properties Descriptor

The next step is to define the properties that will be exposed to the user in the Unica Campaign or Unica Journey. This is specified by creating a properties JSON file. The tutorial files provide this in `testconn/testconn.json`. This file has the following contents:

```
{
    "version": 1,
    "id": "testconn",
    "name": "Testconn",
    "type": "utility",
    "category": "email",
    "description": "Testconn Sample",
    "validate": "action",
    "contexts": [
        "target"
    ],
    "package": "com.hcl.hip.adapters.m4rest:testconn",
    "template_version": 1,
    "select_schema": false,
    "testable": true,
    "properties": [
        {
            "name": "base_url",
            "label": "Base URL",
            "description": "The base URL of the service",
            "type": "string",
            "required": true,
            "scope": "connection"
        },
        {
            "name": "username",
            "label": "User ID",
            "description": "The user ID",
            "type": "string",
            "required": true,
            "scope": "connection"
        },
        {
            "name": "password",
            "label": "Password",
            "description": "The key for the user ID",
            "type": "string",
            "required": true,
            "masked": true,
            "scope": "connection"
        },
        {
            "label": "Subject",
            "name": "subject",
            "type": "string",
            "required": true,
            "description": "The subject of the email",
            "scope": "target_action"
        },
        {
            "label": "Template",
            "name": "template",
            "type": "string",
```

```
            "required": true,
            "enumeration": "template",
            "description": "The email template to use",
            "scope": "target_action"
        }
    ],
    "rest_config": {
        "service": "Testapp",
        "test_endpoint": "Ping",
        "enumerations": [
            {
                "name": "template",
                "endpoint": "Get Templates",
                "array_path": "templates",
                "value_path": "id",
                "label_path": "name"
            }
        ],
        "schema_mapping": [
            {
                "name": "Static fields",
                "static": [
                    {
                        "internal_name": "email",
                        "external_name": "Email",
                        "description": "Email address",
                        "type": "text",
                        "required": true
                    },
                    {
                        "internal_name": "first_name",
                        "external_name": "First name",
                        "description": "First name",
                        "type": "text",
                        "required": true
                    },
                    {
                        "internal_name": "last_name",
                        "external_name": "Last name",
                        "description": "Last name",
                        "type": "text",
                        "required": true
                    }
                ]
            }
        ]
    },
    "implementations": [
        {
            "name": "send_emails",
            "run": {
                "flow": {
                    "template": "SendBulkEmails"
                }
            }
        }
    ]
}
```

Connector information

The fields at the start of the file provide some metadata about the connector. These are explained in more detail in the Reference Guide.

Properties

This descriptor file defines five properties – three are connection properties (base_url, username and password) and two are action properties (Subject and template). The "scope" attribute specifies whether a property is a connection property or whether it is pertinent to a source or target action.

REST Configuration

The rest_config element provides some additional details that reference the services created in Task 2.

- The name of the service must match the service name created earlier.
- The 'test_endpoint' name specifies the 'Ping' endpoint created in the Service Editor.
- The endpoint name in the enumerations must match the endpoint created in the Service Editor.
- The schema_mappings defines a set of static fields that the connector expects on input. These fields can also be dynamically obtained from the resource, but for simplicity this example uses only static fields.

Implementations

The implementations element provides a means to specify the flow that implements the 'send email' action.

Descriptor validation utility

The file can be validated for correctness by running the 'validate descriptor' -vd packager command:

```
~/link/devkit/packager$ ./packager.sh /connectors/testconn /output/testconn -vd
Validating descriptor file ~/connectors/testconn/testconn.json...
Done!
```

Task 5 – Package the Connector

The connector can now be packaged up and deployed. To package the connector:

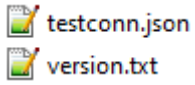
1. Create a local directory (\connectors\testconn in this example).

2. Into this directory, copy the testconn.json properties descriptor.

3. Create a file named version.txt containing a version string, e.g., "v1".

Copyright © 2021 HCL Software

The directory should now contain these files:



Package this into a connector zip by running the packager tool. You can see the options that the packager tool supports by going to the packager directory and typing 'packager'.

4. To package the connector into an output directory called '\output\testconn', invoke the packager tool with -p (package) and -ep (export package) options:

```
C:\packager> packager \connectors\testconn \output\testconn -ep Testconn -
p -h https://localhost:8443/ -u admin -pw ****
Exporting project Testconn to \connectors\testconn\Testconn.zip...
Done.
Exporting the service definition to
\connectors\testconn\service\Testapp.json...
Exporting the service definition as a configuration...
Validating configuration file
C:\connectors\testconn\config\testconnConfig.json...
Variables:
   base_url referenced from:
      endpoint: Get Templates, field: url
      endpoint: Ping, field: url
      endpoint: Send Email, field: url

Validation complete: No issues were found.
Validating descriptor file C:\connectors\testconn\testconn.json...
Packaging files into directory C:\output\testconn...
Done!
```

   In the -u and -pw commands specify the username and password of an administrative user. By default, in a development environment, an admin user is created with username 'admin' and password 'admin'.

This command did a number of things:

- Exported the Link project named "Testconn" to Testconn.zip in the connector directory
- Exported the service definition to service\Testapp.json in the connector directory
- Validated the properties definition file
- Created the output directory, if it did not exist
- Zipped everything up into the output directory

After running the packager, it will have created a properties file called testconn_labels.properties containing label and description strings in the connector directory. This file can be sent to a translation service to create localized descriptors. Translated properties files should be named <basename>_<language>.properties, where language can be de, es, fr, it, ja, ko, pt_BR, ru, zh and zh_TW. When the packager tool is run, it will produce a localized descriptor for each language properties file that it finds.

Copyright © 2021 HCL Software

In the output directory, there will a zip file named testconn_connector_v1.zip. This name is composed of the connector id, the string "_connector_" and the version number.

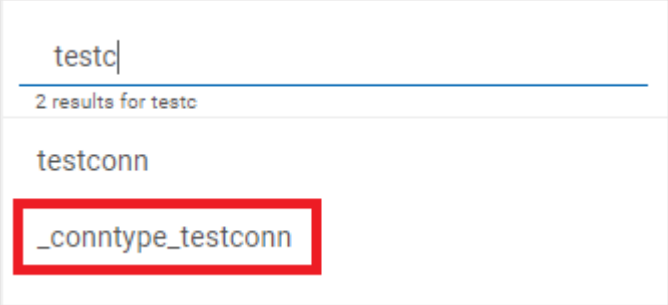
Task 6 – Install the Connector

1. To install the connector, copy it to the modules directory on the server. By default, this is /opt/hipmodules.

2. After copying the file, it needs to be picked up by the design and runtime servers. This can be done by invoking the update_modules API in both servers, but the simplest approach is to simply restart both servers:

```
docker restart hip-server
docker restart hip-rest
```

5. To verify that it was installed correctly, refresh the Designer UI and in the project list filter the projects to look for "testconn". There should be a project named _conntype_testconn



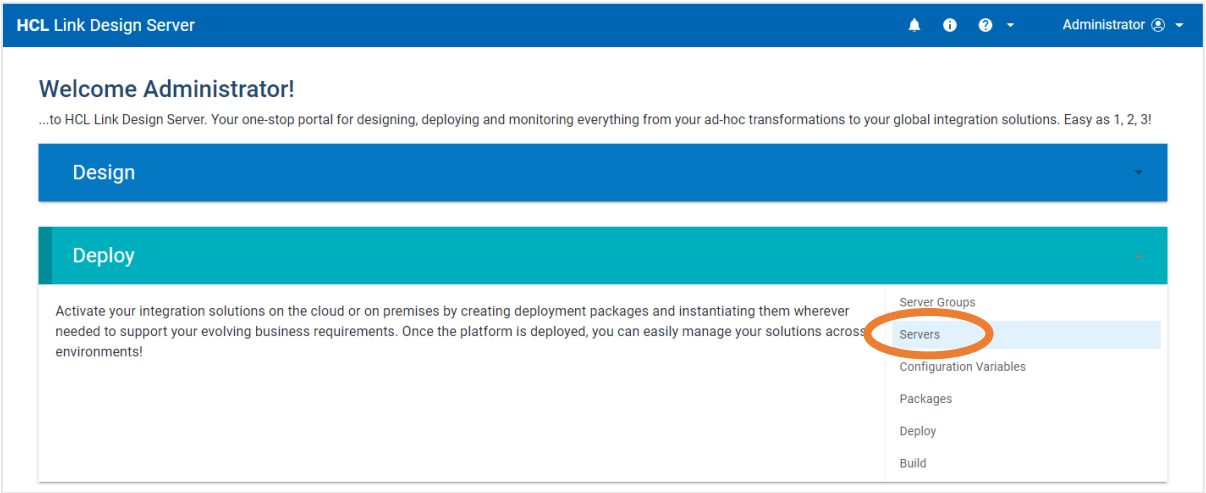
This is a copy of the project that was exported into the packaged connector.

Task 7 – Test the Connector

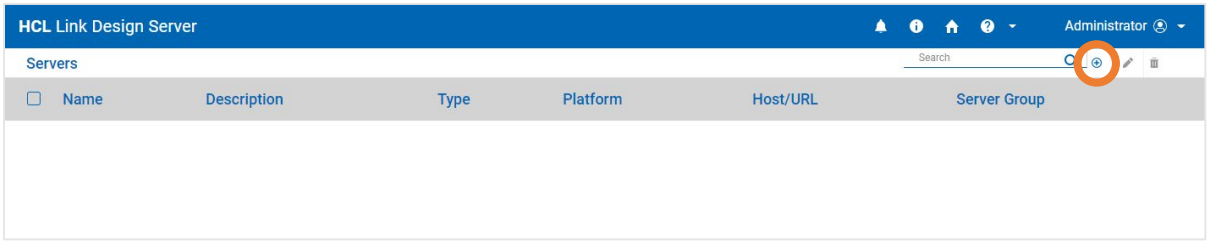
Step 1 – Define a Server

A server definition needs to be created which references the Link runtime.

1. From the Link home page, expand the Deploy options, and click on Servers.



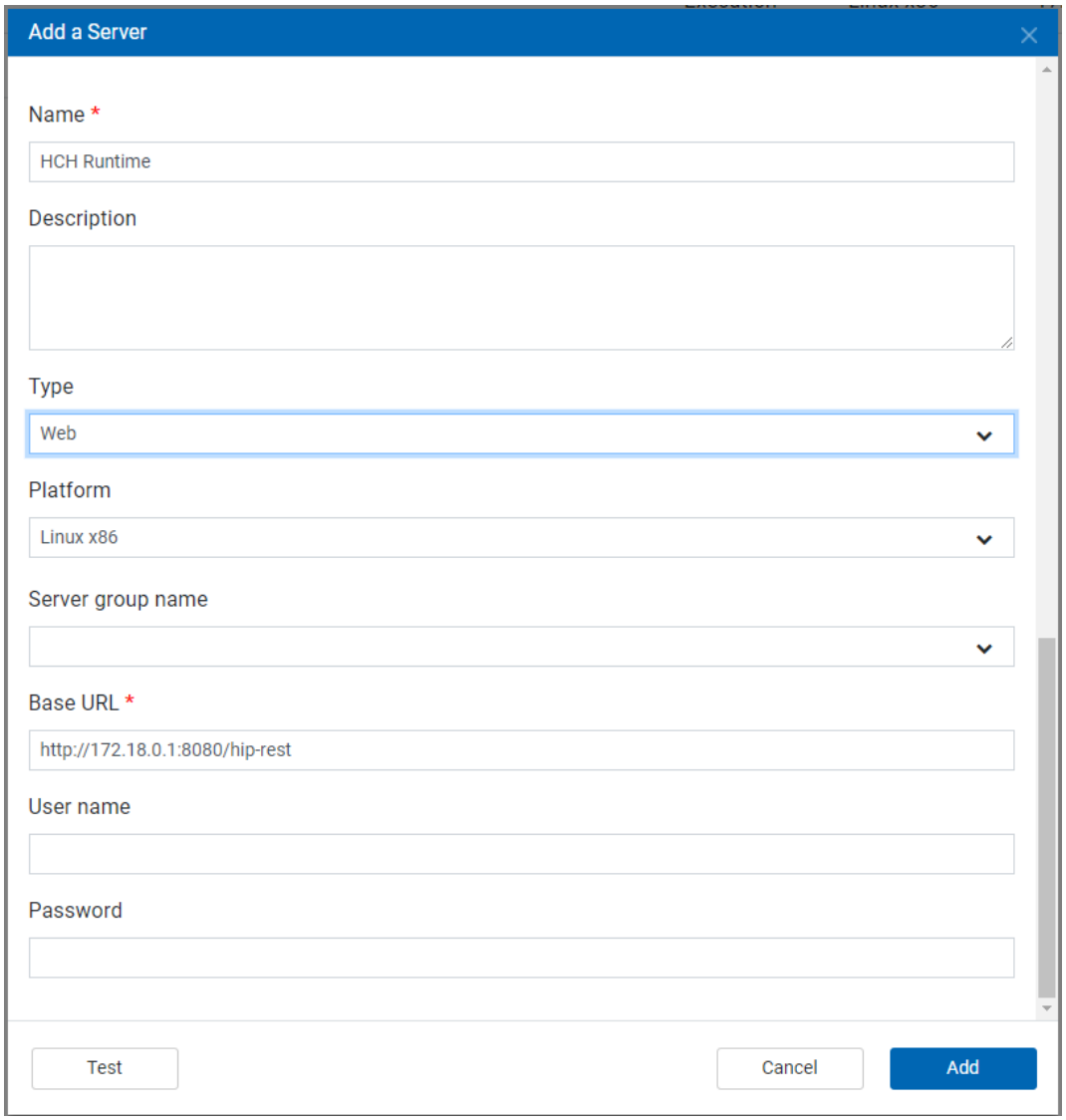
3. Click on the '+' icon to add a new server definition.



4. Name the server 'HCH Runtime'. The name must be exactly this, because the deployment step looks for a server with this name.

5. Change the Type to 'Web'.

6. Specify the Base URL as http://<host>:8080/hip-rest.



7. Click 'Test' to test the connection to the server

8. Click 'Add' to save the server.

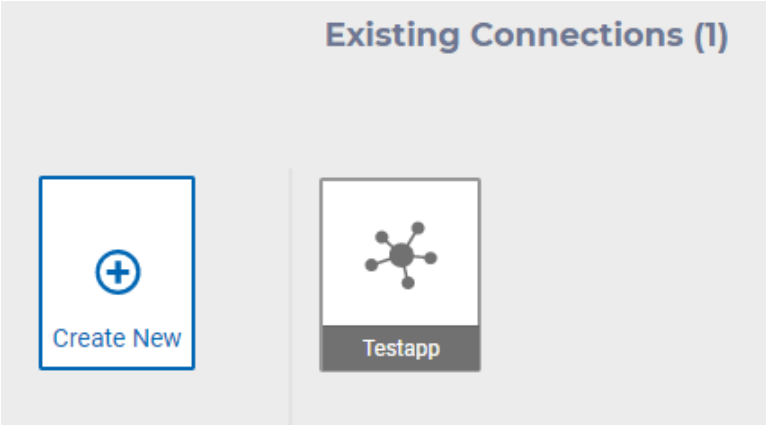
Step 2 – Create a Connection

The next step involves creating a connection to the resource using the connector.

1. Invoke this URL in a browser to open the connections user interface:

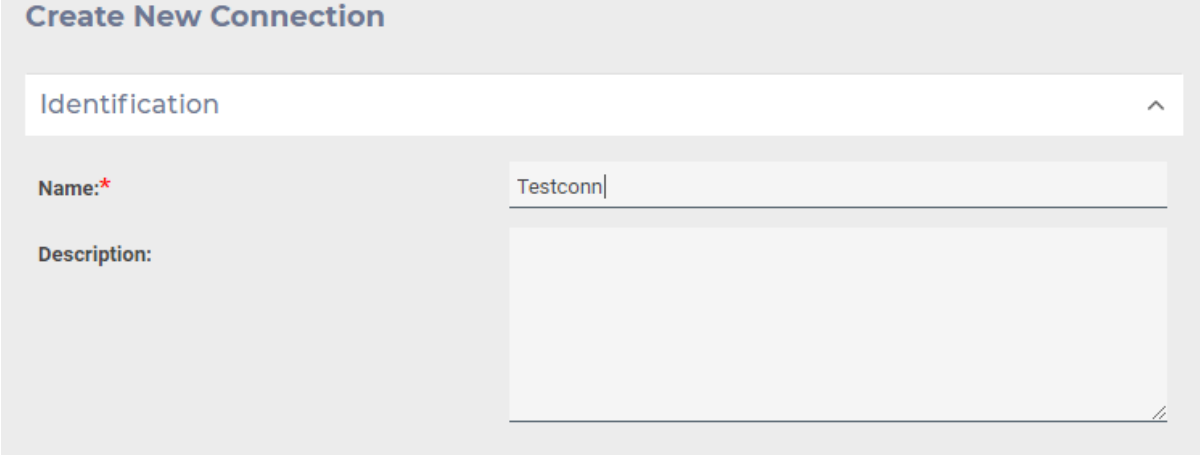
   https://localhost/hch/_app_testapp/existingConnection?dev=true&isAdminUser=true

The URL includes the name of the application (testapp) and invokes the 'Existing Connections' endpoint. This should display the following:

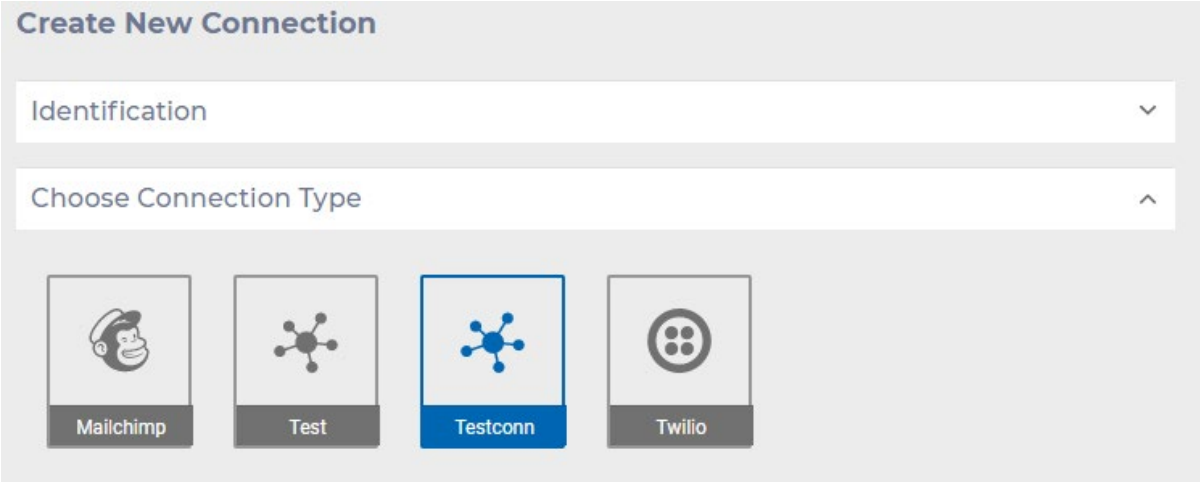


The Testapp connection was created when the testapp application was installed.

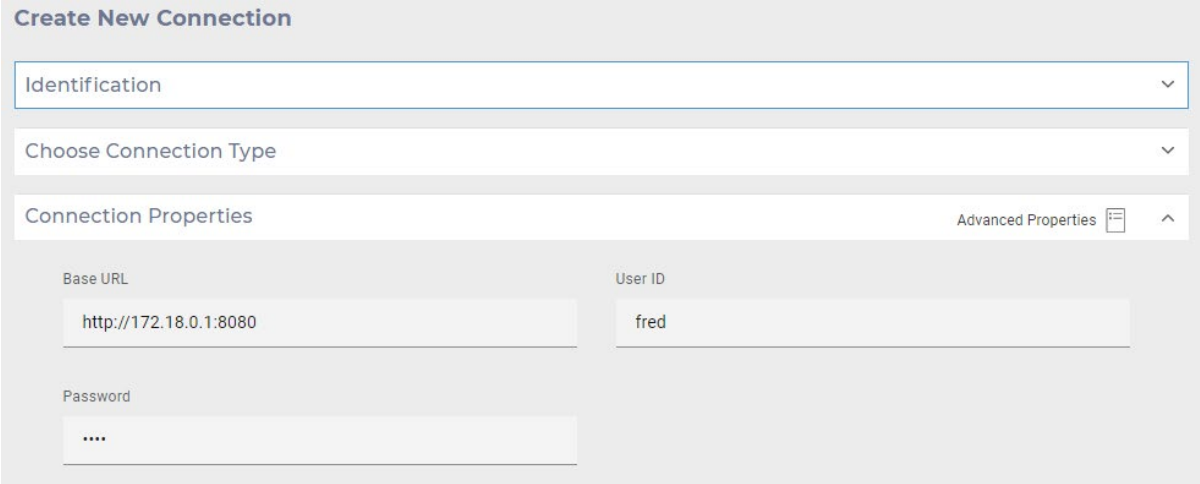
2. Click on 'Create New' and specify a name for your connection:



3. Click on 'Next' and select Testconn from the connection types:



4. Click on 'Next' and enter properties for the connection. Note that these are the properties defined above in the properties descriptor file. Set the IP address to the address of the host machine.



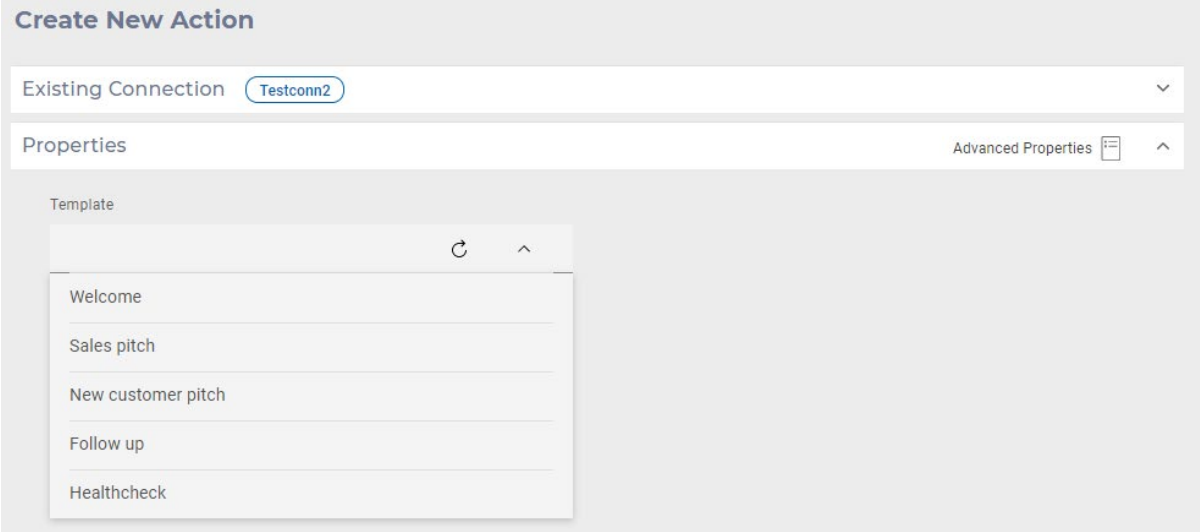
5. Click the Test button to test the connection. This will invoke the "Ping" endpoint that was specified in the descriptor file (`"test_endpoint": "Ping"`).

6. Now we have defined a connection, we can define an action for the connection. Copy this URL to your browser:

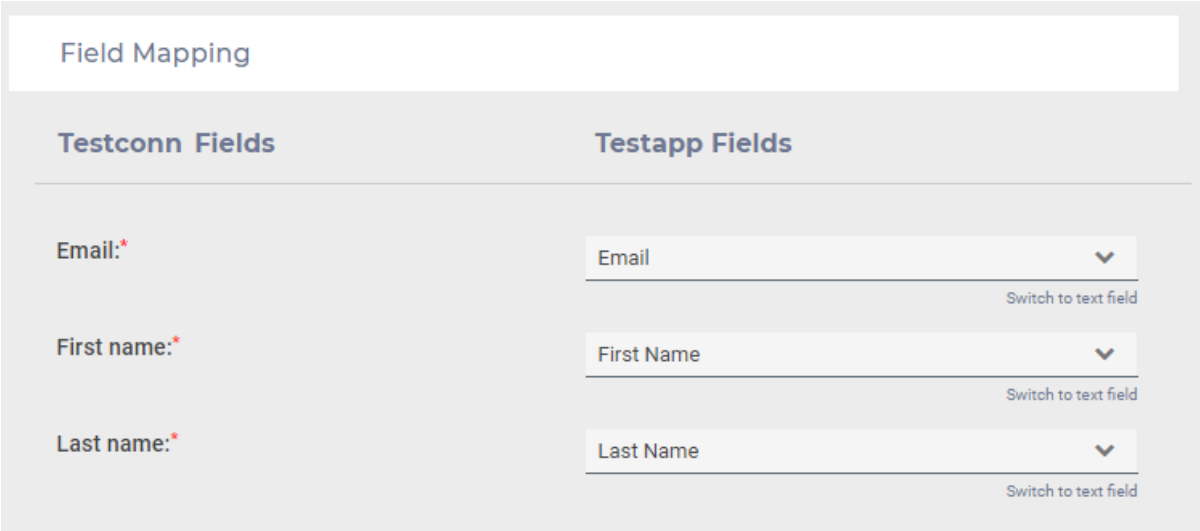
```
https://localhost/hch/_app_testapp/createAction?actionName=tc_action4&connectionName=Testconn&dev=true&deploy=true
```

This specifies the connection that we just created (Testconn) and gives the action a name (tc_action4). This will display the action properties, of which there is only one: Template. Clicking in the Template property invokes the Get Templates endpoint which was listed as an enumeration for the template property:

Copyright © 2021 HCL Software



7. Click Next to display the field mapping screen. This displays the static fields obtaining from the schema mapping in the Testconn descriptor, and allows for the selection of fields from the Testapp application:



8. Finally, click Save. When Save is clicked the following occurs:

   - The properties and field mapping are saved to the Link repository
   - If this is the first action, artifacts are copied from the connector's project (_conntype_testconn) to the application project (_app_testapp)
   - The flow and maps are packaged and deployed to the runtime.

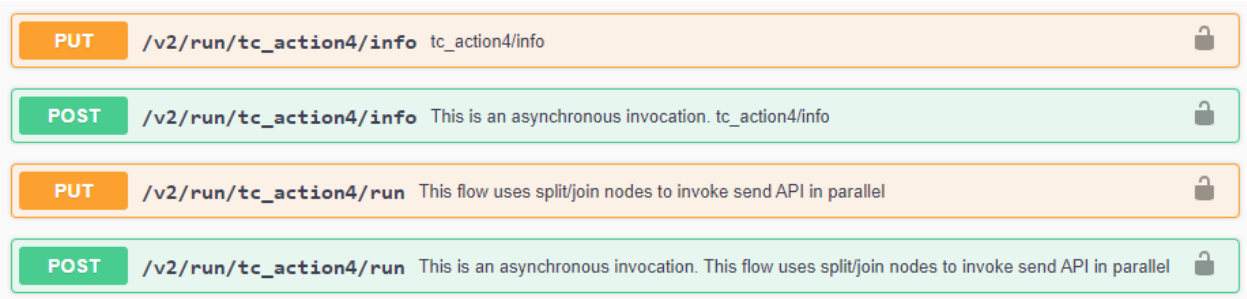
Running the Action

After an action has been deployed it can be run via the URL specified in the package. It is convenient to invoke it from the Swagger page that is generated from the list of deployed packages.

To access the Swagger use this URL, adjusting the hostname accordingly:

http://localhost:8080/hip-rest/api-docs?url=/hip-rest/v2/docs

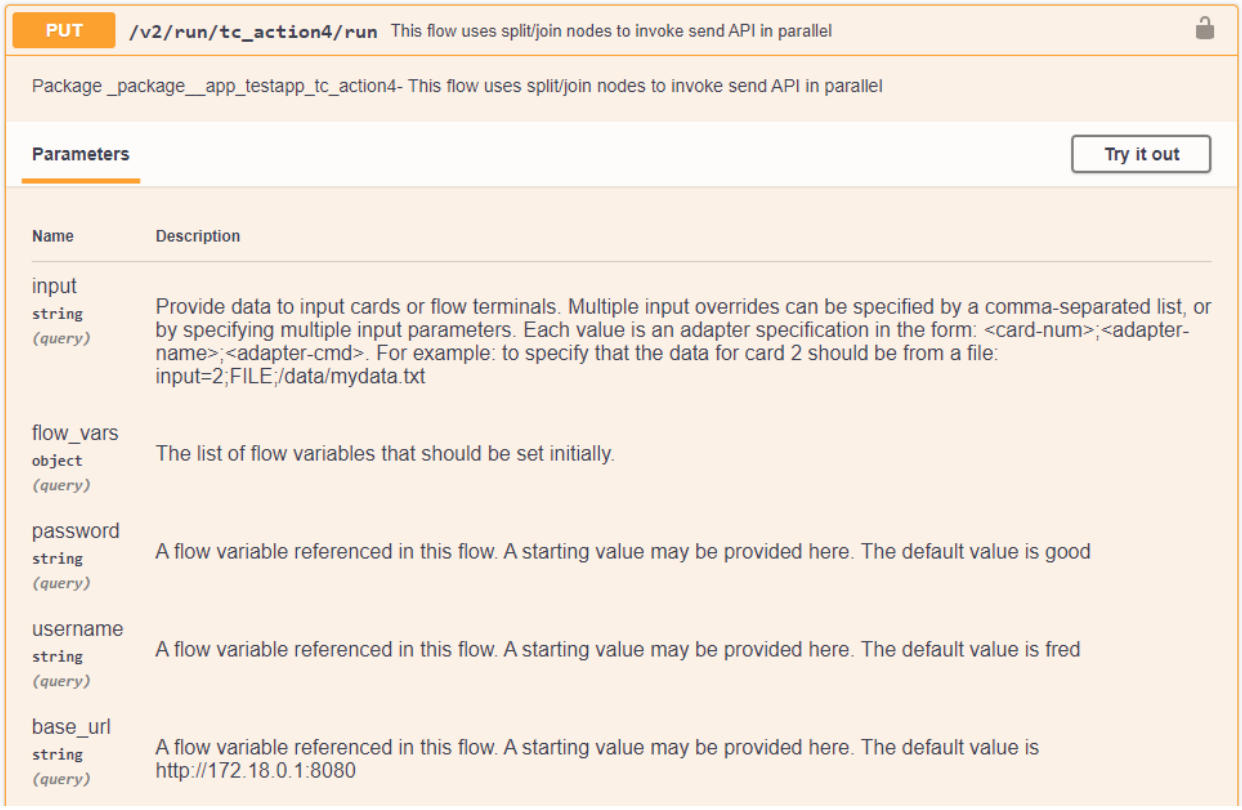
On the Swagger page you will see entries like this that contain the name of the action you created above:



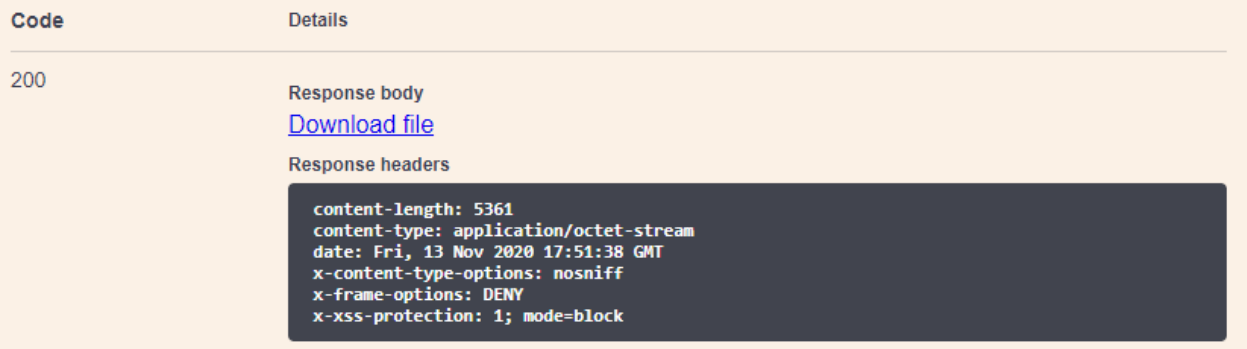
The PUT endpoints are synchronous calls, and the POST endpoints are asynchronous. The info API provides information about the endpoint. Try invoking the PUT /v2/run/<action>/info API. It will return the following:

```
{
    "connection_type": "testconn",
    "connection_name": "Testconn2",
    "action_id": "tc_action4",
    "creation_date": "2020-11-13T16:27:04.407+0000",
    "run": {
        "url_path": "/v2/run/tc_action4/run",
        "fields": [
            {
                "name": "email",
                "type": "text",
                "label": "Email",
                "is_key": false,
                "is_required": true
            },
            {
                "name": "first_name",
                "type": "text",
                "label": "First Name",
                "is_key": false,
                "is_required": true
            },
            {
                "name": "last_name",
                "type": "text",
                "label": "Last Name",
                "is_key": false,
                "is_required": true
            }
        ]
    }
}
```

To invoke the action (run the flow), click on PUT /v2/run/<action>/run and click on the 'Try it Out' button. The flow variables defined in the flow are shown as query parameters with the default value shown, but the values can be overridden:



When running the flow, select the input100.csv file that is in the Link project. The flow will run, and the results can be obtained from the Download file link:



The return parameter determines what gets returned from the flow. If the value is set to 'status' then a summary execution is returned. If set to 'log' then a more detailed log is produced, including audit log for all map nodes.

Next Steps

Having worked through this tutorial you have been introduced to almost all the aspects of developing a connector. The **Creating Link Connectors – Reference** document expands upon the topics covered in this tutorial and should be consulted when developing a Link connector.

Copyright © 2021 HCL Software