Unica Link V12.1.4
Creating Link Connectors

Creating Link Connectors

The purpose of this document and associated project and service is to demonstrate how to create connectors using the Link Connector Wizard.

Contents

1 Introduction 4

2 Set Up 5

3 Understanding the Artifacts 7

3.1 Mockmail Service 7

3.1.1 Connection Test Endpoint 7

3.1.2 Enumeration Endpoint 8

3.1.3 Field Mapping Endpoint 9

3.2 Connector Project 10

3.2.1 Template Flows Vs Action Flows 11

3.2.2 Send Email – Basic 11

3.2.3 Send Email – Error Handling 15

3.2.4 Send Email – Identity 16

3.2.5 Send Email – Split 18

4 Creating a Connector 21

5 Testing the Connector 26

5.1 Create a Connection 28

5.2 Create an Action 30

5.3 Running the Action 33

5.3.1 Running from _app_testapp Project 33

5.3.2 Running from the Runtime Server 35

6 Identity Field Handling 36

7 Flow Deployment Properties 38

8 Flow Requirements 40

8.1 Run Flow 40

8.2 Results Flow 41

9 Flow Concepts 41

9.1 Flow Execution 41
9.2 Node Terminals 43
9.3 Flow Terminals 43
9.4 Flow Variables 44
9.5 Cache Variables 45
9.6 Initialization flow 45
9.7 Split node 46
9.8 REST Client Node 48
9.9 Flow Performance 52
10 Java Plug-Ins 52
10.1 Create Java Class 53
10.2 Adding Plugin to Connector 54

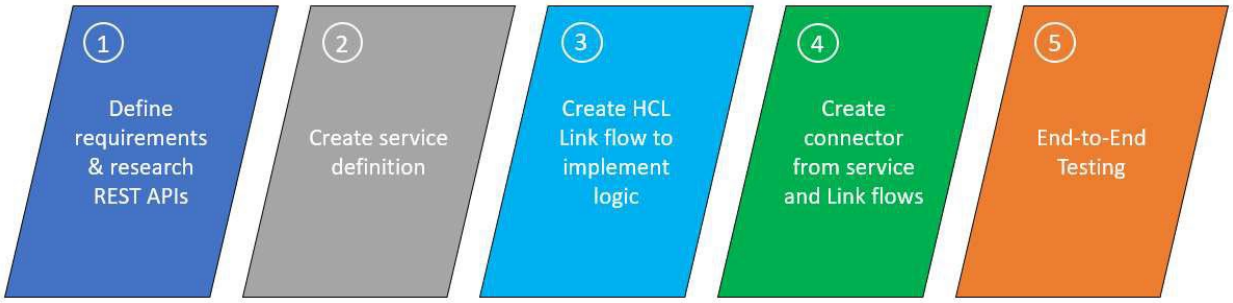
1 Introduction

This document provides a walkthrough of the steps required to create, test, and deploy a Link connector. It is based around “Mockmail”: a dummy email service that is provided from Link server. It does not actually send emails, but its REST APIs emulate those that a real email service would provide.

This document describes:

1. How to create Link flows that invoke a ‘send email’ operation to send a templated email to a list of users.
2. How to create a connector that utilizes these flows.
3. How to deploy and test the connector.

Creating a 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. 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.

4. Next, you use the connector wizard to create a connector from the service definition and the Link integration flows. The connector is then deployed to a test environment.

5. Finally, it is time to do end-to-end testing, which can be done directly from HCL Link’s user interface, from Link runtime’s Swagger page, or from Unica Campaign or Unica Journey.

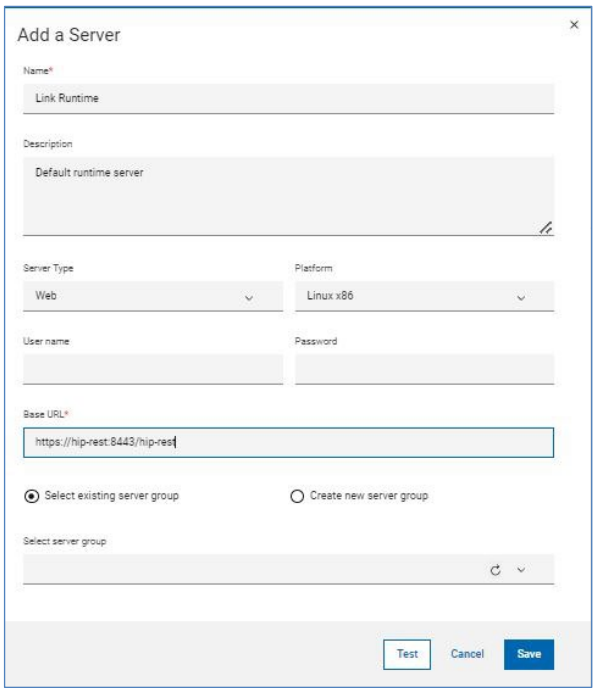
2 Set Up

Perform the following steps to set up the connector artifacts:

1. Import the “Mockmail” Link project and name it “Mockmail”. This will import both the Mockmail project and the Mockmail service.

2. In Link’s server page, create a server named “Link Runtime”. To do so:

   I. Click on “Servers” tab
   II. Click on “Create New Server”
   III. Select server type as Web, and specify the URL of the runtime server:



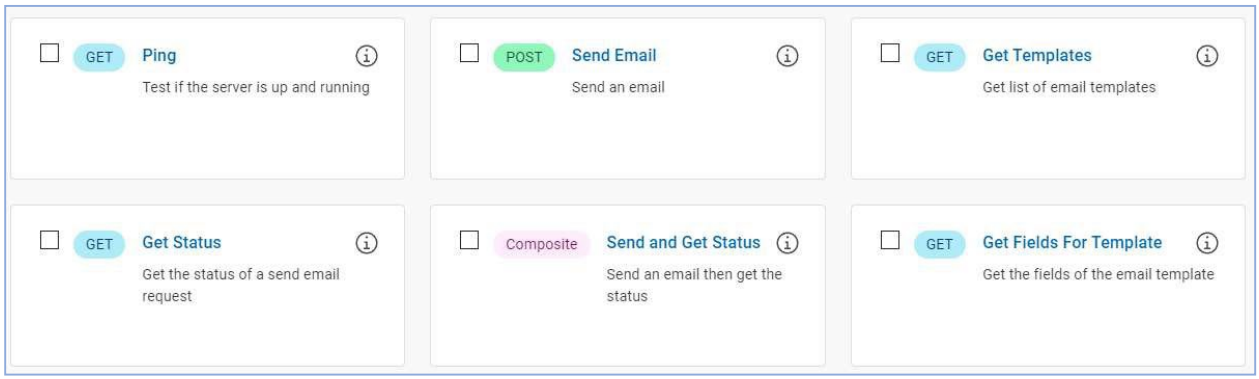
Note: Selection of existing server group or creation of new server group is optional for creation of Link Runtime server.

3 Understanding the Artifacts

Before creating the Mockmail connector, it is important to understand the artifacts that comprise it.

3.1 Mockmail Service

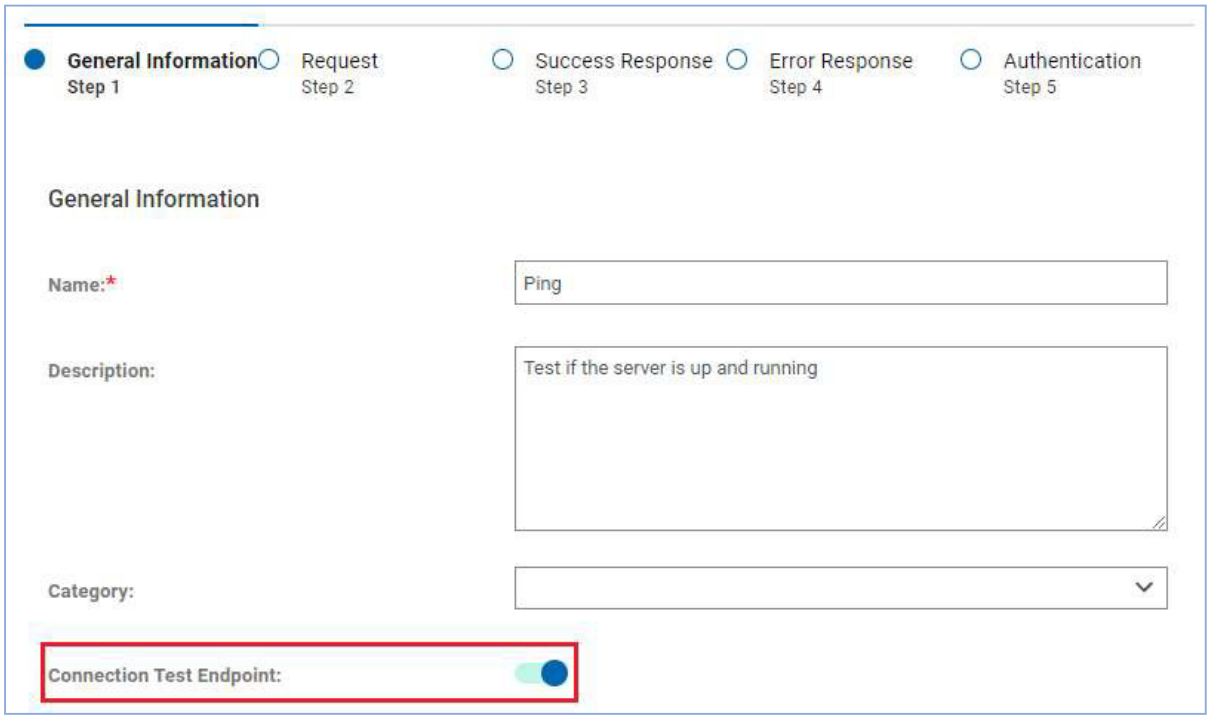
The Mockmail service defines the API that implements the Mockmail service. It contains 6 endpoints:



- **Ping** – Tests that the service can be reached. This is used in 'test connection' functionality
- **Get Templates** – Get a list of templates. This is used to populate the list of values in the connector's Template property
- **Get Fields for Template** – Gets the fields for the specified template. This is used in the field mapping page
- **Send Email** – Mocks sending an email to a specified recipient and returning status as to its successful transmission. This is used in the connector's flows.
- **Get Status** – Get the status of a send operation. *[Not currently used in this demonstration]*
- **Send and Get Status** – A composite endpoint that calls Send Email and Get Status in sequence. *[Not currently used in this demonstration]*

3.1.1 Connection Test Endpoint

The Ping endpoint is identified as a connection test endpoint:

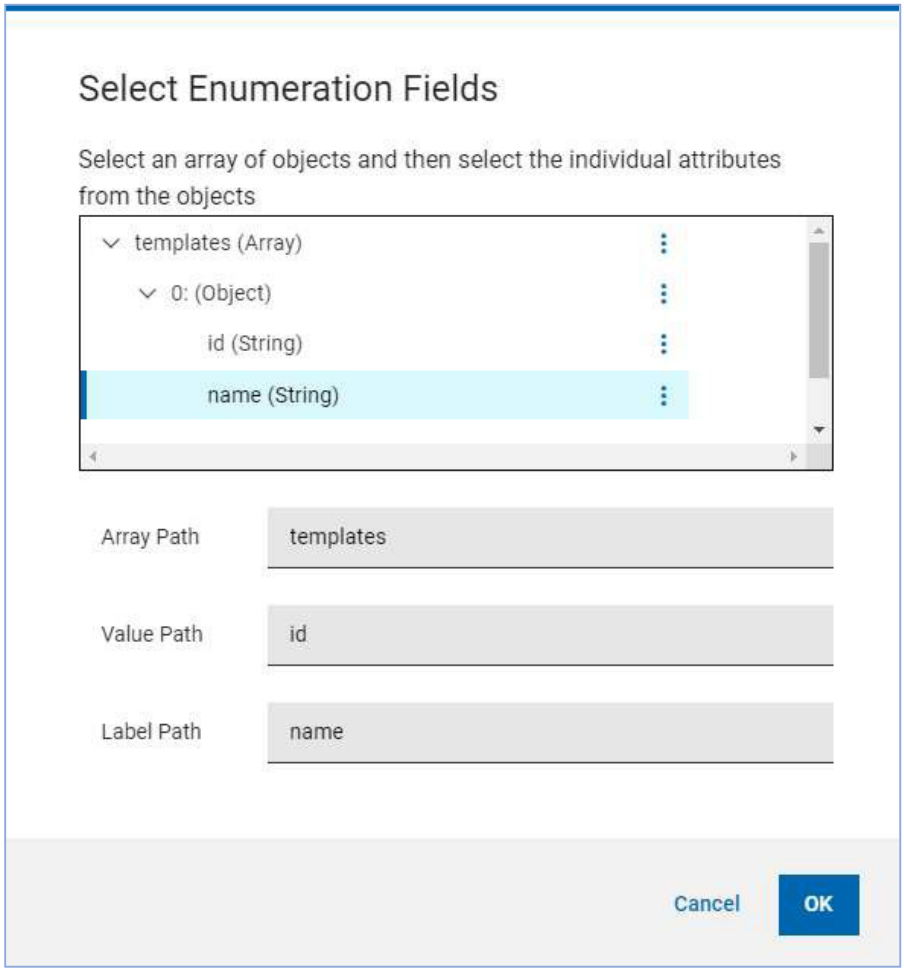


When a connector is created that references this service, an endpoint marked as being a connection test endpoint will be invoked from the **Test** button when defining a connection.

3.1.2 Enumeration Endpoint

An enumeration endpoint is one which provides a list of values for a property, whereby that list is obtained dynamically when a user is configuring a connection or an action. For endpoints that provide enumerated values, the fields comprising the enumeration response can be defined from the Success Response page of the endpoint:

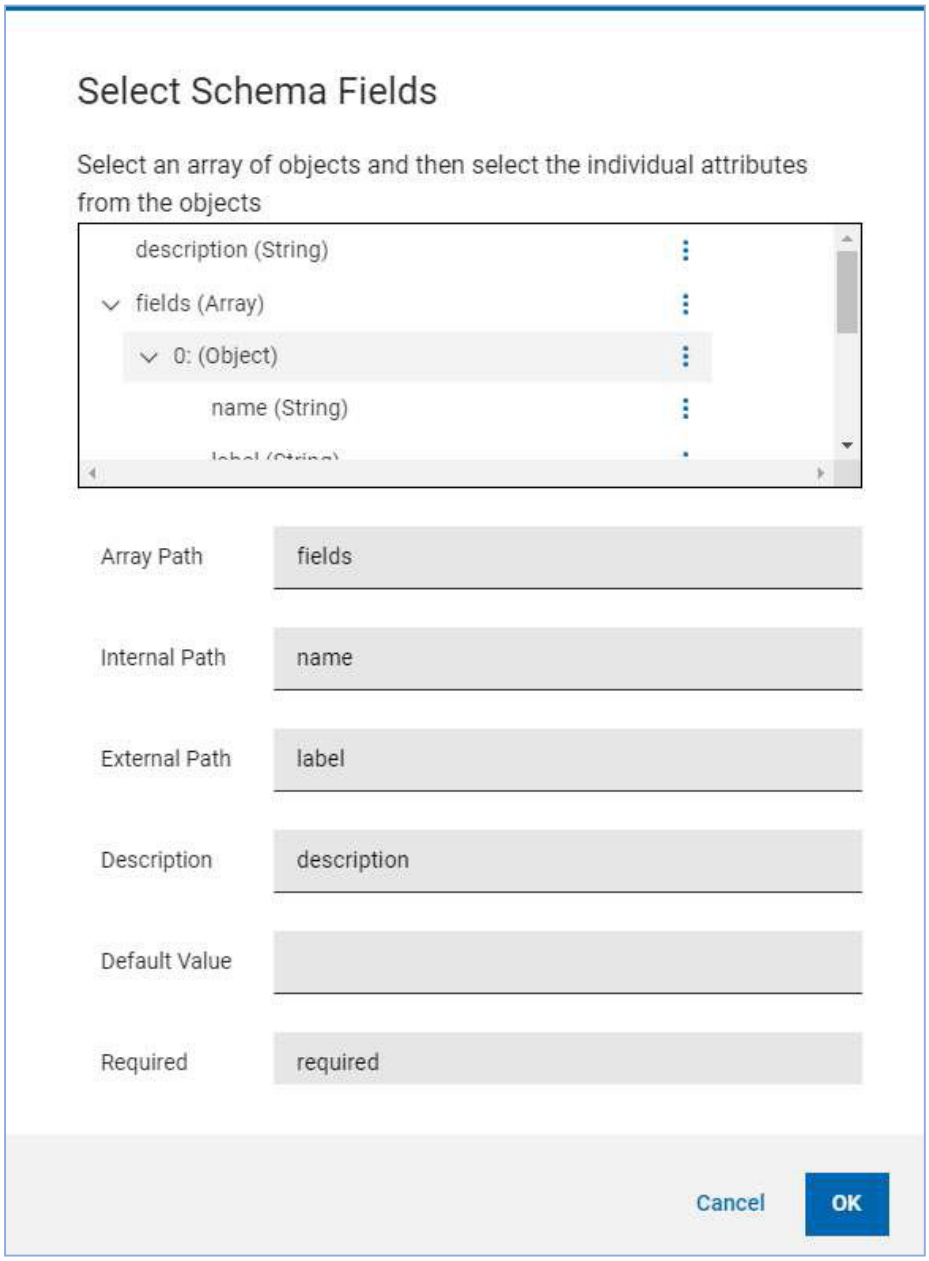
This allows for the paths of the response object to be selected that define the array of results, and the name and value fields within the response objects.



The **Get Templates** endpoint has an enumeration defined. This endpoint is called when a user is selecting a value for the available templates.

3.1.3 Field Mapping Endpoint

Similar to how an enumeration is defined, endpoints that provide field or object definitions can have a field mapping defined to gather the attributes of the field definitions:



The **Get Fields For Template** endpoint has a field mapping defined. This is called when defining a field mapping for an action. The endpoint is called with the selected template ID as an input parameter, which returns the available template fields for the selected template.

3.2 Connector Project

The Mockmail project contains 4 flows, each adding more capability on the prior one:

1. **Send Email – Basic** – Basic functionality to sends emails and report successful sends
2. **Send Email – Error Handling** – Adds functionality to capture failing sends and combine into the returned status
3. **Send Email – Identity** – Adds functionality to handle identity fields
4. **Send Email – Split** – Adds Split/Join functionality to parallelize the execution of the flow

These flows are explained in detail below.

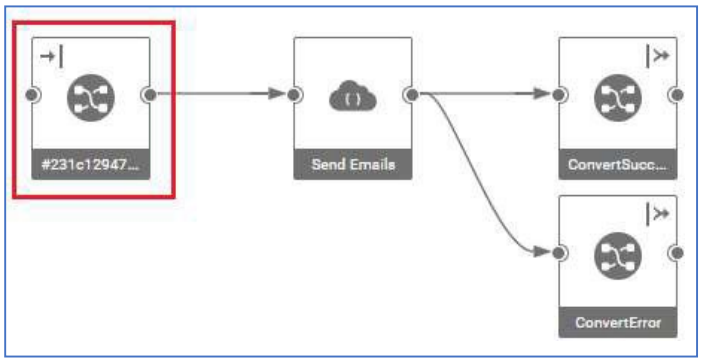
3.2.1 Template Flows Vs Action Flows

Before proceeding, it is important to understand that the flows of a connector are template flows, from which modified flows that will be created when an action is deployed. When an action is configured, the corresponding flow is copied and modified in the following ways:

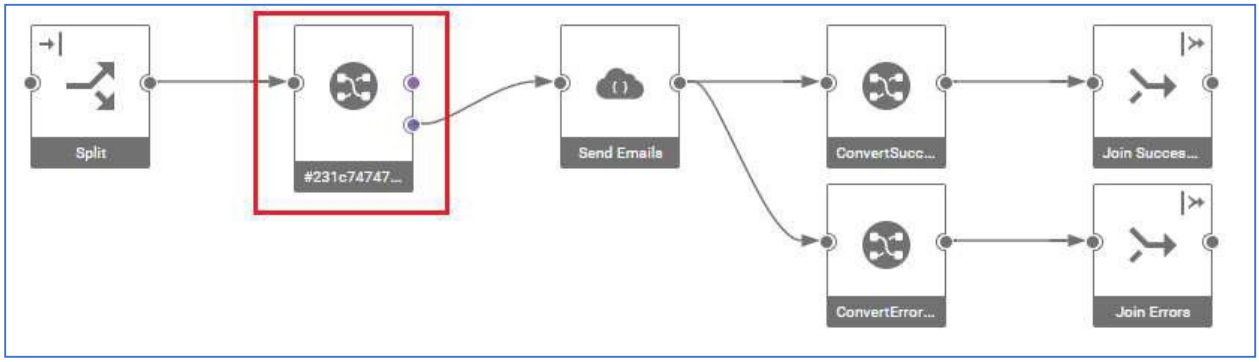
1. Flow variables are set according to the values of action properties
2. Transformation maps are added to the input and output flows if there is a need to handle identity fields, or to convert from one format to another.

Data format conversion is determined by the definition of the input and output terminals. In this demonstration, all of the flows are defined to accept JSON data and to produce CSV data. The testapp application (and Unica applications) are defined to produce and consume CSV data. This means that when these flows are deployed when an action is configured, a map node will be inserted in the flow to perform the conversion from incoming CSV to JSON data.

The placement of this node depends on whether the flow contains a Split node or not. If there is no Split node, then the conversion map is placed at the start of the flow:

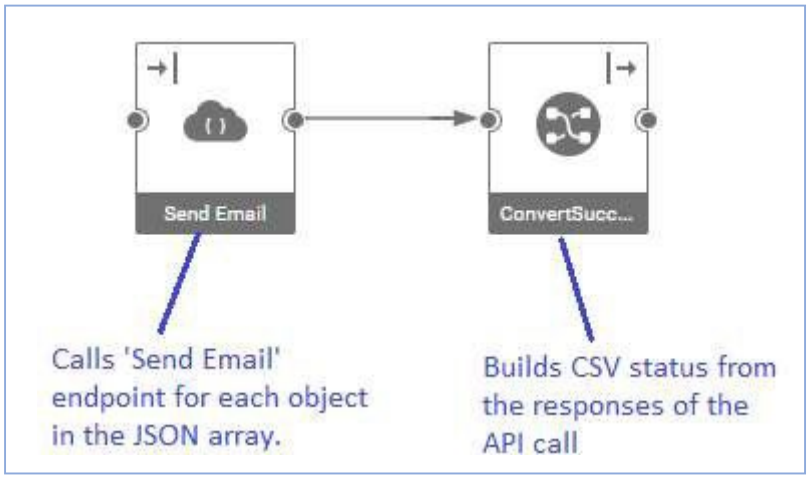


If there is a Split node, then the conversion map is placed after the Split to convert each batch:



3.2.2 Send Email – Basic

The **Send Email – Basic** flow has these 2 nodes:



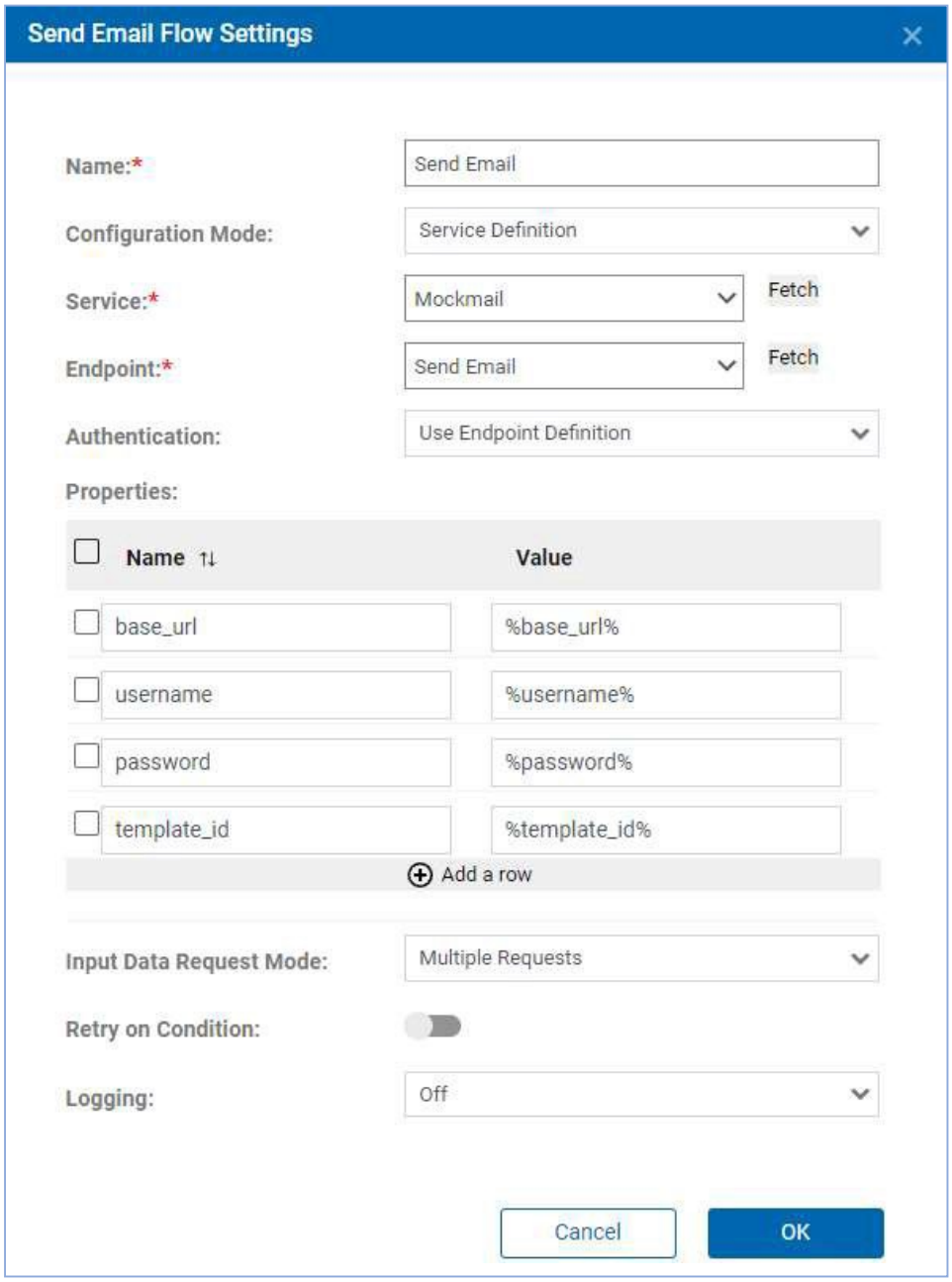
The description of the flow is this:

```
@purpose=run
Basic send email flow
```

The "@purpose=run" is a special property that tells Link to expose this flow as a run operation when creating a connector. If this is missing from the description, the flow will be ignored when creating the connector.

3.2.2.1 REST Client Node

The REST Client node has the following settings:

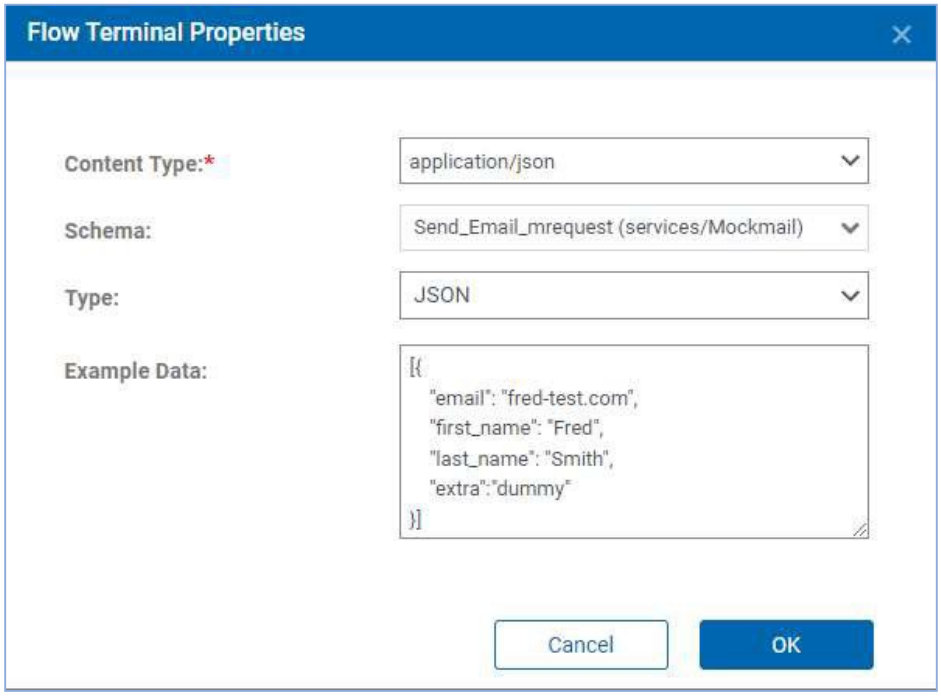


Note:

- The Send Email endpoint from the Mockmail service is selected
- In the properties, the parameters of the endpoint are set to flow variables of the same name (**%base_url%** means get the value from the base_url flow variable). When the connector is generated, its properties will be defined by flow variables discovered in the flows.
- **Input Data Request Mode** is set to **Multiple Requests**. This tells the REST client node to expect a JSON array where each object in the array is the request payload for the API. The API will be called for each object provided in the array.

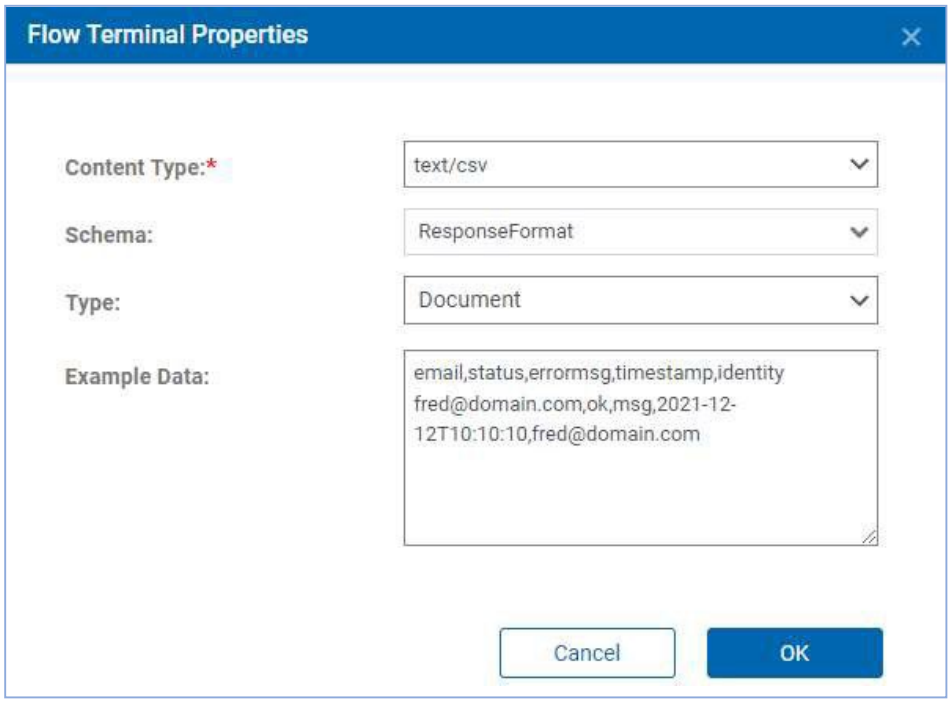
3.2.2.2 Flow Terminals

The flow's input terminal is defined as:



This means that the flow is expecting a JSON array in the format specified by the schema. Note that this schema was generated automatically when the terminal was defined – Link creates it from the request JSON specified in the Send Email endpoint.

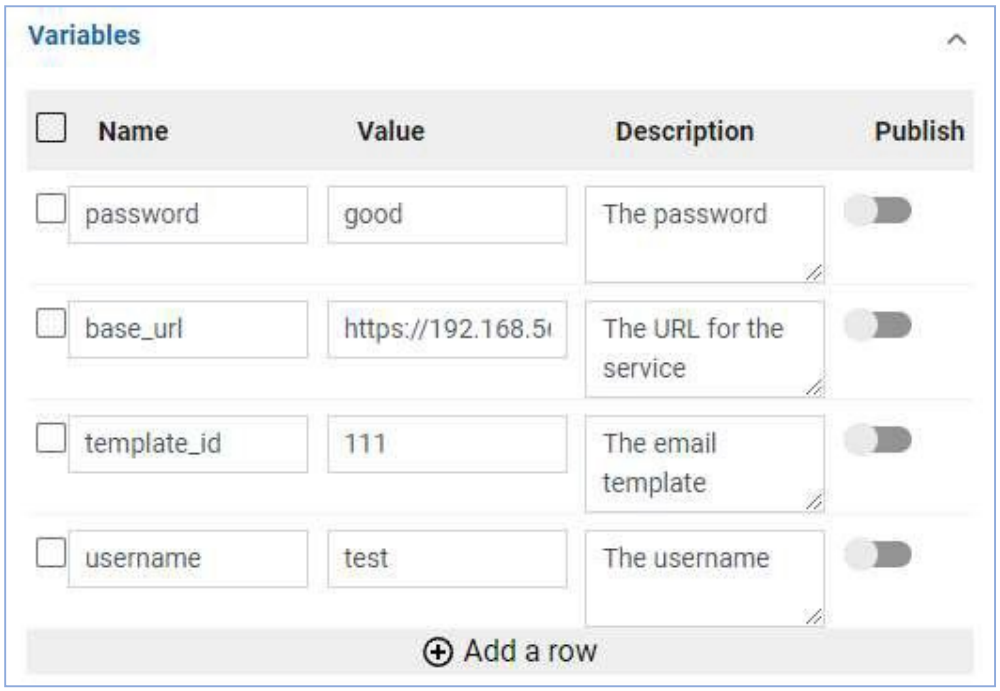
The flow's output terminal is defined as:



This is specifying that the flow will return a CSV format. The schema in this case was obtained by using Link's CSV importer to generate a schema from a sample response CSV.

3.2.2.3 Flow Settings

The flow settings include these flow variables:

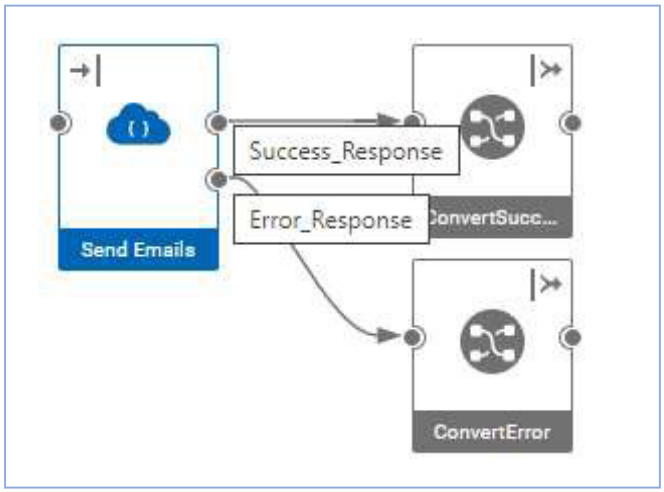


The names of the variables were populated automatically when the settings were opened. The default values and descriptions were manually entered.

These flow variables are used to create the properties of the connector. All the variables of all the flows in the connector are combined. If variables are used in every flow, then they default to connection properties, otherwise they default to action properties.

3.2.3 Send Email – Error Handling

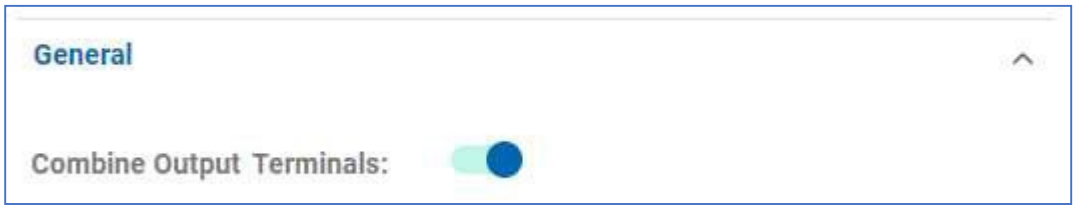
The flow **Send Email – Error Handling** adds handling for failing REST calls.



This flow adds handling of the error_response terminal in the REST Client node. If any API calls are unsuccessful, the responses are sent in a JSON array to this terminal.

A second map **ConvertError** is added to convert from the error response format to the CSV output format.

Note that the icon for the output terminals is showing the combined terminal icon. This is because the flow settings have the Combine Output Terminals property set:



This causes the flow to combine the outputs from multiple nodes to a single output that is returned from the flow. In this flow, the output terminal will return the successful records, following by the error records.

3.2.4 Send Email – Identity

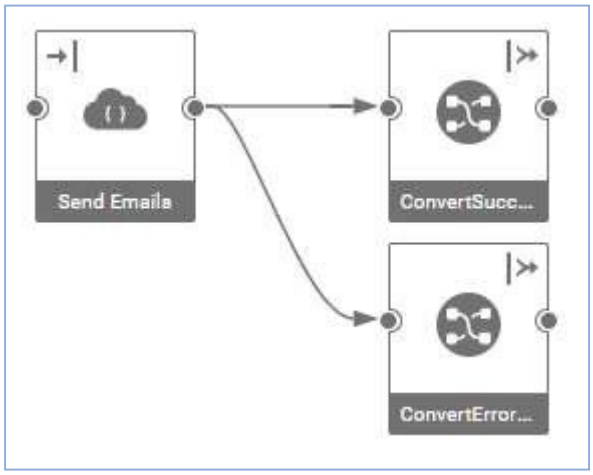
This flow adds handling of identity data to the flow. If the application is providing input data as CSV data, then identity fields are appended to the mapped fields. For example, the expected data might be:

```
email,first_name,last_name,gender,product
name1@test.comFirst1,Last1,M,Item1
```

but an additional identifier is required to uniquely identify the record. This is added after the expected fields:

```
email,first_name,last_name,gender,product,id
name1@test.comFirst1,Last1,M,Item1,0001
```

The flow exploits identity handling to be able to return the identity value with the response records produced by the flow



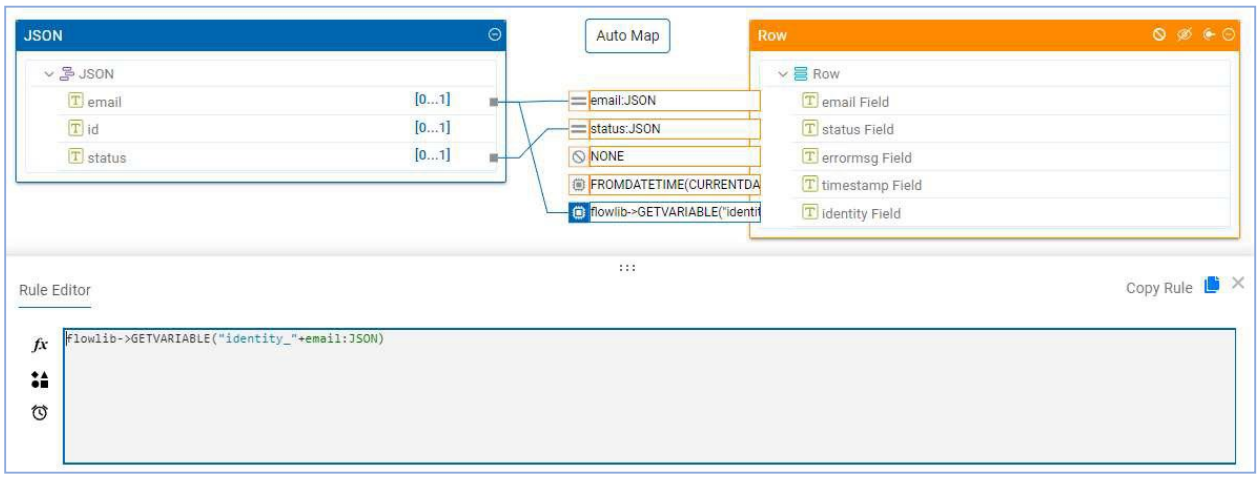
The flow is the same as Send Email – Error Handling but calls different maps to handle the identity logic. It also has an additional property in the flow description:

```
@purpose=run
@identity=variable
Send emails passing through identity element
```

@identity=variable tells Link to store identity values in flow variables. Other possible values for **@identity** are **ignore** and **cache**, which tell Link to discard the identity value or to store in the cache rather than flow variables.

Additionally, **@identityKey=<field>** could have been specified. This specifies which incoming field to use as the key when storing identity values when the flow runs. By default, the first field is used, which is **email** in these flows.

The functional maps f_eachRecordWithIdentity and f_EachErrorWithIdentity get the value of the identity value and set it in the output CSV data:



The rule for the identity field in the output is:

```
flowlib->GETVARIABLE("identity_"+email:JSON)
```

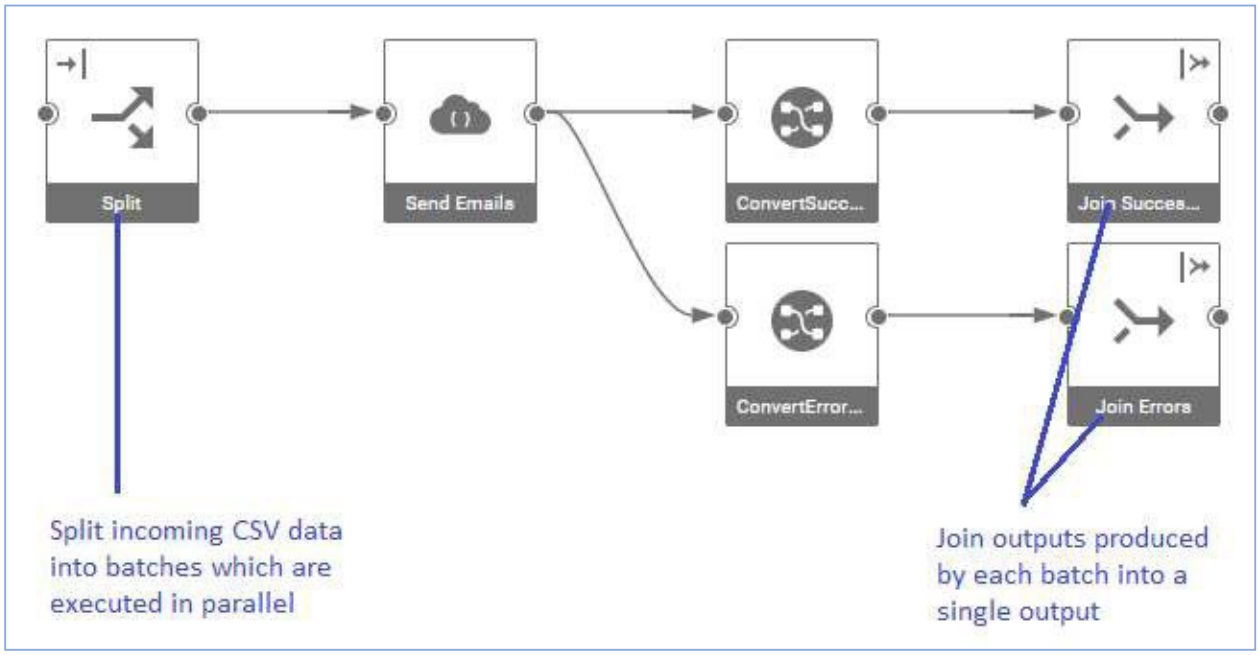
The @identity property tells the deployment how to handle the identity field(s). In this case, because @identity was set to “variable”, the flow variable “identity_<email>” is set by the map that is automatically added to the flow when an action is deployed. When the flow runs, a set of flow variables will be set holding the value of the identity field passed in the input data. The value is then retrieved by this rule and output in the resultant CSV.

3.2.5 Send Email – Split

While the **Send Email – Identity** flow has all the desired functionality (provides error handling and support for identity fields), it will perform poorly if there are many input records. The flow will convert the entire CSV data fed to it to a JSON array of objects (via the generated map that is added when an action is deployed), which the REST client node processes, sequentially calling the API for each record.

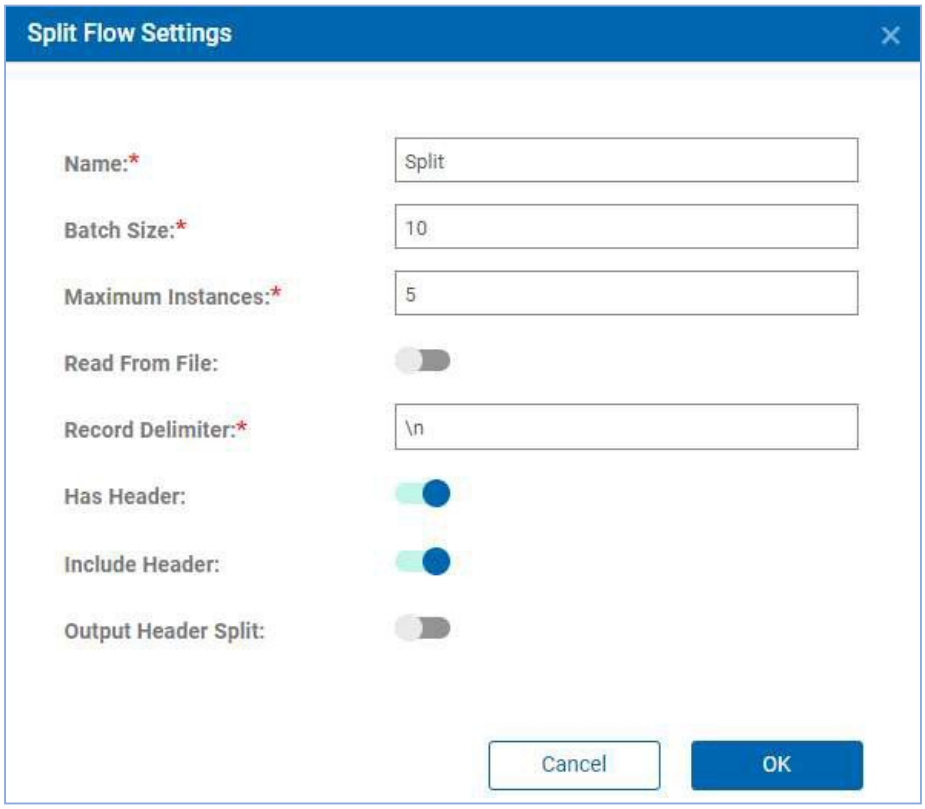
This is probably acceptable for small data volumes but will not scale if thousands or millions of records are provided in the input data.

This is solved by using the Split and Join nodes to break up the large data set into smaller batches which can be executed in parallel:



3.2.5.1 Split Node

The settings of the Split node are:

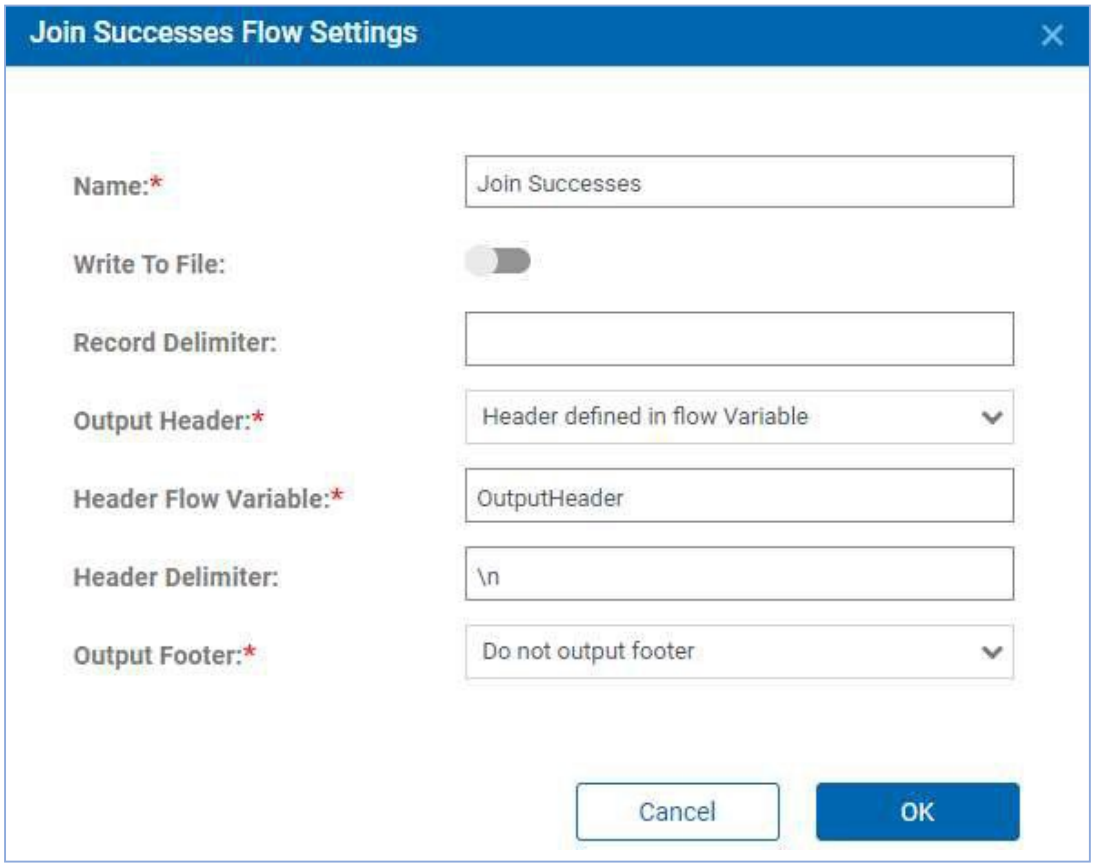


The settings have the following meanings:

- **Batch Size** – the number of records in each batch. In production, this number would be set to a much larger value than 10.
- **Maximum Instances** – the maximum number of parallel threads that will started to process the batches. This setting is important in not overloading the server and breaching API rate limits.
- **Record Delimiter** – the character to split the incoming data on
- **Has Header** – whether the incoming data has a header row
- **Include Header** – whether the header row should be sent to each batch

3.2.5.2 Join Nodes

The Join nodes have the following properties:

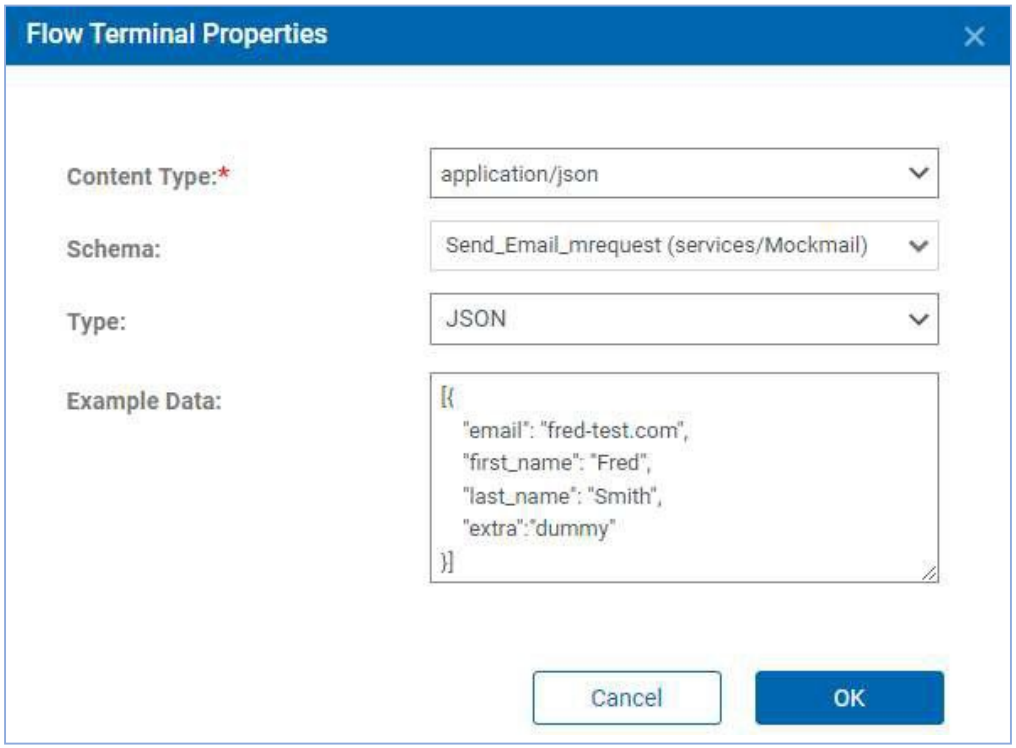


The Join node appends the results of batches created by the Split node. These settings instruct the node to prepend the output with a header that is defined by the flow variable “OutputHeader”. This flow variable is set in the ConvertSuccessWithIdentity and ConvertErrorWithIdentity maps:

It uses the value of the “identity_fields” flow variable which is set by the generated map.

3.2.5.3 Flow Terminals

Although the properties of the Split node indicate CSV input, it is necessary to define the input terminal to indicate that the node downstream of the Split node is expecting JSON data.



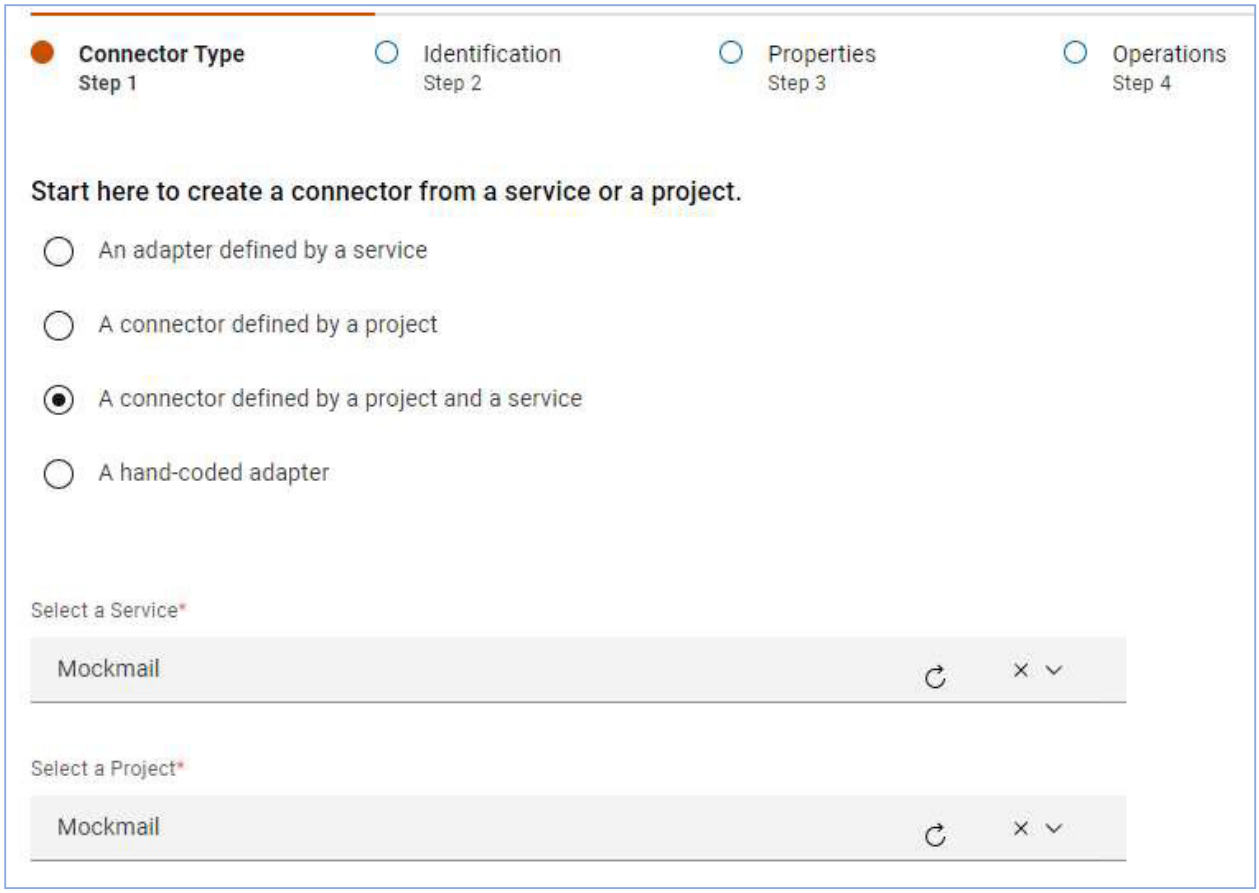
Link uses this information when an action is deployed to determine that a transformation map must be placed after the Split node to convert from the incoming CSV data to the JSON form specified by the input terminal.

4 Creating a Connector

To create a connector for the Mockmail service and flows, follow the steps outlined in this section.

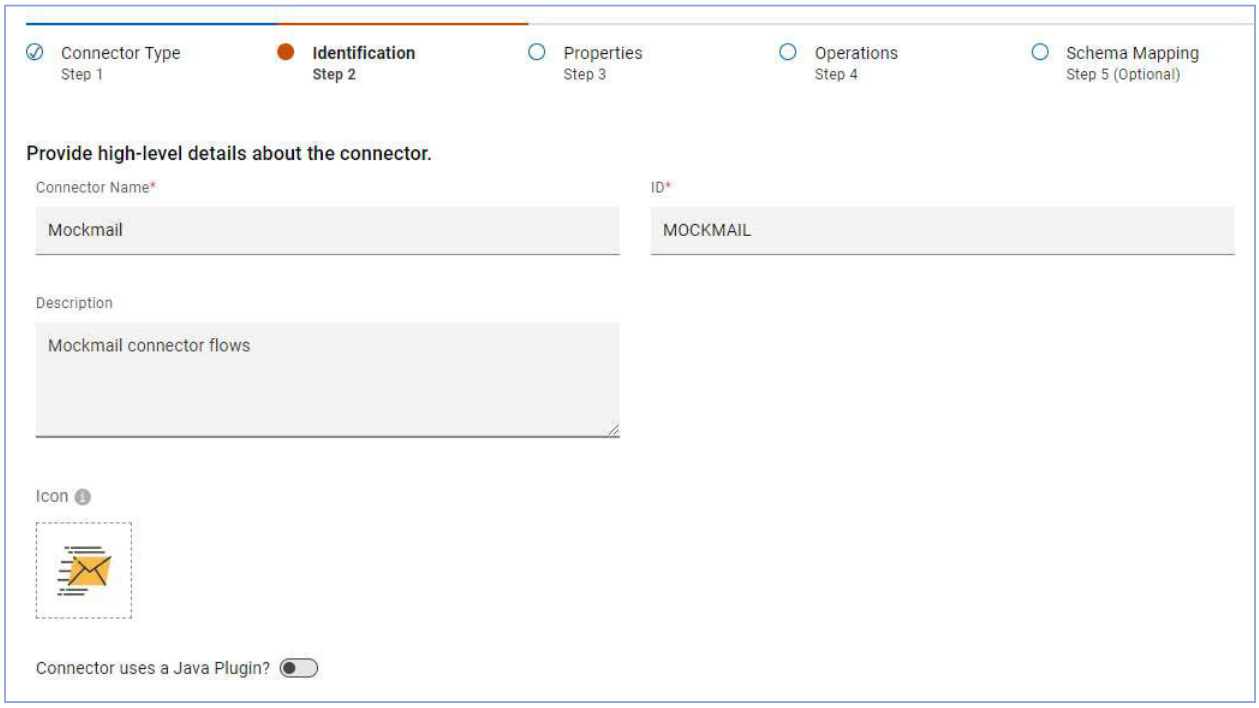
From the Connectors page, select Create New Connector. Select the third option (connector defined by a project and a service). Select the Mockmail service and the Mockmail project:

Note: Refer following link to access about connectors and how they are used:
https://help.hcltechsw.com/hcllink/1.1.4/connectors/concepts/c_connectors_top_lvl.html



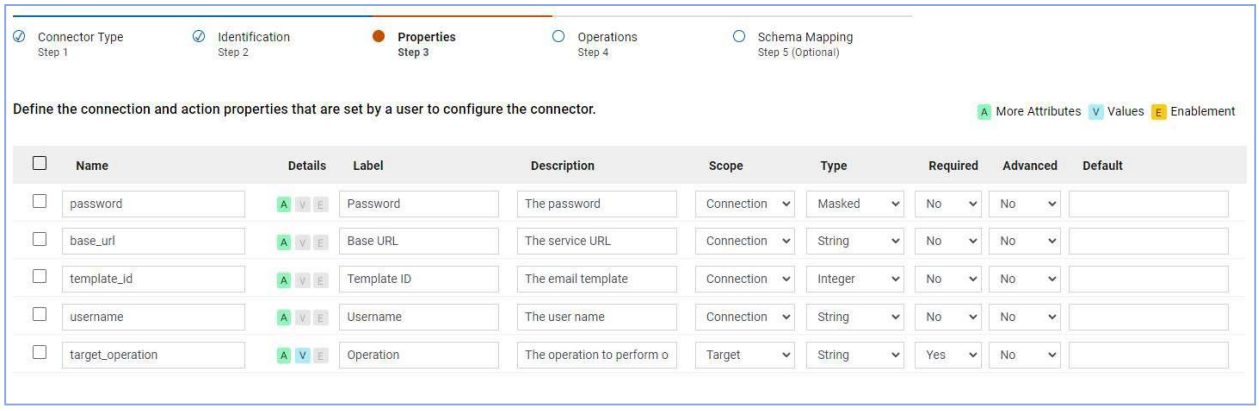
Click next to advance.

In the Identification page, the name and ID will default to Mockmail/MOCKMAIL which derives from the project name. From this page, select an SVG icon to be associated with the connector. A suitable icon is provided with this development kit:



Click next to advance.

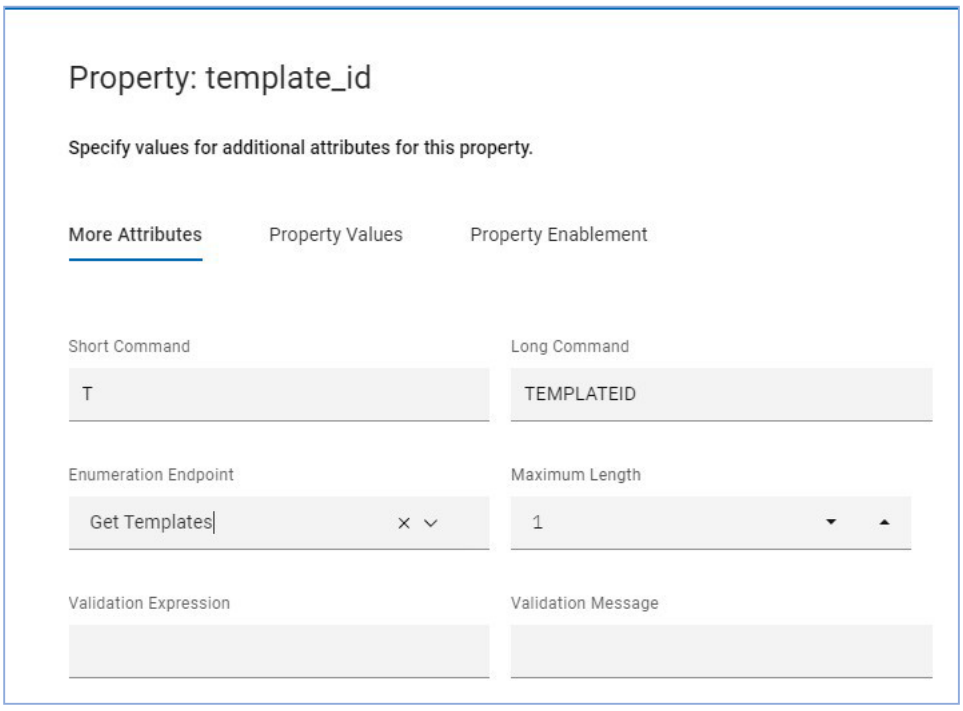
The project and service are examined, and a set of properties are created from the flow variable definitions in the flows:



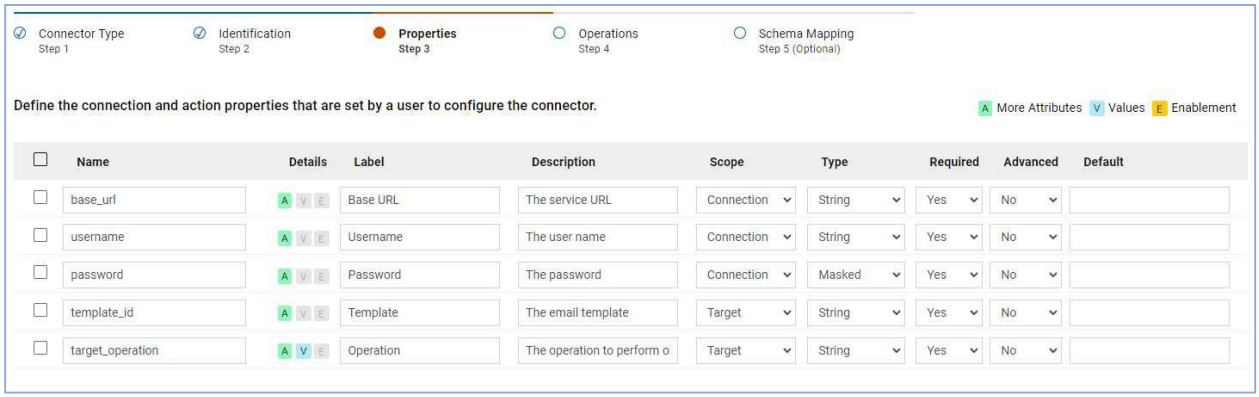
The generated properties have some attributes that need to be adjusted:

1. Improve or add different wording for label and description as necessary. The label is generated from the flow variable name, but an alternate label may be desired.
2. Set properties as required. By default, all properties are optional. To make a property mandatory, change the value in the Required column to Yes.
3. Reorder the properties. Properties are displayed in the user interface in the order they are listed. Properties can be reordered by dragging them in the table.

4. template_id is given Connection scope because it appears in each flow. But more likely, one would want to change the value of the template for each action, so the scope should be changed from Connection to Target
5. template_id is given the type Integer. This is because the default value of the flow variable is an integer, but we need to change this to a String so that we can define an enumeration for this property.
6. Define an enumeration for the template_id property. The “Get Templates” API provides a list of available templates, and we wish to call this API when an action is defined to populate the list. To do so, click on ‘More Attributes’ for the template_id property and select Get Templates from the Enumeration Endpoint attribute. The list of available endpoints listed here is determined by listing those endpoints that define an enumeration.

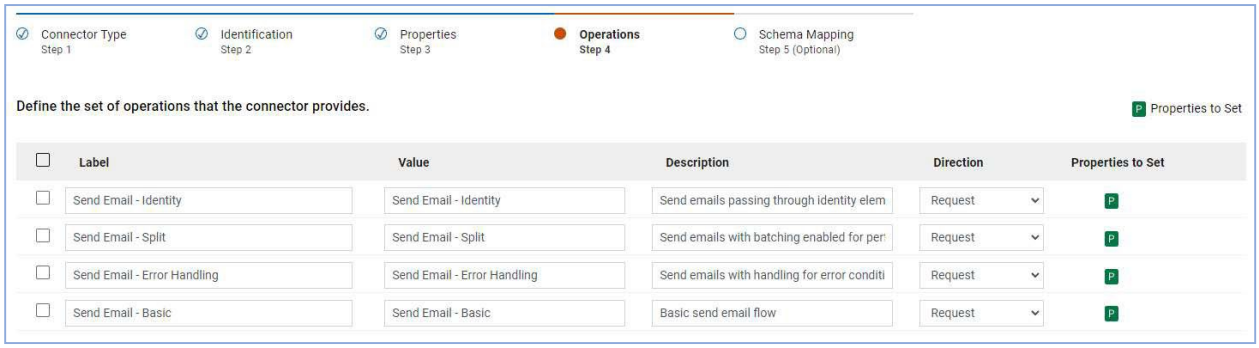


After modifying the property definitions, they look like this:



Click Next to advance.

A list of operations will be shown:

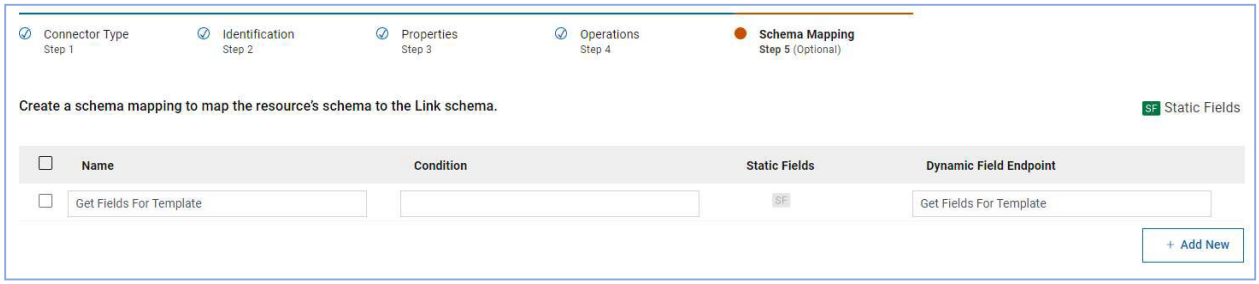


These operations correspond to the flows that were defined in the project. The labels may be modified, but the values should not be changed, since these must match the name of the flows in the connector project.

Whether or not operations are displayed when a connector is utilized depends on the application that is embedding Link, and how that application is configured to interact with Link's user interface.

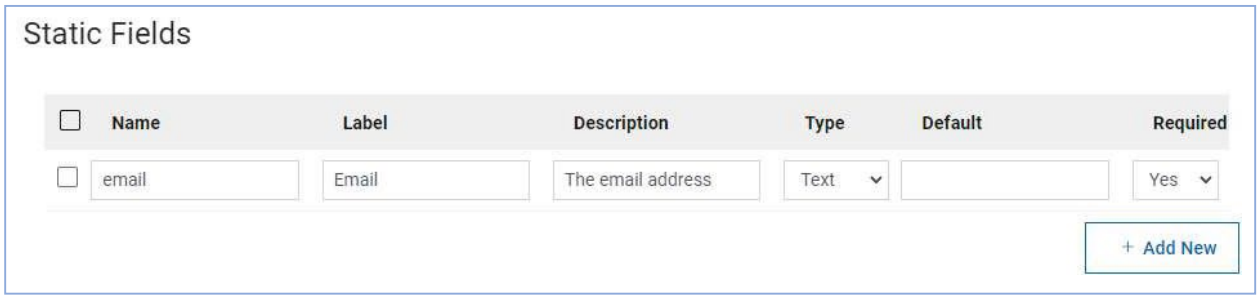
Click Next to advance.

The Schema Mapping table will be shown with a single entry:



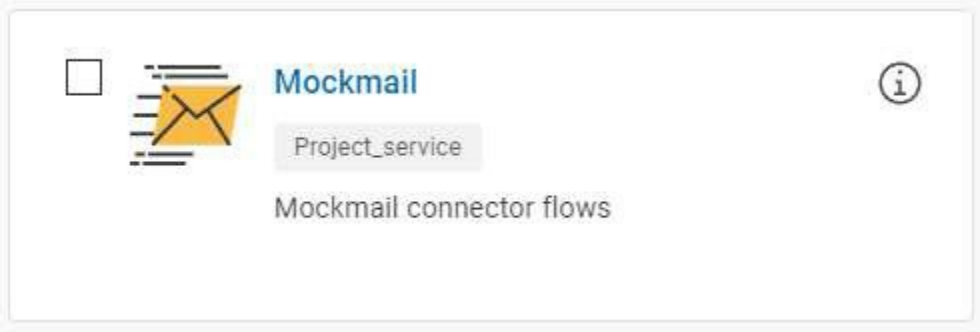
This entry was created because there is a single endpoint in the Mockmail service that defined a field mapping (Get Fields for Template). This API provides a list of fields that a particular email template requires, such as first or last name.

The connector also needs the email column to be passed to the Send Email API, so this should be added as a static field. Click on the SF icon to open the Static Fields table and add an entry:

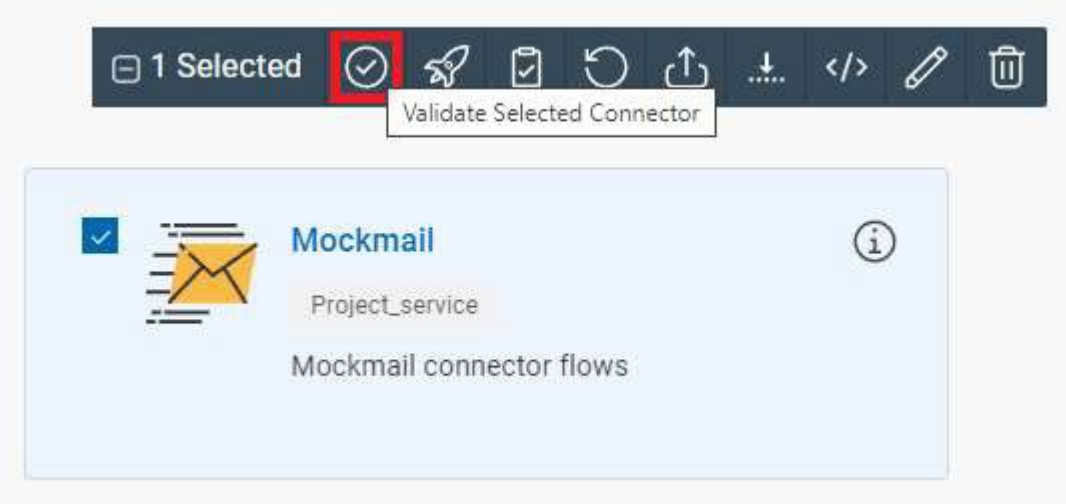


Finally, click Save.

The connector has been created, and appears in the list of connectors:



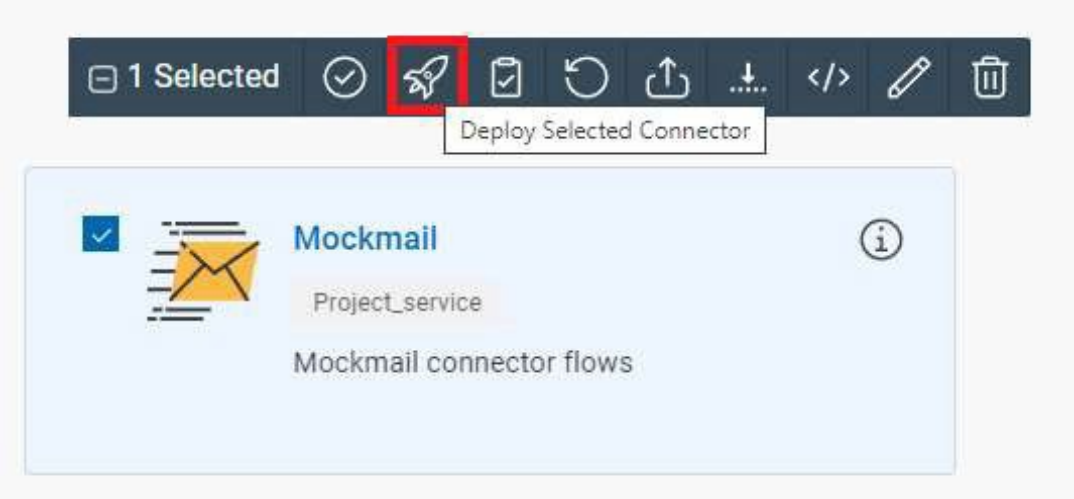
The connector can be validated by selecting the Validate option from the toolbar:



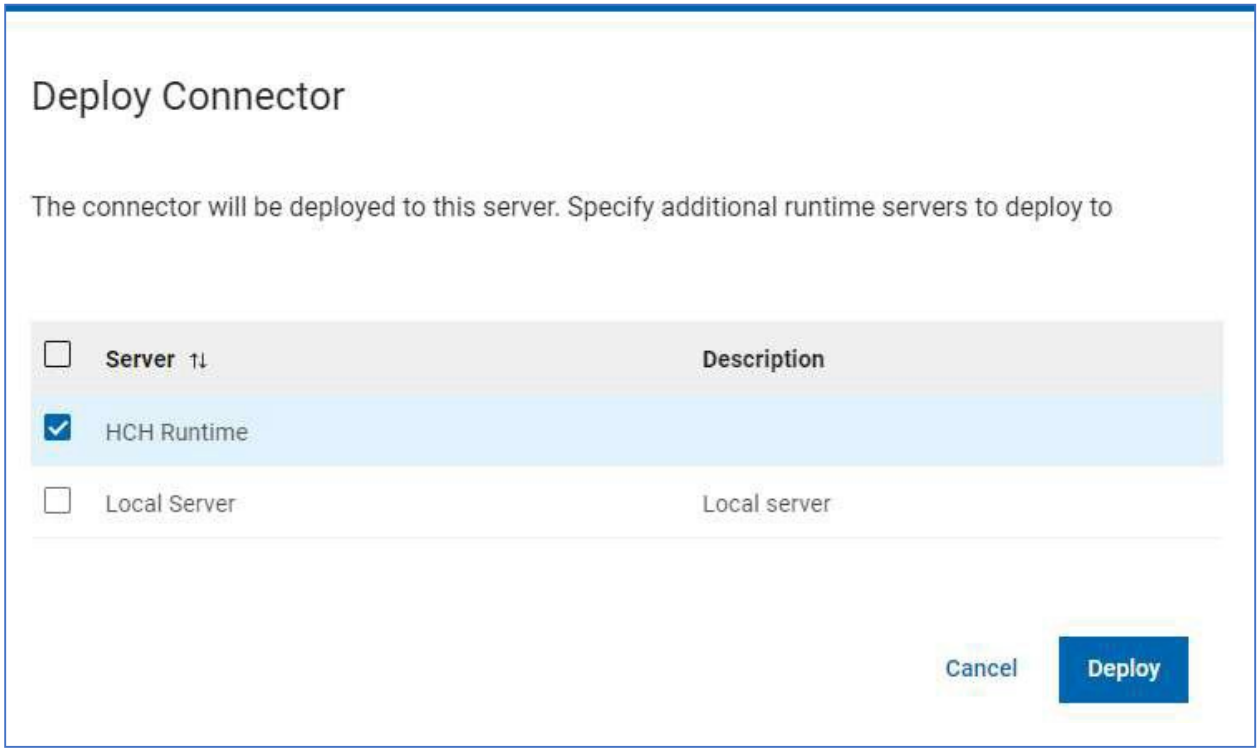
This will ensure that referenced services and endpoints exist, and that the definitions of properties and operations are consistent.

5 Testing the Connector

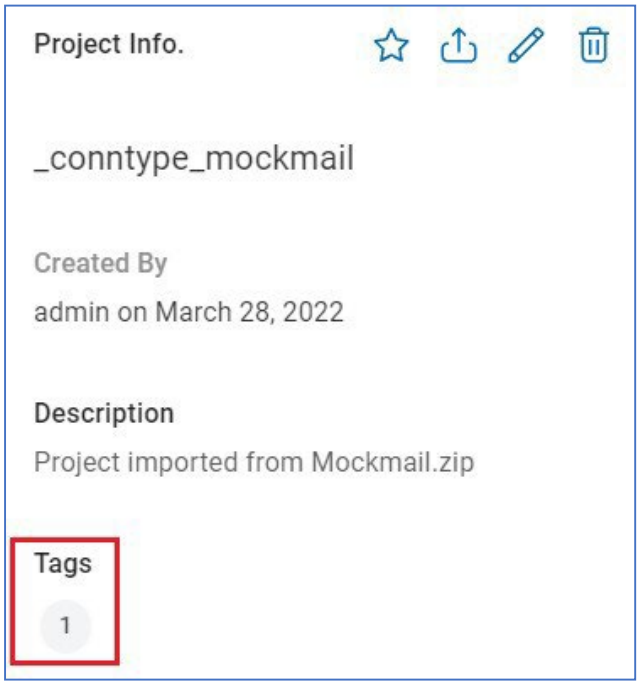
The connector needs to be deployed to the server so that it can be used. To test the connector, it must first be deployed to the server. Select the connector and click Deploy on the toolbar to deploy the connector.



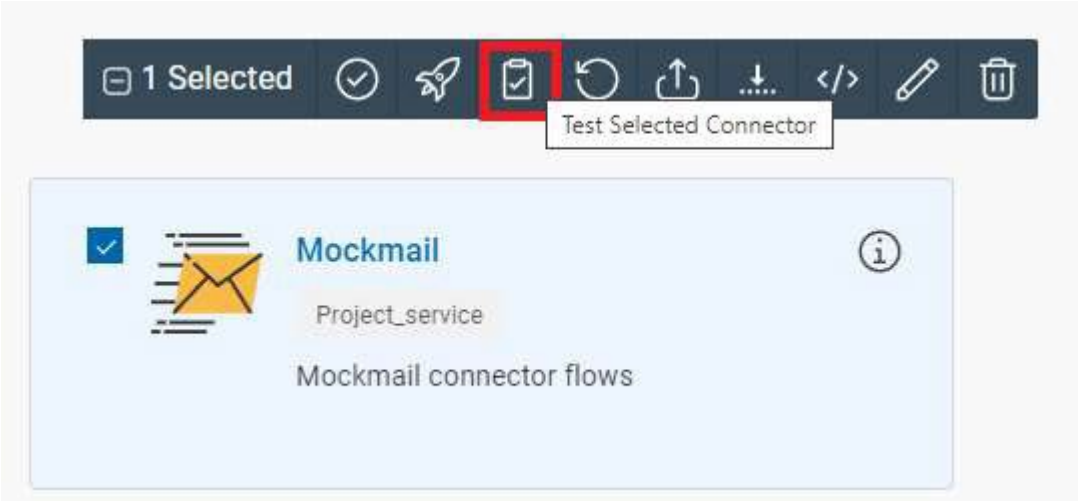
By default, the connector will be deployed to the local “design” server, but not to a runtime server. If a runtime server is defined in your environment, select that server in the list:



After the connector has been deployed, a project will be created in Link named _conntype_<ID>. This project is a copy of the connector project. Whenever the connector is modified the version number of the connector is incremented. The current version of the connector can be seen in the tags of the imported project:



Next, click on the test button to launch the Link embedded user interface:



This opens a new browser tab with the URL:

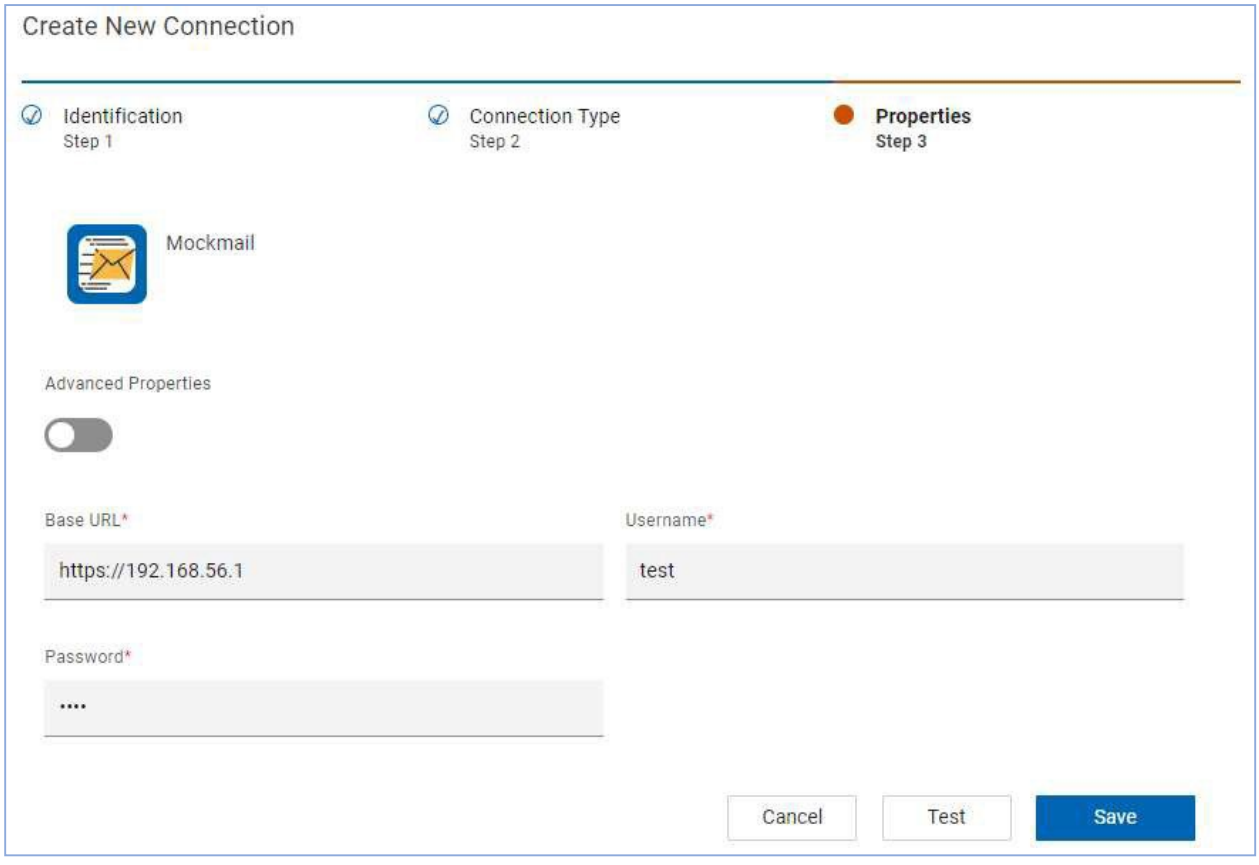
`https://localhost/linkWidget/_app_testapp/allConnections`

It is a good idea to bookmark this URL so you can easily access it in the future.

5.1 Create a Connection

Click on Create New Connection to create a connection that uses the Mockmail connector. The newly created connector will be available as a connection type:

Populate the connection properties to point to the design server, and specify username/password as test/good.



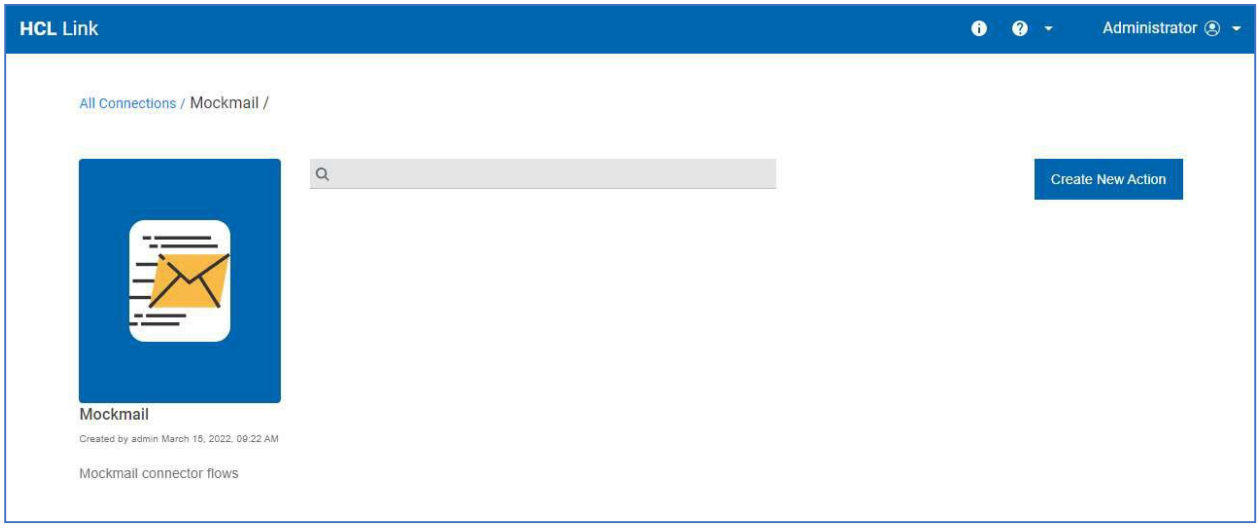
The Test button is shown because one of the endpoints in the Mockmail service (Ping) has the Connection Test Endpoint switch enabled:

This endpoint is called when the Test button is clicked in the connection user interface.

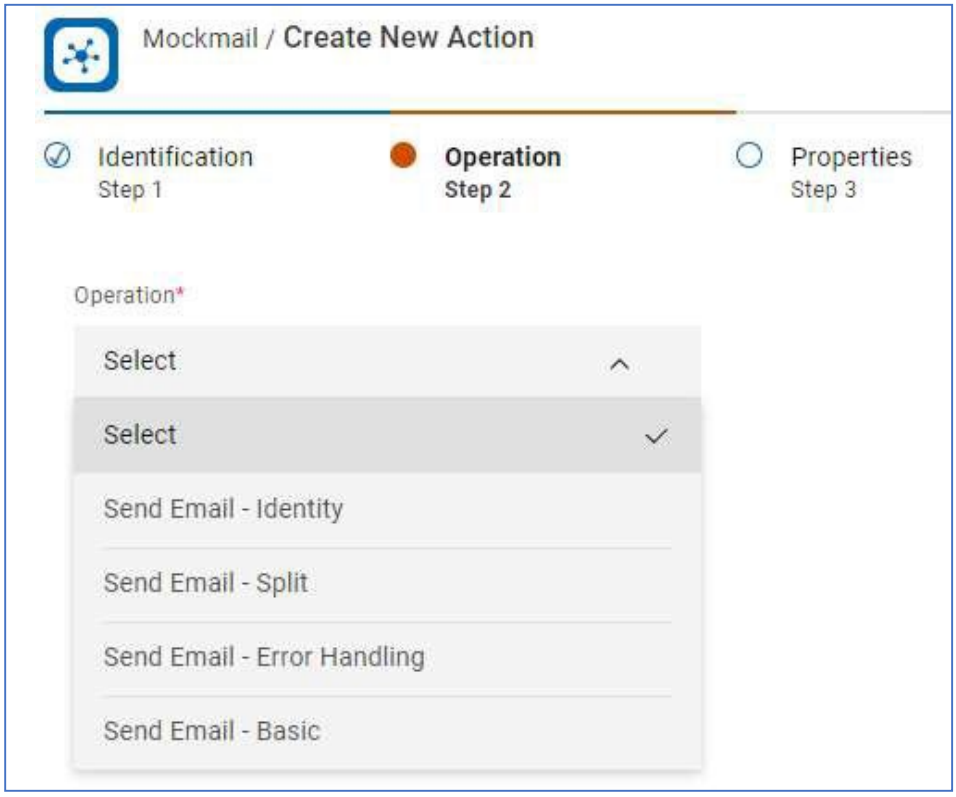
After populating the connection properties, click Save to save the connection.

5.2 Create an Action

From the connection page, click on the connection created above to open the action list for the connection. Click Create New Action to define an action for Mockmail.

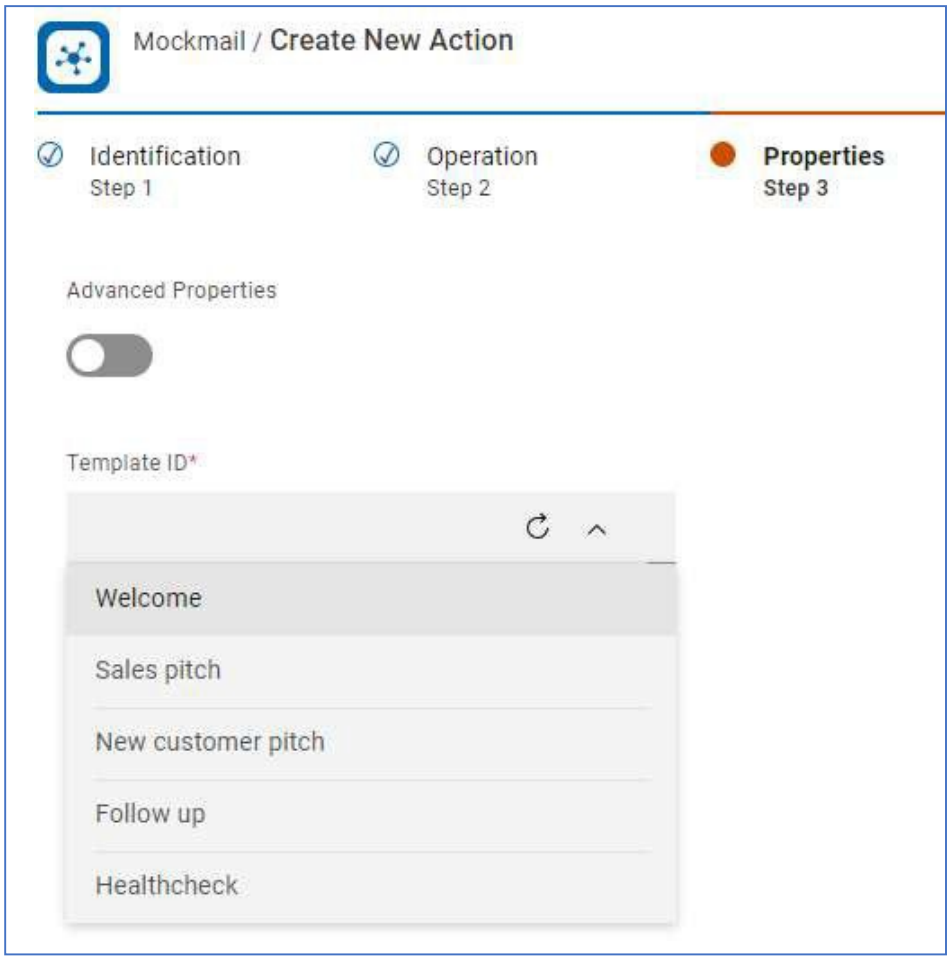


Name the action and click next to show the list of operations:

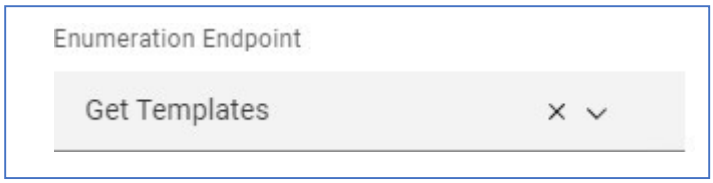


The operations were defined when creating the connector. They correspond to the flows that were provided by the Mockmail project. For this action, select “Send Email – Error Handling” operation and click Next.

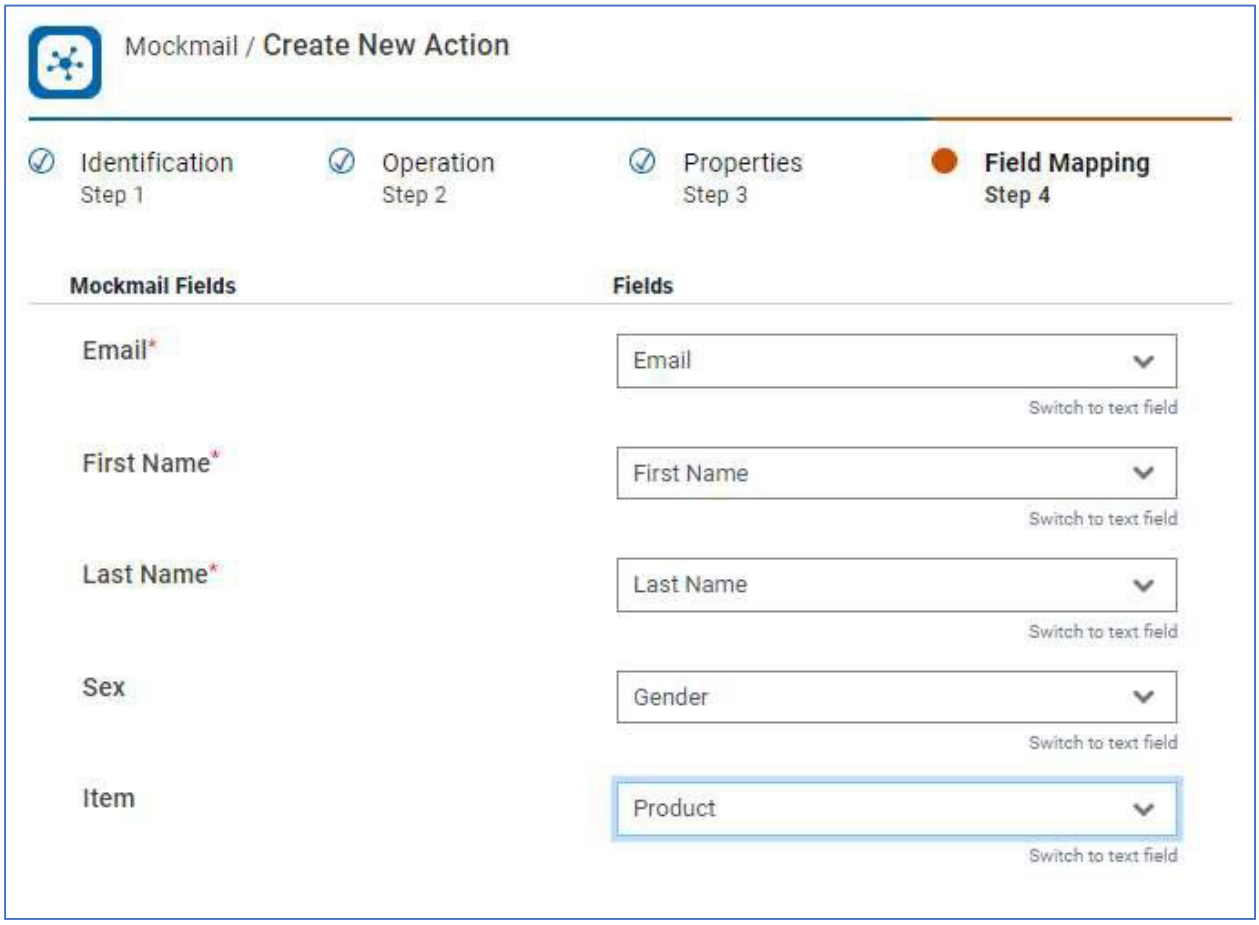
In the Properties tab, the Template ID property is displayed:



The list of values for the Template ID comes from the Get Templates endpoint in the Mockmail service. The property definition of Template ID is configured to call this endpoint to get the enumerated values:



Click next to advance to the field mapping step:



To populate this screen the UI:

- utilized the schema mapping defined in the connector. This provided the Email static field, and the other fields (First Name, Last Name, Sex and Item) came from the selected template.

- the drop-down lists in the Fields were populated from the Link server's dummy application. If the connector were deployed for some other embedded application, then that application would be called to provide the available fields.

Finally, click save. This saves the action and also deploys it to the runtime server. Deploying an action does the following:

- Copies the connector's artifacts to the _app_testapp project (if they have not yet been copied)
- Clones the template flow, and modifies it to add in necessary maps to perform the mappings defined in the field mapping steps, and/or to convert data from the connector's data type (e.g. CSV or JSON) to the application's data type (CSV in the case of testapp).
- Creates a package containing the new flow and related maps and files
- Deploys the package to the runtime server

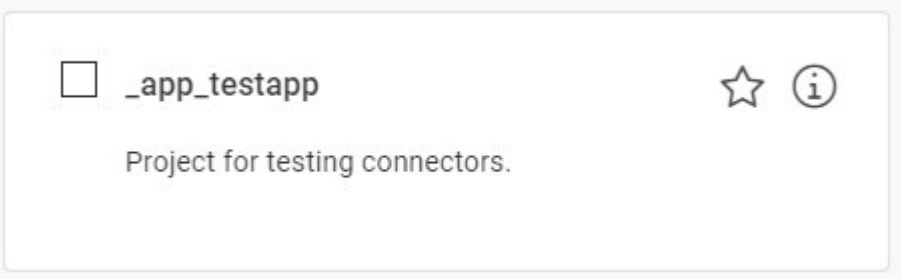
5.3 Running the Action

There are 2 ways the generated action flow can be tested:

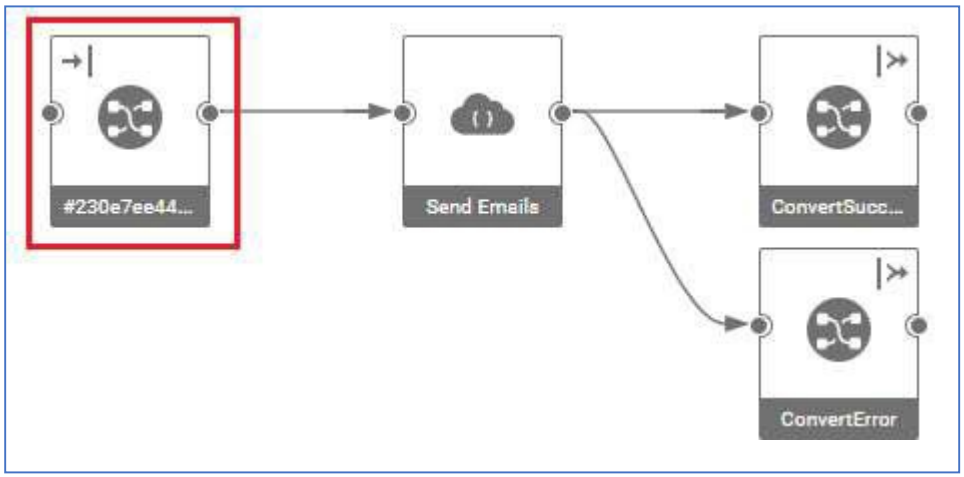
- Running it in the _app_testapp project
- Running it from the Swagger page of the runtime server

5.3.1 Running from _app_testapp Project

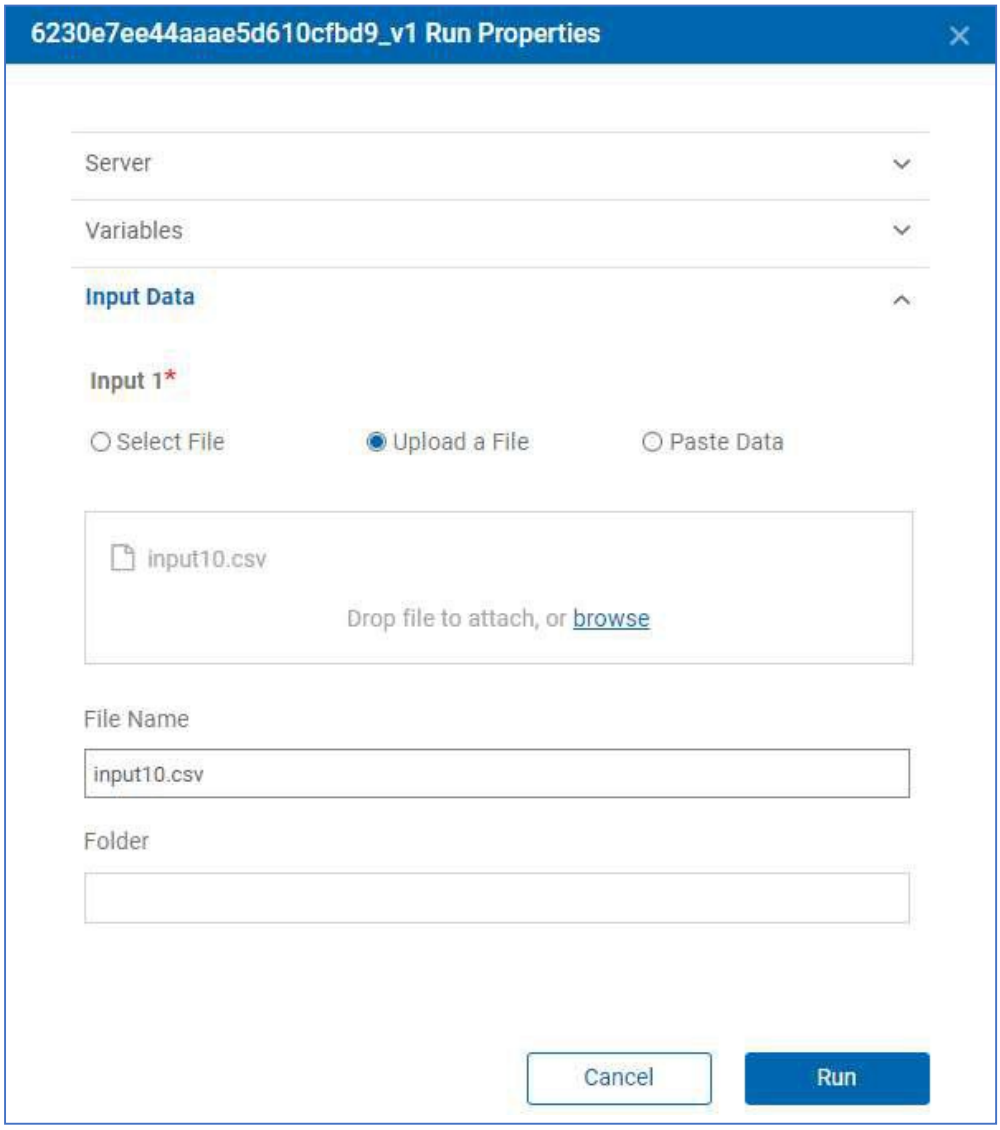
To run the flow in the _app_testapp project, go to the Projects tab in Link and click on _app_testapp project:



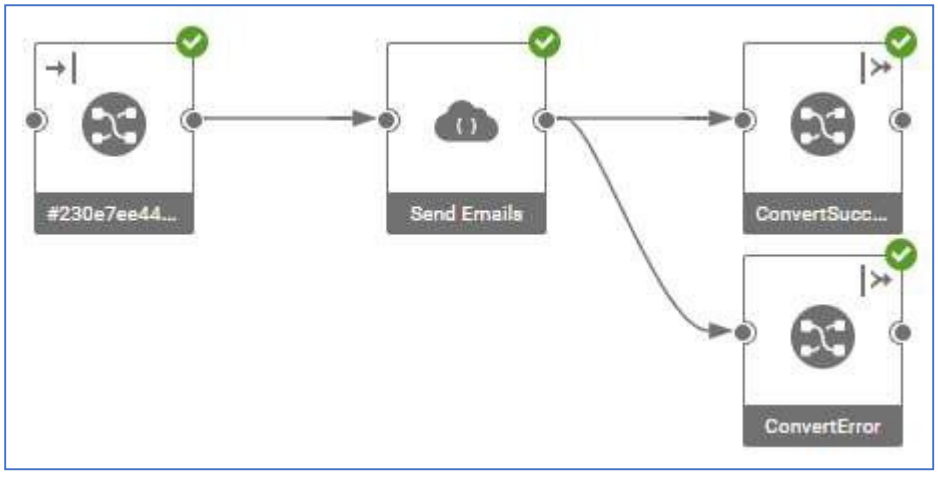
In the flow list the most recently created flow will be the flow corresponding to the action. This flow will be a copy of the template flow selected in the action, but with an additional map added at the front to convert the incoming data according to the mapping specification specified in the action:



To run the flow, click on the Run button and in the input data select the input10.csv file that is provided in the files directory. This file is appropriate if you selected the Welcome template and mapped the fields as shown above in the field mapping screenshot:



After the flow runs the flow design includes markers to show the successful state of each node:



The data sent along each connection can be viewed by right-clicking on the link and selecting View Data. The final data output by the flow can be viewed by clicking on either of the final nodes, and in the structure diagram selecting either View Data or Download Data:

The result data for the input file input10.csv looks like this:

```
email,status,errormsg,timestamp,identity
name1@test.comFirst1,undelivered,,2022-03-16T09:58:53,
name2@test.comFirst2,sent,,2022-03-16T09:58:53,
name4@test.comFirst4,sent,,2022-03-16T09:58:53,
name5@test.comFirst5,sent,,2022-03-16T09:58:53,
name6@test.comFirst6,sent,,2022-03-16T09:58:53,
name7@test.comFirst7,undelivered,,2022-03-16T09:58:53,
name8@test.comFirst8,undelivered,,2022-03-16T09:58:53,
name3-test.comFirst3,failed,email is not valid,2022-03-16T09:58:53,
name9-test.comFirst9,failed,email is not valid,2022-03-16T09:58:53,
```

Note that the results combine the results from the success and error paths in the flow. This is because the Combine Output Terminals setting is enabled in the flow. Note also that the identity value is not set. This is because we selected the “Send Email – Error Handling” flow which does not provide support for handling identity values in the flow.

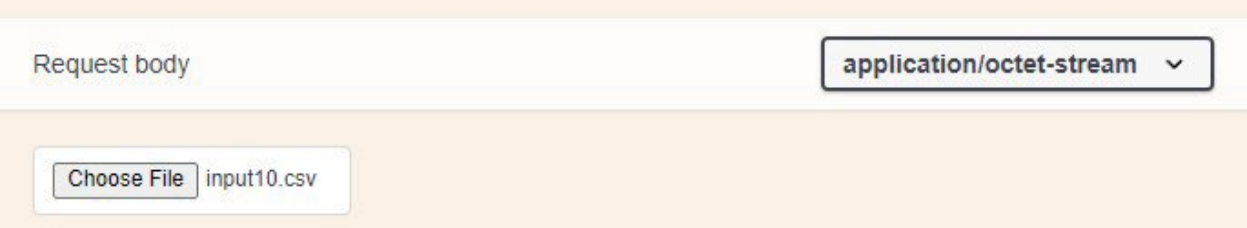
5.3.2 Running from the Runtime Server

The runtime server provides a Swagger page that documents all of the endpoints deployed to the server. This can be accessed via this URL:

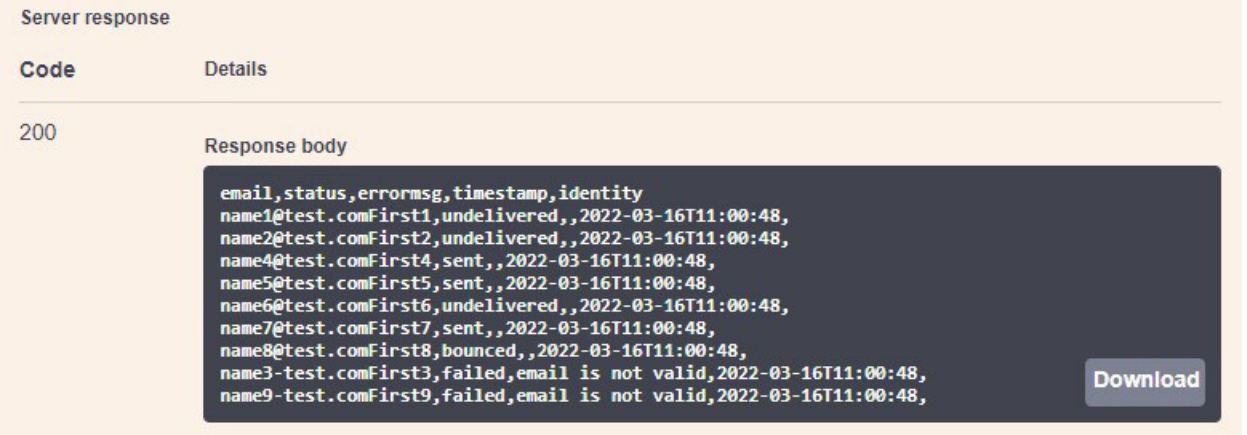
`http://<hostname>:8080/hip-rest/api-docs?url=/hip-rest/v2/docs`

After authorizing, find the endpoint in the deployed endpoints:

Click on **Try it out** button to run the endpoint. To provide the input file, change the request content type to application/octet-stream:



After clicking on **Execute** the flow is run and the output is shown in the Swagger page:



6 Identity Field Handling

Identity fields are additional fields that are sent from the application in addition to the mapped fields. They are used to provide some additional contextual information about a record. For example, the identity could be some internal ID number that is used by the Unica application.

These identity fields are not processed in the flow, but need to be passed through and returned in the output of the flow, associated with the initial record.

For example, data from Unica might be:

```
email,first_name,last_name,internalId1,internalId2
fred@domain.com,Fred,Quimby,A01,B01
```

```
sue@abc.com,Sue,Smith,A45,B67
```

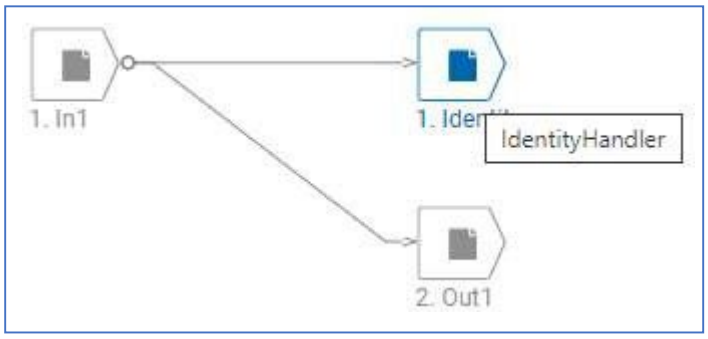
In this example internalaId1 and internalId2 collectively identify the record. In the response from the flow, these need to be returned back to Unica:

```
email,status,errormsg,timestamp,internalId1,internalId2
fred@domain.com,sent,,2022-01-01T10:10:10,A01,B01
sue@abc.com,sent,,2022-01-01T10:10:11,A45,B67
```

To simplify the logic in flows, and to avoid having to pass these identity fields through every link a flow, Link uses flow variables or cache variables to temporarily store the values. When a flow is deployed the generated map processes the identity fields according to the **@identity** deployment property. This can be set to ignore, variable, or cache. Select the option accordingly:

- If the output does not need to return identity fields, then specify 'ignore'
- If the output of a results flow needs to return identity fields, then specify 'cache'. Cache variables are stored on the server and can be accessed by multiple flows.
- If the output of the run flow needs to return identity fields, then specify 'variable'. The variables are available within the scope of the flow instance, but not accessible by other flow instances.

The generated map has an output card IdentityHandler:

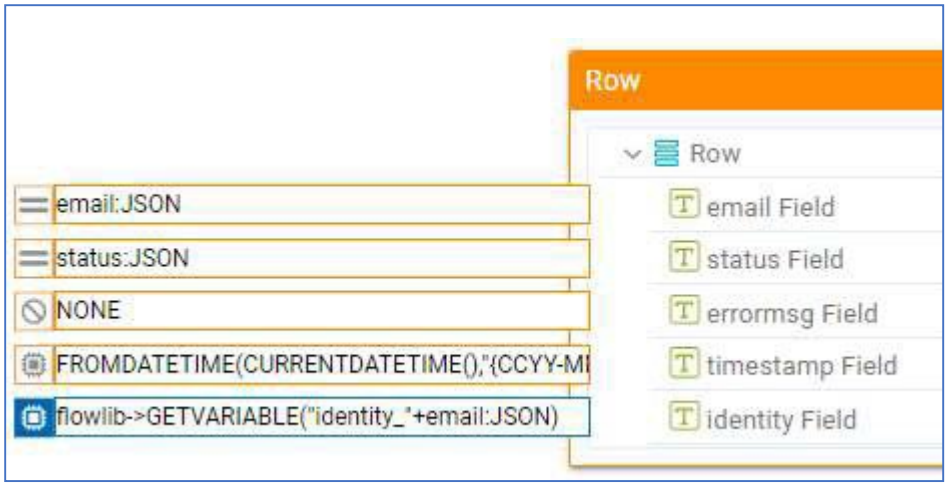


This card populates flow variables or cache variables. If @identity is set to 'variable', the following flow variables are populated:

Flow Variable	Value
field_names	A comma-separated list of the input fields. Using the example from above, this would be: "email,first_name,last_name"
identity_fields	A comma-separated list of the identity fields. Using the example from above, this would be: "internalId1,internalId2"
identity_<key>	For each input record, a flow variable creating the identity key is created, where the value is the value of the identity fields. Using the example from above 2 flow variables would be set: identity_fred@domain.com with value "A01,B01" and

	`identity_sue@abc.com` with value "`A45,B67`"

These flow variables can then be used in the flow's output maps when building response CSV.



If the identity is written to the cache rather than flow variables, then the ID of the flow instance is included in the key names. The names of the variables in the cache are:

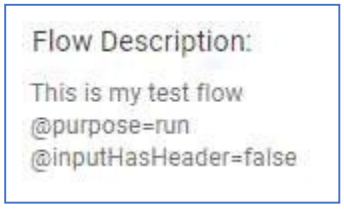
- %_FLOWUUID_%_field_names
- %_FLOWUUID_%_identity_fields
- %_FLOWUUID_%_identity_<key>

The map which builds the output and references these cache variables should always include "%_FLOWUUID_%" to retrieve the variables from the corresponding flow run.

By default, the first field in the incoming data is used as the key in the identity values. If a different key is required, specify the key name using the @identityKey flow deployment property.

7 Flow Deployment Properties

There are a number of deployment properties that can be embedded in the description of a flow that determines the behavior of flow generation during deployment of an action. These properties must be specified in the flow description, each on a new line within the description. For example:



This is the set of properties that can be set:

Parameter	Values	Description
@purpose	run results	Identify flows that implement actions you wish to expose in the connector by tagging

		them with @purpose. Flows that have purpose “run” become operations in the connector. Flows that purpose “results” are run periodically to gather the status of an action.
@identity	**ignore** – Discard identity fields **cache** – Store identity fields in the cache **variable** – Store identity field in a flow variable	Determines what to do with identity fields. This is only supported for CSV data. When storing to the cache or variable, the key will be: `%_FLOWUUID_%_<key-value>` Where key-value is the value of the specified key (see @identityKey)
@identityKey	The name of the field that contains the identity key (e.g. email, phone).	If not specified, then the first column will be used as the key
@inputHasHeader	true | false	The connector is expecting a header row in the CSV data it receives on input
@inputDelimitedDelimiter		For CSV data, the delimiter is a comma. Specify an alternate delimiter if the connector expects something else.
@inputDelimitedTerminator		For CSV data, the row terminator is <CR><LF>. Specify an alternate row terminator if the connector expects something else.
@inputDelimitedRelease		The release character for CSV data for input data
@inputHasFixedFields	true | false	If the input expects a fixed number of fields, set this property to true. The generated map will then always provide this number of fields, regardless of how many fields were mapped. The number of fields that are provided to the connector is determined by the schema of the flow’s input terminal.
@outputHasHeader	true | false	Whether output CSV data is producing a header or not. The default is true.
@outputDelimitedDelimiter		For CSV data, the delimiter is a comma. Specify an alternate delimiter if the connector is producing something else.
@outputDelimitedTerminator		For CSV data, the row terminator is <CR><LF>. Specify an alternate row terminator if the connector is producing something else.

@outputDelimitedRelease		The release character for CSV data for output data

8 Flow Requirements

In most cases, flows should be created to implement the connector's actions.

Maps and other nodes are used in flows to perform complex operations. Flows are either created for a 'run' action, or for a 'results' action. The former performs the primary interaction with the resource, while the latter is used to pick up the results of the 'run'. For example, the 'run' action for a Mailchimp connector creates an audience list, creates a campaign, uploads contact information to the audience list, and then invokes the campaign. The 'results' action for the connector then obtains the results of that campaign, i.e. which audience members opened the emails, or which emails could not be sent.

Whether or not a 'results' action is required depends on whether the 'run' action starts in motion some activity (such as running an email campaign), the results of which must be gathered over time by the results action.

8.1 Run Flow

Run flows may either return data if the operation is immediate/transactional or may prepare and start some activity in the resource.

Run flows must conform to these requirements:

- There must be an input flow terminal on the first node in the flow. This must correspond to a file input terminal where the filename is "%csv_filename%". This way, when the flow is invoked via its REST API, data can either be passed in the HTTP request, or the location of the file to be read can be specified via the csv_filename query parameter.
- The flow should typically use a split node to split the data into smaller transactions that can be run in parallel. This is required to achieve adequate performance when there is a significant amount of data to process.
- If the run flow needs to return results, then there should be corresponding join nodes if the flow contains a split node.
- The input data must be in CSV format. The columns should include the mapped fields, and then followed by some number of identity fields
- If result data is returned from the flow, then it should also be in CSV format, and should return the identity fields and header that were provided in the request. The result file must be named "%results_dir%/%run_identifier%-results.csv".
- The flow must return any contextual information required by the results flow via flow variables, and these must be named "context.<name>". For example, in the case of the Mailchimp connector, the run flow creates a Mailchimp campaign, the id of which is needed to fetch the results of the campaign.

- If any initialization or set up is required before consuming the CSV data, this should be implemented in a separate initialization flow which can be referenced from the run flow.

8.2 Results Flow

If a connector provides a results flow to gather data over time after the run flow is executed it must conform to these requirements:

- An output node must have a flow terminal defined which is defined as appending to a file named "%results_dir%/%run_identifier%-results.csv". This way, when the flow is invoked via its REST API, data can either be returned in the HTTP response, or the location of the file to be written can be specified via the results_dir and run_identifier query parameters.
- The flow should not return duplicate results when invoked repeatedly. This means that it should use some mechanism to save its state so that when invoked again it can gather results since the last time it was invoked. This may require saving a 'last ran' date to a file, using Link's cache, or other means.

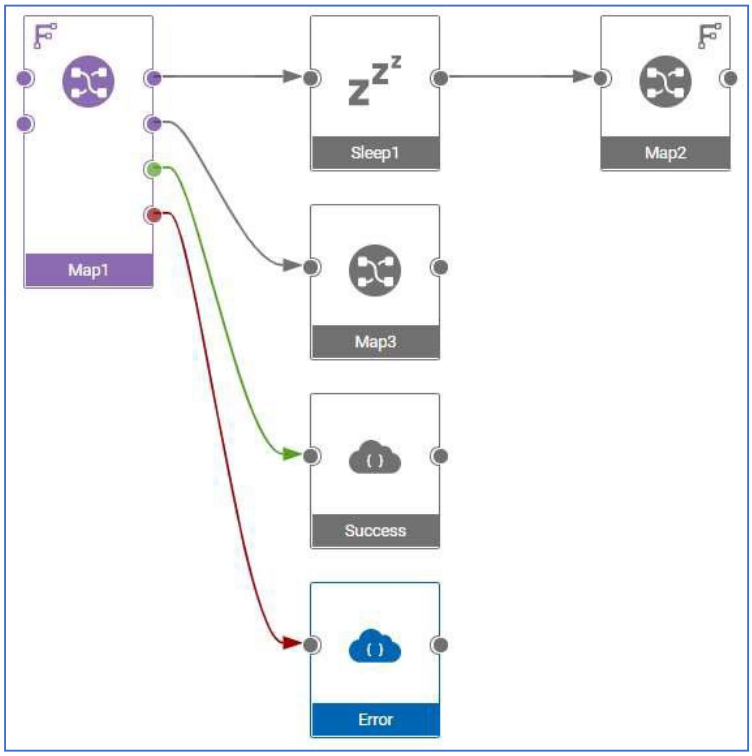
9 Flow Concepts

Some of the more important concepts of flows are explained in this section. The user documentation provides more complete details and should be consulted when developing flows. The sections below focus on concepts that are pertinent to the development of connector flows.

9.1 Flow Execution

It is important to understand the sequence of execution of a flow. A flow is run in a single process in a transactional manner. This means that if any node in the flow fails, then earlier node executions can be rolled back, and 'on error' actions invoked.

To illustrate the execution sequence, consider this flow:



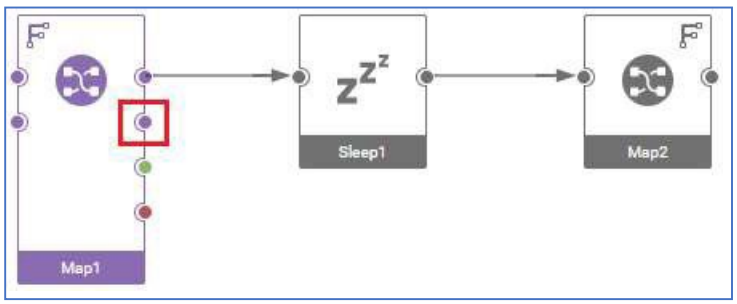
The sequence of execution is:

1. Map1 is started
2. When Output #1 is built in Map1, Sleep1 is invoked
3. Map2 is run
4. Output #2 is built in Map1 and Map3 is run
5. Either Success or Error is run, based on whether Map1 was successful

Note that:

- Map1 is not run in its entirety before calling other nodes. Child nodes are invoked as outputs are produced.
- Map1 will 'fail' if any of its child nodes fail. For example, if Map2 were to fail, that failure is passed back up to Map1, which will also then fail.

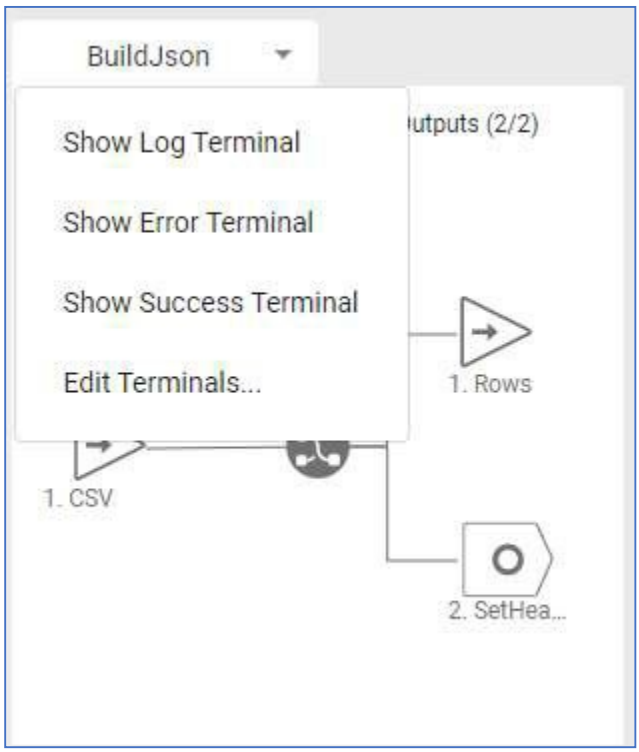
The significance of the first point is best illustrated with another example:



In this simpler case assume that the highlighted output of Map1 produces a file. When Map2 runs that file will not be available because that output has not yet been built. When creating a map, the order of outputs is hence very important.

9.2 Node Terminals

Nodes can have a single input terminal and multiple output terminals. In addition, nodes can also provide success, error and log terminals. To change the terminals that are available in the flow, click on a node, then select the options available from the dropdown menu



The success and error terminals are invoked when the node processing is completed. In the above flow the success terminal is invoked after all of the batches have been processed (the top row of nodes).

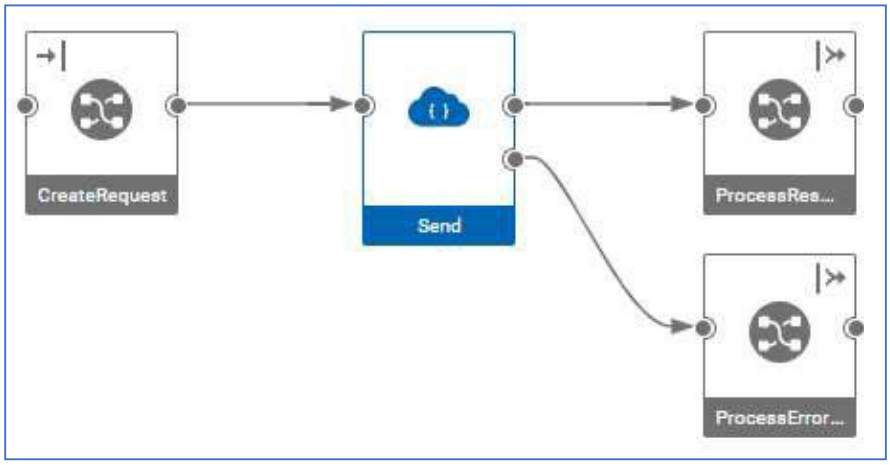
9.3 Flow Terminals

Flow terminals define the calling interface for the flow. If an input flow terminal is set on an input terminal of the first node, then it will receive data from the HTTP request. If an output flow terminal is set on an output node then its data will be returned in the HTTP response.

If there are more than 1 input or output terminal, then the data will be passed to the flow API as multipart/flow-data.

If a flow produces multiple outputs that need to be combined into a single output, that can be done so by enabling this property in the flow settings:

An example of when you may wish to combine outputs is:



In this example, the results of the REST Node's output terminals needs to be combined to a single output. The first output produces responses when the API call is successful, and the second when the API call fails.

9.4 Flow Variables

Data is primarily passed through nodes in a flow by connecting nodes together with links. Another option is to use flow variables, which can be convenient if there is a need to store some data value in one node, then retrieve it later in the flow.

Flow variables can be used as properties in adapter and node settings by specifying the property value as %variable_name%. Additionally, the REST Client node always includes flow variables in the collection of properties that are sent to the node.

Flow variables can be set or their values fetched within maps by using the functions SETVARIABLE(), GETVARIABLE(), INCVARIABLE() and DECVARIABLE().

Flow variables can be defined in a flow by editing the flow settings. Default values for the variables can be specified and can then be overridden when the flow is run. The 'Publish' toggle makes a flow variable appear as a query parameter in the Swagger documentation when a flow is published as an API.

When a flow is deployed from within link the value of any properties defined in the connector's properties will be set as flow variables in the flow.

Flow variables can also be specified when running a flow by providing them as query parameters. This is convenient when developing and testing a flow.

There are some special flow variables:

- _FLOWINSTANCE_ - the flow run number

- _FLOWUUID_ - unique UUID for a flow
- ~SPLIT~ - - the split number after a Split node.

These can be used to generate unique identifiers for file names etc.

9.5 Cache Variables

Link provides a cache mechanism which is similar to flow variables, but the values persist beyond a single run of a flow by being stored in a persistent cache. The cache is a convenient mechanism to use if one wishes to pass values from one flow to another, such as between a run flow and a results flow.

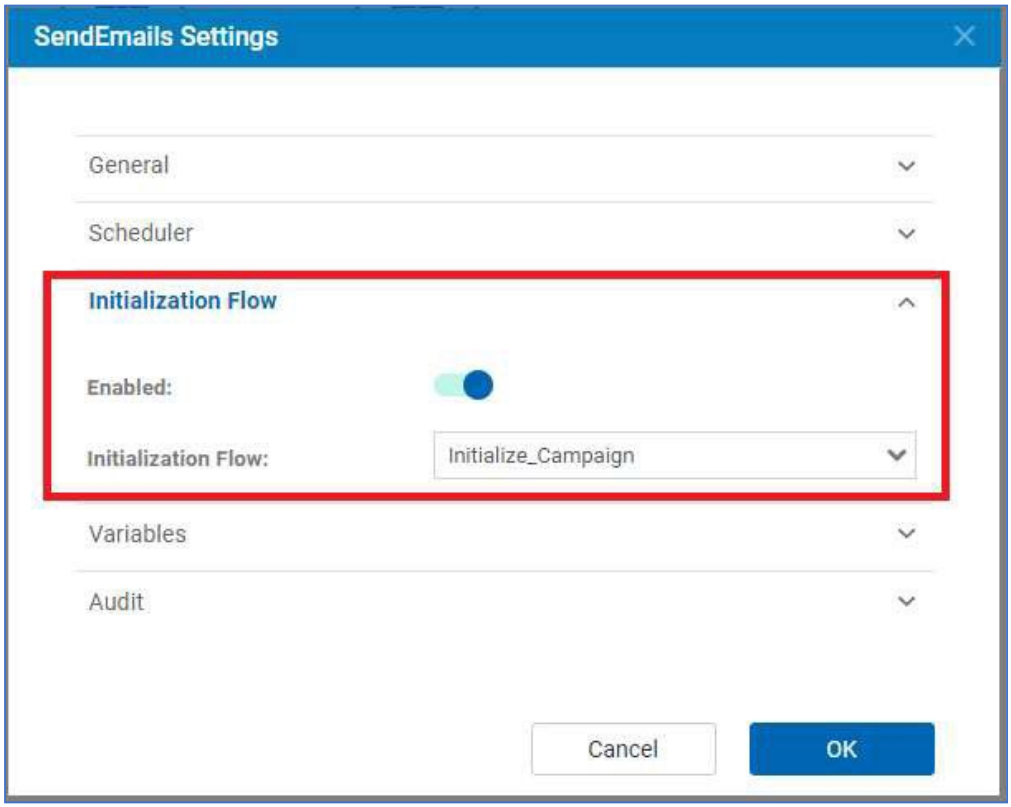
Within a flow, cache variables can be read and written by using the Cache Read and Cache Write nodes:

Within a map, the cache can be accessed by using the CACHEREAD(), CACHEWRITE() and CACHEDELETE() map functions.

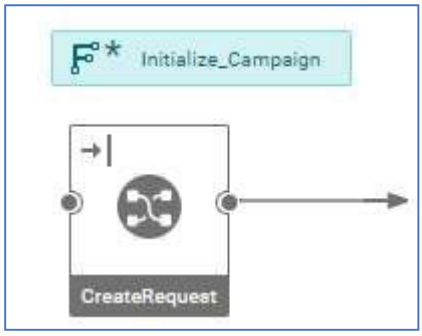
9.6 Initialization flow

If some initialization tasks need to be performed before the data is consumed, that can be achieved by putting the logic into an ***initialization flow***. Such a flow might be invoked to create tables, accounts or other artifacts in the target resource, that are then populated via the main flow. When the main flow is run, it first invokes the initialization flow (if one is defined) before executing the main flow.

The main flow can have an initialization flow associated with it by selecting a flow in the flow's settings:



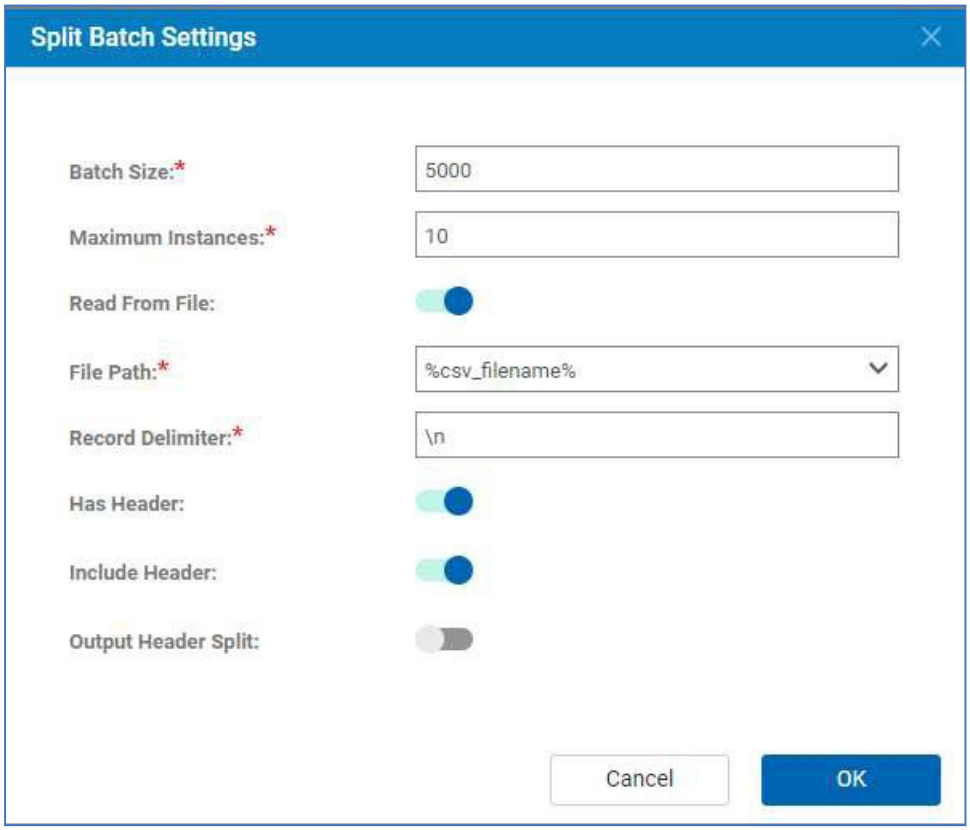
Once an initialization flow is selected, it is indicated on the flow of the main flow:



The initialization flow is run before the first node in the run flow is invoked. This can be used to perform any initialization steps before processing the incoming data.

9.7 Split node

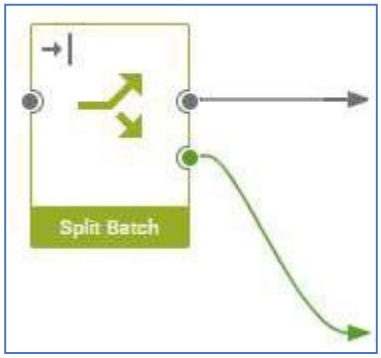
The first node (Split Batch) is a Split node with an input flow terminal defined on its input. The settings for the Split Batch node reference the csv_filename flow variable:



The properties of the node are:

- Batch size – the number of records to process in a thread. The incoming data will be batched according to this parameter, and each batch processes in a new thread concurrently with other batches.
- Maximum Instances – the maximum number of batches that can be processed in parallel. This should be set appropriately for the service being invoked. For example, if rate limiting on an API demands that a certain number of calls can be invoked in parallel, then the instance number should be set no larger than that limit
- File Path – this should always be set to flow variable name csv_filename since the Link framework will always be specifying this variable when reading the data from the file.

The input should have a flow terminal enabled:



This is set by clicking on the node, then in the structure right clicking the input terminal and selecting 'Set Flow Terminal'. The flow processing will then proceed as follows:

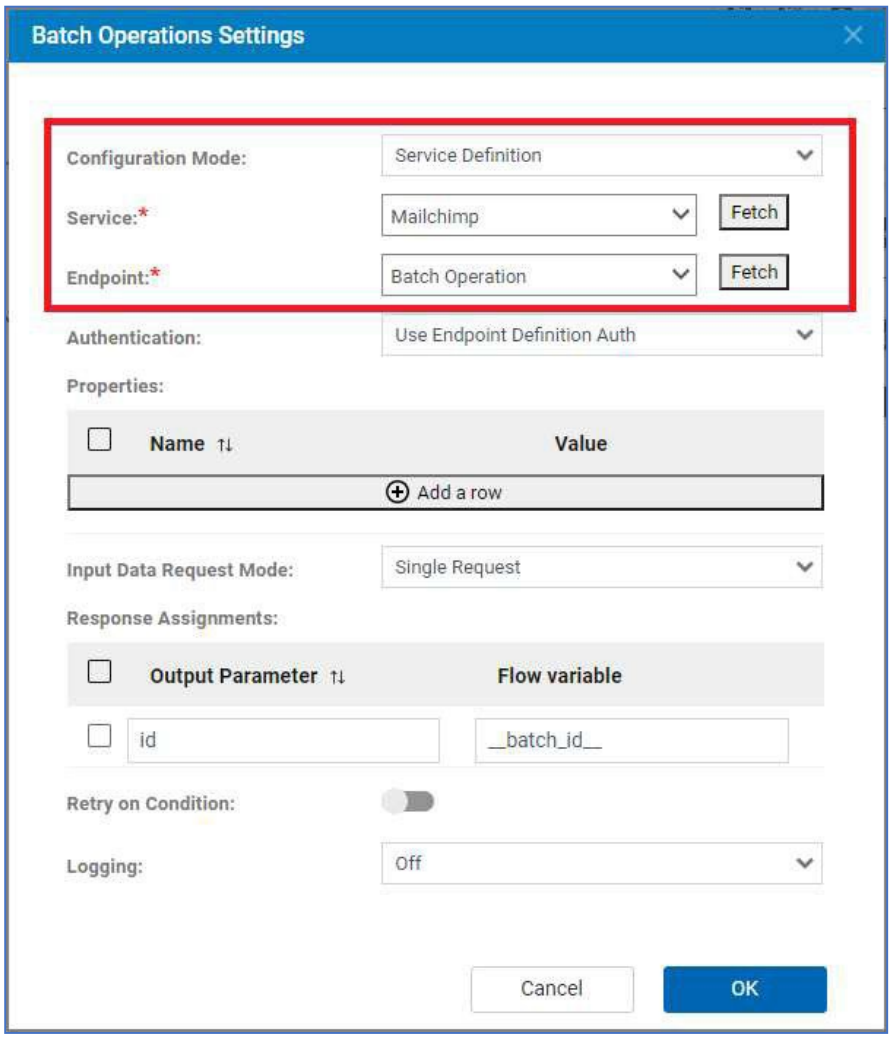
- If data is provided to the input terminal, e.g. by passing data in the HTTP request when running the flow, then that data shall be consumed and the filename specified in the node shall be ignored.
- If no data is provided in the run request, then the filename specified by the csv_filename flow variable will be read

9.8 REST Client Node

The REST Client node invokes a REST API from within a flow. It can be configured in 4 different ways:

- **Generic** – the node is configured will all the properties required to invoke the API (e.g. URL, authentication, headers etc)
- **Configuration script** – a configuration file within the project defines the endpoint
- **Configuration package** – a package installed in the Link server defines the configuration.
- **Service Definition** – reference a service definition defined in the Service Builder

For Link connectors, the last option should generally be used. The settings of the node are:



After a service definition has been created, clicking on Fetch for the Service property will populate the drop-down list with defined services. Then clicking on Fetch for the Endpoint property will return the list of endpoints defined for the service.

Properties can be set to literal values, or to the value of flow variables by specifying the flow variable name as %variable_name%. The set of flow variables will also be automatically added as properties and included in the set of properties provided to the

The Input Data Request Mode has 3 possible values:

- **Single Request** – the data passed to the input link is sent as the request to the REST API
- **Multiple Requests** – the data passed to the input link is a JSON array containing multiple JSON objects. The REST API is invoked for each request object.
- **Template** – the template defined in the request definition in the configuration form the base of the request. The Request Assignments table provides additional fields that are set to either a literal or flow variable value.

When sending JSON objects for either a single or multiple request these special elements can be added to the request:

- **_properties_** - A set of properties that are added to the property set.
- **_context_** - A JSON object that is removed from the request when the API is called and added back into the response returned from the node. This provides a way to pass elements through the flow.

For example, if the API being called requires this request object:

```
{
   "name": "Fred",
   "action": "add"
}
```

Properties and context can be added in the request object:

```
{
   "name": "Fred",
   "action": "add",
   "_properties_": {
      "property1": "value1",
      "property2": "value2"
   },
   "_context_": {
      "passthru1": "value1",
      "passthru2": "value2"
   }
}
```

Alternatively, if one does not wish to modify the request object, the request object can be provided in a _request_ field:

```
{
   "_request_": {
```

```
      "name": "Fred",
      "action": "add"
   },
   "_properties_": {
      "property1": "value1",
      "property2": "value2"
   },
   "_context_": {
      "passthru1": "value1",
      "passthru2": "value2"
   }
}
```

The success or error response will include _context_ if it was provided in the input. If the first form was used the _context_ will be added into the response object:

```
{
   "id": 12314,
   "_context_": {
      "passthru1": "value1",
      "passthru2": "value2"
   }
}
```

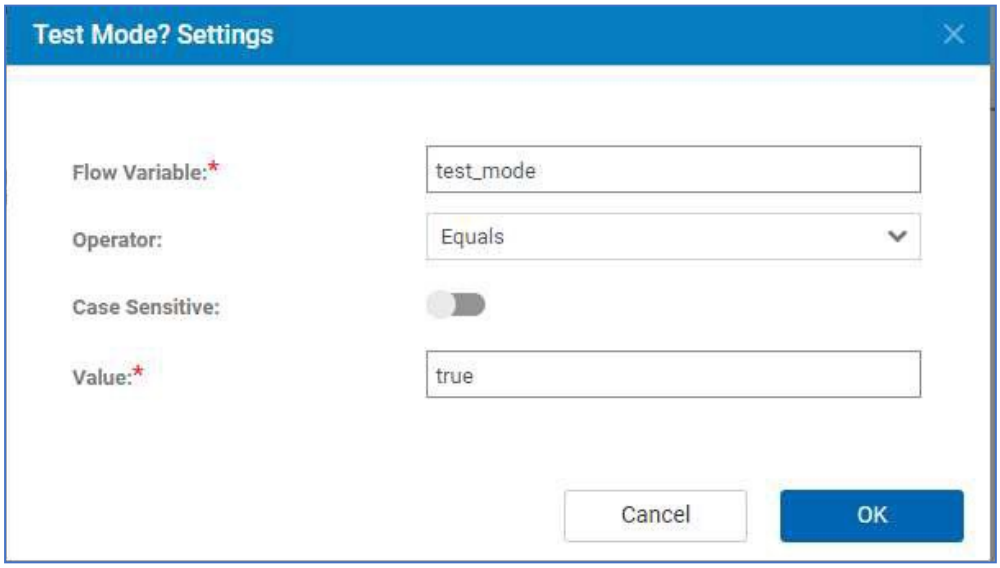
If the second form was used, then the response object will be returned in a _response_ field:

```
{
   "_response_": {
      "id": 12314,
   },
   "_context_": {
      "passthru1": "value1",
      "passthru2": "value2"
   }
}
```

The REST Client node has 2 output terminals: Success_Response and Error_Response. If the REST API returns a 2xx status, the response will be sent to the Success_Response, otherwise it will be sent to the Error_Response. If the data contains multiple requests, then the responses sent to both success and error terminals will contain JSON arrays of responses.

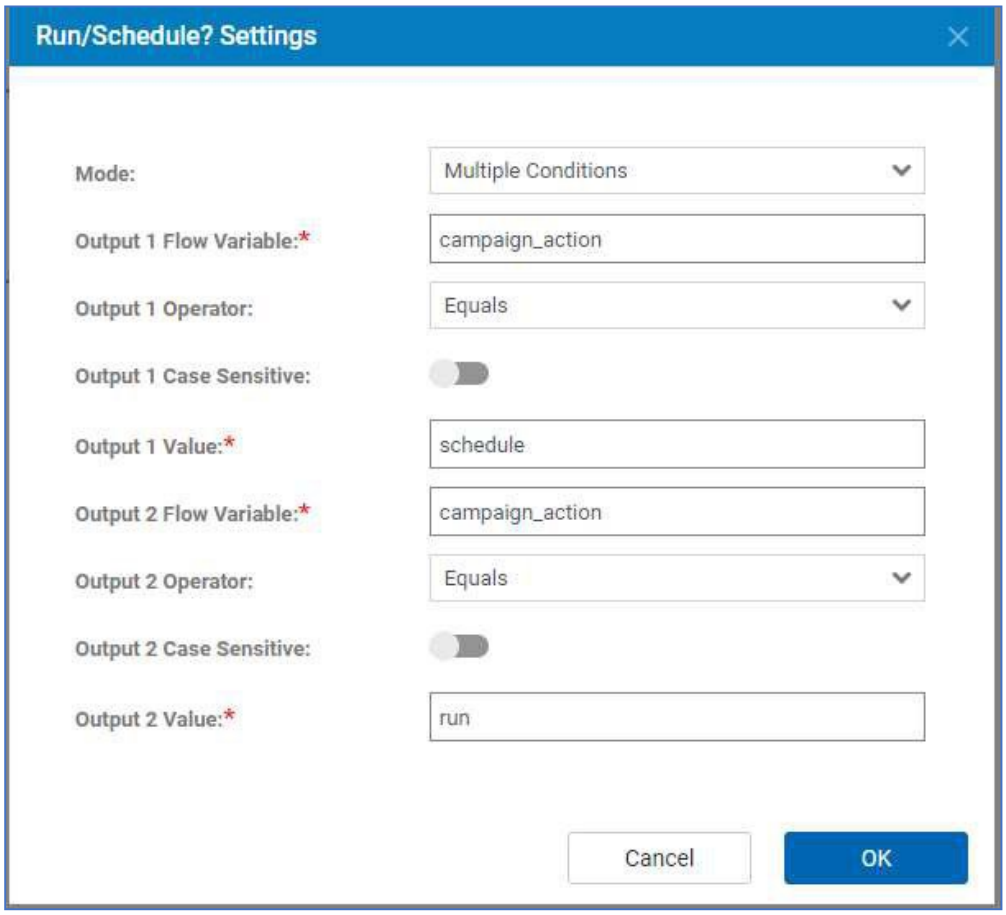
9.8.1.1 Decision & Route Nodes

The Decision and Route node can be used to perform conditional logic within flows. The decisions and routing are based upon the values of flow variables. The settings for a Decision node are:



The Decision node has True and False output terminals. The data sent to the input terminal is either routed to the True or False terminal based on the evaluation of the expression.

The Route node is similar but provides more flexibility. It allows for multiple conditions so that multiple choices can be made. For example:

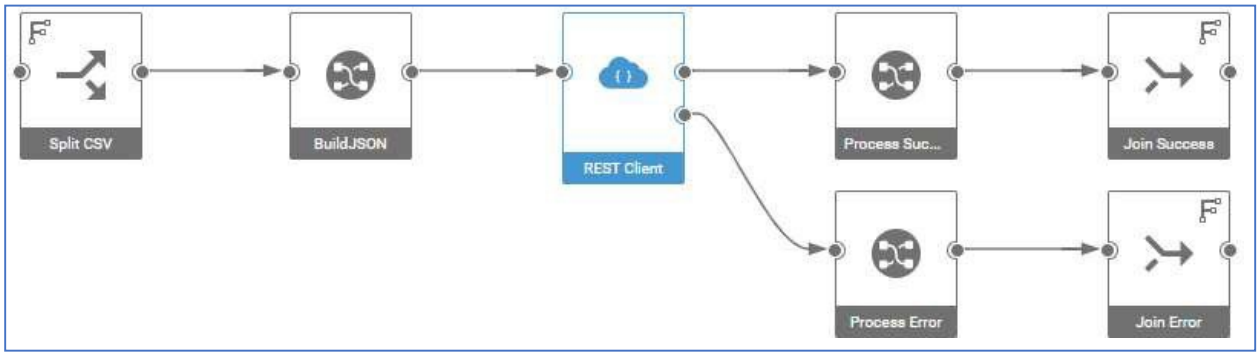


In this example, if the value of campaign_action flow variable is 'schedule', then the incoming data will be sent to Output 1. If the value of campaign_action flow variable is 'run', then the data will be sent to Output 2. If campaign_action is neither then no outputs will be triggered.

9.9 Flow Performance

To achieve satisfactory processing speed when running a flow with a large dataset it is important to split the data into batches using the Split node. The Join node should then be used to gather the results to produce a single output.

To call a REST API for each incoming record the Multiple Requests mode of the REST Client node should be used. A typical flow would have, at a minimum, these nodes:



In this flow, the Split node produces batches of records. For each batch, BuildJSON map creates a JSON Array containing a sequence of request data for the API being called. The Process Success and Process Error maps map the success and error responses which are returned as a JSON arrays to CSV result data. Finally, the Join nodes bring the individual batch results together into a single output.

The batch size should be set such that the size of the data passed to the client is not too large, but at the same time should not be so small that there are many small batches, since there is some small overhead in managing the threads produced by the Split node.

10 Java Plug-Ins

In some cases, it may be necessary to add Java code to modify a REST request or response when the REST client node is invoking a REST API. Example scenarios are:

- Security headers need to be added compute a signature based on the request data and parameters
- To modify the request body to add additional information
- To modify the response to compute something based on the response

Typically, this is not necessary because the requests and response can be processed by maps or other processing nodes. If there is no other solution, then follow these steps to implement and deploy a Java plugin.

The Java plugin is invoked whenever an API call is made for the connector. The plugin can provide methods to modify elements of the request before it is sent to the API, and to modify the response data after calling the API.

10.1 Create Java Class

The development kit includes a dummy plugin class in plugin.zip in the development kit. This should be used as a starting point for developing your own plug. The M4RestPlugin abstract class has 2 optional methods:

```
public RestRequest customizeRequest(String endpointName, RestRequest request, Properties
props, byte[] data)

public RestResponse customizeResponse(String endpointName, Properties props,
RestResponse response)
```

In these calls, parameters are:

- **endpointName** – the name of the endpoint defined in the service
- **props** – the property values set in the REST Client node

The provided class includes a Maven pom.xml file that lists 2 dependencies:

```
<dependency>
	<groupId>hcl.tx</groupId>
	<artifactId>m4rest</artifactId>
	<version>${env.VERSION}</version>
</dependency>
<dependency>
	<groupId>hcl.tx</groupId>
	<artifactId>restutils</artifactId>
	<version>${env.VERSION}</version>
</dependency>
```

These 2 jar files are within the Link installation. They can be found in this directory:

```
<LinkDir>/jars/com.hcl.hip.adapters.m4rest
```

They need to be copied from there and moved to Maven repository by calling:

```
mvn install:install-file -Dfile="m4rest.jar" -DgroupId=hcl.tx -DartifactId=m4rest
-Dversion=1.0 -Dpackaging=jar

mvn install:install-file -Dfile="restutils.jar" -DgroupId=hcl.tx -
DartifactId=restutils -Dversion=1.0 -Dpackaging=jar
```

where the version number must match the version specified in the pom.xml.

The name of the package and the class must match the ID of the connector. So if the connector ID is MOCKMAIL, the package must be called com.hcl.hip.adapters.m4rest.mockmail (all lower case) and the class name must be MockmailPlugin (camel-case).

The sample plugin includes a JUnit test case. It is advisable to test your plugin code by driving it from JUnit.

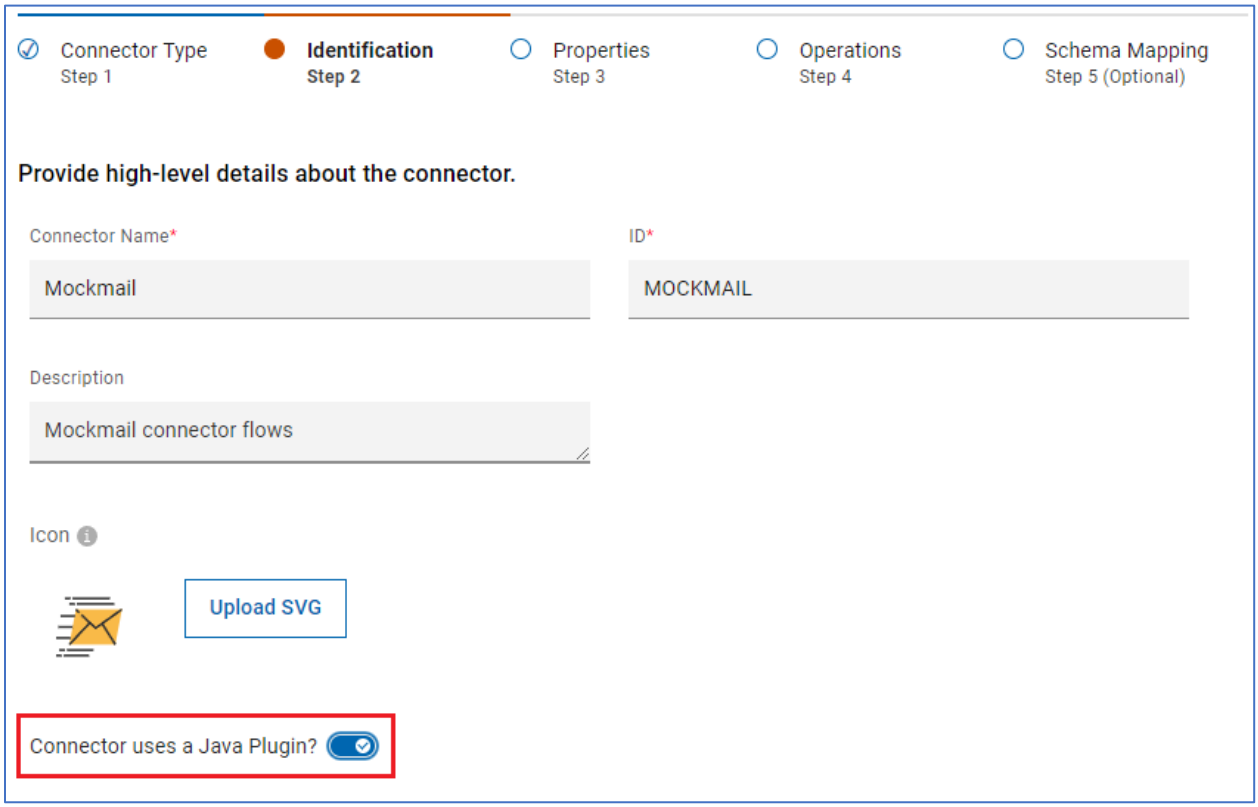
To compile the plugin, from a command line or an IDE invoke Maven with “clean install”. From a command line this is:

```
mvn clean install
```

This will compile the code and run the unit tests.

10.2 Adding Plugin to Connector

The plugin code has to be added to the connector. The first step is to edit the connector and enable the Java Plugin toggle in the Identification step:



This will specify that a class named com.hcl.hip.adapters.m4rest.mockmail.MockmailPlugin will be provided to implement the plugin.

To add the plugin class into the connector, follow these steps:

1. Export the connector
2. Unzip the connector zip file
3. From the command line, go to the <plugin>\target\classes directory that contains the plugin class files.
4. Issue this update command to add the .class files to the connector's jar file:
   `jar uf <unzipped-connector-dir>\jars\mockmail.jar .`
5. Zip up the connector zip file again, preserving its original name and case.

The connector can now be deployed to a Link design or runtime server by copying the zip file to the modules directory of the Link install, and restarting the server. This directory is defined by the environment setting HIP_MODULES_DIR. It defaults to /opt/hipmodules on Linux and <Link>/modules on Windows.