Unica Link V12.1.1
Creating Link Connectors Reference

Contents

1 Introduction 2
2 What is a Link Connector? 2
3 REST API Connectors 3
3.1 Understanding the APIs 3
3.2 Capture request and response formats 3
3.3 Defining a Service 4
3.4 Create Link Flows 6
3.4.1 Run Flow 6
3.4.2 Results Flow 7
3.4.3 Flow Concepts 8
3.5 Test the Flows 19
3.6 Export the Project 20
3.7 Create Connector Descriptor 20
3.7.1 Property Definitions 21
3.7.2 Implementation Mapping 23
3.7.3 Placeholder Nodes 23
3.7.4 Enumerations 25
3.7.5 Schema Mapping 26
3.8 Packaging & Translation 29
3.9 Installation 30
3.10 End-to-End Testing 31
4 Packager 32
5 Connector Descriptor Specification 33
5.1 General Fields 35
5.2 Property Descriptors 35
5.2.1 Property enablement 38
6 Java plugin 38
6.1 Prerequisites 38
6.2 Create a new Maven project 38
6.3 Pom.xml from the plugin project 39
6.4 Eclipse project structure 41
6.5 Override M4RestPlugin methods 41
6.6 Package the connector 42

© Copyright HCL Ltd. 2021 All rights reserved

1 Introduction

This document describes elements of Link that require understanding such that one can create a connector to a resource for Link. The document **Creating Link Connectors – Tutorial** provides a step-by-step walkthrough of the steps to create, test and deploy a connector.

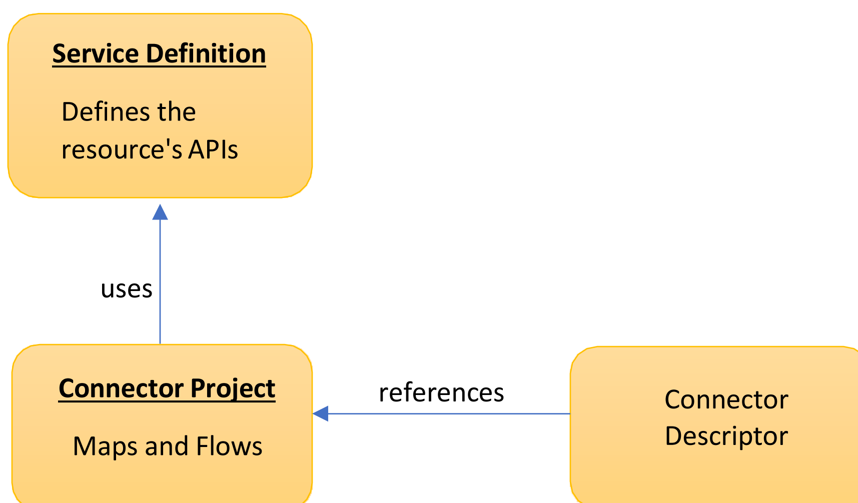
Currently, this document describes only how to create connectors that invoke REST APIs. The document will be extended in the future to describe other types of connectors.

The word "resource" is used throughout this document. This term is used to refer to the system, application or service that Link is connecting to. Examples are Mailchimp, Mandrill, Twilio or Salesforce.

2 What is a Link Connector?

A Link connector is comprised of a set of artifacts that run in Link to enable a Link consumer to access a resource, via REST APIs or other interfaces.

A connector that invokes REST APIs is comprised of these components:



The components are:

- The service definition is created in Link's Service Editor. It is used to define the APIs of the resource that will be invoked by Link when executing an action.
- Maps and flows that use the REST adapter to invoke the endpoints defined within the configuration file.
- A connector descriptor that defines the connection and action properties, and the mapping of maps and flows to action properties.

© Copyright HCL Ltd. 2021 All rights reserved

3 REST API Connectors

The steps for creating a connector that invokes REST APIs are:

1. Determine what APIs need to be invoked
2. Define a service definition in the Service Builder component of Link
3. Create flows that use the REST node and other node types
4. Create connector descriptor to define the properties of the connector
5. Test the maps and flows

3.1 Understanding the APIs

Interactions between Link and the resource include design time APIs, such as:

- Validating the connection to the resource
- Listing objects to be displayed as enumerated values for a property
- Fetching field definitions

Runtime interactions include:

- Creating objects in the resource
- Sending data to the resource
- Getting data from the resource

In some cases, multiple APIs may need to be invoked in series to accomplish some task. There are multiple ways to orchestrate multiple API calls. Simple orchestration can be done within Link's REST node, but more complex orchestration may require maps or flows. These different approaches are described later in this document.

3.2 Capture request and response formats

For each API that is being invoked, the request and/or reply format may need to be captured so it can be populated by a Link map, or so the result can be parsed. Link maps require a sample JSON document that typifies a request or reply. This can either be obtained from the API's documentation, which often has sample request and reply messages, or by invoking the API from the Service Builder.

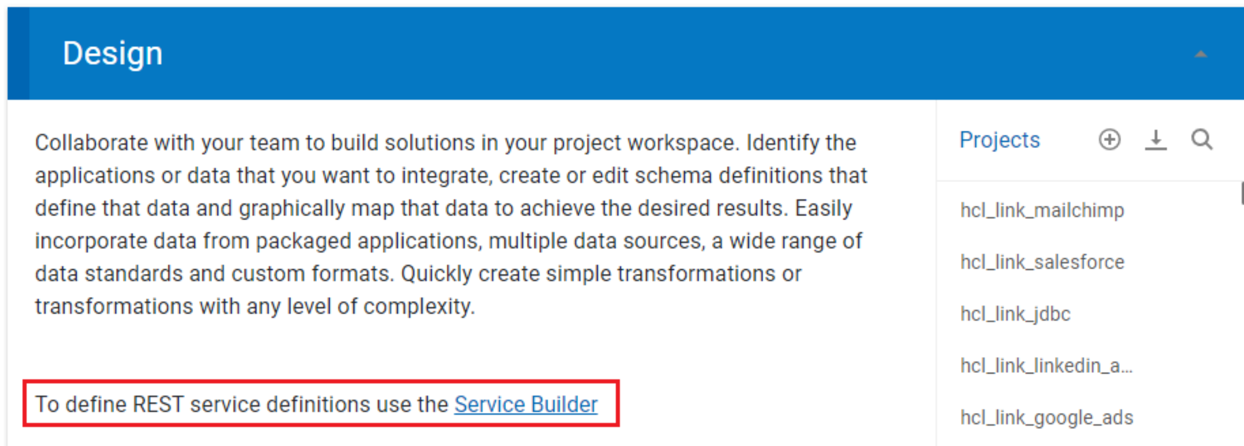
Getting these formats from the API documentation should be pursued with caution. Sometimes the documentation is not accurate, and the actual request or reply could differ from reality.

The Service Builder allows you to specify the details of a service and API, and to invoke the API to generate both successful and error responses. These responses can then be used in the Link maps and flows that implement the action.

© Copyright HCL Ltd. 2021 All rights reserved

3.3 Defining a Service

REST Services are defined in Link by invoking the Service Builder which is accessed from the Link home page:



Service definitions are defined by a service entry that defines URL, authentication and retry strategy:

Service Information Step 1 | Authentication Step 2 | Retry Strategy Step 3 | Define Parameters Step 4

Service Information

Service Name:* Mailchimp

Description: Mailchimp Resource

Base URL:* {$base_url}

and one or more endpoints (APIs):

© Copyright HCL Ltd. 2021 All rights reserved

⌄	☐	Mailchimp	Mailchimp Resource
		Ping	Authentication test
		Get Email Activity	Get email activity for a campaign
		Get Lists	(Meta) Get list of audiences
		Get detail of a list	Get details of a list
		Get list for campaign	Get detail of a list related to a campaign
		Get Templates	Get list of templates
		Get Campaigns	Get list of campaigns

For each endpoint, a relative URL, HTTL method and other parameters are specified:

General Information Step 1 | **Request Step 2** | Success Response Step 3 | Error Response Step 4

Request

Relative URL:* lists

Method:* POST

Query Parameters:

☐	**Parameter**	**Value**

Add a row

The APIs can be invoked directly from the Service Builder to capture the response formats which can then subsequently be used in the map and flow definitions.

© Copyright HCL Ltd. 2021 All rights reserved

3.4 Create Link Flows

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 Mailchimp creates an audience list, creates a campaign, uploads contact information to the audience list, and then invokes the campaign. The 'results' action for Mailchimp 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 have to be gathered over time by the results action.

3.4.1 Run Flow

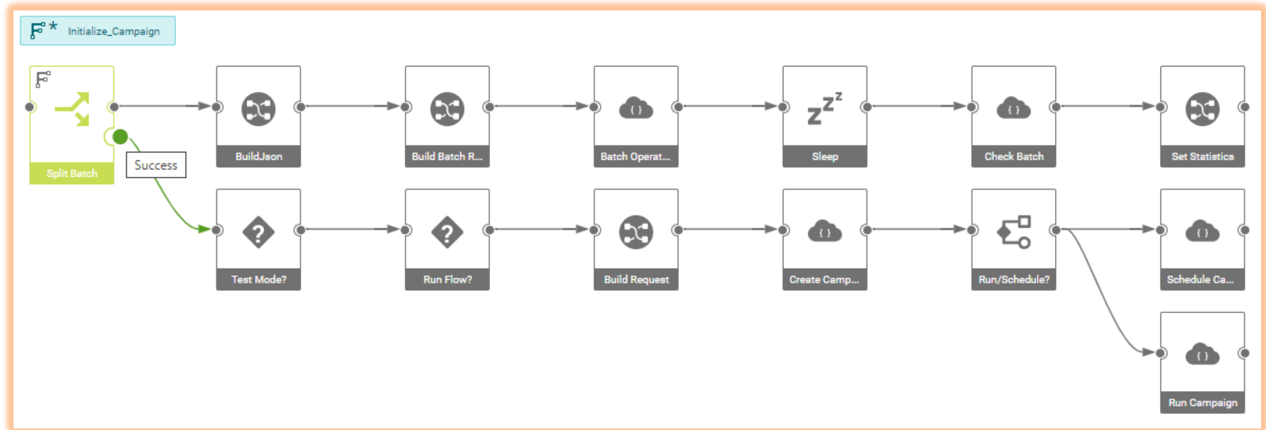
Run flows may either return data if the operation is immediate/transactional or may prepare and start some activity in the resource.

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 with a header row. 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 Mailchimp, 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.

For example, a flow for Mailchimp looks like this:

© Copyright HCL Ltd. 2021 All rights reserved



This flow does the following:

- Splits the incoming CSV data into batches, each batch being processed in parallel.
- Invokes a series of REST APIs for the incoming data
- Once the batch processing is complete, invokes more REST APIs to create a Mailchimp campaign and run or schedule it.

We shall use this flow as an illustration to examine some of the key features.

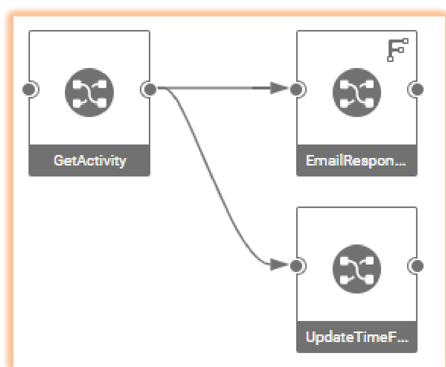
3.4.2 Results Flow

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.

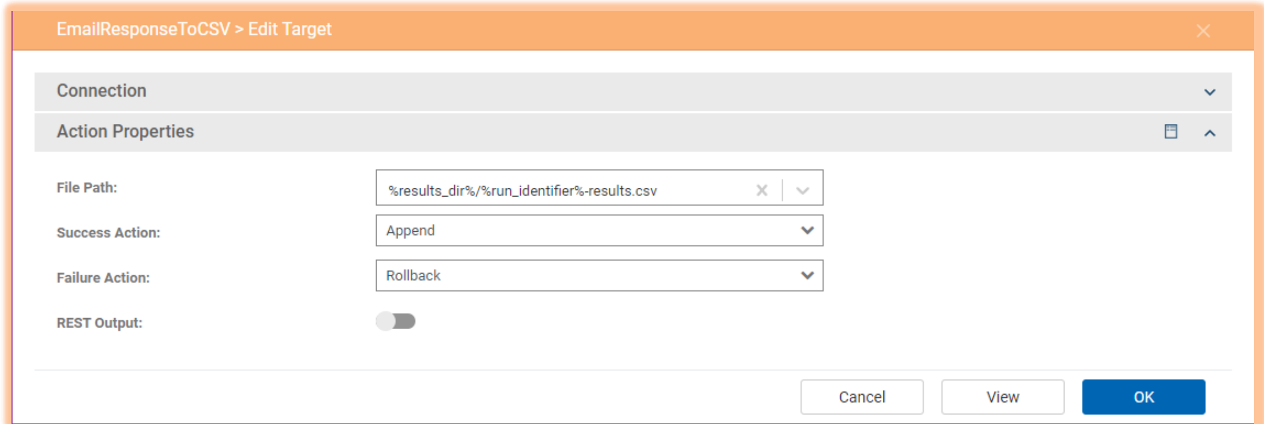
- The flow should not return duplicate results when invoked repeatedly. This means that it should use some mechanism to save its state so that when invoked again it can gather results since the last time it was invoked. This may require saving a 'last ran' date to a file, or other means.

For example, a flow for Mailchimp may look like this:



© Copyright HCL Ltd. 2021 All rights reserved

The settings for the output flow terminal (a card in the map EmailResponses) has these settings:



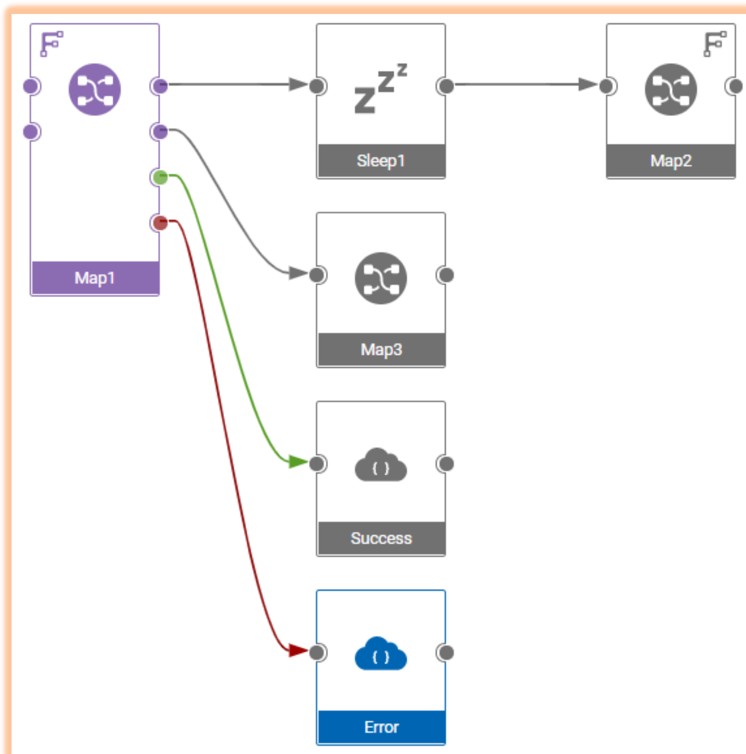
3.4.3 Flow Concepts

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.

3.4.3.1 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:



© Copyright HCL Ltd. 2021 All rights reserved

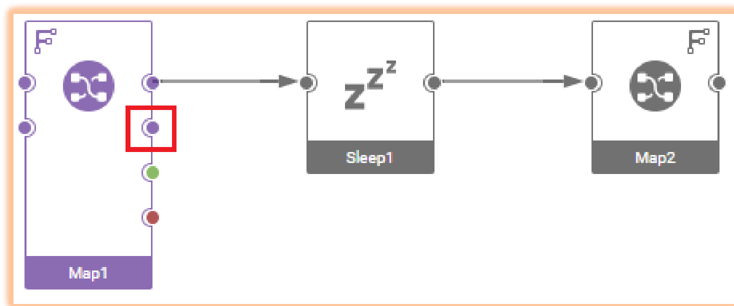
The sequence of execution is:

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:

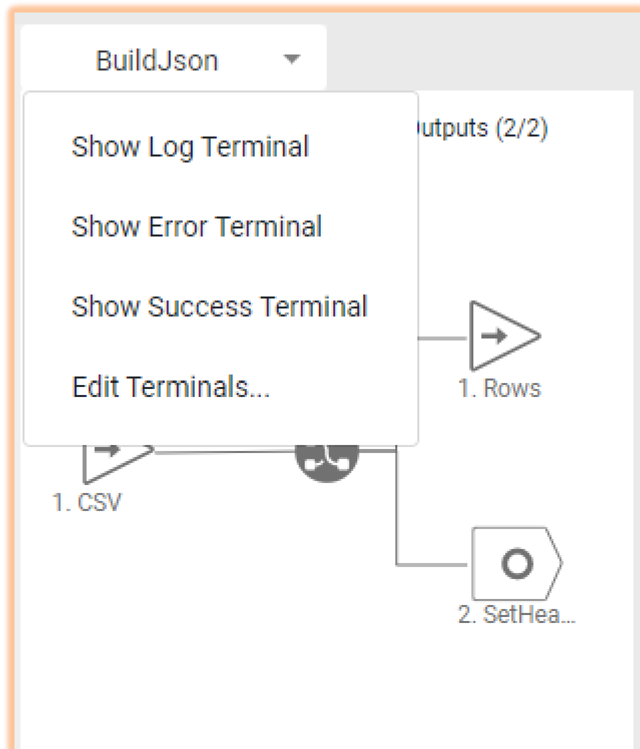


In this simpler case assume that the highlighted output of Map1 produces a file. When Map2 runs that file will not be available because that output has not yet been built. When creating a map, the order of outputs is hence very important.

3.4.3.2 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

© Copyright HCL Ltd. 2021 All rights reserved



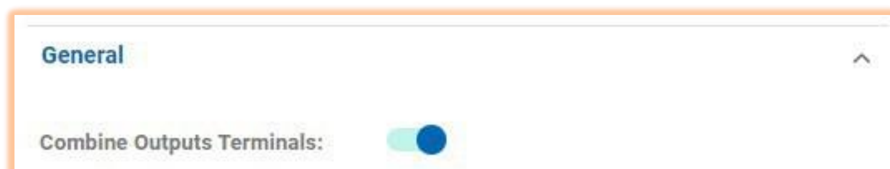
The success and error terminals are invoked when the node processing is completed. In the above flow the success terminal is invoked after all of the batches have been processed (the top row of nodes).

3.4.3.3 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.

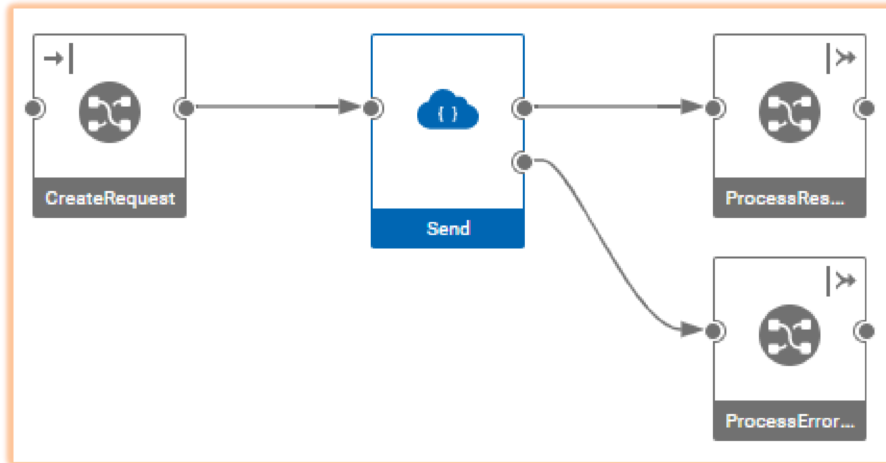
If there are more than 1 input or output terminal, then the data will be passed to the flow API as multipart/flow-data.

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:



© Copyright HCL Ltd. 2021 All rights reserved

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.

3.4.3.4 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.

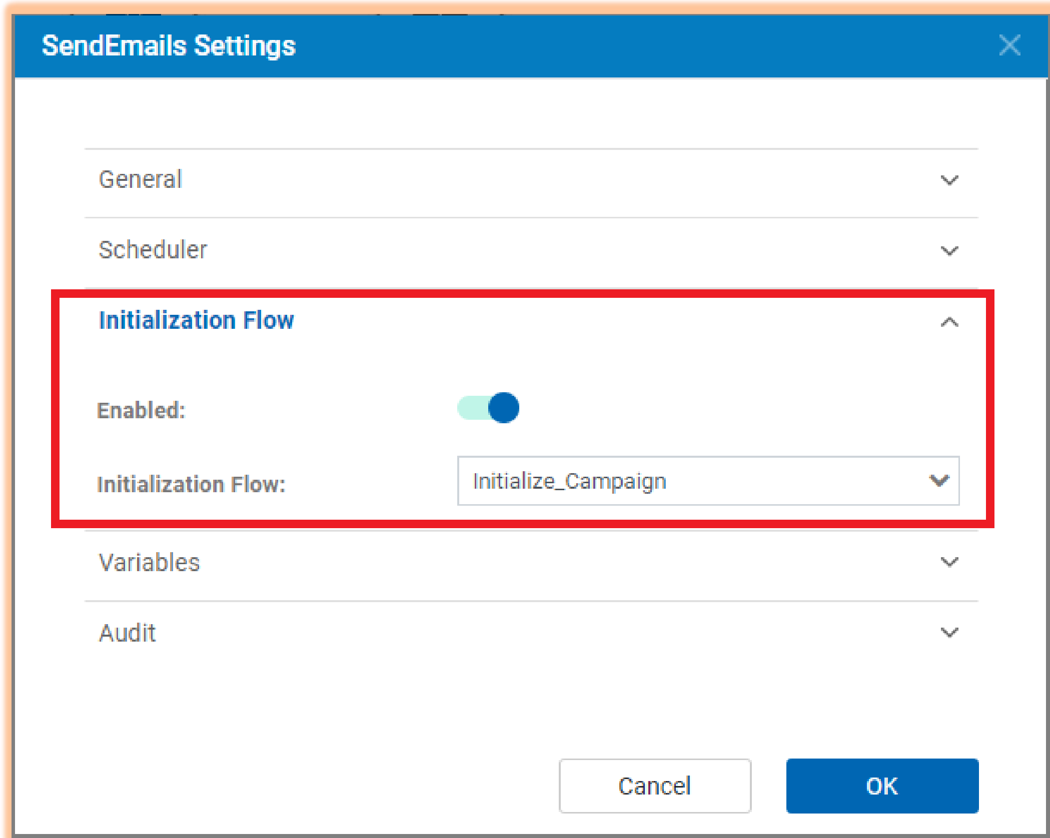
These can be used to generate unique identifiers for file names etc.

© Copyright HCL Ltd. 2021 All rights reserved

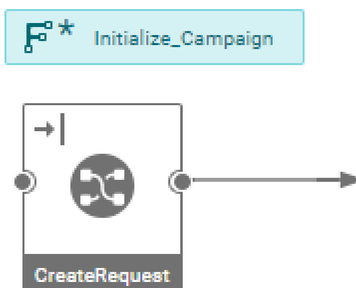
3.4.3.5 Initialization flow

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.

© Copyright HCL Ltd. 2021 All rights reserved

3.4.3.6 *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:

Split Batch Settings

Batch Size:* 5000

Maximum Instances:* 10

Read From File:

File Path:* %csv_filename%

Record Delimiter:* \n

Has Header:

Include Header:

Output Header Split:

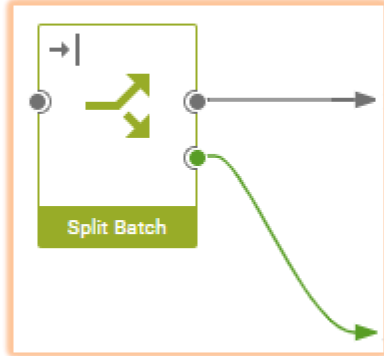
Cancel OK

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.

© Copyright HCL Ltd. 2021 All rights reserved

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

3.4.3.7 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 packager and installer tools (see later) package up a connector and related service definitions as an installable connector.

© Copyright HCL Ltd. 2021 All rights reserved

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.

© Copyright HCL Ltd. 2021 All rights reserved

- **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"
   }
}
```

© Copyright HCL Ltd. 2021 All rights reserved

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"
   }
}
```

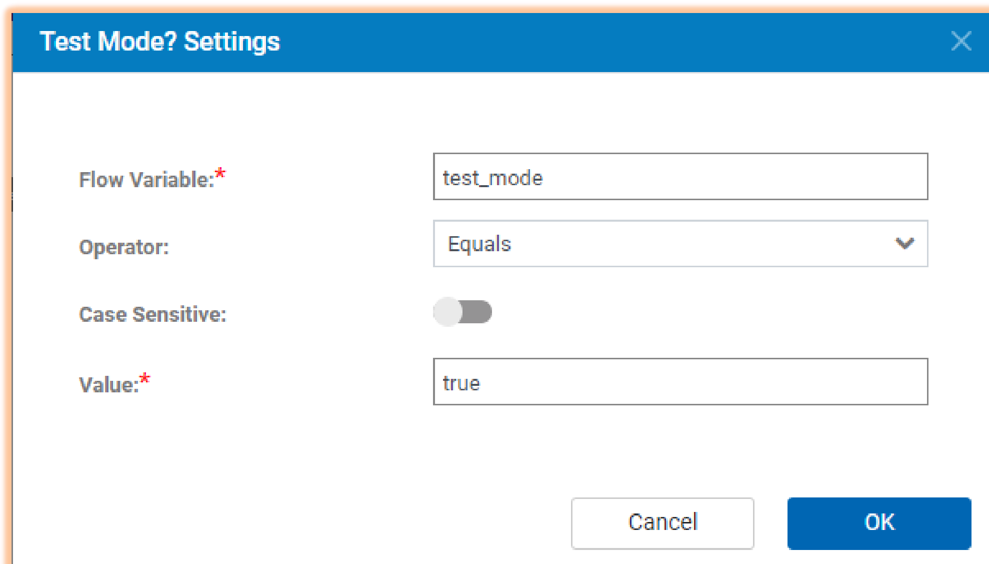
If the second form was used, then the response object will be returned in a _response_ field:

```
{
   "_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 contains JSON arrays of responses.

3.4.3.8 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.

© Copyright HCL Ltd. 2021 All rights reserved

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

Run/Schedule? Settings

Field	Value
Mode:	Multiple Conditions
Output 1 Flow Variable:*	campaign_action
Output 1 Operator:	Equals
Output 1 Case Sensitive:	
Output 1 Value:*	schedule
Output 2 Flow Variable:*	campaign_action
Output 2 Operator:	Equals
Output 2 Case Sensitive:	
Output 2 Value:*	run

Cancel OK

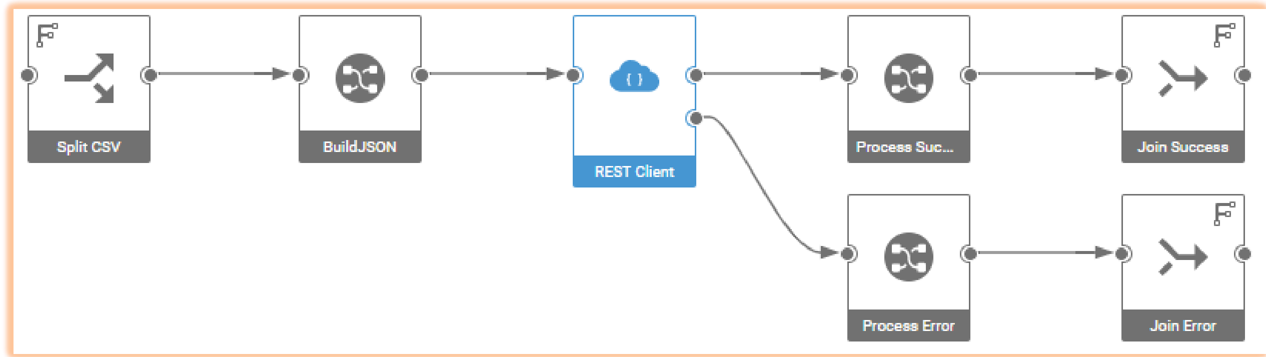
In this example, if the value of campaign_action flow variable is 'schedule', then the incoming data will be sent to Output 1. If the value of campaign_action flow variable is 'run', then the data will be sent to Output 2. If campaign_action is neither then no outputs will be triggered.

3.4.3.9 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.

© Copyright HCL Ltd. 2021 All rights reserved

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.

3.5 Test the Flows

Having created maps and flows they can be tested from the Designer. The steps for doing so are:

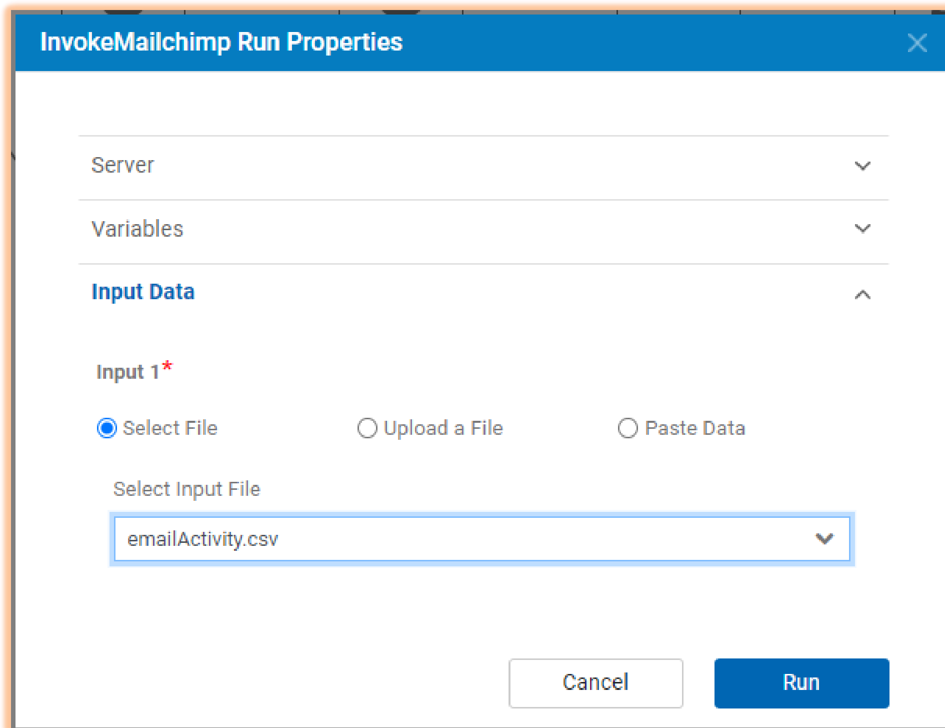
1. After saving a flow, analyze it to ensure it is valid. This is invoked from the toolbar in the Flow Designer

   Analyzing the flow ensures that required properties are defined and that there are not inconsistencies within the flow definition.

2. The flow can be run directly from the Designer. Click on the run button on the toolbar:

© Copyright HCL Ltd. 2021 All rights reserved

This opens a dialog whereby values for flow variables can be entered that are passed to the flow, and input data for the flow can be provided. One can either select a file, upload a new file or simply paste input data into the dialog:



After the flow run has completed, from the Designer one can inspect:

- The log for each node
- The data produced by the output
- The data send down each link

3.6 Export the Project

Once the maps and flows are working as designed, it is wise to export the project and maintain in source control. The export function can be accessed from the menu at top of the Designer.

3.7 Create Connector Descriptor

To expose the connector to Link users, a connector descriptor is required. The connector descriptor defines the properties that are shown in the Link user interface and provides other information that is required by Link to be able to deploy and manage instances of flows.

The descriptor is a JSON file that contains:

- General metadata (e.g. name, description, etc.)
- Property definitions for the connection and action properties
- An association between action definitions and run and results flows
- Enumeration definitions
- Schema mappings

© Copyright HCL Ltd. 2021 All rights reserved

The full specification for connector descriptors is provided in Section 5 Connector Descriptor Specification.

3.7.1 Property Definitions

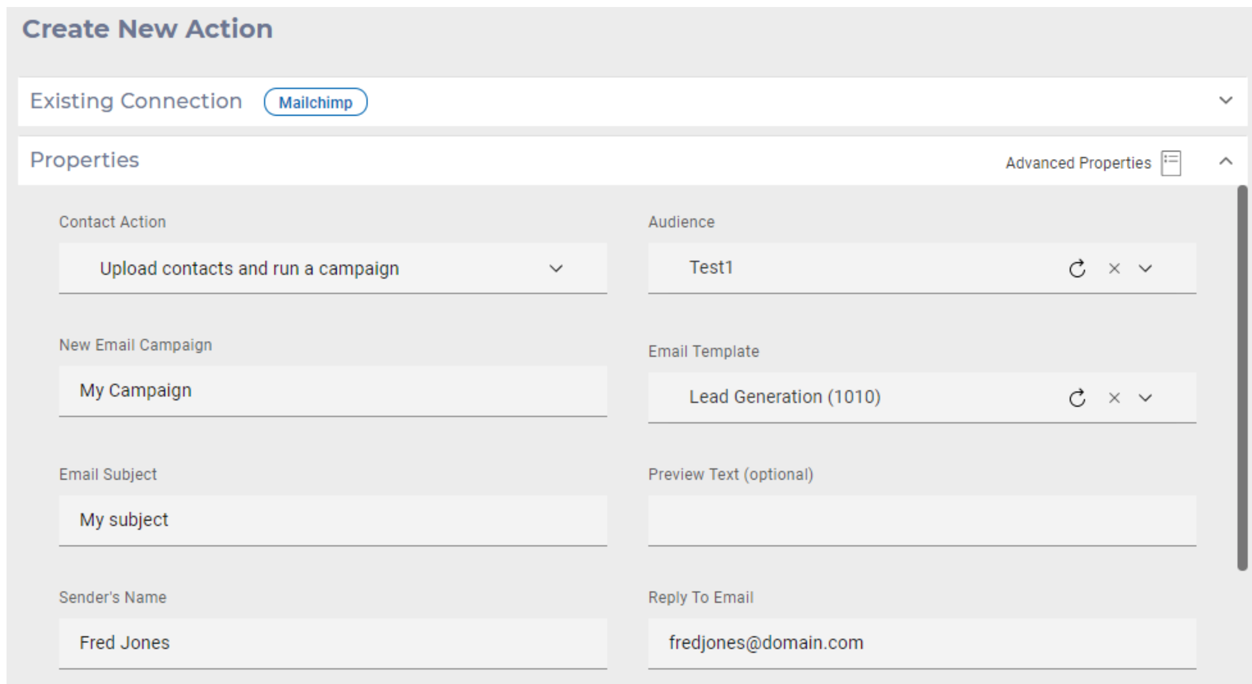
All properties must be defined under the `properties` array element.
An example property definition is:

```
{
    "label": "Base URL",
    "name": "base_url",
    "type": "string",
    "required": true,
    "description": "The base URL to the MailChimp data center",
    "scope": "connection"
}
```

If connection/action properties does not exist for a connector, define the following tag in the connector descriptor file:

```
"properties": []
```

The properties defined in the descriptor are dynamically rendered in the Link user interface:



Having created a properties descriptor, it can be validated by using the -vd option of the packager tool (see section Packager). This performs the following validation:

- Validates the structure of the JSON
- Ensures all cross references and expressions refer to existing properties

The packager tool is also used to produce localized versions of the descriptor when -p package

© Copyright HCL Ltd. 2021 All rights reserved

command is invoked. This does the following:

- Assign property IDs and add cross-references
- Replaces all labels and descriptions with message bundle key names

© Copyright HCL Ltd. 2021 All rights reserved

The output directory contains the output descriptor, and a message bundle. The generated message bundle can then be translated to other languages. The result is a set of JSON files with locale extensions, e.g. MAILCHIMP_en_US.json, MAILCHIMP_jp_JP.json, MAILCHIMP_de_DE.json, etc.

3.7.2 Implementation Mapping

The implementations element associates maps and flows with action properties, such that based on the choices made by the user the appropriate maps and flows are run.

The simplest case defines which flow to run for the action. E.g.

```
"implementations": [
    {
        "name": "send_bulk_sms",
        "run": {
            "flow": {
                "template": "SendSMS"
            }
        }
    }
]
```

This instructs Link to run the SendSMS flow for all actions.

If there are multiple flows to be run based on different action properties, then conditions are used to determine which to run. For example,

```
"implementations": [
    {
        "name": "audience_run",
        "condition": "operation == audience",
        "run": {
            "flow": {
              . . .
            }
        },
        "results": {
            "flow": {
              . . .
            }
        }
    },
    {
        "name": "campaign_run",
        "condition": "operation == campaign",
        . . .
    }
]
```

In this example, conditions determine which flow should be run. The condition specifies an action property and a value. The operation can either be == (equals) or != (not equals).

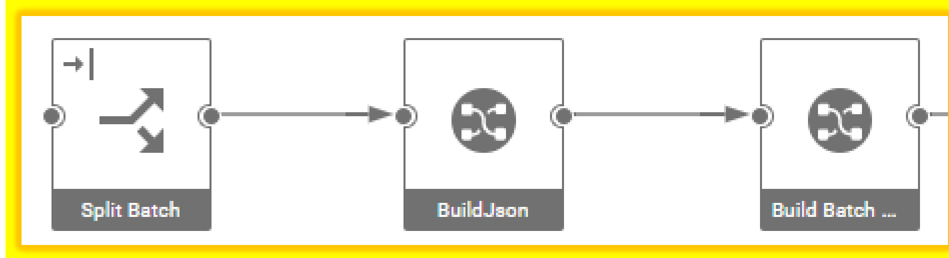
3.7.3 Placeholder Nodes

A connector may contain a mapping node which is replaced when the flow is deployed to match the fields being provided for a particular action. Data incoming to the flows provides the fields mapped from

© Copyright HCL Ltd. 2021 All rights reserved

the calling application. A placeholder node is then required to map that to the fields that the action requires, in the appropriate format.

A typical scenario is that the placeholder node is between a Split node and a map that builds a request for the API being called:



In the implementation section the placeholder node needs to be identified and the desired output format must be specified. For example, an object in the implementations array could be:

```
{
    "name": "Upload Audiences",
    "run": {
        "flow": {
            "template": "Main Flow",
            "placeholders": [
                {
                    "node": "BuildJson",
                    "type": "map",
                    "generate_map": {
                        "from": {
                            "format": "csv_plus",
                            "has_header": true
                        },
                        "to": {
                            "format": "csv",
                            "has_header": true
                        }
                    }
                }
            ]
        }
    }
}
```

The placeholder object instructs deployment to replace the node named BuildJson with a generated map that will map from CSV data to CSV data. The format "csv_plus" means that the incoming data includes the mapped fields ***plus*** some number of identity fields. All connectors should specify csv_plus as the "from" format. The "to" format specifies that the incoming fields should be mapped to a CSV format for the output fields.

Supported output formats are:

Format	Type	Description
csv	Native	CSV data with optional header row. The generated schema creates a group with ',' delimiter

© Copyright HCL Ltd. 2021 All rights reserved

Format	Type	Description
csv_plus	Native	CSV data with identity field column. The generated schema creates a row group without syntax, and a ',' terminator on each field other than the identity field. This then permits the identity field to contain any number of commas.
json_object	Template	Creates a JSON template that is a JSON object which has a named single JSON array field.
json_array	Template	Creates a JSON template that is a JSON array of objects.
json_rows	Native	A native schema where each row is a JSON object, terminated by CR/LF. This format also allows for special type syntax to be computed. This is explained in the descriptions below.
json_rows_plus	Native	A native schema where each row is a JSON object followed by a delimiter "|$|" and then an identity field. Each row is terminated by a CR/LF
json_template	Template	JSON constructed from a template

3.7.4 Enumerations

Enumerations provide a way to populate values for a property by calling a REST API to get the values. For a property that has a dynamic drop down list, the name of the enumeration is specified in the property descriptor. For example:

```
{
    "label": "Template",
    "name": "template",
    "type": "string",
    "required": true,
    "enumeration": "template",
    "description": "The email template to use",
    "scope": "target_action"
}
```

The connector descriptor then requires a corresponding enumeration entry in the "rest_config" object:

```
    "rest_config": {
        "service": "Testapp",
        "enumerations": [
            {
                "name": "template",
                "endpoint": "Get Templates",
                "array_path": "templates",
                "value_path": "id",
                "label_path": "name"
            }
```

This informs the framework that to get the enumerated values, the endpoint "Get Templates" in service "Testapp" should be called. The fields array_path, value_path, label_path and qualifier_path provide the JSON paths to the corresponding fields in the JSON response.

© Copyright HCL Ltd. 2021 All rights reserved

All paths should use '.' notation to separate path elements. The array_path provides the path of the JSON array. If the root of the document is an array, then the path should be specified as an empty string (""). The other paths are specified relative to the array. The meaning of these is:

Field	Meaning
value_path	The value that is stored for the property
label_path	The displayable text shown in the dropdown list. This is optional. If not set the value will be displayed in the UI.
qualifier_path	The qualifier is displayed in parentheses in the dropdown list after the label. This can be used to provide differentiation when labels are not necessarily unique. This is optional.

3.7.5 Schema Mapping

For a given condition, the schema mapping defines a set of field definitions that should be returned when that condition is true. A condition is a property value having a certain value. The set of field definitions can be composed of static and dynamic elements. Static field definitions define fields that are always returned. Dynamic elements specify an endpoint that should be called to get field definitions, and additional attributes that define how to interpret the results of the endpoint.

The schema_mapping is an array of objects where each object has these fields:

Field	Required	Type	Description
name		string	A descriptive name for the mapping definition
condition		string	A condition which if true selects this mapping definition. The condition is required if there is more than one mapping but should not be specified if there is only one.
static		array of objects	One of more statically defined fields
dynamic		array of objects	One or more dynamically defined fields
format		string	The type of generated schema format. This can have the values "json" or "tree".
mode		string	Can have the values "request", "reply" or "both". Generated schema can be different for request and response mode. If not specified, or specified as "both", the schema mapping is valid for both request and response types.
jsonSchema		object	When the format is "json", this field provides a fixed JSON format that is returned as the schema.

A static definition has these fields

Field	Required	Type	Description
internal_name	Y	string	The name of the field
external_name	Y	string	The descriptive name of the field suitable for displaying in a UI

© Copyright HCL Ltd. 2021 All rights reserved

Field	Required	Type	Description
description		string	A description for the field which provides an explanation of the purpose of the field
type	Y	string	The type of the field. It must be one of: **text** – a textual field **integer** – an integer field **number** – any numeric field (integer or decimal) **table** – specifies that there are child field definitions. This is only applicable to dynamic definitions.
default		string	A default value for the field.
required		boolean	Whether the field is required or not. Defaults to false.

A dynamic definition has these fields:

Field	Required	Type	Description
endpoint	Y	string	The name of the endpoint to invoke to fetch the dynamic fields
paths	Y	object	The paths of the dynamic field attributes in the JSON response returned from the endpoint.
type_mapping		array of objects	A list of mappings from types of the endpoint's response to the internal types listed above under the type attribute of static definitions.
schema_path		string	When jsonSchema is configured, the generated dynamic fields will be set in the specified schema_path of static JSON.

The path object for a dynamic definition has these fields:

Field	Required	Type	Description
array	Y	string	The path of the array containing the field definition objects
child_array		string	The path of the array containing child field definitions relative to the array object. This is only applicable for a table type property.
internal_name	Y	string	The path of the field name
external_name		string	The path of the external displayable name of the field. If not specified, the external name will be set to the internal name.
default		string	The path of the default value of the field
description		string	The path of the description for the field
required		string	The path of the required attribute for the field. The value can be a boolean (true/false), or a string value("true"/"false")
type		string	The path of the field type. If not specified, the type of the field will be set to "string"

The type mapping object has these fields:

© Copyright HCL Ltd. 2021 All rights reserved

Field	Required	Type	Description
from	Y	string	The type returned in the response
to	Y	string	The type to which the type should map. The permitted values are shown above in the static definition object.

Consider the following schema mapping definition:

```
{
    "schema_mapping": [
        {
            "name": "Get merge fields",
            "condition": "{$operation} == \"new\"",
            "static": [
                {
                    "internal_name": "email",
                    "external_name": "Email",
                    "description": "Email address",
                    "type": "text",
                    "default": null,
                    "required": true
                }
            ],
            "dynamic": [
                {
                    "endpoint": "getMergeFields",
                    "paths": {
                        "array": "merge_fields",
                        "external_name": "name",
                        "internal_name": "tag",
                        "default": "default_value",
                        "required": "required",
                        "type": "type"
                    },
                    "type_mapping": [
                        {
                            "from": "int",
                            "to": "integer"
                        },
                        {
                            "from": "text",
                            "to": "string"
                        }
                    ]
                }
            ]
        }
    ]
}
```

An explanation of this JSON:

- The condition specifies that this schema mapping definition is applicable when the property operation has the value "new". Since there is only one schema mapping object in the array, the condition should not be specified, but is shown here simply to provide an example.

© Copyright HCL Ltd. 2021 All rights reserved

- This schema is defined by a single static field + the fields returned from the dynamic endpoint getMergeFields
- The static field defines an email field
- The dynamic definition specifies that the getMergeFields should be invoked to get the field definitions.
    - The paths object specifies the paths of the JSON fields providing the field definition
    - The types array specifies the mapping of the contents of the type field to the internal types.

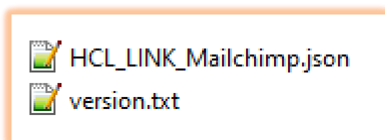
The paths in this object corresponds to this JSON response:

```
{
    "merge_fields": [
        {
            "name": "field1",
            "tag": "Field1",
            "default_value": "100",
            "required": true,
            "type": "int"
        }
    ]
}
```

3.8 Packaging & Translation

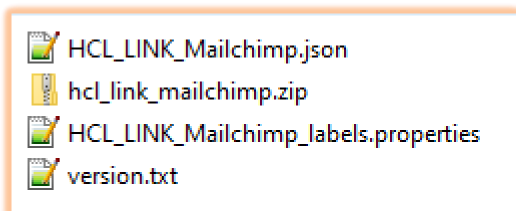
The connector package is comprised of the property descriptor, service definitions and the maps/flows implementing the connector's actions. The packager tool (see Packager) produces files in the correct structure when the 'package' option is specified.

The connector's source directory needs only to contain 2 files:



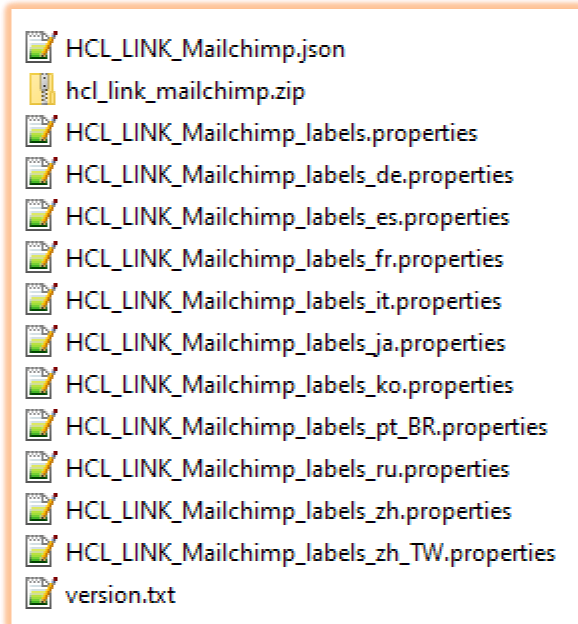
- The connector descriptor
- A text file named version.txt that contains the version name/number of the connector

When the packager is run command strings are extracted from the property descriptor and placed in a properties file named <descriptor>_labels.properties:



© Copyright HCL Ltd. 2021 All rights reserved

This file can then be translated to other languages, and the translated files should then be added back to the connector project:

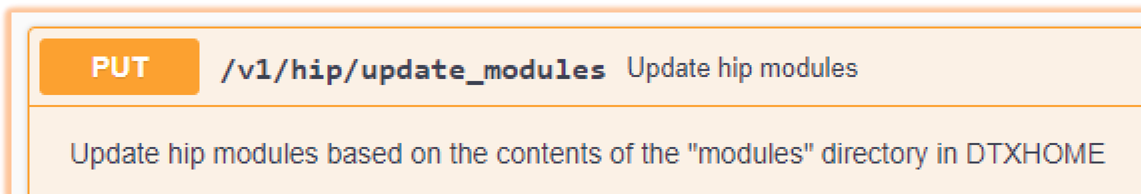


The packager builds a single zip file which can then be deployed to the Link server. This zip file contains localized descriptors, the project zip file, generated configuration files and optionally plugin classes. The name of the zip file is:

```
<connectorID>_connector_<version>.zip
```

3.9 Installation

The connector is deployed by copying the zip file to the 'modules' directory on the Link server. By default, this is /opt/hipmodules. When the server is restarted, any modules found in this directory are automatically installed. Alternatively, this API can be invoked in the runtime server:



The installer:

- Copies the configuration jar to both the design and runtime servers
- Copies the property descriptors to the design server
- Creates a project named _conntype_<connector> in design server, replacing the project if it already exists

© Copyright HCL Ltd. 2021 All rights reserved

3.10 End-to-End Testing

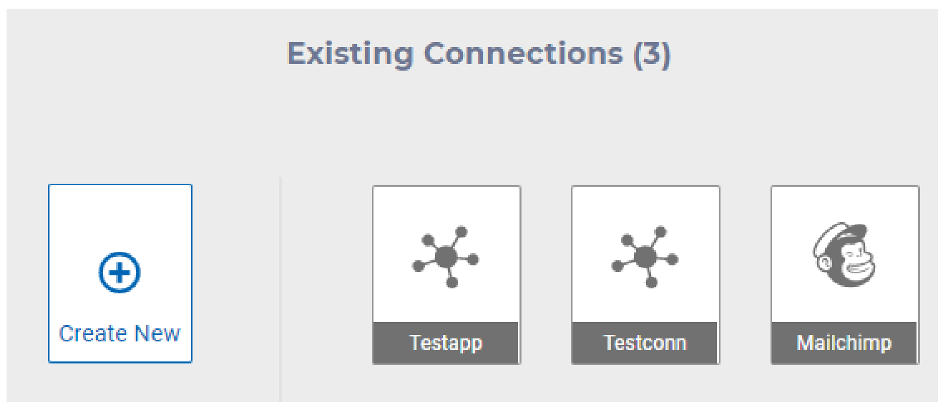
Once the connector has been installed it can be tested end-to-end by invoking the connection and action endpoints.

3.10.1.1 Defining a Connection

To define a connection for the resource, the following URL is invoked:

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

This assumes the 'testapp' application has been installed (see the Tutorial for set up instructions for testapp). This URL should open the existing connections UI. This lists any connections that have been defined and presents a Create New button through which a new connection can be created using the newly-developer connector.



The properties defined as connection properties in the descriptor are displayed in the properties of the new connection.

3.10.1.2 Defining an Action

After a connection has been created, an action can be created that uses the connection. The following URL is invoked to display the action user interface:

https://localhost/hch/_app_testapp/createAction?actionName={action}&connectionName={connection}&dev=true&deploy=true

The URL parameters should be set as follows:

- connectionName – the name of the connection to which the action belongs
- actionName – the name of a new action

When the action properties have been populated, the action is deployed to the runtime environment. The action's URL can then be found on the server's Swagger page at this URL:

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

© Copyright HCL Ltd. 2021 All rights reserved

PUT /v2/run/tc_action4/info tc_action4/info

POST /v2/run/tc_action4/info This is an asynchronous invocation. tc_action4/info

PUT /v2/run/tc_action4/run This flow uses split/join nodes to invoke send API in parallel

POST /v2/run/tc_action4/run This is an asynchronous invocation. This flow uses split/join nodes to invoke send API in parallel

The action can be invoked directly from the Swagger page by expanding an API and clicking on the 'Try it out' button.

4 Packager

The packager tool is used to validate connector artifacts, and to produce the necessary files required to install a connector.

The tool is invoked by running packager.bat on Windows or packager.sh on Linux. If invoked with no arguments it displays the command syntax:

```
Packager <conn-dir> <output-dir> [-vc] [-vd] [-ld] [-p] [-ep project]
[-h host] [-u user] [-pw pwd]
   -vc : validate configuration file
   -vd : validate descriptor
   -ld : localize descriptor
   -p  : package
   -ep : export named project
   -h  : hostname of Link design server. E.g. https://localhost:8443/
   -u  : username for Link
   -pw : password for Link
At least one command option must be specified.
```

The arguments are:

- conn-dir - The directory containing the source connector files.
- output-dir - The directory to write connector artifacts to

The command options are:

- -vc - validate a configuration file. (Configuration files are not discussed in this document because they have been superseded by service definitions)
- -vd - validate a connector descriptor file.
- -ld - performs the localization step of packaging, generating a labels properties file.
- -p - package the connector
- -ep - export the named project to a zip file

If -ep is specified, then -h, -u and -pw must also be specified so that the packager tool can connect to the server and export the project.

© Copyright HCL Ltd. 2021 All rights reserved

For example, to package a Mailchimp connector:

```
packager.bat \Connectors\Mailchimp \Installs\Mailchimp -p -ep Mailchimp -h
https://localhost:8443/ -u admin -pw ****
```

This command will perform the following operations:

1. Export the "Mailchimp" project to a zip file named <connector-id>.zip in the source directory. The connector-id comes from the "id" attribute in the connector descriptor
2. If the connector descriptor references a service, the service definition is exported to the services directory under the source directory
3. Generates a configuration file based on the service definition
4. Performs localization of the connector description, extracting labels and descriptions, and building localized descriptors for all language bundles that are present in the source
5. Zip everything up into the target directory to produce an installable connector

5 Connector Descriptor Specification

The connector descriptor is a JSON file that provides the following information:

- General information about the connector (name, description etc.)
- Definitions for the connector's properties
- One or more implementation descriptors that specify what flow to invoke for run and results actions
- Definitions of enumerations and schema mappings

An example of a descriptor is provided in the 'testconn' connector:

```
{
  "id": "testconn",
  "name": "Testconn",
  "type": "utility",
  "category":  "email",
  "description": "Testconn Sample",
  "contexts": [
    "target"
  ],
  "testable": true,
  "properties": [
    {
      "name": "base_url",
      "label": "Base URL",
      "description": "The base URL of the service",
      "type": "string",
      "required": true,
      "scope": "connection"
    },
    {
      "name": "username",
```

© Copyright HCL Ltd. 2021 All rights reserved

```
        "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": "Template",
        "name": "template",
        "type": "string",
        "required": true,
        "enumeration": "template",
        "description": "The email template to use",
        "scope": "target_action"
      }
    ],
    "rest_config": {
      "service": "Testapp",
      "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
            },
            {
```

© Copyright HCL Ltd. 2021 All rights reserved

```
              "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"
          }
        }
      }
    ]
}
```

This section defines the JSON elements defined above.

5.1 General Fields

Field	Description
id	The id of the connector. This uniquely identifies the connector.
name	The connector's displayable name
type	The type of the connector. This should be set to "app"
category	The connector's category. The value should be one of "email", "sms", "push", "crm", "adtech", or "custom"
description	A displayable description for the connector
contexts	An array of supported directions. Can include "source" and "target"
properties	An array of property definitions. The attributes of these are given below
rest_config	Specifications of the service, enumerations and schema mappings. This is detailed fully below.

5.2 Property Descriptors

Each property definition has the following fields:

© Copyright HCL Ltd. 2021 All rights reserved

Attribute	Required?	Description
name	Y	The property name
label	Y	A displayable label
description	Y	A description for the property. This is displayed as a tooltip in the UI.
type	Y	The type of the property. Permitted values: string, integer, boolean, datetime or file.
scope	Y	Whether the scope is a connection or action property. If an action property, one can further specify whether it pertains to source or target context, or both contexts. Permitted values: connection, source_action, target_action, any_action "any action" means pertains to both source and target
required	Y	A boolean field specifying whether the property is always required or is optional.
masked		If a property is a string, its value can be masked (e.g. for a password) if this boolean field is set to true
default		A default value

values		A string property can contain a list of values that are displayed in a drop down list. The values field is a JSON array containing objects that have value and label fields. The value is the stored property value, and the label is what is displayed in the UI. For example:
		
		`"values": [`
		`{`
		`"value": "hours",`
		`"label": "Hours"`
		`},`
		`{`
		`"value": "minutes",`
		`"label": "Minutes"`
		`},`
		`{`
		`"value": "seconds",`
		`"label": "Seconds"`
		`}`
		`]`

enumeration		If the property has a set of values that can be dynamically obtained, the name of the enumeration is specified for the property.
enabled		An expression that determines whether a property should be displayed or not. See enablement section below this table.
multiline		If a string property has multiline set to true, a text area should be displayed allowing for newline characters.
signed		For integer properties, true if the value permits a sign. By default, integers are unsigned.
min_value		For integer properties, the minimum permitted integer value
max_value		For integer properties, the maximum permitted integer value

© Copyright HCL Ltd. 2021 All rights reserved

defaults		A list of defaults to provide different defaults based on conditional expressions

validation		A validation expression to be applied when a property value is changed. This is an object which contains a regular expression and a corresponding message to be displayed if the value does not match the regular expression. For example:
		
		`"validation": {`
		`    "message": "The value cannot contain only whitespace",`
		`    "regex": "(?!^ +$)^.+$"`
		`}`

enumeration		If a property has enumeration attribute this means that the adapter or importer can dynamically list enumerated values. The UI should provide the means to call the server's enumerate API whereby it passes the enumeration value to identify what is being enumerated.

datetime_properties		If the property is of type datetime, this object provides parameters that configure the date picker widget in the UI. For example:
		
		`"type": "datetime",`
		`"datetime_properties": {`
		`   "select_date": true,`
		`   "select_time": true,`
		`   "future_date": true,`
		`   "minute_increment": 15`
		`}`
		
		The fields of datetime_properties are:
		• select_date – the widget should allow selection of a date
		• select_time – the widget should allow selection of a time
		• future_date – the widget should only permit future dates/times
		• minute_increment – the time selection should only permit multiples of this value

multicolumns		Determines whether properties are displayed in a single column or in 2 columns. It defaults to true.
read_only		If true, properties displayed as read-only.

NOTE: If connection/action properties does not exist for a connector, define the following tag in the connector descriptor file:

```
"properties": [].
```

5.2.1 Property enablement

The "enabled" field allows one to specify an expression that determines whether a property is displayed or hidden, based on the value of one or more other properties. For example, if a property should only be displayed if a boolean property "manual_mode" is true, the expression would be:

```
"enabled": "manual_mode == true"
```

Expressions are of the form: <property-name> <operator> <value | values>where operator is one of:

Operator	Meaning
==	Equals
!=	Not equals
in	In (i.e. is one of the listed values)
!in	Not in (i.e. is not one of the listed values)
<	Less than
<=	Less than or equals
>	Greater than
>=	Greater than or equals

For in and !in operators the value list is enclosed in [ ] with comma-separated values. For example:

```
"enabled": "month in [jan,feb,mar]"
```

More complex expressions can be specified by using AND and OR operators:

```
"enabled": "manual_mode == true AND month in [jan,feb,mar]
```

6 Java plugin

The Java plugin is an optional component of a connector that can be used to manipulate the request, or response of endpoints executed through the REST adapter.

6.1 Prerequisites

- Maven Repository
- Link libraries:
    - **m4rest.jar**
    - **m4base.jar**
    - **restutils.jar**

These dependencies can be copied from the Link install and then must be placed in the local maven repository.

6.2 Create a new Maven project

1. Create a new maven project inside the connector's source folder and add the mentioned dependencies in the project’s **pom.xml** file.

© Copyright HCL Ltd. 2021 All rights reserved

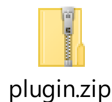
Name	Date modified	Type	Size
plugin	27-05-2021 20:38	File folder	
service	26-05-2021 20:38	File folder	
HCL_LINK_Unipix	27-05-2021 20:17	JSON File	4 KB
hcl_link_unipix	27-05-2021 15:18	Compressed (zipp...	347 KB
HCL_LINK_Unipix_labels	27-05-2021 20:20	PROPERTIES File	2 KB
version	27-05-2021 18:04	Text Document	1 KB

› Data (D:) › repo › hch › connectors › unipix › plugin

Name	Date modified	Type	Size
.settings	27-05-2021 18:00	File folder	
src	27-05-2021 18:00	File folder	
.classpath	09-04-2021 15:21	CLASSPATH File	2 KB
.project	27-05-2021 09:34	PROJECT File	1 KB
pom	27-05-2021 09:34	XML Document	2 KB

Note: The project name should be "**plugin**" as shown in above.

Below is the plugin's sample project.



plugin.zip

The sample can be considered as reference to create plugin projects with connector specific names and logics to customize the request/response.

6.3 Pom.xml from the plugin project

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
       <modelVersion>4.0.0</modelVersion>

       <groupId>hcl</groupId>
       <artifactId>sample-plugin</artifactId>
       <version>0.0.1</version>
       <packaging>jar</packaging>

       <name>Sample Plugin</name>
```

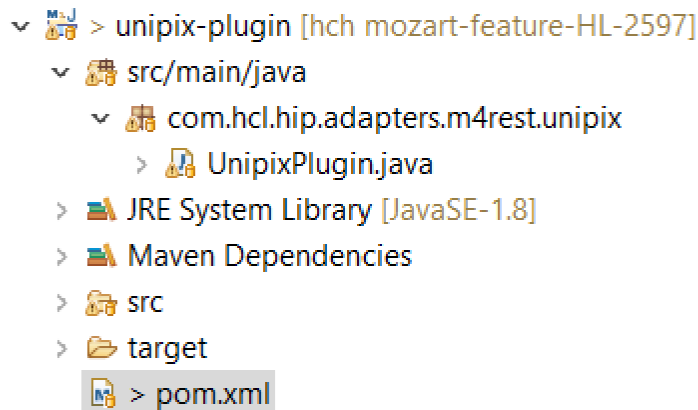
© Copyright HCL Ltd. 2021 All rights reserved

```xml
	<url>http://maven.apache.org</url>

	<properties>
		<build.version>${env.VERSION}</build.version>
		<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
		<java.version>1.8</java.version>
		<maven.compiler.source>${java.version}</maven.compiler.source>
		<maven.compiler.target>${java.version}</maven.compiler.target>
		<jackson-version>2.10.1</jackson-version>
	</properties>

	<dependencies>
		<dependency>
			<groupId>junit</groupId>
			<artifactId>junit</artifactId>
			<version>3.8.1</version>
			<scope>test</scope>
		</dependency>
		<dependency>
			<groupId>hcl.tx</groupId>
			<artifactId>m4rest</artifactId>
			<version>${env.VERSION}</version>
		</dependency>
		<dependency>
			<groupId>hcl.tx</groupId>
			<artifactId>m4base</artifactId>
			<version>${env.VERSION}</version>
		</dependency>
		<dependency>
			<groupId>hcl.tx</groupId>
			<artifactId>restutils</artifactId>
			<version>${env.VERSION}3</version>
		</dependency>
		<dependency>
			<groupId>com.fasterxml.jackson.datatype</groupId>
			<artifactId>jackson-datatype-joda</artifactId>
			<version>${jackson-version}</version>
		</dependency>
		<dependency>
			<groupId>com.fasterxml.jackson.core</groupId>
			<artifactId>jackson-core</artifactId>
			<version>${jackson-version}</version>
		</dependency>
	</dependencies>
</project>
```

2. Create a class, then extend it with m4rest's **M4RestPlugin** class. Then we need to override its methods to provide our own implementation of customizing request/response.

© Copyright HCL Ltd. 2021 All rights reserved

6.4 Eclipse project structure



6.5 Override M4RestPlugin methods

Below is an example of a class that overrides the **M4RestPlugin** methods to customize request and response .

```
package com.hcl.hip.adapters.m4rest.sample;
import java.util.Properties;
import com.hcl.hip.adapters.m4rest.M4RestPlugin;
import com.hcl.hip.adapters.m4rest.RestResponse;
import com.hcl.hip.tools.restutils.model.RestRequest;

public class SamplePlugin extends M4RestPlugin
{
    @Override
    public RestRequest customizeRequest(String endpoint, RestRequest
request, Properties props, byte[] data)
    {
           // Write your own code to customize the request
           return request;
    }

    @Override
    public RestResponse customizeResponse(String endpointName,
Properties props, RestResponse response)
    {
           // Write your own code to customize the response
           return response;
    }
}
```

The examples shown here are two commonly used methods of the use of **M4RestPlugin** for customizing the request and response. There are other methods to modify Endpoint that could be used depending upon the use case.

© Copyright HCL Ltd. 2021 All rights reserved

6.6 Package the connector

Follow these procedures to package a connector with a plugin:

1. Build the plugin project first. Doing so ensures that the **.jar** is created inside the target directory of the project.
2. Package the connector using the packager tool.

If a connector does not have plugin in it, it can be directly packaged.

It the connector does have a plugin in it, you must follow these steps, otherwise the plugin would not be included in the packaged **.zip** folder.

© Copyright HCL Ltd. 2021 All rights reserved