TIBCO Cloud™ Integration - Flogo® (PAYG)
User's Guide

Version 2.11.0

January 2021

Contents

Figures

TIBCO Documentation and Support Services

How to Access TIBCO Documentation

Documentation for TIBCO products is available on the TIBCO Product Documentation website, mainly in HTML and PDF formats.

The TIBCO Product Documentation website is updated frequently and is more current than any other documentation included with the product. To access the latest documentation, visit https://docs.tibco.com.

Product-Specific Documentation

Documentation for TIBCO Cloud™ Integration - Flogo® (PAYG) is available on the TIBCO Cloud Integration - Flogo (PAYG) Product Documentation page.

The following documents can be found on the TIBCO Documentation site:

- *TIBCO Cloud™ Integration - Flogo® (PAYG) Release Notes*
- *TIBCO Cloud™ Integration - Flogo® (PAYG) Getting Started*
- *TIBCO Cloud™ Integration - Flogo® (PAYG) User's Guide*
- *TIBCO Cloud™ Integration - Flogo® (PAYG) Activities and Triggers Guide*

How to Contact TIBCO Support

You can contact TIBCO Support in the following ways:

- For an overview of TIBCO Support, visit http://www.tibco.com/services/support.
- For accessing the Support Knowledge Base and getting personalized content about products you are interested in, visit the TIBCO Support portal at https://support.tibco.com.
- For creating a Support case, you must have a valid maintenance or support contract with TIBCO. You also need a user name and password to log in to https://support.tibco.com. If you do not have a user name, you can request one by clicking Register on the website.

How to Join TIBCO Community

TIBCO Community is the official channel for TIBCO customers, partners, and employee subject matter experts to share and access their collective experience. TIBCO Community offers access to Q&A forums, product wikis, and best practices. It also offers access to extensions, adapters, solution accelerators, and tools that extend and enable customers to gain full value from TIBCO products. In addition, users can submit and vote on feature requests from within the TIBCO Ideas Portal. For a free registration, go to https://community.tibco.com.

Introduction

Concepts

This section describes the main concepts used in the TIBCO Cloud Integration - Flogo (PAYG) environment.

Apps

Flogo apps are developed as event-driven apps using triggers and actions and contain the logic to process incoming events. A Flogo app consists of one or more triggers and one or more flows.

Trigger

Triggers receive events from external sources such as Kafka, Salesforce, GraphQL and so on. Handlers residing in the triggers, dispatch events to flows. TIBCO Cloud Integration - Flogo (PAYG) provides a set of out-of-the-box triggers as well as a range of connectors for receiving events from a variety of external systems.

Flow

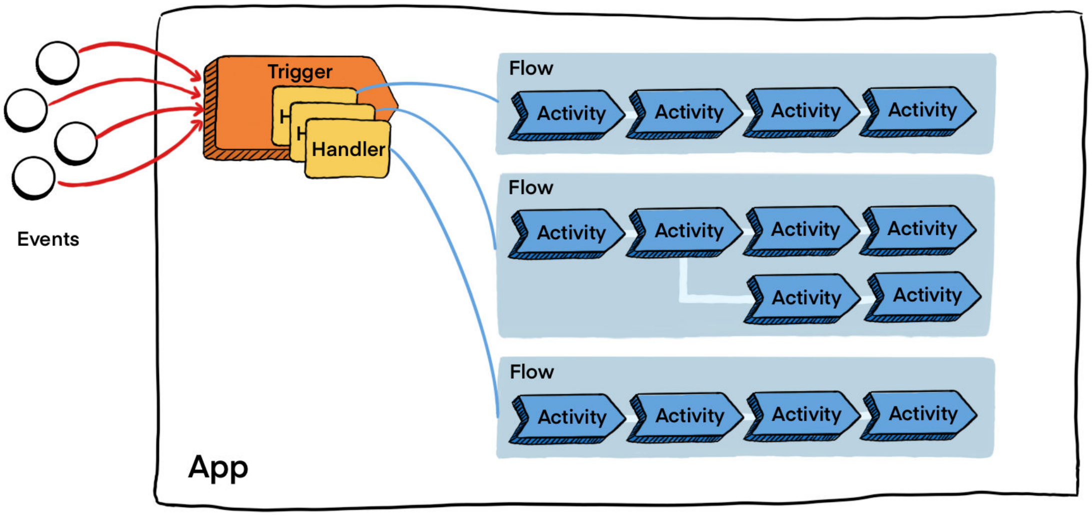
The flow allows you to implement the business logic as a process. You can visually design and test the flows using the Web UI. A flow can consist of one or more activities that perform a specific task. Activities are linked in order to facilitate flow of data between them and can contain conditional logic for branching. Each flow is also connected to a default error handler. A Flogo app can have one or more flows. A flow can be activated by one or more Triggers within the app.

Activity

Activities perform specific tasks within the flow. A flow typically contains multiple activities.

How TIBCO Cloud Integration - Flogo (PAYG) Works

The trigger consists of one or more handlers that serve as the means of communication between the trigger and the flow. When the trigger receives an event, the trigger uses the respective flow handlers to pass the data from the event on to the flow in the form of flow input. The business logic in the flow then can use the event data coming in through the flow input. When the trigger expects a reply from the flow, the data from the flow is passed on to the trigger in the form of flow output. A flow can contain one or more conditional branches.



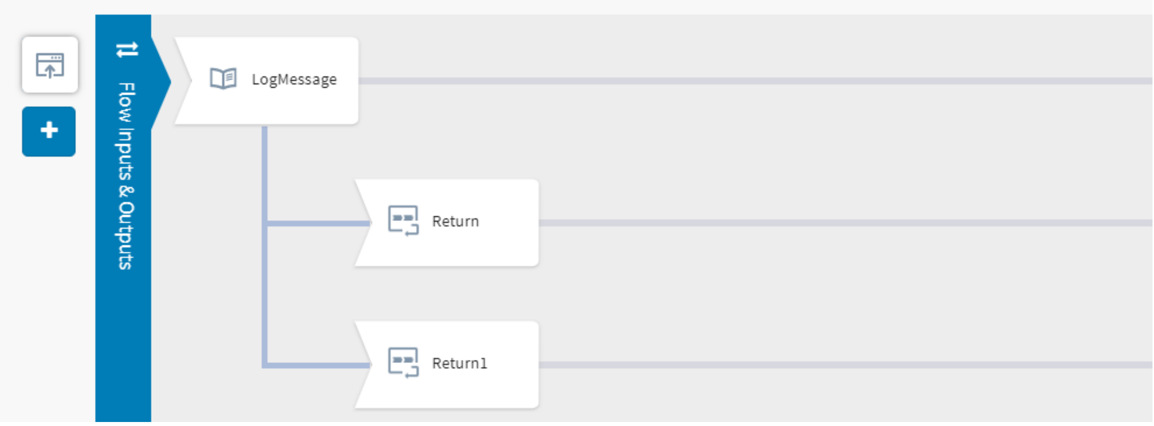
In a nutshell, to use TIBCO Cloud Integration - Flogo (PAYG), you have to follow these steps:

1. Create an app.
2. Create a flow in your app.
3. Add one or more activities to the flow and configure them.
4. Optionally, add a trigger to your flow. You can add one or more triggers to a flow as and when you need them.
5. Build your app.

Creating your First REST API

This tutorial walks you through building a simple REST service in TIBCO Cloud™ Integration - Flogo® (PAYG). It walks you through creating a basic airlines flight reservation app which takes input from a user (a potential passenger), checks to see if the passenger's last name is "Jones" and for passengers with last name "Jones" upgrades them to Business Class.

The app contains a REST flow, FlightBookings, which implements the POST HTTP operation to book a seat on the flight for the passenger requesting it. The flow is triggered by the **ReceiveHTTPMessage** REST trigger which listens for a request to book a seat that comes in from a passenger. The flow contains a **LogMessage** activity that you configure to log a custom message when a request has been received successfully. The **LogMessage** activity has two branches. Each branch must have its own **Return** activity as the last activity in the branch.



When a request comes in:

1. The first branch accepts requests with any last name that appears in the REST request.
2. The second branch then checks to see if the last name in the request is "Jones". You configure this check (condition) in the **Branch Mapping Settings** for this branch. If the last name is "Jones" then the flow outputs the passenger details with the Class element automatically set to Business Class meaning that the passenger's seat has been booked in Business Class. You configure this action of automatically setting the Class element to BusinessClass for passengers with last name "Jones" in the **Return** activity for this branch.
3. If the last name received is not "Jones" the control is transferred to the first branch whose flow outputs the details of the request as received with the Class to Economy.

Creating the JSON Schema for REST Request and Response

In this tutorial, a simple JSON schema is created for the REST request that the service receives and the response that the service sends back. The following is the structure of the JSON schema used in this tutorial. This JSON message is converted internally into a JSON schema:

```
{
  "Class" : "string",
  "Cost" : 0,
  "DepartureDate" : "2017-05-27",
  "DeparturePoint" : "string",
  "Destination" : "string",
  "FirstName" : "string",
  "Id" : 0,
  "LastName" : "string"
}
```

High-level Steps in the Tutorial

The high-level steps for creating and configuring the airline flight reservation service REST app in this tutorial are as follows:

1. Create a New TIBCO Flogo® App
2. Create a Flow in the App with a REST Trigger
3. Map Trigger Output to Flow Input. This is the bridge between the trigger and the flow where the trigger passes on the request data to the flow input.
4. Map Flow Output to Trigger Reply. This is the bridge between the flow output and the response that the trigger sends back to the HTTP request it received. After the flow has finished executing, the output of the flow execution is passed back to the trigger by the **Return** activity. Hence, we map the flow output to the trigger reply. This mapping is done in the trigger configuration.
5. Add a Log Message Activity to the Flow and configure a message that the activity must log in the logs for the app as soon as it receives a request.
6. Add the First Branch and Configure It to accept any last name that appears in the REST request.
7. Add a Second Branch to Check for Last Name "Jones". If the last name of the passenger is "Jones", this branch is executed and the passenger is placed in Business Class.
8. Validate the App to make sure that there are no errors or warnings in any flows or activities.
9. Build the App
10. Test the App

Step 1: Create a New TIBCO Flogo® App

Create a new TIBCO Flogo® App in TIBCO Cloud Integration - Flogo (PAYG).

Follow these steps to create a new app:

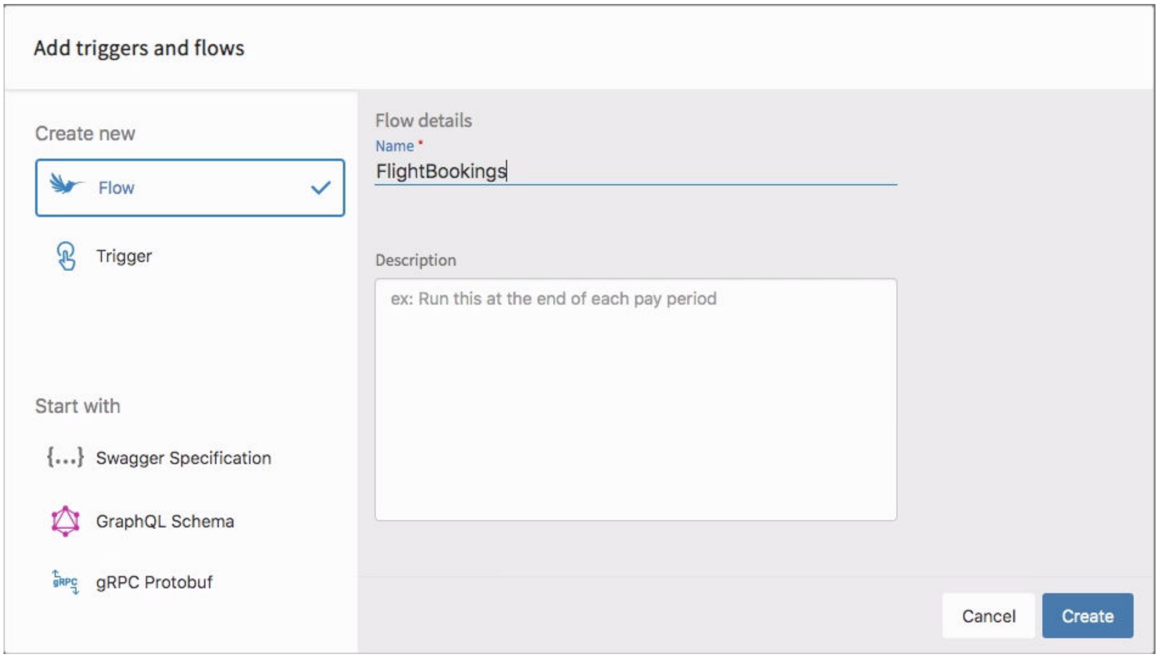
1. Open the **Apps** tab in TIBCO Cloud Integration - Flogo (PAYG).
2. Click **Create/Import app**.
3. By default, a Flogo app is named `New_Flogo_App_<sequential-number>` where the *sequential-number* is the next number of a new app being created. Click the default app name next to the Flogo app icon to make it editable. Edit the app name to `FlightApp` and click anywhere outside the name to persist your change.

Step 2: Create a Flow in the App with the REST Trigger (Receive HTTP Message)

Every app must have at least one flow. Create a new flow with the REST trigger. The **ReceiveHTTPMessage** REST trigger listens for an incoming REST request that contains the details of the passenger who wants to book a flight. You configure the expected fields for the request in the REST trigger in JSON schema format.

Follow these steps to create a flow:

1. Click **Create**. The **Flow** option is selected by default in the **Add triggers and flows** dialog.
2. Enter `FlightBookings` in the **Name** text box and optionally a description for the flow in the **Description** text box and click **Create**.

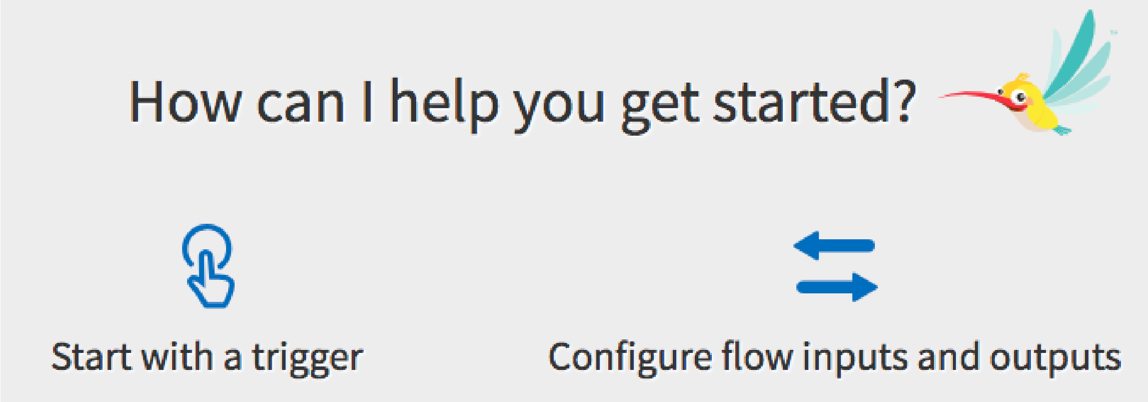


Add triggers and flows

Name * FlightBookings

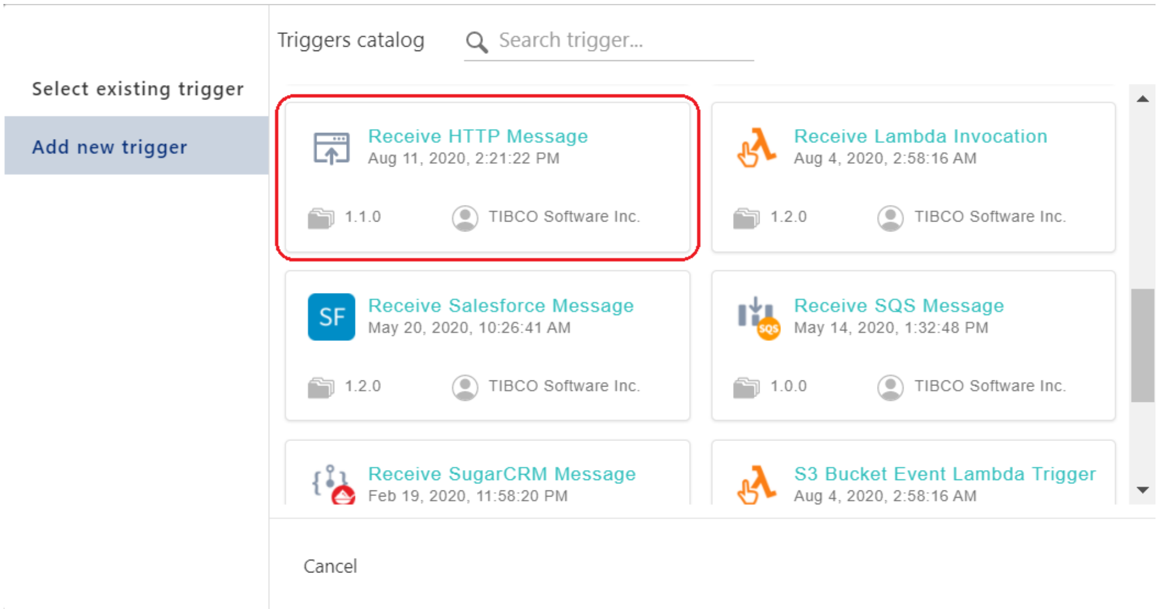
Description ex: Run this at the end of each pay period

3. Click **Start with a trigger**.



How can I help you get started?
Start with a trigger
Configure flow inputs and outputs

4. Click the **Receive HTTP Message** card in the **Triggers catalog**.



Triggers catalog

Search trigger...

Select existing trigger

Add new trigger

Receive HTTP Message
Aug 11, 2020, 2:21:22 PM
1.1.0 TIBCO Software Inc.

Receive Lambda Invocation
Aug 4, 2020, 2:58:16 AM
1.2.0 TIBCO Software Inc.

Receive Salesforce Message
May 20, 2020, 10:26:41 AM
1.2.0 TIBCO Software Inc.

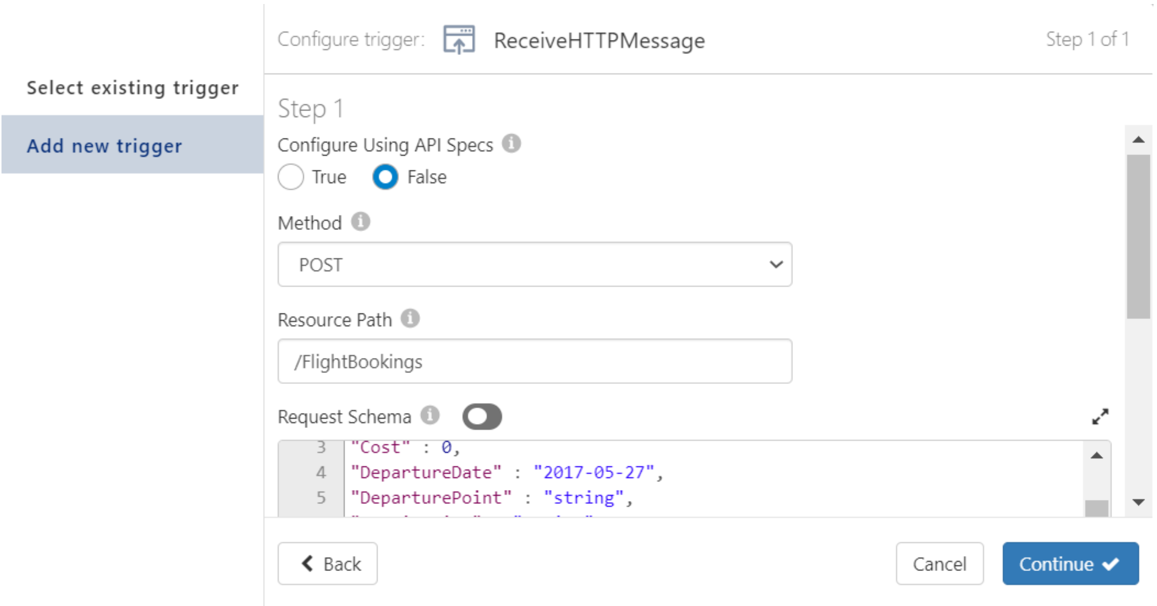
Receive SQS Message
May 14, 2020, 1:32:48 PM
1.0.0 TIBCO Software Inc.

Receive SugarCRM Message
Feb 19, 2020, 11:58:20 PM

S3 Bucket Event Lambda Trigger
Aug 4, 2020, 2:58:16 AM

Cancel

5. In the Configure trigger: ReceiveHTTPMessage dialog, do the following:



Configure trigger: ReceiveHTTPMessage Step 1 of 1

Select existing trigger

Add new trigger

Step 1

Configure Using API Specs

○ True ● False

Method

POST

Resource Path

/FlightBookings

Request Schema

```
"Cost" : 0,
"DepartureDate" : "2017-05-27",
"DeparturePoint" : "string",
```

Back Cancel Continue

a. Select **POST** as the **Method**.

b. Enter `/FlightBookings` in the **Resource path** text box.

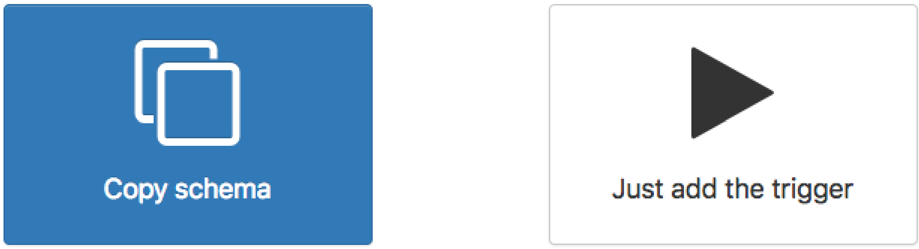
c. Enter the following schema that an incoming request must adhere to in the **Enter a JSON Schema or an example of your JSON message** box.

> Make sure to use straight quotes when entering the schema elements and values.

```
{
  "Class" : "string",
  "Cost" : 0,
  "DepartureDate" : "2017-05-27",
  "DeparturePoint" : "string",
  "Destination" : "string",
  "FirstName" : "string",
  "Id" : 0,
  "LastName" : "string"
}
```

d. Click **Continue**.

6. Select **Copy Schema** when prompted as shown below.

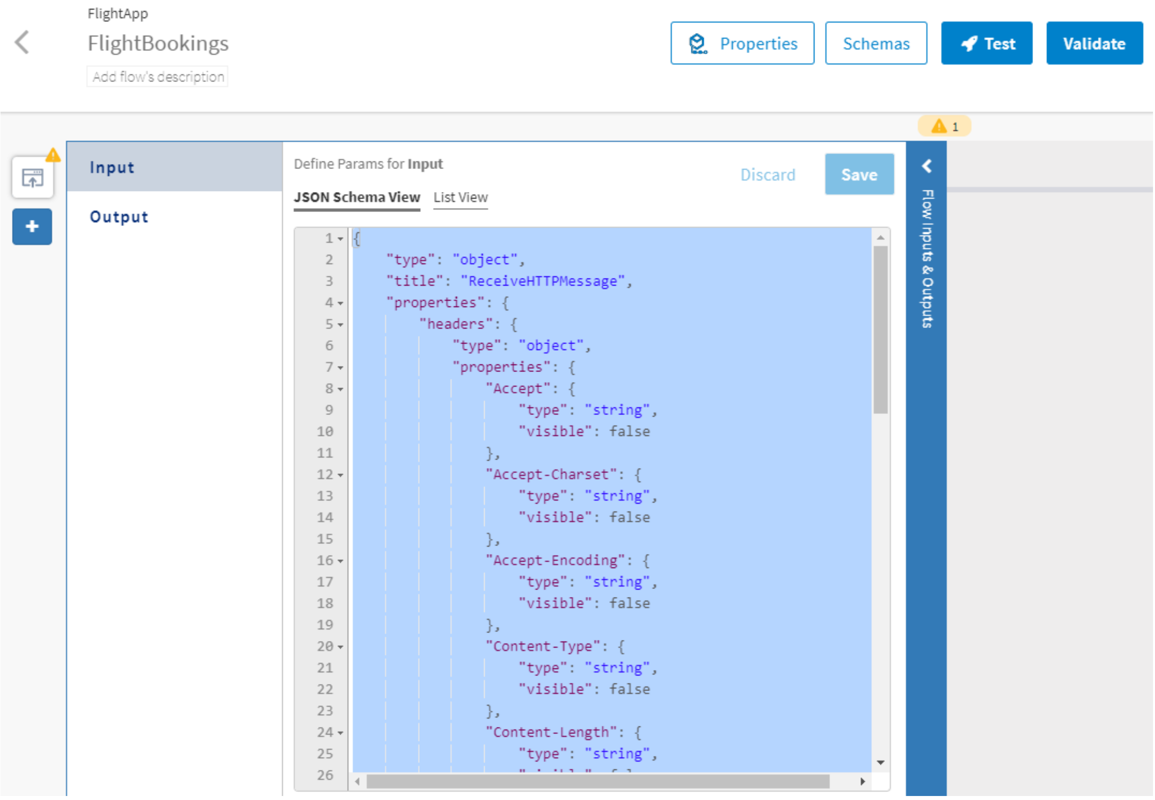


The schema that you entered when creating the trigger automatically gets copied to the following locations when the trigger is added:

- Flow input in the **Input Settings** tab of the **Flow Inputs & Outputs** accordion tab.
- It is displayed in a tree format in the **Map to Flow Inputs** tab of the trigger. This allows you to map the elements from the trigger output to flow input elements, such that the trigger output feeds into the flow input.
- The **Reply Settings** tab of the trigger, if the trigger has a reply. In the next step, you map the flow output schema elements to the trigger reply. By doing so, you send the output from the flow back to the trigger such that it becomes the trigger reply.

A new flow is created attached to a REST trigger.

Your flow should look similar to the following:



```
{
    "type": "object",
    "title": "ReceiveHTTPMessage",
    "properties": {
        "headers": {
            "type": "object",
            "properties": {
                "Accept": {
                    "type": "string",
                    "visible": false
                },
                "Accept-Charset": {
                    "type": "string",
                    "visible": false
                },
                "Accept-Encoding": {
                    "type": "string",
                    "visible": false
                },
                "Content-Type": {
                    "type": "string",
                    "visible": false
                },
                "Content-Length": {
                    "type": "string",
```

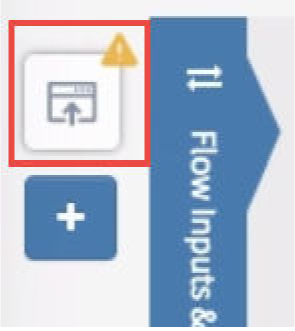
7. Collapse the **Flow Inputs & Outputs** accordion tab by clicking the left-facing arrow above the tab name in the blue vertical bar.

Step 3: Map Trigger Output to Flow Input

When TIBCO Cloud Integration - Flogo (PAYG) receives a flight booking request from a passenger (a REST request), the data from the request is output by the **ReceiveHTTPMessage** REST trigger. For the request to be processed, this output must be consumed by the flow in the form of flow input. Hence, you must map the trigger output to the flow input.

To do so, follow these steps:

1. Click the REST trigger on the top left corner of the flow to open its configuration dialog.



2. Click the **Map to Flow Inputs** tab.
3. Click **headers** under **Flow Input**, then click **$trigger** > **headers** under Trigger Output to map the headers.
4. Click **body** under **Flow Input**, then click **$trigger** > **body** to map the elements under **body**.
5. Click **Save**.

Step 4: Map Flow Output to Trigger Reply

When the flow has finished executing, its output must be sent back to the trigger for the trigger to send a reply to the REST request initiator. Hence, the flow output data must be mapped to the trigger reply which returns the result of the flow execution to the REST request initiator.

To map the flow output to the trigger reply:

1. Click the **Map from Flow Outputs** tab.

   a. Click the **code** under **Trigger Reply** to open the mapper.

   b. Click **$flow** > **code** under **Flow Output**.

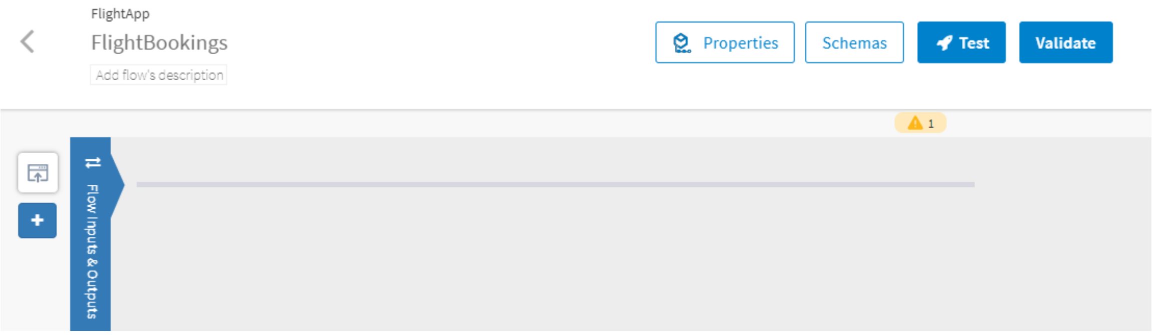
   c. Click **data** under **Trigger Reply**.

   d. Click **$flow** > **data** under **Flow Output**.

   e. Click **Save** to save your changes.

   f. Click **x** to close the dialog.

   The flow should look like the following:



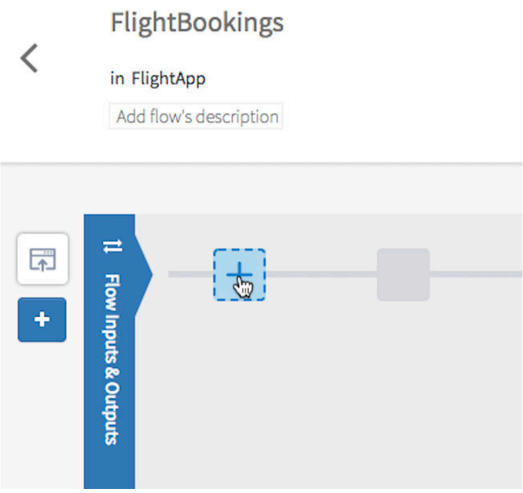
Step 5: Add a Log Message Activity to the Flow

The flow uses the **LogMessage** activity to log an entry in the app logs when the trigger receives a request from the passenger that reaches the trigger in the form of a REST request.

Follow these steps to add a **LogMessage** activity:

1. Add a new **LogMessage** activity from the **General** tab and configure it. To do so,

    a. Hover your mouse cursor to the right of the **Flow Inputs & Outputs** tab and click +.



    b. In the **Add Activity** dialog, click **General**, then click **Log Message**.

    c. Configure the **LogMessage** activity with a message to log when it receives an incoming request from the **ReceiveHTTPMessage** trigger. To do so:

        a. Click the **Input** tab.

        b. Click **message** to open the mapper to the right.

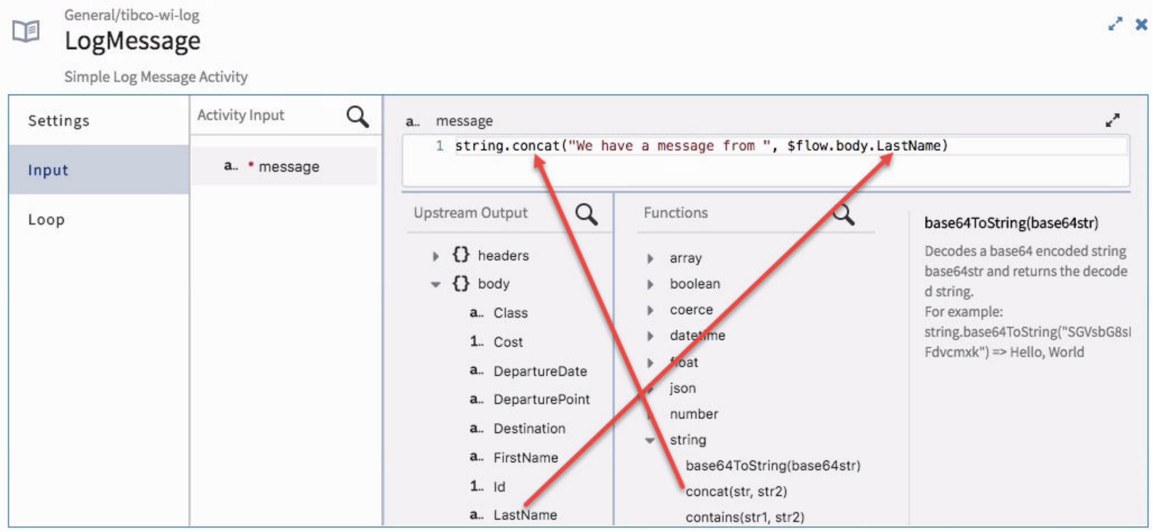
        c. Configure a message to be logged by the **LogMessage** activity when the flow receives the input from the request that the trigger received and passed on to the flow. To configure the message, expand the **string** category under **Functions** and click **concat(str, str2)** to add this function to the **message** text box.

        d. Select **str** in the and replace it by entering `"We have received a message from "` (include the quotes too).

        e. Replace **str2** with the last name of the passenger who booked the flight. The last name of the passenger is passed on from the trigger to the flow. We had mapped this trigger output to flow input in step 3 above. Hence it is now available for mapping under $flow in Upstream Output.

To map the **LastName** do the following:

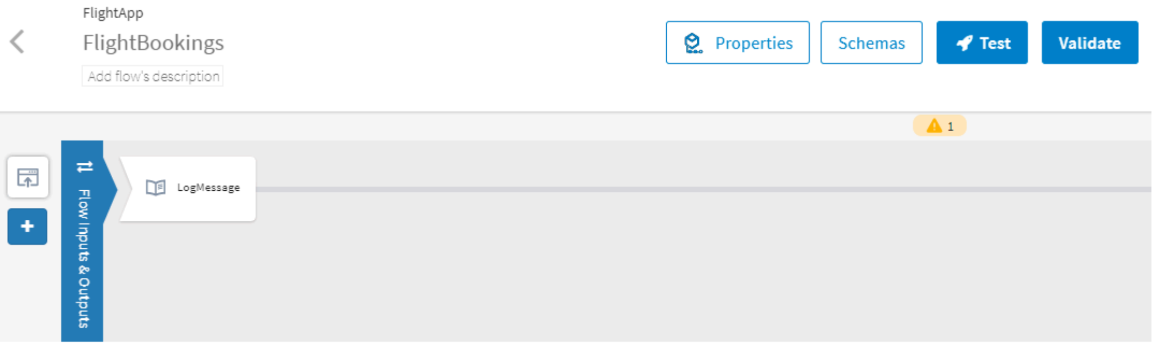
1. Select **str2**.
2. Expand **$flow** > **body** under **Upstream Output**.
3. Click **LastName**.
4. Click **Save** to save your changes.



```
string.concat("We have a message from ", $flow.body.LastName)
```

d. Click the **x** on the upper right side of the **LogMessage** box to close it. The **LogMessage** activity is added to the right of the **Flow Inputs & Outputs** tab.

Your flow should now look like the following:

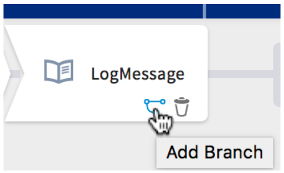


Step 6: Add the First Branch and Configure It

We want the flight seat booking to be based on the last name of the passenger. If the last name is "Jones" we want to book the passenger in the Business Class. If the passenger's last name is anything other than "Jones", we want the passenger's seat to be booked in the Economy class. To accomplish this, use the conditional branching feature in TIBCO Cloud Integration - Flogo (PAYG). Add a branch from the **LogMessage** activity.

Do the following to add a branch and configure its condition to accept any last name:

1. Hover your mouse cursor over the **LogMessage** activity and click .



The branch gets added with the **Add Activity** dialog open.

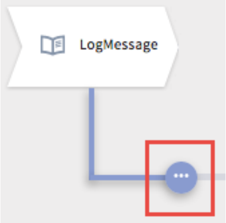
2. Click the **x** on the upper right corner of the **Add Activity** dialog to close it.
3. Add a **Return** activity to the branch. To do so,

   a. Hover your mouse cursor to the end of the branch and click the (+) icon.

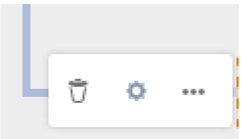
   b. Scroll to the **Default** category and click it.

   c. Click the **Return** card to add the activity.

   d. Click the **x** to close the configuration dialog. You must now configure the Return activity with a condition to read the last name of the passenger.
4. Hover your mouse cursor to the end of the branch until you see a button with three dots placed horizontally.



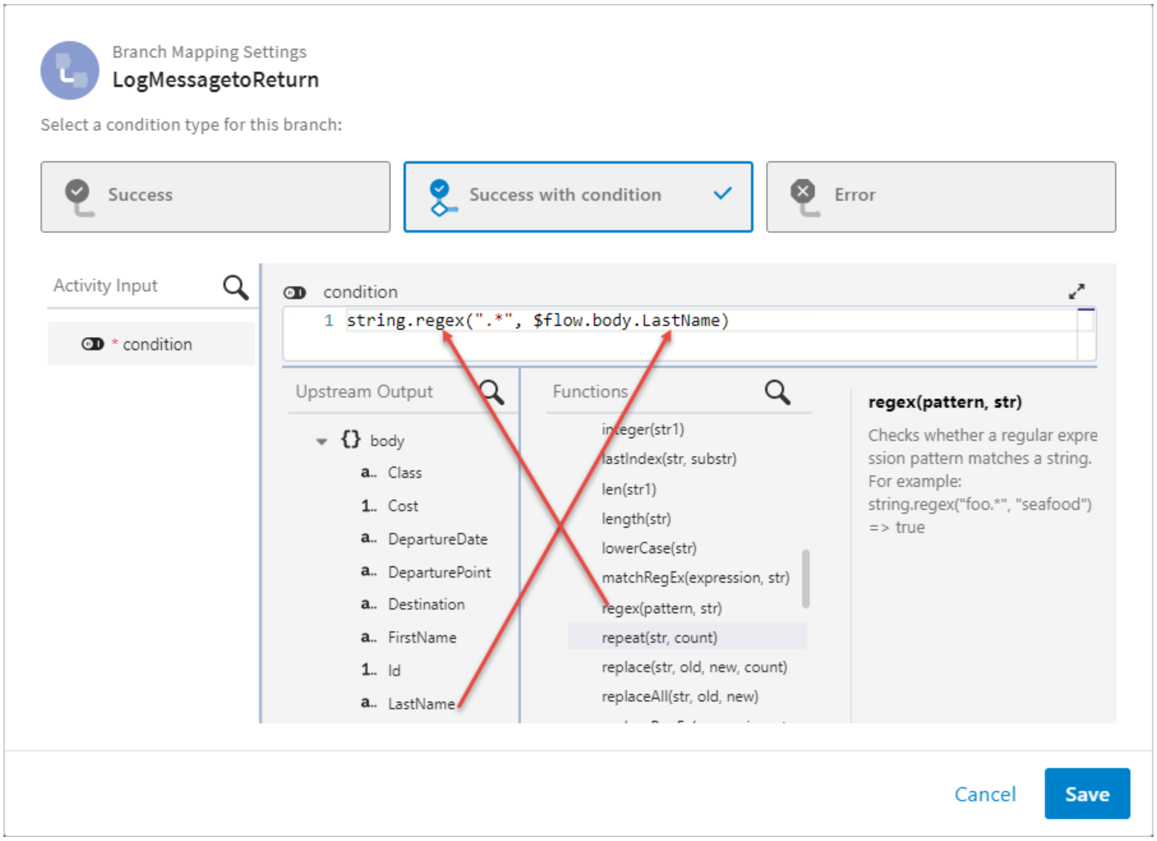
Click the button to expose the following options:



Click (⚙). The **Branch Mapping Settings** dialog opens.

Select the **Success with condition** branch condition.

1. Click **condition** to open the mapper on the right.
2. Configure the branch condition with a regular expression that accepts all last names.
   1. Expand the **string** category under **Functions** and click **regex(pattern, str)** to add this function to the **condition** text box.
   2. Replace **pattern** in the expression by manually entering `".*"` (include the quotes too).
   3. Replace **str** in the function with the last name that is mapped from the flow output under **Upstream Output**. To do so, expand **$flow** > **body** and click on **LastName**.

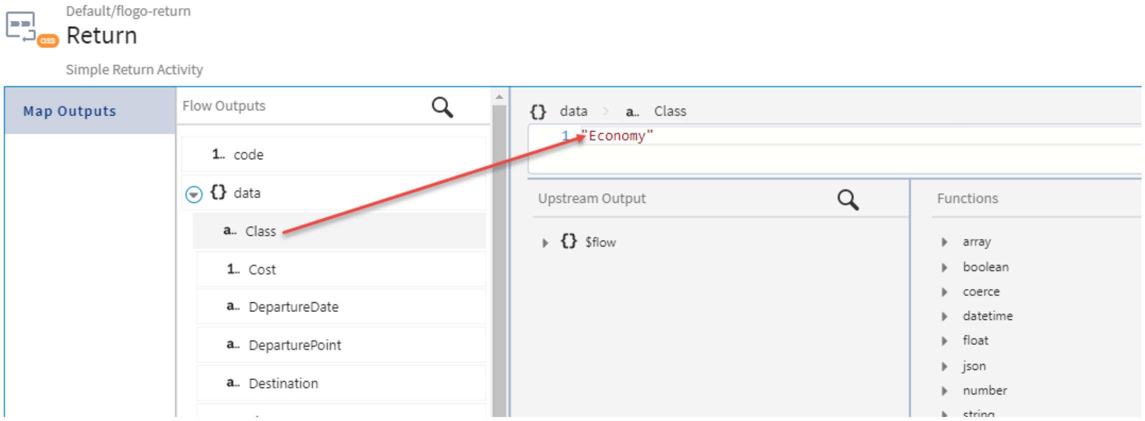


```
string.regex(".*", $flow.body.LastName)
```

4. Click **Save**.

5. Configure the **Return** activity for the branch to output the flow results if this branch executes (when the passenger's last name is anything but Jones). To configure the activity, follow these steps:

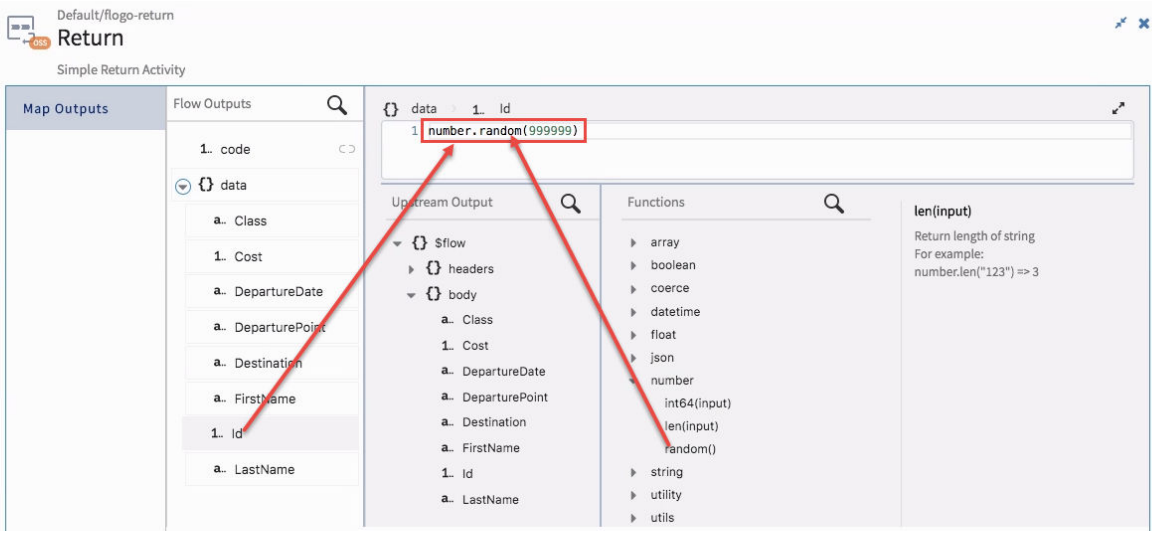
   1. Click on the **Return** activity to open its configuration.
   2. Expand **data** under **Flow Outputs**.
   3. Click **code** to open the mapper and enter `200` in the text box.
   4. Expand **data** under **Flow Outputs**.
   5. Click **Class** and enter `"Economy"`.



```
"Economy"
```

   6. Expand **$flow** > **body** under **Upstream Output**.
   7. One by one, map all remaining elements of the flow outputs under **data** (Cost, DepartureDate, DeparturePoint, Destination, FirstName, and LastName) except **Id**, by first clicking on them under **data** and then clicking on the corresponding element under **body**. Do **not** map **Id**.

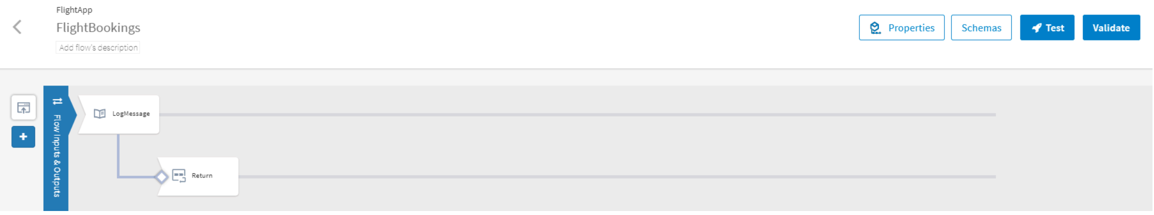
8. Click **Id** under **data** to configure it.
9. Expand the **number** category under **Functions** and click **random()**.
10. Enter `999999` as an input parameter to the `random()` function.



```
number.random(999999)
```

11. Click **Save**.
12. Click **x** to close the dialog.

Your flow should continue to look like this:



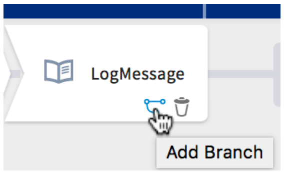
Step 7: Add a Second Branch to Check for Last Name "Jones"

The second branch you add from the **LogMessage** activity checks the **LastName** element in the incoming request to see if it is "Jones". If the passenger's last name is "Jones", the passenger's seat is automatically upgraded to Business Class.

The string for the last name is case sensitive. So, "Jones" is viewed as different from "jones".

To add a second branch from the **LogMessage** activity, follow these steps:

1. Hover your mouse cursor over the **LogMessage** activity and click .



The branch gets added with the **Add Activity** dialog open.

2. Click the **x** on the upper right corner of the **Add Activity** dialog to close it.
3. Add a **Return** activity. To do so,

a. Hover your mouse cursor to the end of the branch and click the (+) icon.

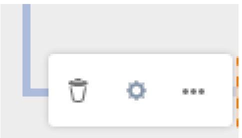
b. Scroll to the **Default** category and click it.

c. Click the **Return** card to add the activity.

d. Click the **x** to close the configuration dialog. You must now configure this branch with a condition to read the last name of the passenger and check to see if it is "Jones".

4. Hover your mouse cursor to the end of the branch until you see a button with three dots placed horizontally.

   Click the button to expose the following options:



   Click (⚙). The **Branch Mapping Settings** dialog opens.

   Select the **Success with condition** branch condition.

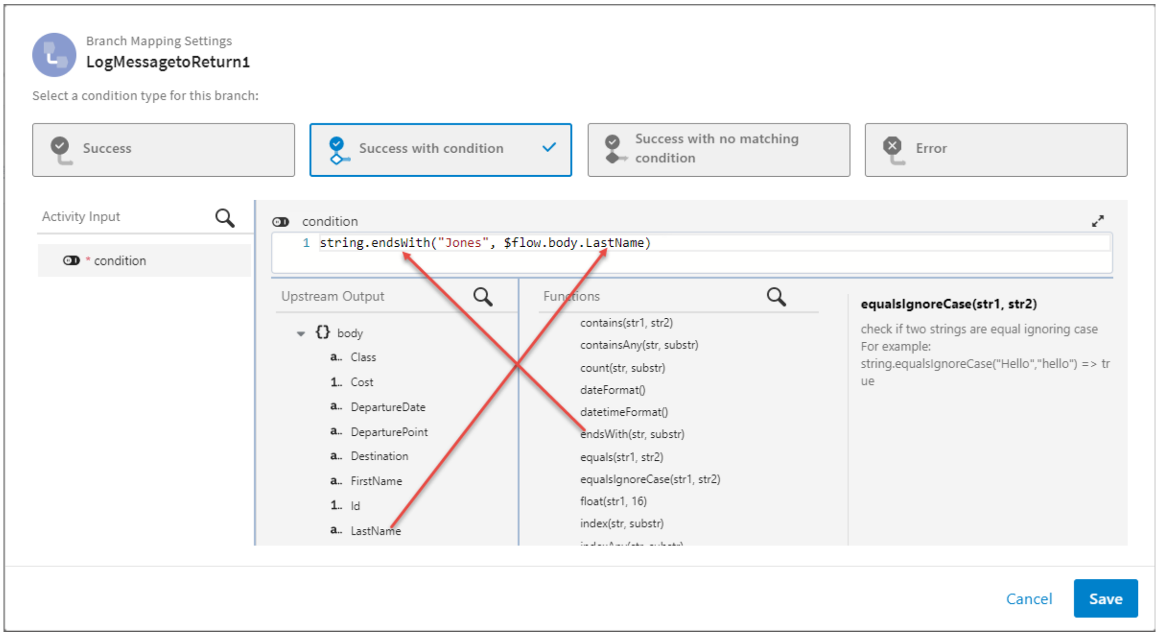
5. Configure the condition for the branch to check if the last name ends with "Jones".

   > The string for the last name is case sensitive. So, "Jones" is considered different from "jones".

   1. Click **condition** to open the mapper.
   2. Expand **$flow** > **body** under **Upstream Output**.
   3. Expand the **string** function group and click **endsWith(str, substr)**.
   4. Replace **str** by manually typing `"Jones"` (include the quotes).

      > The string for the last name is case sensitive. So, "Jones" is viewed as different from "jones".

   5. Select **substr**, then click **LastName** under **body**. This replaces the **str** with the last name extracted from the output of the **ReceiveHTTPMessage** trigger.



```
string.endsWith("Jones", $flow.body.LastName)
```

equalsIgnoreCase(str1, str2)

check if two strings are equal ignoring case
For example:
string.equalsIgnoreCase("Hello","hello") => true

   6. Click **Save**.

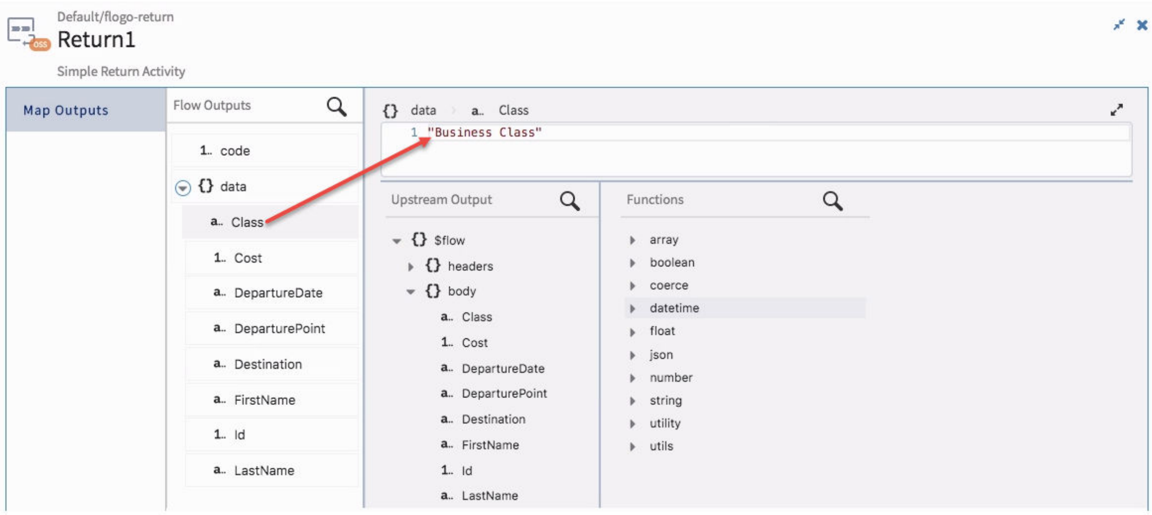
6. Configure the **Return1** activity to send the flow results back to the trigger if the second branch executes (it executes when the passenger's last name is Jones).

   a. Click **Return1** to open its configuration dialog.

   b. Expand **data** under **Flow Outputs**.

   c. Click **code** and enter `200` in its text box.

   d. Click **Class** under **data** and enter `"Business Class"` (include the surrounding double quotes).



```
"Business Class"
```

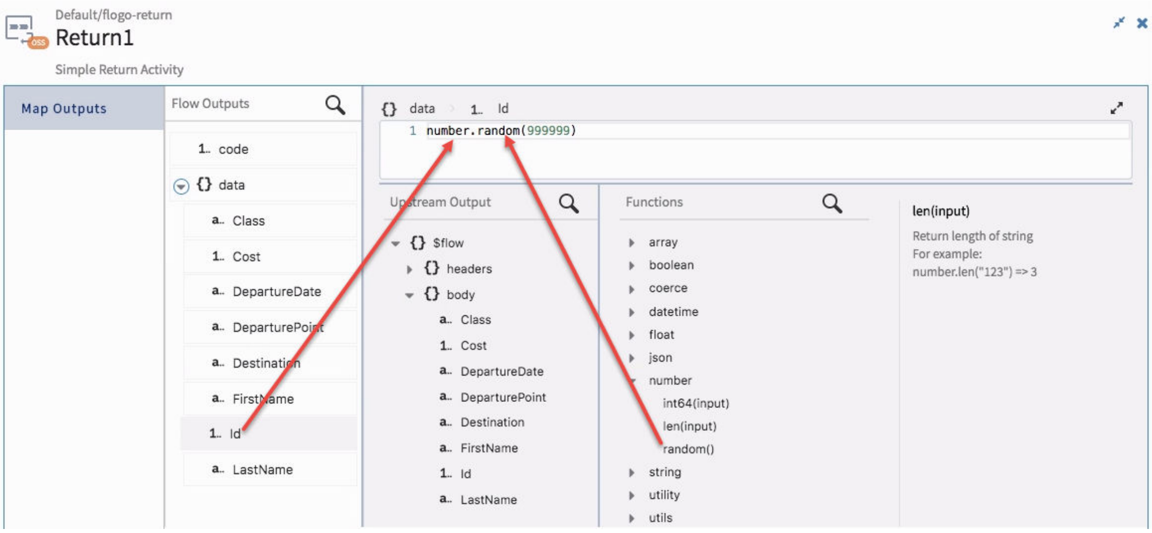
   e. Expand **$flow** > **body** under **Upstream Output**.

   f. Map all remaining elements under **data** (Cost, DepartureDate, DeparturePoint, Destination, FirstName, and LastName) except **Id**, by clicking on them one at a time and then clicking on the corresponding element under **body**. Do **not** map **Id**.

   g. Click **Id** under **data** to configure it.

   h. Expand the **number** function and click **random()**.

   i. Enter `999999` as the input to the random() function.

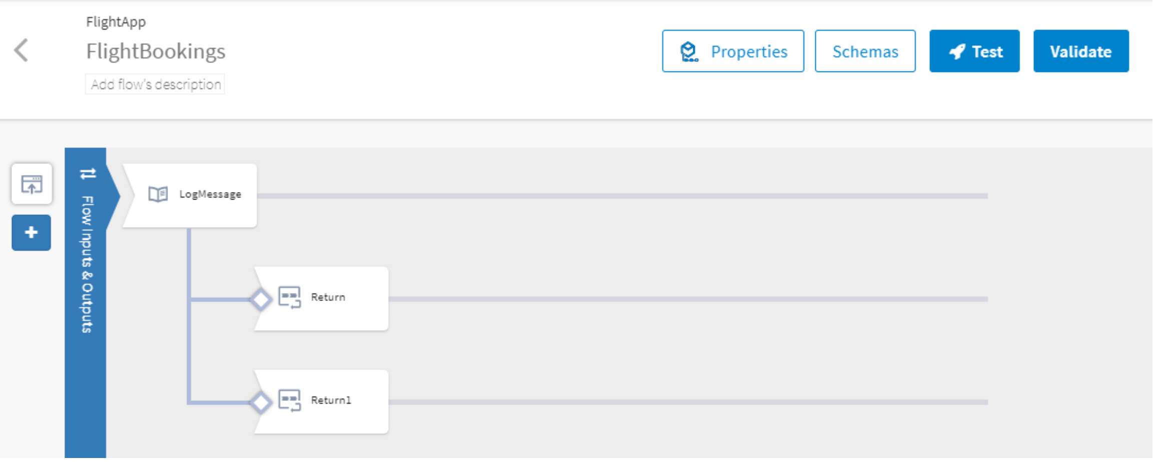


```
number.random(999999)
```

   j. Click **Save**.

k. Click **x** to close the dialog.

Your flow should look like the following:



Step 8: Validate the App

Your app is now ready. Before you push the app to the cloud, be sure to validate all the flows to confirm that there are no errors or warnings. To do so click the **Validate** button. TIBCO Cloud Integration - Flogo (PAYG) validates each flow and activity within the flow. If there are any errors or warnings, you see the respective icons next to the flow name or activity tab which contains the error or warning.

On successful validation, you get the following message:

The validation was completed successfully

App Development

TIBCO Cloud Integration - Flogo (PAYG) offers a wizard driven approach to app development. You can create apps in TIBCO Cloud Integration - Flogo (PAYG) using only a browser. It is powered by Project Flogo™, a lightweight integration engine.

For more information about Project Flogo™, go to http://flogo.io.

Editing the same app in two browser tabs is not supported.

Creating and Managing a Flogo App in the Web UI

This section describes how to create and manage Flogo apps.

Creating a TIBCO Flogo® App

You can create a Flogo® App from the Apps page.

Procedure

1. Open the **Apps** page.
2. Click **Create/Import app**.

   The app details page opens. By default, the app is named in a sequential order in the format `New_Flogo_App_<sequential_number>`. For example, if you created three apps without renaming them, then the first one has a default name of `New_Flogo_App_1`, the second one is called `New_Flogo_App_2` and the third one is called `New_Flogo_App_3`. The version of a newly-created app is 1.0.0 and is displayed as `v: 1.0.0` beside the name of the app. You can edit the version of the app. For more information, refer to Editing the Version of an App.
3. Edit the app name to a meaningful string. To do so, click anywhere within the app name and edit it, then click anywhere outside the text box to persist your change.

   The app name must not contain any spaces. It must start with a letter or underscore and can contain letters, digits, periods, dashes, and underscores.
4. Click **Create**.

   The card for an app type is disabled if the plan you purchased does not entitle you to create certain types of apps such as BusinessWorks app, Flogo app, Mock app, or Node.js app, or use certain capabilities within an app type.

   You can now create one or more flows for the app. See the Creating a Flow topic and its sub-topics for details on creating a flow.

Creating an App from a Saved Specification

If you have an existing specification saved in either the TIBCO Cloud™ Integration - API Modeler or on your local machine, you can use the specification to create a Flogo App. Currently, TIBCO Cloud Integration - Flogo (PAYG) supports app creation using a Swagger Specification 2.0, OpenAPI Specification 3.0, GraphQL Schema, or gRPC Protobuf.

The specification must exist prior to creating the Flogo App.

Refer to the appropriate topics under the Building APIs section for information on how to create a Flogo App using the specification:

Using an OpenAPI Specification

Using GraphQL Schema

Using gRPC

Validating your App

After you have created the flows in your app, you must validate the app before you push it to the cloud. To validate your app, click the **Validate** button on the app details page. This validates each flow and activity. If a flow or activity has an error, it displays an error or warning icon on the top right corner of the flow or activity.

TIBCO Cloud Integration - Flogo (PAYG) does not retain the results of the previous validation if you navigate into a flow after you have validated the app. For example, if two of your flows have errors, and you navigate into one of those flows to fix the error, when you get back to the app detail page, the results of the second flow validation are lost. If you navigate into a flow after you have validated the app, you must validate the app again irrespective of whether you have made changes to the flow or not.

Refer to Viewing Errors and Warnings section for more details.

Editing a TIBCO Flogo App

You can edit your Flogo® App from the Apps page. You can edit flows, triggers, and so on.

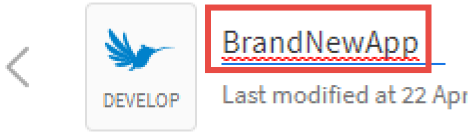
Editing the same app in two browser tabs is not supported.

Renaming an App

You can rename an existing app.
To rename an existing app, do the following:

Procedure

1. Open the app details page by clicking the app name.
2. Click anywhere in the app name and edit the name.



3. Click away from the app name to persist your changes.

Editing the Version of an App

When you create an app, the default version of the app is 1.0.0. You can edit the version of an app.

The format of a valid app version is:

```
xxx.xxx.xxx
```

Alphabets or special characters are not allowed in an app version.

Some examples of valid app versions are:

```
1.1.1
11.22.13
111.222.333
```

Procedure

1. Open the app details page.
   Beside the name of the app, the version of the app is displayed as follows:

   ```
   New_Flogo_App_<sequential_number> v: 1.0.0
   ```

   For a newly-created app, the version is 1.0.0.
2. To edit the version of the app, click on the version number and specify the new version.
   The new version of the app is reflected everywhere. For example, in runtime logs.

Reverting Changes to an App

After editing an existing app, as long as you have not the app, you can revert the app to the state that it was in after the latest . This reverts the changes you just made.
To revert your edits to an existing app before it:

Procedure

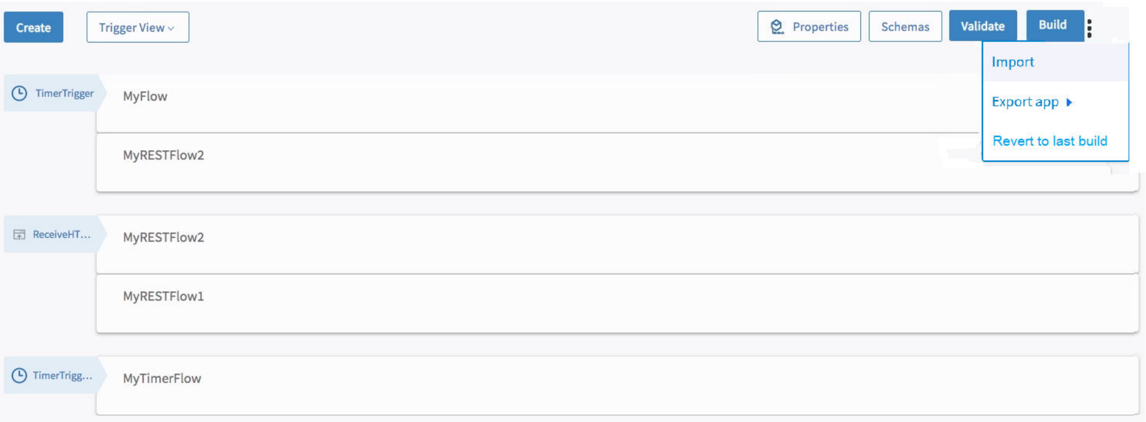
1. On the app details page, click the (⋮) icon to reveal the options.
2. Click . The is enabled only if you have made edits to the app after the last , but have not the app with those edits.

Switching Between Display Views on the App Page

When you click an app name on the **Apps** page, the app details page opens. The flows in the app are listed on the app details page. You have the option to view this page in the **Trigger View** or **Flow View**. By default, it opens in the **Trigger View**. Click **Trigger View** and select **Flow View** from the drop-down menu to switch to the flow view. When you are in the flow view, click **Flow View** and select **Trigger View** from the drop-down menu to go back to the trigger view.
Trigger View

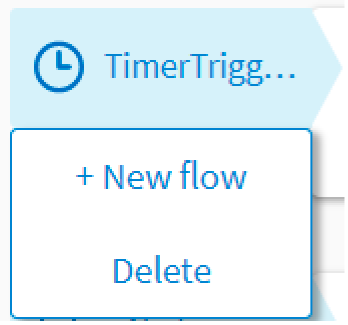
In this view, the flows are displayed attached to the trigger(s) that they use. If a flow is attached to multiple triggers, it is attached to each trigger separately. So, you can see it multiple times on the page but attached to different triggers. Flows that are not attached to any trigger display **No trigger** in place of the trigger name.

Trigger View

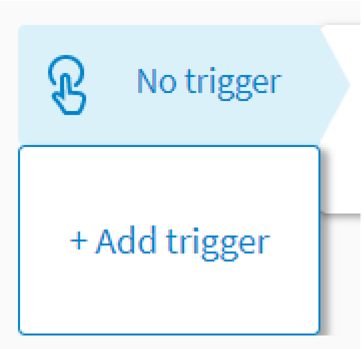


In the image above, **MyRESTFlow2** is attached to both **TimerTrigger** and **ReceiveHTTPMessage** trigger as shown, hence it appears twice. The **MyTimerTrigger** flow was created with a new Timer trigger, hence it is not attached to the top Timer trigger which has two flows attached to it and **TimerTrigger** appears twice on the page.

Hovering on a trigger displays the **New flow** option. Click the **New flow** option to create a new flow to attach the newly created flow to that trigger.



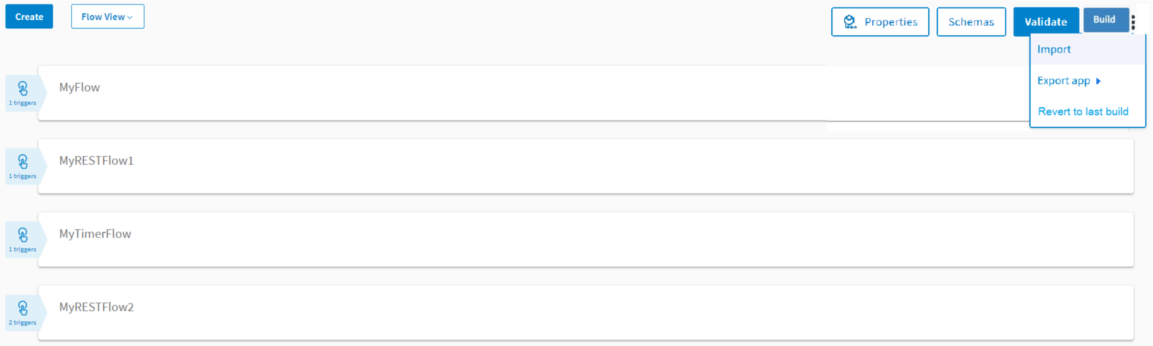
Hovering over **No trigger** displays the **Add trigger** option which takes you to the triggers catalog.



Flow View

In this view, each flow is shown separately and the trigger that it is attached to is shown on the extreme left of the flow. Shown below is a **Flow View** representation of the **Trigger View** image above:

Flow View



Notice that **MyRESTFlow2** shows two triggers. That is because this flow is attached to two triggers as you can see in the **Trigger View**. A blank flow shows 0 triggers against it as it is not attached to any triggers.

Deleting an App

You can delete an app using the **Delete app** icon which appears when you hover your mouse cursor to the end of the app row.

To delete an app:

Procedure

1. On the **Apps** page, hover your mouse cursor to the end of the app row until the **Delete app** icon (🗑) appears.
2. Click the **Delete app** icon.
3. On the confirmation dialog box, click **Delete app**.

Result

The selected app is deleted.

Exporting and Importing an App

You can export and import apps and use them as templates to quick start development, or simply put them in a version control system such as GitHub.

Exporting an App

Here are a few things to keep in mind before you export an app:

- When you export an app, all flows in your app get exported. You cannot pick and choose flows to export.
- Passwords configured in any activity within any flow or connection in the app to be exported are removed in the exported app. You must manually configure the credentials in the flows after importing such apps.
- Some apps created in Project Flogo™ use the `any` data type. The `any` data type is not supported in TIBCO Cloud Integration - Flogo (PAYG). Such apps get imported successfully, but the element of type `any` gets converted into an empty object. You must explicitly use the mapper to populate the empty object with member elements.

To export an app, follow these steps:

Procedure

1. On the **Apps** page, click the app to open the app details page.
2. Click the hamburger menu (⋮).
3. Click **Export**.

Exporting an App's JSON File

When an App's binary is built, the `.json` file is embedded within the binary file. To export the `.json` file from the binary file to the disk, use the following command.

```
./<app-binary-name> --export app
```

The `.json` is exported as `<app-binary-name>.json`.

> To provide a different file name to the exported `.json`, use the following command:
>
> ```
> ./<app-binary-name> --export -o <new-app-binary-name>.json app
> ```

Importing an App

You can import the triggers and flows from an exported app into an existing app or into a new app. The target app into which you want to import must exist. You can import an app by dragging and dropping its `.json` file into the Web UI.

Flogo apps that are exported from TIBCO Cloud Integration - Flogo (PAYG) 2.5.0 and later cannot be imported into previous versions of TIBCO Cloud Integration - Flogo (PAYG).

Here are a few things to keep in mind before you import an app:

- If any flow in the app uses extensions developed by the community, those extensions must be available to the target app into which you are importing, at the time of the import. Flows that make use of extensions that are not available to the target app are not imported.
- Passwords configured in any activity within any flow or connection in the app to be exported are removed in the exported app. You must manually configure the credentials in the flows after importing such apps.
- Some apps created in Project Flogo™ use the `any` data type. The `any` data type is not supported in TIBCO Cloud Integration - Flogo (PAYG). Such apps get imported successfully, but the element of type `any` gets converted into an empty object. You must explicitly use the mapper to populate the empty object with member elements.
- When importing an app, be aware that the `long` and `double` data types get converted to the `number` data type.

The suffixes used in the Mapper have undergone some changes, because of which you may receive a mapper-related warning in the **Import app** dialog when importing an existing app. Click **Continue** and the app imports successfully. After the import completes, be sure to re-map the properties in the activities that show errors. This ensures that they switch to the new suffix format. The following table shows you the changes in the suffixes:

Original suffix appearing in imported apps	New suffix used by the Mapper (after you re-map)	For example...	Used when mapping...

activity_id.activity_parameter	$activity[*activity_id*].*activity_parameter*	Old suffix:
		`$InvokeRESTService.responseBody.userId`
		New suffix after re-mapping property:
		`$activity[InvokeRESTService].responseBody.userId`
			When mapping to a parameter in the activity's output. Used to resolve activity params.

$TriggerData	$trigger	Old suffix:
		`$TriggerData.queryParams.title`
		New suffix after re-mapping property:
		`$trigger.queryParams.title`
			When mapping from the output of the trigger to flow input

N/A There was no equivalent for this in the old mapper	$flow.headers.*parameter* $flow.body.*parameter*	`$flow` is a newly introduced suffix which did not have an equivalent suffix in the old mapper.	When mapping to any parameter in the flow's header or input schema (schema entered in the **Input** tab of **Flow Inputs & Outputs** dialog) which is the same as the output of the trigger, since the output of the trigger is mapped to the input of the flow. Used to resolve parameters from within the current flow. If a flow has a single trigger and no input parameters defined, then the output of the trigger is made available via `$flow`.

- In imported apps, the passwords and secrets for any connections configured in the app do not get imported. You must reconfigure any password or secret for the connection after the app has been imported.
- When you import an app which does not have a **Return** activity in any flow (main or branched flow), the **Return** activity is not added automatically by default. However, if an existing app already has **Return** activities in main or branched flows, the app is imported as expected.

Importing into a new or empty app

1. Create a new app if you do not already have one. See Creating a Flogo App for details on creating an empty app.
2. On the app details page, select **Import app**.
3. Navigate to or drag and drop the `.json` file for the app that you want to import.
4. Click **Upload**. The **Import app** dialog displays some generic errors and warnings as well as any specific errors or warnings pertaining to the app you are importing. It validates whether all the activities and triggers used in the app are available in the **Extensions** tab.
5. You have the option to import all flows from the source app or selectively import flows.

    - **Import all** - To import all the triggers and flows in the app, select **Import All**. If any activities or triggers are missing in the **Extensions** tab, the import process ignores the flows that contain those activities resulting in those flows not being imported. If the existing app already has activities or triggers with the same name as the ones you are importing, you see a warning that they will be overwritten. If you do not want to overwrite the flows, you can click **Back** and clear the selection then click **Next**. If you do so, the duplicate flows that you de-selected will not get imported. You have the option to rename the flows in the Web UI and export the app and re-import it.
    - **Selective import** - Select **Selective Import**, to import only specific triggers and flows from the app. The **Import app** dialog displays a list of triggers with a check box next to each one. If any activities or triggers are missing in the **Extensions** tab, the activities or triggers missing in the **Extensions** tab are not listed in the **Import app** dialog, hence you will not be able to select them to import. By default, all check boxes are selected. Clear the check box next to the triggers that you do not want to import. All flows associated with the selected trigger(s) get imported by default. If you do not select a trigger, the flows and their subflows associated with the unselected trigger(s) are listed in the next screen.
6. Click **Next**.

    If you had not selected a trigger in the previous dialog, the flows associated with that trigger are displayed. You have the option to select one or more of these flows such that the flows get imported as blank flows that are not attached to any trigger. By default, all flows are selected. Clear the check box for the flows that you do not want to import. If your flows have subflows, and you select only the main flow but do not select the subflow, the main flow gets imported without the subflow. Click **Next**.

Importing into an app that has existing flows

When importing the app into another app that has existing flows, keep the following in mind:

- If the existing app already has flows, activities or triggers with the same name as the ones you are importing, a warning is displayed. You can opt not to import those flows, activities, or triggers. You can go back and rename them in the Web UI and export the app again and re-import it.
- If any of the flows that were imported with the app had credentials such as a password or a connection, be sure to re-configure them.

1. Open the app details page by clicking the app name.

2. Click the hamburger menu (⋮) and select **Import**.
3. Navigate to or drag and drop the `.json` file for the app that you want to import.
4. Click **Upload**. The **Import app** dialog displays some generic errors and warnings as well as any specific errors or warnings pertaining to the app you are importing. It validates whether all the activities and triggers used in the app are available in the **Extensions** tab.
5. You have the option to import all flows from the source app or selectively import flows.

   - **Import all** - To import all the triggers and flows in the app, select **Import All**. If any activities or triggers are missing in the **Extensions** tab, the import process ignores the flows that contain those activities resulting in those flows not being imported. If the existing app already has activities or triggers with the same name as the ones you are importing, you see a warning that they will be overwritten. If you do not want to overwrite the flows, you can click **Back** and clear the selection then click **Next**. If you do so, the duplicate flows that you de-selected will not get imported. You have the option to rename the flows in the Web UI and export the app and re-import it.
   - **Selective import** - Select **Selective Import**, to import only specific triggers and flows from the app. The **Import app** dialog displays a list of triggers with a check box next to each one. If any activities or triggers are missing in the **Extensions** tab, the activities or triggers missing in the **Extensions** tab are not listed in the **Import app** dialog, hence you will not be able to select them to import. By default, all check boxes are selected. Clear the check box next to the triggers that you do not want to import. All flows associated with the selected trigger(s) get imported by default. If you do not select a trigger, the flows and their subflows associated with the unselected trigger(s) are listed in the next screen.
6. Click **Next**.

   If you had not selected a trigger in the previous dialog, the flows associated with that trigger are displayed. You have the option to select one or more of these flows such that the flows get imported as blank flows that are not attached to any trigger. By default, all flows are selected. Clear the check box for the flows that you do not want to import. If your flows have subflows, and you select only the main flow but do not select the subflow, the main flow gets imported without the subflow.
7. Click **Next**.

Importing flows without importing the triggers that they are attached to

1. Select **Selective Import** when importing the app.
2. Clear the check box for the triggers that you do not want to import.
3. Click **Next**. A list of flows is displayed.
4. Select the flows that you would like to import and click **Next**. The flows are imported as blank flows without being attached to a trigger.

Handling connections when importing an app

Each connection in TIBCO Cloud Integration - Flogo (PAYG) contains a unique internal ID. The IDs are not exposed in the Web UI and are unique based on the user who created them.

When TIBCO Cloud Integration - Flogo (PAYG) compares connections, it does so by comparing their internal IDs. It considers two connections identical if they have the same connection type and same connection ID. It considers two connections as similar if they have the same connection type, but different connection ID.

Hence, if the app you are importing was not created by you, then any connections used in that app can not have the same ID as any existing connection of the same type that you might already have in your installation of TIBCO Cloud Integration - Flogo (PAYG). For example, if you import an app created by some other user that has some Salesforce connections, even though your installation of TIBCO Cloud Integration - Flogo (PAYG) might already have some existing Salesforce connections, the connections are considered similar, because they are of the same type (Salesforce) but not identical because they do not have the same ID because they were not created by the same user.

When importing an app containing a connection, if your target app has an existing connection with an identical internal ID as the connection in the app being imported, a new connection does **not** get created. The imported app uses the existing connection in such a case. The connection credentials do not get exported with the app. If a new connection gets created, you must re-configure the connection credentials after the app has been imported.

Keep the following in mind when you import an app with connections:

Import all

If you had selected **Import all** when importing the app, you have the following options:

- If you are the owner who created the app to be imported, if identical connections exist in your environment, the existing connections are automatically re-used.
- If identical connections do not exist, then new connections get created without passwords. You must set the password for such connections after the app has been imported.
- If there are similar connections (same type but different IDs) in the host app, TIBCO Cloud Integration - Flogo (PAYG) does *not* re-use those connections. It creates new connections without passwords. You must set the password for such connections after the app has been imported.

Selective import

If you chose to do a selective import when importing an app, the **Import app** dialog lists the connections that are used in the flows and triggers that you selected for import in the app to be imported. It displays a drop-down menu next to each connection. You have the following options:

- If you have any existing identical connections (same connection type and same connection ID) in the host app, that connection is automatically selected in the drop-down menu next to the connection. You have the option to re-use the existing identical connection by leaving it pre-selected.
- If there are any similar connections in the host app (same connection type but different connection ID), you can select the similar connection from the drop-down menu next to it.
- You always have the option to select **Create new connection** from the drop-down menus for any of the connections. TIBCO Cloud Integration - Flogo (PAYG) creates new connections with no passwords. You must manually create a password for the new connection after importing the app.

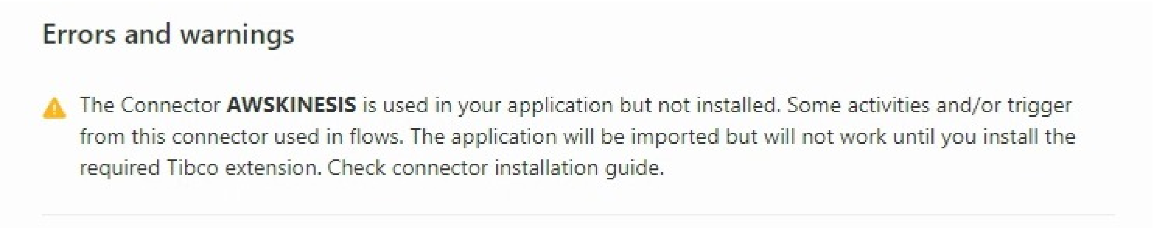
Resolving Missing Activities and Triggers

When you import an app that contains one or more activities or triggers that are not installed in your environment, you see a warning in the **Import App** dialog.

> When importing an app that has a connection configured in it, but the connector is not installed in your environment, after you install the connector, the connection configuration field values of type SECRETS are retained post installation as long as they were not configured using application properties. If you had configured your SECRETS as application properties, you will need to reconfigure them after installing the missing connector. This is because all application properties in the app are wiped out when the app is imported.

To resolve missing activities or triggers for which TIBCO provides connectors

When an activity or trigger used in an app being imported is missing from your TIBCO Cloud Integration - Flogo (PAYG) environment, the flows in the app get imported, but you see a warning in the **Import App** dialog.



Errors and warnings

⚠ The Connector **AWSKINESIS** is used in your application but not installed. Some activities and/or trigger from this connector used in flows. The application will be imported but will not work until you install the required Tibco extension. Check connector installation guide.

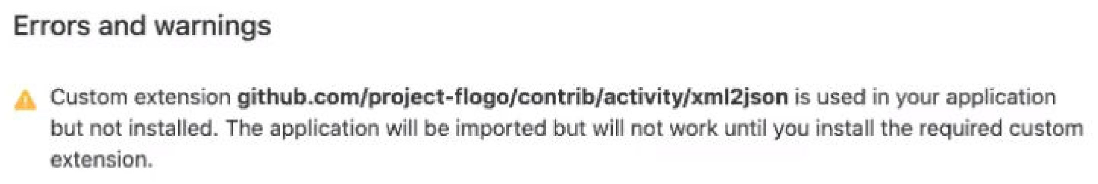
When you validate your app by clicking on the **Validate** button in the app details dialog, you see an error marker ( ) next to the flow name. This indicates that one or more activities or triggers are missing. The number next to it indicates how many activities or triggers that are missing appear in the flow. When you click on the missing activities or triggers, you are prompted to refer to connector installation guide.

Do not upload a TIBCO connector using **Upload Extension**. For more information on how to install a TIBCO connector, refer to connector installation guide.

This is also true when you copy an app into the designated folder (the folder you specified when you started your Web UI) for your apps on your local machine.

To resolve your custom activities or triggers that are missing

When one or more of your custom activities or triggers used in the app being imported are missing from your TIBCO Cloud Integration - Flogo (PAYG) installation, you see a warning in your **Import App** dialog similar to the following:



Errors and warnings

⚠ Custom extension **github.com/project-flogo/contrib/activity/xml2json** is used in your application but not installed. The application will be imported but will not work until you install the required custom extension.

Once the app is imported, you see an error marker ( ) next to the flow name. After you install the missing activity of trigger, this marker goes away. The number next to the error indicates how many activities or triggers are missing in the flow.

To install the missing custom activities or triggers, do the following:

1. Click the flow name to open the flow details page. The **Upload an extension** dialog opens. You upload custom activity or trigger from the Git repository, hence only the **From Git repository** option is enabled.
2. Click **From Git repository**. The **Git repository URL** text box is pre-populated.
3. Click **Import**. TIBCO Cloud Integration - Flogo (PAYG) downloads the activity or trigger from the Git repository and uploads it into your **Extensions** tab. Refer to the section, Uploading Extensions for details on this option.

App File Persistence

Your Flogo app files get persisted to the directory that you specify on your local machine. You can use an external source control system such as Git or SVN to store your apps. You can then check in and check out your apps locally from the remote repository. This makes it possible for you to implement the Continuous Integration/Continuous Deployment (CI/CD) pipeline by leveraging any tool available in the market to integrate your app development with the app deployment.

When you start the Flogo Web UI, you are prompted to point to the directory where you have checked out your apps. If you do not provide any path, the apps are stored in the default directory which is: `<FLOGO_HOME>/data/localstack/apps`.

If you restart your Web UI, at the Web UI restart, if you want to continue using the same directory that you had specified, click Enter on your keyboard when it prompts you to set the path. It stores your path preference that you set the last time.

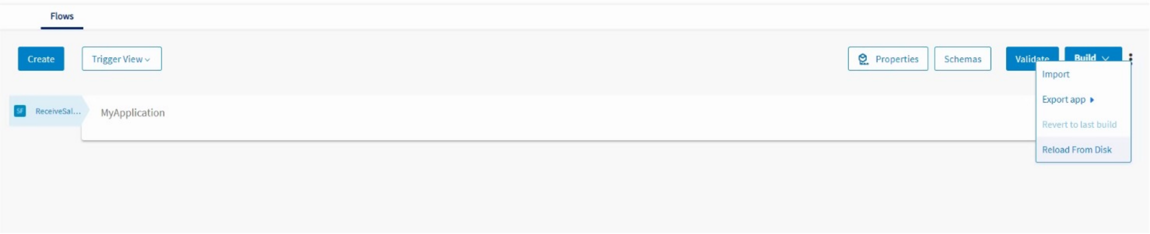
After the UI starts, you should be able to see all your apps on the app list page in the UI. From this point on, when you create a new app or make a change to an existing app, the changes are saved to the directory location that you provided when starting TIBCO Cloud Integration - Flogo (PAYG).

Each app that you store on your local machine has its own folder and the folder name must be identical to the app name. If another user makes changes to your app, you must sync your local repository with the remote repository (do a pull) in order to get the changes made by that user.

- The app file name must be called `flogo.json`.
- The folder name containing the app must be identical to the app name appearing in the `flogo.json` file for the app.

Loading new apps from the disk - When a new app is added to the directory, refreshing the browser loads the app into the Web UI. You do not need to restart the Web UI.

Loading updated app from the disk - In case the `flogo.json` on the disk is updated (due to minor changes or checkout newer version from the source control system), click the **Reload from Disk** to load updated app into the Web UI. Be aware that this action overrides existing changes in the app. **Reload from Disk** option is available under the hamburger menu that is next to the other buttons on the app page.



If another user adds a new app to your remote repository, the app gets downloaded to your local repository when you do a pull from the remote repository. For the new app to display in your TIBCO Cloud Integration - Flogo (PAYG) Web UI, you must refresh your browser. You do not need to restart either the browser or TIBCO Cloud Integration - Flogo (PAYG).

You can import any exported app to TIBCO Cloud Integration - Flogo (PAYG). To do so, create a folder with an identical name as the app name in your local repository, then copy the `flogo.json` file for the app to the folder. For apps that are created in the Web UI, TIBCO Cloud Integration - Flogo (PAYG) automatically generates a unique ID for each app. But, if you load an existing `flogo.json` file, the app may or may not have an app ID defined in it. TIBCO Cloud Integration - Flogo (PAYG) checks to see if an ID exists in the `flogo.json` file for the app. If an ID does not exist for the app, TIBCO Cloud Integration - Flogo (PAYG) generates a unique ID and adds an ID attribute in the `flogo.json` file before loading the app.

Note the following:

- If you change the ID of the app in your flogo.json file, you see a duplicate app in the Web UI. Refresh your browser to fix this issue. If you continue to work on the app with old app ID, your changes are lost when you restart the Web UI.
- All apps that exist in the path that you provided during TIBCO Cloud Integration - Flogo (PAYG) installation get loaded in the Web UI. You cannot selectively choose the apps to be loaded in the Web UI.
- Any Launch Configurations (containing your test data for the app) associated with the app are stored in the ***<app_folder>*** > **test** folder along with the `flogo.json` file for the app.
- File permissions - You must have "write" permission for the app directory on your local machine. Otherwise, the app is not loaded and displayed in the UI. An error is displayed in the log located in `<`*`FLOGO_HOME`*`>/<`*`FLOGO_VERSION`*`>/logs/studio.logs`.

- When importing an app, if any extensions are missing, a broken plug-in icon is displayed on the missing activity.
- If the app has any missing extension or if a connector uses the associated connection, you see the connection post installation of the missing extension or connector.
- If you add an app to your local application repository, if that app has any missing extension, after uploading the missing extension, the connection in the extension maintains the secrets and passwords that were already configured in the connection for the app. Refer to Resolving Missing Extensions section for details on how to resolve missing extensions in an app.
- You may notice change in secret encrypted values in `flogo.json` after opening the apps in Web UI. This does not affect the run time.
- We recommend that you do not modify `flogo.json` manually to avoid any mishaps.
- When upgrading to TIBCO Cloud Integration - Flogo (PAYG) the current version from an older TIBCO Cloud Integration - Flogo (PAYG) version, the existing apps automatically get migrated to the directory that you have created on your local disk. You do not need to migrate them manually.
- If your application repository gets deleted while in use, you must restart the Flogo Web UI and set a new application repository. Do not continue to work with the deleted repository. Also keep in mind that even if you recreate a directory with the same name, your changes do not take effect until you restart the Web UI.

Creating Flows and Triggers

An app can have one or more flows and a flow can be attached to one or more triggers.

Flows

Each flow represents specific business logic in an app. A flow contains one or more activities. The flow execution is started by a trigger. A new flow can be created only from the app details page.

Triggers

You have the option to create a trigger without creating a flow. You can create a trigger from an existing specification that you have saved in either the TIBCO Cloud™ Integration - API Modeler or on your local machine. Optionally, you can create a trigger when creating a flow by selecting the **Start with a trigger** option during flow creation. A trigger can have multiple flows attached to it.

Flows

This section contains information on creating and managing flows in your app.

Creating a Flow

Every app has at least one flow. Each flow can be attached to one or more triggers. You have the option to begin by creating a blank flow (flow without a trigger) too and attaching the flow to one or more triggers at a later time. Use the **Create** link on the app details page to create the first flow in an app. If there are existing flows in an app, click the **Create** button to create additional flows.

Prerequisites

Before creating a flow that uses connectors, make sure that you have created the necessary connections.

If an app has multiple triggers that require a port to be specified, for example, REST and/or GraphQL triggers, make sure that the port number is unique for each trigger. Two triggers in the same app cannot run on the same port.

For flows that are attached to multiple triggers, you cannot disable a trigger, specify a particular trigger to execute, or specify the order in which the triggers execute. When a flow executes, all triggers get initialized in the order that they appear within the flow.

The output of a trigger provides the input to the flow. Hence, it must be mapped to the flow input. When creating a flow without a trigger, there must be a well-defined contract within the flow which specifies the input to the flow and the output expected after the flow completes execution. You define this contract in the **Flow Inputs & Outputs** dialog. The **Flow Inputs & Outputs** contract works as a bridge between the flow and the trigger, hence every trigger has to be configured to map its output to the **Input** parameters defined in **Flow Inputs & Outputs**. You do this in the **Map to Flow Inputs** tab of the trigger.

Likewise, for triggers such as the **ReceiveHTTPMessage** REST trigger that send back a reply to the caller, the trigger reply must be mapped to the flow outputs (parameters configured in the **Output** tab of the **Flow Inputs & Outputs** accordion tab). You do this mapping in the **Map from Flow Outputs** tab of the trigger.

A **Return** activity is not added by default. Depending on your requirements, you must add and configure the **Return** activity manually. For example, if any trigger needs to send a response back to a server, its output must be mapped to the output of the **Return** activity in the flow.

The **Map Outputs** tab of the **Return** activity displays the flow output schema that you configured in the **Output** tab of the **Flow Inputs & Outputs** accordion tab. The **Map from Flow Output** tab of the trigger constitutes the trigger reply. This tab also displays the flow output schema that you configured in the **Output** tab of the **Flow Inputs & Outputs** accordion tab.

You must do the following when using a **ReceiveHTTPMessage** REST trigger:

- Add a **Return** activity at the end of the flow.
- In the **Map Outputs** tab of the **Return** activity, map the elements in the schema to the data coming from the upstream activities.
- In the **Map from Flow Output** tab of the trigger, map the trigger reply elements to the flow output elements.

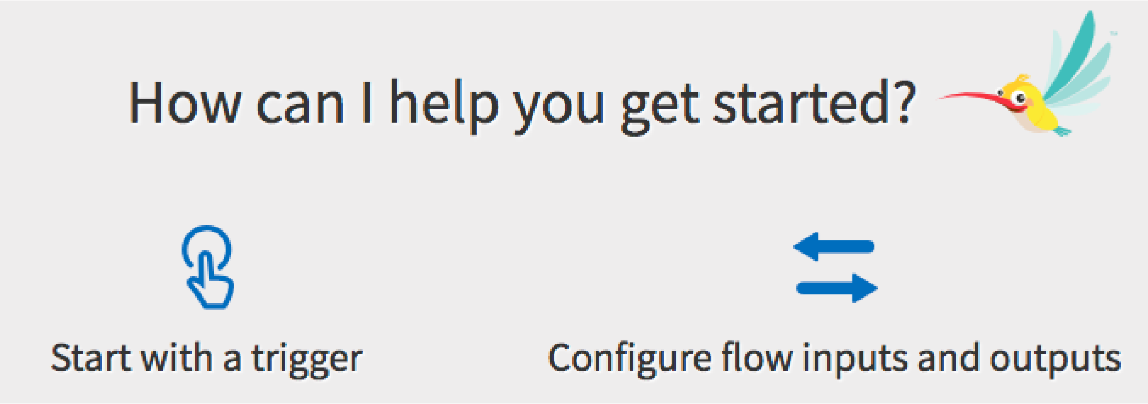
Follow these steps to create a flow:

Procedure

1. Click an app name on the **Apps** page in TIBCO Cloud Integration - Flogo (PAYG) to open its page.
2. Click the **Create** link if this is the first flow in the app. If one or more flows exist, click the **Create** button. The **Add triggers and flows** dialog opens.
3. Enter a name for the flow in the **Name** text box.

   Flow names within an app must be unique. An app cannot contain two flows with the same name.
4. Optionally, enter a brief description of what the flow does in the **Description** text box. The **Flow** option is selected by default.

   To create a flow from a specification, select the specification under **Start with** and refer to the appropriate section under Building APIs.
5. Click **Create**.
   You will be prompted to select one of the following options:


How can I help you get started?
Start with a trigger
Configure flow inputs and outputs

- **Start with a trigger** - If you know the trigger with which you want to activate the flow, select this option. If this is the first flow, the Trigger catalog opens. Select a trigger from the catalog. Refer to the appropriate section under Starting with a Trigger for more details on the type of trigger that you want to create. If there are existing flows attached to triggers, you get a dialog that gives you an option to either use an existing trigger or use a new trigger that has not been used in an existing flow within the app.
- **Configure flow inputs and outputs** - Select this option if you know the algorithm for the flow, but do not yet know the circumstances that will cause the flow to execute. It will create a blank flow that is not attached to any trigger. Flow inputs and outputs create a contract between the trigger and the flow. When you create a trigger, you must map the output of the trigger to the input of the flow. This contract serves as a bridge between the trigger and the flow. You have the option to attach your flow to one or more triggers at any later time after the flow has been created.

A flow gets created. If you selected **Start with a trigger**, the flow is attached to the trigger you selected. If you selected **Configure flow inputs and outputs**, a blank flow without a trigger gets created.

Selecting a Trigger When Creating a New Flow

When creating a new flow, you have the option to either select an existing trigger or select one from the triggers catalog.

Trigger configuration fields are categorized into two groups as explained below. A single trigger can be associated with multiple handlers.

- **Trigger Settings** - these settings are common to the trigger across all flows that use that trigger. If and when a flow attached to the trigger changes any **Trigger Settings** field, the change gets propagated to all flows attached to the trigger. A warning message gets displayed saying so and asking you to confirm before the changes are committed.
- **Handler Settings** - these settings are applicable to a specific flow attached to the trigger. Hence, each flow can set its own values for the **Handler Settings** fields in the trigger. To do so, open the flow and click on the trigger to open its configuration dialog. Click the **Settings** tab and edit the fields in the **Handler Settings** section.

Deciding when to create a new trigger when there is an existing trigger of the same type

There may be cases when a specific type of trigger already exists, for example, there might be a REST trigger that already exists. When creating a new REST flow, you will be prompted to select the existing REST trigger or create a new trigger by selecting it from the triggers catalog. If you want a REST trigger with a different trigger setting than the one that already exists, maybe a different port or different security options, you must select the **Create new** option and select the trigger from the ensuing trigger catalog. This will create a new REST trigger and attach your new flow to it.

Starting with a Trigger

When creating a new flow, if you know the circumstances under which you want the flow to activate, select **Start with a trigger** option and select an available trigger that will activate the flow.

> If an app has multiple triggers that require a port to be specified, for example, REST and/or GraphQL triggers, make sure that the port number is unique for each trigger. Two triggers in the same app cannot run on the same port.

If you are unsure of the circumstances under which the flow should be activated, or if you want the flow to be activated under more than one situation, use the **Configure flow inputs and outputs** option and attach the flow to the trigger(s) at a later time as needed. See Creating a Flow without a Trigger for more details on this.

Refer to the following sections for more details:

Creating a Flow Attached to a REST Trigger

Ceating a Flow Attached to a GraphQL Trigger

Creating a Flow Attached to the gRPC Trigger

Creating a Flow Attached to Other Triggers

Creating a Flow Attached to a REST (Receive HTTP Message) Trigger

When creating a flow with a REST trigger, you have the option to either enter the schema in the **Configure trigger** dialog during flow creation, or you can use a Swagger 2.0 or OpenAPI 3.0 specification file that you have saved either in TIBCO Cloud™ Integration - API Modeler or on your local machine.

If you want to use a specification file, refer to the Using an OpenAPI Specification section for details.

You can create a REST flow by entering a JSON schema or dragging and dropping an API specification JSON file. See Using an OpenAPI Specification section for how to use a specification file.

> If you modify the **Reply Settings** tab of a **ReceiveHTTPMessage** trigger, the corresponding **ConfigureHTTPResponse** activities within that flow do not change appropriately. This happens specifically when removing fields from the **Reply Settings** tab. Redo the mappings for the **ConfigureHTTPResponse** activity.

To create a REST flow by entering the schema, do the following:

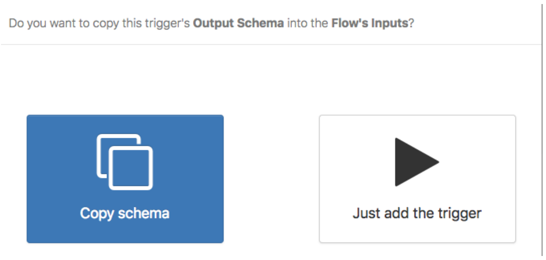
Procedure

1. Click an app name on the **Apps** page in TIBCO Cloud Integration - Flogo (PAYG) to open its page.
2. Click **Create**.
   The **Add triggers and flows** dialog opens.
3. Enter a name for the flow in the **Name** text box.

   Flow names within an app must be unique. An app cannot contain two flows with the same name.
4. Optionally, enter a brief description of what the flow does in the **Description** text box.
5. **New flow** is selected by default. Click **Create**.
6. Select **Start with a trigger**.
   The triggers catalog opens with all the available triggers showing.
7. Click **Receive HTTP Message** card to create a REST trigger.
   The trigger configuration dialog opens.
8. Select the REST operation under **Method** that you want to implement by clicking it.

A flow can have multiple REST triggers. Two REST triggers cannot have an identical port, path, and method combination. Each REST trigger needs to differ from the other REST triggers for the same flow with either a unique port, path, or operation (GET, PUT, POST, DELETE).

9. Enter a resource path in the **Resource Path** text box.
10. Enter the JSON schema or JSON sample data for the operation in the **Enter a JSON Schema or an example of your JSON message** text box. This will be the schema for both input and output.
11. Click **Continue**.
12. Select one of the following dialog:



Do you want to copy this trigger's **Output Schema** into the **Flow's Inputs**?

If you select **Copy Schema**, the schema that you entered in this step above automatically gets copied or displayed in a tree format to the following locations when the trigger gets added:

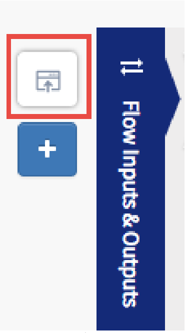
- Trigger output, in the **Map to Flow Inputs** tab of the trigger
- Flow input, in the **Input Settings** tab of the **Flow Inputs & Outputs** accordion tab.
- Trigger reply (If the trigger has a reply), in the **Reply Settings** of the trigger.

Refer to the "REST Trigger" section in the *TIBCO Flogo® Activities and Triggers Guide* for details on configuration parameters.

If you select **Just add the trigger**, a REST trigger gets added to the flow without any configuration. You can configure this REST trigger by clicking on the trigger from the app details page at a later time. Any changes made to the trigger must be explicitly saved by clicking **Save**.

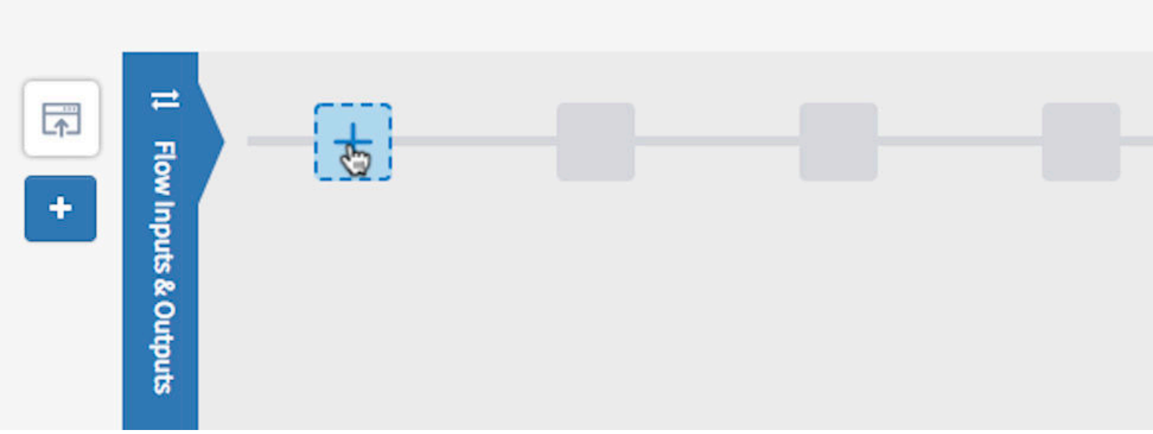
The flow page opens.

13. Map the trigger output to the flow input.
    a) Open the trigger configuration dialog by clicking on the trigger:



    b) Open the **Map to Flow Inputs** tab.
    c) Map the elements under **Flow input** to their corresponding elements under **Trigger Output** one at a time.
14. Map the flow output to the trigger reply as follows:
    a) In the trigger configuration dialog, click the **Map from Flow Outputs** tab.
    b) Map the elements under **Trigger Reply** to their corresponding elements under **Flow Output**.
    c) Close the dialog by clicking on the **x** at its top right corner.

15. Click **Save** to save your changes.
16. Add activities to the flow by hovering your mouse cursor next to the **Flow Inputs & Outputs** tab and clicking the plus sign.



Creating a Flow attached to the GraphQL Trigger

You create GraphQL flows by uploading a GraphQL schema file with an extension `.gql` or `.graphql`. TIBCO Cloud Integration - Flogo (PAYG) creates the appropriate flows based on your schema. When the flow gets created, a GraphQL trigger automatically gets generated and attached to each flow that gets created.
Refer to the section, Using GraphQL Schema on how to create a flow using a GraphQL schema. Also, refer to the "GraphQL Trigger" section in the *TIBCO Flogo® Activities and Triggers Guide* for details on the GraphQL trigger.

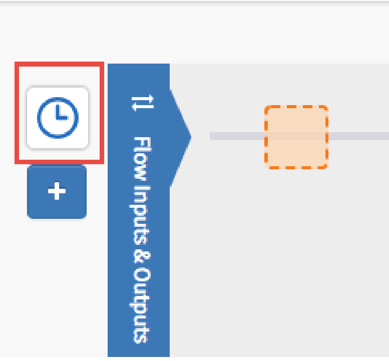
Creating a Flow Attached to Other Triggers

This section applies to triggers that are not REST, gRPC or GraphQL triggers.
To create a flow with such a trigger, follow these steps:

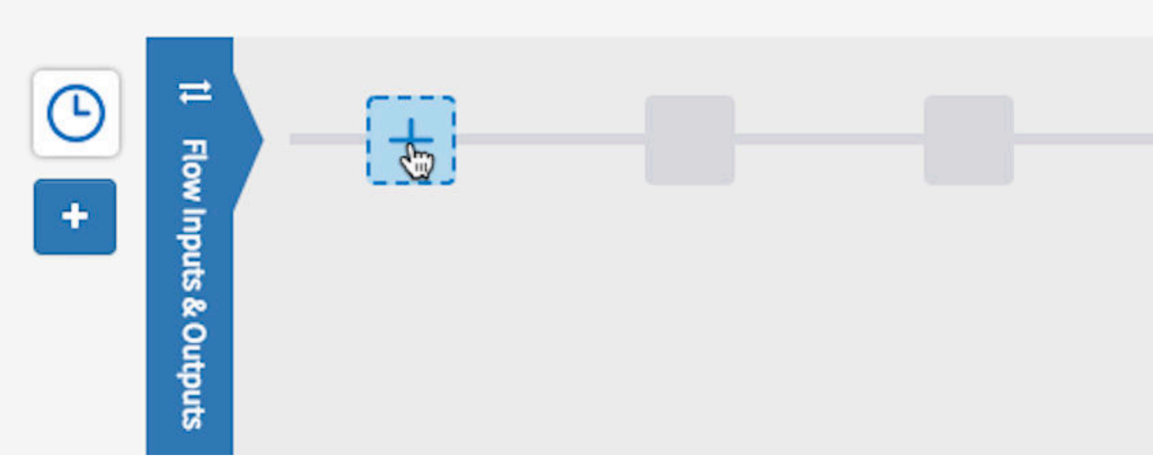
Procedure

1. Click an app name in the **Apps** page in TIBCO Cloud Integration - Flogo (PAYG) to open its page.
2. If this is the first flow in the app, click the **Create** link, or if another flow exists in the app, click the **Create** button.
   The **Add triggers and flows** dialog opens.
3. Enter a name for the flow in the **Name** text box.

   Flow names within an app must be unique. An app cannot contain two flows with the same name.
4. Optionally, enter a brief description of what the flow does in the **Description** text box and click **Create**.
5. Select **Start with a trigger**.
   The triggers catalog opens. If there are existing triggers in the app, they are displayed in the **Select existing trigger** tab.
6. To create a new trigger, click **Add new trigger**.
7. Click the trigger that you want to add.
   The flow details page is displayed with the trigger.
8. Click the trigger to display its properties. For example, the image below shows the Timer trigger.



9. Configure the properties for the trigger. See the respective trigger section in the *TIBCO Flogo® Activities and Triggers Guide* for details.
10. Add activities to the flow by clicking the + icon next to the **Flow Inputs & Outputs** tab:



Creating a Flow Attached to a gRPC Trigger

You create gRPC flows by uploading a gRPC Protobuf file with an extension, `.proto`. TIBCO Cloud Integration - Flogo (PAYG) creates the appropriate flows based on your methods. It implements one flow per method. A gRPC trigger automatically gets generated and the flows are attached to the trigger.

See Using gRPC for information on how to create a flow using a gRPC Protobuf. Also, refer to the "gRPC Trigger" section in the *TIBCO Flogo® Activities and Triggers Guide* for details on the gRPC trigger.

Creating a Blank Flow (Flow without a Trigger)

You can create a flow in the Flogo App without attaching it to a trigger. This method of creating a blank flow is useful when the logic for the flow is available, but you do not know the condition under which the flow should activate. You can start by creating a flow with the logic and attach it to one or more triggers at a later time.

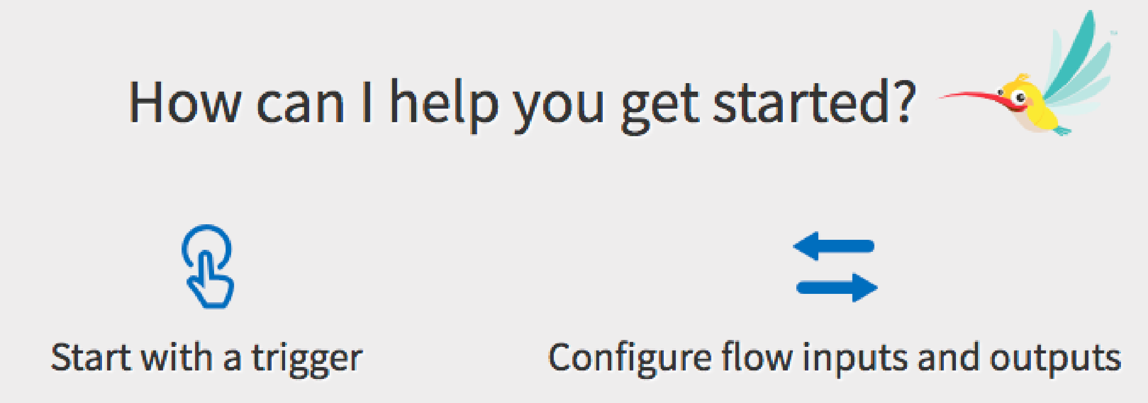
Follow these steps to create a flow without a trigger:

Procedure

1. Click an app name on the **Apps** page in TIBCO Cloud Integration - Flogo (PAYG) to open its page.
2. If this is the first flow in the app, click the **Create** link. If one or more flows exist in the app, click the **Create** button.
   The **Add triggers and flows** dialog opens.

3. Enter a name for the flow in the **Name** text box.

   Flow names within an app must be unique.
4. Optionally, enter a brief description of what the flow does in the **Description** text box. The **Flow** button is selected by default.
5. Click **Create**.
   You will be prompted to select one of the following options:


How can I help you get started?
Start with a trigger
Configure flow inputs and outputs

6. Select **Configure flow inputs and outputs**.
   The flow gets created with the flow details page open displaying the **Input** tab of the **Flow Inputs & Outputs** tab.

   You can configure the inputs and/or outputs to the flow in the **Input** or **Output** tab respectively. See Flow Inputs & Outputs Tab.

   Mapping trigger outputs to flow inputs and flow outputs to trigger reply creates a contract between the trigger and the flow. Hence, when you attach the flow to a trigger later, you must map the output of the trigger to the flow input. You have the option to attach your flow to one or more triggers at a later time after the flow has been created. See Attaching a Flow to One or More Triggers for details.
7. Enter a JSON schema containing the input fields to the flow in the **Input Settings** tab and click **Save**.
8. Enter the JSON schema containing the flow output fields in the **Output Settings** tab and click **Save**.
9. Click the left facing arrow on top of the blue label when done to retract the **Flow Inputs & Outputs** page.
10. Add a **Return** activity (from the **Default** category) to the flow if you want the flow to return some data. Click and drag the **Return** activity to the right to make room to add other activities.
11. Hover your mouse over the shaded square to expose the add activity button (+). Click the add activity button to add an activity.
    After adding an activity, be sure to configure its properties by clicking on the activity tile. If there are any errors in the activity, fix the errors before proceeding. See Errors and Warnings section for more details.
12. Continue adding activities by clicking the successive (+) buttons. To add an activity between two existing activities, you can drag the activities to the right to make room for the new activity, then click the + button to add the new activity.
13. If you added a **Return** activity, click the **Return** activity to configure the parameters that the flow outputs after completing execution. The **Return** activity displays the parameters that you had configured in the **Output Settings** tab of the **Flow Inputs & Outputs** dialog.

Anything that the flow outputs after execution must be mapped into the **Return** activity of the flow. If any trigger needs to send a response back to a server, its output must be mapped to the output of the **Return** activity. This is done in the **Map from Flow Outputs** tab of the trigger.

14. When you are ready to add a trigger, refer to Adding Triggers to a Flow to add one or more triggers to the flow. For triggers that need to send back a response to the server, you must map the flow output (elements in the **Return** activity) to the reply of the trigger (**Map from Flow Outputs** tab in the trigger configuration dialog).

Flow Input & Output Tab

Use these tabs to configure the input to the flow and the flow output. These tabs are particularly useful when you create blank flows that are not attached to any triggers.

> The schemas for input and output to a flow can be entered or modified only in this **Flow Inputs & Outputs** accordion tab. You cannot coerce the flow input or output from outside this accordion tab.

Both these tabs (the **Input** tab and the **Output** tab) have two views:

- **JSON schema view:**

  You can enter either the JSON data or JSON schema in this view. You must click **Save** to save your changes or **Discard** to revert the changes. If you entered JSON data, the data is converted to a JSON schema automatically when you click **Save**.

- **List view:** This view allows you to view the data that you entered in the JSON schema view in a list format. In this view, you can:

  – Enter your data directly by adding parameters one at a time

  – Mark parameters as required by selecting its check box.

  – When creating a parameter, if you select its data type as an array or an object, an ellipsis (...) appears to the right of the data type. Click the ellipsis to provide a schema for the object or array.

  – Use an app-level schema by selecting the **Use an app-level schema** button. On the **Schemas** page that appears, click **Select** beside the schema that you want to use. The name of the schema is displayed beside the **Use an app-level schema** button and the schema is displayed in a read-only mode.

  > You cannot edit an app-level schema in the **List** View if the **Use an app-level schema** button is selected. To edit an app-level schema, follow the instructions in the section Editing an App-level Schema. You can, however, switch to another app-level schema by clicking **Change** and selecting another app-level schema. You can also unbind the app-level schema (by deselecting the **Use an app-level schema** button) from a trigger, activity, or the input and output of a flow. After you unbind the app-level schema, you can make changes to it using the schema editor in the **List** View.

  – Click **Save** to save the changes or **Discard** to discard your changes.

Attaching a Flow to One or More Triggers

If you had created a blank flow without attaching it to a trigger, you can attach it to an existing trigger that is being used by another flow in the same app.
A flow that was created without being attached to a trigger has its input and output parameters defined in the **Flow Inputs & Outputs** accordion tab. You can access it by clicking the blue bar with the same label. The output from the trigger is the input to the flow. So, you must map the input parameters defined in the **Input** tab of this dialog to the trigger output parameters. This mapping must be done in the trigger. The mapping creates a contract between the trigger and the flow and is mandatory for the flow and the trigger to interact with each other.

Procedure

1. You can use one of these methods to attach a flow to a trigger:
   - From the app details page:
     1. Open the app details page by clicking on the app.
     2. Hover over **No trigger**, then click **Add trigger**.
   - From the flow details page:
     1. Open the flow details page by clicking the flow name on the app details page.
     2. Click the **Add a new trigger** icon (+).
2. If there are existing triggers in the app, the **Select existing trigger** tab displays the existing triggers. To use an existing trigger, click **Select existing trigger**, then click the trigger you want to use in the right pane. You also have the option to create a new trigger by selecting the **Add new trigger** tab and selecting a new trigger to create from the **Triggers catalog**. If there are no existing triggers in the app, you only see the **Add new trigger** tab. Select a trigger from the **Triggers catalog**.
3. For REST and GraphQL triggers, you will be prompted to enter additional handler setting details. Refer to the "REST Trigger" section in the *TIBCO Flogo® Activities and Triggers Guide*. Refer to the "GraphQL Trigger" section in the *TIBCO Flogo® Activities and Triggers Guide*.
   The trigger is created and you see its icon to the left of the **Flow Inputs & Outputs** accordion tab.
4. Click the trigger icon to configure the trigger as needed. For REST and GraphQL triggers, be sure to map the trigger outputs to flow inputs and the flow outputs to the trigger reply.
5. Optionally, attach the flow to additional triggers by clicking the + icon and following the steps above.

Catching Errors

You can configure a flow to catch errors at two levels:

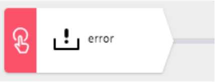
- At the flow level by configuring the Error Handler in the flow. Refer to the section, Creating an Error Handler Flow for more details on configuring the Error Handler in the flow.
- At the activity level by creating an error branch from an activity. Refer to the Types of Branch Conditions subsection under the section, Creating a Flow Execution Branch for details on how to create an error branch from an activity.

Creating An Error Handler Flow

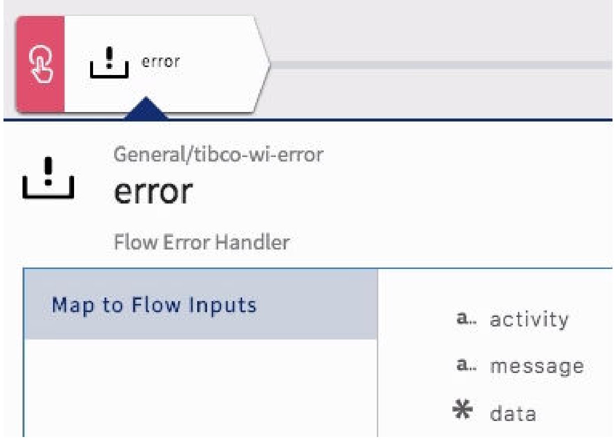
Use the error handler to catch exceptions that occur when executing a flow. The error handler is designed to catch exceptions in any or all activities within a flow. If there are multiple flows in an app, the error handler must be configured for each flow separately. Branching is supported for error handlers flows similar to the other flows.
To configure the error handler, follow these steps:

Procedure

1. Click an existing activity in a flow.
2. Click the **Error handler** tab.
   The error handler opens with the **error** activity displayed.



Clicking the **error** activity exposes the fields that you can configure for an error that will be thrown by the activity.



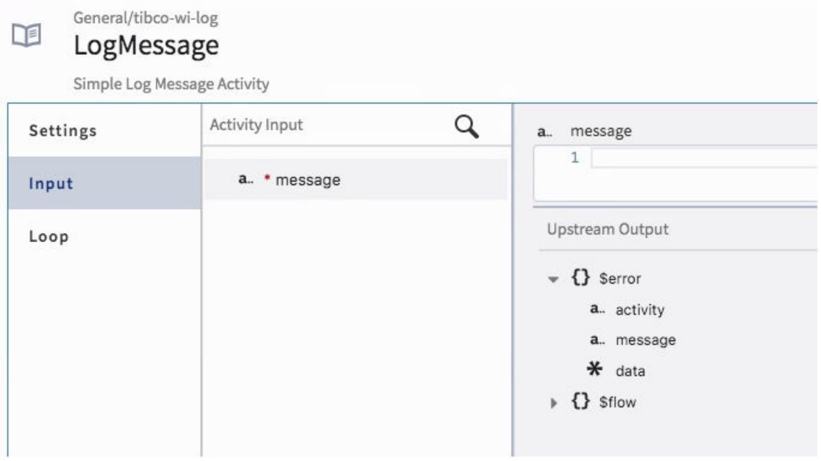
The **Map to Flow Inputs** tab of the **error** activity has three elements, **activity**, **message**, and **data**. The **activity** element is used to output the name of the activity that is throwing the error, the **message** element is used to output the error message string, and the **data** element can be configured to output any data related to the error. The **message** element in the **Input** tab of any activity in the Error Handler flow can be configured to output one or all of these three elements.

3. Hover your mouse next to the **error** activity to expose the + button.

4. Click the + button to add an activity for which you want to configure the error message.
   The **Input** tab of that activity displays **message** in its input schema. This is a required element which you must mandatorily map.

   > A **Return** activity is not added by default. Depending on your requirements, you must add the **Return** activity manually.

5. Click **message** in the input schema to open its mapper.



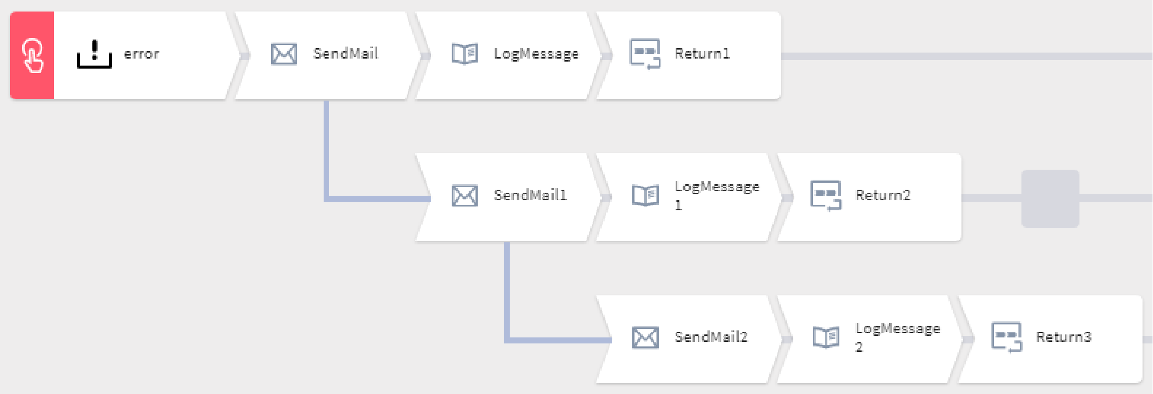
6. Expand {} **$error** to expose the **activity**, **message**, and **data** elements that you can configure for the error message.
   To map the **message** element under **Activity Input**, you can either manually type in the error string enclosed in double quotes or use the **concat** function under **string** in the mapper to output the activity name along with a message. See Using Functions for more details.

7. Continue configuring the error message for each activity in the flow.

   The error for any activity in any flow in the app, if any, is output in the log for the app when the app is .

Result

Here is an example of how an error handler flow looks after it is configured:



Viewing Errors and Warnings

TIBCO Cloud Integration - Flogo (PAYG) uses distinct icons to display errors and warnings within an app.

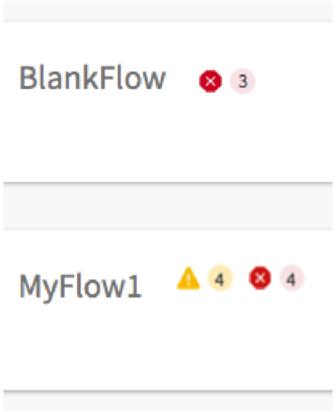
The following icons are used:

⊗ - Error icon. You must resolve the errors before building the app. Errors should not be ignored.

⚠ - Warning icon. Warnings are thrown to alert you of something that might need to change in the entity where the warning icon is displayed. You have the option to ignore the warning and move on.

Here is the hierarchy of errors and warnings reporting in TIBCO Cloud Integration - Flogo (PAYG):

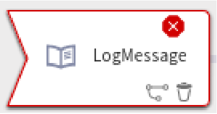
Flow level reporting - When you click on an app name, the app details page opens displaying the list of flows in the app. If there are errors or warnings in a flow, appropriate icons are displayed next to the flow name along with a number, where the number indicates an aggregate number of errors or warnings in the flow. If there are no errors or warning these icons do not display. In the image below, BlankFlow flow has three errors and MyFlow1 has 4 errors that should be resolved and 4 warnings that can be ignored.



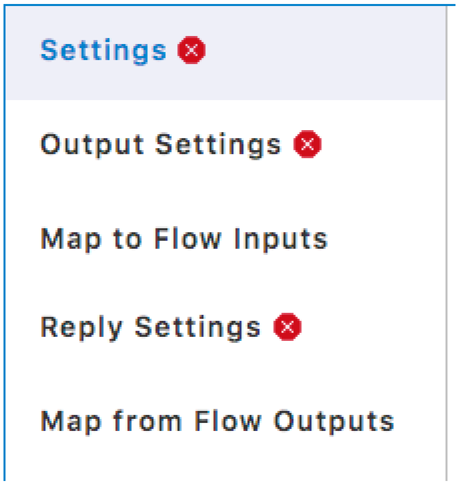
BlankFlow ⊗ 3

MyFlow1 ⚠ 4 ⊗ 4

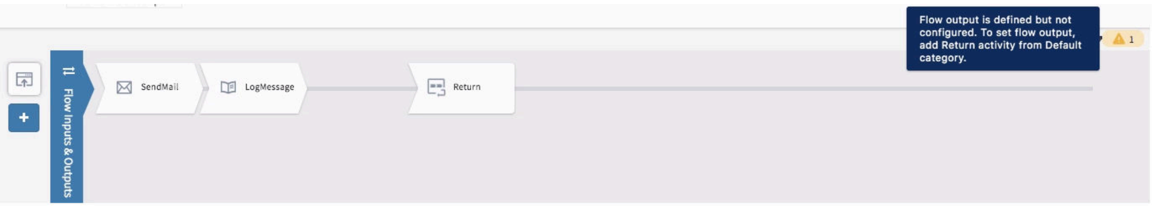
Activity and Trigger level reporting - When you click on a flow name, the flow details page opens displaying the implementation of the flow. This page displays errors if any at the activity level. For instance, the **LogMessage** activity below displays an error symbol within the activity configuration. You should resolve the error before proceeding.



Activity and Trigger configuration tab level reporting - When you click on an activity or a trigger in the flow, its configuration page opens, displaying the various tabs. Click a tab to see the errors or warnings in the configuration within that tab.



Blank space between activities - If there are empty spaces between the activities, you see an error message as shown below. For example, the following image shows a blank space between the **LogMessage** activity and the **Return** activity. Resolve this by clicking and dragging the **Return** activity to the left such that there are no blank spaces in the flow.


Flow output is defined but not configured. To set flow output, add Return activity from Default category.

Using Subflows

TIBCO Cloud Integration - Flogo (PAYG) provides the ability to call any flow from another flow in the same app. The flow being called becomes the subflow of the caller flow. This helps in separating the common app logic by extracting the reusable components in the app and creating standalone flows for them within the app. Any flow in the app can become a subflow for another flow within the same app. Also, there are no restrictions on how many subflows a flow can have or how many times the same subflow can be called or iterated in another flow. Hence, subflows are useful when you want to iterate a piece of app logic more than once or have the same piece of logic repeat in multiple locations within the app.

Here are a few points to keep in mind when creating and using subflows:

- The subflow and its calling flow must both reside within the same app. You cannot call a flow from another app as a subflow in your app.
- Since you can call any flow from any other flow within the app, you must be careful not to create cyclical dependency where a flow calls a subflow and the subflow in turn calls its calling flow. This will result in an infinite calling cycle and you will receive an error "Cyclic dependency detected in the subflow".
- **Important!!** You can delete any flow in an app even though the flow might be in use as a subflow within another flow. You will not receive any error messages at the time of deletion, but when you run the app, its execution fails with an error.
- You can configure the iteration details in the **Loop** tab of the **Start a SubFlow** activity. The **Start a SubFlow** activity iterates multiple times, resulting in the subflow being called multiple times.

Creating Subflows

You create a subflow exactly like you would create any other blank flow.
To create a subflow, do the following:

Procedure

1. Identify the piece of logic in your app that you want to reuse elsewhere in the app or iterate multiple times.
2. Create a flow without a trigger for that logic. See the Creating a flow without a trigger section for details on how to create such a flow.
3. To use this flow as a subflow within another flow, you must add a **Start a SubFlow** activity at the location in the calling flow from where you want to call the subflow. For example, if you want to call a subflow after the third activity in your calling flow, insert a **Start a SubFlow** activity as the fourth activity in the calling flow. To do so, follow these steps:
    1. Open the calling flow.
    2. On the flow details page, click the (+) button in the location within the flow from where you want to call the subflow. The **Add Activity** dialog opens.
    3. Click the **Default** tab and select the **Start a SubFlow** activity.
    4. Configure the **StartaSubFlow** activity to point to the subflow you want to call by selecting the subflow from the **Select flow** dropdown list in the **Settings** tab.

        The schemas that you had configured in the **Input Settings** and **Output Settings** of the **Flow Inputs&Outputs** tab in the selected subflow appear in the **Input** and **Output** tabs of the **StartaSubFlow** activity.

        You can now configure the input and output for the subflow in the **StartaSubFlow** activity. If you add additional input and/or output parameters in the **Flow Inputs & Outputs** tab of your subflow, they become available to configure from the **Input** and/or **Output** tabs of the **StartaSubFlow** activity. The output from the **StartaSubFlow** activity is available for use as input in all activities that appear after it.

        At app runtime, the **StartaSubFlow** activity in the calling flow calls the selected subflow.
    5. If you want your subflow to iterate multiple times, be sure to configure the iteration details in the **Loop** tab of the **StartaSubFlow** activity. Refer to Using the Loop section for details on how to configure the **Loop** tab.
    6. Build the app. See Building the App section for details on how to build the app.

Creating a Flow Execution Branch

Activities in a flow can have one or more branches. If you specify a condition for a branch, the branch executes only when the condition is met. You also have the option to create an error branch from an activity. The purpose of the error branch is to catch any errors that might occur during the execution of the activity. Branching is also supported for Error Handler flows, which serve the purpose of catching all errors at the flow level.

> You cannot create a branch from a trigger or a **Return** activity.

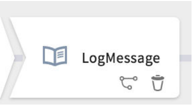
A **Return** activity ends the flow execution. So, regardless of whether the flow execution encounters a **Return** activity in a branch or at the end of the flow itself, as soon as the flow execution encounters a **Return** activity anywhere, it exits the flow from that location.

> A **Return** activity is not added by default. Depending on your requirements, you must add the **Return** activity manually. For example, if any trigger needs to send a response back to a server, its output must be mapped to the output of the **Return** activity in the flow.

To create a flow execution branch, follow these steps:

Procedure

1. From the **Apps** page, click the app name then click the flow name to open the flow details page.
2. Hover your mouse cursor over the activity to expose the icons for adding a branch and deleting the activity in the bottom right of the tile.

3. Click the **Add Branch** icon ( ).
   A branch gets created and the **Add Activity** dialog opens.

   Each branch has a label associated with it. The label has the following format:

   - When branching to an empty activity:

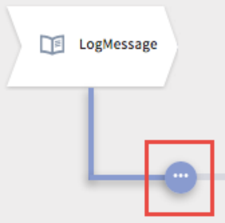
     `<Name of activity in main flow>to`

     For example, `LogMessageto`.
   - When branching to a specific activity:

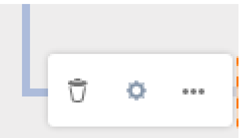
     `<Name of activity in main flow>to<Name of activity in branch>`

     For example, `LogMessagetoInvokeRESTService`.

4. Add activities to the branch flow as you would do to any other flow by clicking on the ⊞ button.
5. If you want the flow execution to terminate after this branch executes successfully, be sure to add and configure the **Return** activity at the end of the branch. If you do not want the flow execution to terminate, do not add a **Return** activity at the end of the branch.
6. Hover your mouse cursor to the end of the branch until you see a button with three dots placed horizontally.



7. Click the button to expose the following options:



8. Click the branch settings button (⚙).
   The **Branch Mapping Settings** dialog opens.
9. Select a branch condition: **Success**, **Success with condition**, **Success with no matching condition**, or **Error**.
   See the section, Types of Branch Conditions, for details on the three conditions.
10. Click **Save**.
11. Add condition to a branch as need be. See Setting Branch Conditions for details.

Types of Branch Conditions

TIBCO Cloud Integration - Flogo (PAYG) supports multiple types of branch conditions.

You must select one of the following conditions during branch creation:

- **Success**

  A success branch gets executed whenever an activity executes successfully. If there is an error in the activity execution, this branch does not execute. The branch has no conditions set in it.
- **Success with condition**

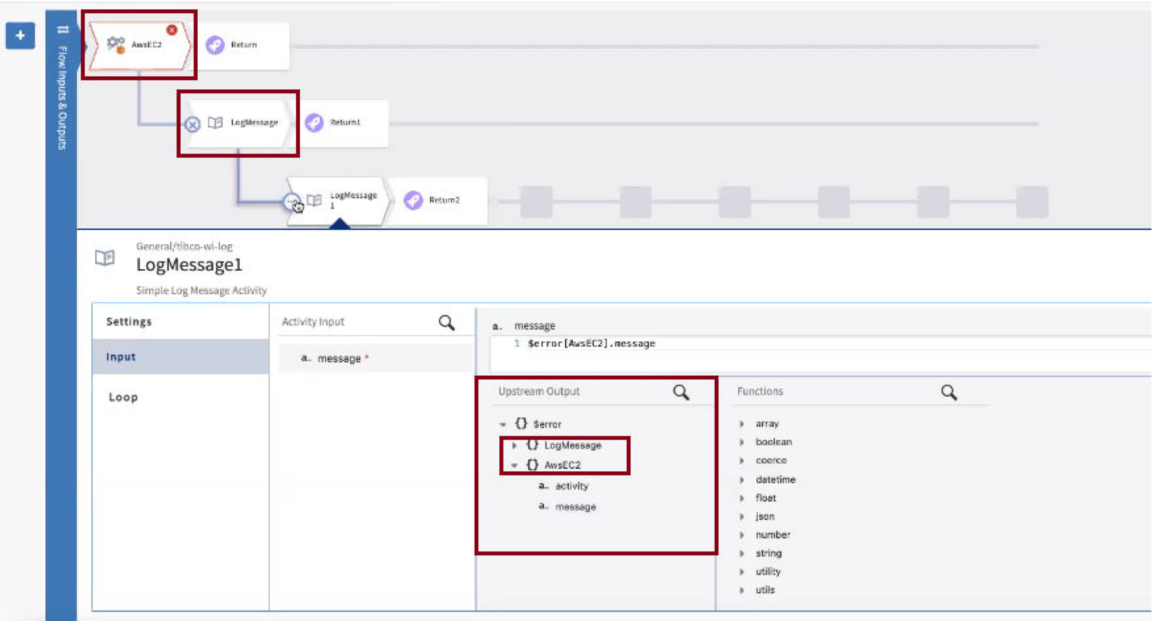
  Select this condition if you want a branch to execute only when a particular condition is met. If you select this condition and do not provide the condition, the branch never gets executed.

  You can form an expression using anything available under upstream activity outputs and available functions which should evaluate to a boolean result value.
- **Success with no matching condition**

  This branch condition is displayed only when you already have an existing **Success with condition** branch.
- **Error**

  A branch with this condition executes if there are errors in the execution of the activity. An activity can have only one **Error** branch.

  Details of the error, such as the activity and the type of the error message, are returned in `$error`. For example:



```
1 $error[AwsEC2].message
```

The **Error** branch flow differs from the error handler flow in that the error branch is designed to catch exceptions at the activity level from which the error branch originates, whereas the error handler flow is designed to catch exceptions that occur in any activity within the flow. So, if you handle the errors by creating an error branch at the activity level, the flow execution control never transfers to the error handler flow.

Order in which Branches Get Executed

When an activity has multiple branches, regardless of the number of branches or the order in which the branches appear in the Web UI, the branch execution follows a pre-defined order.

The flow execution will terminate if it encounters a **Return** activity at the end of any branch. In such situations, the activities that are placed after the branched activity in the main flow do not get executed.

The order in which the branches get executed is as follows:

1. **Success** branch and **Success with condition** branch

   When an activity has both **Success** and **Success with condition** branches, the order of execution depends on the order in which each branch was created. The branch that was created last gets executed first. All **Success** branches get executed unconditionally, but a **Success with condition** branch gets executed only if its branch condition is met.

2. **Success with no matching condition** branch

   This branch condition is displayed only when there is at least one existing **Success with condition** branch for the activity. The **Success with no matching condition** branch is typically used when you want a specific outcome in the event that none of the **Success with condition** branches meet their condition.

   - The **Success with no matching condition** branch executes only if none of the **Success with condition** branches execute. If the **Success with condition** branch executes and it does not have a **Return** activity at the end of the branch, the flow execution control gets passed to the main flow. If the **Success with condition** has a **Return** activity, the flow execution gets terminated after the **Success with condition** branch executes.

   - If an activity has one or more **Success with condition** branches but does not have any **Success with no matching condition** branch, if no matching condition is found, none of the **Success with condition** branches get executed. But, since there is no **Success with no matching condition** branch, the flow execution control gets passed back to the main flow and the activity next to the branched activity gets executed.

- If you delete all **Success with condition** branches without deleting the **Success with no matching condition** branch, you receive a warning informing you that the **Success with no matching condition** branch is orphaned.

The **Error** branch gets executed as soon as the flow execution encounters an error.

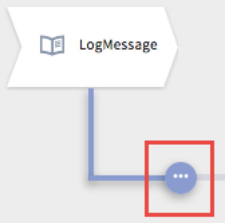
Setting Branch Conditions

You can set conditions on a branch such that only if the condition is met the branch will execute.

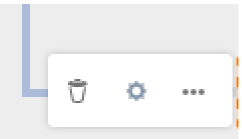
To set conditions on a branch, follow these steps:

Procedure

1. Hover your mouse cursor to the end of the branch until you see a button with three dots placed horizontally.

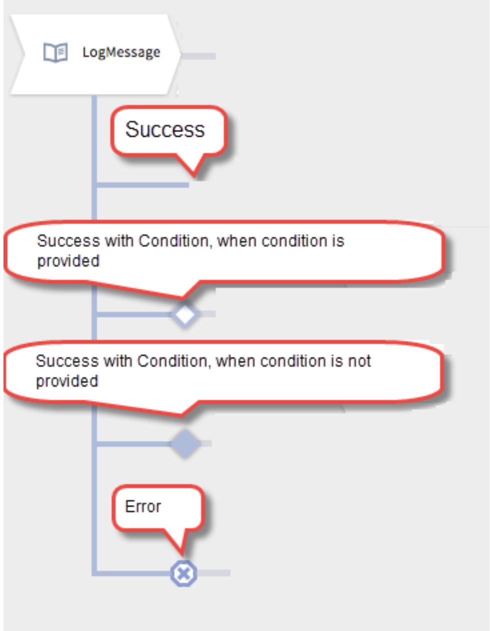
2. Click the button to expose the following options:

3. Click (⚙).
   The **Branch Mapping Settings** dialog opens.
4. Select a branch condition: **Success**, **Success with condition**, or **Error**. If you already have a **Success with condition** branch present, you will also see **Success with no matching condition**.
   See the section, Types of Branch Conditions, for details on the three conditions.
5. Click **Save**.
6. If you selected **Success with condition**, the mapper opens for you to set the condition. Click **condition**. The mapper is exposed to the right of the dialog. The functions that you can use to form the condition are shown under **Functions**.
7. Enter an expression with the condition or click a field from the output of a preceding activity to use it. The output from preceding activities appears under the left **Upstream Output** in a tree format.

   ! The condition must resolve to a boolean type.

   The following image shows how the branches appear based on the branch condition:

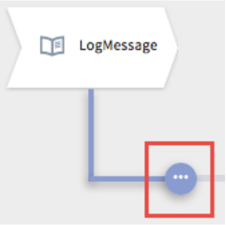

LogMessage
Success
Success with Condition, when condition is provided
Success with Condition, when condition is not provided
Error

Deleting a Branch

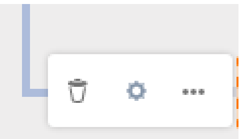
You can delete a branch at any time after creating it.
To delete a branch:

Procedure

1. Hover your mouse cursor to the end of the branch until you see a button with three dots placed horizontally.

2. Click the button to expose the following options:

3. Click (🗑).

   If the branch to be deleted contains a sub-task located under branches or sub-branches, the following confirmation dialog is displayed.

4. On the confirmation dialog, click **Delete branch**.

Result

The selected branch is deleted.

Duplicating a Flow

You can duplicate an existing flow in an app. All activities in the flow along with their existing configurations get duplicated to a new flow in the app and the duplicate of the original flow gets created with a default name beginning with "Copy of" in the same app. You can rename the flow by clicking on the flow name on the top left corner of the flow details page. Duplicating a flow saves you time and effort in situations when you want to create a flow with similar or same activities as an existing flow in the app. After you have duplicated the flow, you can add more activities, rearrange existing activities by dragging and dropping them in the desired location, or delete activities from the flow duplicate.

The triggers in the flow do not get duplicated. Also, if a flow has subflows, the subflows do not get duplicated.

To duplicate a flow, follow these steps:

Procedure

1. Open the **Apps** page and click on the app to open the app details page.
2. Hover your mouse to the extreme right of the flow that you want to duplicate until the **Duplicate flow** icon (⧉) displays.
3. Click the **Duplicate flow** icon. A duplicate of the flow gets created in the app.
4. Edit the duplicated flow as needed to add, rearrange, or delete activities in the flow and rebuild the app.

Editing a Flow

You can edit the flow name or its description after creating the flow. You can also add more activities, rearrange existing activities by dragging and dropping them in the desired location, or delete activities from the flow.
To edit a flow, follow these steps:

Procedure

1. On the **Apps** page, click the app name to open the app details page.
2. Click the flow name which opens the flow page. You must rebuild the app after making the required changes.
   To edit the flow name click anywhere in the flow name and edit the name. To add a new activity between two existing activities, you must make space for the new activity by dragging each of the activities one space to its right starting with the last activity in the flow. Once the space is made, click the ⊞ button in the blank space that you just created to add a new activity.

Reverting Changes to a Flow

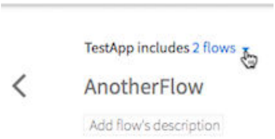
If you have multiple flows in an app, you cannot revert changes made to a single flow. However, you can click the option under the hamburger menu (⋮) that is next to the other buttons on the app page, to revert *all* changes made to the app and revert the app to the state that it was in after the last . All changes made to the app will be lost. This can be done as long as you have not the app after making the changes.

Switching Between Flows in an App

If an app has multiple flows, you can switch between the flows within an app.
When you have one flow open, to switch to another flow within the app, do the following:

Procedure

1. Click the down arrow to the right of the flow name and select the flow page you want to open from the drop down list.

   The **Select to view other flows in <*appname*>** panel opens with all the flows in the app listed and the currently open flow selected.
2. Click the flow to which you want to switch.

Deleting a Flow

You can delete a flow from the app details page.
To delete a flow:

Procedure

1. On the Apps page, click the app name to open its app details page.
2. Hover your mouse cursor to the extreme right of the flow name that you want to delete until the **Delete flow** icon (🗑) displays.
3. Click the **Delete flow** icon.
4. On the confirmation dialog, click **Delete**.

Result

The selected flow is deleted.

> If multiple flows are attached to a trigger only the specific flow gets deleted, but if the flow you are deleting is the only flow attached to the trigger, the trigger gets deleted as well.

Adding an Activity

After a flow is created, you must add activities to the flow.

Procedure

1. From the **Apps** page, click the app name then click the flow name to open the flow details page.
2. Hover your mouse cursor to the right of the **Flow Inputs and Outputs** accordion tab to expose the (+) buttons.
3. Click the (+) button.
4. In the **Add Activity** dialog, click the category tab from which you want to add an activity. For example, to add a general activity such as **Log Message**, click the **General** tab.
5. Select the activity you want to add by clicking it.
   The activity gets added to the flow.

6. To rearrange the order in which the activities appear in the flow, click the activity and drag and drop it to the required location within the flow.
7. Click the activity to open its configuration dialog and configure it.

Searching for a Category or Activity

You can search for an activity or category by entering the activity name or category name in the **Search activity and category** box in the **Add Activity** dialog.

You can enter either the full name of the activity or category or you can enter its partial name (a string of characters appearing in the name) in the **Search activity and category** box.

- All categories whose names either wholly match the search string or contain the partial search string in their name get displayed.
- When a category displays in the search result, only those activities in the category whose name contain the search string get displayed. If the category also contains other activities whose names do not match or contain the search string, such activities are not displayed.
- For any activity whose name wholly or partially matches the search string, the category that contains that activity is displayed. For example, if you enter "delete" in the search box, since there are activities whose name contains the string "delete" in Marketo, Salesforce, Zoho-CRM, and so on, all these categories are displayed, even though the category names themselves do not contain the string "delete".

Configuring an Activity

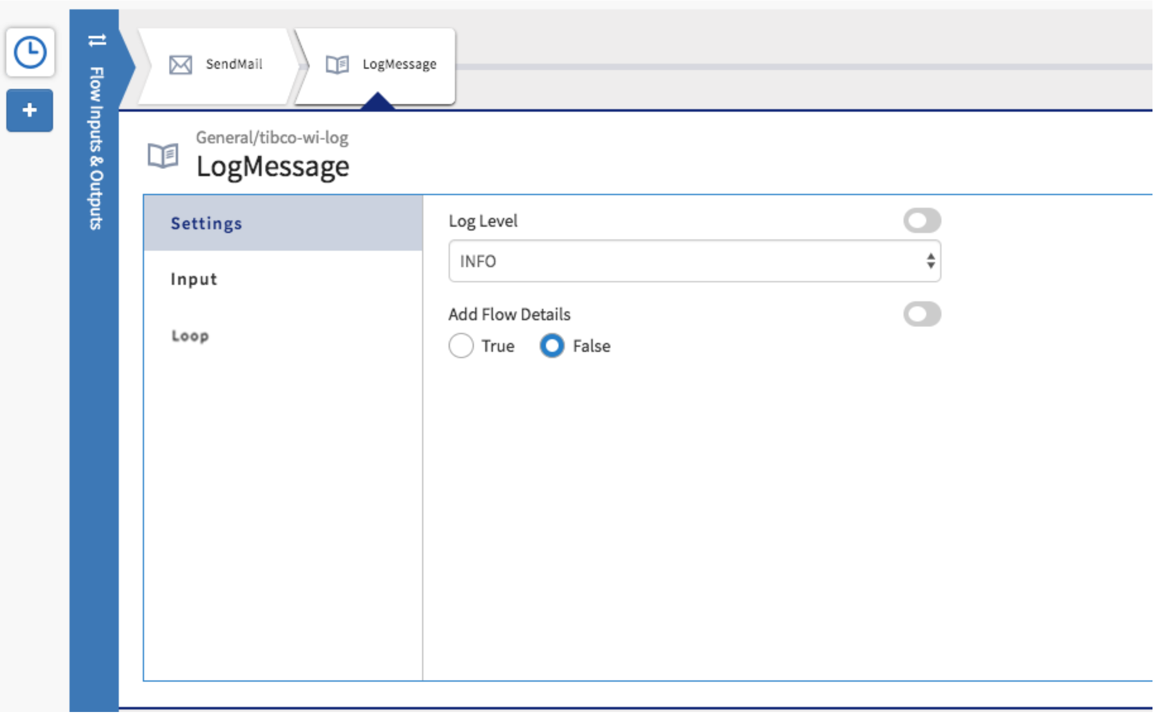
After adding an activity, you must configure it with any input data that the activity might need and the output schema for activities that generate an output.
There are three ways to configure data for an activity:

- Configuring static data where you manually type the data in the mapper for the field, for example, type in a string that you want to output. Strings must be enclosed in double quotes. Numbers must be typed in without quotes.
- Mapping an activity input to the output from one of the activities preceding it in the flow, provided that the previous activities have some output.
- Using functions, for example, the `concat` function to concatenate two strings. Nested functions are currently not supported.

To configure an activity, do the following:

Procedure

1. On the flow details page, click an activity.
   The configuration box opens beneath the activity.



General/tibco-wi-log
LogMessage

Settings

Log Level: INFO

Add Flow Details: ○ True ● False

2. Click on each tab in the configuration box under the activity name and either manually enter the required value, use a function, or in the **Input** tab, map the output from the trigger or a preceding activity using the mapper. Refer to the Mapper section for details on mapping.
   If one or more activities are not configured properly in a flow, the error or warning icon is displayed on its upper right corner. Click the activity whose tab contains the error or warning. Refer to Errors and Warnings section for more details.

Duplicating an Activity

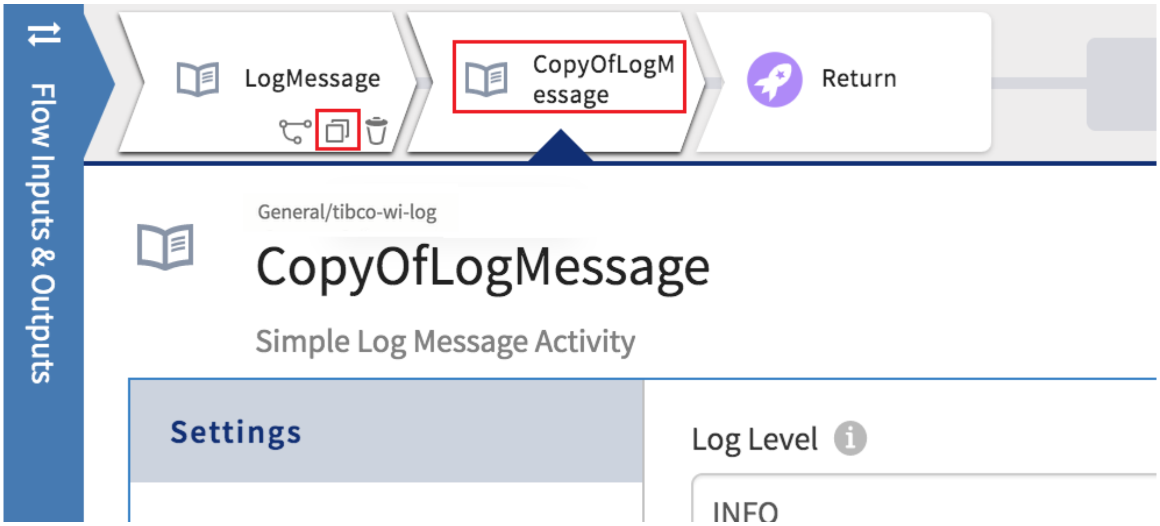
You can duplicate an activity within the same flow. The activity along with the existing configuration is duplicated to a new activity. The duplicate of the original activity is created with a default name beginning with `CopyOf`. You can rename the activity by clicking on the activity name. Duplicating an activity saves you time and effort in situations when you want to create an activity with similar or same configurations as an existing activity in the flow. After you duplicate the activity, you can change the configuration, move it around in the flow by dragging and dropping it to the required location, or delete it from the flow.

A trigger within a flow cannot be duplicated.

Procedure

1. From the **Apps** page, click the app name, and then click the flow name to open the flow details page.
2. Hover your mouse cursor over the activity that you want to copy and click ⧉.

   For example, in the following screenshot, the **LogMessage** activity is duplicated and added to the flow. The duplicate activity is called **CopyOfLogMessage**:



3. Configure the duplicated activity as required.

Using the Loop Feature in an Activity

When creating a flow, you may want to iterate a certain piece of logic multiple times. For example, you may want to send an email to multiple people based on the output of a certain activity (let's call it *activity1*) in your flow. You can do this by adding a **SendMail** activity following *activity1* in your flow and configure the **SendMail** activity to iterate multiple times when *activity1* outputs the desired result. Each iteration of the **SendMail** activity is used to send an email to one recipient. This saves you the effort of creating multiple **SendMail** activities.
Keep the following in mind when using the Loop feature:

- Iteration is supported for an activity only. You configure the iteration details in the **Loop** tab of the activity.
- There are certain activities that do not require iteration, for example, the **Return** activity, whose purpose is to exit the flow execution and return data to the trigger. The **Loop** tab is not available in such activities.
- You cannot iterate through a trigger.
- For apps that were created in Project Flogo™ and imported into TIBCO Cloud Integration - Flogo (PAYG), the `key` type in the **Loop** tab is converted from string to the relevant data type of value in TIBCO Cloud Integration - Flogo (PAYG).

To configure multiple iterations of an activity, do the following:

Procedure

1. Click the activity in the flow to expose its configuration tabs.
2. Click the **Loop** tab.
3. Select a type of iteration from the **Type** menu.
   The default type is **None**, which means the activity will not iterate.

   Iterate

   This type allows you to enter a number that represents the number of times you would like the activity to iterate without taking into account any condition for iterating.

   Click **iterator** to open the mapper to its right. You can either enter a number (integer) to specify the number of times the activity must iterate or you can set an expression for the loop by either entering the expression manually or mapping the output from the preceding activities or trigger. You can also use available functions along with the output from previous activities and/or manually entered values to form the loop expression. The loop expression determines the number of times the activity iterates.

> The loop expression must either return a number or an array. The array can be of any data type. If your loop expression returns a number, for example 3, your activity will iterate three times. If your loop expression returns an array, the activity will iterate as many times as the length of the array. You can hover over the expression after entering the expression to make sure that the expression is valid. If the expression is not valid, you will see a validation error pop up.

If you select this type, the **Input** tab of the activity displays the `$iteration` scope in the output area of the mapper. `$iteration` contains three properties, **key**, **index**, and **value**. **index** is used to hold the index of the current iteration and **value** holds the value that exists at the index location of the current iteration if the loop expression evaluates to an array. If the loop expression evaluates to an array of objects, **value** also displays the schema of the object. If the loop expression evaluates to a number, the **value** will contain the same integer as the **index** for each iteration. To examine the result of each iteration of the activity, you can map **index** and **value** to the **message** input property in the **LogMessage** activity and print them. **key** is used to hold the element name when configuring a condition if the value evaluates to an object. However, you can map only to the output of the last iteration if you did not set the **Accumulate Output** check box to **Yes**. See Accumulating the Activity Output for All Iterations section for more details on this.

Repeat while true

Refer to https://github.com/TIBCOSoftware/tci-flogo/tree/master/samples/app-dev/loops.sample for an example of how to use this feature.

Select this type if you want to set up a condition for the iteration. This acts like the `do-while` loop where the first iteration is executed without checking the condition and the subsequent iterations exit the loop or continue after checking the condition. You set the condition under which you want the activity to iterate by setting the **condition** element. The condition gets evaluated before the next iteration of the activity. The activity iterates only if the condition evaluates to true. It stops iterating once the condition evaluates to false. Click **condition**, and manually enter an expression for the condition. For example, `$iteration[index] > 5`.

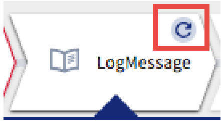
Keep in mind that even though the index for the **Repeat while true** iteration begins at zero, it does not iterate n+1 times. If you enter 4 as the iterator value then it will execute as the following iterations: 0,1,2,3.

By default, the results of only the final iteration are saved and available. All previous iteration results are ignored. If you would like the results of all iterations to be stored and available, set **Accumulate** to **Yes**.

You have the option to set a time interval (in ms) between each iteration, which can help you manage the throughput for your machine. To spread the iterations out, set the **Delay** element. Default delay time is 0 ms, which results in no delay.

Result

After you enter the loop expression, the loop icon appears on the top right corner of the activity leaf as follows:



Accumulating the Activity Output for All Iterations

When using the Loop tab to iterate over an activity, you have the option to specify if you want the Loop to output the cumulative data from all iterations. You can do so by setting the **Accumulate** check box to **Yes**. If the **Accumulate** check box is not selected, only the output from the last iteration is retained and available for mapping in the downstream activities.

When the **Accumulate** check box is set to **Yes**, the activity accumulates the data from each iteration and outputs that collective data as an array of objects, where each object contains the output from the corresponding iteration. The accumulated results will be displayed as an array in the downstream activities in the mapper and be available for mapping.

When mapping to an element within an object in the output array of the activity, you must provide the index of the element to which you want to map. For instance, when you click on a property within the object under **responseBody**, the expression displayed in the mapper will be `$activity [`*`<activity-name>`*`] [<<index>>].responseBody.`*`<property-name>`*. You must replace `<<index>>` with the actual index of the object to whose property you want to map.

When the **Accumulate** check box is not selected, the output of the Loop displays an object that contains only the data from the last iteration. Data from all previous iterations is ignored. When mapping to an element in the output object of the activity, when you click on a property within the object under **responseBody**, the expression displayed in the mapper will be `$activity [`*`<activity-name>`*`].responseBody.`*`<property-name>`*.

The **Output** tab of the activity changes based on your selection of the **Accumulate** check box. The parent element (the name of the activity and the data type of the iteration output) is displayed regardless of your selection. If you set the **Accumulate** check box to **Yes**, the data type of the parent element is an array of objects. If you did not select the check box, the data type of the parent element is an object. The **Output** tab contents will also be available in the mapper allowing for the downstream activities to map to them.

Accessing the Activity Outputs in Repeat While True Loop

This feature is useful when an activity needs to use the loop feature to do batch processing or fetch multiple records by executing the activity multiple times. With each iteration of the activity, its output will available for mapping to the activity input.

For example, if a Salesforce query returns 200 records and a locator, the locator can be used by the next batch query. This locator can be mapped into the activity input by mapping the activity output from the previous iteration which contains the locator for the next record set or query. Hence when the next iteration of the activity runs, it fetches the next record indicated by the locator that was mapped to the activity input.

This feature is available in all activities that generate an output (have an **Output** tab).

To use this feature, follow these steps:

Procedure

1. In the **Loops** tab, set the **Type** to **Repeat while true**.
2. Set the **Access output in input mappings** to **Yes**. This allows the output of the activity iteration to be available in the **Upstream Output** for mapping. Now you can map your activity output to the activity input parameter.
3. Enter a **condition** in its text box. The activity evaluates this condition before each run. If the condition evaluates to `true` the activity executes. Note that the output is only available in subsequent iterations after the first iteration. Since the activity output is not available for the first iteration, your condition must perform a check to see if it is the first iteration of the activity. For example, use `$iteration[index]> 0 && isdefined($activity[SFQuery].output.locator)` to begin your condition. The `$iteration[index]> 0` checks to make sure that it is not the first run of the activity. The `isdefined($activity[SFQuery].output.locator)` function checks whether the output field exists.

Deleting an Activity

You can delete an activity in a flow from the flow details page.

Procedure

1. On the **Apps** page, click the app name then click the flow name to open the flow details page.
2. Hover your mouse cursor over the activity you want to delete and click 🗑.

Triggers

Triggers are used to activate flows. This section contains information on creating and managing triggers in your app.

Creating a Trigger without a Flow

You have the option to either create a trigger as a part of the process of creating a flow or you can create a trigger without creating a flow.

Refer to the section, Creating a Flow, to create a trigger during the flow creation process.

To create a trigger without creating a flow, follow the steps below:

Procedure

1. On the app details page, click **Create**.
   The **Add Triggers and Flows** dialog opens.
2. Under **Create new**, click **Trigger** to select it.
   The triggers catalog opens to the right.
3. Select the trigger you want to create in the triggers catalog.
   The trigger gets created with a placeholder for a flow attached to it.

Deleting a Trigger

You can delete a trigger from the app details page by hovering over the trigger and clicking **Delete**.

Synchronizing Schema Between Trigger and Flow

If you make any changes to the schema that you entered when creating the trigger, you must explicitly save any changes you make, then propagate the changes to the flow input and flow output. This is done by synchronizing the schemas.
To synchronize the schema between the trigger and the flow, do the following:

Procedure

1. Click the trigger to open its configuration details.
2. Make your changes and click **Save**. If you do not click **Save**, you will see a warning message asking you to first save your changes before the schema can be synchronized.
3. Click the **Sync** button on the top right corner.
   The trigger output schema is copied to flow inputs and trigger reply schema is copied to flow outputs.

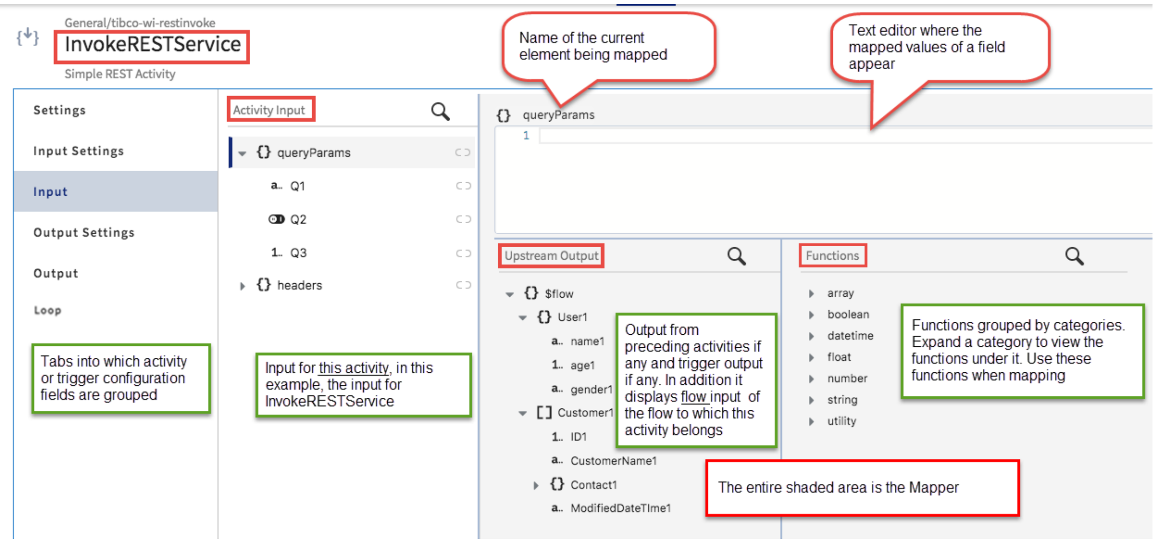
Data Mappings

TIBCO Cloud Integration - Flogo (PAYG) provides a graphical data mapper to map data between the activities within a flow, and between the trigger and the flows attached to the trigger within an app. Use the mapper to enter the flow or activity input values manually or map the input schema elements to output data of the same data type from preceding activities, trigger, or the flow itself.

Data Mappings Interface

An activity has access to the output data from the trigger to which the flow is attached in addition to the output from any of the activities that precede it in the same flow provided that the trigger or activity has an output. This data is displayed in a tree structure under **Upstream Output** in the Mapper. The input schema for the activity is displayed in the pane to the left of the **Upstream Output** pane. You can map data coming from the upstream output to the input fields of the activity. Activities also have access to the input fields of a flow to which the activity belongs. You enter the flow input schema in the **Input Settings** tab of the **Flow Inputs and Outputs** accordion tab.

When you click an activity or trigger on the flow details page, the configuration page for that activity or trigger opens. The following image is an example of the configuration page that opens if you clicked on the **InvokeRESTService** activity. The image describes the areas of the Mapper.



Name of the current element being mapped

Text editor where the mapped values of a field appear

Tabs into which activity or trigger configuration fields are grouped

Input for this activity, in this example, the input for InvokeRESTService

Output from preceding activities if any and trigger output if any. In addition it displays flow input of the flow to which this activity belongs

Functions grouped by categories. Expand a category to view the functions under it. Use these functions when mapping

The entire shaded area is the Mapper

The left most pane displays the tabs for the configuration fields for that activity or trigger. Each activity or trigger has one or more of the following tabs:

- **Settings**

  For triggers, this tab displays the Trigger settings and Handler settings. Trigger settings are specific to that particular trigger and Handler settings are settings applicable to a specific flow attached to that trigger. Each flow attached to that trigger can have its own handler settings.

- **Input Settings**

  This tab allows you to enter the schema for the flow or activity input.

- **Input**

  This tab displays the schema you entered in the **Input Settings** tab in a tree format. You can manually enter values for any elements in the input schema or map any input element to the output from previous activities or triggers in this tab.

- **Output Settings**

  This tab allows you to enter the schema for the flow or activity output.

- **Output**

  This tab displays the schema you entered in the **Output Settings** tab in a tree format. The schema displayed in this tab is set to read-only as it is for informational purposes only.

- **Reply Settings**

  This tab is applicable only to triggers that send replies back to the caller, such as the REST or GraphQL triggers. You enter the trigger reply schema in this tab.

- **Map to Flow Inputs**

  This tab is applicable only to triggers that have an output, such as the REST or GraphQL triggers. You manually enter or map the elements from the trigger output (schema set in **Output Settings** tab) to the flow input elements (schema entered in **Input Settings** tab of the **Flow Inputs & Outputs** accordion tab). This allows the output from the trigger to become the input to the flow.

- **Map from Flow Outputs**

  This tab is specific to triggers that need to send a reply to the caller, such as the REST or GraphQL triggers. You manually enter or map the elements from the output of the flow (schema set in **Reply Settings** tab) to the flow output elements (schema entered in **Output Settings** tab of the **Flow Inputs & Outputs**). This allows the output of the flow to become the reply that the trigger sends back to the request that it receives.

- **Loop**

  Use this tab to enter the iteration details for activities that you want to iterate.

When mapping, you can use data from the following sources:

- Literal values - Literal values can be strings or numeric values. These values can either be manually typed in or mapped to a value from the output of the trigger or a preceding activity in the same flow. To specify a string, enclose the string in double quotes. To specify a number, type the number into the text box for the field. Constants and literal values can also be used as input to functions and expressions.
- Direct mapping of an input element to an element of the same type in the **Upstream Output**.
- Mapping using functions - The mapper provides commonly used functions that you can use in conjunction with the data to be mapped. The functions are categorized into groups. Click a function to use its output in your input data. When you use a function, placeholders are displayed for the function parameters. You click a placeholder parameter within the function, then click an element from the **Upstream Output** to replace the placeholder. Functions are grouped into logical categories. Refer to Using Functions section for more details.
- Expressions - You can enter an expression whose evaluated value will be mapped to the input field. Refer to Using Expressions for more details.

Scopes in Data Mappings

The **Upstream Output** area in the mapper displays the output data from preceding activities, trigger, and/or flow inputs. This area groups the output elements based on a scope. A scope represents a boundary in the **Upstream Output** within which an input element can be mapped. For example, when mapping an input element to an element from the output of a trigger, the scope of the input element is represented in **Upstream Output** as **$trigger**. The following scopes are currently supported by the mapper.

Scope Name	Used to...	Available in...
$trigger	Map flow input to trigger output.	Trigger (**Map to Flow Inputs** tab) to map flow inputs to trigger outputs.

Scope Name	Used to...	Available in...
$flow	Map flow output to trigger reply.	- Trigger (**Map to Flow Outputs** tab) to map flow output to trigger reply. - Activities (**Input** tab) to map activity input to flow input. - Return activity (**Map Output** tab) to map flow output to flow input.
$activity.*[activity-name]*	Map input elements of the activity to elements from the output of previous activities.	$activity represents the scope of an activity. *[activity-name]* indicates the activity whose scope you are defining. Each preceding activity has its own scope in the mapper.
$iteration	Keep track of the current iteration and is available only when iterator is enabled for an activity in the **Loop** tab of the activity.	**Input** tab of an activity that has Loop enabled. Displays only when the Loop for the activity is enabled. Two elements are displayed under **$iteration**: - **key** - This element represents the iteration index, hence it is always of type number. For example, if the Loop expression is set to an array, the key element will represent the array index of the current iteration. - **value** - the value can be of any type depending on what is being iterated. For example, if you are iterating through an array of strings, the value is of type string.
$property *[property-name]*	Map to app properties that are defined in the app.	As long as there are app properties defined in the app, this scope is available for mapping from any activity that allows mapping.
$loop	Map elements within an array.	$loop is prefixed to the element name when mapping an element that is within an array. The scope of $loop is the current array that you are iterating through.

Reserved Keywords to be Avoided in Schemas

TIBCO Cloud Integration - Flogo (PAYG) uses some words as keywords or reserved names. Do not use such words in your schema. When you import an app, if the schema entered in the **Input** or **Output** tab of an activity or trigger contains reserved names, after the app is imported, such attributes get treated as special characters which results in runtime errors.

Avoid using the keywords listed below in your schema:

- break
- case
- catch
- class

- const
- continue
- debugger
- default
- delete
- do
- else
- enum
- export
- extends
- false
- finally
- for
- function
- get
- if
- import
- instanceof
- in
- new
- null
- return
- set
- super
- switch
- this
- throw
- true
- try
- typeof
- var
- void
- while
- with

Mapping Different Types of Data

The mapper opens when you click any element in the input schema tree in an activity configuration tab.

Mapping for the following is supported:

- A single element from the input to another single element in the output.

  If the single element comes from an array in the output, then you must manually add the array index to use. For example, $flow.body.Account.Address**[0]**.city

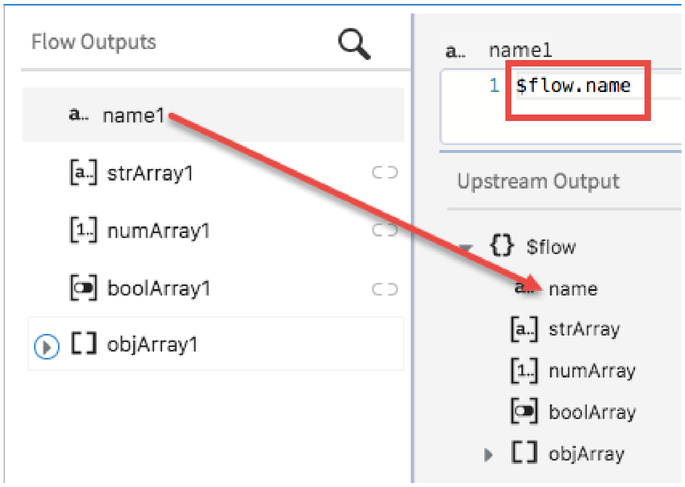
- A standalone object (an object that is not in an array)
- An array of primitive data type to another array of primitive data type.
- An array of non-primitive data types (object data type or a nested array) to another array of the same non-primitive data type.

Be sure to keep the following in mind when using the mapper:

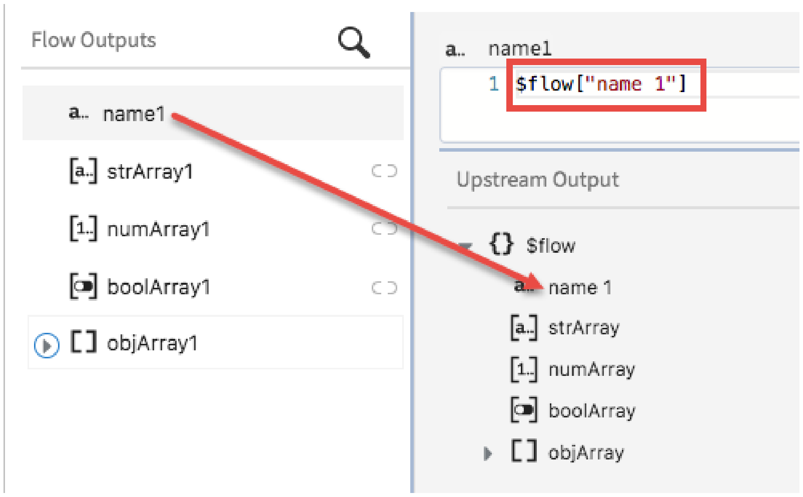
- Make sure that you map all elements that are marked as required (have a red asterisk against them), whether they are standalone primitive types, within an object, or within an array. When mapping identical objects or arrays, such elements get automatically mapped, but if you are mapping non-identical objects or arrays, be sure to map the elements marked as required individually.
- The `in` and `new` attributes are treated as special characters if you use them in the schema that you enter in the REST activity or trigger. For example, mappings such as `$flow.body ["in"]` and `$flow.body ["new"]` are not supported. If an imported app contains these attributes after the app is imported into TIBCO Cloud Integration - Flogo (PAYG), it results in runtime errors.
- Use of anonymous array is not supported in the **Flow Input & Output** tab and the **Return** activity configurations. To map to an anonymous array, you must create a top level object or a root element and render that.
- You can not use a scope (identified with a beginning `$` sign) in an expression, for example `renderJSON($flow, true)`. You can use an object or element under it for example, `renderJSON($flow.input, true)`.

You can only map one element at a time.

If the output element names contain special characters other than an underscore (_), they appear in bracket notation in the mapping text box. In the example below, in the upper image, **name** under **Upstream Output** does not contain any special characters, hence it is displayed in dot notation. In the lower image **name 1** contains a space, hence it appears in the bracket notation.



```
1 $flow.name
```

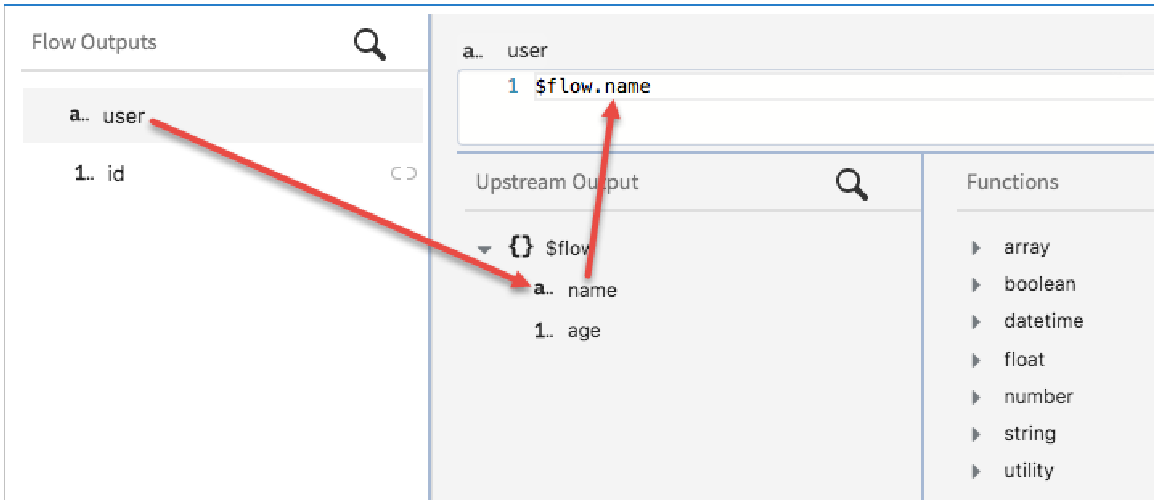


```
1 $flow["name 1"]
```

Mapping a Single Element of Primitive Data Type

You can map a single element of a primitive data type to a single element of the same type in the output schema under **Upstream Output**.

Click the element to be mapped (destination element) first, then click the element under **Upstream Output** (source from which the data comes in) to which you want to map it. In the example below, click **user** (destination), then click **name** (source) to map the **name** to **user**.

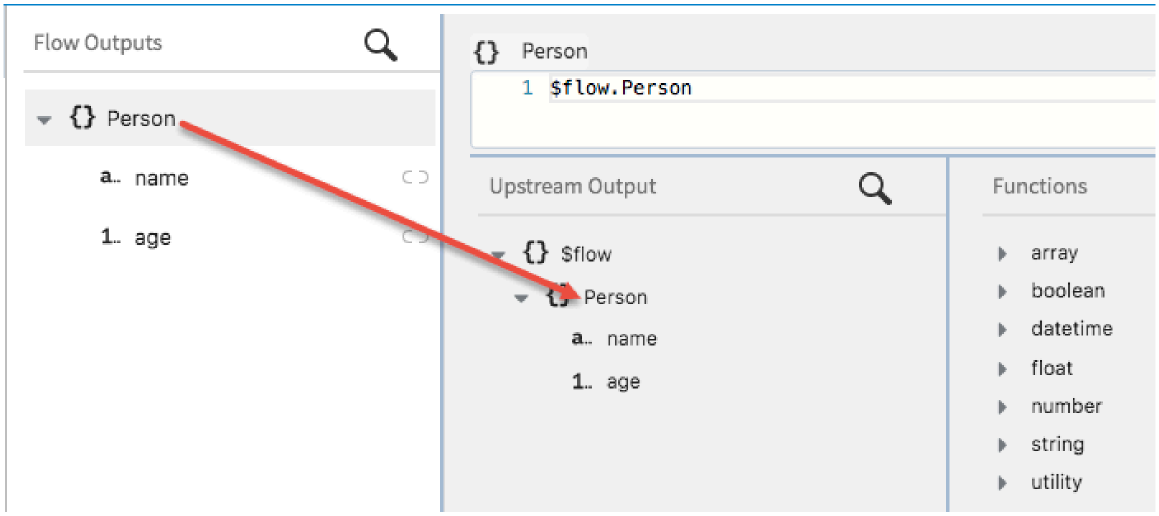


```
1 $flow.name
```

Mapping an Object

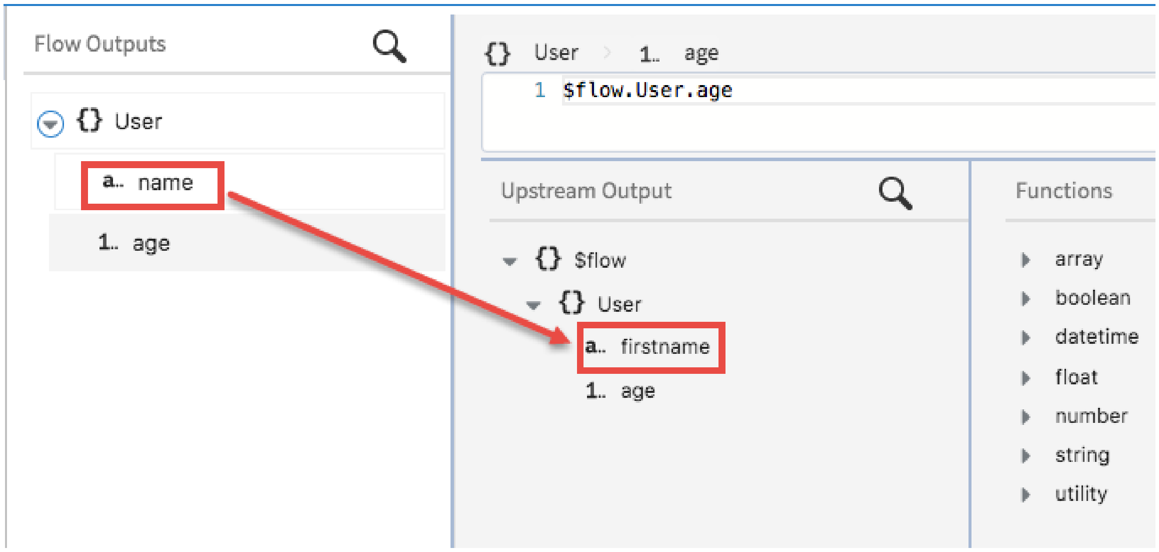
Standalone objects (objects not within an array) whose property data types match, can be mapped at the root level. If the destination object is identical to the source object under Upstream Output (both, the names of the properties as well as their data types match exactly), you need not match the elements in the object individually. If the property names are not identical, then you must map each property individually within the object.

For example, in the image below the **Person** objects are identical. So, you can map **Person** to **Person**. You need not map **name** and **age** individually.



```
1 $flow.Person
```

In the following image, the data types match but the property names do not match. In such a case, you must map each property individually in addition to mapping the object root.



```
$flow.User.age
```

Mapping Arrays

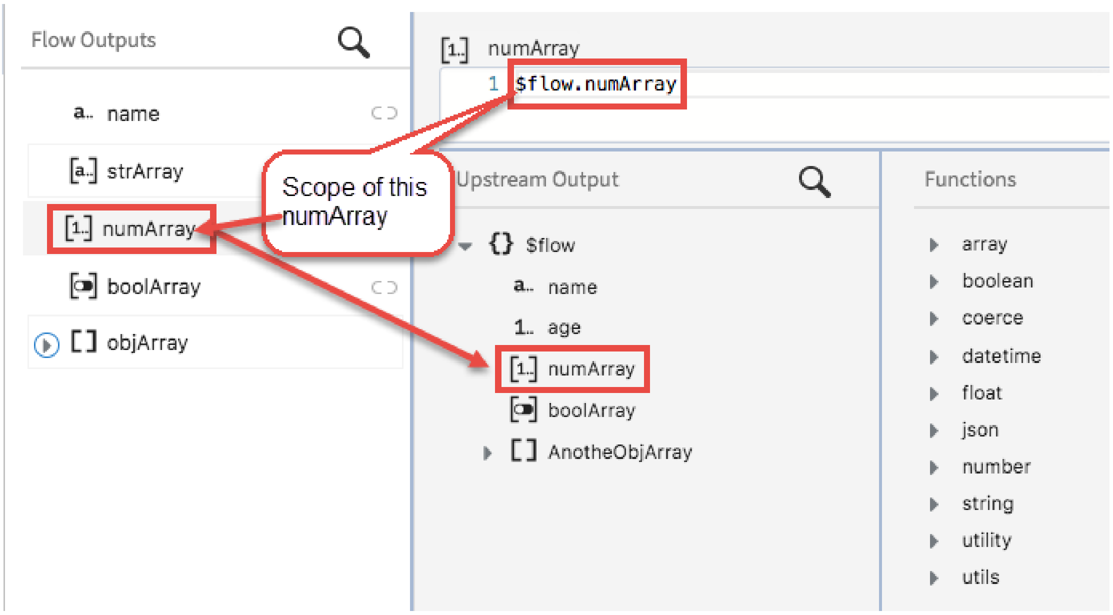
When mapping arrays, you must first map their array root before you can map their child elements.

The following mappings are supported when mapping arrays.

- Mapping arrays of primitive data types
- Mapping an array of objects
- Mapping nested arrays

Mapping an Array of Primitive Data Types

To map arrays of the same primitive data type, you only need to map the array root. You need not map the array elements.
Here is an example of mapping arrays of primitive data types:



```
$flow.numArray
```

Scope of this numArray

The array names need not match, but their data types must match. In **Upstream Output**, `$flow` points to **numArray** in **Upstream Output** which is the scope for **numArray** in the input.

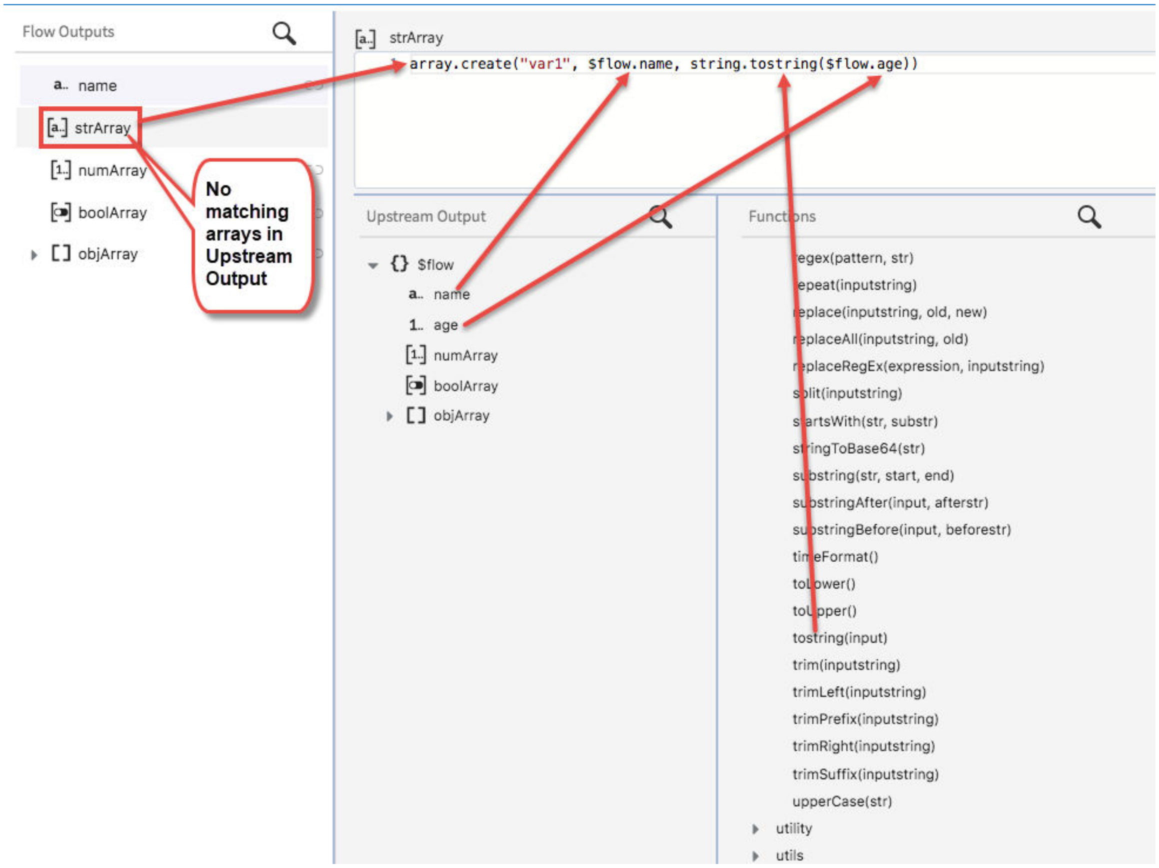
When you do not have a matching data type array in your output

If you want to map an array of primitive data types, but you do not have an array of the same data type in your **Upstream Output**, you can create an array using the `array.create(item)` function.

> `array.create(item)` can only be used to create an array of primitive data types. You cannot use it to create an array of objects.

To do so, follow these steps:

1. Click the array for which you want to do the mapping in the input schema. The mapper opens to its right.
2. Under **Functions**, click **array** to expand it.
3. Click `create(item)`. It appears in the text box above.
4. Click `item` to replace it with the output element to use to create the array.
5. Click the element in the **Upstream Output** with which you want to replace `item`. In the following image, to map **strArray**, you would need to create an array since there is no array of strings under **Upstream Output**. So, you map **strArray** by creating an array. The `array.create()` function accepts any of the following: a hardcoded string, an element from **Upstream Output**, an expression, or a function as shown below as long as they all evaluate to the appropriate data type.



```
array.create("var1", $flow.name, string.tostring($flow.age))
```

Mapping Complex Arrays - Using the array.forEach() Function

Complex arrays are arrays of objects that can optionally contain nested arrays. Complex arrays are mapped using the `array.forEach()` function. The `array.forEach()` function can be used with or without arguments.

The `array.forEach()` function cannot be used within any other function or expression.

For examples, refer to https://github.com/TIBCOSoftware/tci-flogo/tree/master/samples/app-dev/array.forEach.sample.

When you use `array.forEach()` without any arguments, you define an implicit scope comprising of everything available in the **Upstream Output**. It is equivalent to creating an implicit array with a single object element comprising of everything in the **Upstream Output**. Hence, the resulting length of the array is always one element.

To create a confined scope within the **Upstream Output**, use `array.forEach()` with arguments. You can do so by entering the mapping manually or by selecting the `forEach()` function under the **array** category under **Functions**. The `forEach()` function can accept three arguments. When mapping identical arrays, the `array.forEach()` function gets inserted with the first two arguments by default.

The first argument defines the scope within the **Upstream Output**. Simply put, the input object or array can only be mapped to elements in the **Upstream Output** that fall within the boundary indicated by its scope.

The second argument is a scoping variable given to the scope that you have defined in the first argument. The scoping variable name by default, is the same as the input element name for which you are defining the scope. By doing so, the mapper associates the input object to its scope by the scoping variable. Once there is a scoping variable for the scope, the mapper uses that scoping variable to refer to the scope in future mappings. You can edit the scoping variable to any string that might be more meaningful to you. The scoping variable is particularly useful when mapping the child elements in nested arrays.

The third argument is optional. When iterating through an upstream output array, you can enter a filter to specify a particular condition for mapping as the third argument. When using the filter as the third argument, you must mandatorily enter the scoping variable as the second argument. Only array elements that match the filter get mapped. For instance, if you are iterating through an array, `array1`, in the upstream output with a filter that says `$loop.name=="Jane"` as the third argument, if `array1` has ten elements and only four out of them match the condition of the filter, only those four elements will be mapped to the input array and the remaining six will be skipped. This results in the size of the input array to be only four elements, even though `array1` has ten elements. See the section, Filtering Array Elements to Map Based on a Condition for more details.

If you have used the `array.forEach()` in a legacy app, to update your app with the current changes in the `array.forEach()` function, delete the old mapping and remap the elements. A scoping variable is now included in the mapping. For example, if the old mapping is: `array.forEach($flow.body.Book)`, after the remap, the mapping should be: `array.forEach($flow.body.Book, "Book")` where `"Book"` is the scoping variable.

Understanding `array.ForEach()` Function with an Example

The example in this section illustrates an array, `cakes`.

Consider the example below which is available for you to experiment with at https://github.com/TIBCOSoftware/tci-flogo/tree/master/samples/app-dev/array.forEach.sample.

```
{
   "cakes":[
      {
         "id":"0001",
         "type":"donut",
         "name":"Bundt Cake",
         "ppu":0.55,
         "batters":{
            "batter":[
               {
                  "id":"1001",
                  "type":"Regular"
               },
               {
                  "id":"1002",
                  "type":"Chocolate"
               },
               {
                  "id":"1003",
                  "type":"Blueberry"
               },
               {
                  "id":"1004",
                  "type":"Devil's Food"
               }
            ]
         },
         "topping":[
            {
               "id":"5001",
               "type":"None"
            },
            {
               "id":"5002",
               "type":"Glazed"
            },
            {
               "id":"5005",
               "type":"Sugar"
            },
            {
               "id":"5007",
               "type":"Powdered Sugar"
            },
            {
               "id":"5006",
               "type":"Chocolate with Sprinkles"
            },
            {
               "id":"5003",
               "type":"Chocolate"
            },
            {
               "id":"5004",
               "type":"Maple"
            }
         ]
      },
      {
         "id":"0002",
         "type":"Butter Cake",
         "name":"Raised",
         "ppu":0.55,
         "batters":{
            "batter":[
               {
                  "id":"1001",
                  "type":"Regular"
               }
            ]
         },
         "topping":[
            {
               "id":"5001",
               "type":"None"
            },
            {
               "id":"5002",
               "type":"Glazed"
            },
            {
               "id":"5005",
               "type":"Sugar"
            },
            {
               "id":"5003",
               "type":"Chocolate"
            },
            {
               "id":"5004",
               "type":"Maple"
            }
         ]
      },
      {
         "id":"0003",
         "type":"Biscuit Cake",
         "name":"Old Fashioned",
         "ppu":0.55,
         "batters":{
            "batter":[
               {
                  "id":"1001",
                  "type":"Regular"
               },
               {
                  "id":"1002",
                  "type":"Chocolate"
               }
            ]
         },
         "topping":[
            {
               "id":"5001",
               "type":"None"
            },
            {
               "id":"5002",
               "type":"Glazed"
            },
            {
               "id":"5003",
               "type":"Chocolate"
            },
            {
               "id":"5004",
               "type":"Maple"
            }
         ]
      }
   ]
}
```

In the above example, the `cakes` array has two nested arrays called `topping` and `batter`. You can use `array.forEach()` function to iterate the `cakes` array or you can also iterate its nested arrays, `topping` or `batter` while iterating the `cakes` array. Basically, you can iterate through nested arrays while iterating through the parent array.

The `array.forEach()` function can take three arguments:

1. The first argument is the source array to iterate over.
2. The second argument is the scope, which typically consists of the array you are iterating over and so has the same name as that array.
3. The third optional argument is the condition to use to pull information when looping through the array.

For example, if you want to filter the `cakes` array based on `type` "donut", you would use the following expression:

```
array.forEach($flow.body.cakes,"cakes",$loop.type=="donut")
```

where

- `$ flow.body.cakes` is the source array to iterate over.
- `"cakes"` is the scope name. This is the scope for your mapping. Each scope has a name to it. By default, the scope name is the same as the name of the source array, in this case "cakes".
- `$loop.type=="donut"` is the condition to filter the array elements.

To filter the `cakes` array based on `type` "donut", you would use the following:

```
array.forEach($flow.body.cakes,"cakes",$loop.type=="donut")
```

To filter the `batter` array inside `cakes` array based on its `type` "Regular", use:

```
array.forEach($loop[cakes].batters.batter,"batter",$loop.type=="Regular")
```

Here, `$loop[cakes]` indicates that the `cakes` parent array is being iterated over and during each iteration of the `cakes` array, the `batter` array is also being iterated.

To filter the `topping` array inside `cakes` array based on its `type` "Powdered Sugar", use:

```
 array.forEach($loop[cakes].topping,"topping",$loop.type=="Powdered Sugar")
```

Here, while iterating the `cakes` parent array, we are also iterating over the `topping` array.

Mapping Identical Arrays of Objects

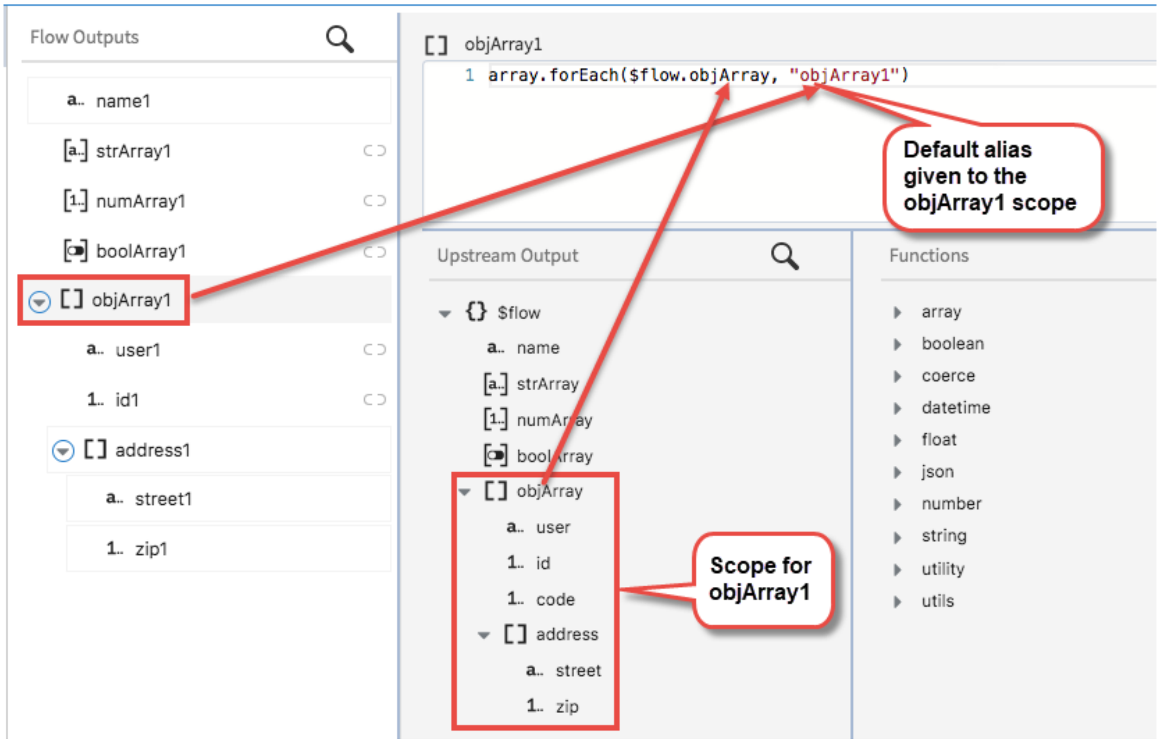
When mapping an array of objects in the input to an identical array of objects (matching property names and data types) in the **Upstream Output**, keep the following in mind:

- Map the array at the root level. The `array.forEach()` function automatically gets inserted with the array scope and a scoping variable for the scope as its arguments. You need not map the array object properties individually if you want all properties to be mapped and if the object property names are identical. The properties get automatically mapped.
- If you do not want all the properties within the object to be mapped or if the names of object properties do not match, you must map the object properties individually too after mapping the root. If you do not do the child mapping individually, the mismatched properties in the objects remain unmapped if the properties are not marked as required (marked with a red asterisk). If such a property is marked as required, then you see a warning.
- The size of the input array is determined by the size of the array in the **Upstream Output** to which you are mapping.

To map identical arrays of objects, follow these steps:

Procedure

1. Click on the input array root (**objArray1** in the example image below).
2. Click the array you want to map to in **Upstream Output** (**objArray** in the image below). The `array.forEach()` function appears in the text box. If the names of all the child elements match, the child elements get mapped automatically. You need not match each child element individually. In this example, none of the child names match, so you would need to do the individual mapping otherwise none of the elements get mapped.



```
array.forEach($flow.objArray, "objArray1")
```

Shown below is the syntax of `array.forEach()` in the image above :

```
array.forEach($flow.objArray, "objArray1")
```

The "**objArray1**" in red font is the scoping variable which constitutes the scope of the current input array. Basically, this means that you can map any element in **objArray1** with an element of same data type in **flow.objArray** in the **Upstream Output**. So, you are defining the scope of **objArray1** to be all the elements within **objArray**.

Mapping Array Child Elements to Non-Array Elements or to an Element in a Non-Matching Array

There may be situations when you want to map an element within an array of objects to an output element that is not in an array or belongs to a non-matching array in the **Upstream Output**. In such a situation, you must create an array with a single element. You do this by using the `array.forEach()` function without any arguments. When you use `array.forEach()` function without arguments, it creates an array with a single object element. The single object element treats everything in the **Upstream Output** as the children of the newly created array object element. This allows you to map to any of the **Upstream Output** elements as they are now treated as if they were within an array.

When using `array.forEach()` without arguments, be sure to map the child elements individually too in addition to setting the array root to `array.forEach()` without arguments. If you do not map the child elements individually, no child elements get mapped. Only elements that you have specifically mapped acquire the mapped values.

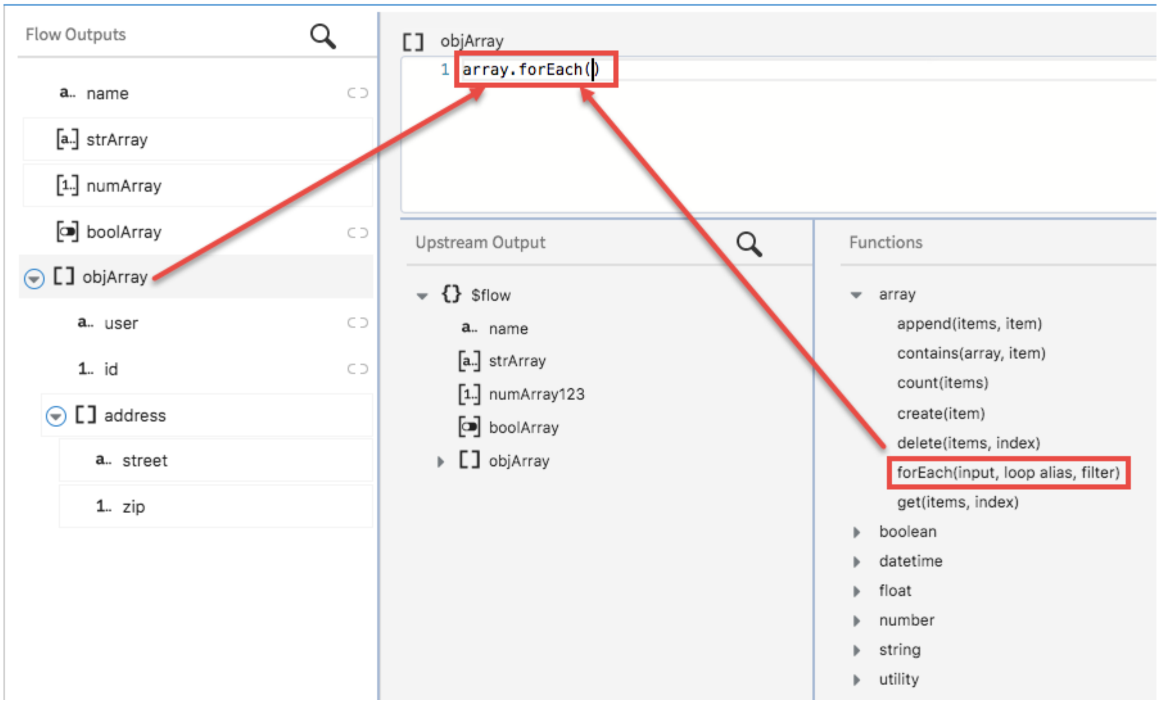
Keep in mind that in this scenario, the resulting length of the array is always one element.

Mapping an array child element to a non-array element is a two step process:

Procedure

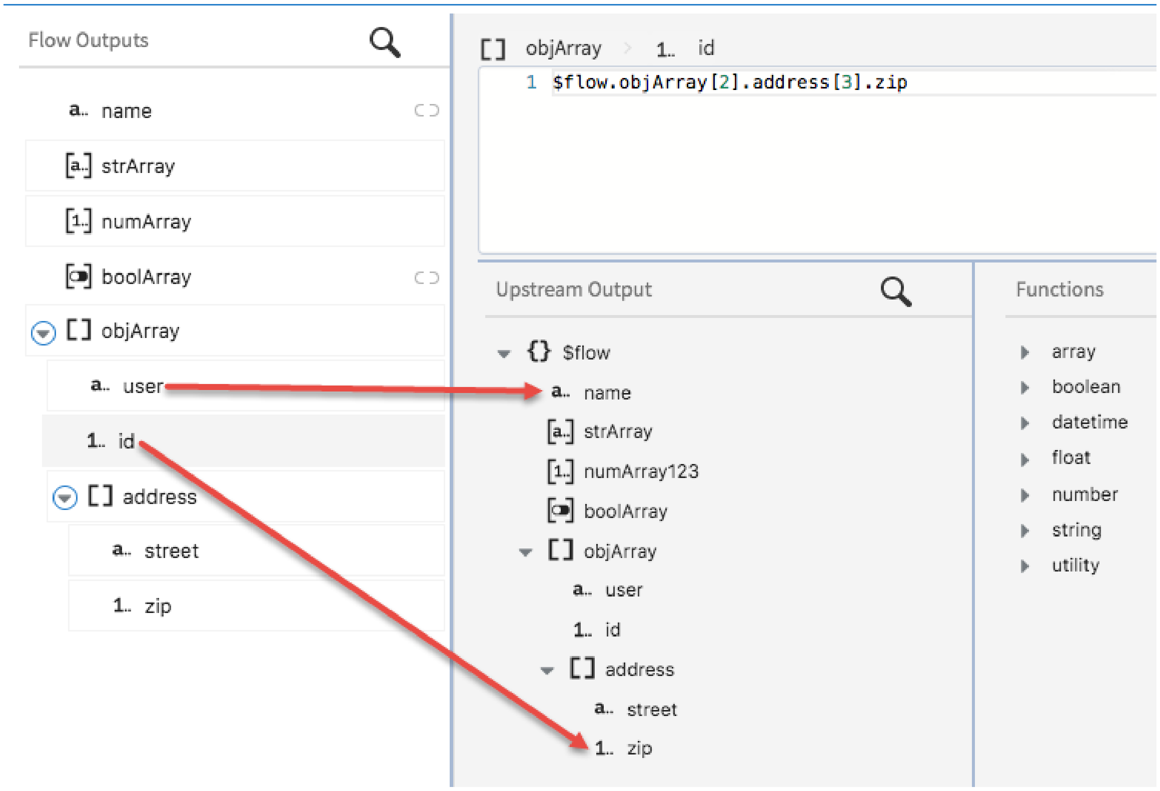
1. Click the input array root (**objArray** in the example below) and either manually enter `array.forEach()` without arguments or select the function from the **array** category under **Functions** and delete the place holder arguments.

   This creates an array of objects with a single element in it. The element contains everything under **Upstream Output**, hence allowing you to map to any element in the **Upstream Output**. The element you are mapping to can be a non-array element or reside within a nested array.

   ```
   1 array.forEach()
   ```

2. Map each element in the input array individually to any element of the same data type under **Upstream Output**.



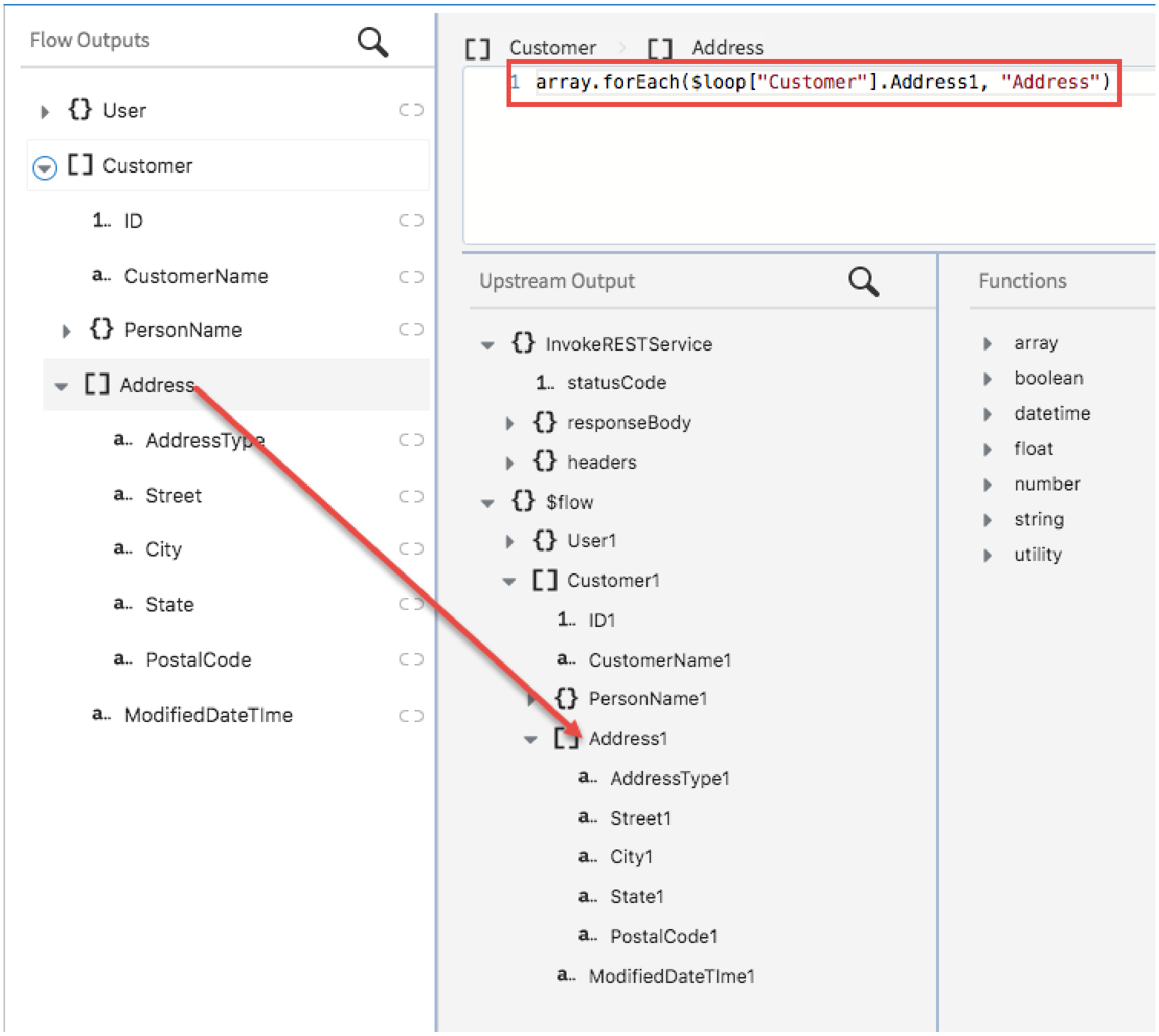
```
$flow.objArray[2].address[3].zip
```

In case you are mapping to an element inside an array, you must provide the index of the array. If you are mapping to an element in a nested array, you must provide the index for both the parent and the nested array as shown above.

Mapping Nested Arrays

Before you map a nested array, you must map its parent root. The scoping variable is particularly useful when mapping the child elements in nested arrays.

The example below is that of a nested array, where **Address** is a nested array whose parent is **Customer**:

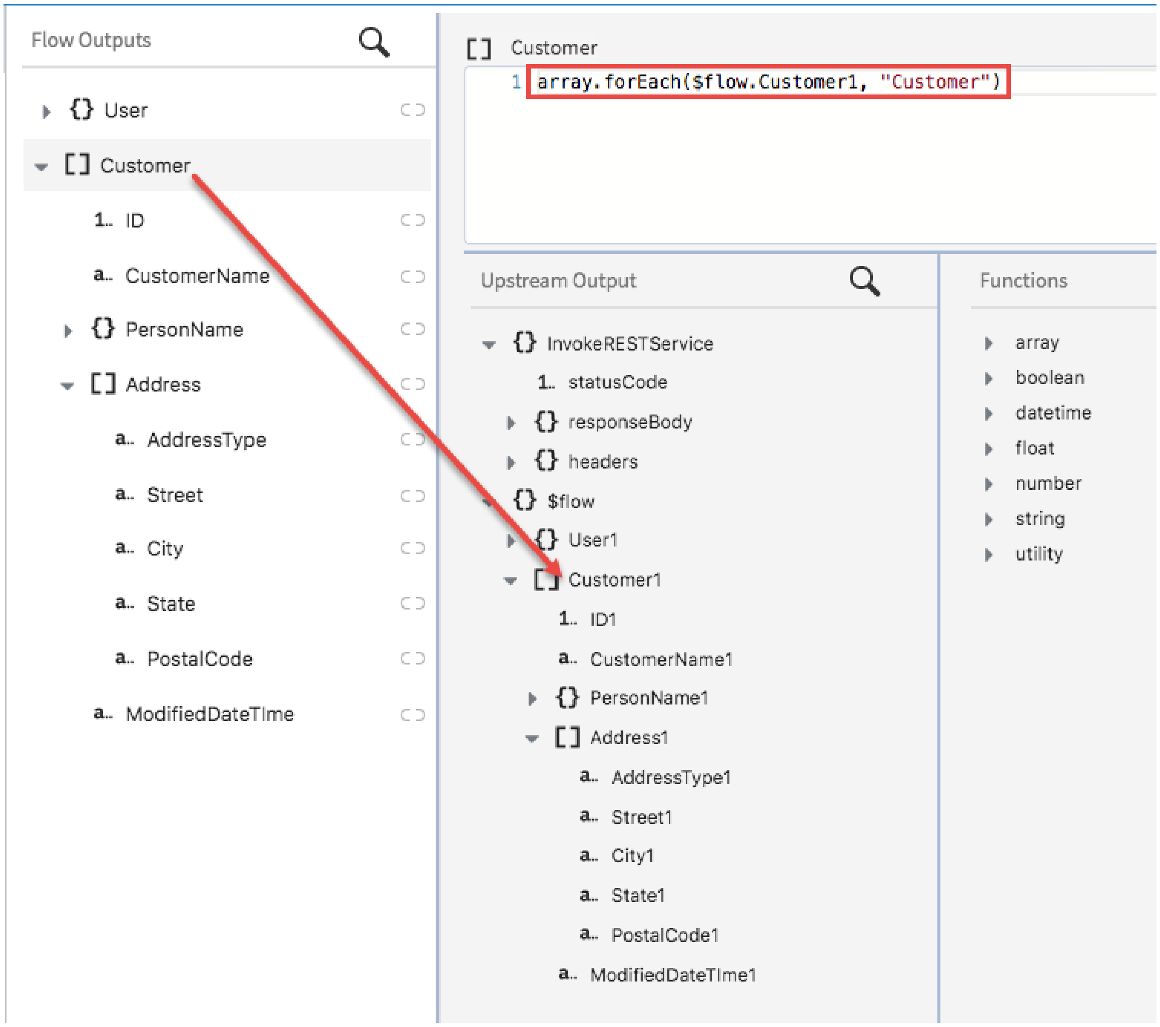


```
array.forEach($loop["Customer"].Address1, "Address")
```

To map **Address**, do the following:

Procedure

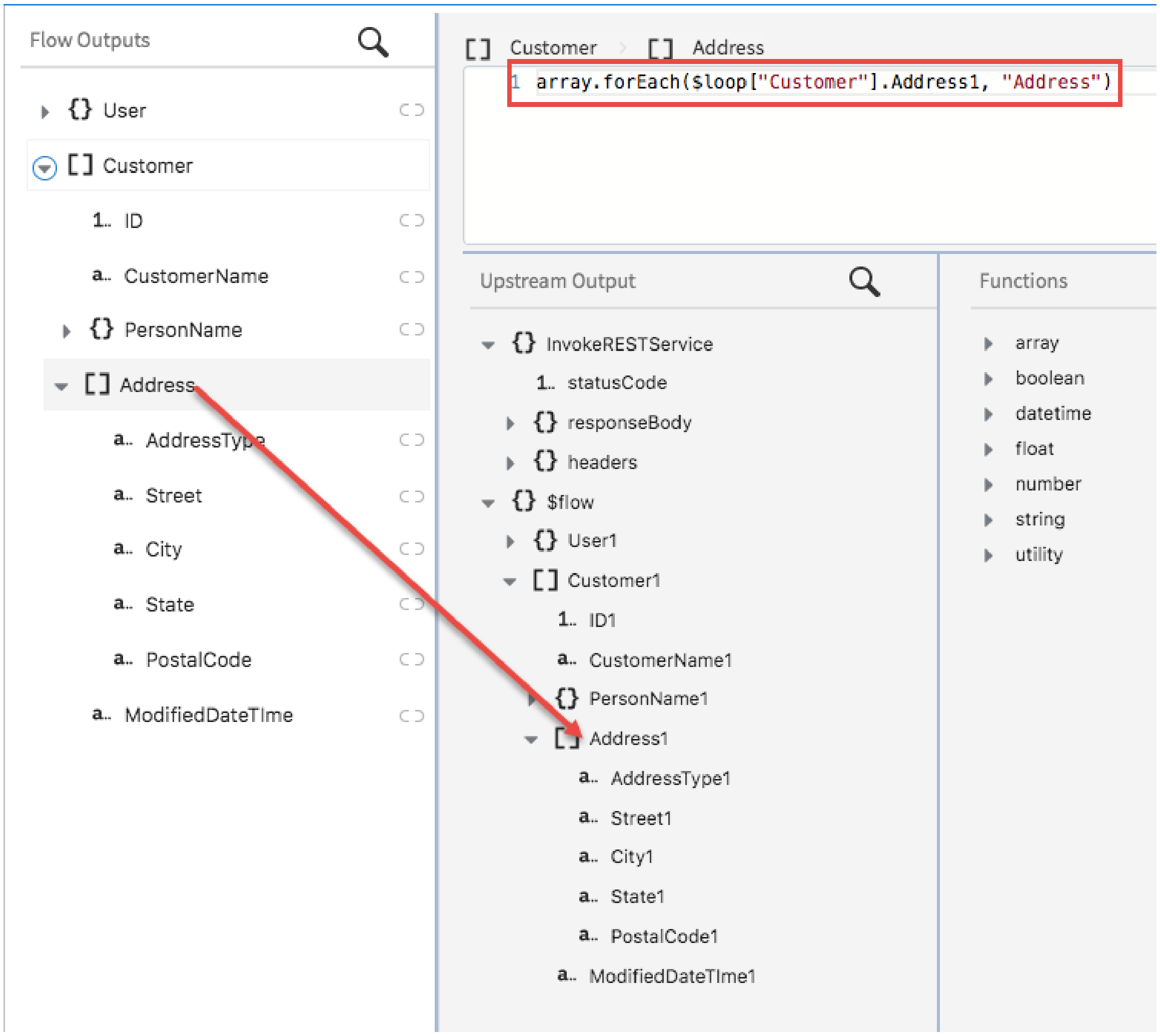
1. Map its parent, **Customer**. When you map **Customer**, you automatically set the scope of **Customer**.



```
array.forEach($flow.Customer1, "Customer")
```

In the image above, **Customer** is mapped to **Customer1**. The first argument, `$flow.Customer1` is the source array (from which **Customer** gets the data) that you are mapping to. This defines the scope (boundary) in the **Upstream Output** within which you can map **Customer**. So, this is the scope of **Customer**. The second argument, "Customer", is the scoping variable given to this scope - the loop here refers to the iteration of **Customer**. By default, the scoping variable has the same name as the loop for which the scope is being defined (in this case **Customer**). You can edit the scoping variable to any string that might be more meaningful to you. This is equivalent to saying that mapping of a child element of **Customer** can happen only to children of **Customer1** in **Upstream Output**.

2. Map **Address**. Now the scope of **Address** gets defined.



```
array.forEach($loop["Customer"].Address1, "Address")
```

Notice that the mapping for **Address**:

- contains the parent scope as well. The parent scope is referred to by its scoping variable, `"Customer"`. Remember that the scope of **Customer** was set when you mapped **Customer** to **Customer1** in the first step above, so we can now simply refer to the parent scope by its scoping variable, `"Customer"`.
- `$loop["Customer"]` refers to the iteration of the **Customer1** array. `$loop` represents the memory address of the **Customer1** (the scope for **Customer**) in **Upstream Output**.
- `$loop["Customer"].Address1` is the scope of **Address**. This scope is denoted by the scoping variable `"Address"`, which is the second variable in this mapping. Since **Address** is a nested array of **Customer**, when you map to **Address** or its child elements, its mapping includes the scope of **Customer** as well.

Mapping Child Elements within a Nested Array Scope

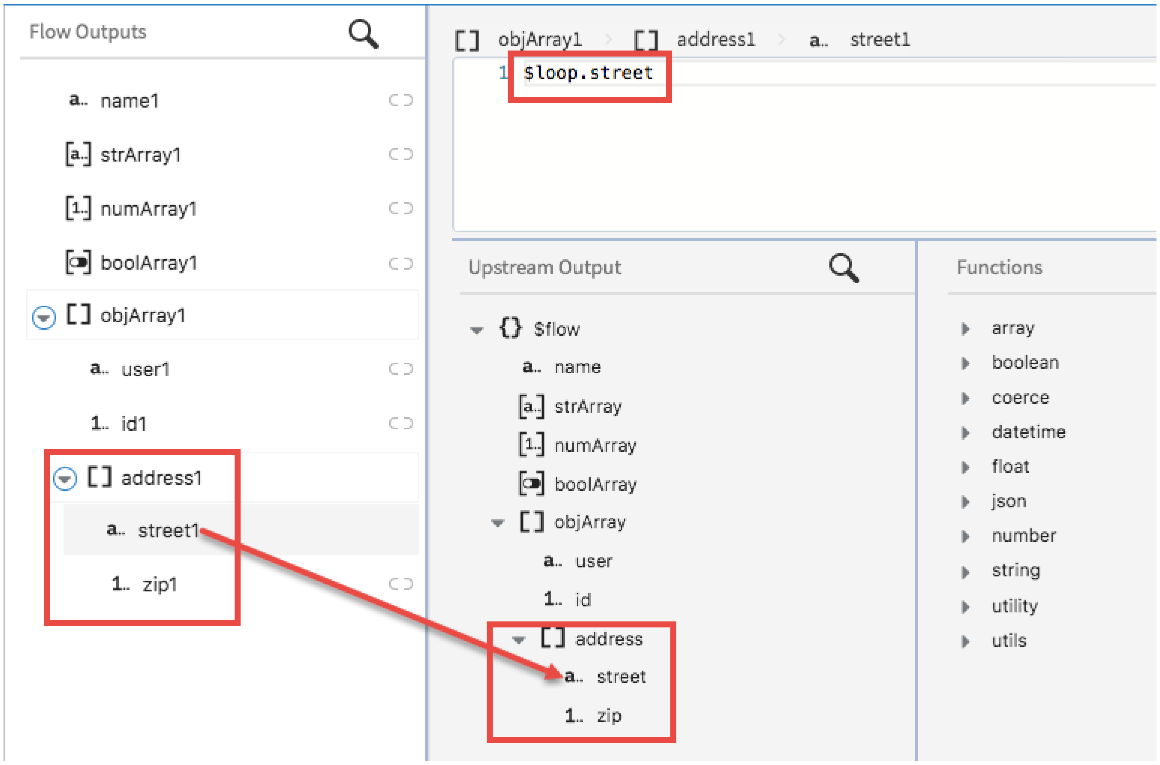
A child element in the input array can be directly mapped to a child element of the same data type within the array scope. Since mapping is done within the nested array scope, you do not need to explicitly state the scoping variable for the nested array scope. The mapping appears as `$loop.`*`<element>`*.

To map a nested array child element, follow these steps:

Procedure

1. Map the parent of the nested array.
2. Map the nested array itself.
3. Map the nested array child elements if the names are not identical or if you do not want to map all elements in the nested array.

   In the following example, since **street** is within the scope of **address1**, **street1** is directly mapped to **street**. `$loop` in the following example implicitly points to **address** which is the scope for **address1** in the input schema.



```
$loop.street
```

Mapping a Nested Array Child Element outside the Nested Array Scope

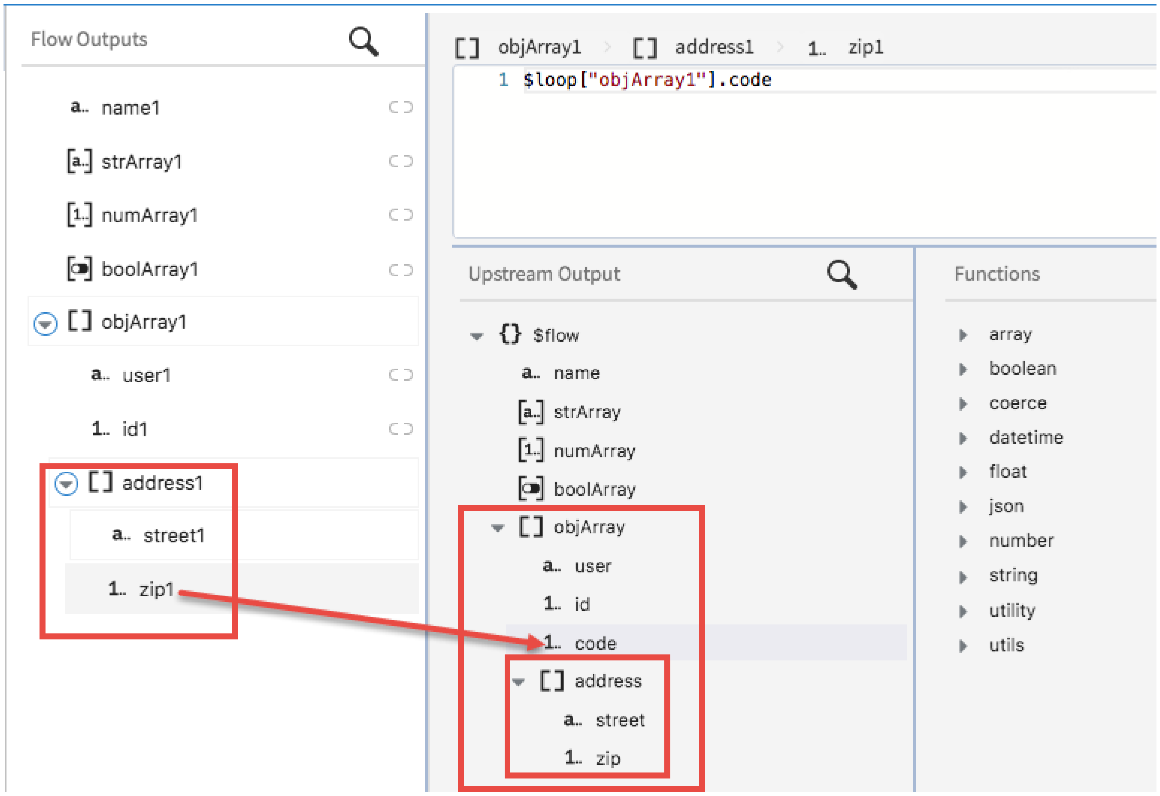
To map a nested array child element outside the nested array scope but within its parent array, you must use the scoping variable of the parent array.

To do so, follow these steps:

Procedure

1. Map the parent array root.
2. Map the nested array root.
3. Map the nested array child element.

   In the example below, `$loop` implicitly points to **address**. In addition, the mapping also explicitly specifies the scope of the parent, `"objArray1"`. This is because **zip1** is mapped to **code** which is outside the scope of **address1**, but within the scope of its parent array, **objArray1**.



```
$loop["objArray1"].code
```

Mapping an Element from a Parent Array to a Child Element in a Nested Array within the Parent

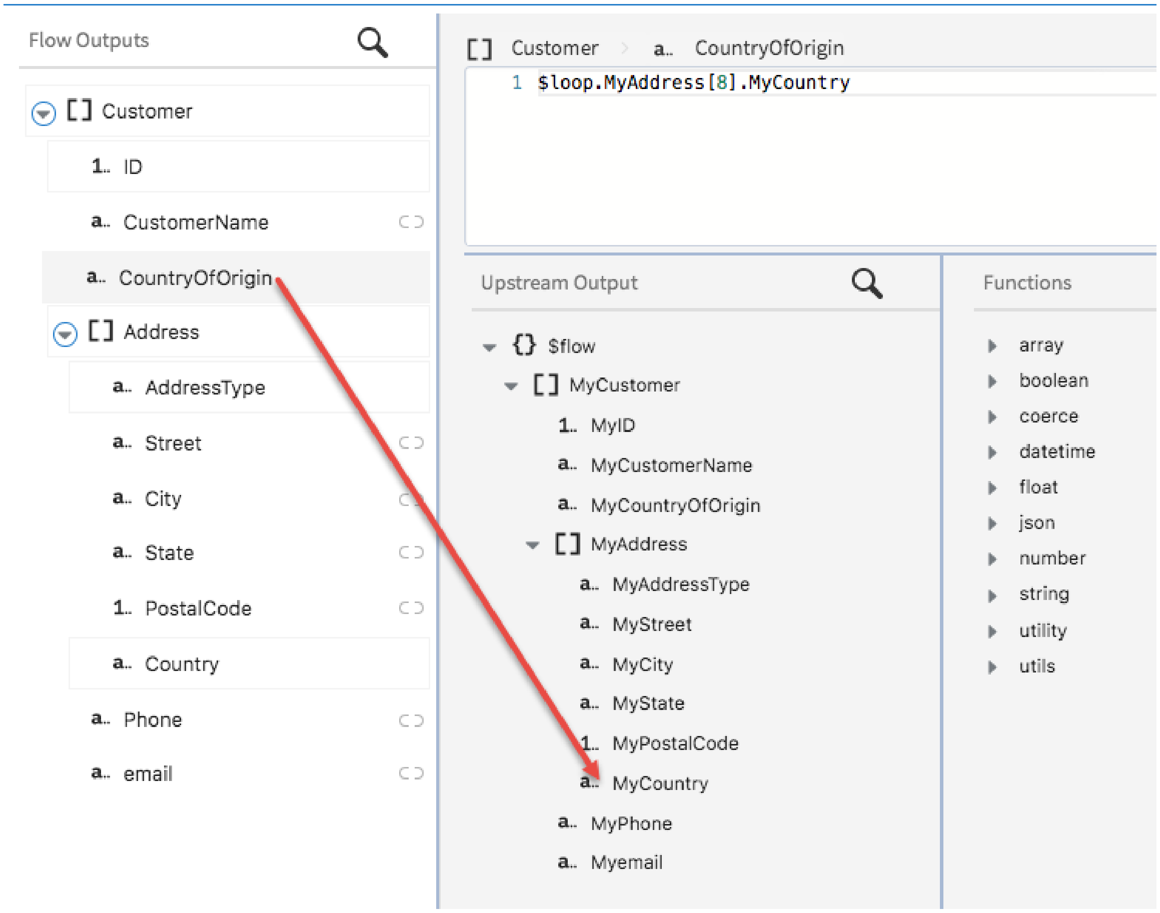
When mapping a primitive data type child element of the parent array to a child element of its nested array, the scope in the mapping is implicitly set to the scope of the parent array. In addition, you must provide the index of the nested array element whose variable you want to map to.

To do so, follow these steps:

Procedure

1. Map the parent array root.
2. Map the nested array root.
3. Map the parent array element.

   In the example below, `$loop` is implicitly set to the scope of **Customer** which is **MyCustomer**. Notice that you must provide the index of the object in the **MyAddress** array whose **MyCountry** element you want to map to.



```
$loop.MyAddress[8].MyCountry
```

Filtering Array Elements to Map Based on a Condition

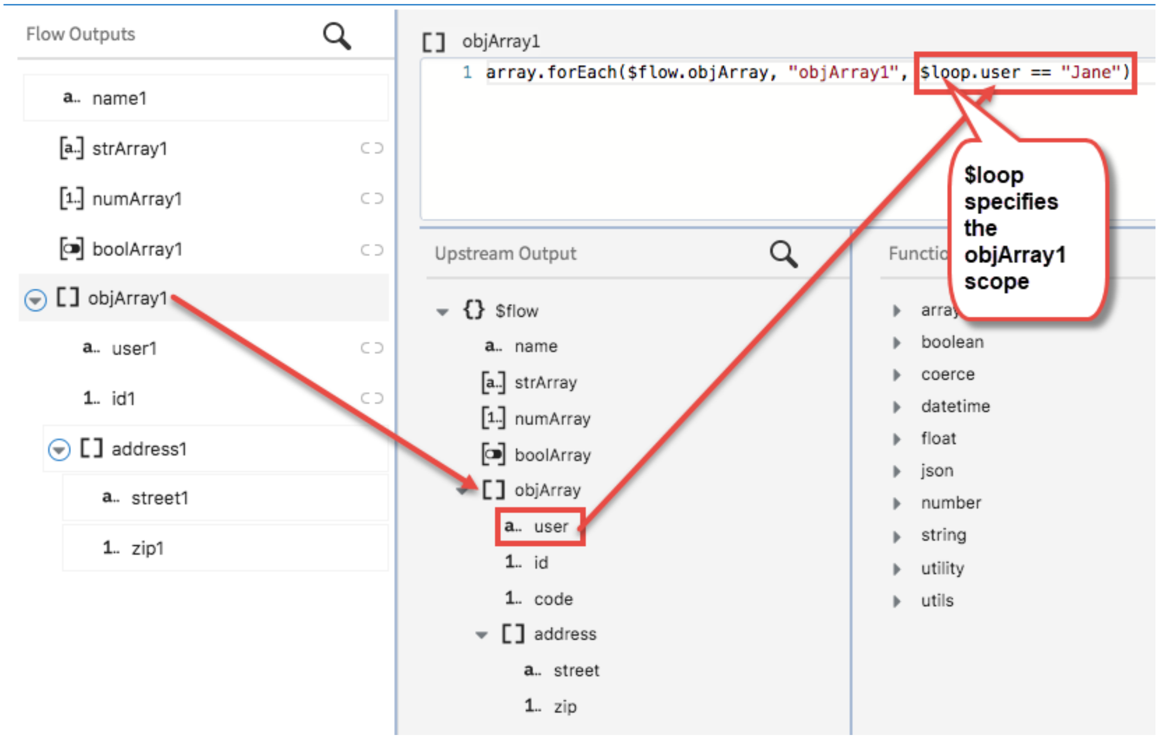
When mapping arrays of objects, you can filter the objects that gets mapped by specifying a filter in the mapper.

You specify this filter as the third argument in the `array.forEach()` function, the first argument being the scope of the element being mapped and the second argument being the scoping variable.

Prerequisites

To specify the filter as the third argument, you must mandatorily specify both the first two arguments in `array.forEach()` - the scope as well as the scoping variable.

Here's an example that contains a filter as the third argument:



```
array.forEach($flow.objArray, "objArray1", $loop.user == "Jane")
```

$loop specifies the objArray1 scope

The above example indicates the following:

- **objArray1** is being mapped to **objArray** in **Upstream Output**
- when iterating through **objArray** in the **Upstream Output**, only the array elements in **objArray** whose child element, **user**, is "Jane" get mapped. If **user** is not equal to "Jane" the iteration for that object is skipped and **objArray1** does not acquire that object.
- **$loop** here specifies the scope of the current loop that is being iterated, in this case **objArray**, whose scope is **objArray1** in **Upstream Output**.

Mapping JSON Data with the `json.path()` Function

Use the `json.path()` function to query an element within JSON data. The JSON data being queried can come from the output of an activity or trigger. In the mapper, you can use the `json.path()` function by itself when providing value to an input parameter or use it within expressions to refer to data within a JSON structure.

This function takes two arguments:

- the search path to the element within the JSON data
- the JSON object that contains the JSON data you are searching

You can specify a filter to be used by the `json.path()` function to narrow down the results returned by the `json.path()` function.

In order to reach the desired node or a specific field in the node in a JSON data, you must follow a specific notation defined in the JsonPath specification. Refer to https://github.com/oliveagle/jsonpath for details on the notation to be used and specific examples of using the notation.

Consider the example below which is available for you to experiment with at https://github.com/TIBCOSoftware/tci-flogo/tree/master/samples/app-dev/json.path.sample.

Examples

The following is an example of how to use the function:

```
json.path("$.store.book[?(@.price > 10)].title", $flow.body)
```

In this example, `$.store.book[?(@.price > 10)].title` is the query path. `[?(@.price > 10)]` is a filter used to narrow down the query results. `$flow.body` is the JSON object against which the query is run (in this case the JSON object comes from the flow input, hence `$flow`). So, this query basically says 'search the books array within `$flow.body` JSON object and return the title of the books whose price is more than $10'.

Consider the following sample JSON data:

```
{
  "store": {
    "book": [
      {
        "category": "reference",
        "author": "Nigel Rees",
        "title": "Sayings of the Century",
        "Availability": [
          {
            "Country": "India",
            "Quantity": 4000,
            "Address": [
              {
                "city": "houston"
              }
            ]
          }
        ],
        "price": 8.95
      },
      {
        "category": "fiction2",
        "author": "Evelyn Waugh",
        "title": "Sword of Honour",
        "Availability": [
          {
            "Country": "USA",
            "Quantity": 5000,
            "Address": [
              {
                "city": "sugarland"
              }
            ]
          }
        ],
        "price": 12.99
      },
      {
        "category": "fiction3",
        "author": "Herman Melville",
        "title": "Moby Dick",
        "isbn": "0-553-21311-3",
        "Availability": [
          {
            "Country": "UK",
            "Quantity": 7000,
            "Address": [
              {
                "city": "stafford"
              }
            ]
          }
        ],
        "price": 8.99
      },
      {
        "category": "fiction4",
        "author": "J. R. R. Tolkien",
        "title": "The Lord of the Rings",
        "isbn": "0-395-19395-8",
        "Availability": [
          {
            "Country": "Australia",
            "Quantity": 2000,
            "Address": [
              {
                "city": "aaaaa"
              }
            ]
          }
        ],
        "price": 22.99
      }
    ],
    "bicycle": {
      "color": "red",
      "price": 19.95
    }
  },
  "expensive": 10
}
```

The following are examples of some JSON query paths that search the JSON data above and return the `category` of the book. In the examples below, the second input parameter for this function, `data` is the name of the file that contains the above JSON code.

- `json.path("$.store.book[?(@.Availability[?(@.Quantity >= 6000)])].category", $flow.data)`

  In the example above, the query scope is the entire `book` array. The filter used to query this array is the condition, `[?(@.Availability[?(@.Quantity >= 6000)])]`. Only the `category` values for the `book` elements that have `Quantity >= 6000` is returned. So, this query returns `fiction3`.

- `json.path("$.store.book[?(@.author == 'Nigel Rees')].category", $flow.data)`

  returns `reference` since it uses the filter `[?(@.author == 'Nigel Rees')]` and the only book authored by Nigel Rees in this array of books has its `category` as reference.

- `json.path("$.store.book[?(@.Availability[?(@.Address[?(@.city == 'sugarland')])])].category", $flow.data)`

  This query is an example of a nested filter where `[?(@.Availability[?(@.Address[?(@.city == 'sugarland')])])]` is the outer filter and the nested filter within it is `[?(@.city == 'sugarland')]`. It returns `reference`.

- `json.path("$.store.book[0].category", $flow.data)`

  This query does not use a filter. It returns `reference`, since your query scope is limited to the `book[0]` element only within the `store` object and your request is to return the value of `categoy`.

Constructing the `any`, `param`, or `object` Data Type in Mapper

When mapping values for data type `any` or `object`, you must manually enter the values in the mapper text box.
Below are some examples of how to construct the data type `any`:

Assigning a literal value to data type `any`

To assign literal values to the `any` data type, you click on the element of type `any`, then simply enter the values you want to assign to it in the mapper text box. For example, to assign the string `Hello!` enter:

```
"Hello!"
```

Assigning an object value to an object or element of data type `any`

Here is an example of how to assign literal values to an object:

```
{
   "Author": "Martin Fowler",
   "ISBN": "0-321-12742-0",
   "Price": "$45"
}
```

where "Author", "ISBN", and "Price" are the object properties. You can use a function instead of a literal value when assigning values for each element. See the "Using a function" section below for details on how to use a function.

Assigning an array value to an object or data type `any`

Here is an example of how to assign an array value to an array of objects or to an element of data type `any`:

```
[
{
   "Author": "Martin Fowler",
   "ISBN": "0-321-12742-0",
   "Price": "$45"
}
]
```

You can use a function instead of a literal value when assigning values for each element. See the "Using a function" section below for details on how to use a function.

Assigning a value from the upstream output

When mapping to an element from the upstream output, the data type of the source element whose value you are assigning determines the data type of the destination element. For example, if you assign the value of an array, then the target element (the element of data type `any`) will be treated as an array, likewise for a string, number, boolean, or object. For example, if you are mapping `$flow.Author` which is an array, then the Author object in the input (destination object) would also be an array. In other words, there will be direct assignment from the source to the destination.

- **Single Element of Primitive Data Type**: To assign the value of a single element of a primitive data type that belongs to the output of the trigger, a preceding activity, or the flow input, you must enter the expression for it. For example to assign the value of `isbn` which comes from the flow input, enter the expression:

  ```
  "=$flow.isbn"
  ```

  where `$flow` is the scope within which `isbn` falls.
- **An object**: When assigning an object, you must create a `mapping` node within the object. The `mapping` node is used to define how the object should be constructed and the various fields within the object mapped. For example, to assign the `bookDetails` object, enter:

  ```
  {
  "mapping": {
  "Author": "=$flow.author",
  "ISBN": "=$flow.name",
  "Price": 20,
  "BestSeller": true
  }
  }
  ```

  You can use a function instead of a literal value when assigning values for each element. See the "Using a function" section below for details on how to use a function.
- **An array of objects**: The following two examples show you how to assign values to arrays:
  - **Building a new array**

To provide values for an array that has a fixed size (where the number of elements are declared), you must provide the values for each array element. For example, if the array has two elements, you must provide the values for each property of the object for both objects. Here is an example of how to do that:

```
{
  "mapping": {
    "books": [
      {
        "author": "=$loop.author",
        "title": "=$loop.title",
        "price": "=$loop.price"
      },
      {
        "author": "Author2",
        "title": "BookTile",
        "price": 19.8
      }
    ]
  }
}
```

In the example above `books` is an array of two elements. The values for each property for both elements are provided.

You can use a function instead of a literal value when assigning values for each element. See the "Using a function" section below for details on how to use a function.

– **Building an Array from an upstream output array**

In the following example, `books` is an array of books coming from the upstream output. To iterate over the array, `$fow.store.books` in upstream output, and assign its values to the input array, you would enter the following in the mapper text box:

```
{
  "mapping": {
    "@foreach($flow.store.books)": {
      "author": "=$loop.author",
      "title": "=$loop.title",
      "price": "=$loop.price"
    }
  }
}
```

The `"@foreach($flow.store.books)"` indicates that you are iterating an array of objects where the `$flow.store.books` is the array. `$flow` is the scope within which `store.books` falls and `$loop` represents the scope for each property within the object. Refer to the following section for details on the `forEach()` function.

- **Using a function**: The following example leverages the output of a REST Invoke activity to get a `pet` from the public `petstore` service. The mapper uses the `string.concat()` function and assigns the function return value to the `description` field in the `data` structure:

```
{
  "mapping": {
    "data.description": "=string.concat(\"The pet category name is:
\",$activity[rest_3].result.category.name)"
  }
}
```

Assigning Values to the `param` Data Type

When you import an app that was originally created in Project Flogo, the app could contain elements that are of data type `param`. The `param` data type is similar to the `object` data type in that it consists of key-value pairs. The difference between an `object` and a `param` is that the `object` can contain values of any data type whereas the values for elements in the `param` data type *must* be of data type `string` only.

Here's an example of assigning values to a `param` data type element:

```
{
"mapping": {
```

```
"Author": "=$flow.author",
"ISBN": "=$flow.name",
"Price": "$20"
    }
}
```

Coercing of Activity Input, Output, and Trigger Reply Fields

In the OSS marked activity input, output, or trigger reply configuration, if you have defined a parameter, but have not defined or cannot define a schema for the parameter, you can coerce the parameter to take the value from a schema that you dynamically define during design time. This feature is particularly useful for apps that were created in Project Flogo and imported to TIBCO Cloud Integration - Flogo (PAYG). Such apps will likely have activities for which input parameters or output are not defined with a schema. Currently, coercion of parameters is supported only for the following data types:

- array
- object
- param
- any

After you enter the schema, it is displayed in a tree format under **Activity Input**, **Output** tab, or **Trigger Reply** in the mapper. All subsequent activities will also display the elements of the schema under the activity in the Upstream Output. The schema elements will now be available for you to map.

Note the following:

- Coercing is supported only in the **Default** category activities which are the activities marked as OSS, except for the **Return** and **Start a SubFlow** activities. These two activities display flow-level data. The flow-level inputs and outputs can be entered or modified only in the **Flow Inputs & Outputs** accordion tab, hence they cannot be coerced from within the **Input** tab of the activity itself.
- Currently, coercion is supported only for top-level parameters. Nested coercion (for example, an object within an object) is not supported.
- Currently, coercing a schema for trigger input is not supported. The coercing option is not available in the **Map to Flow Inputs** tab in the trigger configuration. This is because the parameters you see in this tab are flow input parameters and are not related to the trigger. You have the option to coerce these parameters in the **Input** tab of the **Flow Inputs & Outputs** accordion tab.
- After you have mapped a child element within a parameter, if you change the name of the parent or the child, your mapping will be lost. However, if you change the data type of the element, the mapping is preserved, but you see an error related to the mismatch in data type.
- The schema you enter is preserved when you export and import the app.
- If you edit the schema at a later time, as long as you click the **Apply** button after editing, your edits will display in the mapper. You must then click **Save** in the mapper to persist your schema changes.
- You cannot coerce a parameter or edit its schema in any activity appearing in a subflow. For example, if the **OracleDatabaseQuery** activity appears in both the main flow and the subflow, you cannot edit the schema of any of its parameters in the subflow. But you can edit the schema of the **OracleDatabaseQuery** activity in the main flow. This is because the subflow activity input and output schemas are inherited from the main flow. There is a possibility that the same subflow could be used in multiple main flows, so if you edit an activity in the subflow it could break another main flow that uses the subflow.

To provide the schema for coercion, do the following:

Procedure

1. On the flow details page, click on the activity or trigger as the case may be, to open its configuration.
2. Click any of the following that what you want to configure:

   - **Input** tab to configure a parameter in the activity input
   - **Output** tab to configure the schema for the activity output
   - **Map from Flow Outputs** tab if configuring the trigger reply
3. To configure a schema

   - for a parameter in activity input, hover your mouse cursor over the parameter name for which you want to configure the schema under **Activity Input**.
   - for the activity output, hover your mouse cursor over the parameter name for which you want to configure the schema.
   - for a parameter in the trigger reply, hover your mouse cursor over the parameter name.

   The **Coerce with schema** icon (•••) that appears next to it.
4. Click the **Coerce with schema** icon.

   > The **Coerce with schema** icon appears against the parameter name for only those parameters that do not have a schema defined in the **Input Settings** tab (or a schema cannot be defined because the activity does not have an **Input Settings** tab, for example, the OSS-marked activities) *and* whose data type is one of the following: array, param, object, or any.
5. Enter the schema for the parameter or activity output and click **Apply**. The mapper validates that the data type of the schema you entered matches the data type of the parameter being coerced. If the data types do not match, the **Apply** button remains disabled and you see an error.

   - For activity input and trigger reply, the schema you enter displays in a tree format under the parameter name in the mapper.
   - For the activity output, the schema is displayed in a tree format in the **Output** tab of the activity. Upstream Output displays the output of the preceding activities.
6. Click **Save** to persist the schema into the database or **Discard** to discard the schema.
   Now you can map the child elements within the parameter. In the case of the activity **Output** tab, the output tree does not display in the current activity, but will display in the mapper for subsequent activities only. Once persisted in the database, these schema trees get displayed in the Upstream Output area of the mapper for subsequent activities. This allows you to map to them in subsequent activities.

Clear Mapping of Child Elements in Objects and Arrays

After mapping an array or an object, you can clear the mapping of all the child elements within that array or object with one click. The mapping is cleared at the root level and mapping for everything under that root gets cleared, even the nested arrays and objects, should there be any. To clear mapping for individual elements in an array or object selectively, click on that element and delete the mapping for it.
To clear the mappings for all child elements of an array or object, do the following:

Procedure

1. In the mapper, click on the array or object root element to select it.
2. Hover your mouse cursor to the right end of the root name until the **Clear Mapping** icon ( ) appears.
3. Click the **Clear Mapping** icon.

Ignoring Missing Object Properties when Mapping Objects

There may be instances when you map objects where one or more object properties might be missing in the source or target object. The mapper can be set to ignore such cases.
If you want the mapper to ignore such cases, you must set the `FLOGO_MAPPING_SKIP_MISSING` engine variable to `true`. The mapper will ignore the missing mapping as long as the element is optional (not marked as mandatory with a red asterisk against it). Elements marked as mandatory must be mapped. For more details, see the section on Environment Variables.

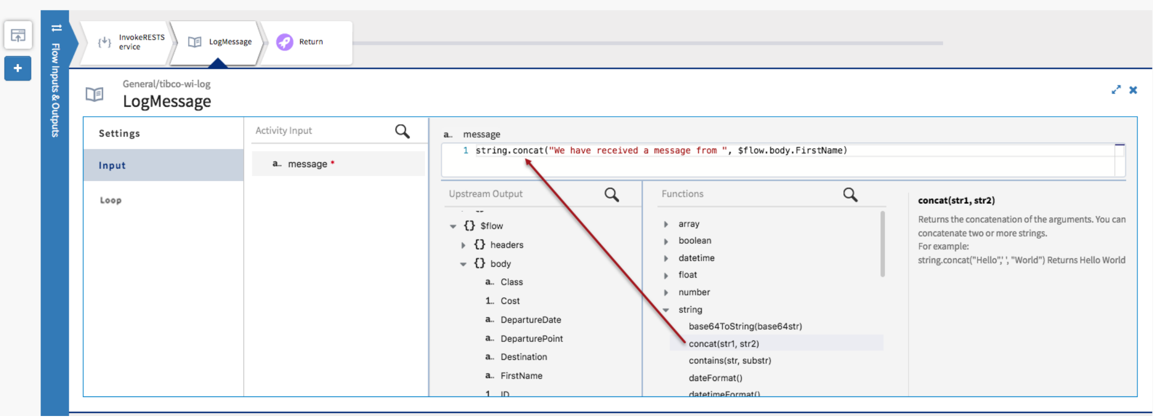
Using Functions

You can use a function from the list of functions available under **Functions** in the mapper. Input parameters to the function can either be mapped from an element under **Upstream Output**, a literal value, or an expression that evaluates to the appropriate data type or any combination of them.

The procedure below illustrates an example that concatenates two strings and assigns the concatenated value to **message**. We manually enter a value for the first string (**str1**) and map the second string to **FirstName** under **body**. The value for **FirstName** comes from the flow input.

Procedure

1. Click **message** to open the mapper to the right.
2. Expand the **string** function group in the right column and click **concat(str1, str2)**.

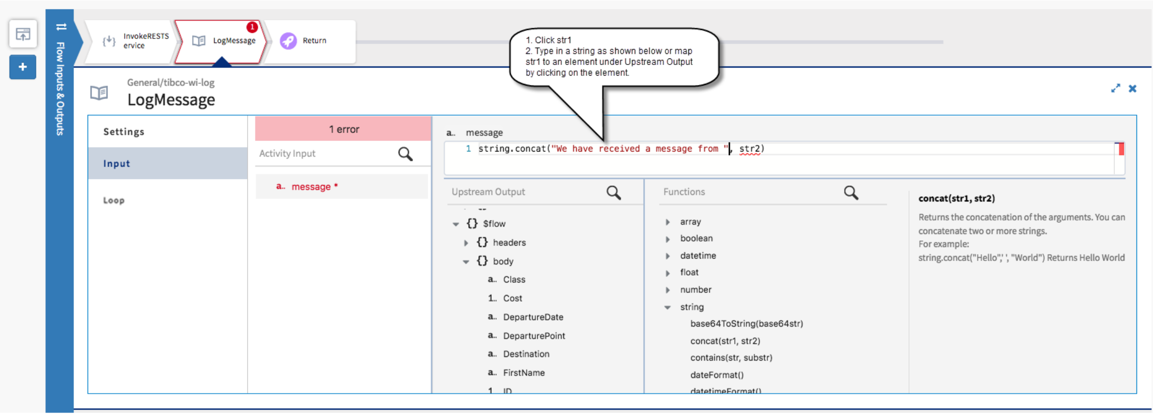


```
string.concat("We have received a message from ", $flow.body.FirstName)
```

concat(str1, str2)

Returns the concatenation of the arguments. You can concatenate two or more strings.
For example:
string.concat("Hello",' ', "World") Returns Hello World

3. Select **str1** in the function and type `"We have received a message from "` (be sure to include the double quotes as shown below) to replace **str1** with it.



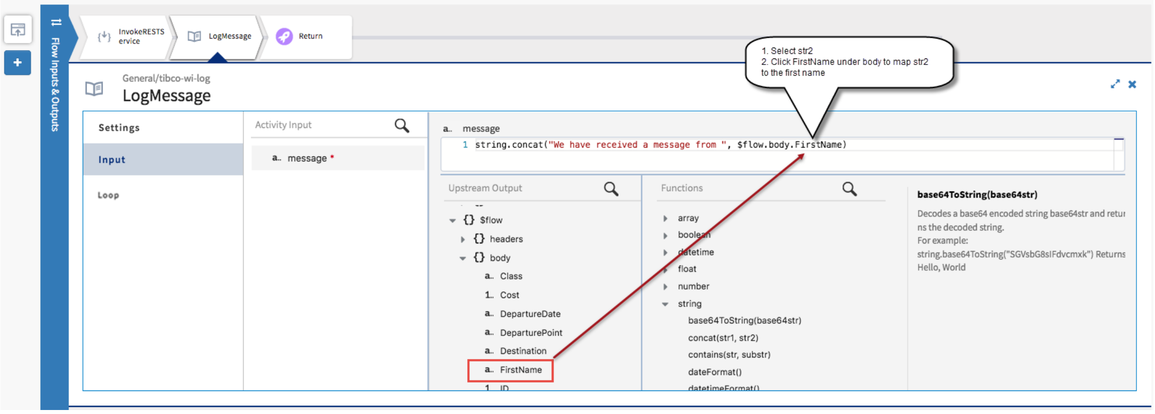
1. Click str1
2. Type in a string as shown below or map str1 to an element under Upstream Output by clicking on the element.

```
string.concat("We have received a message from ", str2)
```

concat(str1, str2)

Returns the concatenation of the arguments. You can concatenate two or more strings.
For example:
string.concat("Hello",' ', "World") Returns Hello World

4. Select **str2** and click **FirstName** under **body**.



1. Select str2
2. Click FirstName under body to map str2 to the first name

```
string.concat("We have received a message from ", $flow.body.FirstName)
```

base64ToString(base64str)

Decodes a base64 encoded string base64str and returns the decoded string.
For example:
string.base64ToString("SGVsbG8sIFdvcmxk") Returns Hello, World

At runtime, the output from the `concat` function gets mapped to **message**.

Using the `array.forEach()` Function

Refer to the section, Mapping Complex Arrays, for a detailed explanation on using the `array.forEach()` function.

Using the json.path() Function

Refer to the section, Mapping JSON Data with the json.path() Function , for a detailed explanation on using the `json.path()` function.

Using Expressions

You can use two categories of data mapping expressions in TIBCO Cloud Integration - Flogo (PAYG).

Basic Expression

Basic expressions can be written using any combination of the following by using operators:

- literal values
- functions
- previous activity or trigger output

Refer to Supported Operators for details on the operators that can be used within a basic expression.

Here are some examples of basic expressions:

```
string.concat("Rest Invoke response status
code:",$activity[InvokeRESTService].statusCode)
```

```
string.length($activity[InvokeRESTService].responseBody.data) >=7
```

```
$activity[InvokeRESTService].statusCode == 200 &&
$activity[InvokeRESTService].responseBody.data == "Success"
```

Ternary Expression

Ternary expressions are assembled as follows: expression ? boolean true : boolean false

Here is an example of basic ternary expression:

```
$activity[InvokeRESTService].statusCode == 200 ? "Response successfully":"Response
failed, status code not 200"
```

In the above example `$activity[InvokeRESTService].statusCode == 200` is the expression to be evaluated. If the expression evaluates to true (meaning statusCode equals 200), it returns `Response successfully`. If the expression evaluates to false (meaning statusCode does not equal 200), it returns "`Response failed, status code not 200`".

Here is an example of a nested ternary expression:

```
$activity[InvokeRESTService].statusCode == 200 ?
$activity[InvokeRESTService].responseBody.data == "Success" ? "Response with correct
data" : "Status ok but data unexpected" : "Response failed, status code not 200"
```

The example above checks first to see if `statusCode` equals 200. If the `statusCode` does not equal 200, it outputs `Response failed, status code not 200`. If the `statusCode` equals 200, only then it checks to see if the `responseBody.data` is equal to "Success". If the `responseBody.data` is equal to "Success", it outputs `Response with correct data`. If the `responseBody.data` is not equal to "Success", it outputs `Status ok but data unexpected`.

Supported Operators

TIBCO Cloud Integration - Flogo (PAYG) supports the operators that are listed below.

- ==
- ||
- &&
- !=
- >
- <
- >=
- <=
- +
- -
- /
- %
- Ternary operators - nested ternary operators are supported. For example, `$activity[InvokeRESTService].statusCode==200?($activity[InvokeRESTService].statusCode==200?true:false):false`

Developing APIs

TIBCO Cloud Integration - Flogo (PAYG) lets you take an API-first development approach to implementing APIs from a Swagger Specification 2.0, OpenAPI Specification 3.0, or GraphQL schema. After you upload an API specification file or a GraphQL schema, TIBCO Cloud Integration - Flogo (PAYG) validates the file and if the validation passes, it automatically creates the Flogo flows and triggers for you.

Using an OpenAPI Specification

TIBCO Cloud Integration - Flogo (PAYG) gives you an option to create the Flogo app logic (flows) by importing an API specification file. You can simply drag a specification file to the TIBCO Cloud Integration - Flogo (PAYG) UI or navigate to it. If you have an existing specification file stored in the TIBCO Cloud™ Integration - API Modeler, select it when creating the flow. The flows for your app are automatically created based on the definitions in the specification file that you uploaded.

When you create an app from a specification, the **ConfigureHTTPResponse** and **Return** activities are automatically added in the flow. The mappings from trigger output to flow inputs get configured for you based on the definitions in the specification. The output of the **ConfigureHTTPResponse** activity is automatically mapped to the **Return** activity input. However, you must configure the input to the

ConfigureHTTPResponse activity manually. If you have multiple response codes configured in the REST trigger, the first response code is configured in the **ConfigureHTTPResponse** activity by default. The only exception to this is if you have a response code of 200 configured. In that case, the 200 response code is configured in the **ConfigureHTTPResponse** activity by default.

Before the TIBCO Flogo® App is created, a validation process ensures that the features defined in the specification are supported in TIBCO Cloud Integration - Flogo (PAYG).

Considerations when using an API specification file to create a Flogo App:

- TIBCO Cloud Integration - Flogo (PAYG) supports Swagger Specification 2.0 and OpenAPI Specification 3.0.
- Currently, TIBCO Cloud Integration - Flogo (PAYG) supports only the JSON format.
- Cyclic dependency is not supported when creating flows from specifications. For example, if you have a type Book that contains an object element of type, Author. The type Author in turn contains an element of type Book which represents the books written by the author. To retrieve the Author, it creates a cyclic dependency where the Author object contains the Book object and the Book type in turn contains the Author object.
- Not all data types are supported in TIBCO Cloud Integration - Flogo (PAYG). A data type that appears in your specification but is not supported in TIBCO Cloud Integration - Flogo (PAYG) will result in an error being displayed.
- Schema references within schemas are not supported.
- If the specification has a response code other than 200 (OK) or 500 (Error), the method that contains the unsupported response code are not created.
- You can enter a schema for the response code 200, but the 500 response code must be a string.
- Basepath element in the schema is not supported.

If you get a validation error, you can either cancel the process of generating the app or click **Continue**. If you opt to continue, TIBCO Cloud Integration - Flogo (PAYG) continues with the app creation and ignores the parts of the specification that did not pass the validation.

The REST reply data type is by default set to `any` data type. To configure the reply to an explicit data type, see Configuring the REST Reply section.

To create an app using an API specification, follow these steps:

To upload your specification file, follow these steps:

Procedure

1. Open the app details page and click **Create**.
   The **Add triggers and flows** dialog opens.
2. Click **Swagger Specification** under **Start with**.
3. You have two options:
   - Create a flow using an API specification that exists in TIBCO Cloud™ Integration-API Modeler. In the **API Specs** tab, select the specification that you want to use.
   - Use an API specification saved locally on your computer by uploading it to TIBCO Cloud Integration - Flogo (PAYG). Click **Upload file** to open the tab. Browse to the saved API specification on your local machine or drag and drop your saved API specification to the **Add triggers and flows** dialog.

   TIBCO Cloud Integration - Flogo (PAYG) validates your file extension. If your file extension is `.json`, you see a green check mark and the **Upload** button appears.

   TIBCO Cloud Integration - Flogo (PAYG) validates the contents of your file and if it passes the validation, it creates the flows based on the definitions in the file. One flow gets created for each method and path combination defined in the file. If there are errors in your file, you get warning messages saying so, but you have the option to continue with creating the flows. If you click **Continue**, the flows get created for supported methods only. Other issues must be fixed before you can upload the file again.

Currently, the following are not supported:

- PATCH method
- Form data content type
- Same root having a static path and a parameterised path in the file, for example, `/foo/bar` and `/foo/{id}`. But having two static paths are supported, for example, `/foo/bar` and `/foo/bar1`

The Flogo App gets created and the flow details page opens with the flows created. Each flow corresponds to a method defined in the specification that passed the validation. These flows are automatically created based on the specification that you used to create the app. The flows are created with a REST trigger (**ReceiveHTTPMessage**) and have a REST **Return** activity appended to them by default.

4. In each flow, do the following:

a) Open the flow by clicking on its name.

b) Click the trigger to open its configuration dialog.

c) Map the following:

- In the **Map to Flow Inputs** tab, map the **Trigger Output** to **Flow Input**.
- In the **Map from Flow Outputs** tab, map the **Flow Output** to **Trigger Reply**.

To test the deployed app, follow the procedure in Testing the Deployed App section.

You can also download the specification used to create the app by following the procedure in Downloading the API Specification Used section.

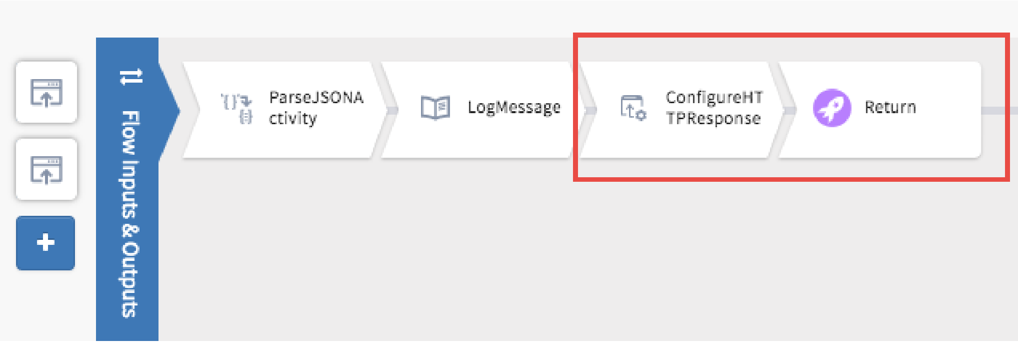
You also have the option to copy the Endpoint URL from the **Endpoints** tab by clicking **Copy spec URL**.

Or you can click the (⧉) icon next to the Endpoint URL itself.

Configuring the REST Reply

When creating a REST app from a Swagger 2.0 or OpenAPI 3.0 API specification, the **ReceiveHTTPMessage** reply data type is set to `any` by default. You must explicitly configure the reply type.

To explicitly configure the reply type, add a **ConfigureHTTPResponse** activity in the flow. This activity must immediately precede the **Return** activity in the flow.



You can configure custom codes that you want to use in the HTTP reply in the **Reply Settings** tab of the **ReceiveHTTPMessage** trigger.

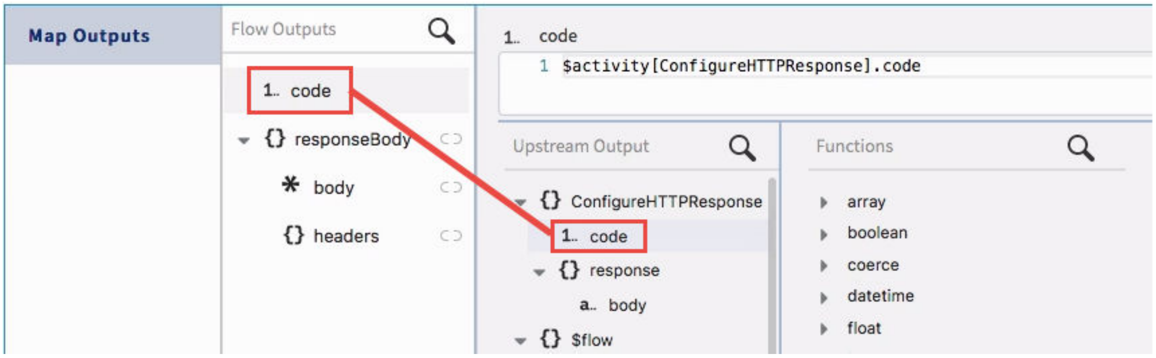
Follow these steps to configure your HTTP reply:

Procedure

1. Open the REST trigger configuration pane by clicking on it.
2. In the **Reply Settings** tab of the **ReceiveHTTPMessage** REST trigger, configure the custom codes that you want to use. Refer to the section, "REST Trigger" in the *Activities and Triggers Guide*.
3. Add a **ConfigureHTTPResponse** activity immediately preceding the **Return** activity in the flow.
4. Open the **ConfigureHTTPResponse** activity by clicking on it and configure it as follows:
   a) In the **Settings** tab:
      1. If your flow is attached to multiple REST triggers, select the trigger in which you have configured the code you want to use from the **Trigger Name** drop-down menu. The **Trigger Name** field does not display if your flow is attached to only one REST trigger.
      2. Select a response code from the **Code** field menu. Only the codes configured in the selected trigger are displayed in the menu.

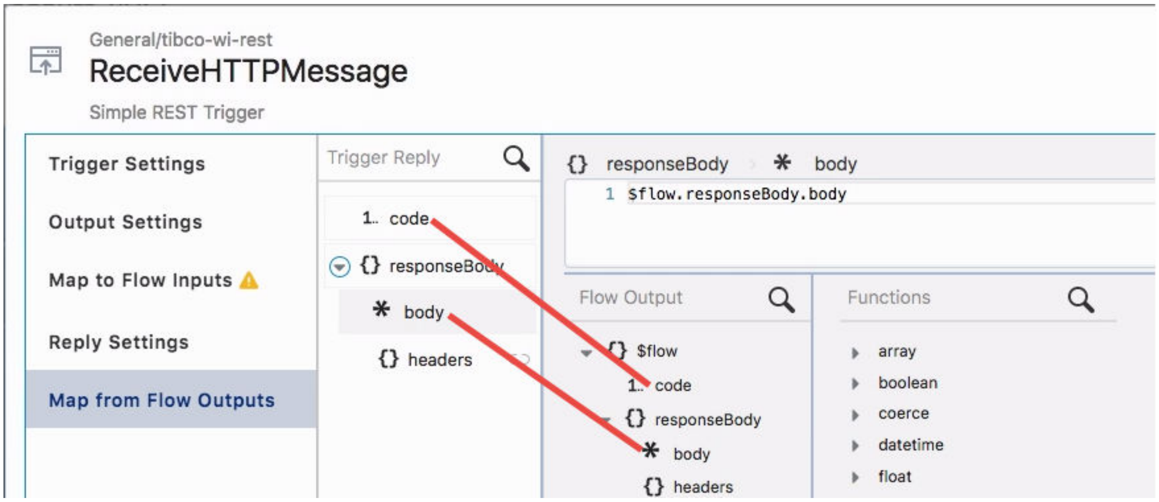
   b) The **Input** tab displays the schema for the response code. Map the elements or manually enter a value for the elements.

   c) Click **Save**.
5. Configure the **Return** activity by mapping the **code** and **body** (which is currently of data type `any`).



```
$activity[ConfigureHTTPResponse].code
```

6. Click **Save**.
7. In the **Map from Flow Outputs** tab in the **ReceiveHTTPMessage** trigger, map the **code** and **body** to the corresponding elements from the flow output.



```
$flow.responseBody.body
```

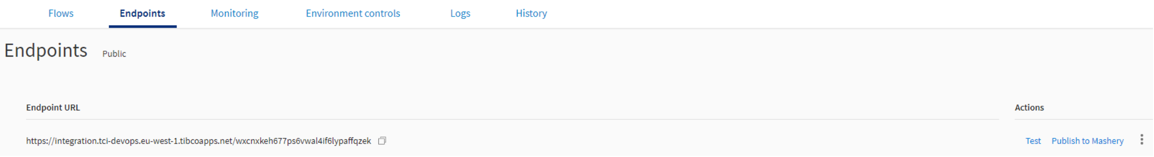
8. Click **Save**.

Testing the Deployed App

The deployed app can be tested using its own API specification.
To test the deployed app, do the following:

Procedure

1. Open the app.
2. Click **Endpoints** to open its tab.



Endpoints Public

Endpoint URL	Actions
https://integration.tci-devops.eu-west-1.tibcoapps.net/wxcnxkeh677ps6vwal4if6lypaffqzek	Test Publish to Mashery ⋮

3. Click **Test**.

Downloading the API Specification Used

You can download the API specification used to create the app.
To download the specification, follow these steps:

Procedure

1. Open the app.
2. Click **Endpoints** to open its tab.
3. Click the shortcut menu (⋮) to the extreme right of the Endpoint URL.
4. Click **Download spec**.
   The downloaded specification may not be exactly the same as the original specification that was used to create the app. This could happen because TIBCO Cloud Integration - Flogo (PAYG) follows its own convention when generating a specification from its apps. Also, any changes that you might have made after creating the app, will be reflected in the downloaded specification, but will not change the original specification from which you created the app. The original specification will remain untouched. Use the downloaded specification only for testing the app.

Using GraphQL Schema

GraphQL provides a powerful query language for your APIs enabling clients to get the exact data that they need. It has the ability to get data from multiple resources in a single request by aggregating the requested data to form one result set. GraphQL provides a single endpoint for accessing data in terms of types and fields.

TIBCO Cloud Integration - Flogo (PAYG) provides an out-of-the-box GraphQL trigger which turns your Flogo app into a GraphQL server implementation. Each Flogo flow in the app acts like a GraphQL field resolver. So, the output of the flow must match the return type of the field in the schema.

TIBCO Cloud Integration - Flogo (PAYG) allows you to create GraphQL triggers by dragging and dropping your GraphQL schema file into the Web UI or by navigating to the file. A flow gets automatically created for every query and mutation type in your schema. You must then open the flow and define what kind of data you want the flow to return. This saves you the time and effort to programmatically define data structures on the server.

This section assumes that you are familiar with GraphQL. To learn about GraphQL, refer to the GraphQL documentation.

GraphQL server implementation in TIBCO Cloud Integration - Flogo (PAYG)

To obtain samples of GraphQL schemas and app JSON files, go to https://github.com/project-flogo/graphql.

To use GraphQL in TIBCO Cloud Integration - Flogo (PAYG), you must create a GraphQL trigger. Use one of the methods below to create a GraphQL trigger.

The implementation of GraphQL server in TIBCO Cloud Integration - Flogo (PAYG) currently does not return the specified field ordering in a query when a request is received. It does not affect the correctness of the response returned, but affects the readability and is non-compliant to current specification.

The GraphQL schema must have either `.gql` or `.graphql` extension.

For details on the GraphQL trigger refer to the "GraphQL Trigger" section in the *TIBCO Flogo® Activities and Triggers Guide*.

Creating a new GraphQL trigger

To create a new GraphQL trigger, follow these steps:

1. Open the app details page.
2. Click **Create**. The **Add triggers and flows** dialog opens.
3. Under **Create new**, select **Trigger**.
4. Select the **GraphQL Trigger** card.
5. Click **Browse** and navigate to your locally stored GraphQL schema file to upload it.
6. Click **Create**. The new GraphQL trigger gets created with a placeholder for a flow attached to it.

Once the trigger is created from the wizard, the trigger configuration is fixed and the **Operation Field** and **Resolver For** cannot be changed.

For more information, see the "GraphQL Trigger" section in the *TIBCO Flogo® Activities and Triggers Guide*.

To implement a single method in your `.gql` file

To implement a single method, follow these steps:

1. In TIBCO Cloud Integration - Flogo (PAYG), open the app details page and click **Create**.
2. In the **Add triggers and flows** dialog, click **Flow** under the **Create new**.
3. Enter a name for the flow in the **Name** text box. Optionally, enter a description for the flow in the **Description** text box.
4. Click **Create**.
5. Select **Start with a trigger**.
6. In the **Triggers catalog**, select the **GraphQL Trigger** card.
7. Follow the on screen prompts to configure the trigger. See the section, "GraphQL Trigger", in the *Activities and Triggers Guide* for details on configuring the trigger. A flow with the name you specified gets created and attached to the newly created GraphQL trigger. This flow implements the method that you selected.

If needed, you can later make changes to the GraphQL schema file and upload it using the GraphQL trigger without creating a new flow. For more information, see the "GraphQL Trigger" section in the *TIBCO Flogo® Activities and Triggers Guide*.

To implement all methods defined in your `.gql` file

You can create flows to implement all methods defined in your `.gql` file. To do so follow these steps:

1. In the app details page, click **Create**. The **Add triggers and flows** dialog opens.
2. Select **Flow** under **Create new**.
3. Click **GraphQL Schema** under **Start with**.
4. Click **GraphQL Schema** and upload your `<name>.gql` file by either dragging and dropping it to the **Add triggers and flows** dialog or navigating to it using the **browse to upload** link. TIBCO Cloud Integration - Flogo (PAYG) validates the file extension. You see a green check mark and the **Upload** button appears.
5. Click **Upload**. TIBCO Cloud Integration - Flogo (PAYG) validates the contents of your schema and if it passes the validation, it creates the flows based on the methods defined in your schema file. One flow is created for each method in your schema. All the flows are attached the same trigger.

Manually attaching a flow to an existing GraphQL trigger in the app

If you have an existing flow in an app, you can manually attach it to a GraphQL trigger. To do so, follow these steps:

1. Click the flow name to open the flow details page.
2. Click the + icon to the left of your flow. The existing GraphQL triggers in the app display by default.
3. Select one of the existing GraphQL triggers and follow the on-screen directions.

Limitations on constructs in a GraphQL schema

TIBCO Cloud Integration - Flogo (PAYG) currently does not support the following GraphQL constructs:

- Custom scalar types
- Custom directives
- Subscription type
- Cyclic dependency in schema. For example, if you have a type `Book` which contains an object element of type, `Author`. The type `Author` in turn contains an element of type `Book` which represents the books written by the author. To retrieve the `Author`, it creates a cyclic dependency where the `Author` object contains the `Book` object and the `Book` type in turn contains the `Author` object.

Using gRPC

The out-of-the-box gRPC trigger in TIBCO Cloud Integration - Flogo (PAYG) uses a `.proto` file to define one or more services and the various Remote Procedure Calls (methods) under the service. For an understanding of gRPC concepts, refer to the gRPC documentation.

- In TIBCO Cloud Integration - Flogo (PAYG), the gRPC Trigger supports only protocol buffers (`.proto`) as the Interface Definition Language (IDL) for describing both the service interface and the structure of the payload messages.
- Currently, TIBCO Cloud Integration - Flogo (PAYG) only supports the proto3 version of the protocol buffer. Refer to the Creating a Flow Attached to a gRPC Trigger section for details on flow and gRPC trigger creation.
- While creating the `.proto` file, consider the limitations in section "Limitations when creating the `.proto` file".
- You must not use the same gRPC `.proto` file for a gRPC trigger and gRPC activities in the same app. The package names for the gRPC trigger and gRPC activities must be unique.
- The gRPC trigger and gRPC activity do not support options in the `.proto` file. If your `.proto` file contains any options, be sure to remove the options in `.proto` file before using it.

Creating a new gRPC trigger

To create a new gRPC trigger, follow these steps:

1. Open the app details page.
2. Click **Create**. The **Add triggers and flows** dialog opens.
3. Under **Create new**, select **Trigger**.
4. Select the **gRPC Trigger** card.
5. Follow the on-screen prompts to configure the trigger. See the section, "gRPC Trigger", in the *Activities and Triggers Guide* for details on configuring the trigger.
6. Click **Create**. The new gRPC trigger gets created with a placeholder for a flow attached to it.

To implement a single method in your `.proto` file

You can implement only a single method from your `.proto` file. To do so, follow these steps:

1. In TIBCO Cloud Integration - Flogo (PAYG), open the **Apps** page and click **Create**.
2. In the **Add triggers and flows** dialog, click **Flow** under the **Create new**.
3. Enter a name for the flow in the **Name** text box. Optionally, enter a description for the flow in the **Description** text box.
4. Click **Create**.
5. Select **Start with a trigger**.
6. In the **Triggers catalog**, select the **gRPC Trigger** card.
7. Follow the on-screen prompts to configure the trigger. See the section, "gRPC Trigger", in the *Activities and Triggers Guide* for details on configuring the trigger. A flow with the name you specified gets created and attached to the newly created gRPC trigger.

To implement all methods defined in your `.proto` file

You can generate the gRPC trigger along with implementing one flow per method defined in your `.proto` file. The flows will all be attached to the same trigger. To do so follow these steps:

1. On the app details page, click **Create**. The **Add triggers and flows** dialog opens.
2. Select **Flow** under **Create new**.
3. Click **gRPC Protobuf** under **Start with**.
4. Click **gRPC Protobuf** and upload your `<name>.proto` file by either dragging and dropping it to the **Add triggers and flows** dialog or navigating to it using the **browse to upload** link. TIBCO Cloud Integration - Flogo (PAYG) validates the file extension. You see a green check mark and the **Upload** button appears.
5. Click **Upload**. TIBCO Cloud Integration - Flogo (PAYG) validates the contents of your schema and if it passes the validation, it creates the flows based on the methods defined in your schema file. One flow is created for each method in your schema. All the flows are attached to the same trigger.

Manually adding an existing flow to the trigger

If you have an existing flow, you can manually add the flow to a gRPC trigger. To do so:

1. Open the flow details page by clicking on the flow name.
2. Click the + icon to the left of your flow.

You have the option to either select an existing gRPC trigger that is used in another flow in the same app or you can create a new gRPC trigger and attach this flow to the new trigger.

- To select an existing gRPC trigger already used in the app, click **gRPC Trigger**. The **Port** field is disabled, as this trigger is already in use by other flows. Refer to the "gRPC Trigger" section in the *TIBCO Cloud™ Integration - Flogo® (PAYG) Activities and Triggers Guide* for details on the other fields.
- To create a new gRPC trigger and attach the flow to it, click **Add new trigger** and click on the gRPC Trigger from the **Triggers catalog**. Refer to the "gRPC Trigger" section in the *TIBCO Cloud™ Integration - Flogo® (PAYG) Activities and Triggers Guide* for details on the other fields.

3. Click **Finish**.

Limitations when creating the `.proto` file

You must adhere to the following limitations when creating your `.proto` file:

- Currently, importing a `.proto` file into another `.proto` file is not supported. Hence, you cannot use `import` statements in a `.proto` file.
- Streaming is not supported in either the request or the response.
- Since `import` statements are not supported in `.proto` files, you cannot use data types that need to be imported from other `.proto` files, such as `google.protobuf.Timestamp` and `google.protobuf.Any`.
- Cyclic dependency in request or response messages is not supported.
- Setting a default value to a blank field within a message is not supported.
- Maps for data definition are not supported.
- Oneof - gRPC mandates that you enter a value for only one field. TIBCO Cloud Integration - Flogo (PAYG) considers all fields optional in order to allow you to select any field and enter a value for it. If you enter a value for multiple fields, only the value you entered for the last field displayed will be accepted and the remaining field values above it will be ignored.

Building the App Binary for an App Containing gRPC Trigger from the CLI

To build an app that contains a gRPC trigger from the CLI:

1. Install `protoc` and `protoc-gen-go` libraries.

   The minimum supported versions are:

   `protoc` 3.8.0

   `protoc-gen-go` 1.3.2

   Refer to the gRPC.io documentation for details on installing the libraries.
2. Make sure the two libraries are included in your system PATH.
3. Build the app binary. For more information, refer to Building the App.

Conversion of Data Types in Protocol Buffer to JSON Data Types

JSON does not support all data types supported in protobuf. Hence, TIBCO Cloud Integration - Flogo (PAYG) converts some of the data types to an equivalent data type in JSON. Here is a list of data types supported by protobuf and their representation in TIBCO Cloud Integration - Flogo (PAYG).

This data type supported in protocol buffer…	…is converted to this JSON data type in TIBCO Cloud Integration - Flogo (PAYG)
int32	number
int64	number
double	number
float	number
uint32	number
uint64	number
sint32	number
sint64	number
fixed32	number
fixed64	number
bool	boolean
byte	string
string	string

Using App Properties and Schemas

This section discusses how to create app properties, which you can use when populating field values. It also describes how to create a schema that can be reused in your app.

App Properties

You can configure some supported fields with app properties when configuring triggers and activities. Connection-related app properties cannot be used for configuration anywhere within an app. Their only purpose is to allow you to change a connection value if need be during runtime. Configuration fields in your flow that require their values to be changed when the app goes from a testing stage to production are best configured using app properties instead of hard coding their values. App properties for triggers and activities reside within the app. App properties for connections are not modifiable from the **App Properties** dialog in the app.

The URL field in an activity is a good example of a field for which you would want different values – maybe an internal URL when testing the app and an external URL when the app goes into production. You may want the URL used in the activity to change when the app goes from a test environment to production. In such a case, it is best to configure the URL field in the activity with an app property instead of hardcoding the URL. This way, you can change the URL by changing the value of the app property used to configure the URL field.

An app property value can have one of the following data types:

- string
- boolean
- number
- password

Values for the password data type is encrypted and will not be visible by default. But when configuring the password value, you can click on the **Show/Hide password property value** icon (👁) to see the value temporarily in order to verify that it has been entered correctly.

App properties are saved within the app, so when you export or import an app, app properties configured in the app also get exported or imported with the app. Properties of data type `password` do not retain their values when an app is exported. So, you must reconfigure the password after importing the app.

Creating App Properties

You can create an app property as a standalone property or as a part of a group. Use a group to organize app properties under a parent. A parent acts as an umbrella to hold related app properties and is basically a label with a meaningful name. A parent does not have a data type associated with it. For instance, if you want to group all app properties associated with a particular activity, you can create a group with a parent that has the activity name and create all that activity-related app properties under that parent.

As an example, you can create LOG_LEVEL as a standalone app property without a parent. Or you can create it as a part of a hierarchy such as LOG.LOG_LEVEL with the parent of the hierarchy being LOG and LOG_LEVEL being the app property under LOG. Keep in mind that if you group properties, you must refer to them using the dot notation starting from the parent. For example, the LOG_LEVEL property must be referred to as LOG.LOG_LEVEL. You can nest a group within a group.

Once you create an app property either as a standalone property or under a group, you cannot move the property around to another location in the properties list.

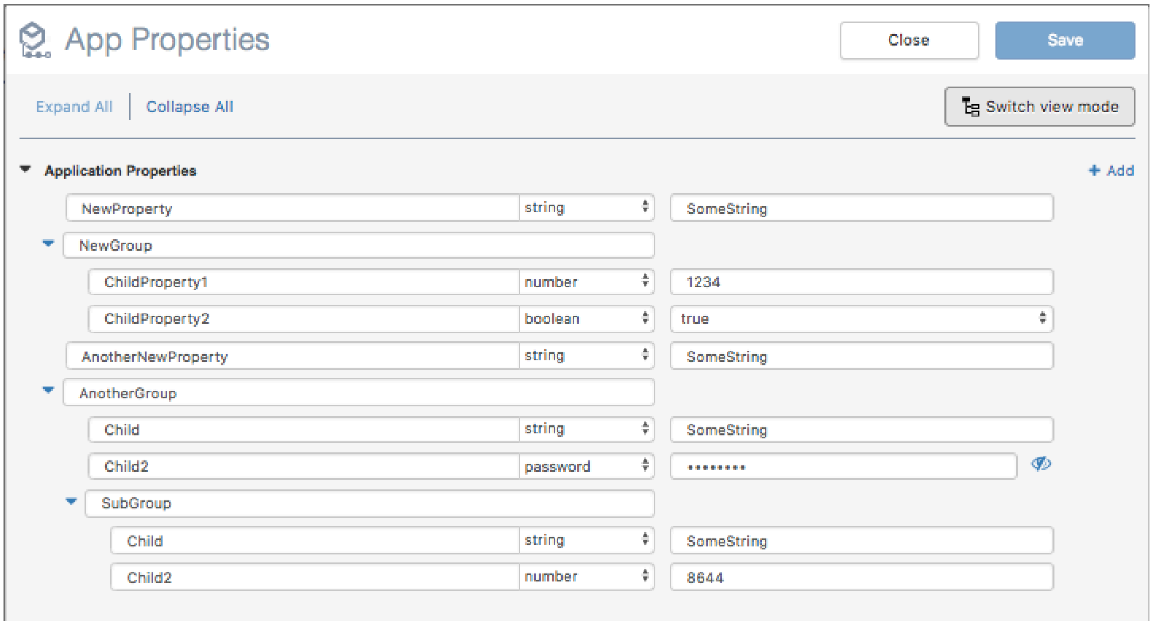
The **App Properties** dialog allows you to view your app properties in two formats (views). Refer to App Properties Dialog Views for details.

App Properties Dialog Views

You can view existing app properties for an app in the **App Properties** dialog. The **App Properties** dialog lets you view the properties in two formats (views), the list view format and the tree view format. By default, the **App Properties** dialog opens in the tree view. To toggle between views click **Switch view mode**.

Tree view

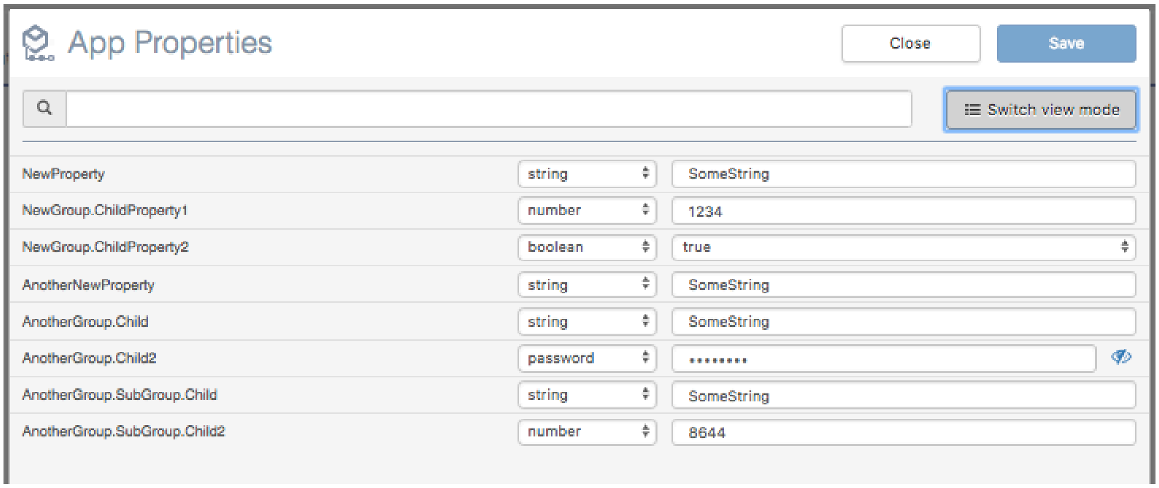
The tree view is the default view in which the **App Properties** dialog opens. In this view, you can add a new property, delete an existing property as well as edit the data type and value of a property.



NewProperty	string	SomeString
NewGroup		
ChildProperty1	number	1234
ChildProperty2	boolean	true
AnotherNewProperty	string	SomeString
AnotherGroup		
Child	string	SomeString
Child2	password	••••••••
SubGroup		
Child	string	SomeString
Child2	number	8644

List view

In the list view, you can edit the data type and value of a property but cannot add a new property or delete an existing property. The image below shows you how the properties in the above tree view image are presented in the list view.



NewProperty	string	SomeString
NewGroup.ChildProperty1	number	1234
NewGroup.ChildProperty2	boolean	true
AnotherNewProperty	string	SomeString
AnotherGroup.Child	string	SomeString
AnotherGroup.Child2	password	••••••••
AnotherGroup.SubGroup.Child	string	SomeString
AnotherGroup.SubGroup.Child2	number	8644

Creating a Standalone App Property

To create a standalone app property for your app, follow the steps below.

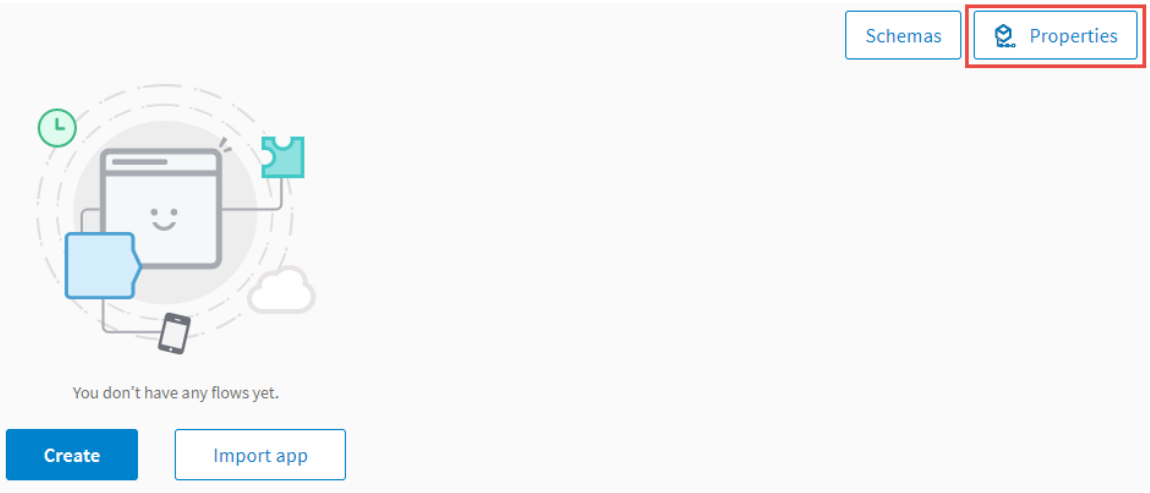
Once you create an app property either as a standalone property or under a group, you cannot move the property around to another location in the properties list.

To create a group, see Creating a Group.

No two standalone properties (properties that are not in a group) can have identical names. Also, property names within the same group must be unique.

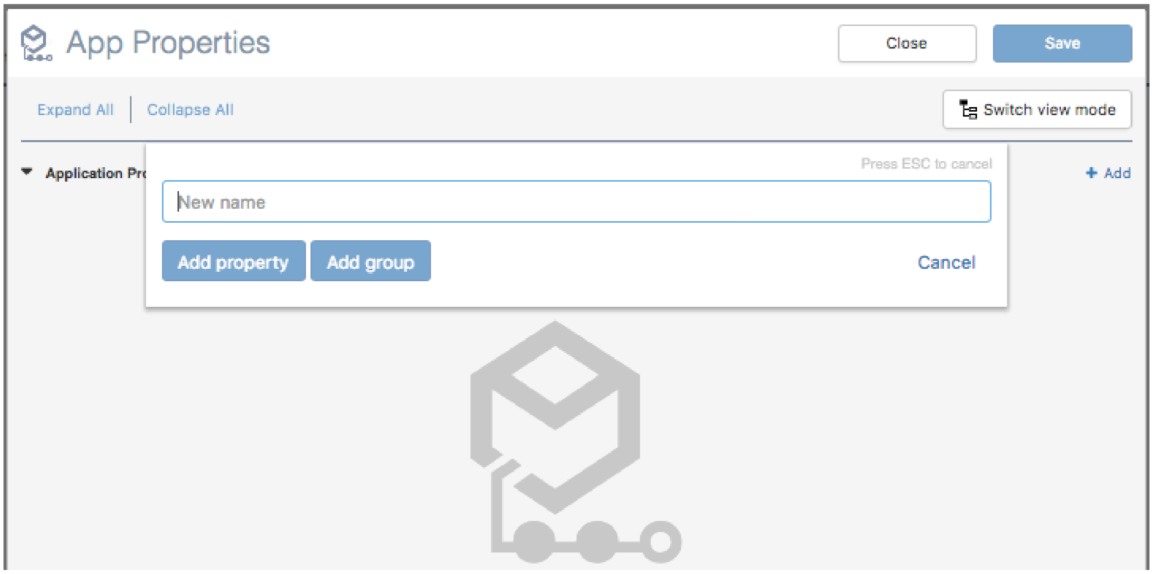
Procedure

1. If your app does not exist, create a new app, and click the **Properties** button shown on the screen below.

   If your app already exists, then open the app details page and click **Properties**.

   The **App Properties** dialog opens.

   If you already have existing properties, they will be displayed. Click **Add** to add another property.

2. Enter a meaningful name for the property you want to create.

   The property name must not contain any spaces or special characters other than a dash (-) or an underscore (_).

3. Click **Add property** to create a standalone property.
   The property gets created. When you create a property, by default, the property list is presented in a tree view. To view the properties in a list view mode, click **Switch view mode**.

4. Select the data type for the new property from its drop-down list.

5. Enter a default value for the property in the text box next to the property.

6. Click **Save**.
   TIBCO Cloud Integration - Flogo (PAYG) runs validation in the background as you create a property. The validation takes into consideration the property type and default value of the property that you entered. The **Save** button gets enabled only when the validation is successful. Make sure you do not skip this step of saving your newly created property or group.

Creating a Group

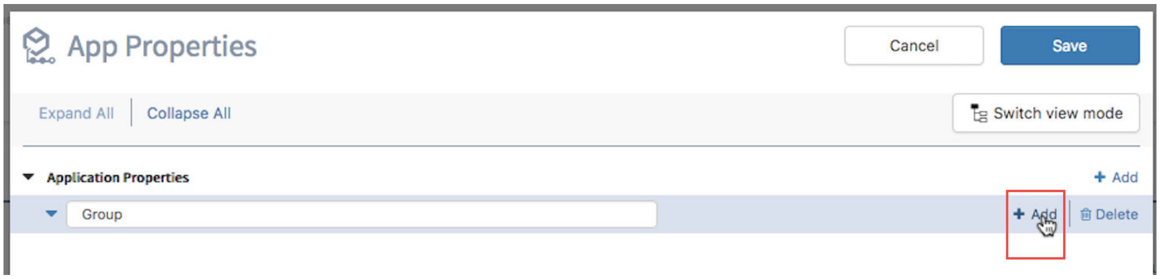
You can create one or more standalone app properties or group app properties such that they show up in a hierarchy. A group (or hierarchy) consists of a parent node, which is just a label and does not have a data type associated with it. You must create properties within the parent. You can do so in the **Application Properties** dialog. When creating a group you must add the parent first then create the app properties under the parent.

Once you create an app property either as a standalone property or under a group, you cannot move the property around to another location in the properties list.

Group names within an app must be unique. Also, property names within a group must be unique.

Procedure

1. Open the app details page and click **App Properties**.
2. Click **Add** on the upper right corner to add the group.
3. Enter a meaningful name for the group and click **Add group**.
   The group gets created. The group is simply a label and cannot be used by itself. So, you must add properties within the group.
4. To add a property within the group, hover your mouse cursor to the extreme right of the group until the **Add** button appears in the group row.



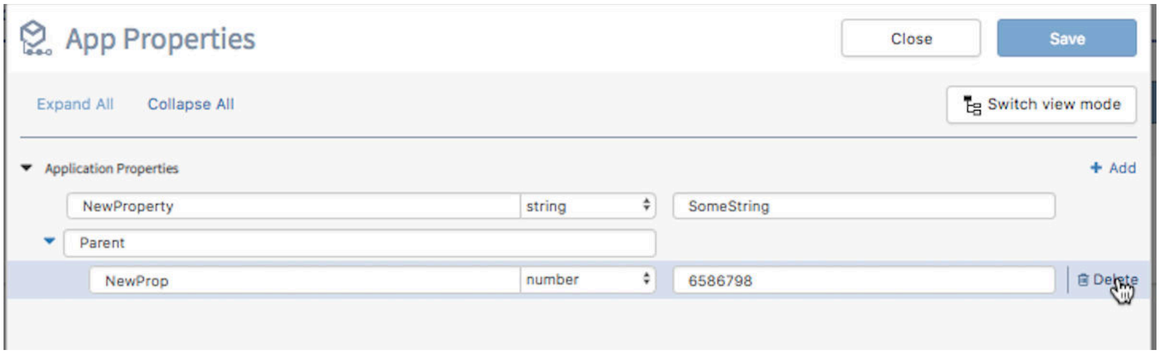
5. Click the **Add** button and enter a name for the property and click **Add property**.
6. Select a data type for the property and enter a value. Entering a value and selecting a data type are mandatory. The **Save** button will remain disabled without them.
7. Click **Save**.
   The property gets created under the parent.

Deleting a Group or Property

An existing property must be deleted from the tree view only in the **App Properties** dialog. The list view does not allow you to add or delete properties. Deleting a child property does not delete its parent, but deleting a parent will delete all properties under it.
To delete a property, follow these steps:

Procedure

1. Open the **App Properties** dialog from the app details page.
2. Hover your mouse cursor to the extreme right end of the property and click **Delete**. To delete a parent, hover your mouse cursor to the extreme right end of the parent and click **Delete**.



NewProperty | string | SomeString
Parent
NewProp | number | 6586798

3. Click **Save**.

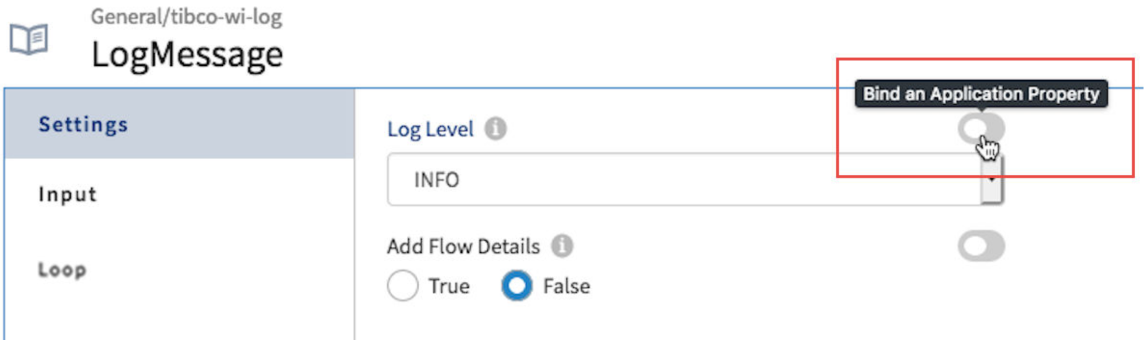
Using App Properties in a Flow

Configuring a field with an app property is recommended for fields that require their values to be overridden when the app goes into production. Hence, the decision as to which fields in an activity should support app properties (which fields can be configured using an app property) must be decided at the time when the extension for the category is being developed. The fields that can be configured using an app property display a slider button against their names in the UI.
Connection-specific app properties are visible in the **App Properties** dialog after you select a connection when configuring the activity or trigger, but they appear in read-only mode. This is because connections are reusable across apps and connection-related app properties are managed (refreshed) automatically. Connection-related app properties cannot be used for configuration anywhere within an app. Their only purpose is to allow you to change a connection value if need be during runtime. For more details on how the connection properties get created and used, see Using App Properties in Connections.

To configure a field with an app property, follow these steps:

Procedure

1. Open the flow details page.
2. Click the activity whose field you want to configure with an app property.
   This opens the configuration pane for the activity.
3. Click the slider ( ) against the name of the field you want to configure with an app property. If the field does not display a slider, the field can not be configured with an app property.

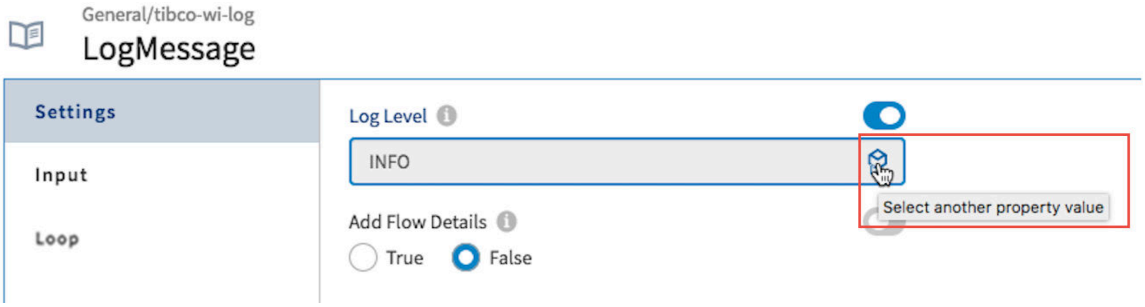


The **App Properties** dialog opens. Only those app properties whose data type match the data type of the field are displayed.

4. Click the property you want to configure for the field.

   The property name appears in the text box for the field and the default value of the property gets implicitly assigned to the field.

   After configuring the property, if you want to change a field to use a different property, hover your mouse cursor over the end of the text box for the field until the **Select another property value** icon appears. Click the **Select another property value** icon.



   For a field that has been configured with an app property, you can unlink the property from the field. Refer to Unlinking an App Property from a Field Value for more details.

Using App Properties in the Mapper

You can use app properties when mapping an input field. The app properties available for mapping are grouped under the **$property** domain-specific scope in the mapper. All mapper rules and conditions apply to the use of app properties as well. For example, the data type of the app property value must match with the input field data type when mapping. Connection-related app properties that are used by any connection field in an activity do not appear under **$property** since they cannot be accessed. Connection-related app properties cannot be used for configuration anywhere within an app. Their only purpose is to allow you to change a connection value if need be during runtime. Hence, they cannot be used to map input fields.

Refer to the section on Mapper for details on how to use the mapper.

Unlinking an App Property from a Field Value

For a field that has been configured with an app property, if you decide at a later time not to use the app

property, you can click and slide its slider ball (●) to the left. This will remove the app property from the field (unlink it from the field) but will leave the field configured with the default value of the app property. The field retains the default value of the app property, but it gets be disassociated from the app property and will appear as a manually entered value. Hence, if you change the default value of the app property beyond this point, it will not affect the value of the field.

Using App Properties in Connections

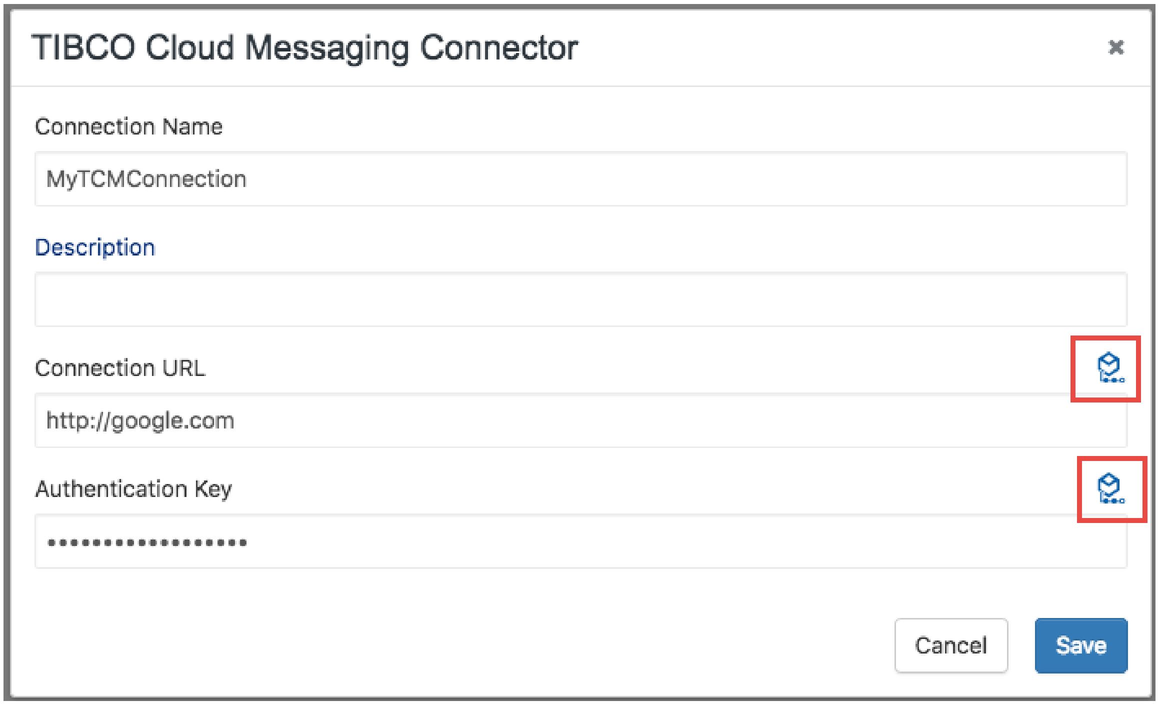
Connection-related app properties cannot be used for configuration anywhere within an app. Their only purpose is to allow you to change a connection value if need be during runtime. Hence, you cannot use connection-related app properties to map an input field as these properties are not visible in the mapper. You can only view them in the **App Properties** dialog, but cannot edit, update, or delete them. Before you your app, their values can only be edited in the connection details dialog, the dialog where you provided the credentials for the connection. You can open this dialog by editing the connection from the **Connections** page in the UI. Connection-related properties are useful when you want to change the value for one of the connection fields, for example a URL, when an app goes from the testing stage to production.

How the connection-related app properties get created

You cannot explicitly create connection-related properties. When you select a connection in the **Connection** field of an activity, the supported properties associated with that connection automatically get created and populated in the **App Properties** dialog.

While creating a connection, the fields in the connection details dialog that support app properties are marked with icon. One property gets created for each field that is marked with in the connection details dialog The values you enter for such fields in the connection details dialog become the default values for the connection properties. The properties take their name from the connection field they represent in the connection details dialog.

You begin by creating a connection. In the example below, only the **Connection URL** and **Authentication Key** fields support app properties. These are the only two fields that display against them.



TIBCO Cloud Messaging Connector

Connection Name

MyTCMConnection

Description

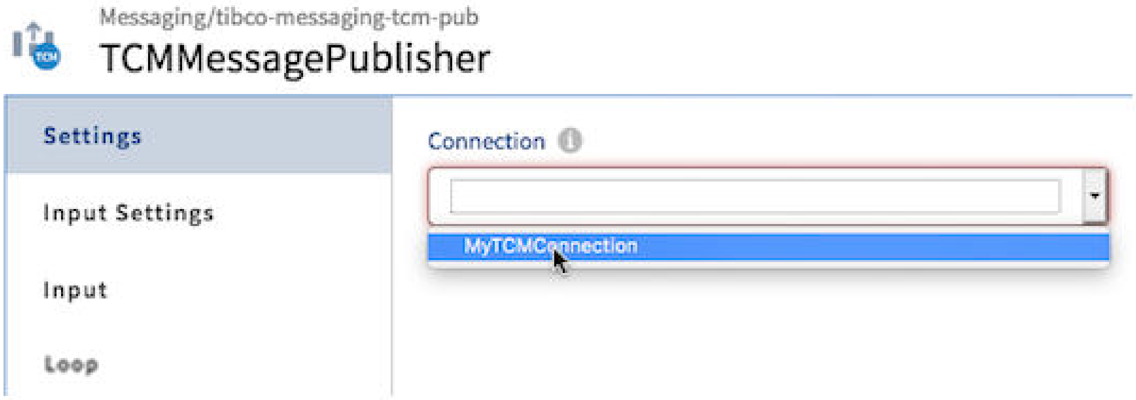
Connection URL

http://google.com

Authentication Key

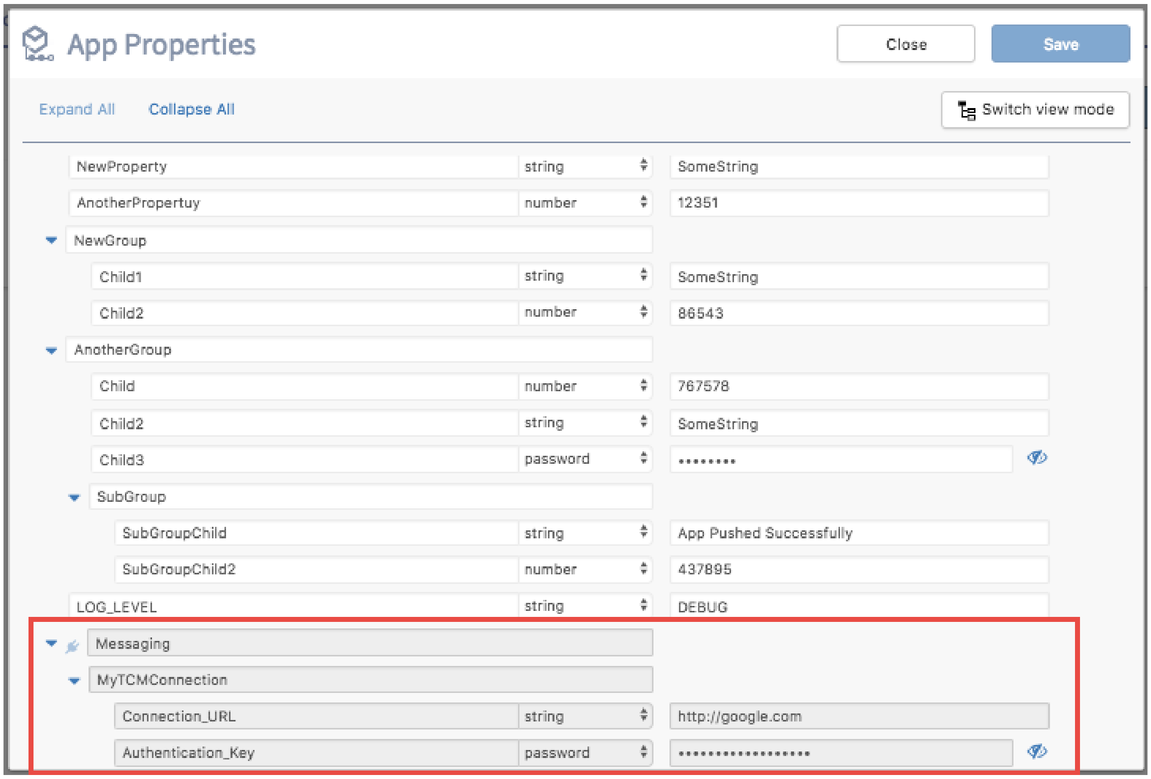
••••••••••••••••••

Once the connection is created, you can use it to configure the **Connection** field in an activity. In the example below, the connection created above is being used to configure the **Connection** field of the **TCMMessagePublisher** activity.



Connection
MyTCMConnection

After configuring the Connection field with the connection, if you open the **App Properties** dialog, you should see the connection properties for the field (enclosed in the red box in the image below) displayed. Notice that only the supported properties (Connection URL and Authentication Key) are displayed in a read-only mode.



NewProperty	string	SomeString
AnotherPropertuy	number	12351
NewGroup		
Child1	string	SomeString
Child2	number	86543
AnotherGroup		
Child	number	767578
Child2	string	SomeString
Child3	password	••••••••
SubGroup		
SubGroupChild	string	App Pushed Successfully
SubGroupChild2	number	437895
LOG_LEVEL	string	DEBUG
Messaging		
MyTCMConnection		
Connection_URL	string	http://google.com
Authentication_Key	password	••••••••••••••••••

The properties that get displayed in the **App Properties** dialog change dynamically based on your selection of the connection to use. You can only view the connection properties. You cannot edit or delete them from the **App Properties** dialog. Deleting the activity that uses the connection will automatically remove the associated connection properties that the activity used from the **App Properties** dialog.

Using connection-related app properties

Connection-related app properties are not available for use from the mapper. You can use these properties to change a connection value (for example, a URL or password) just before an app goes from a testing stage to production. Their values cannot be changed from the **App Properties** dialog. Change their values in the connection details dialog before the app.

Editing an App Property

You can change the default value or data type of an app property at any time.

Changing the Default Value of a Property from the App Properties Dialog

You can change the default value of an existing app property at any time after creating the property. Before you , you can change the default value in the **App Properties** dialog.

To change the default value of an existing app property, follow these steps:

Procedure

1. Open the **App Properties** dialog by clicking the **Properties** button on the app details page.
2. Click inside the text box for the property value you want to change.
3. Edit the value.
4. Click **Save**.

Changing the Name or Data Type of an App Property after Using It

If you change either the name of an app property or its data type after you have used the property to configure a field in an activity or trigger, the field displays an error message. You must explicitly reconfigure the field to use the modified property by deleting the property from the text box for the field and adding the modified property.

When Importing an App

An app being imported could have its own app properties. The app properties get imported along with the app. If an app property in the app being imported has a name that is identical to a property in the host app, you will see a warning message saying so with a choice to either overwrite the existing host property (by clicking **Continue**) with the property definition from the imported app or cancel the import process altogether.

App properties of type `password` do not retain their values when the app is exported, hence you must reconfigure the default values of all app properties of data type `password` after you import the app.

Exporting App Properties to a File

You can export the app properties to a JSON file or a `.properties` file. The exported JSON file can be used to override app property values. When using the exported properties file, the values in the properties file get validated by the app during runtime. If a property value in the file is invalid, you get an error saying so and the app proceeds to use the default value for that property instead.

Exporting the app properties to a JSON file

Exporting the app properties to a JSON file allows you to override the default app property values during app runtime. It is useful if you want to test your app by plugging in different test data with successive test runs of your app. You can set the app properties in the exported file to a different value during each run of the app. The default app property values get overridden with their values that you set in the exported file.

To export the app properties to a JSON file, run the following command from the directory where your app resides:

```
./<app-binary-name> -export props-json
```

The properties get exported to `<`*`app-binary-name`*`>-props.json` file.

Exporting app properties to a `.properties` file

You cannot use a `.properties` file format to override the app properties that were externalized using environment variables. To export the app properties to a `.properties` file, run the following command from the directory where your app resides:

```
./<app-binary-name> -export props-env
```

The properties get exported to `<app-binary-name>-env.properties` file. The names of the app properties appear in all uppercase in the exported `env.properties` file. For example, a property named `Message` will appear as `MESSAGE`. A hierarcy such as `x.y.z` will appear as `X_Y_Z`.

App Schemas

You can define a JSON or Avro schema such that it is available for reuse across an app. Creating an app-level schema saves you time and effort of entering the same schema multiple times. An app-level schema can be used in any flow, activity, or trigger configuration where a schema editor is provided. You can simply pick an existing schema from a list. For example, app-level schemas are available from the following locations:

- Inputs or Outputs tab of a flow (including Error Handler flows and subflows)
- Input or Output Settings tab of an activity
- Output or Reply Settings tab of a trigger

App-level schemas are filtered based on the type of the activity or trigger. For example, only JSON schemas are displayed in a REST trigger or activity configuration.

Currently, TIBCO Cloud Integration - Flogo (PAYG) only supports the JSON and Avro types of schemas.

Defining an App-Level Schema

Procedure

1. On the App Details page, click **Schemas**.
   The Schemas page opens.
2. Click **+Schema**.
3. In the **Schema Name** field, enter a schema name.
4. Select the type of schema. You can select either JSON or Avro schema. The default is JSON schema.
5. Enter the schema in the schema editor.

   If you enter JSON data in the editor, it is automatically converted to JSON schema.
6. Click **Save**.

Result

After the schema is defined, it can be used in any activity or trigger configuration by using the **Use an app-level schema** button in the schema editor of the activity or trigger.

Editing an App-Level Schema

When you make changes to an app-level schema, the changes are automatically reflected everywhere the schema is used.

To edit an app-level schema, follow these steps:

Procedure

1. On the App Details page, click **Schemas**.
   The **Schemas** page opens.
2. Expand the schema to be edited.
3. Edit the schema name or the schema in the editor, as required.
4. Click **Save**.
   If the app-level schema is used in any flow, activity, or trigger, a warning is displayed.

Deleting an App-Level Schema

Deleting a schema removes its reference from all the places where it is used, but it retains a copy of the schema in the fields that use the schema.

Procedure

1. On the App Details page, click **Schemas**.
   The **Schemas** page opens.
2. Click the **Delete** icon beside the schema to be deleted.

Result

After confirmation, the selected schema is deleted.

Using an App-Level Schema

You can use an app-level schema from a flow, trigger, or activity from the following tabs:

- Inputs or Outputs tab of a flow
- Input or Output Settings tab of an activity
- Output or Reply Settings tab of a trigger

Flow Input & Output Tab

Use these tabs to configure the input to the flow and the flow output. These tabs are particularly useful when you create blank flows that are not attached to any triggers.

The schemas for input and output to a flow can be entered or modified only in this **Flow Inputs & Outputs** accordion tab. You cannot coerce the flow input or output from outside this accordion tab.

Both these tabs (the **Input** tab and the **Output** tab) have two views:

- **JSON schema view:**

  You can enter either the JSON data or JSON schema in this view. You must click **Save** to save your changes or **Discard** to revert the changes. If you entered JSON data, the data is converted to a JSON schema automatically when you click **Save**.
- **List view:** This view allows you to view the data that you entered in the JSON schema view in a list format. In this view, you can:
  - Enter your data directly by adding parameters one at a time
  - Mark parameters as required by selecting its check box.
  - When creating a parameter, if you select its data type as an array or an object, an ellipsis (...) appears to the right of the data type. Click the ellipsis to provide a schema for the object or array.

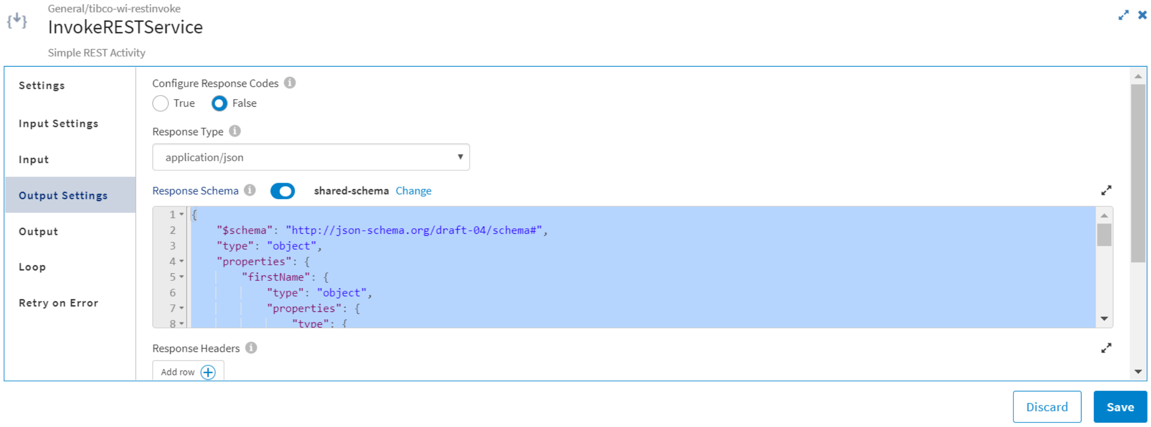
– Use an app-level schema by selecting the **Use an app-level schema** button. On the **Schemas** page that appears, click **Select** beside the schema that you want to use. The name of the schema is displayed beside the **Use an app-level schema** button and the schema is displayed in a read-only mode.

> You cannot edit an app-level schema in the **List** View if the **Use an app-level schema** button is selected. To edit an app-level schema, follow the instructions in the section Editing an App-level Schema. You can, however, switch to another app-level schema by clicking **Change** and selecting another app-level schema. You can also unbind the app-level schema (by deselecting the **Use an app-level schema** button) from a trigger, activity, or the input and output of a flow. After you unbind the app-level schema, you can make changes to it using the schema editor in the **List** View.

– Click **Save** to save the changes or **Discard** to discard your changes.

Input or Output Settings Tab of an Activity

When configuring an activity, you can select an app-level schema in its **Input** or **Output Settings** Tab. For example, the following screenshot shows an app-level schema selected in the **Response Schema** field of the **Output Settings** tab of an InvokeRESTService activity.



General/tibco-wi-restinvoke
InvokeRESTService
Simple REST Activity

Configure Response Codes
○ True ● False

Response Type
application/json

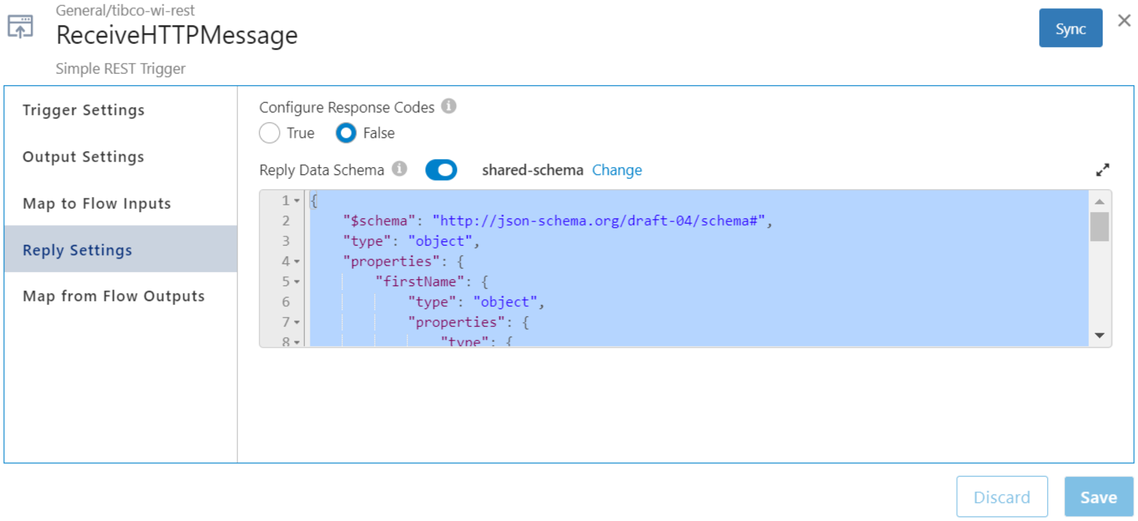
Response Schema ● shared-schema Change

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "type": "object",
    "properties": {
        "firstName": {
            "type": "object",
            "properties": {
                "type": {
```

Response Headers

Output or Reply Settings Tab of a Trigger

When configuring a trigger, you can select an app-level schema in its **Output** or **Reply Settings** Tab. For example, the following screenshot shows an app-level schema selected in the **Reply Data Schema** field of the **Reply Settings** tab of a ReceiveHTTPMessage trigger.



General/tibco-wi-rest
ReceiveHTTPMessage
Simple REST Trigger

Configure Response Codes
○ True ● False

Reply Data Schema ● shared-schema Change

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "type": "object",
    "properties": {
        "firstName": {
            "type": "object",
            "properties": {
                "type": {
```

If there is change in the schema attached to a trigger, click **Sync** to synchronize it with the input and/or output of the flow.

Using Connectors

TIBCO Cloud Integration - Flogo (PAYG) offers the ability to use connectors that have enterprise support and also connectors that are custom developed extensions.

This section is applicable only if you have uploaded custom extensions for connectors. The **Extensions** tab displays your uploaded extensions.

To use the TIBCO Flogo connectors, follow these steps:

1. Create one or more connections.
2. If you do not already have an app, create an app.
3. Create a flow.
4. Add the activities pertaining to the connector you use as needed.
5. Build the app.

Creating Connections

You must create connections before using the connectors in a flow. TIBCO Cloud Integration - Flogo (PAYG) uses the configuration provided in these connections to connect to the respective app, data sources, systems, or SaaS.

Prerequisites

You must have valid accounts for the SaaS apps to which you want to connect.

To create a connection, click the **Connections** tab on the TIBCO Cloud Integration - Flogo (PAYG) page.

To create a connection using a connector tile:

1. If this is the first connection you are creating, click the **Create connection** link. For subsequent connections, click the **Create** button on the **Connections** page.
2. Click the connector tile for which you want to create a connection.
3. Follow the instructions to configure the connection when prompted.

Make sure that the pop-up blocker in your browser is configured to always allow pop-ups from an app site. On MacOS, clicking the link to the site will result in the connection details page hanging, so make sure to select the radio button for "**Always allow pop-ups from <*site*>**."

Editing Connections

You can edit the name and other settings of your connection.
To edit an existing connection:

Procedure

1. In TIBCO Cloud Integration - Flogo (PAYG), click the **Connections** tab to open its page.
2. In the list of existing connections, click the connection that you want to edit. Edit the connection details in the connection details dialog that opens.
3. Click **Save**.

Deleting Connections

You can delete an existing connection.

Procedure

1. In TIBCO Cloud Integration - Flogo (PAYG), click the **Connections** tab to open its page.
2. In the list of existing connections, hover over the connection name that you want to delete until you see the **Delete connection** icon (🗑) appear at the end of the row.

   If the connection is being used by an app, you can see a blue icon in the **Usage** column. Hover over the icon to see which apps use the connection.

   > You cannot delete such connections.

3. Click the **Delete connection** icon.
4. On the confirmation dialog, click **Delete connection**.

Result

The selected connection is deleted.

Uploading Extensions

TIBCO Cloud Integration - Flogo (PAYG) gives you the ability to make use of your own extensions.

You can create extensions for TIBCO Cloud Integration - Flogo (PAYG) or you can upload a Project Flogo™ extension into TIBCO Cloud Integration - Flogo (PAYG).

Uploading TIBCO Cloud Integration - Flogo (PAYG) Extensions

You can create and contribute extensions for the following:

- activities
- triggers (you can define custom triggers that you can upload and use to create a flow)
- connectors (a connector provides configuration details to connect external apps, for example, Salesforce)
- functions (to be used inside the mapper when mapping elements)

After creating your extension, you upload its `.zip` file using the upload dialog.

The extension you upload must follow the guidelines found on the GitHub page, https://tibcosoftware.github.io/tci-flogo/.

Before you Upload

Keep the following in mind before you upload your extension:

- When uploading your activity or trigger extension, by default TIBCO Cloud Integration - Flogo (PAYG) compiles your extension before uploading it. If you would like to skip the compilation process, make sure to compile all the `*.ts` files in your extension and generate a `.js` file for each `.ts` file. The `.js` file must have an identical name as its corresponding `.ts` file.
- You will be responsible for the lifecycle (uploading, updating, deleting) of the extension that you contribute. Any extension that you contribute will be visible and available for use only to you.
- When creating your activity or trigger extension, if you did not specify a category for the extension, the extension will be placed in the **Default** category.

- The category name for an extension must be unique. If a category by the name already exists, the upload will completely overwrite the category. Out-of-the-box contributions cannot be overwritten.
- Special characters are not supported in activity and trigger names. You will get a validation error while uploading, if any names contain special characters.
- Uploading new extension(s) to an existing category will overwrite the entire category and all its contents. So, in order to add a new extension to an existing category while keeping the extension(s) that already exist in that category, be sure to include the existing extension(s) along with the new activity, connection, or trigger when creating the `.zip` file to be uploaded.
- You cannot delete a single extension from any category other than the **Default** category. To delete a single trigger, activity, or connector from a category, you must re-upload the whole category which includes all the extensions you want to keep minus the extension you want to delete. The same applies when editing an extension within a category - after editing an extension on your local machine, make sure to re-upload the whole category, the edited extension plus all the existing extensions in the category. Uploading only the edited extension will overwrite the category causing you to lose the other extensions in the category.

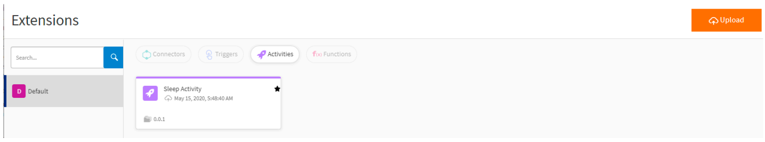
An extension that you upload to TIBCO Cloud Integration - Flogo (PAYG) is available for use in any flow that currently exists in your app or a flow that you will create in future.

To upload an extension, follow these steps:

This procedure assumes that you have the `.zip` file for your extension available for upload.

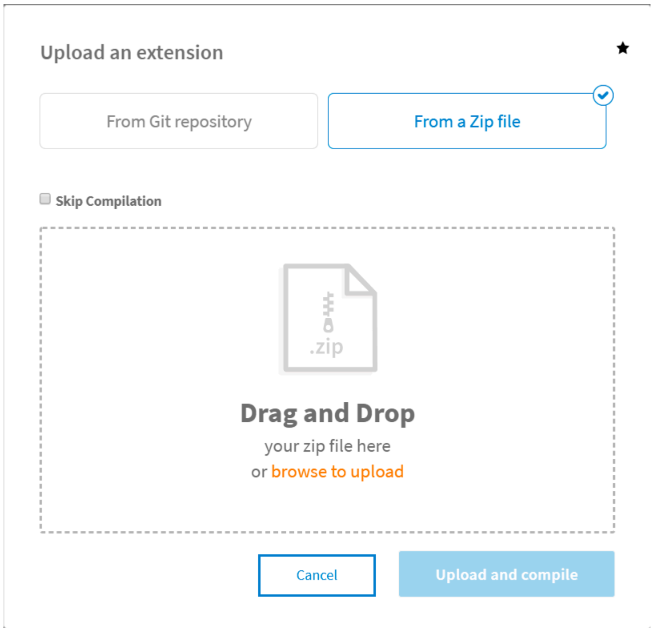
1. On the Apps page, click **Environment and Tools** tab.
2. Under **Connector Management & Extensions**, click **Extensions**.
3. If this is the first extension, click the **Upload an extension** button.

   If there are existing extensions, click the **Upload** button on the upper right corner:

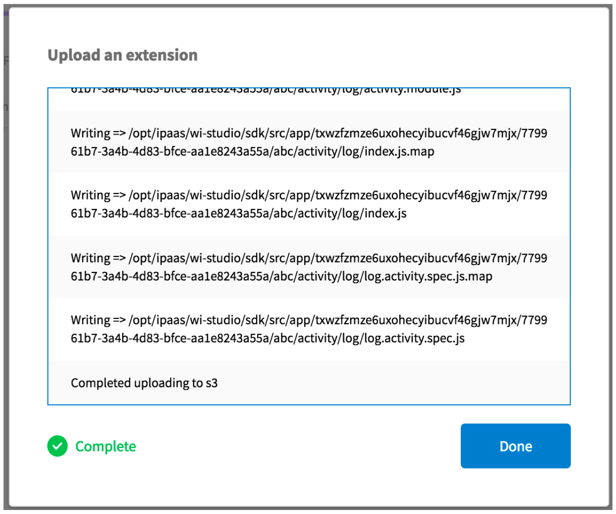
   The **Upload an extension** dialog opens. If you want to upload from the Git repository select **From Git repository**. See the section, Pulling Extensions from an Open Source Public Git Repository for more details on this.

   To upload an extension residing in a .zip file locally, click **From a Zip file**.



4. Click the **browse to upload** link and navigate to your extension `.zip` file. Alternatively, drag and drop the `.zip` file from your local machine to the area defined by a dotted line in the **Upload an extension** dialog.
5. If you would like to skip the compilation process, select the **Skip Compilation** check box. If the check box is selected, TIBCO Cloud Integration - Flogo (PAYG) performs a check before uploading to make sure that every `.ts` file has a corresponding `.js` file present. If a .ts file does not have a .js file, the validation fails and your extension does not upload.
6. Click **Upload and compile**.

   TIBCO Cloud Integration - Flogo (PAYG) validates the contents in the `.zip` file. If the `.zip` file contains a valid folder structure, it compiles the extension code. Once the code is compiled successfully, it uploads the extension to TIBCO Cloud Integration - Flogo (PAYG). You can view the progress of the upload or any errors that occur in the logs:



```
61b7-3a4b-4d83-bfce-aa1e8243a55a/abc/activity/log/activity.module.js

Writing => /opt/ipaas/wi-studio/sdk/src/app/txwzfzmze6uxohecyibucvf46gjw7mjx/7799
61b7-3a4b-4d83-bfce-aa1e8243a55a/abc/activity/log/index.js.map

Writing => /opt/ipaas/wi-studio/sdk/src/app/txwzfzmze6uxohecyibucvf46gjw7mjx/7799
61b7-3a4b-4d83-bfce-aa1e8243a55a/abc/activity/log/index.js

Writing => /opt/ipaas/wi-studio/sdk/src/app/txwzfzmze6uxohecyibucvf46gjw7mjx/7799
61b7-3a4b-4d83-bfce-aa1e8243a55a/abc/activity/log/log.activity.spec.js.map

Writing => /opt/ipaas/wi-studio/sdk/src/app/txwzfzmze6uxohecyibucvf46gjw7mjx/7799
61b7-3a4b-4d83-bfce-aa1e8243a55a/abc/activity/log/log.activity.spec.js

Completed uploading to s3
```

   You will see a **Complete** message after the extension is successfully uploaded. If there were any compilation errors during the upload, you see an error message and the upload exits. You can copy-paste the error message if need be.
7. Click **Done** to close the dialog.

You can view your extension on the **Extensions** page. The newly added extension appears under the category that you specified. If you had not specified a category for the extension, it appears in the

Default category. The activities are denoted by the 🚀 symbol and the connectors have the ⬡ symbol next to them.

The new extension displays the following:

- timestamp when the extension was loaded
- name of the extension contributor
- version of the extension

The **Search** field that appears above the category searches within the categories for the activity, trigger, or connector you specified in the search text box. You can filter the displayed extensions by clicking the **Triggers**, **Connectors**, or **Activities** buttons.

The extension is now available for you to use. If you uploaded an activity, the activity will be available for use when creating a flow or editing an existing flow. The activity will appear under the category you defined for it when creating the extension. The output of the activity will be available in the mapper just as it is for any default activities that come with the TIBCO Cloud Integration - Flogo (PAYG).

If you uploaded a connector, the connector will be available for creating new connections on the **Add Connections** > **Select connection type** dialog.

If you uploaded a trigger, the trigger will be available for you to select in the **Create a Flow** dialog, which when selected will create the flow with your trigger.

Uploading Project Flogo™ Extension

You can upload the following extensions created in Project Flogo™ to TIBCO Cloud Integration - Flogo (PAYG):

- activities
- triggers
- functions

To use an extension created in Project Flogo™, get the GitHub URL for the extension and upload it using the method described in Pulling Extensions from an Open Source Public Git Repository.

Changing Log Levels

When uploading an extension, you can see the logs on the screen. You can set the `FLOGO_LOG_LEVEL` engine variable to alter the log level at runtime. Be sure to do so *before* you begin uploading your extension. For details on the `FLOGO_LOG_LEVEL` engine variable, see the Environment Variables section.

Pulling Extensions from an Open Source Public Git Repository

You can upload extensions that are available from an open source public Git repository by pulling them directly into TIBCO Cloud Integration - Flogo (PAYG). This section describes how to pull the **Default** category Project Flogo™ extensions directly from an external public Git repository and upload it to TIBCO Cloud Integration - Flogo (PAYG). Pulling from private Git repositories is currently not supported.
Keep the following in mind when pulling the contribution:

- You can download only from public repositories. Accessing private Git repositories is not supported.
- The Git repository link should be the reference of the activity and not the URL.
- The repository link needs to be a reference of the contribution and must not begin with `http://` or `https://`, for example to pull the LogMessage activity from Project Flogo™ Git repository, use `github.com/project-flogo/contrib/activity/log`

- Any new default category contribution that you pull from the Git repository get appended to the ones that already exist for the category in TIBCO Cloud Integration - Flogo (PAYG). Contributions pulled and uploaded to other categories in TIBCO Cloud Integration - Flogo (PAYG), overwrites the category itself. Hence, if there are any existing activities in the category, they get deleted when the category is overwritten.
- Default category extensions can only be downloaded one at a time.

To pull an extension from a public Git repository, follow these steps:

Procedure

1. On the **Extensions** page, click **Upload**.
   The **Upload an extension** dialog opens.
2. Click **From Git repository**.
   When you select this option you are prompted to enter the location of the Git repository from which you want to pull the extension.
3. Enter the reference to the extension in the Git repository.

   > Make sure you do not enter the initial `http://`.

4. Click **Import**.
   TIBCO Cloud Integration - Flogo (PAYG) validates the format of the reference you entered in the **Git repository URL** text box. The **Import** button remains disabled until you enter a valid reference format. A `.zip` file for the activity gets generated and uploaded to the **Default** category on the Extensions page in TIBCO Cloud Integration - Flogo (PAYG). Once you start the process of downloading the contribution from the Git repository, you cannot cancel the process or switch to the process of uploading **From a Zip file**. You must wait for the upload process to complete, then click **Done**.
5. Click **Done**.
   The extension you uploaded appears on the **Extensions** page.

Adding Custom Golang Code or Dependencies to the App

You can use custom Golang code with your extensions. You can also use custom Golang code in your Flogo apps directly without using it in any extension. Your custom extensions and apps may also have dependencies. In the case of both, the custom Golang code and the dependencies, you have the option to either dynamically pull the latest version of the packages from `github.com` during build time, or to pull a specific version of the package and store it locally in `<`*FLOGO_HOME*`>/lib/ext/src` folder ahead of time. Storing a specific version of the package locally reduces your build time. Since the packages will be available locally, the build process saves the time it takes to pull them from an external source. Keep in mind that when the packages are pulled from the `github.com` repository during build time, the absolute latest version of the packages get pulled. So, if you want a specific version of the package, you must pull the package ahead of time and store it locally.

When you build the app binary, the build process first searches for the packages locally. If it does not find a package locally, it attempts to pull the package from `github.com` repository.

Adding custom Golang code to your app

> Once the custom Golang code is placed in the `<`*FLOGO_HOME*`>/lib/ext/src` folder, it will be a part of all apps built from that environment.

> Using custom Golang code is not supported in TIBCO Cloud™ Integration - Flogo®.

If you want to plug in a custom Golang code and include it as a part of your engine binary or Flogo app, you can use one of the following two methods:

- Pull and place the package in `<FLOGO_HOME>/lib/ext/src` directory

  Or

- Add the package to `deps.go`

To incorporate it into your build environment, do the following:

1. Pull the package from the Git repository ahead of time. This is not required if you want the package pulled dynamically during build time.

   a. Pull the package from the external Git repository (`github.com`) to your local environment.

   b. Place the package in `<FLOGO_HOME>/lib/ext/src` folder.

   c. Copy any dependent Golang packages required by your custom code into `<FLOGO_HOME>/lib/ext/src` folder.

      Skipping this step will pull the latest versions of the dependencies at build time.

2. Create a `deps.go` file under `<FLOGO_HOME>/lib/ext/src` folder if it does not already exist, and add an entry for the package pointing to its public URL.

   Here are two examples of the `deps.go` file content:

   ```
   package main


   import _ "github.com/<username>/contrib/eventlistener"
   ```

   ```
   package main

   import (
       _ "github.com/<username>/contrib/eventlistener"
       _ "github.com/<username>/contrib/example"
   )
   ```

   Add the public `github.com` URLs for all your custom Golang code packages here.

   The package name must be `main`.

   The build process uses the package URL stated in the `deps.go` file to look for the package on `github.com` if the package is not found locally. Once the code package is pulled, `deps.go` initializes the package.

Adding versioned dependencies to your custom extension

If your custom extension uses a third party library over which you have no control, you might want the extension to always use a specific version of the library. You can download the specific version of the library and place it under `<FLOGO_HOME>/lib/ext/src` folder. You must not add an entry in the `deps.go` file for dependency packages. Build the app binary.

Deleting Extensions or Extension Categories

You can delete an existing extension or extension category from the Apps page.
To delete an extension or extension category:

Procedure

1. On the Apps page, click **Environment and Tools** tab.
2. Under **Connector Management & Extensions**, click **Extensions**.
3. Hover your mouse cursor on the extension or extension category that you want to delete until the **Delete Extension** icon (🗑) displays.

4. Click 🗑.
5. On the confirmation dialog, click **Delete**.

Result

The selected extension or extension category is deleted.

Flow Tester

After you design a flow, use the Flow Tester to test the flow.

When designing a flow, runtime errors can go undetected until you build the app to execute the flow. It can become particularly cumbersome to test flows that start with a trigger, since the triggers activate based on an external event. So, before you can test the flow, you need to configure the external app to send a message to the trigger in order to activate the trigger and consequently execute the flow. The Flow Tester eliminates the need to activate the trigger in order to execute the flow.

You provide the input to the flow in the Flow Tester. The Flow Tester executes the flow on demand without using a trigger. Each activity executes independently and displays its logs. This lets you detect errors in the flow upfront without actually building the app.

The Flow Tester is disabled when activity type contributions are missing in the flow execution.

You can run the Flow Tester in the debug mode with the following features only:

- Test run the flow
- See the flow execution
- Select an activity which is executed and see the inputs and outputs
- Change the inputs to other valid values and start the activity from that point onwards

You cannot:

- Insert a debug point and stop the flow execution at a tile
- Skip a tile from test execution

Testing Flows from the UI

You can use the Flow Tester from the TIBCO Cloud Integration - Flogo (PAYG) Web UI or you can use the CLI to run the `test` command in the Flow Tester. This section describes how to use the Flow Tester from the Web UI.
When using the Flow Tester from the Web UI, you must populate your test data in the Launch Configuration. Launch Configuration is a mechanism used by the Flow Tester in the Web UI to store your test data.

What is a Launch Configuration?

A Launch Configuration is a test configuration that contains a set of data with which to test the flow. Create a Launch Configuration to hold the test data that you want to use as input to the flow. Launch Configurations allow you to save and use your input data without having to enter it every time you want to test or retest the flow.

Blank flows use data configured in the **Input Settings** tab of the **Flow Inputs & Outputs** accordion tab as the input to the flow. Flows created with a trigger use the output of the trigger as input to the flow.

Launch Configurations are particularly useful when you want to test the flow multiple times with complex data or multiple sets of data. Create a Launch Configuration once with the data and use the Launch Configuration as input to the flow instead of manually entering the data every time you want to execute the flow. You can create multiple Launch Configurations, each containing a different set of data. A Launch

Configuration can contain only one set of data. To test a flow with multiple sets of data, create multiple Launch Configurations for a flow with each containing one set of data, then test the flow with one Launch Configuration at a time.

Once you create a Launch Configuration, it automatically gets saved and is available to you until you explicitly delete it.

> When exporting an app, if the app contains Launch Configurations, the Launch Configurations do **not** get exported with the app. Launch Configurations in an app must be exported independent of the app export.

Creating and Using a Launch Configuration

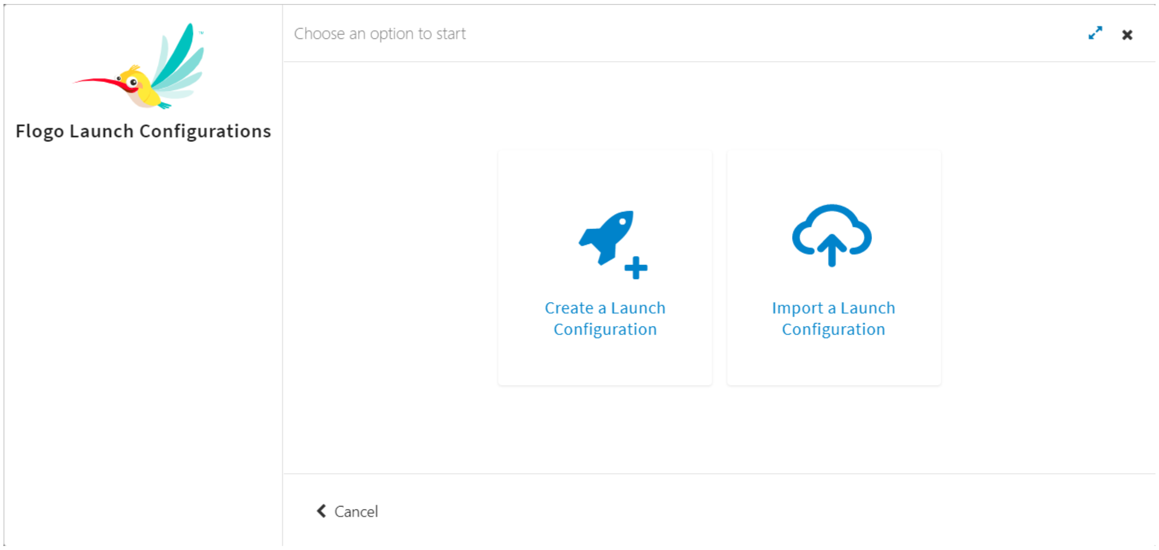
Launch Configurations need not be explicitly saved. They persist even after you exit TIBCO Cloud Integration - Flogo (PAYG) and log back in later.

Creating your first Launch Configuration

To create the very first Launch Configuration in a flow:

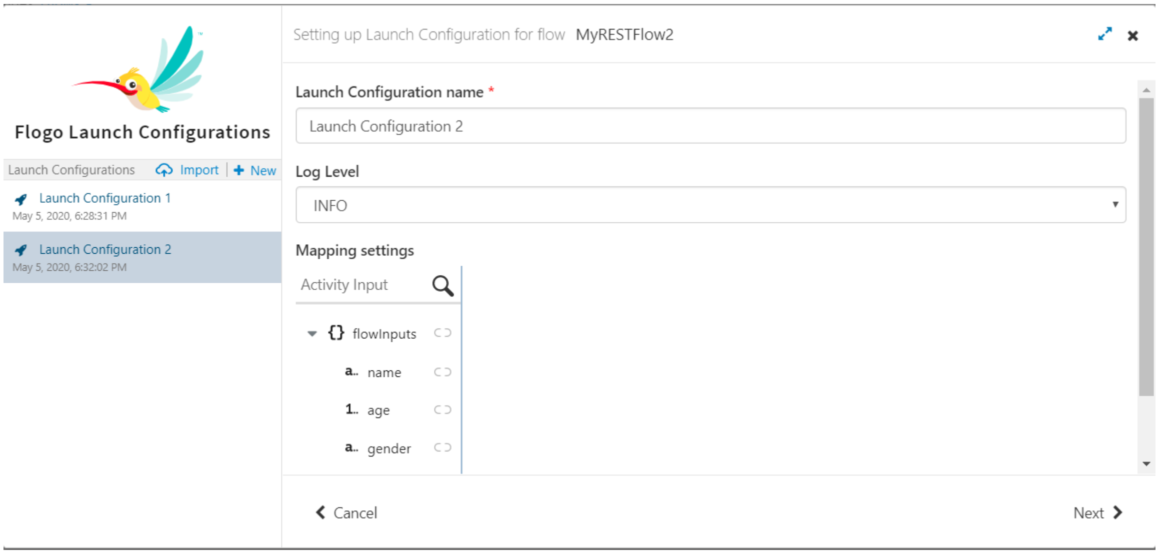
1. On the flow details page, click **Test** 🚀 Test.

   You can either start a new Launch Configuration by clicking **Create a Launch Configuration** or use an existing Launch Configuration that you had exported from another flow by clicking **Import a Launch Configuration**.

   
   Flogo Launch Configurations
   Choose an option to start
   Create a Launch Configuration
   Import a Launch Configuration
   Cancel

2. Click **Create a Launch Configuration**.

   The Flow Tester opens with the left pane displaying the Launch Configuration name. By default, a new Launch Configuration is named "Launch Configuration x" where x stands for a number. For example, since this is the first Launch Configuration that you are creating, the name of the Launch Configuration displays as `Launch Configuration 1`. The next Launch Configuration you create will be named `Launch Configuration 2`. You can edit the name in the right pane. The right pane opens the mapper which displays the flow input tree.



Launch Configuration 1
May 5, 2020, 6:28:31 PM

Launch Configuration 2
May 5, 2020, 6:32:02 PM

Setting up Launch Configuration for flow MyRESTFlow2

Launch Configuration name *: Launch Configuration 2

Log Level: INFO

Mapping settings

Activity Input

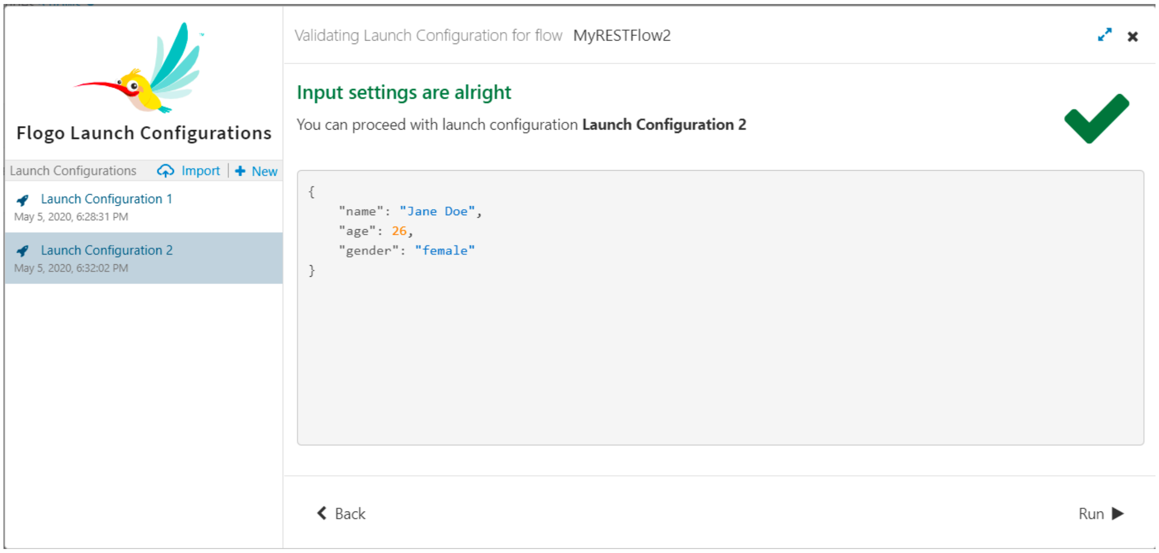
- {} flowInputs
  - a.. name
  - 1.. age
  - a.. gender

3. Optionally, enter a meaningful string to replace the default name in the **Launch Configuration name** text box.
4. Select the log level from the **Log Level** drop down menu.
5. Enter the values for the elements in the input tree. Refer to Configuring a Launch Configuration for details on entering values.

   > If your flow does not require an input, for example, if your flow was created with a Timer trigger which does not have an output (consequently no input to the flow), you can continue testing the flow without using the mapper in the Flow Tester.

6. Click **Next**.

   The input values you entered are displayed and validated. If no errors are found you get the message, **Input settings are alright**.



Launch Configuration 1
May 5, 2020, 6:28:31 PM

Launch Configuration 2
May 5, 2020, 6:32:02 PM

Validating Launch Configuration for flow MyRESTFlow2

Input settings are alright

You can proceed with launch configuration **Launch Configuration 2**

```
{
    "name": "Jane Doe",
    "age": 26,
    "gender": "female"
}
```

7. Click **Run** to execute the flow with the input data you provided in the step above.

   All the activities in the flow are executed. For details, see What can you do using the Flow Tester?.

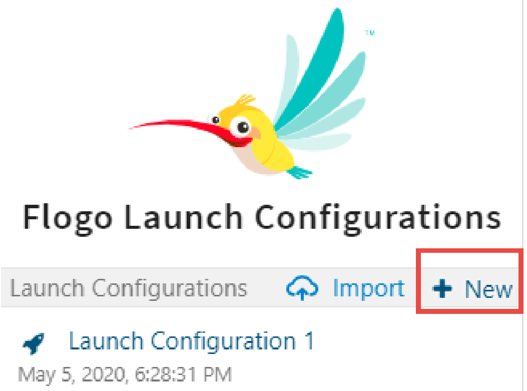
8. Click **Stop Testing**.

Creating Subsequent Launch Configurations

If you have an existing Launch Configuration:

Procedure

1. Click **New** to create another Launch Configuration.



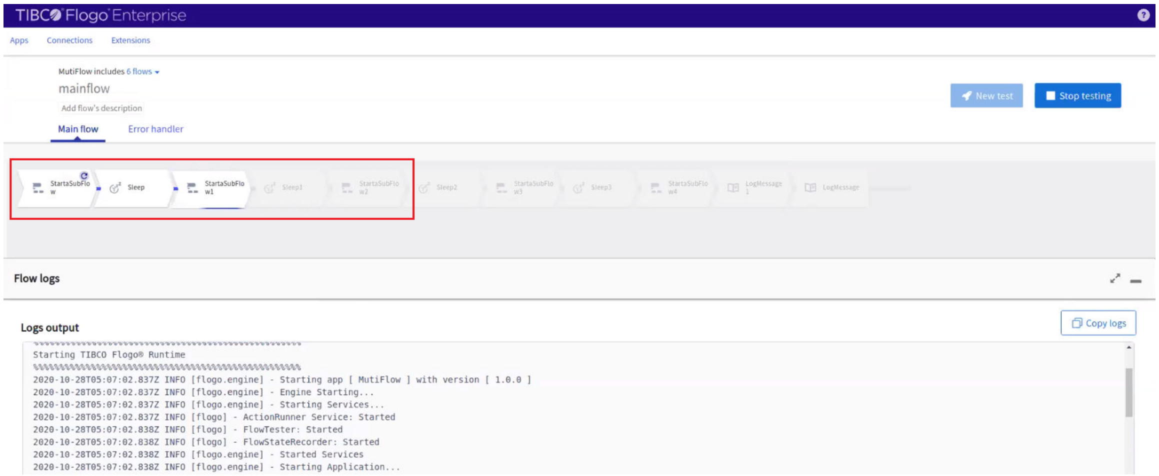
2. Follow the procedure from step 3 onwards in Creating your first Launch Configuration.

What can you do using the Flow Tester?

When you use the Flow Tester to test a flow, all the activities in the flow are executed. While the flow tester is active, you cannot add or delete an activity in the flow.

When an activity is being executed, a blue animation is displayed around it. When the execution of the activity is completed, the activity is highlighted in the flow and the blue animation moves to the next activity. Activities that have not completed execution are greyed out. This helps you see the progress made in the execution of the flow.

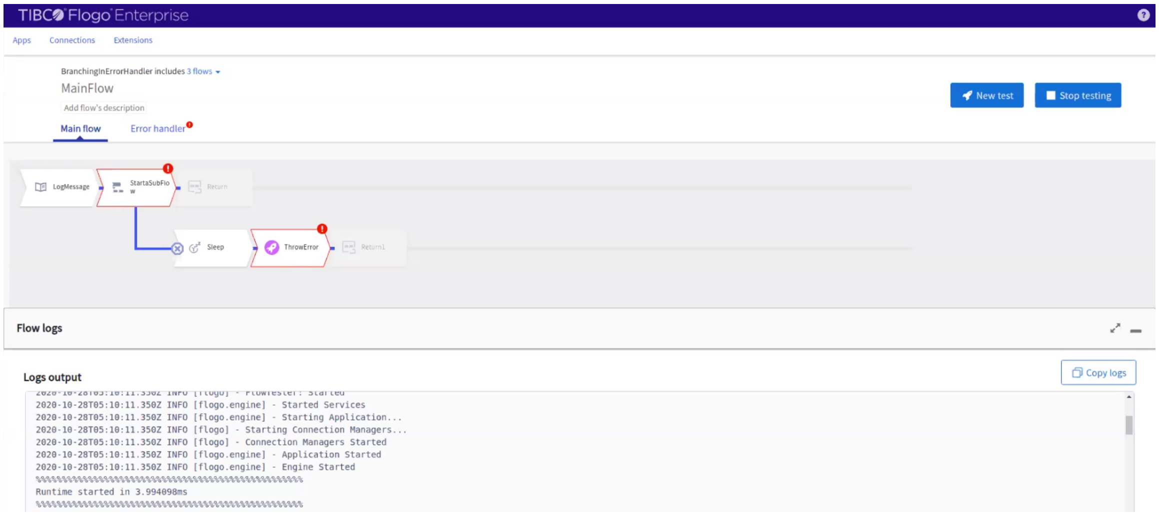
It is a good practice to stop testing by clicking **Stop Testing** when you finish running a flow in the Flow Tester, as the login session remains active for the entire time that you are in the testing mode.



```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Starting TIBCO Flogo® Runtime
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2020-10-28T05:07:02.837Z INFO [flogo.engine] - Starting app [ MutiFlow ] with version [ 1.0.0 ]
2020-10-28T05:07:02.837Z INFO [flogo.engine] - Engine Starting...
2020-10-28T05:07:02.837Z INFO [flogo.engine] - Starting Services...
2020-10-28T05:07:02.837Z INFO [flogo] - ActionRunner Service: Started
2020-10-28T05:07:02.838Z INFO [flogo] - FlowTester: Started
2020-10-28T05:07:02.838Z INFO [flogo] - FlowStateRecorder: Started
2020-10-28T05:07:02.838Z INFO [flogo.engine] - Started Services
2020-10-28T05:07:02.838Z INFO [flogo.engine] - Starting Application...
```

Handling errors

If the activity encounters an error, a red error icon is displayed on the activity and also on the **Error handler** tab (if the error handler is run in the background). You can click the **Error handler** tab to find out till where the execution took place successfully. Note that when you navigate back to the Main flow, the red error icon is not displayed on the **Error handler** tab.



```
2020-10-28T05:10:11.350Z INFO [flogo] - FlowTester: Started
2020-10-28T05:10:11.350Z INFO [flogo.engine] - Started Services
2020-10-28T05:10:11.350Z INFO [flogo.engine] - Starting Application...
2020-10-28T05:10:11.350Z INFO [flogo] - Starting Connection Managers...
2020-10-28T05:10:11.350Z INFO [flogo] - Connection Managers Started
2020-10-28T05:10:11.350Z INFO [flogo.engine] - Application Started
2020-10-28T05:10:11.350Z INFO [flogo.engine] - Engine Started
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Runtime started in 3.994098ms
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

- If you start the execution from the **Error handler** tab, execution is moved on to the Main flow tab (as an error handler is always a part of the main flow).

  You can, however, start a test from a tile in the **Error Handler** tab. In this case, the execution starts from the **Error Handler** tab.

- If the execution is started from a sub-flow, the execution does not move to the Main flow and acts as a normal tile execution (as a sub-flow is an independent flow).

Executing the flow from a specific activity onwards

You can debug a specific activity in the **Main flow** or **Error Handler** flow. If it is successful, the output is shown in the **Output** tab. If an activity does not have any output (for example, the Return activity), it shows the **Output** tab as blank. If the activity is erroneous, the error is shown in the **Errors** tab.

To execute the flow from a specific activity (and not from the beginning of the flow) with different input data, perform the following steps.

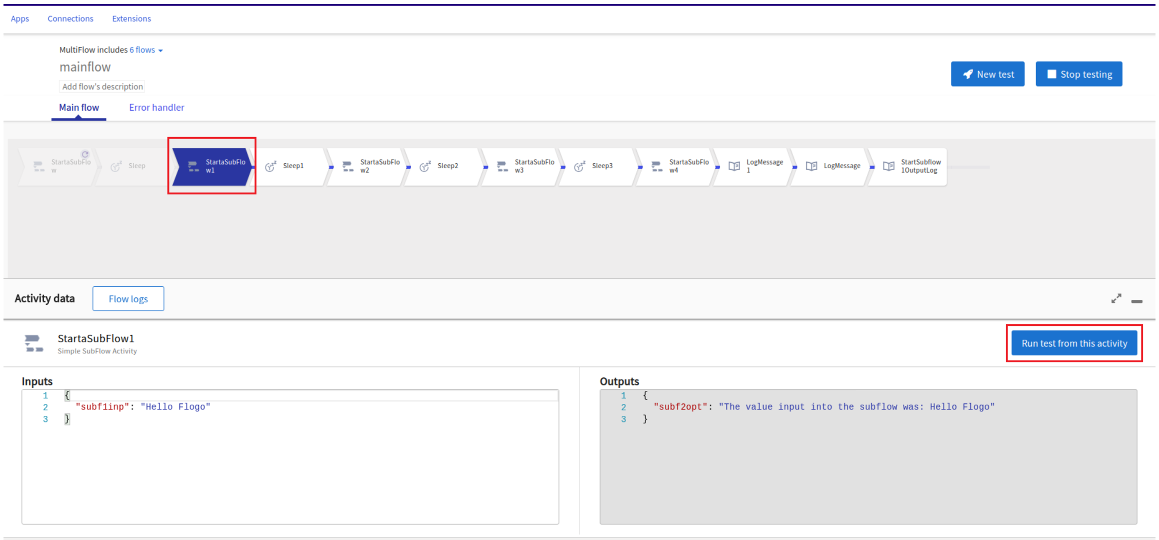
- This can be done only after the entire flow has been executed at least once.

- When you start the execution of a flow from a specific activity in the flow, you cannot start the execution again from any activity prior to the current activity. If you need to do that, you must launch a new test.

  For example, a flow includes A1 -> A2 -> A3 -> A4 -> A5 activities and execution is started from the A3 activity onwards. In subsequent executions, you cannot start the execution from any activity prior to A3; execution always starts from A3 onwards. If you want to run the flow from an activity prior to A3, you must launch a new test.

1. Select the activity from which you want to run the flow.

   The activity is highlighted in blue. The activity data is displayed in the **Inputs** and **Outputs** tab. If an error is returned, an **Error** tab is displayed in place of the **Outputs** tab.



Inputs

```
{
  "subf1inp": "Hello Flogo"
}
```

Outputs

```
{
  "subf2opt": "The value input into the subflow was: Hello Flogo"
}
```

2. In the **Inputs** tab, change the input values as required. You can do dynamic mappings here.
3. Click **Run test from this activity**.

The execution begins from the current activity. The logs are also displayed only for the current activity and subsequent activities in the flow.

> Once the execution starts from a tile, you cannot access preceding tasks executed in the previous runs. The previous activities are greyed out. If you want to run the flow from a previous activity, you must launch a new test.

Logging information

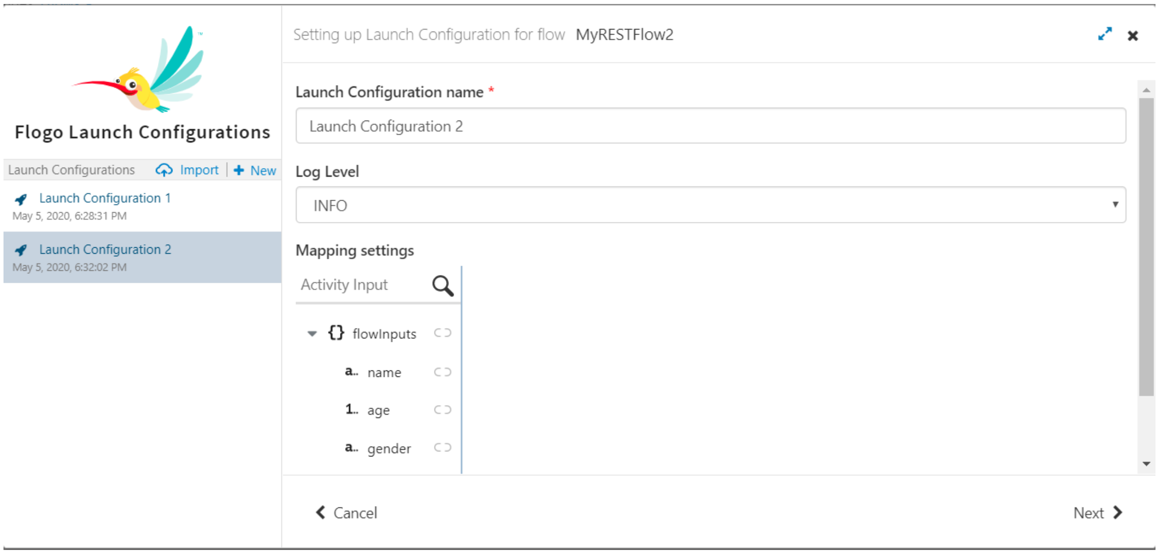
As the activities are executed, the runtime engine logs for the activities are displayed in the Logs output window. The format of the logs is similar to the logs displayed while running an app binary.

To copy these logs, you can click **Copy logs**.

You can also switch from the **Flow logs** view to the activity data view by clicking **Activity data**.

Configuring a Launch Configuration

When you click a Launch Configuration name, its mapper opens to its right. The mapper displays the input tree in the left pane.



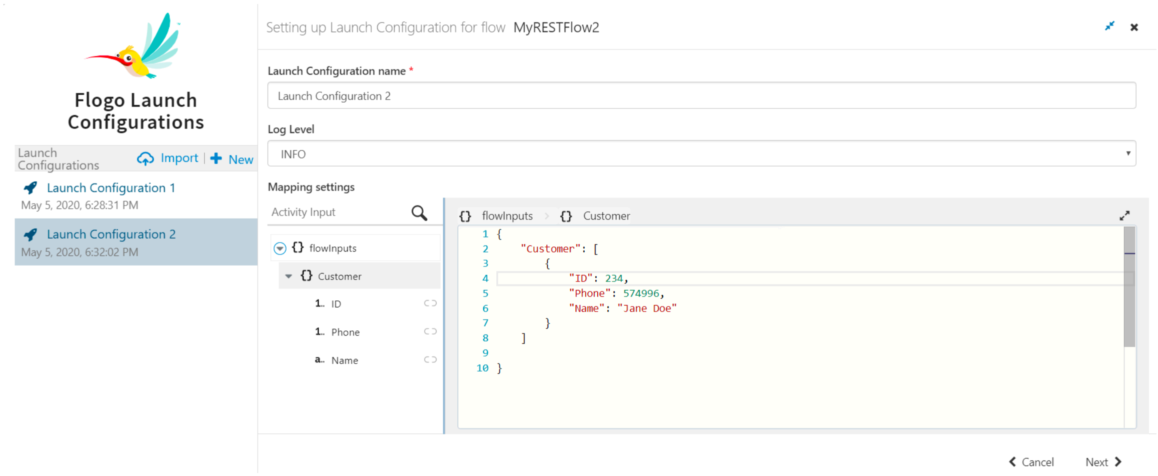
Launch Configuration name *: Launch Configuration 2

Log Level: INFO

Procedure

1. Expand the input tree.
2. Click an element to add a value for the element.
3. Enter the value for that element in the text box to its right.
   When entering values for the elements, be aware of the following:

   - The input tree for a Launch Configuration mapper displays the input you configured in the **Flow Inputs & Outputs** accordian tab for blank flows. For flows created with a trigger, it displays the output schema of the trigger.
   - For flow inputs that contain only single objects, you must enter the input values at the root level. The example below shows how to enter the values for a single object, `Customer`:



Setting up Launch Configuration for flow MyRESTFlow2

Launch Configuration name *: Launch Configuration 2

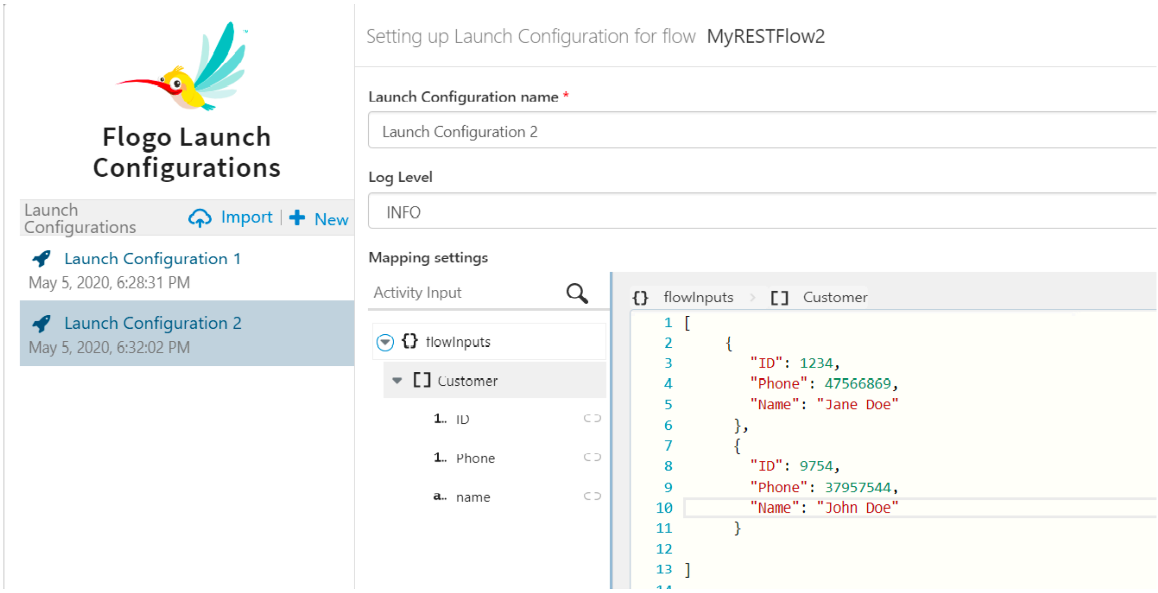
Log Level: INFO

Mapping settings

```
{
    "Customer": [
        {
            "ID": 234,
            "Phone": 574996,
            "Name": "Jane Doe"
        }
    ]

}
```

   - When mapping an array of objects, the input must be provided at the array root level, which means that you must provide input for all objects in the array by clicking on the root of the array. You cannot configure the input at the array element level. In the example below, `Customer` is an array of objects. Each object within the `Customer` array contains `ID`, `Phone`, and `Name` elements. When providing values for `Customer`, you cannot give the input at the element (`ID`, `Phone`, or `Name`) level. Doing so does not specify the index of the `Customer` object for which you are assigning the value(s). Hence, you must assign the value to the whole `Customer` object. Since the `Customer` array has multiple objects, assign values to each object in the `Customer` array by separating the objects with a comma delimiter. The array size will be determined based on the number of objects for which you provide values. In the example below, the array size is two since there are two objects for which values have been provided.



Setting up Launch Configuration for flow MyRESTFlow2

Launch Configuration name *: Launch Configuration 2

Log Level: INFO

Mapping settings

```
[
    {
        "ID": 1234,
        "Phone": 47566869,
        "Name": "Jane Doe"
    },
    {
        "ID": 9754,
        "Phone": 37957544,
        "Name": "John Doe"
    }

]
```

4. Click **Next**.
   The mapper performs validations to ensure the validity of the JSON structure and also validates that you have entered values for all elements that are marked as required in the schema. If there are any errors in your input, when you click **Next**, the mapper displays a list of errors.

   In your test environment, only the validation errors related to invalid JSON structure will prevent you from proceeding with your testing. Errors pertaining to missing values for required elements serve as a warning, but allow you to proceed with your testing. This is because it is possible that an element that is marked as a required field in the schema may not have been used in the activity at the time of testing. In that case, the element is not needed for the flow to run. But in the production environment, your app will not run successfully until you provide input values for all elements marked as required in your schema.

Exporting a Launch Configuration

There may be occasions when you want to use the same test data configurations for testing multiple flows. You have the option to create a Launch Configuration that contains this data in one flow, export the Launch Configuration, then import it into each of the other flows. The ability to export a Launch Configuration is particularly useful when the data set is very complex. In such a scenario, you can export a Launch Configuration, import it into another flow and test the flow with the imported Launch Configuration. Reusing a Launch Configuration by exporting and importing it saves you the time and effort needed to create a separate Launch Configuration for each flow.
To export a Launch Configuration, follow these steps:

Procedure

1. In the Flow Tester, hover your mouse cursor to the extreme right of the Launch Configuration name that you want to export.

2. Click the **Export Launch Configuration** (⬇) icon.
   A file with the name *<flow-name>_<Launch Configuration-name>*`.json` gets downloaded to your `Downloads` directory. You can import this file into another flow and use the Launch Configuration that you just exported. Refer to Importing a Launch Configuration for details on how to import a Launch Configuration.

The Launch Configuration name is not preserved, so the imported Launch Configuration is given a default name of "Launch Configuration x" where x stands for the next number in the series of existing Launch Configurations. For example, if you have two existing Launch Configurations in the flow, the imported Launch Configuration is named Launch Configuration 3. You have the option to edit the name to make it more meaningful.

Importing a Launch Configuration

Launch Configurations are stored as JSON files, so when you export a Launch Configuration, you export its JSON file. You import the Launch Configuration that was exported from another flow by importing the JSON file of the Launch Configuration into the flow.

The Launch Configuration name is not preserved, so the imported Launch Configuration is given a default name of "Launch Configuration x" where x stands for the next number in the series of existing Launch Configurations. For example, if you have two existing Launch Configurations in the flow, the imported Launch Configuration is named Launch Configuration 3. You have the option to edit the name to make it more meaningful.

To import a Launch Configuration, follow this procedure:

Prerequisites

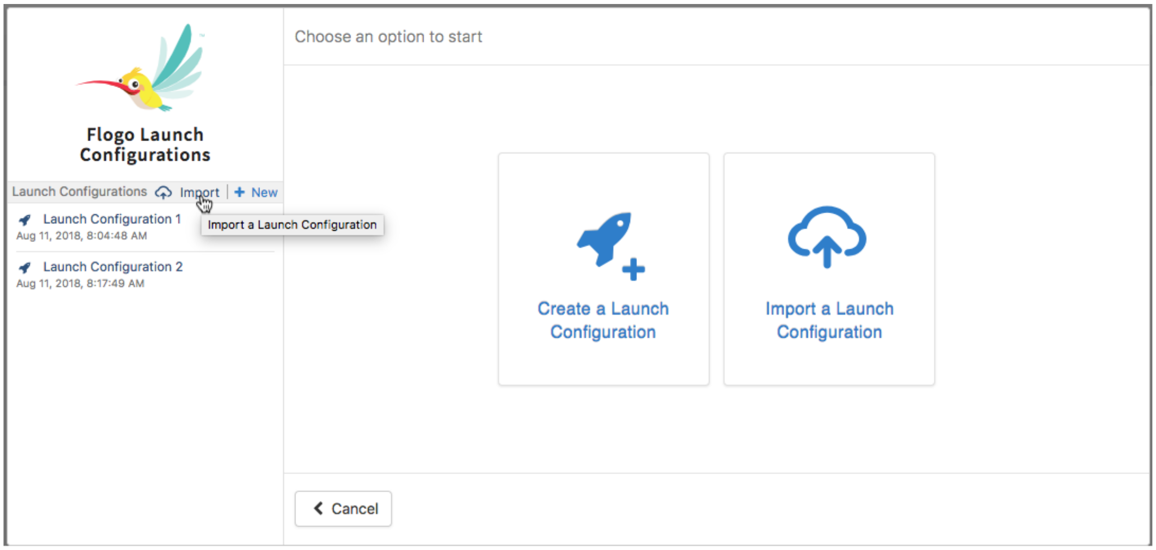
You must export the Launch Configuration you want to import and have its JSON file accessible before you follow the procedure below.

If this is the first Launch Configuration

1. If this is the first Launch Configuration in the flow (no existing Launch Configurations), click **Test** on the flow details page.
2. Click **Import a Launch Configuration**.
3. You have the option to either drag and drop the JSON file of the Launch Configuration you want to import, or navigate to the file using the **browse to upload** link.
4. Click **Import**. Data in the Launch Configuration being imported gets validated. In case there are any errors, they are displayed in the **Import** dialog.

When there are existing Launch Configurations

If there are existing Launch Configurations in the flow, click **Import** in the Flow Tester and either drag and drop the JSON file that was exported from another flow, or navigate to the file using the **browse to upload** link, then click **Import**.


Flogo Launch Configurations
Launch Configurations
Import
+ New
Import a Launch Configuration
Launch Configuration 1
Aug 11, 2018, 8:04:48 AM
Launch Configuration 2
Aug 11, 2018, 8:17:49 AM
Choose an option to start
Create a Launch Configuration
Import a Launch Configuration
Cancel

Cloning a Launch Configuration

Whereas exporting-and-importing a Launch Configuration is useful for using the same set of data in two or more flows, cloning a Launch Configuration is useful when you want to test the same flow with two sets of data that have only minor differences.

A good use case for cloning

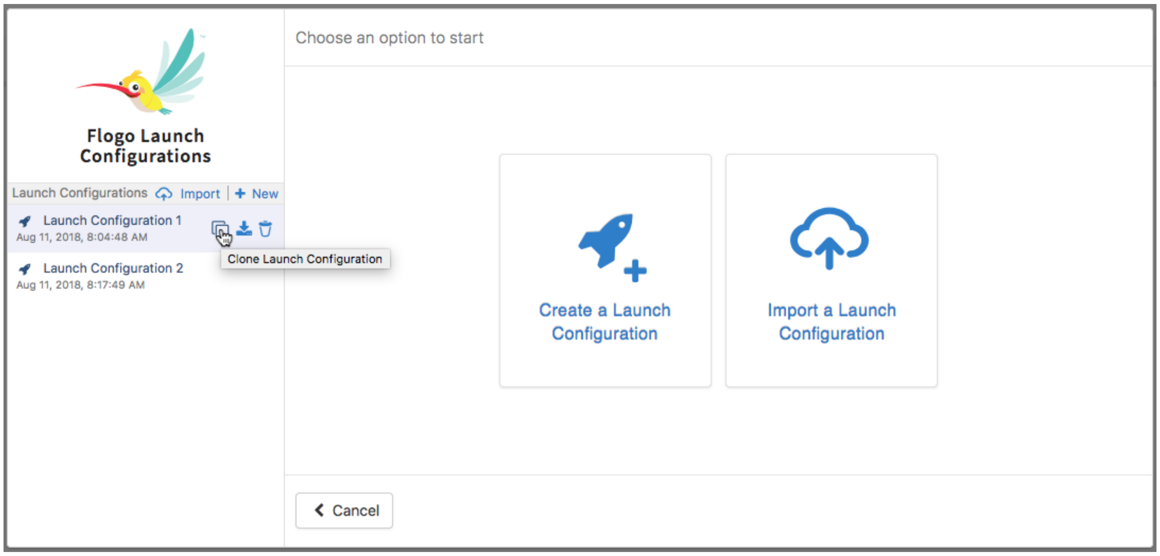
Clone a Launch Configuration when you need to test a flow multiple times using the same input schema, but different values for one or more element in the schema during each round of testing. You can start by creating a Launch Configuration, then cloning it, then editing the cloned Launch Configuration. You can create as many clones as needed. Each clone is a separate Launch Configuration having the same input schema. You can change the values for the elements in each cloned Launch Configuration as required. Use the original Launch Configuration for one round of testing and the cloned Launch Configuration(s) for the subsequent round(s). This saves you the effort of editing a single Launch Configuration.

To clone an existing Launch Configuration, follow these steps:

Procedure

1. In the Flow Tester, hover your mouse cursor to the extreme right of the Launch Configuration name that you want to clone.

2. Click the **Clone Launch Configuration** (⧉) icon. The cloned Launch Configuration will be named `Copy` *`<name-of-the-original-Launch Configuration>`* by default. You can edit the name of the Launch Configuration in the **Launch Configuration name** text box.


Flogo Launch Configurations
Launch Configurations
Import
+ New
Launch Configuration 1
Aug 11, 2018, 8:04:48 AM
Clone Launch Configuration
Launch Configuration 2
Aug 11, 2018, 8:17:49 AM
Choose an option to start
Create a Launch Configuration
Import a Launch Configuration
Cancel

Deleting a Launch Configuration

When you create a Launch Configuration, it automatically gets saved until you explicitly delete it. To delete a Launch Configuration, follow these steps:

Procedure

1. In the Flow Tester, hover your mouse cursor to the extreme right of the Launch Configuration name that you want to delete.

2. Click the **Delete Launch Configuration** (🗑) icon.

Deployment and Configuration

After you have created and validated your app, you must build the app binary. You can optionally push the app to TIBCO Cloud Integration.

Building an App Binary

This section instructs you on how to build an app binary.

Building the App

After you have created your app, you have the option to either export the app (without building it) or build it. Exporting an app allows you to import it elsewhere, for example in TIBCO Cloud™ Integration - Flogo®. When you build the app, its deployable artifact gets created and downloaded to your local machine. Each operating system has its own build target. You must select the right target for your operating system when building the app. You can use the built artifact to run the app in TIBCO Cloud Integration - Flogo (PAYG).

> Be sure that you have Docker installed on your machine. Refer to the product Readme for the supported versions of Docker.

Follow these steps to build an app:

> For app binaries that were created in TIBCO Cloud Integration - Flogo (PAYG) 2.5 or older versions, if the app binary was created using an `<app>.json` file and contains a flow starting with a trigger and the app binary was created from the CLI using the build tool, the app gets successfully built but throws an error at runtime.

Procedure

1. Open the **Apps** page in TIBCO Cloud Integration - Flogo (PAYG).
2. Click the app that you want to build to open the page for that app.
3. Click the **Validate** button to explicitly validate the flows and activities in the app. Fix any errors before pushing the app.
4. Click **Build**.
5. Click the build target option that is compatible with your operating system.
   The app begins to build. When done, the deployable artifact gets downloaded to your local machine. In the case of Docker, a Docker image gets created in your Docker storage area.

   > Any uppercase letters in your app name get converted to lowercase in the Docker image name. For instance, if your app is named `MyApp`, the Docker image that gets generated will be named `myapp`.

6. To run the app, follow these steps:
   On Linux

   1. Open a terminal.
   2. Run: `chmod +x <app-file-name>`
   3. Run: `./<app-file-name>`

   For Docker Image

   > Any uppercase letters in your app name get converted to lowercase in the Docker image name. For instance, if your app is named `MyApp`, the Docker image that gets generated will be named `myapp`. So be sure to use all lowercase letters in the *app-file-name* in the command below.

1. Open a terminal.
2. Run: `docker run -p <<host-port-number>>:<port-on-docker> flogo/<app-file-name>`

Environment Variables

This section lists the environment variables that are associated with the TIBCO Cloud Integration - Flogo (PAYG) runtime environment.

Environment Variable Name	Default Values	Description
FLOGO_RUNNER_QUEUE	50	The maximum number of events from all triggers that can be queued by the app engine.
FLOGO_RUNNER_WORKERS	5	The maximum number of concurrent events that can be executed by the app engine from the queue.
FLOGO_HTTP_SERVICE_PORT	N/A	Used to set the port number to enable runtime HTTP service which provides APIs for healthcheck and statistics.
FLOGO_LOG_LEVEL	INFO	Used to set a log level for the Flogo App. Supported values are: • INFO • DEBUG • WARN • ERROR
FLOGO_LOG_FORMAT	TEXT	Used to switch logging format between text and JSON. For example, to use the JSON format, set FLOGO_LOG_FORMAT=JSON ./<*app-name*>
FLOGO_MAPPING_SKIP_MISSING	False	When mapping objects, if one or more elements is missing in either the source or target object, the mapper throws an error when `FLOGO_MAPPING_SKIP_MISSING` is set to `false`. Set this environment variable to `true`, if you would like to return a null instead of receiving an error.
FLOGO_APP_METRICS_LOG_EMITTER_ENABLE	False	If you set this property to `True`, the app metrics are displayed in the logs with the values set in FLOGO_APP_METRICS_LOG_EMMITTER_CONFIG. App metrics are not displayed in the logs if this environment variable is set to `False`. To set it to `True`, run: `export FLOGO_APP_METRICS_LOG_EMITTER_ENABLE=true`

Environment Variable Name	Default Values	Description

FLOGO_APP_METRICS_LOG_EMITTER_CONFIG	Both flow and activity	This property can be set to either flow level or activity level. Depending on which level you set, the app metrics will display only for that level. Also, you can provide an (interval in seconds) at which to display the app metrics.
		For example to set the interval to 30 seconds and get the app metrics for flow, run:
		`export FLOGO_APP_METRICS_LOG_EMITTER_CONFIG=‘{“interval”:“30s”,“type”:[“flow”]}’`
		To set the interval for 10 seconds and get the app metrics for both flow and activities, run:
		`export FLOGO_APP_METRICS_LOG_EMITTER_CONFIG=‘{“interval”:“30s”,“type”:[“flow”, “activity”]}’`

App Configuration Management

TIBCO Cloud Integration - Flogo (PAYG) allows you to externalize app configuration using app properties, so that you can run the same app binary in different environments without making changes to your app. It integrates with configuration management systems such as Consul and AWS Systems Manager Parameter Store to get the values of app properties at runtime.

You can switch between configuration management systems without making changes to your app. You can do this by running the command to set the configuration-management-system-specific environment variable from the command line. Since the environment variables are set for the specific configuration management system, at runtime the app connects to that specific configuration management system to pull the values for the app properties.

Consul

The Consul provides a key/value store for managing app configuration externally. TIBCO Cloud Integration - Flogo (PAYG) allows you to fetch values for app properties from Consul and override them at runtime.

This document assumes that you have set up Consul and know how Consul is used to store service configuration. Refer to the Consul documentation for Consul specific information.

A Flogo app connects to the Consul server as its client by setting the environment variable, `FLOG_APPS_PROPS_CONSUL`. At runtime, you must provide the Consul server endpoint in order for your app to connect to a Consul server. You have the option to enter the values for the Consul connection parameters either by typing in their values as JSON strings, or creating a file that contains the values and using the file as input.

Consul can be started with or without `acl_token`. If using ACL token, make sure to have ACL configured in Consul.

Using Consul with TIBCO Cloud Integration - Flogo (PAYG)

Below is a high-level workflow for using Consul with your Flogo app.

Prerequisites

You must have access to Consul before following this procedure. This document assumes that you have set up Consul and know how Consul is used to store service configuration. For information on Consul, refer to the Consul documentation.

At a high level, to use Consul to override app properties in your app (properties that were set in TIBCO Cloud Integration - Flogo (PAYG)), do the following:

Procedure

1. Export your app binary from TIBCO Cloud Integration - Flogo (PAYG). Refer to Exporting and Importing an App for details on how to export the app.
2. Configure key/value pairs in Consul for the app properties whose values you want to override. At runtime, the app fetches these values from the Consul and uses them to replace their default values that were set in the app.

   > When setting up the Key in Consul, make sure that the Key name matches exactly with the corresponding app property name in the **Application Properties** dialog in TIBCO Cloud Integration - Flogo (PAYG). If the property name does not match exactly, you will receive a warning message and the app will use the default value for the property that you configured in TIBCO Cloud Integration - Flogo (PAYG).

3. Set the `FLOGO_APP_PROPS_CONSUL` environment variable to set the Consul server connection parameters. See Setting the Consul Connection Parameters for details.

Consul Connection Parameters

Provide the following configuration information during runtime to connect to the Consul server.

Property Name	Required	Description
server_address	Yes	Address of the Consul server which could be run locally or elsewhere in the cloud.

Property Name	Required	Description
key_prefix	No	Prefix to be prepended to the lookup key. This is essentially the hierarchy that your app follows to get to the Key location in the Consul. This is helpful in case key hierarchy is not fixed and may change based on the environment during runtime. It is also helpful in case you want to switch to a different configuration service such as AWS param store. Although it is a good idea to include the app name in the `key_prefix`, it is not required. `key_prefix` can be any hierarchy that is meaningful to you. As an example of a `key_prefix`, if you have an app property (for example, `Message`) which has two different values depending on the environment from which it is being accessed (for example, `dev` or `test` environment), your `<key_prefix>` for the two values can be `/dev/<APPNAME>/` and `/test/<APPNAME>/`. At run time, the right value for `Message` will be picked up depending on which `<key_prefix>` you specify in the `FLOGO_APP_PROPS_CONSUL` environment variable. Hence, setting a `<key_prefix>` allows you to change the values of the app properties at runtime without modifying your app.

Property Name	Required	Description
acl_token	No	Use this parameter if you have key access protected by ACL. Tokens specify which keys can be accessed from the Consul. You create the token on the ACL tab in Consul. During runtime, if you use the `acl_token` parameter, Key access to your app will be based on the token you specify. To protect the token, encrypt the token for the `key_prefix` where your Key resides and provide the encrypted value of that token by prefixing the `acl_token` parameter with SECRET. For example, `"acl_token": "SECRET:QZLOrtN3gOEpXgUuud6jprgo/WzLR7j+Twv28/4KCp7573snZWo+hGuQauuR2o/7TJ+ZLQ=="`. Note that the encrypted value follows the `key_prefix` format. Provide the encrypted value of the token as the SECRET. SECRETS get decrypted at runtime. To encrypt the token, you obtain the token from the Consul and encrypt it using the app binary by running the following command from the directory in which your app binary is located: `./<app_binary> --encryptsecret <token_copied_from_Consul>` The command outputs the encrypted token which you can use as the SECRET. Since special characters (such as `! | < > & `) are shell command directives, if they appear in the token string, when encrypting the token, you must use a back slash (\) to escape such characters.
insecure_connection	No	By default, set to `False`. Set to `True` if you want to connect to a secure Consul server without specifying client certificates. This should only be used in test environments.

Setting the Consul Connection Parameters

You set the values for app properties that you want to override by creating a Key/Value pair for each property in Consul. You can create a standalone property or a hierarchy that groups multiple related properties.

Prerequisites

This document assumes that you have access to Consul and are familiar with its use.

To create a standalone property (without hierarchy), you simply enter the property name as the name of the Key when creating the Key in Consul. To create a property within a hierarchy enter the hierarchy in the following format in the **Create Key** field in Consul: `<key_prefix>/<key_name>` where `<key_prefix>` is a meaningful string or hierarchy that serves as a path to the key in Consul and `<key_name>` is the name of the app property whose value you want to override. For example, in `dev/Timer/Message` and `test/Timer/Message`, `dev/Timer` and `test/Timer` are the `<key_prefix>` which could stand for the dev and test environments and `Message` is the key name. During runtime, you provide the `<key_prefix>` value which tells your app the location in Consul from where to access the property values.

The Key name in Consule must be identical to its counterpart in the **Application Properties** dialog in TIBCO Cloud Integration - Flogo (PAYG). If the key name does not match exactly, you will receive a warning message and the app will use the default value that you configured for the property in TIBCO Cloud Integration - Flogo (PAYG).

A single app property, for example `Message`, will be looked up by your app as either `Message` or `<key_prefix>/Message` in Consul. An app property within a hierarchy such as `x.y.z` will be looked up as `x/y/z` or `<key_prefix>/x/y/z` in Consul. Note that the dot in the hierarchy is represented by a forward slash (/) in Consul.

After you have configured the app properties in Consul, you need to set the environment variable, `FLOGO_APP_PROPS_CONSUL`, with the Consul connection parameters in order for your app to connect to the Consul. When you set the environment variable, it triggers the app to run, which connects to the Consul using the Consul connection parameters you provided and pulls the app property values from the `key_prefix` location you set by matching the app property name with the `key_name`. Hence, it is mandatory for the Key names to be identical to the app property names defined in the **Application Properties** dialog in TIBCO Cloud Integration - Flogo (PAYG).

You can set the `FLOGO_APP_PROPS_CONSUL` environment variable either by directly entering the values as a JSON string on the command line or placing the properties in a file and using the file as input to the `FLOGO_APP_PROPS_CONSUL` environment variable.

Entering the Consul Parameter Values as a JSON String

To enter the Consul parameters as a JSON string, enter the parameters as key/value pairs using the comma delimiter. The following examples illustrate how to set the values as JSON strings. You would run the following from the location where your app resides:

An example when not using security without tokens enabled:

```
FLOGO_APP_PROPS_CONSUL="{\"server_address\":\"http:\/\/127.0.0.1:8500\"}" ./Timer-
darwin-amd64
```

where `Timer-darwin-amd64` is the name of the app binary.

An example when tokens are enabled and app properties are within a hierarchy:

```
FLOGO_APP_PROPS_CONSUL="{"server_address":"http://127.0.0.1:8500","key_prefix":"/dev/
Timer","acl_token":"SECRET:b0UaK3bTyD9wN+ZJkmlKRmojhAv+"}"
```

where `/dev/Timer` is the path and `SECRET` is the encrypted value of the token obtained from the Consul.

This command directs your app to connect to the Consul at the `server_address` and pull the values for the properties from the Consul and run your app with those values.

Refer to Consul Connection Parameters section for a description of the parameters. Refer to Encrypting Password Values for details on how to encrypt a value.

Setting the Consul Parameter Values Using a File

To set the parameter values in a file, create a `.json` file, for example, `consul_config.json` containing the parameter values in key/value pairs. Here's an example:

```
{
    "server_address": "http://127.0.0.1:32819",
    "key_prefix": "/dev/<APPNAME>/",
    "acl_token": "SECRET:b0UaK3bTyD9wN+ZJkmlKRmojhAv+"
}
```

You would place the `consul_config.json` file in the same directory which contains your app binary.

You would then run the following from the location where your app binary resides to set the FLOGO_APP_PROPS_CONSUL environment variable. For example, to use the consul_config.json file from the example above, you would run:

```
FLOGO_APP_PROPS_CONSUL=consul_config.json ./<app_binary_name>
```

The command extracts the Consul server connection parameters from the file and connects to the Consul to pull the app properties values from the Consul and run your app with those values.

Consul properties can also be run using docker by passing the same arguments for the docker image of a binary app.

AWS Systems Manager Parameter Store

AWS Systems Manager Parameter Store is a capability provided by AWS Systems Manager for managing configuration data. You can use the Parameter Store to centrally store configuration parameters for your apps.

Your Flogo app connects to the AWS Systems Manager Parameter Store server as its client. At runtime, you are required to provide the Parameter Store server connection details by setting the FLOGO_APP_PROPS_AWS environment variable in order for your app to connect to the Parameter Store server. You have the option to enter the values for the Parameter Store connection parameters either by typing in their values as JSON strings, or creating a file that contains the values and using the file as input.

Using the Parameter Store with TIBCO Cloud Integration - Flogo (PAYG)

Below is a high-level workflow for using AWS Systems Manager Parameter Store with your Flogo app.

Prerequisites

This document assumes that you have an AWS account, have access to the AWS Systems Manager and know how to use the AWS Systems Manager Parameter Store. Refer to the AWS documentation for the information on the AWS Systems Manager Parameter Store.

Overview

To use the Parameter Store to override app properties set in TIBCO Cloud Integration - Flogo (PAYG), do the following:

1. Build an app binary which has the app properties already configured in TIBCO Cloud Integration - Flogo (PAYG). Refer to Building the App for details on how to build the app.
2. Configure the app properties that you want to override in the Parameter Store. At runtime, the app fetches these values from the Parameter Store and uses them to replace their default values that were set in the app.
3. Set the FLOGO_APP_PROPS_AWS environment variable to set the Parameter Store connection parameters from the command line.

   When you run the command for setting the FLOGO_APP_PROPS_AWS environment variable, it runs your app, connects to the Parameter Store, and fetches the overridden values for the app properties from the Parameter Store. Only the values for properties that were configured in the Parameter Store will be overridden. The remaining app properties will get their values from the **Application Properties** dialog.

See the Setting the Parameter Store Connection Parameters and Parameter Store Connection Parameters sections for details.

Parameter Store Connection Parameters

To connect to AWS Systems Manager Parameter Store, provide the configuration below at runtime.

Property Name	Required	Data Type	Description
access_key_id	Yes	String	Access ID for your AWS account. To protect access key, an encrypted value can be provided in this configuration. See Encrypting Password Values section for information on how to encrypt a string. 📝 The encrypted value must be prefixed with SECRET: e.g. SECRET:b0UaK3bTyD9wN +ZJkmlKRmojhAv+ This configuration is optional if `use_iam_role` is set to `true`.
secret_access_key	Yes	String	Secret access key for your AWS account. This account must have access to the Parameter Store. To protect secret access key, an encrypted value can be provided in this configuration. See the Encrypting Password Values section for information on how to encrypt a string. 📝 The encrypted value must be prefixed with SECRET: for example, `SECRET:b0UaK3bTyD9wN +ZJkmlKRmojhAv+` This configuration is optional if `use_iam_role` is set to `true`.
region	Yes	String	Select a geographic area where your Parameter Store is located. This configuration is optional if `use_iam_role` is set to `true` and your Parameter Store is configured in the same region as the running service. When running in AWS services (for example, EC2, ECS, EKS), this configuration is optional if the Parameter Store is in the same region as these services.

Property Name	Required	Data Type	Description
param_prefix	No	String	This is essentially the hierarchy that your app follows to get to the app property location in the Parameter Store. It is the prefix to be prepended to the lookup parameter. This is helpful in case the parameter hierarchy is not fixed and may change based on the environment during runtime. This is also helpful in case you want to switch to a different configuration service such as the Consul KV store. As an example of a `param_prefix`, if you have an app property (for example, `Message`) which has two different values depending on the environment from which it is being accessed (for example dev or test environment), your `param_prefix` for the two values can be `/dev/<APPNAME/` and `/test/<APPNAME/`. At run time, the right value for `Message` will be picked up depending on which param_prefix you specify in the `FLOGO_APP_PROPS_AWS` environment variable. Hence, setting a param_prefix allows you to change the values of the app properties at runtime without modifying your app.
use_iam_role	No	Boolean	Set to `true` if the Flogo app is running in the AWS services (such as EC2, ECS, EKS) and you want to leverage IAM role (such as instance role or task role) to fetch parameters from the Parameter Store. In that case, `access_key_id`, and `secret_access_key` are not required.

Setting the Parameter Store Connection Parameters

You can use the AWS Systems Manager Parameter Store to override the property value set in your Flogo app. You do so by creating the property in the Parameter Store and assigning it the value with which to override the default value set in the app. You can create a standalone property or a hierarchy (group) in which your property resides.

Prerequisites

This document assumes that you have an AWS account and the Parameter Store and are familiar with its use. Refer to the AWS documentation for more information on the Parameter Store.

To create a standalone property (without hierarchy), you simply enter the property name when creating it. To create a property within a hierarchy enter the hierarchy in the following format when creating the property: *`<param_prefix>/<property_name>`*, where `<param_prefix>` is a meaningful string or hierarchy that serves as a path to the property name in Parameter Store and `<property_name>` is the name of the app property whose value you want to override. For example, in `dev/Timer/Message` and `test/Timer/Message` `/dev/Timer` and `test/Timer` are the `<param_prefix>` which could stand for the dev and test environments respectively, and `Message` is the key name. During runtime, you provide the `<param_prefix>` value which tells your app the location in Parameter Store from where to access the property values.

The parameter name in Parameter Store must be identical to its counterpart (app property) in the **Application Properties** dialog in TIBCO Cloud Integration - Flogo (PAYG). If the parameter names do not match exactly, you will receive a warning message and the app will use the default value that you configured for the property in TIBCO Cloud Integration - Flogo (PAYG).

A single app property, for example `Message`, will be looked up by your app as either `Message` or `<param_prefix>/Message` in Parameter Store. An app property within a hierarchy such as `x.y.z` will be looked up as `x/y/z` or `<param_prefix>/x/y/z` in Parameter Store. Note that the dot in the hierarchy is represented by a forward slash (/) in the Parameter Store.

After you have configured the app properties in the Parameter Store, you need to set the environment variable, `FLOGO_APP_PROPS_AWS`, with the Parameter Store connection parameters in order for your app to connect to the Parameter Store. When you set the environment variable, it triggers your app to run, which connects to the Parameter Store using the Parameter Store connection parameters you provided and pulls the app property values from the `param_prefix` location you set by matching the app property name with the `param_name`. Hence, it is mandatory for the property names to be identical to the app property names defined in the **Application Properties** dialog in TIBCO Cloud Integration - Flogo (PAYG).

You can set the `FLOGO_APP_PROPS_AWS` environment variable either by manually entering the values as a JSON string on the command line or placing the properties in a file and using the file as input to the `FLOGO_APP_PROPS_AWS` environment variable.

If your Container is Not Running on ECS or EKS

If the container in which your app resides is running external to ECS, you must enter the values for `access_key_id` and `secret_access_key` parameters when setting the `FLOGO_APP_PROPS_AWS` environment variable.

Entering the Parameter Store Values as a JSON String

To enter the Parameter Store connection parameters as a JSON string, enter the parameters and their value using the comma delimiter. The following example illustrates how to set the values as JSON strings. This would be run from the location where your app resides:

```
FLOGO_APP_PROPS_AWS="{"access_key_id":"SECRET:XXXXXXXXXXXXX","secret_access_key":"SECRE
T:XXXXXXXXXXX","region":"us-west-2","param_prefix":"/MyFlogoApp/Dev/"}"
```

where `/MyFlogoApp/Dev/` is the param_prefix (path to the properties) and `SECRET` is the encrypted version of the key or key_id obtained from the Parameter Store.

This will connect to the Parameter Store and pull the values for the properties and override their default values that were set in the app.

Refer to Parameter Store Connection Parameters section for a description of the parameters.

Setting the Parameter Store Values Using a File

To set the parameter values in a file, create a `.json` file, for example, `aws_config.json` containing the parameter values. Here's an example:

```
{
    "access_key_id": "SECRET:b0UaK3bTyD9wN+ZJkmlKRmojhAv+",
    "param_prefix": "/MyFlogoApp/dev/",
    "secret_access_key": "SECRET:b0UaK3bTyD9wN+ZJkmlKRmojhAv+",
    "region": "us-west-2",
}
```

Place the `aws_config.json` file in the same directory which contains your app binary.

You would then run the following from the location where your app binary resides to set the `FLOGO_APP_PROPS_AWS` environment variable. For example, to use the `aws_config.json` file from the example above, run:

```
FLOGO_APP_PROPS_AWS=aws_config.json ./<app_binary_name>
```

This will connect to the Parameter Store to pull the overridden app properties values from the Parameter Store and run your app with those values.

If your Container is Running on ECS or EKS

In case your Flogo apps are running in ECS and intend to leverage the EC2 instance credentials, set `use_iam_role` to `true` . The values for `access_key_id` and `secret_access_key` will be gathered from the running container. Ensure that the ECS task has the permission to access the param store.

The IAM role that you use must have permissions to access parameter(s) from the AWS Systems Manager Parameter Store. The following policy must be configured for IAM role:

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Action":[
            "ssm:GetParamaters",
            "ssm:GetParamatersByPath",
         ],
         "Effect":"Allow",
         "Resource":"*"
      }
   ]
}
```

The following is an example of how to set the `FLOGO_APP_PROPS_AWS` environment variable when your container is running on ECS. Notice that the values for `access_key_id` and `secret_access_key` are omitted:

```
FLOGO_APP_PROPS_AWS="{\"use_iam_role\":true, \"region\":\"us-west-2\"}" ./Timer-darwin-
amd64
```

Environment Variables

TIBCO Cloud Integration - Flogo (PAYG) allows you to externalize the configuration of app properties using environment variables.

Using environment variables with app properties is a two-step process:

1. Create one environment variable per app property.
2. Set the `FLOGO_APP_PROPS_ENV=auto` environment variable, which directs it to fetch the values of the app properties for which you have created environment variables.

App binaries that were generated from a version of TIBCO Cloud Integration - Flogo (PAYG) older than 2.4.0 do not support app properties override using environment variables. For example, if you attempt to run an older app binary from TIBCO Cloud Integration - Flogo (PAYG) 2.4.0 (which supports the environment variable functionality) and override app properties in the app using environment variables, the binary runs normally but the app property override gets ignored.

Using Environment Variables to Override App Property Values

The use of environment variables to assign new values to your app properties at runtime is a handy method that you can use to test your app with multiple data sets.

Follow these two steps to set up the environment variables and use them during app runtime.

Step 1: Create environment variables for your app properties

You start by creating one environment variable for each app property that you want to externalize. To do so, run:

```
export <app-property-name>="<value>"
```

For example, if your app property name is `username`, run `export username="abc@xyz.com"` or `export USERNAME="abc@xyz.com"`

A few things to note about this command:

- Since special characters (such as `! | &lt; &gt; &amp;@ `) are shell command directives, if they appear in *value*, enclosing the *value* in double quotes tells the system to treat such characters as literal values instead of shell command directives.
- The *app-property-name* must match the app property exactly or it can use all uppercase letters. For example, the app property, `Message`, can either be entered as `Message` or `MESSAGE`, but not as `message`.
- If you want to use a hierarchy for your app property, be sure to use underscores (_) between each level instead of the dot notation. For example, for an app property named `x.y.z`, the environment variable name should be either `x_y_z` or `X_Y_Z`.

Step 2: Set `FLOGO_APP_PROPS_ENV=auto` environment variable

To use the environment variables during app runtime, set the `FLOGO_APP_PROPS_ENV=auto` environment variable.

To do so, run:

```
FLOGO_APP_PROPS_ENV=auto ./<app-binary>
```

For example, `FLOGO_APP_PROPS_ENV=auto MESSAGE="This is variable 1." LOGLEVEL=DEBUG ./Timer-darwin-amd64`

When setting variables of type `password` be sure to encrypt its value for security reasons. See the section, Encrypting Password Values, for more details.

Setting the `FLOGO_APP_PROPS_ENV=auto` directs your app to search the list of environment variables for each app property by matching the environment variable name to the app property name. When it finds a matching environment variable for a property, the app pulls the value for the property from the environment variable and runs the app with those values. Hence, it is mandatory that the app property name exactly match the environment variable name for the property.

App properties that were not set as environment variables will pick up the default values set for them in the app. You will see a warning message similar to the following in the output: `<property_name> could not be resolved. Using default values.`

Encrypting Password Values

When entering passwords on command line or in a file, it is always a good idea to encrypt their values for security reasons. TIBCO Cloud Integration - Flogo (PAYG) has a utility that you can use to encrypt passwords.

Prerequisites

You must have the password to be encrypted handy in order to run the utility.

To encrypt a password, run the following:

Procedure

1. Open a command prompt or a terminal.

2. Navigate to the location of the app binary and run the following command:

   ```
   ./<app_binary> --encryptsecret <value_to_be_encrypted>
   ```

   The command outputs the encrypted value which you can use when setting the password in a file or setting the password from the command line or using environment variables. For example, `export PASSWORD="SECRET:t90Ixj+QYCMFbqCEo/UnELlPPhrClMzv"`.

   Note that the password value is enclosed in double quotes. Since special characters (such as `! | &lt; &gt; &amp; `) are shell command directives, if such characters appear in the encrypted string, using double quotes around the encrypted value will direct your system to treat them as literal characters. Also, the encrypted value must be preceded by `SECRET:`

   Keep in mind that when you run the `env` command to list the environment variables, the command does not output the environment variable for the password.

Container Deployments for AWS Marketplace

Once your app is ready, you must build a docker image for the app which you then upload to AWS.

Build the Flogo Application Docker Image

Build the application docker image by extending the runtime base docker image and copying the app binary into it.
To do this, follow the steps below:

Procedure

1. Tag the Flogo Runtime Docker Image. For example, use **`flogo-runtime:<version>`**
2. Create a Docker file in the directory containing the Flogo app binary and add the following commands to the Docker file.

   ```
   FROM <FLOGO_RUNTIME_IMAGE>:<TAG>
   ADD <FLOGO_APP_BINARY_NAME> flogoapp
   ```

   You must configure the application binary as `flogoapp` before you can run it.

   For example:

   ```
   FROM flogo-runtime:latest
   ADD hello_world-linux_amd64 flogoapp
   ```

   ```
   FROM flogo-runtime:2.7.0
   ADD hello_world-linux_amd64 flogoapp
   ```
3. Run the docker build command to build the app docker image.

   ```
   docker build -t <APP_IMAGE_NAME>:<TAG>
   ```
4. Tag the image using the Docker tag command:

   ```
   docker tag application_name:latest
   <AWS_account_id>.dkr.ecr.<region_name>.amazonaws.com/application_name:latest
   ```
5. Push the app docker image to the newly created AWS repository using the Docker push command:

   ```
   docker push <AWS_account_id>.dkr.ecr.<region_name>.amazonaws.com/
   application_name:latest
   ```

   For more information about ECR, see the ECR documentation.
6. Run the app container in ECS using Fargate.

   a) Download the following deployment templates:

   - **Set up a Fargate Stack for Flogo app deployment**
   - **Run Flogo app on Fargate**

For more information about deployment templates, refer to About AWS Deployment Templates.

b) On AWS, navigate to **CloudFormation** in **Services**.

c) Using the **Create Stack** option, create a stack for the **Set up a Fargate Stack for Flogo app deployment** template. Upload the **Set up a Fargate Stack for Flogo app deployment** template. Provide a unique name for the stack, continue clicking **Next**, and then click **Create Stack**.

d) Using the **Create Stack** option, create another stack for the **Run Flogo app on Fargate** template. Upload the **Run Flogo app on Fargate** template. Provide a unique name for the stack in the **Stack name** field. Provide the ECR image URL in the **ImageURL** field. In the **StackName** field, use the same name that was used in step c (for **Set up a Fargate Stack for Flogo app deployment** template). Continue clicking **Next** and click **Create Stack**.

e) On AWS, navigate to **ECS** in **Services**. The cluster you created is displayed.

f) Navigate to the service in the cluster. After the service is successfully running, you can access the load balancer URL.

7. For ECS and ECR, while configuring the task definition, set and attach the following policy to the IAM role configured for the task. Otherwise, the app will not start:

```
{
  "Version":"2012-10-17",
  "Statement":[
     {
       "Action":[
         "aws-marketplace:RegisterUsage"
      ],
       "Effect":"Allow",
       "Resource":"*"
      }
   ]
}
```

See the Amazon website for more information regarding the use and licensing of the Amazon Linux Docker Image.

About AWS Deployment Templates

TIBCO provides templates to jumpstart your development and let you get started quickly.

Templates for ECS and Fargate

Under **Container Images > Deployment template**, click **Set up a Fargate Stack for Flogo app deployment** to download the template and run it automatically to set up the application for you.

Example: Deploying a Flogo App on AWS EKS

This example guides you through deploying a Flogo app on Amazon Elastic Kubernetes Service (Amazon EKS).

Prerequisites

- Build a Docker image of the Flogo app which you want to upload to the AWS ECR repository. For more information, refer to Build the Flogo App Docker Image.
- Create an AWS account.
- Create an AWS ECR Repository. Push the Flogo app image from the local repository to the AWS ECR repository.
- Install the latest version of AWS CLI using the following command:

```
pip install awscli
```

- Configure the AWS CLI credentials using the following command:

```
aws configure
```

- Install eksctl, a command-line utility for creating and managing Kubernetes clusters on Amazon EKS. For example, on Linux, use the following command:

  ```
  curl --silent --location
  "https://github.com/weaveworks/eksctl/releases/download/latest_release/eksctl_$(uname
  -s)_amd64.tar.gz" | tar xz -C /tmp
  ```

  ```
  sudo mv /tmp/eksctl /usr/local/bin
  ```

  ```
  eksctl version
  ```

- Install and configure kubectl for Amazon EKS. Kubernetes uses the kubectl command-line utility for communicating with the cluster API server.

For more information on installing and configuring eksctl and kubectl, refer to the section "Getting Started with eksctleksctl" in the Amazon EKS documentation.

Deploying a Flogo App on AWS EKS

Procedure

1. Create a new EKS cluster using the following command:

   ```
   eksctl create cluster <cluster name>
   ```

   For example:

   ```
   eksctl create cluster flogoDemo
   ```

   By default, the kubeconfig configuration file is created at the default kubeconfig path (.kube/config) in your home directory or merged with an existing kubeconfig at that location.

2. Set up the OIDC ID provider (IdP) in AWS. This enables the IAM role for service accounts on EKS cluster:

   ```
   eksctl utils associate-iam-oidc-provider --name <cluster name> --approve
   ```

   For example:

   ```
   eksctl utils associate-iam-oidc-provider --name flogoDemo --approve
   ```

3. Create a new policy which allows you to access to AWS Marketplace Metering.

   The JSON policy looks as follows:

   ```
   {
       "Version": "2012-10-17",
       "Statement": [
           {
               "Action": [
                   "aws-marketplace:RegisterUsage"
               ],
               "Effect": "Allow",
               "Resource": "*"
           }
       ]
   }
   ```

4. Add this policy to the role created in step # 1.

5. Create a Kubernetes service account, set up the IAM role that defines access to the targeted services (such as AWS Marketplace Metering), specify the IAM trust policy that allows the specified Kubernetes service account to assume the IAM role:

   ```
   eksctl create iamserviceaccount --name
   <Name of service account> --namespace default --cluster <cluster
   name> --attach-policy-arn <Policy ARN> --approve
   ```

   Where:

   <Name of service account> is the identity of your app towards the Kubernetes API server. The pod that hosts your app uses this service account.

`<Policy ARN>` is the policy created in step # 3.

For example:

```
eksctl create iamserviceaccount --name my-serviceaccount --namespace default --
cluster flogoDemo --attach-policy-arn arn:aws:iam::338799163723:policy/
marketPlaceRegisterUsage --approve
```

6. Set up a pod:
   a) Set up a pod to use the service account created in the previous step. Add the details to a pod spec, for example, `pod-definition-sample.yaml`:

   ```
   kind: Pod
   apiVersion: v1
   metadata:
     name: sample-pod
   spec:
     containers:
       - image: <Flogo application image from ECR>
         name: sample-pod
         stdin: true
         tty: true
     serviceAccountName: <Name of service account>
   ```

   Where `<Flogo application image from ECR>` is the URI of the image pushed to the AWS ECR Repository or the name of the image from the public Docker hub repository.

   b) Create the pod and deploy the app:

   ```
   kubectl apply -f pod-definition-sample.yaml
   ```

   A pod is created.

   The `kubectl get pods` command returns the following:

   ```
   NAME          READY   STATUS    RESTARTS   AGE
   sample-pod    1/1     Running   0          7s
   ```

   The `kubectl logs <pod name>` command returns the following logs (as an example):

   ```
   ######################## Starting Flogo Application #######################
   TIBCO Flogo® Runtime - 2.7.0 (Powered by Project Flogo™ - v0.9.3)
   TIBCO Flogo® connector for General -  1.1.0.251
   2019-09-15T06:59:52.007Z INFO [flogo] - AWS region environment variable not
   present, obtaining via ec2 instance metadata.
   2019-09-15T06:59:53.288Z INFO [flogo] - Product usage is registered with AWS
   metering service.
   2019-09-15T06:59:53.288Z INFO [flogo] - Standard TIBCO connectors used in the app
   - 1
   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        Starting TIBCO Flogo® Runtime
   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   2019-09-15T06:59:53.289Z WARN [flogo] - unable to create child logger named:
   ReceiveHTTPMessage - unable to create child logger
   2019-09-15T06:59:53.289Z INFO [general-trigger-rest] - Name: ReceiveHTTPMessage,
   Port: 9999
   2019-09-15T06:59:53.289Z INFO [general-trigger-rest] - ReceiveHTTPMessage:
   Registered handler [Method: GET, Path: /books]
   2019-09-15T06:59:53.289Z INFO [flogo.engine] - Starting app [ fe-270-payg-rest ]
   with version [ 1.1.0 ]
   2019-09-15T06:59:53.289Z INFO [flogo.engine] - Engine Starting...
   2019-09-15T06:59:53.289Z INFO [flogo.engine] - Starting Services...
   2019-09-15T06:59:53.289Z INFO [flogo] - ActionRunner Service: Started
   2019-09-15T06:59:53.289Z INFO [flogo.engine] - Started Services
   2019-09-15T06:59:53.289Z INFO [flogo.engine] - Starting Application...
   2019-09-15T06:59:53.289Z INFO [flogo] - Starting Triggers...
   2019-09-15T06:59:53.289Z INFO [general-trigger-rest] - Starting
   ReceiveHTTPMessage...
   2019-09-15T06:59:53.289Z INFO [general-trigger-rest] - Started ReceiveHTTPMessage
   2019-09-15T06:59:53.289Z INFO [flogo] - Trigger [ ReceiveHTTPMessage ]: Started
   2019-09-15T06:59:53.289Z INFO [flogo] - Triggers Started
   2019-09-15T06:59:53.289Z INFO [flogo.engine] - Application Started
   2019-09-15T06:59:53.289Z INFO [flogo.engine] - Engine Started
   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        Runtime started in 1.767401ms
   ```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2019-09-15T06:59:53.290Z INFO [flogo] - Management Service started successfully
on Port[7777]
```

7. If you want to deploy a Flogo App with a REST endpoint:

   a) Create a Service in Kubernetes using a YAML file. For example, `service-definition-sample.yaml`:

```
apiVersion: v1
kind: Service
metadata:
  name: flogo-rest
  labels:
    app: flogo-rest
spec:
  type: LoadBalancer
  ports:
  - port: 9999
    targetPort: 9999
    name: app
  selector:
    app: flogo-rest
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: flogo-rest
spec:
  replicas: 1
  selector:
    matchLabels:
      app: flogo-rest
  template:
    metadata:
      name: flogo-rest
      labels:
        app: flogo-rest
    spec:
      containers:
        - name: flogo-rest
          image: <Flogo application image from ECR>
          imagePullPolicy: Always
          ports:
            - containerPort: 9999
      serviceAccountName: <Name of service account>
```

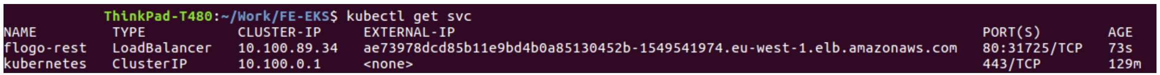
   b) Create a service using the following command:

```
kubectl apply -f service-definition-sample.yaml
```

   c) You can get service information, such as External-IP/PORT, using the following command:

```
kubectl get svc <name of service>
```

   For example, the command returns:



```
           ThinkPad-T480:~/Work/FE-EKS$ kubectl get svc
NAME          TYPE           CLUSTER-IP      EXTERNAL-IP                                                              PORT(S)          AGE
flogo-rest    LoadBalancer   10.100.89.34    ae73978dcd85b11e9bd4b0a85130452b-1549541974.eu-west-1.elb.amazonaws.com  80:31725/TCP     73s
kubernetes    ClusterIP      10.100.0.1      <none>                                                                   443/TCP          129m
```

   d) Access the REST endpoint of the Flogo app as follows:

```
http://<EXTERNAL-IP>:PORT/<Resource name defined in Flogo app>
```

   For example:

```
http://a8674b572d7a811e99b4206034ad335a-1518682477.eu-
west-1.elb.amazonaws.com:9999/books
```

8. (Optional) Deploy the Kubernetes Web UI (Dashboard) to your EKS cluster. For more information, refer to https://docs.aws.amazon.com/eks/latest/userguide/dashboard-tutorial.html.

9. To clean up all the Kubernetes resources (such deployments, pods, replica sets, services, secrets, and so on), run the following command:

```
kubectl delete all --all
```

Troubleshooting

Issue	Resolution

While running a Flogo application on a pod or running it as a Kubernetes service, the following error is displayed.
`Can not start application due to permission issue with current task role. Below policy must be configured for the task IAM role.`
`{`
`  "Version":"2012-10-17",`
`"Statement":[`
`     {`
`        "Action":[`
`           "aws-marketplace:RegisterUsage"`
`        ],`
`        "Effect":"Allow",`
`        "Resource":"*"`
`     }`
`  ]`
`}`
	Make sure you have created a policy with the specified JSON and attached the policy to the Kubernetes service account as described in step # 5 of section "Deploying a Flogo App on AWS EKS.

While running the `kubectl get pods` or `kubectl get svc` commands, the following error is displayed. `You must be logged in to the server (Unauthorized)`	Make sure the `kubeconfig` configuration file is created at the default `kubeconfig` path (`.kube/config`) in the home directory and it has user information in it.

For more troubleshooting information, see the Amazon EKS Documentation.

Serverless Deployments

Calling Lambda Functions

AWS Lambda is a serverless compute service provided by Amazon Web Services (AWS). Lambda functions automatically run pieces of code in response to specific events while also managing the resources that the code requires to run. Refer to the AWS documentation for more details on AWS Lambda.

Creating a Connection with the AWS Connector

You must create AWS connections before you use the Lambda trigger or activity in a flow.

AWS Lambda is supported on the Linux platform only.

To create an AWS connection, do the following:

1. In TIBCO Cloud Integration - Flogo (PAYG), click **Connections** to open its page.
2. Click the **AWS Connector** card.
3. Enter the connection details. Refer to the section, AWS Connection Details for details on the connection parameters.
4. Click **Save**.

Your connection gets created and will be available for you to select in the drop down menu when adding a **Lambda** activity or trigger.

AWS Connection Details

To establish the connection to the Amazon Kinesis connector, you must specify the following configurations in the AWS Connector dialog box.

The AWS Connector dialog box contains the following fields:

Field	Description
Name	Specify a unique name for the connection that you are creating. This is displayed in the **Connection Name** drop-down list for all the Amazon Kinesis activities.
Description	A short description of the connection.
Custom Endpoint	(Optional) To enable the AWS connection to an AWS or AWS compatible service running at the URL specified in the **Endpoint** field, set this field to **True**.
Endpoint	This field is available only when **Custom Endpoint** is set to **True**. Enter the service endpoint URL in the following format: *<protocol>://<host>:<port>*. For example, you can configure a MinIO cloud storage server endpoint.
Region	Region for the Amazon connection.
Access key ID	Access key ID of the AWS account (from the **Security Credentials** field of IAM Management Console). For details, see the AWS documentation.
Secret access key	Enter the secret access key. This is the access key ID that is associated with your AWS account. For details, see the AWS documentation.
Use Assume Role	This enables you to assume a role from another AWS account. By default, it is set to **False** (indicating that you cannot assume a role from another AWS account). When set to **True**, provide the following information: • **Role ARN** - Amazon Resource Name of the role to be assumed • **Role Session Name** - Any string used to identify the assumed role session • **External ID** - A unique identifier that might be required when you assume a role in another account • **Expiration Duration** - The duration in seconds of the role session. The value can range from 900 seconds (15 minutes) to the maximum session duration setting that you specify for the role. For details, see the AWS documentation.

Monitoring

This section contains information about how to monitor your apps.

App Metrics

For REST APIs, the following methods can be used to enable and disable app metrics at runtime.

Method	Description	Status Code
POST /app/metrics	Enable instrumentation metrics collection	200 - If successfully enabled 409 - If the metrics collection is already enabled
DELETE /app/metrics	Disable metrics collection	200 - If successfully disabled 404 - If metrics collection is not enabled
GET /app/metrics/flows	Retrieve metrics for all flows	200 - Successfully returned metrics data 404 - If the metrics collection is not enabled 500 - If there is an issue when returning metrics data
GET /app/metrics/flow/*<flowname>*	Retrieve metrics for a given flow	200 - Successfully returned metrics data 400 - If the flow name is incorrect 404 - If the metrics collection is not enabled 500 - If there is an issue returning metrics data
GET /app/metrics/flow/*<flowname>*/activities	Retrieve metrics for all activities in a given flow	200 - Successfully returned metrics data 400 - If the flow name is incorrect 404 - If the metrics collection is not enabled 500 - If there is an issue returning the metrics data

Enabling App Metrics

Set the FLOGO_HTTP_SERVICE_PORT environment variable to point to the port number of the HTTP service that provides APIs for collecting app metrics. This enables the runtime HTTP service.

Procedure

1. Run the following:

   ```
   FLOGO_HTTP_SERVICE_PORT=<port>  ./<app-binary>
   ```

2. Run the curl command for the appropriate REST method. Refer to App Statistics for details on each method. Some examples are:

   ```
   curl  -X POST http://localhost:7777/app/metrics
   curl  -X GET http://localhost:7777/app/metrics/flows
   curl  -X DELETE http://localhost:7777/app/metrics
   ```

Enabling statistics collection using environment variables

To enable metrics collection through environment variable, do the following:

Procedure

1. Run the following:

   ```
   FLOGO_HTTP_SERVICE_PORT=<port> FLOGO_APP_METRICS=true ./<appname>
   ```

2. Run the curl command for the appropriate REST method. Refer to App Statistics for details on each method. Some examples are:

   ```
   curl  -X GET http://localhost:7777/app/metrics/flows
   curl  -X DELETE http://localhost:7777/app/metrics/flows
   ```

Example: retrieve specific metrics for an app

The following is an example of how you would run the above steps for a fictitious app named REST_Echo.

```
FLOGO_HTTP_SERVICE_PORT=7777 FLOGO_APP_METRICS=true ./REST_Echo-darwin-amd64


curl  -X GET http://localhost:7777/app/metrics/flows

{"app_name":"REST_Echo","app_version":"1.0.0","flows":
[{"started":127639,"completed":126784,"failed":0,"avg_exec_time":0,"min_exec_time
":0,"max_exec_time":4,"flow_name":"PostBooks"}]}



curl  -X GET http://localhost:7777/app/metrics/flow/PostBooks/activities
{"app_name":"REST_Echo","app_version":"1.0.0","tasks":
[{"started":127389,"completed":126908,"failed":0,"avg_exec_time":0,"min_exec_time
":0,"max_exec_time":4,"flow_name":"PostBooks","task_name":"Return"}]}
```

Logging App Metrics

You can record app metrics of flows and activities to the console logs. To enable the logging of app metrics, use the following environment variables:

Environment Variable Name	Default Values	Description
FLOGO_APP_METRICS_LOG_EMITTER_ENABLE	False	This property can be set to either `True` or `False`: • `True`: App metrics are displayed in the logs with the values set in FLOGO_APP_METRICS_LOG_EMMITTER_CONFIG. • `False`: App metrics are not displayed in the logs. If this variable is not provided, the default values are used.

FLOGO_APP_METRICS_LOG_EMITTER_CONFIG	Both flow and activity	This property can be set to either `flow` level or `activity` level. The format for setting the property is:
		`{"interval":"<interval_in_seconds>","type":["flow","activity"]}`
		where:
		• `interval` is the time interval (in seconds) after which the app metrics are displayed in the console.
		• `type` is the level at which the app metrics are to be displayed - `flow` or `activity`. Depending on which level you set, the app metrics are displayed only for that level.
		For example:
		`{"interval":"1s","type":["flow","activity"]}`

For a list of list of fields or app metrics returned in the response, refer to Fields returned in the response.

Fields returned in the response

The following table describes the fields that can be returned in the response.

Flow

Name	Description
app_name	Name of the app
app_version	Version of the app
flow_name	Name of the flow
started	Total number of times a given flow is started
completed	Total number of times a given flow is completed

Name	Description
failed	Total number of times a given flow has failed
avg_exec_time	Average execution time of a given flow for successfully completed executions
min_exec_time	Minimum execution time for a given flow
max_exec_time	Maximum execution time for a given flow

Activity

Name	Description
app_name	Name of the app
app_version	Version of the app
flow_name	Name of the flow
activity_name	Name of the activity
started	Total number of times a given activity is started
completed	Total number of times a given activity is completed
failed	Total number of times a given activity has failed
avg_exec_time	Average execution time of a given activity for successfully completed executions
min_exec_time	Minimum execution time for a given activity
max_exec_time	Maximum execution time for a given activity

Prometheus

TIBCO Cloud Integration - Flogo (PAYG) supports integration with Prometheus for app metrics monitoring. Prometheus is a monitoring tool which helps in analyzing the app metrics for flows and activities.

Prometheus servers scrape data from the HTTP `/metrics` endpoint of the apps.

Prometheus integrates with Grafana which provides better visual anlytics.

Flogo apps expose the following flow and activity metrics to Prometheus. These metrics are measured in milliseconds:

Labels	Description
flogo_flow_execution_count: Total number of times the flow is started, completed, or failed	
ApplicationName	Name of app
ApplicationVersion	Version of app

Labels	Description
FlowName	Name of flow
State	State of the flow. One of the following states: • Started • Completed • Failed
flogo_flow_duration_msec: Total time (in ms) taken by the flow for successful completion or failure	
ApplicationName	Name of app
ApplicationVersion	Version of app
FlowName	Name of flow
State	State of the flow. One of the following states: • Completed • Failed
flogo_activity_execution_count: Total number of times the activity is started, completed, or failed	
ApplicationName	Name of app
ApplicationVersion	Version of app
FlowName	Name of flow
ActivityName	Name of activity
State	State of the activity. One of the following states: • Started • Completed • Failed
flogo_activity_execution_duration_msec: Total time (in ms) taken by the activity for successful completion or failure	
ApplicationName	Name of app
ApplicationVersion	Version of app
FlowName	Name of flow
ActivityName	Name of activity

Labels	Description
State	State of the activity. One of the following states: - Completed - Failed
Deprecated in TIBCO Cloud Integration - Flogo (PAYG) 2.10.0. **flogo_flow_metrics**: Used for flow-level queries	
ApplicationName	Name of app
ApplicationVersion	Version of app
FlowName	Name of flow
Started	Total number of times flow is started
Completed	Total number of times flow is completed
Failed	Total number of times flow is failed
Deprecated in TIBCO Cloud Integration - Flogo (PAYG) 2.10.0. **flogo_activity_metrics**: Used for activity-level queries	
ApplicationName	Name of app
ApplicationVersion	Version of app
FlowName	Name of flow
ActivityName	Name of Activity
Started	Total number of times activity is started in given flow
Completed	Total number of times activity is completed in given flow
Failed	Total number of times activity is failed in given flow

For a list of some often-used flow-level queries, refer to the section, Often-Used Queries.

Using Prometheus to Analyze Flogo App Metrics

To enable Prometheus monitoring of Flogo apps, run the following:

```
FLOGO_HTTP_SERVICE_PORT=7779 FLOGO_APP_METRICS_PROMETHEUS=true ./<app-binary>
```

Setting `FLOGO_APP_METRICS_PROMETHEUS` variable to `true` enables Prometheus monitoring of Flogo apps. The variable, `FLOGO_HTTP_SERVICE_PORT`, is used to set the port number on which the Prometheus endpoint is available.

Use the following endpoint URL in Prometheus server configuration: `http://<APP_HOST_IP>:<FLOGO_HTTP_SERVICE_PORT>/metrics` for example, `http:// 192.0.2.0:7779/metrics`

Often-Used Queries

Prometheus uses the PromQL query language. This section lists some of the most commonly and often-used queries at the flow-level.

Flow-level Queries

To Get this Metric	Use this Query
Total number of flows that got successfully executed per app	`count(flogo_flow_execution_count{State="Completed"}) by (ApplicationName, FlowName)`
Total number of flows that failed per app	`count(flogo_flow_execution_count{State="Failed"}) by (ApplicationName, FlowName)`
Total number of flows that executed successfully across all apps (when you are collecting metrics for multiple apps)	`count(flogo_flow_execution_count{State="Completed"})`
Total number of flows that failed across all apps (when you are collecting metrics for multiple apps)	`count(flogo_flow_execution_count{State="Failed"})`
Total time taken by flows which got executed successfully	`sum(flogo_flow_execution_duration_msec{State="Completed"}) by (ApplicationName, FlowName)`
Total time taken by flows which failed	`sum(flogo_flow_execution_duration_msec{State="Failed"}) by (ApplicationName, FlowName)`
Minimum time taken by the flows that got executed successfully (what was the minimum time taken by a flow from amongst the flows that executed successfully)	`min(flogo_flow_execution_duration_msec{State="Completed"}) by (ApplicationName)`
Minimum time taken by flows which failed	`min(flogo_flow_execution_duration_msec{State="Failed"}) by (ApplicationName)`
Maximum time taken by flows which executed successfully	`min(flogo_flow_execution_duration_msec{State="Completed"}) by (ApplicationName)`
Maximum time taken by flows which failed	`max(flogo_flow_execution_duration_msec{State="Failed"}) by (ApplicationName)`
Average time taken by flows which executed successfully	`avg(flogo_flow_execution_duration_msec{State="Completed"}) by (ApplicationName, FlowName)`
Average time taken by flows which failed	`avg(flogo_flow_execution_duration_msec{State="Failed"}) by (ApplicationName, FlowName)`

Activity-level Queries

To Get this Metric	Use this Query
Total number of activities that got successfully executed per flow and app	`count(flogo_activity_execution_count{State="Completed"}) by (ApplicationName, FlowName,ActivityName)`
Total number of activities that failed per flow and app	`count(flogo_activity_execution_count{State="Failed"}) by (ApplicationName, FlowName,ActivityName)`
Total number of activities that executed successfully across all apps (when you are collecting metrics for multiple apps)	`count(flogo_activity_execution_count{State="Completed"})`
Total number of activities that failed across all apps (when you are collecting metrics for multiple apps)	`count(flogo_activity_execution_count{State="Failed"})`
Individual time taken by activities which got executed successfully per app and flow	`sum(flogo_activity_execution_duration_msec{State="Failed"}) by (ApplicationName, FlowName,ActivityName)`
Individual time taken by activities which failed per app and flow	`sum(flogo_activity_execution_duration_msec{State="Failed"}) by (ApplicationName, FlowName,ActivityName)`
Minimum time taken by the activity that got executed successfully within a given flow and app	`min(flogo_activity_execution_duration_msec{State="Completed"}) by (ApplicationName, FlowName,ActivityName)`
Minimum time taken by a failed activity within a given flow and app	`min(flogo_activity_execution_duration_msec{State="Failed"}) by (ApplicationName, FlowName,ActivityName)`
Maximum time taken by an activity which executed successfully within a given flow and app	`max(flogo_activity_execution_duration_msec{State="Completed"}) by (ApplicationName, FlowName,ActivityName)`
Maximum time taken by an activity which failed within a given flow and app	`max(flogo_activity_execution_duration_msec{State="Failed"}) by (ApplicationName, FlowName,ActivityName)`
Average time taken by an activity which executed successfully within a given flow and app	`avg(flogo_activity_execution_duration_msec{State="Completed"}) by (ApplicationName, FlowName,ActivityName)`
Average time taken by an activity which failed within a given flow and app	`avg(flogo_activity_execution_duration_msec{State="Failed"}) by (ApplicationName, FlowName,ActivityName)`

App Tracing

App tracing allows you to log information when a program is executing. The trace log can then be used for diagnostic purposes such as debugging failures in the program execution.

Tracing is used to help you identify issues with your app (performance of the app or simply debugging an issue) instead of going through stack traces. The use of tracing is particularly useful in a distributed microservice architecture environment where all the apps are instrumented by some kind of tracing framework and while the tracing framework runs in the background, you can monitor each trace in the UI. You can use that to track any abnormalities or issues to identify the location of the problem.

Some Considerations

Keep the following in mind when using the tracing capability in TIBCO Cloud Integration - Flogo (PAYG):

- At any given point of time only one tracer can be registered - if you try to register multiple tracers, only the first one that you register is accepted and used at run time to trace all the activities of the flow.
- All the traces start at the flow level. There are two relations between spans - a span is either the child of a parent span or the span is a span that follows (comes after) another span. You should be able to see all the operations and the traces for the flows and activities that are part of an app.
- Tracing can be done across apps by passing the tracing context from one app to another. To trace across multiple apps, you must make sure that all apps are instrumented with similar sort of tracing frameworks, such as OpenTracing semantics, so that they understand the framework language. Otherwise it will not be possible for you to get a holistic following of the entire trace through multiple services.
- When looping is enabled for an activity, each loop is considered one span, since each loop calls the server which triggers a server flow.
- If a span is passed on to the trigger, that span becomes the parent span. You should be able to see how much time is taken between the time the event is received by the trigger and the time the trigger replies back. This only works for triggers that support the extraction of the context from the underlying technology, for instance triggers that support HTTP headers.

  The **ReceiveHTTPMessage** REST trigger and **InvokeRESTService** activity are supported for this release where the REST trigger is able to extract the context from the request and **InvokeRESTService** activity is able to inject the context into the request. If two Flogo apps are both Opentracing-enabled, when one app calls the other, you can see the chain of events (invocation and how much time is taken by each invocation) in the Jaeger UI. If app A is calling app B, the total request time taken by app A is the cumulative of the time taken by all activities in app A plus the time taken by the service that it calls. If you open up each invocation separately, you can see the details of how much time was taken by each activity in that invocation.
- Triggers that support span (for instance the REST trigger) are always the parent, so any flows that are attached to that trigger are always the child of the trigger span. Trigger span is completed only after the request goes to the flow and the flow returns.
- A subflow becomes a child of the activity from which it is called.

OpenTracing

TIBCO Cloud Integration - Flogo (PAYG) supports OpenTracing for app tracing.

Jaeger

TIBCO Cloud Integration - Flogo (PAYG) provides an implementation of the OpenTracing framework using the Jaeger backend. In TIBCO Cloud Integration - Flogo (PAYG), the Flogo App binary is built with Jaeger implementation and can be enabled by setting the `FLOGO_APP_MONITORING_OT_JAEGER` environment variable to `true`. You can track how the flow went through, execution time for each activity, or in case of failure, the cause of the failure.

Each app is displayed as a service in the Jaeger UI. In TIBCO Cloud Integration - Flogo (PAYG), each flow is one operation (trace) and each activity in the flow is a span of the trace. A trace is the complete lifecycle of a group of spans. The flow is the root span and its activities are its child spans.

Pre-requisites The following pre-requisites must be met before using the tracing capability in TIBCO Cloud Integration - Flogo (PAYG):

- By default, Jaeger is not enabled in Flogo, hence tracing is not enabled. To enable Jaeger, set the `FLOGO_APP_MONITORING_OT_JAEGER` environment variable to `true`.
- Ensure that the Jaeger server is installed, running, and accessible to TIBCO Cloud Integration - Flogo (PAYG).
- If your Jaegar server is running on a machine other than the machine on which your app resides, be sure to set the `JAEGER_AGENT_HOST` and the `JAEGER_AGENT_PORT` `JAEGER_ENDPOINT=http://<JAEGER_HOST>:<HTTP_TRACE_COLLECTOR_PORT>/api/traces` environment variables. Refer to the https://github.com/jaegertracing/jaeger-client-go#environment-variables page for the environment variables that you can set.

TIBCO Cloud Integration - Flogo (PAYG)-Related Tags in Jaegar

In OpenTracing, each trace and span have their own tags. Tags are useful for filtering traces, for example if you want to search for a specific trace or time interval.

Adding your own custom tags for any one span (activity) only is currently not supported. Any custom tags that you create will be added to **all** spans and traces.

TIBCO Cloud Integration - Flogo (PAYG) introduces the following Flogo-specific tags:

For flows	
`flow_name`	Name of the flow
`flow_id`	Unique instance IDs that are generated by the Flogo engine. They are used to identify specific instances of a flow (such as when the same flow is triggered multiple times)
For activities	
`flow_name`	Name of the flow
`flow_id`	Unique instance IDs that are generated by the Flogo engine. They are used to identify specific instances of a flow (such as when the same flow is triggered multiple times)
`task_name`	Name of activity
`taskInstance_id`	Unique instance ID that are generated by the Flogo engine. This identity is used to identify the specific instance of an activity when an activity is iterated multiple times. This ID is used in looping contructs such as **iterator** or **Repeat while true**.
For subflows	

`parent_flow`	Name of the parent flow
`parent_flow_id`	Unique ID of the parent flow
`flow_name`	Name of the subflow
`flow_id`	Unique instance IDs that are generated by the Flogo engine. They are used to identify specific instances of a flow (such as when the same flow is triggered multiple times)

The tag values are automatically generated by the TIBCO Cloud Integration - Flogo (PAYG) runtime. You cannot override the default values. You have the option to set custom tags by setting them in the environment variable `JAEGER_TAGS` as key/value pair. Keep in mind that these tags will be added to **all** spans and traces.

Refer to the https://github.com/jaegertracing/jaeger-client-go#environment-variables page for the environment variables that you can set.

Best Practices in TIBCO Cloud Integration - Flogo (PAYG)

TIBCO recommends some best practices stated below for efficient development of Flogo apps.

Development

Flow Design

- **Re-use with subflows**

  If you are executing the same set of activities within multiple flows of the Flogo app, you should put them in a subflow instead of adding the same logic in multiple flows again and again. For example, error handling and common logging logic.

  Sub-flows can be called from other flows, thus enabling the logic to be reused. A subflow does not have a trigger associated with it. It always gets triggered from another flow within the same app.

- **Terminate the flow execution using a Return activity**

  Add a **Return** activity at the end of the flow, when you want to terminate the flow execution and the flow has some output which needs to be returned to either the trigger (in case of REST flows) or the parent flow (in the case of a branch flow). An Error Handler flow must also have a **Return** activity at the end.

- **Copying a flow or an activity**

  In scenarios where you want to create a flow or an activity that is very similar to an existing flow in your app, you can do so by duplicating the existing flow then making your minimal changes to the flow duplicate. You need not create a new flow. For details on how to duplicate a flow, see Duplicating a Flow. You can also copy activities. For details on how to copy an activity, see Duplicating an Activity.

- **Use of ConfigureHTTPResponse activity**

  If you define a response code in your REST trigger, **ReceiveHTTPMessage**, configure the return value for the response code in the **ConfigureHTTPResponse** activity.

  The **Return** activity is a generic activity to return data to a trigger. However, when developing a REST/HTTP API, you may need to use different schema for different HTTP response codes. You can configure the **ReceiveHTTPMessage** trigger to use different schema for different response codes by either using the Swagger 2.0 or OpenAPI 3.0 specification, or manually adding them on the trigger configuration.

  In such a scenario, you should add the **ConfigureHTTPResponse** activity in the flow before the **Return** activity, in order to construct the response data for a specific response code. **ConfigureHTTPResponse** activity allows you to select a response code, generate the input based on the schema defined on the trigger for that code and map data from the upstream activities to the input.

  You can then map output of the **ConfigureHTTPResponse** activity to the **Return** activity to return the data and response code.

  When you call a REST API from a Flow using the **InvokeRESTService** activity, you can enable the 'Configure Response Codes' option to handle the response codes returned by the API. You can add specific codes. for example 200, 404, and define a schema for each of them using this option. You can also define status code range in a format such as 2xx if the same schema is being used for all codes in that range.

- **Reserved keywords**

  TIBCO Cloud Integration - Flogo (PAYG) uses some words as keywords or reserved names. Do not use these words in your schema. For a complete list of the keywords to be avoided, see the section, Reserved Keywords to be Avoided in Schemas.

Mapper

- **Synchronizing schema**

If you make any changes to the trigger configuration after the trigger was created, you must click the **Sync** button in order for the schema changes to be propagated to the flow parameters. Refer to the section, Synchronizing Schema Between Trigger and Flow for more details.

- **Using Expressions and Functions**

  Within any one flow, use the mapper to pass data between the activities, between the trigger and the activities, or the trigger and the flow. When mapping, you can use data from the following sources:

  - Literal values - Literal values can be strings or numeric values. These values can either be manually typed in, or mapped to a value from the output of the trigger or a preceding activity in the same flow. To specify a string, enclose the string in double quotes. To specify a number, type the number into the text box for the field. Constants and literal values can also be used as input to functions and expressions.
  - Direct mapping of an input element to an element of the same type in the Upstream Output.
  - Mapping using functions - The mapper provides commonly used functions that you can use in conjunction with the data to be mapped. The functions are categorized into groups. Click a function to use its output in your input data. When you use a function, placeholders are displayed for the function parameters. You click a placeholder parameter within the function, then click an element from the Upstream Output to replace the placeholder. Functions are grouped into logical categories. Refer to the Using Functions section for more details
  - Expressions - You can enter an expression whose evaluated value will be mapped to the input field. Refer to the section, Using Expressions for more details.

- **Complex data mappings**

  – Using the `array.forEach()` mapper function, you can map complex nested arrays, filter elements of an array based on a condition, map array elements to non-array elements or elements of another array with a different structure. See the following sections for details:

    – Mapping complex arrays - Using the array.forEach() Function

    – Mapping Array Child Elements to Non-Array Elements or to an Element in a Non-Matching Array

    – Filtering Array Elements to Map Based on a Condition

    – Mapping an Identical Array of Objects

  – You can extract a particular element from a complex JSON object. The `json.path()` function takes JSONPath expression as an argument. JSONPath is an XPATH like query language for querying an element from JSON data. Refer to Using the json.path() function for more details.

Branches

- **Branch conditions**

  You can design conditional flows by creating one or more branches from an activity and defining the branch types as well as the conditions for executing these branches. Refer to the Creating a Flow Execution Branch section for details on how to create branches, the type of branches you can create, and the order in which the branches get executed in a flow.

Error handling

Errors can be handled at the activity level or at the flow level. To catch errors at the activity level, use an error branch. In this case, the flow control transfers to the main branch when there is an error during activity execution. Refer to the section, Catching Errors for more details on error handling. To catch errors at the flow level (when you want to catch all errors during the flow execution regardless of the activities from which the errors are thrown), use the Error Handler at the bottom left on the flow page to create an error flow. Since this flow must have a **Return** activity in the end, the flow execution gets terminated after the Error Handler flow executes. The control never goes back to the main flow. Refer to the section, Catching Errors, for more details.

In order to handle network faults TIBCO Cloud Integration - Flogo (PAYG) provides ability to configure the Timeout and Retry on Error settings for some specific activities such as **InvokeRESTService** and **TCMMessagePublisher**. Refer to the *TIBCO Flogo® Activities and Triggers Guide* for details on each **General** category activity and trigger.

Deployment and Configuration

Using environment variables

When deploying a Flogo app, you can override values of the app properties using environment variables. For details on using environment variables, see the section on Environment Variables.

Externalize configuration using app properties

When developing Cloud-Native microservices, we recommend that you separate the configuration from the app logic. You should avoid hard-coding values for configuration parameters in the Flogo app and use the app properties instead.

The use of app properties allows you to externalize the app configuration. Externalizing the configuration in turn allows you to change the value for any property without having to update the app. This is particularly useful when testing your app with different configurations and automating deployments across multiple environments as part of the CI/CD strategy configurations and automating deployments across multiple environments as part of the CI/CD strategy. For details on using app properties, see the section, App Properties.

Legal and Third-Party Notices

This software may be available on multiple operating systems. However, not all operating system platforms for a specific software version are released at the same time. Please see the readme.txt file for the availability of this software version on a specific operating system platform.

THIS DOCUMENT IS PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS DOCUMENT COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THIS DOCUMENT. TIBCO SOFTWARE INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS DOCUMENT AT ANY TIME.

THE CONTENTS OF THIS DOCUMENT MAY BE MODIFIED AND/OR QUALIFIED, DIRECTLY OR INDIRECTLY, BY OTHER DOCUMENTATION WHICH ACCOMPANIES THIS SOFTWARE, INCLUDING BUT NOT LIMITED TO ANY RELEASE NOTES AND "READ ME" FILES.

This and other products of TIBCO Software Inc. may be covered by registered patents. Please refer to TIBCO's Virtual Patent Marking document (https://www.tibco.com/patents) for details.