TIBCO Flogo® Enterprise

User Guide

Version 2.25.6 | June 2025

Contents

Introduction

TIBCO Flogo® Enterprise is an open-core product based on Project Flogo™, an open-source ecosystem for event-driven apps. Its ultra-light app engine offers you the flexibility to deploy your Flogo apps in containers, as serverless functions, or as static binaries on IoT edge devices. You can quickly implement microservices, serverless functions, event-driven apps, integrations, and APIs.

Flogo apps are created in TIBCO Cloud™ Integration, which provides a wizard-driven web-based tool to create integration apps without having to leave your browser. Additionally, you can now use the TIBCO Flogo® Extension for Visual Studio Code to develop apps in a similar wizard-driven approach. This extension allows you to build apps directly from within VSCode, powered by Project Flogo®, a lightweight integration engine. For more information on creating and using Flogo apps, see TIBCO Cloud™ Integration product documentation and TIBCO Flogo® Extension for Visual Studio Code product documentation.

Concepts

This section describes the main concepts used in the Flogo Enterprise 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 Apache Kafka®, Salesforce, GraphQL. Handlers residing in the triggers, dispatch events to flows. Flogo Enterprise provides a set of out-of-the-box triggers. Also provides a range of connectors for receiving events from a variety of external systems.

Flow

The Flogo provides a set of actions for processing events in a manner suitable to your implementation logic. The flow allows you to implement the business logic as a process.

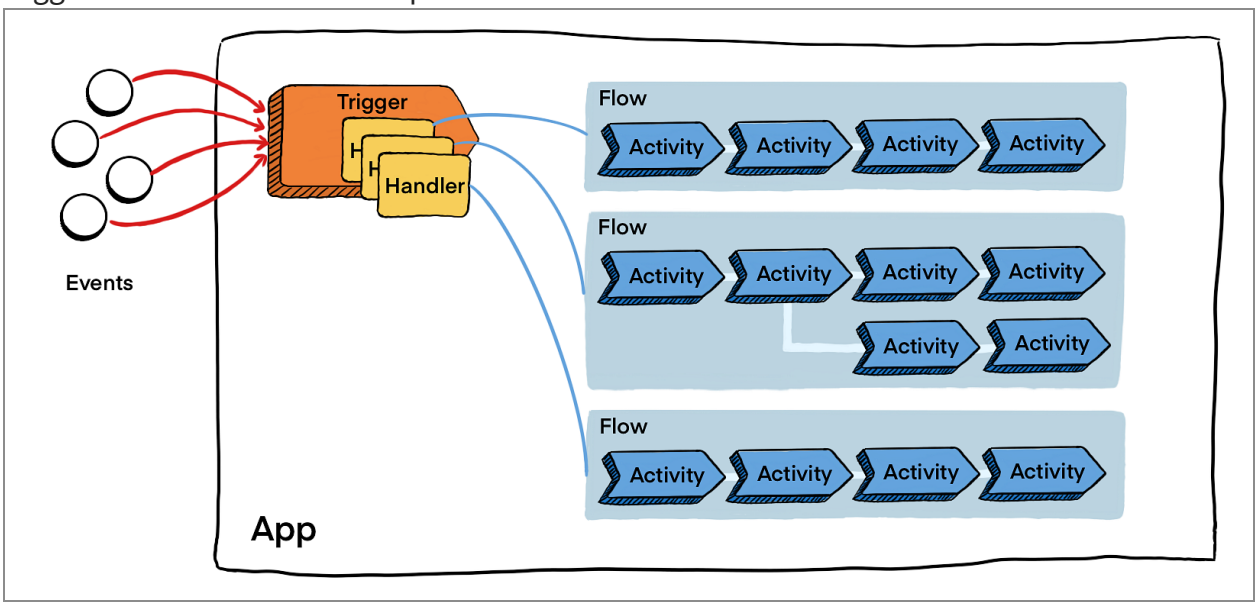
You can visually design and test the flows using the UI. A flow consists of one or more activities that perform a specific task. Activities are linked to facilitate the flow of data between them and contain conditional logic for branching. Each flow is also connected to a default error handler. A Flogo 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 Flogo 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, it uses the respective flow handlers to pass the data from the event 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.



Summary:

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 the steps to build a simple app with a REST service in Flogo Enterprise. It shows how to create a basic app that returns the booking details of a specific customer based on a query sent to the app. In this tutorial, the query sent to the app checks whether the passenger's family name is `"Jones"`. The app then returns the booking details.

For the sake of this tutorial, the sample data used are: A passenger whose family name is `"Jones"` and travels by the `"Business"` class. All other customers travel by `"Economy"` class.

Overall Structure of the App

This app contains:

- **ReceiveHTTPMessage** trigger: This trigger listens for an HTTP GET request containing the family name of the passenger requesting flight booking details. After it receives a request, it triggers the flow attached to the trigger.
- **FlightBookings** flow: This flow is attached to the **ReceiveHTTPMessage** trigger. This flow handles the business logic of the app. In this flow, you must configure a **LogMessage** activity to log a custom message when a request is received successfully. The **LogMessage** activity has two success branches:
    - The first branch accepts requests with any family name and uses a condition to check if the family name in the request is `"Jones"`. It runs a **Return** activity to return the information of a flight booked in `"Business"` class for Jones.
    - The second branch runs when the first branch runs as false (that is, the family name is not `"Jones"`). It runs a **Return1** activity to return the information of a flight booked in `"Economy"` class if the family name is not `"Jones"`.

Note: Each branch must have its **Return** activity as the last activity in the branch.

Procedure

The high-level steps for creating and configuring the app in this tutorial are as follows:

Procedure

1. Create a new app.
2. Create a JSON schema to reuse it across your app. The JSON schema describes the format of the JSON data used in the tutorial. In this tutorial, we use a simple JSON schema for the request that the REST service receives and the response that the service sends back. You can specify the JSON schema directly or specify JSON data, which is converted to JSON schema automatically.
3. Create a flow and add a REST trigger (Receive HTTP Message).
4. 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.
5. 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 running, 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.
6. Add a LogMessage Activity to the flow and configure a message that the activity must log in to the logs for the app as soon as it receives a request.
7. Add the first branch to check whether the passenger's last name is Jones to return the information of a flight booked in `"Business"` class for `Jones`.
8. Add a second branch to process any other passengers and return the information of a flight booked in `"Economy"` class if the family name is not `Jones`.
9. Validate the app to make sure that there are no errors or warnings in any flows or activities.
10. Build the App.
11. Test the app.

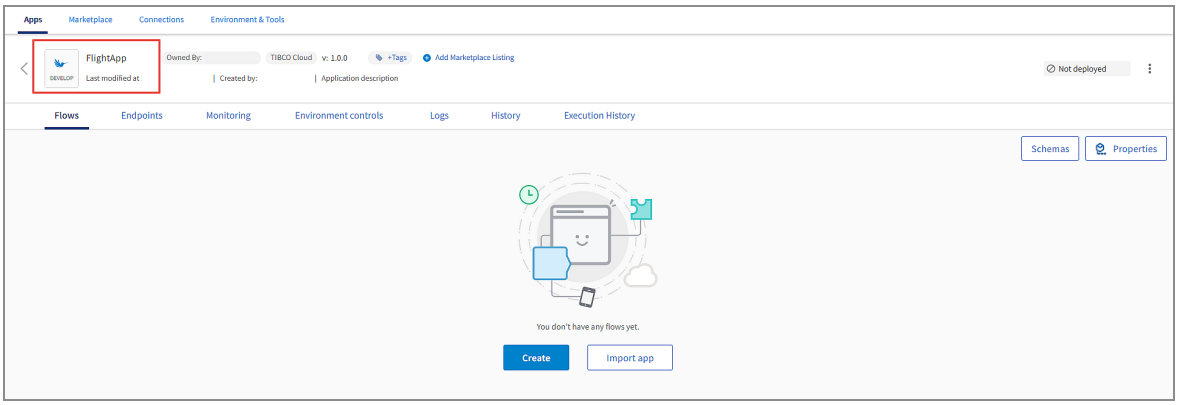
Step 1: Create an app

To create a Flogo app:

Procedure

1. Click **Apps**.
2. Click **Create/Import**. The **What do you want to build?** dialog opens.
3. To create a Flogo app:
   - Under **Quickstart** > **All app types;Apps**, click **Create a Flogo app**.
   - In the block that displays below your selection, click **Create Flogo app**.

   A Flogo app is created with the default name in the `New_Flogo_App_<sequential_app_number>` format.
4. Click the default app name to make it editable. Change the app name to `FlightApp` and click anywhere outside the name to save the changes made to the name.



Step 2: Create a JSON schema

Procedure

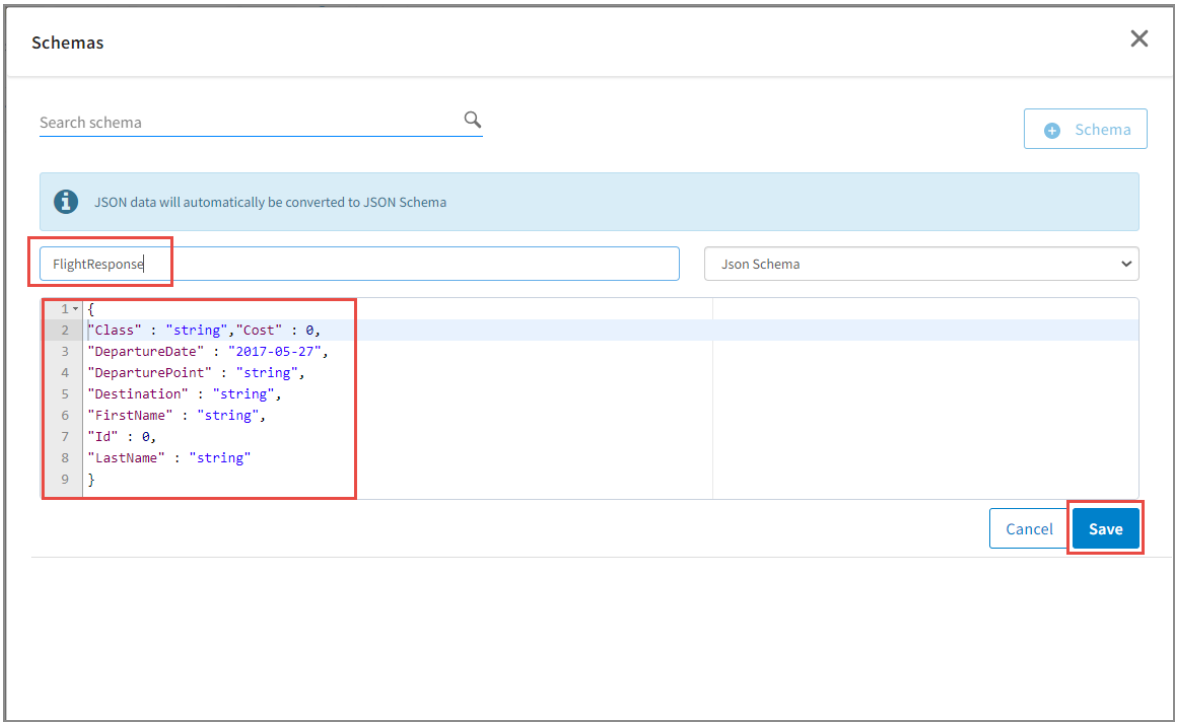
1. Copy the following JSON sample to use in your app:

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

```
"Destination" : "string",
"FirstName" : "string",
"Id" : 0,
"LastName" : "string"
}
```

Note: Ensure that you use straight quotes when entering the schema elements and values.

2. On the **Apps**page, in the **Flows** section, click **Schemas**.
3. In the **Schemas** dialog that opens, click **Schema** to add a JSON schema.
4. Name your schema as `FlightResponse` and paste the copied schema into the text editor. Alternatively, if you enter JSON data in the editor, the JSON data is automatically converted to JSON schema.



5. Click **Save**.

Step 3: Create a flow and add a REST trigger

Every app must have at least one flow and, in most cases, a trigger that initiates the flow. Create a flow with the REST trigger. The **ReceiveHTTPMessage** REST trigger listens for an

incoming REST request that contains the details of a passenger who wants to book a flight. Specify the fields for the request in the REST trigger in JSON schema format.

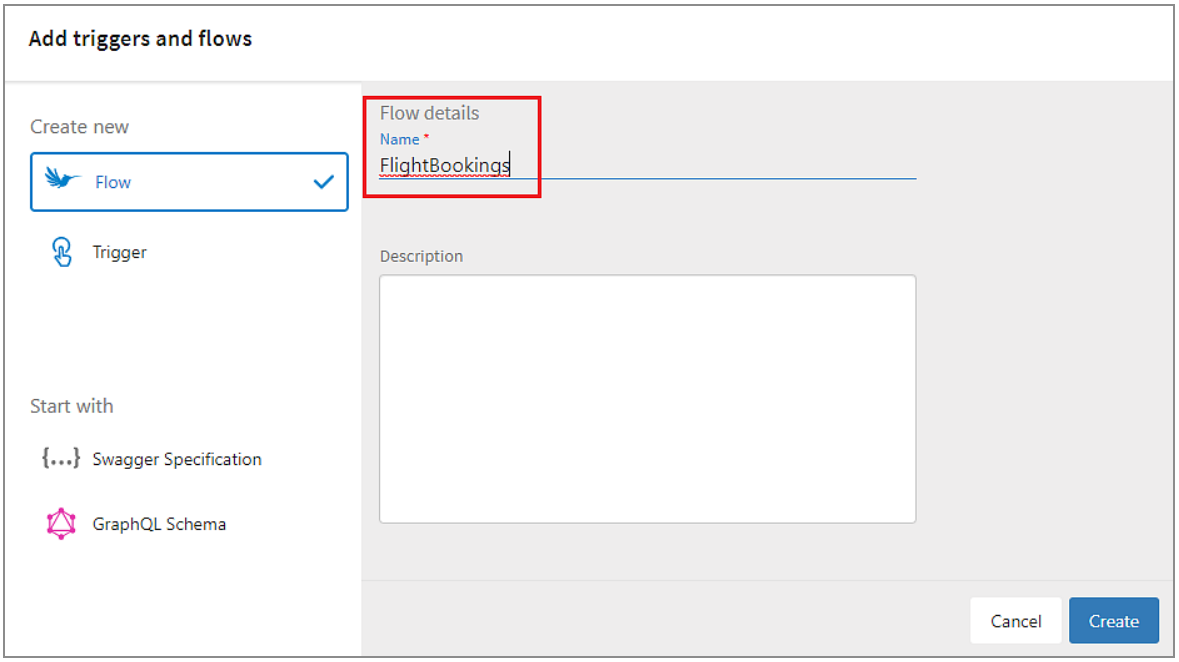
To create a flow:

Procedure

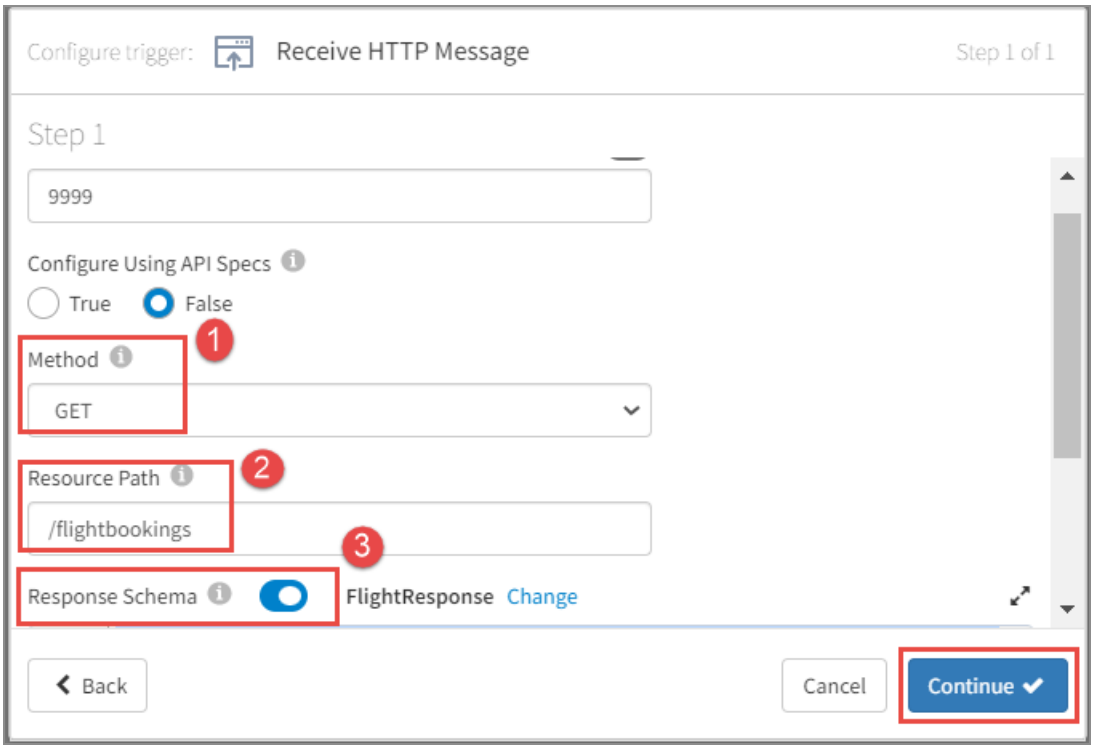
1. On the **Flows** page, click **Create**.

   The **Add triggers and flows** dialog is displayed. The **Flow** option is selected by default.

2. In the **Flow details** section, provide the following details and click **Create**:
   Name: `FlightBookings`.
   Description: `Optional description of the flow`.



3. On the `FlightBookings` flow page, click the **Triggers** icon. The trigger palette opens.
4. From the **Triggers** palette, drag the **Receive HTTP Message** trigger to the **Triggers** area on the left. The `Configure trigger: ReceiveHTTPMessage` dialog opens.



a. Select **GET** as the **Method**.

b. Enter `/flightbookings` in the **Resource path** box.

c. Enable the **Use App Level Schema** toggle next to **Response Schema** to open the **Schemas** dialog and select the **FlightResponse** schema you defined earlier.

   The selected schema is automatically displayed in the **Response Schema** box.

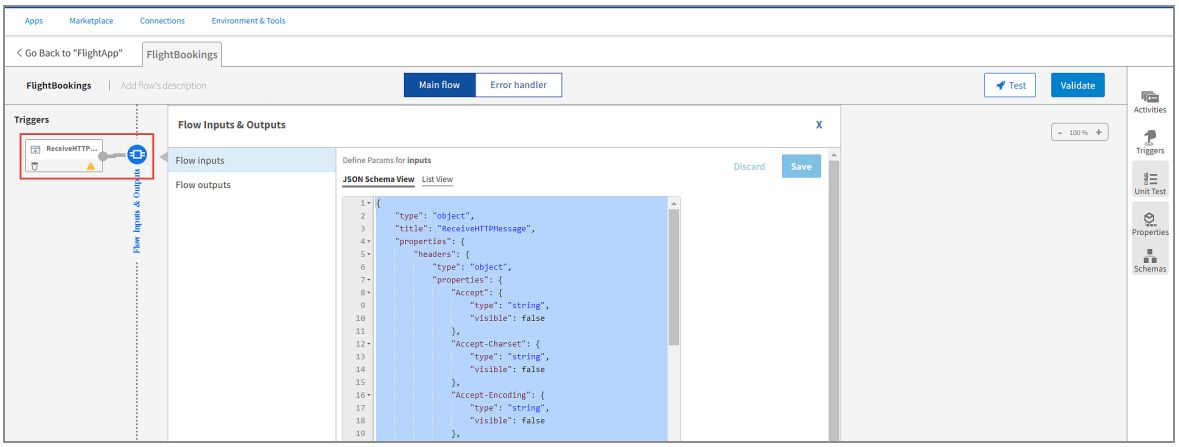
d. Click **Continue**.

5. Next, select **Copy Schema** when prompted.

   The schema that you entered when creating the trigger is automatically copied to the **Flow Inputs & Outputs** tab to match the input and output of the trigger.

   A new flow is created and attached to a REST trigger.

   Your flow must look similar to the following image:



6. Lastly, close the **Flow Inputs & Outputs** tab.

Step 4: Map trigger output to flow input

When REST trigger receives a request from a passenger (a REST request), the data from the request is produced by the **ReceiveHTTPMessage** REST trigger. For the request to be processed, this output must be used by the flow in the form of flow input. Hence, you must map the trigger output to the flow input.

To do this:

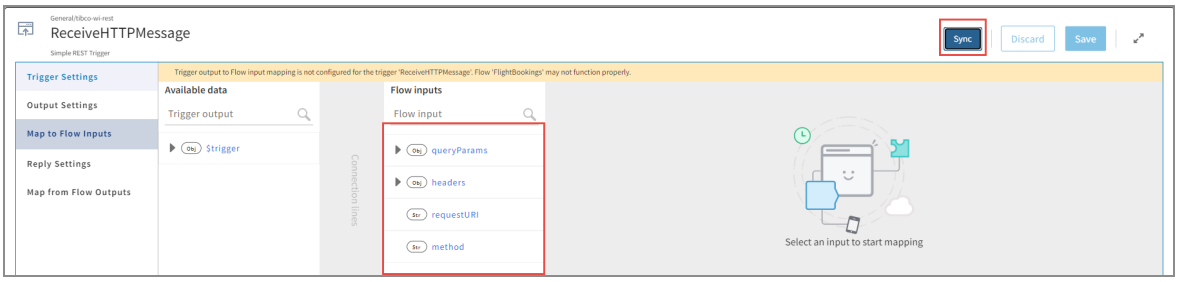
Procedure

1. Click the **REST Trigger** icon to open its configuration dialog.

   In the **Configuration** dialog, multiple tabs are displayed in a column on the left. **Trigger Settings** is selected by default.

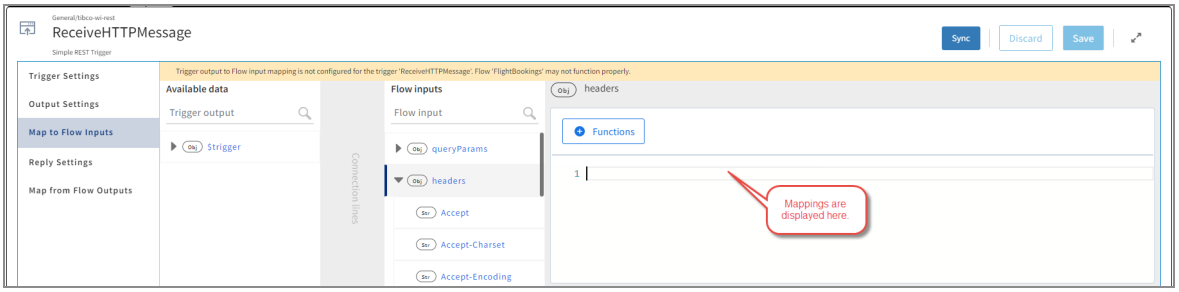
2. Click **Output Settings** to add the query parameter.
3. Click **Add row** to add a query parameter.
4. In the new row in the **Query Parameters** table, enter the value of **ParameterName** as `lastname` and click **Save** in the same row (in the **Actions** column).
5. To start the mapping, click the **Map to Flow Inputs** tab and configure the mapping of the trigger output. On the **Map to Flow Inputs** tab, the **Available data** and **Flow inputs** panes are displayed. **Flow inputs** is the list of flow inputs that can be mapped to the trigger outputs in the **Available data** pane. Only headers are displayed in the flow inputs. The new query parameter is not visible yet.
6. **Save** the trigger configuration and click **Sync** to display the new values. Now,

queryParams must open in the **Flow inputs** column.



7. In the **Flow inputs** column, click **headers**.

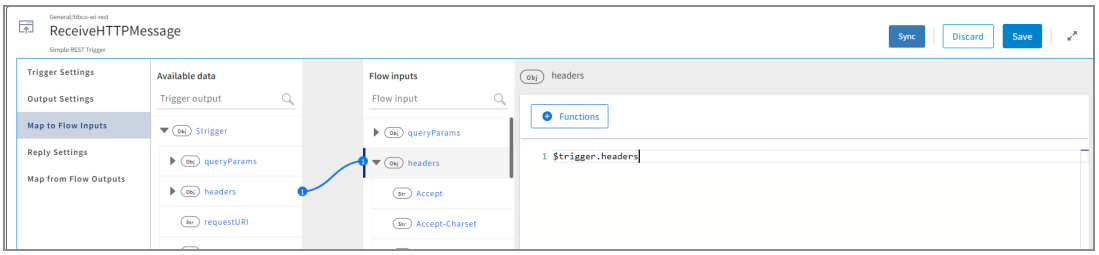
   The **headers** text editor on the right of **Flow inputs** is initially empty.



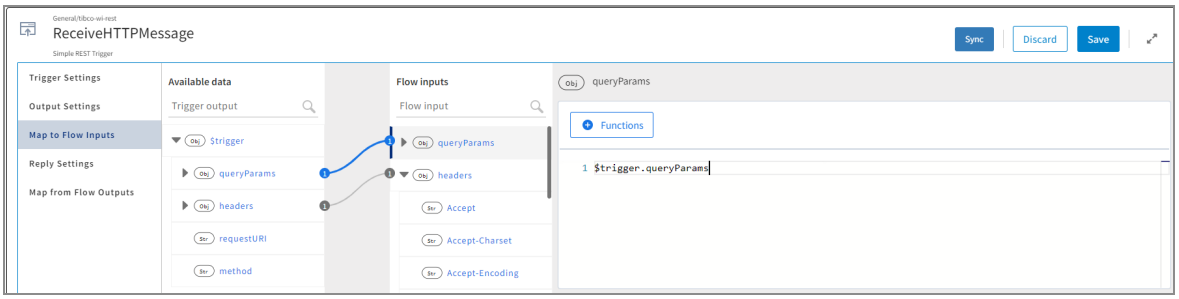
8. To map the trigger output headers to the flow input header:

   a. Expand **$trigger** to see all the trigger outputs available. This displays the headers and body.

   b. Drag **headers** from the **Available data** pane to **headers** in the **Flow inputs** pane. Alternatively, click **headers** from the **Flow inputs** pane, drag **headers** from the **Available data** pane into the text editor.
   The text editor must now display **$trigger.headers** and a connection line between the two panes. This indicates that you have successfully mapped the trigger output headers to the flow input header. The numbers at the end of the connection line indicate the total number of mappings for the selected element.



9. To map the flow input, in the **Flow inputs** column, click **queryParams**. The data mapper view is the same as the one while mapping headers. The **queryParams** text editor is initially empty. Drag **queryParams** from the **Available data** pane and drop it on **queryParams** in the **Flow inputs** pane. The text editor must now display **$trigger.queryParams**. This indicates that you have successfully mapped the trigger output **queryParams** to the flow input **queryParams**.



10. To save your progress, click **Save**.

    This completes the mapping of flow inputs.

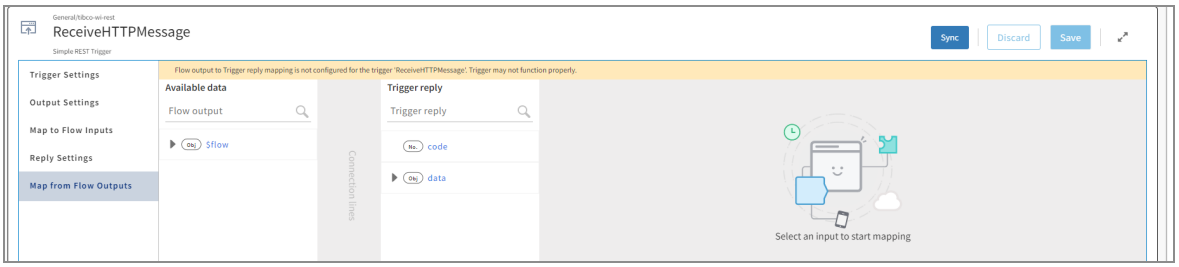
Step 5: Map the flow output to trigger reply

When the execution of the flow is completed, the 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 then returns the result of the flow execution to the REST request initiator.

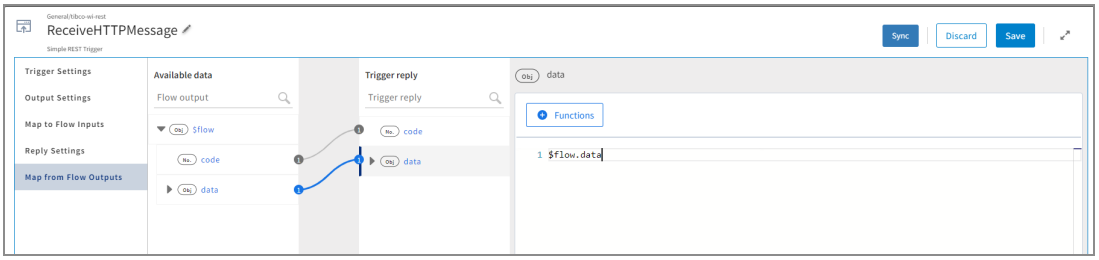
To map the flow output to the trigger reply:

Procedure

1. In the left pane, click the **Map from Flow Outputs** tab to configure the mapping of the trigger reply. The **Available data** and **Trigger reply** panes are displayed. You can map the following trigger replies to the flow outputs - **code** and **data**.



2. In the **Map from Flow Outputs** section:
    a. The **Available data** pane displays the data available for the mapping. **$flow** is displayed in this pane. To see all the flow outputs available for the mapping, expand **$flow**. This displays **code** and **data**.
    b. Drag **code** from **Available data** and drop it on **code** in the **Trigger reply** pane. **$flow.code** is displayed in the **code** text editor. You have successfully mapped the **code** in **Trigger reply** to the **code** in **Available data**.
    c. Repeat the same steps to map **data** from **Trigger reply** with **data** from **Available data**.



> **Note:** You can expand **data** in both the **Trigger reply** pane and the **Available data** pane to see the tree structure of the data you have defined in the schema.

3. Click **Save** and close the trigger dialog.

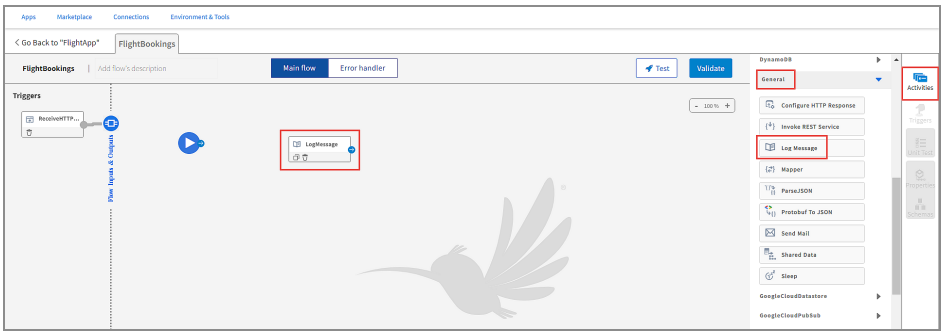
Step 6: Add a Log Message Activity to the flow

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.

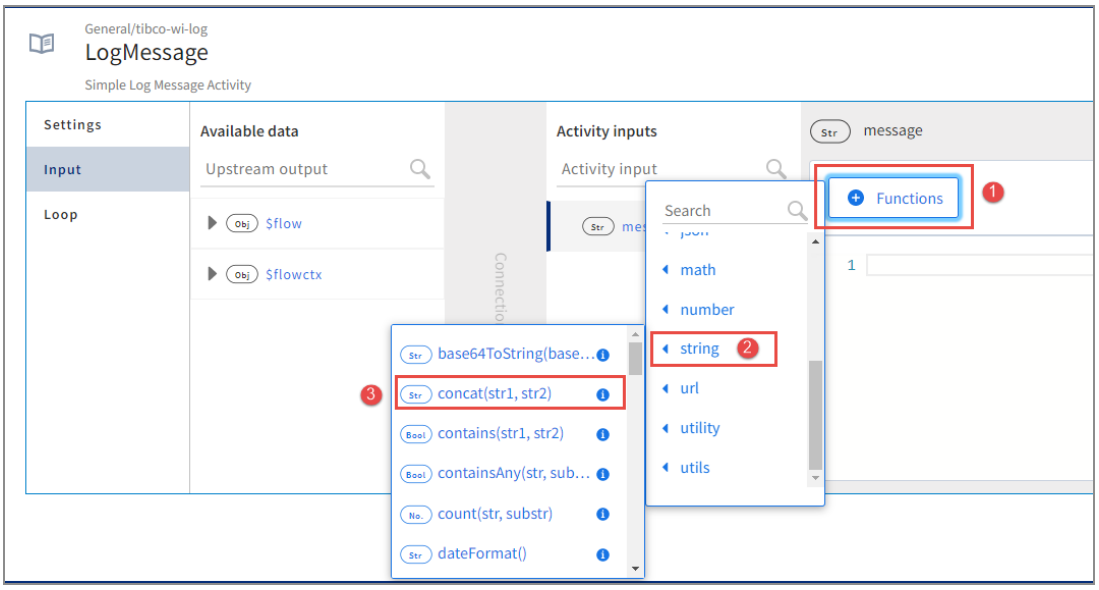
To add a **LogMessage** activity:

Procedure

1. On the `FlightBookings` flow page, click **Activities**, the activities palette opens.
2. In the **Activities** palette, under **General** tab, select **Log Message** and drag it to the activities area.



3. Drag a connection line from **StartActivity** to the **LogMessage** activity that you have created.
4. Now, to configure the **LogMessage** activity with a message to log when it receives an incoming request from the **ReceiveHTTPMessage** trigger:
    a. Click the **LogMessage** activity to open the configurations dialog.
    b. Click the **Input** tab. The **Available data** and **Activity inputs** columns are displayed on the right side of the **LogMessage** activity tabs.
    c. Click the **message** to open the mapper to the right. Configure a message to be logged by the **LogMessage** activity when the input from the request that the trigger received is passed on to and received by the flow.
    d. To configure the message, click **Functions**, and expand the **string**. Click **concat (str, str2)** to add the function to the **message** box.



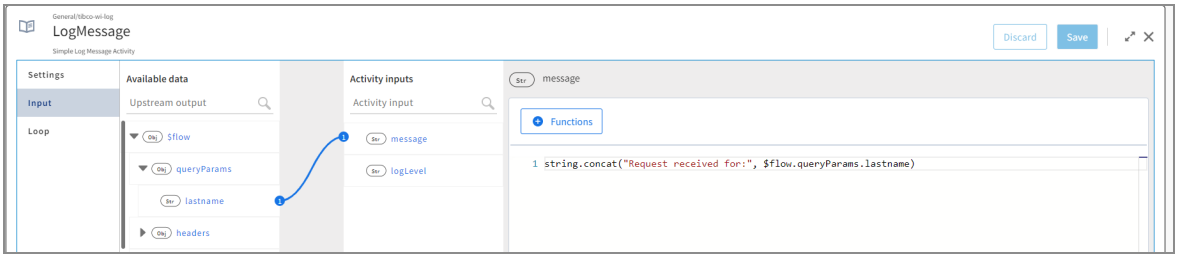
e. Select **str** in the box and replace it by entering **"Request received for: "** (include the quotes too): `string.concat(Request received for: ", str2)`.

5. Replace **str2** with the family name of the passenger who booked the flight. (The family name of the passenger is passed on from the trigger to the flow. We had mapped this trigger output to flow input previously. Hence it is now available for mapping under **$flow** in **Available data**.)

   a. In the **Available data** pane, expand **$flow** and expand **queryParams**.

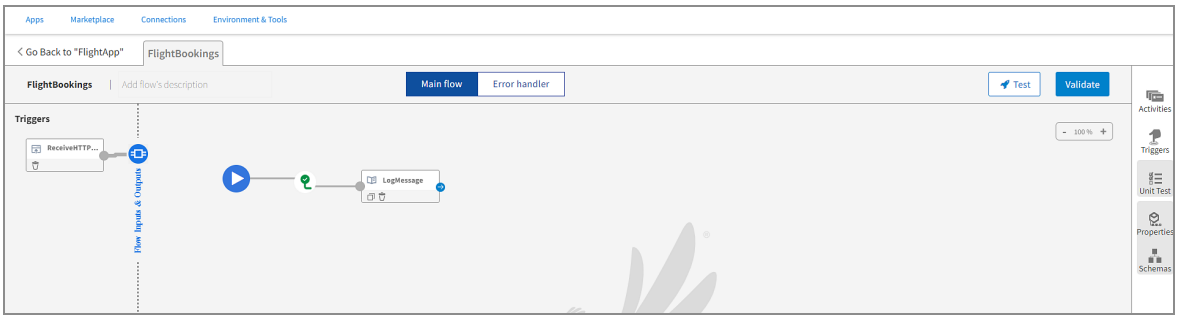
   b. Drag **lastname** and drop it in place of **str2**.

   c. Click **Save**.



6. Close the **LogMessage** dialog.

   Your flow must now look like this:

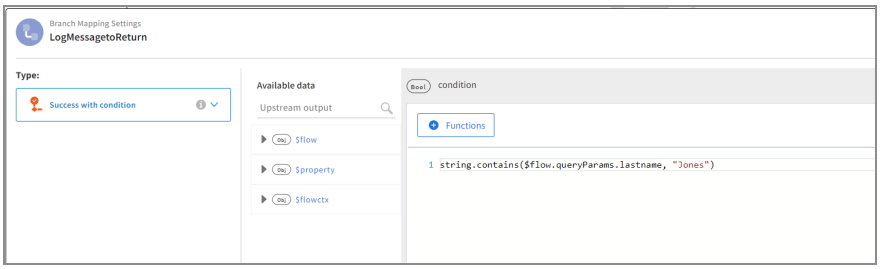


Step 7: Add the first Return Activity branch

To add a **Return** activity and the branch to configure its condition to look for the family name `"Jones"`:

Procedure

1. From the **Activities** palette, drag **Return** activity available under **Default** category to the activity area.
2. Now, to configure a connection line between a **LogMessage** activity to the **Return** activity. Configure the branch with a condition to read the family name of the passenger.
3. Drag a highlighted arrow from the **LogMessage** activity to the **Return** activity.
4. Hover over and click the branch label on the connection line you just created. The configuration window for branch condition opens.
5. In the **Branch Mapping Settings** dialog that opens, select the **Success with condition** branch condition.
    a. Click **Functions**. Select the **string>>contains(str1, str2)**. The selected function is added to the **condition** text editor.
    b. Configure **str1** in the expression to take the value of the family name that the user enters. In the **Available data**, expand **$flow** > **queryParams**. Drag **lastname** to **str1**. This family name is the name entered by the user in the search query.
    c. Replace **str2** in the condition by manually typing `"Jones"`.

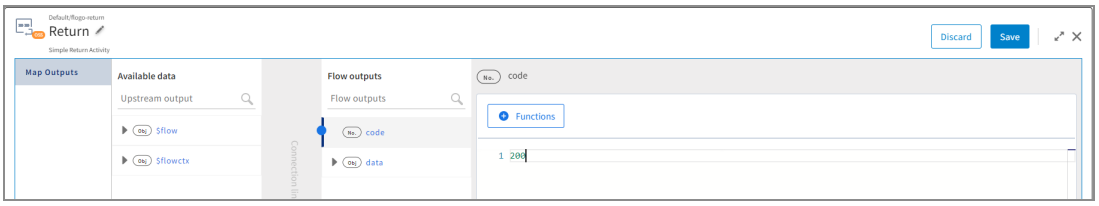


d. Click **Save**. This branch runs when the name entered as a query parameter is `Jones`.

6. Now, Configure the **Return** activity for the branch to produce the flow results if this branch runs (when the passenger's family name is anything but `Jones`):

   a. Click the return activity to open the configuration dialog.

   b. Click **code** under **Flow outputs** to open the mapper and type **200** in the **code** box, which is the HTTP success code.



   c. Expand the next flow output **data**. All the different elements under **data** that are returned by this activity are displayed. Assign a value to each field under **data**.

   d. Start by clicking **Class** under **data** and type `"Business"` as `Jones` is traveling by `"Business"` class.

   e. Click **Cost** to type a number of your choice. You can also use a function to randomize the value. To do so, in the **Functions** section, expand the **number** category and click **random()**. Enter **5000** as an input parameter to the **random ()** function.

   f. Click **DepartureDate** to enter the departure date in any format of your choice. Use quotation marks as the date needs to be specified as a string. For example, “01/01/21” or "January 1, 2021" are valid values.

   g. Click **DeparturePoint** to enter the departure airport name of your choice. Use quotation marks as the departure point needs to be specified as a string. For

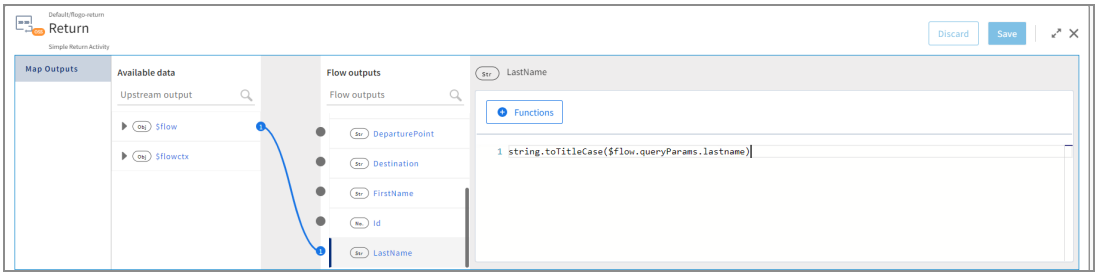
example, “LAX” or “LHR” are valid values.

h. Click **Destination** to enter a string for this field. For example, "Paris" or “JFK” are valid values.

i. Click **FirstName** to enter the first name associated with the family name `Jones`. For example, "Brian" or "Paul" are valid values.

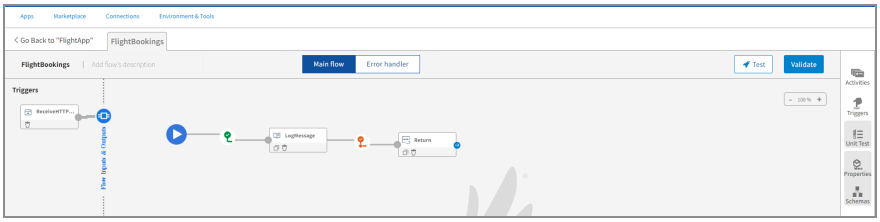
j. Click **Id** to enter a number of your choice. You can also use a function to randomize the value. To do so, in the **Functions** section, expand the **number** category and click **random()**. Enter **999999** as an input parameter to the **random()** function.

k. Click **LastName** to map this field to the query parameter **lastname**. Before doing so, we can use a string function to capitalize the family name that is returned by our app. To do so, under **Functions**, expand the **string** and click **toTitleCase(str)**. Once **string.toTitleCase(str)** is added to your box, select **str** to replace it with the query parameter. Expand **$flow** and then **queryParams** under **Available data**. Drag **lastname** and drop it in place of **str**. The text editor must look like this:



l. Click **Save** and then close the **`Return Activity Configuration`** dialog.

Your flow must look like this:



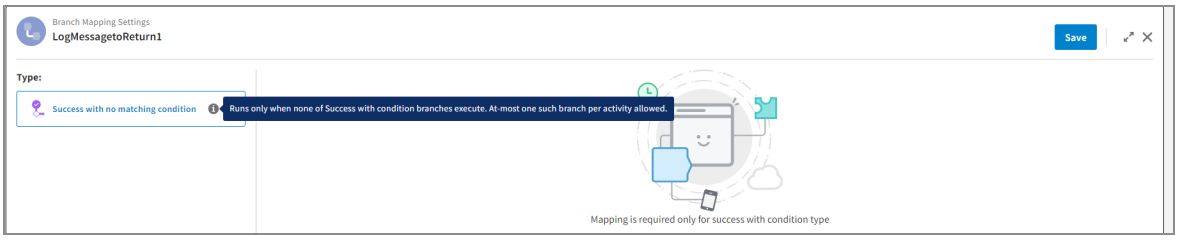
Step 8: Add a second Return Activity branch

The second branch that you add from the **LogMessage** activity runs when the success condition of the first branch is not matched. If the passenger's family name is not `"Jones"`, the passenger's seat is in `"Economy"` class.

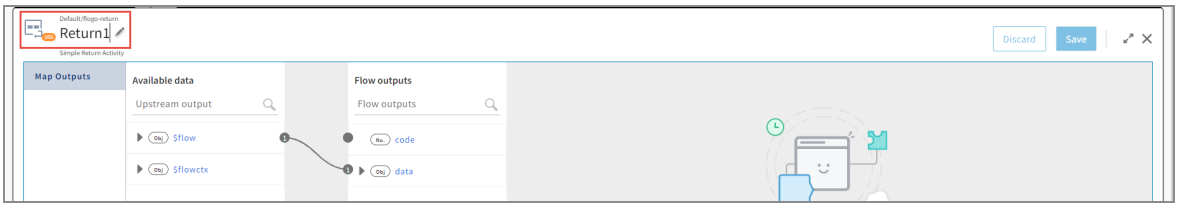
To add a second branch from the **LogMessage** activity:

Procedure

1. Duplicate the **Return** activity from the first branch instead of manually adding another **Return** activity. You can copy the activity by clicking . The copied activity is displayed next to your original **Return** activity:
2. Click the **CopyOfReturn** activity to configure the response this branch return.
3. First, to create a connection between the **LogMessage** activity and the **Return1** activity, hover over to the LogMessage activity, you see that an arrow highlighted. Drag the arrow to the **Return** activity.
4. Select the **Success with no matching condition** branch condition. If the conditions of all the other **Success with condition** branches are not true, this branch is run. This means, if the family name entered as a query parameter is not `Jones`, this second branch is run.

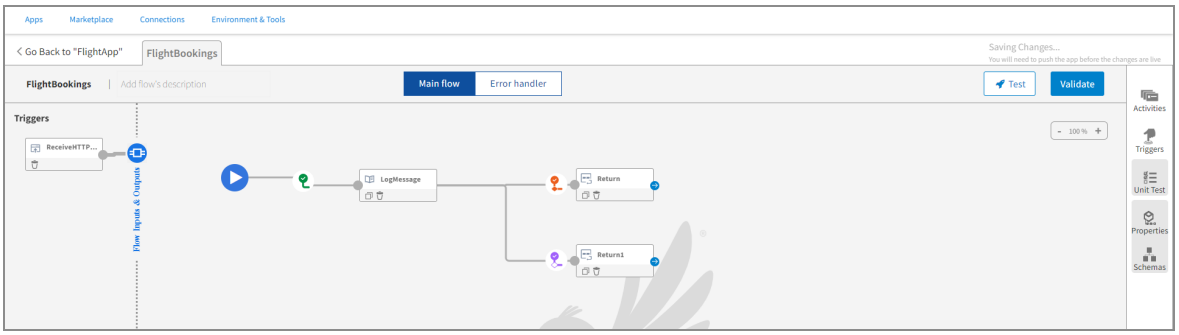


5. Now, in the configuration window click the name of the activity to make it editable and rename the activity.



6. In the **Flow outputs** section, expand **data**, select **Class**, and type `"Economy”` as this branch must return `"Economy"` class bookings.
7. Click **Save** and close the dialog.

   Your flow must look like this:



Step 9: Validate the app

Your app is now ready. Before you push the app to the Cloud, validate all the flows for any errors or warnings. To do so, click **Validate**. Flogo validates each flow and activity within the flow. For any errors or warnings, you see the respective icons next to the flow name or activity tab, which contains the error or warning.

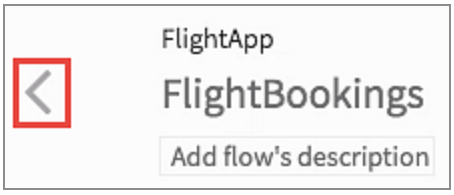
On successful validation, you get the following message:

Step 10: Build the App

Your app is now ready to be built. You can build a Flogo app using a s an executable file.

Procedure

1. Click the left arrow next to the flow name to open the FlightApp page.



2. Click **Build**.
3. Select your target platform from the **Build** drop-down list. Select **Windows/amd64** on Windows, **Darwin/amd64** on Macintosh or **Linux/amd64**, or **Linux/86** on Linux from the list.

You see a build log with the progress of the build command. When the build completes, you see an executable file called `FlightApp-darwin_<processor>` in your `/Downloads`directory.

Step 11: Test the app

Now that the app has been built successfully, you run the app. Once it runs successfully, you can test your API in a REST client.

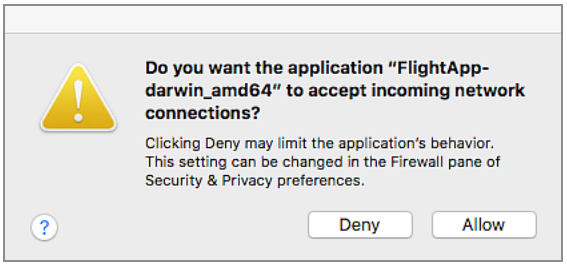
On Macintosh and Linux platforms:

To test the app:

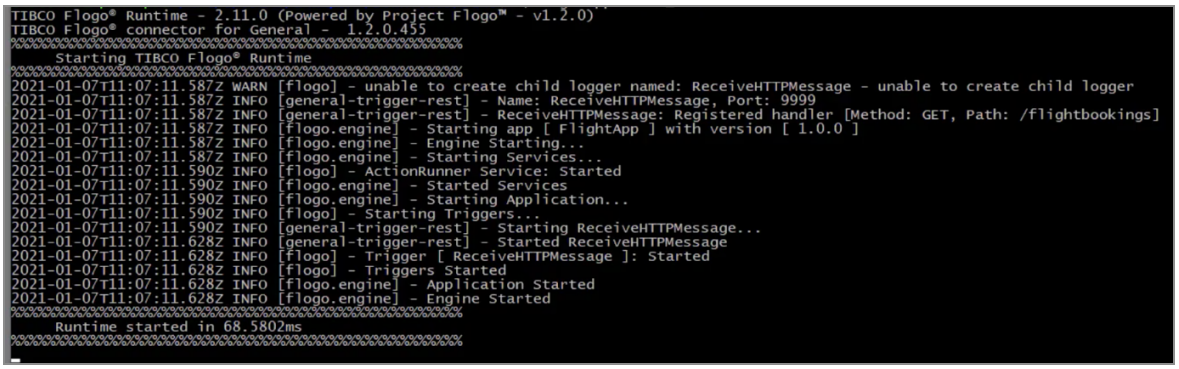
Procedure

1. Open a terminal and change the directory to the location of `FlightApp-darwin_amd64`, `FlightApp-linux_amd64`, or `FlightApp-linux_86` file depending on your platform.
2. Run the following commands:
   - `chmod +x <FlightApp-darwin_amd64>`
   - `./FlightApp-darwin_amd64`

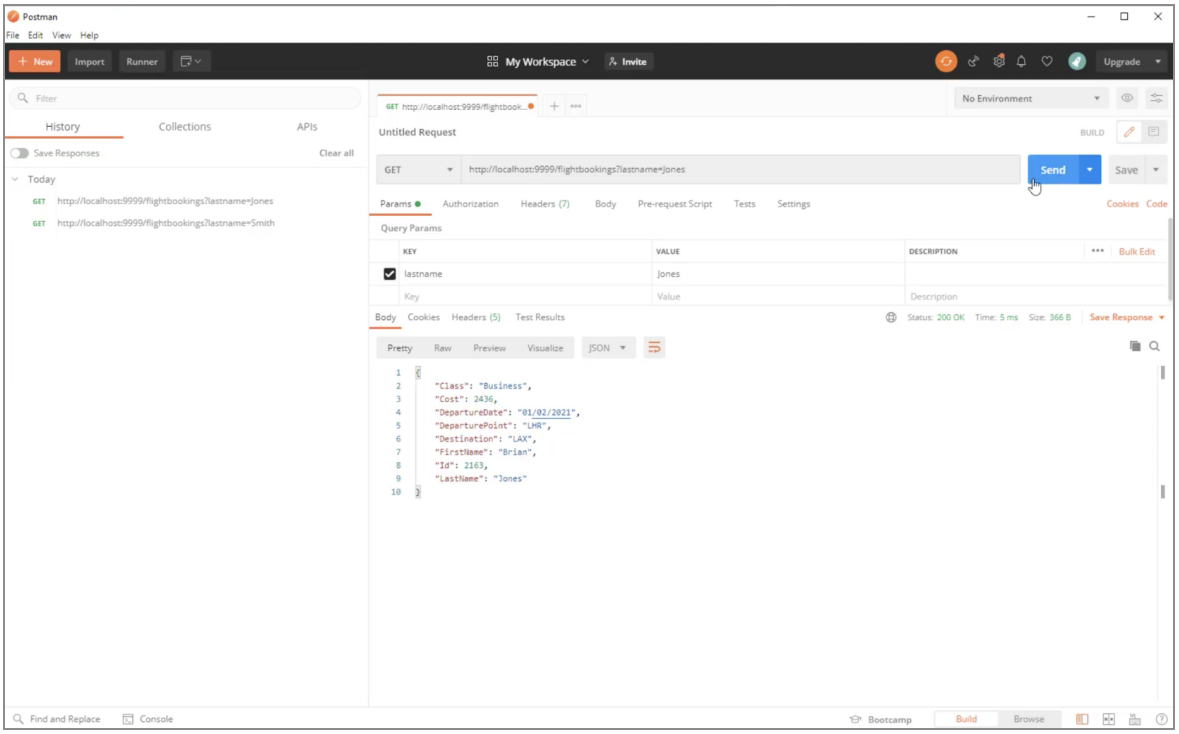
   Note: In the commands, use the file name specific to your platform - `FlightApp-linux_amd64` or `FlightApp-linux_86` in the case of Linux.
3. Click **Allow** in the following dialog:



The following messages are displayed in the console:

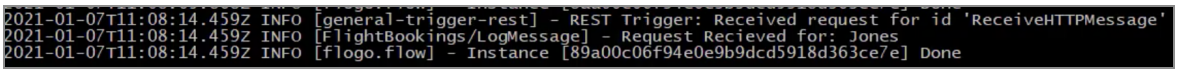


4. Make a note of the port number `9999` and path `/flightbookings` in the logs.
5. You can test your API in a REST client such as Postman by entering the port number `9999`, path `/flightbookings`, and query parameter `lastname`. For example, `http://localhost:9999/flightbookings?lastname=jones`.



In the above example, note that since the query parameter sent has the family name as `"Jones"`, the **Class** in the response has been automatically set to `"Business"` class.

6. Go back to your terminal. You must see the logs you configured with the **Log** activity.


2021-01-07T11:08:14.459Z INFO [general-trigger-rest] - REST Trigger: Received request for id 'ReceiveHTTPMessage'
2021-01-07T11:08:14.459Z INFO [FlightBookings/LogMessage] - Request Recieved for: Jones
2021-01-07T11:08:14.459Z INFO [flogo.flow] - Instance [89a00c06f94e0e9b9dcd5918d363ce7e] Done

App Development

For detailed instructions and information on app deployment, refer to the appropriate section based on your platform:

- To deploy an app using TIBCO Cloud™ Integration, see App Deployment.
- To deploy an app using TIBCO Flogo® Extension for Visual Studio Code, see App Deployment.

Deployment and Configuration

After you have created and validated your app, you can build an app executable to deploy and run it.

Building an App Executable

For detailed instructions and information on building an app executable, refer to the appropriate section based on your platform:

- To build an app executable using TIBCO Cloud™ Integration, see Building an App Executable.
- To build an app executable using TIBCO Flogo® Extension for Visual Studio Code, see Building an App Executable.

Building Flogo App Executable and Docker Image Using Flogo - App Build CLI

TIBCO Flogo® - App Build Command Line Interface is a command-line utility for building Flogo applications, especially in CI/CD pipelines. It provides a consistent and easy-to-use interface for building and testing Flogo applications.

Flogo® - App Build Command Line Interface (CLI) Usage

- Build Flogo application executables (`.exe`) on Windows, Linux, or Darwin platforms.
- Test Flogo applications using Flogo app files (`.flogo` or `.json`), Flogo app executables, and `.flogotest` files.
- Build Docker images for your Flogo applications.
- Package Flogo executables into Docker images.
- Build TIBCO Platform deployment ZIP files for your Flogo applications.

Flogo - App Build CLI Commands

flogobuild

Usage

```
flogobuild [COMMAND]
```

Commands

Command	Description
build-docker-image	Builds a Flogo application Docker image.
build-exe	Builds a Flogo application executable.
build-tp-deployment	Builds a TIBCO Platform deployment zip file for the Flogo application.
create-context	Creates a context for building and packaging Flogo applications.
delete-context	Deletes an existing context.
help	Displays help about any command.
list-context	Lists available contexts.
package-docker-image	Packages an existing Linux-based Flogo application executable as a Docker image.
set-default-context	Sets one of the configured contexts as the default for building and packaging Flogo applications.
test-app	Runs unit test cases.
version	Displays the version.

create-context

Note: Before you begin using the Flogo - App Build CLI to build executables, create Docker images, or run unit tests, it is mandatory to create the context by specifying the appropriate flags in the command.

Usage

```
flogobuild create-context [flags]
```

Available Flags

Flags	Description
`-n, --context-name`	This flag is required. Specify the name of the context.
`-e, --ems-home-directory`	Specify the path to the EMS home directory.
`-h, --help`	Displays help for create-context.
`-i, --ibmmq-home-directory`	Specify the path to the IBM MQ home directory.
`--set-default`	Sets the new context as default.
`-u, --user-extension-directory`	Specify the path to the user extension directory.
`-v, --vsc-extension-file`	This flag is required. Specify the path to the Flogo VSCode extension VSIX file.

Note: By default, the context is created at `/home/<user-name>/tibco/.fecli`. To override this default directory, set the `FLOGO_CLI_CTX_DIR` environment variable and specify the desired destination directory for storing the context.

Example

```
flogobuild create-context --context-name "<context-name>" --vsc-extension-file "<path-to-vsc-file>\<.vsix file>" --user-extension-directory "<extension-directory>" --ems-home-directory "<ems-home-directory>" --ibmmq-home-directory "<ibmq-home-directory>"
```

build-docker-image

Usage

```
flogobuild build-docker-image [flags]
```

Available Flags

Flag	Description
`-f , --app-json-file`	This flag is required. Specify the path to the Flogo application file (`.json`/`.flogo`).
`-c, --context-name`	Specify the name of the context to be used for building applications. If not provided, the default context is used.
`-d, --docker-file`	Specify the path to the Flogo application Dockerfile.
`-i, --docker-image-name`	This flag is required. Specify the Docker image name and tag, for example, `flogotestapp:1.0.0`.
`-n, --exe-name`	Specify the name of the executable file. If not provided, the app name is used as the executable file name.
`-h, --help`	Displays help for build-docker-image.
`-o, --output-directory`	Specify the directory where the executable file is

Flag	Description
	to be created. If not provided, the executable is created in the current directory.

Example

```
flogobuild build-docker-image -f "<path-to-flogo-app-file>\<.flogo/.json file>" -i "<docker-img-name>" -c "<context-name>" -n "<executable-file-name>" -o "<path-for-exe-file>" -d "<path-for-dockerfile>"
```

build-exe

Usage

```
flogobuild build-exe [flags]
```

Available Flags

Flag	Description
`-f, --app-json-file`	This flag is required. Specify the path to the Flogo application file (`.json`/`.flogo`).
`-c, --context-name`	Specify the name of the context to be used for building applications. If not provided, the default context is used.
`-n, --exe-name`	Specify the name of the executable file. If not provided, the app name is used as the executable file name.
`-h, --help`	Displays help for build-exe.
`-o, --output-directory`	Specify the directory where the executable file is to be created. If not provided, the executable is

Flag	Description
	created in the current directory.
`-p, --platform`	Specify the platform type for the app executable. Specify a value in GOOS/GOARCH format. For example, ["linux/amd64", "windows/amd64", "darwin/amd64", "darwin/arm64"].

Example

```
flogobuild build-exe -f "<path-to-flogo-app-file>\<.flogo/.json file>" -c "<context-name>" -n "<executable-file-name>" -o "<path-for-exe-file>" -p darwin/arm64
```

build-tp-deployment

Usage

```
flogobuild build-tp-deployment [flags]
```

Available Flags

Flag	Description
`-f, --app-json-files`	This flag is required. Specify the path to the Flogo application file (`.json`/`.flogo`).
`-c, --context-name`	Specify the name of the context to be used for building applications. If not provided, the default context is used.
`-h, --help`	Displays help for build-exe.
`-o, --output-directory`	This flag is required.

Flag	Description
	Specify the directory where the deployment zip is to be created.
`-t, --tags`	Specify a comma-separated list of tags for the deployment.
`-z, --zipfile-name`	Specify the name of the zip file to be created. If not provided, build.zip is used.

Example

```
flogobuild build-tp-deployment -f "<path-to-flogo-app-file>\<.flogo/.json file>" -c "<context-name>" -o "<path-for-exe-file>" -t "platform,buildZip" -z "<zip-file-name>"
```

delete-context

Usage

```
flogobuild delete-context [flags]
```

Available Flags

Flag	Description
`-c, --context-name`	This flag is required. Specify the name of the context to be deleted.
`-h, --help`	Displays help for delete-context.

Example

```
flogobuild delete-context -c <context-name>
```

list-context

Example

```
flogobuild list-context
```

package-docker-image

Usage

```
flogobuild package-docker-image [flags]
```

Available Flags

Flag	Description
`-d, --docker-file`	Specify the path to the Flogo application Dockerfile.
`-i, --docker-image-name`	This flag is required. Specify the Docker image name and tag.
`-n, --exe-name`	This flag is required. Specify the path to the executable file.
`-h, --help`	Displays help for package-docker-image.

Example

```
flogobuild package-docker-image --docker-image-name "<docker-image-name>" -
--exe-name "<path-to-exe-file>" -d "<path-to-dockerfile>"
```

set-default-context

Usage

```
flogobuild set-default-context [flags]
```

Available Flags

Flag	Description
`-c, --context-name`	This flag is required. Specify the name of the context to be set as default.
`-h, --help`	Displays help for set-default-context.

Example

```
flogobuild set-default-context -c <Context-name>
```

test-app

Usage

```
flogobuild test-app [flags]
```

Available Flags

Flag	Description
`-e, --app-exe-file`	Specify the path to the executable file to be tested. This parameter is required if the `--app-json-file` flag is not provided.
`-a, --app-json-file`	Specify the path to the Flogo application file (`.json`/`.flogo`). This parameter is required if the `--app-exe-file` flag is not provided.
`-p, --app-props-file`	Specify the path to the application property file.
`-c, --context-name`	Specify the name of the context to be used for building applications. If not provided, the default context is used.

Flag	Description
`-v, --env-vars-file`	Specify the path to the environment variable property file.
`-h, --help`	Displays help for test-app.
`-d, --output-directory`	Specify the path to the output directory.
`-o, --output-file`	Specify the name of the output file.
`-f, --test-file`	This flag is required. Specify the path to the test file.
`-t, --test-suites`	Specify a comma-separated list of test suites to run.

Example

```
flogobuild test-app --app-exe-file "<path-to-exe-file>" --test-file "<path-to-test-files>" --context-name <Context-name> --output-directory "<output-dir>" --output-file "<output-file-name>" --test-suites "<Testsuite_1,Testsuite_2>" --app-props-file "<path-to-app-prop-file>/<prop-file>" --env-vars-file "<path-to-env-file>/<env-file>"
```

App Configuration Management

Flogo allows you to externalize app configuration using app properties so that you can run the same app binary in different environments without modifying 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 modifying 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. Flogo Enterprise allows you to fetch values for app properties from Consul and override them at runtime.

Note: This section assumes that you have set up Consul and know-how Consul is used to storing 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 for your app to connect to a Consul server. You have the option to enter the values for the Consul connection parameters. You can either type in their values as JSON strings or create a file that contains the values and use the file as input.

Consul can be started with or without `acl_token`. If using an ACL token, make sure to have the ACL configured in Consul.

Using Consul

Below is a high-level workflow for using Consul with your Flogo app.

Before you begin

You must have access to Consul.

Set up Consul and understand how Consul is used to storing service configuration. For information on Consul, refer to the Consul documentation.
To use Consul to override app properties in your app (properties that were set in Flogo Enterprise):

Procedure

1. Export your app binary from Flogo Enterprise. 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 that you want to override. At runtime, the app fetches these values from the Consul and uses them to replace the default values that were set in the app.

3. **Important:** 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 Flogo Enterprise. If the property name does not match exactly, a warning message is displayed, and the app uses the default value for the property that you configured in Flogo Enterprise.
4. 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.
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 the key hierarchy is not fixed and may change based on the environment during runtime. It is also helpful in case that you want to switch to a different configuration service such as the 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`) that 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` is 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 is based on the token you specify. To protect the token, encrypt the token for the `key_prefix` where your Key resides and provides 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. Then, 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 that you can use as the SECRET. **Note:** Since special characters (such as `!|<>&`) are shell command directives, if they appear in the token string when encrypting the token, you must use a backslash (\) to escape such characters.
insecure_connection	No	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. **Default:** `False`

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.

Before you begin

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. When you 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 that tells your app the location in Consul from where to access the property values.

> ⚠ **Warning:** The Key name in Consul must be identical to its counterpart in the **Application Properties** dialog in Flogo Enterprise. If the key name does not match exactly, a warning message is displayed, and the app uses the default value that you configured for the property in Flogo Enterprise.

> ⚠ **Warning:** A single app property, for example, `Message`, is 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` is 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 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, the Key names must be identical to the app property names defined in the **Application Properties** dialog in Flogo Enterprise.

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.

For a description of the parameters, see Consul Connection Parameters.

For details on how to encrypt a value, see Encrypting Password Values.

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 is an example:

```
{
    "server_address": "http://127.0.0.1:32819",
    "key_prefix": "/dev/<APPNAME>/",
    "acl_token": "SECRET:b0UaK3bTyD9wN+ZJkmlKRmojhAv+"
}
```

Place the `consul_config.json` file in the same directory that contains your app binary.

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, 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 runs 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. For more information, see Building the App.

Overriding an App Property at Runtime

While using config management services like Consul or AWS Params store, you can update or override an app property at runtime without restarting or redeploying the app.

Note: Currently, this functionality is only available for app properties mapped in activities. It is not available for app properties in triggers and connections.

Before you begin

Set the following environment variables:

Environment Variable	Description
FLOGO_APP_PROP_RECONFIGURE=true	Specifies that app properties can be updated or overridden at runtime.
FLOGO_APP_PROP_SNAPSHOTS=true	Used along with FLOGO_APP_PROP_RECONFIGURE. If you do not want your application to pick the updated app properties dynamically for a running flow, set this variable to true. The updated values are effective only for new flows and not existing ones.

FLOGO_HTTP_SERVICE_PORT=<port number>	Specifies the service port. For apps running in TCI, you do not need to specify the port. The default is 7777.
FLOGO_APP_PROPS_CONSUL="{"server_address":"http://127.0.0.1:8500"}"	Specifies the Consul server address.

Overriding Values by Specifying New Values

Procedure

1. In the Flogo app, create an app property and map it to the activities as required.
2. Create the same key as the app property and add some value.
3. Run the app with the environment variables in the "Before you begin" section. The app takes all the configured values.
4. Update the values.
5. To reconfigure the app property values, use the API as follows:

   ```
   curl -X PUT localhost:7777/app/config/refresh
   ```

   A successful response is returned from the API.
6. Open the app property update logs to verify that the new app property values are used by the activities.

Overriding Values by Specifying New Values in the API Directly

You can specify the new values of app properties directly through the body of the reconfigure API. This method takes priority over any other resolver specified.

Example:

```
curl -X PUT -H "Content-Type: application/json" -d '{"Property_
1":"Value"}' localhost:7777/app/config/refresh
```

Important Considerations

- If the same property exists in Consul, the property from the body of the reconfigure API is used.
- Any new request on the API does not save property values provided on a previous request.
- Properties mentioned in an earlier request and not mentioned in the new request take values if present from other resolvers mentioned or the last saved value.
- Properties that are not mentioned in any resolver take the value from TIBCO Cloud Integration.

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 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 by creating a file that contains the values and using the file as input.

Using the Parameter Store

Below is a high-level workflow for using AWS Systems Manager Parameter Store with your Flogo app.

Before you begin

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 Flogo Enterprise:

1. Build an app binary that has the app properties already configured in Flogo Enterprise. For more information on building an app binary, see Building 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 the 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 are overridden. The remaining app properties 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 the 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 the access key, an encrypted value can be provided in this configuration. See Encrypting Password Values section for information on how to encrypt a string. **Note:** The encrypted value must be prefixed with SECRET: For example, `SECRET:b0UaK3bTyD9wN+ZJkmlKRmojhAv+` This configuration is optional if `use_iam_role` is

Property Name	Required	Data Type	Description
			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 the 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. **Note:** 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.
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 that 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`) that has two

Property Name	Required	Data Type	Description
			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` is 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 use the 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.
session_ token	No	String	Enter session token if you are using temporary security credentials. Temporary credentials expire after a specified interval. For more information, see the AWS documentation.

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.

Before you begin

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 the Parameter Store from where to access the property values.

> ⚠ **Warning:** The parameter name in the Parameter Store must be identical to its counterpart (app property) in the **Application Properties** dialog in Flogo Enterprise. If the parameter names do not match exactly, a warning message is displayed, and the app uses the default value that you configured for the property in Flogo Enterprise.

> ⚠ **Warning:** A single app property, for example, `Message`, is looked up by your app as either `Message` or `<param_prefix>/Message` in the Parameter Store. An app property within a hierarchy such as `x.y.z` is looked up as `x/y/z` or `<param_prefix>/x/y/z` in the 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 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, the property names must be identical to the app property names defined in the **Application Properties** dialog in Flogo Enterprise.

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":"SECRET:XXXXXXXXXX",
"region":"us-west-2",
"session_
token":"SECRET:1UBrEIezye8W1mmx7NLAiQzopmp58kUa02XdpmxYqVvkGKUrdN+wgCeH3
mxZ"
"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 connects to the Parameter Store, pulls the values for the properties, and overrides the default values that were set in the app.

Refer to the 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 is an example:

```
{
"access_key_id": "SECRET:b0UaK3bTyD9wN+ZJkmlKRmojhAv+",
"param_prefix": "/MyFlogoApp/dev/",
"secret_access_key": "SECRET:b0UaK3bTyD9wN+ZJkmlKRmojhAv+",
"region": "us-west-2",
"session_
token":"SECRET:1UBrEIezye8W1mmx7NLAiQzopmp58kUa02XdpmxYqVvkGKUrdN+wgCeH3
mxZ"
}
```

Place the `aws_config.json` file in the same directory, which contains your app binary.

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 connects to the Parameter Store, pulls the overridden app properties values from the Parameter Store, and runs 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 use the EC2 instance credentials, set `use_iam_role` to `true`. The values for `access_key_id` and `secret_access_key` are gathered from the running container. Ensure that the ECS task has permission to access the param store.

The IAM role that you use must have permissions to access the parameter(s) from the AWS Systems Manager Parameter Store. The following policy must be configured for the 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
```

AWS AppConfig

AWS AppConfig is a feature provided by AWS System Manager, which lets you create, manage, and quickly deploy application configurations. You can use AWS AppConfig to simplify the task of configuring changes in application configuration, validating the changed configurations, deploying the new configurations and monitoring it.

Using AWS AppConfig, you can override the Flogo app properties at runtime. Your Flogo app retrieves configuration data by establishing the connection with AWS AppConfig. To enable the connection between your Flogo app and AWS AppConfig, you are required to set the value of FLOGO_APP_PROPS_AWS_APPCONFIG to True. Here, the session retrieves the data from AppConfig only once at the start of the session.

Using the AppConfig

Below is a high-level work flow for using AWS Systems Manager AppConfig with your Flogo app.

Before you begin

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 AppConfig. Refer to the AWS documentation for the information on the AWS Systems Manager AppConfig.

Overview

1. Build an app executable that has the app properties already configured in Flogo. For more information on building an app executable, see Building an App Executable.

   In case of TCI, for a new app, you need to set the engine variables for the Flogo app before pushing it to TCI. For an existing app you can configure the engine variables and push the updates to the app in the TCI.

2. Configure AWS AppConfig to work with your Flogo application. To define the properties in AWS AppConfig:

   a. Create an application in AWS Appconfig to organize and manage configuration data.

   b. Select the environment of the application in the Appconfig same as that of the environment of your Flogo app.

c. Create a configuration profile.

   A configuration profile enables AWS AppConfig to access your hosted configuration versions in its stored location. You can store configurations in YAML, JSON, or as text documents in the AWS AppConfig hosted configuration store.

   Refer to AWS documentation for detailed procedure to set up the AWS AppConfig.

3. Configure the app properties that you want to override in the AppConfig. At runtime, the app fetches these values from the AppConfig and uses them to replace the default values that were set in your Flogo app.
4. Set the value of the parameter FLOGO_APP_PROPS_AWS_APPCONFIG to True to establish the connection between your Flogo app and AWS AppConfig.

Note: If you change the app properties values in AWS AppConfig, then you need to repush the app to TCI or re-execute the app executable.

AppConfig Client Configuration

IAM role that you would be using to fetch the configuration details must have permissions to access configurations from AWS AppConfig. For the same, Following policy must be configured for IAM role:

Caution: Code snippets in the PDF could have undesired line breaks due to space constraints and should be verified before directly copying and running it in your program.

```
{
 "Version": "2012-10-17",
 "Statement": [
   {
    "Sid": "VisualEditor0",
    "Effect": "Allow",
    "Action": [
      "appconfig:GetLatestConfiguration",
      "appconfig:StartConfigurationSession",
      "appconfig:ListApplications",
```

```
        "appconfig:GetApplication",
        "appconfig:ListEnvironments",
        "appconfig:GetEnvironment",
        "appconfig:ListConfigurationProfiles",
        "appconfig:GetConfigurationProfile",
        "appconfig:GetConfiguration",
        "appconfig:ListDeployments",
        "appconfig:GetDeployment"
      ],
      "Resource": "*"
     }
   ]
 }
```

To connect to the AWS Systems Manager AppConfig, provide below configuration at runtime.

Property Name	Required	Data Type	Description
FLOGO_APP_PROPS_AWS_APPCONFIG	Yes	Boolean	Set this as **True** to enable the AWS AppConfig support feature.
AWS_APPCONFIG_PROFILE_NAME	Yes	String	This is name of the configuration profile created while defining the properties in AppConfig.
AWS_APPCONFIG_ENV_NAME	Yes	String	This is name of the environment provided while creating application in the AppConfig.
AWS_APPCONFIG_APP_IDENTIFIER_NAME	No	String	Set app identifier name for AWS AppConfig. If the name is not set, it takes the name as that of your Flogo app.

Property Name	Required	Data Type	Description
			It is required only if your AWS AppConfig app identifier name does not match with the Flogo app name.
AWS_APPCONFIG_REGION	No	String	Select AWS region where your Appconfig is located. This field is not required when your app binary (executable) is running on AWS EC2 instance in the same region as that of your AppConfig region. For all other cases, you must set the region.
AWS_APPCONFIG_ACCESS_KEY_ID	No	String	If the access key ID is not provided, it is picked up by following the AWS default credentials provider chain. For flogo app deployment on TCI, you must provide this value.
AWS_APPCONFIG_SECRET_ACCESS_KEY	No	String	If the secret access key is not provided, it is picked up by following the AWS default credentials provider chain. For flogo app

Property Name	Required	Data Type	Description
			deployment on TCI, you must provide this value.
AWS_APPCONFIG_SESSION_TOKEN	No	String	Set this if you want to use your session token for AWS AppConfig API calls.
AWS_APPCONFIG_ASSUMEDROLE_ARN	No	String	Set the assume role ARN if you want to use assumed role to fetch the values from AWS AppConfig.

Tip: For sensitive fields such as ACCESS_KEY_ID, SECRET_ACCESS_KEY, and SESSION_TOKEN an encrypted value can be provided in this configuration. See the Encrypting Password Values section for information on how to encrypt a string.

Note: The encrypted value must be prefixed with SECRET: For example, SECRET:b0UaK3bTyD9wN+ZJkmlKRmojhAv+

Environment Variables

Flogo Enterprise allows you to externalize the configuration of app properties using environment variables.

Using environment variables with app properties is a two-step process:

Procedure

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.

Note: App binaries that were generated from a version of Flogo Enterprise 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 Flogo Enterprise 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.

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. The `.properties` file can be used to create a ConfigMap in Kubernetes. 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 the 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. The `.properties` file is useful when creating the ConfigMap in Kubernetes. 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` appears as `MESSAGE`. A hierarchy such as `x.y.z` appears as `X_Y_Z`.

Using a JSON File to Override App Property Values

To override an app prop using a JSON file:

Procedure

1. In the JSON file, make sure that the app property which you want to override is set as follows:
   `"<property>":"<value>"`
   For example:

   ```
   {
   "IntegerOverrideVal":453,
   "StringOverridingValue":"hello",
   "BoolValue":true
   }
   ```

   Note: Only for certificates, the format of the property must be:
   `"<property>":"<encoded_value>"`
   To get the encoded value of the contents, you can use https://www.base64encode.org/ or any other base64 encoding tool.

2. Execute the binary of the app using the `FLOGO_APP_PROPS_JSON` environment variable as follows:
   `FLOGO_APP_PROPS_JSON=/<filepath>/<JSON filename>.json ./<binary>`

Example: Overriding a Certificate Using a JSON File

You can override a server key and certificate using an app property. You would, typically, need to override a certificate if the existing certificate has expired or you want to use a custom certificate. You can directly override the certificate at runtime instead of re-configuring the app. In such a case:

Procedure

1. In the JSON file, set the `ServerKey` and `ServerCertificate` app properties as follows:

   ```
   {
   "ServerKey":"LS0tLS1CRUdJTiBQUklWQVRFIEtFWS0tLS0tCk1JSUV2Z0l",
   "ServerCertificate":"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1J",
   }
   ```

2. Execute the binary of the app using the `FLOGO_APP_PROPS_JSON` environment variable as follows:
   ```
   FLOGO_APP_PROPS_JSON=/home/john/Downloads/appPropOverride.json
   ./RestSSLService-linux_amd64
   ```

Overriding Security Certificate 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.

> ⚠ **Warning:** Using environment variables to override app properties in Lambda apps is not currently supported.

Follow these 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 `!| < > &@ `) 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`

> **Note:** When setting variables of type `password` be sure to encrypt its value for security reasons. For more information, see Encrypting Password Values.

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 matches the environment variable name for the property.

App properties that were not set as environment variables pick up the default values set for them in the app. A warning message similar to the following is displayed in the output: `<property_name> could not be resolved. Using default values.`

Example: Overriding a Certificate Using an Environment Variable

You can override a server key and certificate using an app property. You would, typically, need to override a certificate if the existing certificate has expired or you want to use a custom certificate. You can directly override the certificate at runtime instead of reconfiguring the app. In such a case:

1. Export the base64 encoded values of the content of the file in the terminal itself as follows:
   `export ServerCertificate=<base64encodedCertificateFileContent>`
   `export ServerKey=<base64encodedKeyFileContent>`
2. Set the `FLOGO_APP_PROPS_ENV=auto` environment variable as follows:
   `FLOGO_APP_PROPS_ENV=auto ./<app-binary>`

Encrypting Password Values

When entering passwords on the command line or in a file, it is always a good idea to encrypt their values for security reasons. Flogo Enterprise has a utility that you can use to encrypt passwords.

Before you begin

You must have the password to be encrypted handy 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 `! | <, >, &, `) are shell command directives, if such characters appear in the encrypted string, using double quotes around the encrypted value directs 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.

Azure Key Vault Secrets

Azure Key Vault stores and manages secrets, such as passwords and database connection strings. TIBCO Flogo® Enterprise retrieves values for Flogo app properties from Azure Key Vault Secrets and overrides them at runtime.

Integrating Azure Key Vault

To integrate with Azure Key Vault, set the following environment variables for your application:

- `FLOGO_APP_PROPS_AZURE_KEYVAULT`: Set to `true` to enable Azure Key Vault integration.
- `FLOGO_AZURE_KEYVAULT_NAME`: Specify the name of your Azure Key Vault.

Authentication Methods

Flogo supports the following authentication methods for accessing Azure Key Vault:

1. Service Principal with Secret
2. Managed Identities for Azure Resources

To configure Azure Key Vault credential management, set the following environment variables at runtime based on your authentication method:

For Service Principal with Secret

Variable	Description
`FLOGO_APP_PROPS_AZURE_KEYVAULT`	Set this to `true`.

Variable	Description
FLOGO_AZURE_KEYVAULT_NAME	Specify the Azure Key Vault name.
AZURE_TENANT_ID	The Microsoft Entra tenant (directory) ID.
AZURE_CLIENT_ID	The client (application) ID of an App Registration in the tenant.
AZURE_CLIENT_SECRET	The client secret generated for the App Registration.

For Managed Identities for Azure Resources

Variable	Description
FLOGO_APP_PROPS_AZURE_KEYVAULT	Set this to `true`.
FLOGO_AZURE_KEYVAULT_NAME	Specify the Azure Key Vault name

Note:

- Flogo always fetches the current version of the Azure Key Vault Secret.
- The identity used by your application must have at least the **Key Vault Secrets User** role to retrieve secrets from Azure Key Vault.
- If Azure Key Vault is using access policies to manage permissions, then the identity used by your application must have at least **Get** Secret permissions to retrieve secrets from Azure Key Vault.

Setting Azure Key Vault Secrets

To override the app property values, create a key-value pair for each property in Azure Key Vault Secrets. You can create standalone properties or organize them in a hierarchy.

- For a standalone property, use the property name as the secret name in Azure Key Vault. If the property name contains an underscore (_), it is replaced with a dash (-).

  For example: If the app property name is `Property_1`, then the secret lookup is done with name `Property-1` in Azure Key Vault.

- For hierarchical properties, if the property name contains dot (.) or underscore (_),

they are replaced with dash (-).

For example: If the app property name is `Group_1.Group_2.Property_1`, then the secret lookup is done with name `Group-1-Group-2-Property-1` in Azure Key Vault.

Warning: The secret name in Azure Key Vault must exactly match the property name in the Flogo Application Properties dialog (after replacing underscores and dots with dashes as described). If the names do not match or if the secret is disabled, Flogo displays a warning and uses the default property value.

Container Deployments

You can run Flogo apps as containerized apps in Docker containers and use Kubernetes to deploy, manage and scale the apps.

Kubernetes

You can package a Flogo app binary in a docker image, then push the docker image to a container registry and run the Flogo apps on a Kubernetes cluster as a pod.

For information on deploying apps in a Kubernetes environment, see Deploying Flogo apps to a Kubernetes.

Deploying Flogo Apps to Kubernetes

You can deploy your Flogo apps to a Kubernetes Cluster running locally on bare metal servers, on VMs in hybrid cloud environments, or on fully managed services provided by various cloud providers such as Amazon EKS, Azure Container Service, or Google Kubernetes Engine. Refer to the Kubernetes documentation for more information. To do so, you must create a docker image locally for your app, then push the image to a container registry. When you apply the appropriate app deployment configuration to the Kubernetes cluster, one or more docker containers get created from the docker image that is encapsulated in one or more Kubernetes pods based on the deployment configuration.

Before you begin

You must have:

- The Kubernetes cluster running on your choice of environment
- Docker 1.18.x or greater installed on your machine
- `kubectl` installed on your machine

Procedure

1. Build a docker image for your app. You can use one of the following ways to build a docker image:
   - **Using the UI:**
     a. Build a docker image using the Flogo Enterprise UI. For details, see Building the App.
     b. Tag the generated docker image from the command line:

     ```
     docker tag <image-id> <app-name>:<version>
     ```

     the app tag must be in the format, <*app-name*>:<*app-version*>.
   - **From a Linux binary:**
     a. Build a Linux binary using the `Linux/amd64` option. For details, see Building the App.
     b. Provide run permission to the app binary: `chmod +x <app-binary>`
     c. Create a docker file. For example:

     ```
     FROM <OS-version>  # for example, FROM alpine:3.7
     WORKDIR /app
     ADD <app-binary> <path-to-app-in-docker-container> # for example,
     ADD flogo-rest-linux_amd64 /app/flogo-rest
     CMD ["/app/flogo-rest"]
     ```

     d. Build the docker image using the docker file. Run the following command:

     ```
     docker build -t <app-tag> -f <path-to-Dockerfile> .
     ```

     the app tag must be in the format <*app-name*>:<*app-version*>
   - **From the CLI:**
     a. Export your app as a JSON file (for example, `flogo-rest.json`) by clicking

Export app on the flow details page.

b. Build a Linux binary for the app from the CLI. Open a command prompt and change directory to `<FLOGO_HOME>/<version>/bin` and run:

```
builder-<platform>_<arch> build -p linux/amd64 -f <path-
to-the-.json-file>
```

This generates a linux app binary.

c. Provide run permission to the app binary:

```
chmod +x <app-binary>
```

d. Create a docker file. For example:

```
FROM <OS-version>  # for example, FROM alpine:3.7
WORKDIR /app
ADD <app-binary> <path-to-app-in-docker-container> # for example,
ADD flogo-rest-linux_amd64 /app/flogo-rest
CMD ["/app/flogo-rest"]
```

e. Build the docker image using the docker file. Run the following command:

```
docker build -t <app-tag> -f <path-to-Dockerfile> .
```

The app tag must be in the format <*app-name*>:<*app-version*>

2. Run the docker image locally to verify that all looks good:

```
docker run -it -p 9999:9999 <app-tag>
```

3. Authenticate docker with the container registry where you want to push the docker image.
4. Tag the docker image by running the following command:

```
docker tag <app-tag> <CONTAINER_REGISTRY_URI>/<app-tag>
```

the app tag must be in the format, <*app-name*>:<*app-version*>

5. Push the local docker image to the container registry by running the following

command:

```
docker push <CONTAINER_REGISTRY_URI>/<app-tag>
```

Note: Refer to the documentation for your container registry for the exact commands to authenticate docker, tag docker image, and push it to the registry.

6. To deploy your app on Kubernetes, run your app by creating a Kubernetes deployment object. Follow these steps to do so:
    a. Create a YAML file. For example, the YAML file below describes a deployment that runs the `gcr.io/<GCP_PROJECT_ID>/<docker-image-name>:<tag>` docker image on the Google Cloud.

```
apiVersion: apps/v1 # for versions before 1.9.0 use
apps/v1beta2
kind: Deployment
metadata:
  name: flogo-app-deployment
spec:
  selector:
    matchLabels:
      app: flogo-app
  replicas: 2 # tells deployment to run 2 pods matching the
template
  template:
    metadata:
      labels:
        app: flogo-app
    spec:
      containers:
      - name: flogo-app
        image: gcr.io/<GCP_PROJECT_ID>/<docker-image-
name>:<tag>
        ports:
      - containerPort: 9999
```

    b. Create a Kubernetes deployment by running the following command:

```
kubectl apply -f deployment.yaml
```

Using ConfigMaps with a Flogo App

Flogo apps running in Kubernetes can use ConfigMaps for the app configuration through environment variables. When you bind the ConfigMap with your pod, all the properties in the ConfigMap get injected into the pod as environment variables. If your pod has multiple containers, you can specify the container into which you want to inject the environment variables in the `.yml` file of the app. When running the app in Kubernetes, you use the ConfigMap. You can create a ConfigMap using a `.property` file that was exported from your Flogo app.

To create a ConfigMap when running your app in Kubernetes:

> **Important:** If you update the app properties in Flogo Enterprise, you must recreate the ConfigMap and repush the app for your changes to take effect in Kubernetes.

Procedure

1. Export the Flogo app properties to a `.properties` file. Refer to the section Exporting App Properties to a File for details.
2. Update the generated `.properties` file as desired.
3. Create a ConfigMap using the `.properties` file. Run the following command:

   ```
   kubectl create configmap <name-of-configmap-file-to-be-created> --from-env-file=<exported-app-prop-filename>.properties
   ```

   For example, if your exported file name is `Timer-env.properties` and you want the generated ConfigMap to be called `flogo-rest-config` the command would be similar to the following:

   ```
   kubectl create configmap flogo-rest-config --from-env-file=Timer-env.properties
   ```

4. Update the Kubernetes deployment configuration YAML file for the app to let your app know that you want to use environment variables. Add the following:

   ```
   env:
           - name: "FLOGO_APP_PROPS_ENV"
   ```

```
          value: "auto"
        envFrom:
        - configMapRef:
            name: <name-of-the-configmap>
```

Note: Refer to the Kubernetes documentation for instructions on how to configure a pod to use ConfigMaps.

5. Build the docker image for the app binary by running the following command:

```
docker build -t  <CONTAINER_REGISTRY_URI>/<app-tag>
```

6. Push the resulting image to the container registry using the following command:

```
kubectl apply -f <appname>.yaml
```

Managing Sensitive Information Using Kubernetes Secrets

You can resolve the values of the app properties in a Flogo app deployed on Kubernetes using Kubernetes Secrets. Kubernetes secret object lets you store and manage sensitive information like passwords or keys. This section explains how a secret can be used with a Kubernetes pod.

For more information on Kubernetes secrets, refer to the Kubernetes documentation.

Configuring the Secrets

To use the Kubernetes secrets in a Flogo app, you must set `FLOGO_APP_PROPS_K8S_VOLUME` with the `volume_path` configuration parameter at runtime:

- The secret key name must match the app property name. For example, if the property is `DB_PASS`, the secret key name must be `DB_PASS`. For example:

```
echo -n 'flogo123>./DB_PASS.txt
```

```
kubectl create secret generic my-first-secret --from-file=./DB_
PASS.txt'
```

where `DB_PASS.txt` contains the password for the database and `DB_PASS` is set as a property in the Flogo app.

- If you want to use a hierarchy for your app property, ensure that you use an underscore (_) between each level instead of the dot notation in the name of the secret. For example, for an app property named x.y.z, the name of the secret must be x_y_z.

Specifying the Path of the Volume Where the Secrets are Mounted

To specify the path to the volume where the secrets are mounted, you can specify the `volume_path` parameter in a JSON file or as a JSON string.

In a JSON File

1. Set the `volume_path` parameter in a `.json` file. For example, `k8s_secrets_config.json` contains:

```
{
 "volume_path": "/etc/test"
}
```

2. Set the path to the `.json` file in the `FLOGO_APP_PROPS_K8S_VOLUME` environment variable. For example:

```
FLOGO_APP_PROPS_K8S_VOLUME=k8s_secrets_config.json
```

As a JSON String

Set the `FLOGO_APP_PROPS_K8S_VOLUME` environment variable as a JSON string as follows:

```
FLOGO_APP_PROPS_K8S_VOLUME="{\"volume_path\":\"\/etc\/test\"}"
```

Sample YAML File

Caution: Code snippets in the PDF could have undesired line breaks due to space constraints and should be verified before directly copying and running it in your program.

```
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    app: sampleapp
  name: sampleapp
  namespace: default
spec:
  template:
    metadata:
      labels:
        app: sampleapp
    spec:
      containers:
        -
          env:
            -
              name: FLOGO_APP_PROPS_K8S_VOLUME
              value: "{\"volume_path\":  \"/etc/test\"}"
            -
              name: FLOGO_APP_PROPS_ENV
              value: auto
        envFrom:
            -
              configMapRef:
               -   name: first-configmap
          image: "gcr.io/<project_name>/sampleapp:latest"
          imagePullPolicy: Always
         - name: sampleapp
          volumeMounts:
            -
              mountPath: /etc/test
              name: test
              readOnly: true
      volumes:
        -
          name: test
```

```
          secret:
            secretName: my-first-secret
```

Amazon Elastic Container Service (ECS) and Fargate

You can package a Flogo app binary in a docker image, push the docker image to Amazon ECR, then run, manage, and scale the Flogo app in Docker containers using Amazon ECS and AWS Fargate.

Deploying a Flogo App to Amazon ECS and Fargate

Procedure

1. Build a Flogo app as a docker image.
2. Push the Flogo docker image to Amazon Elastic Container Registry (ECR) as follows:
    a. Authenticate Docker to the ECR Registry using the following command. For more information, refer to the AWS documentation for Registry Authentication.

    ```
    aws ecr get-login
    ```

    b. Tag the Flogo app Docker image with the ECR registry, repository, and optional image tag name combination:

    ```
    docker tag <flogo_app_docker_image> <aws_account_
    id>.dkr.ecr.<region>.amazonaws.com/<ecr_repository_name>:<tag>
    ```

    c. Push the tagged Docker image to the ECR registry:

    ```
    docker push <aws_account_
    id>.dkr.ecr.<region>.amazonaws.com/<ecr_repository_name>:<tag>
    ```

3. Create a cluster in which to run your apps. For more information on how to create an Amazon ECS Cluster, refer to the AWS documentation for Creating Cluster.
4. Create a task definition. The task definition defines what docker image to run and how to run it. For more information on how to create a task definition, refer to the

AWS documentation available for Creating Task Definition.

5. Run the app in containers. After creating the task definition, you can open the app containers either by manually running tasks or by creating a service using the Amazon ECS Service Scheduler. For more information on how to create a service, refer to the AWS documentation available at Creating Service.

Pivotal Cloud Foundry

You can deploy a Flogo app binary to the Pivotal Application Service (PAS) of Pivotal Cloud Foundry (PCF) using the Binary Buildpack. For more information, see the section Deploying a Flogo App to Pivotal Application Service.

Deploying a Flogo App to Pivotal Application Service

After installing the Cloud Foundry Command Line Interface (cf CLI), you can push a Flogo app to the Pivotal Application Service. For more information on Pivotal Cloud Foundry, Pivotal Application Service, and its CLI, refer to the Pivotal Cloud Foundry documentation.

Before you begin

- Run the following command to ensure that the Cloud Foundry command-line client is installed successfully:

```
$ cf version
```

This command returns information about the currently installed version of the Cloud Foundry command-line client. For example:

```
cf version 6.42.0+0cba12168.2019-01-10
```

- Run the following command to authenticate yourself in the Pivotal Cloud Foundry:

```
$ cf login
```

Building a Linux Binary

From the UI

To build a Linux binary from the UI:

Procedure

1. From the UI, build a Linux binary using the `Linux/amd64` option. See the Building the App section for details.
2. Provide run permission to the app binary: `chmod +x <app-binary>`
3. Follow the steps in the appropriate section below.

From the CLI

To build a Linux binary from the CLI:

Procedure

1. Export your app as a JSON file (for example, `flogo-rest.json`) by clicking **Export app** on the flow details page.
2. Build a Linux binary for the app from the CLI. Open a command prompt and change the directory to `<FLOGO_HOME>/<version>/bin` and run:

   ```
   builder-<platform>_<arch> build -p linux/amd64 -f <path-to-
   the-.json-file>
   ```

   This generates a Linux app binary.
3. Provide run permission to the app binary: `chmod +x <app-binary>`
4. Follow the steps in the appropriate section below.

Without Using a manifest.yml File

Procedure

1. Create a temporary folder.

2. Copy the `linux/amd64` binary of the app, which you had created in Building a Linux Binary and save it to the temporary folder created in step 1.

   Note:
   - Ensure that you do not save the binary to a path that already contains other files and directories.
   - In your Flogo app, for a REST trigger, ensure that the port is set to 8080 in the trigger configuration.

3. In a command window, navigate to the path where you saved the binary and run the following command:

```
$ cf push <NAME_IN_PCF> -c './<APP_BINARY_NAME>' -b binary_
buildpack -u none
```

   For example:

```
cf push test1 -c ./Timer-linux_amd64 -b binary_buildpack -u none
```

   For the `-u` argument, depending on the health check, provide value as `none`, `port`, `http`, or `process`. For example, if the app is a REST API exposing an HTTP endpoint, use `port` after `-u`.

   Note: In your Flogo app, for a REST trigger, ensure that the port is set to 8080 in the trigger configuration.

4. After successfully deploying the app to the Pivotal Application Service, you can check the log of the app using the following command:

```
$ cf logs <APP_NAME_IN_PCF> --recent
```

Using a manifest.yml File

Procedure

1. Create a temporary folder.
2. Copy the `linux/amd64` binary of the app, which you had created in Building a Linux

Binary and save it to the temporary folder created in step 1.

> **Note:** In your Flogo app, for a REST trigger, ensure that the port is set to 8080 in the trigger configuration.

You have two options:

- If you do not mention Path in the `manifest.yml` file, you must have both `manifest.yml` and the app binary in the same directory.
- If you have the `manifest.yaml` file and the app binary in different directories, you must mention the following in the `manifest.yml` file:

```
path:  <app binary path>
```

3. Create a manifest file in YAML. The following manifest file illustrates some YAML conventions:

```
# this manifest deploys REST APP to Pivotal Cloud Foundry

applications:
- name: REST_APP
  memory: 100M
  instances: 1
  buildpack: binary_buildpack
  command: ./REST-linux_amd64
  disk_quota: 100M
  health-check-type: port
```

> **Note:** `REST-linux_amd64` indicates the name of app binary.

4. Save the `manifest.yml` file and run the following command in the same directory:

```
$ cf push
```

Result

The Flogo app is successfully pushed to the Pivotal Cloud Foundry.

Using Spring Cloud Configuration to Override App Properties

You can use Spring Cloud Configuration to override the properties of Flogo apps running on Pivotal Cloud Foundry.
To do so:

1. Create a Repository and Properties File on Github
2. Setup Spring Cloud Configuration on Pivotal Cloud Foundry
3. Use Spring Cloud Configuration Service with Flogo

Create a Repository and Properties File on Github

Procedure

1. Create a repository on Github.
2. In the repository created in step 1 above, create properties file with the following file naming convention:

   ```
   <APP_NAME>-<PROFILE>.properties
   ```

   For example, if a Flogo app name is `PCFAPP` and the profile name is `DEV`, the properties file name must be `PCFAPP-DEV.properties`.
3. Populate the `<APP_NAME>-<PROFILE>.properties` file with the key-value pairs for the overridden app properties.

Note:

- The name of the property must match the name of the app property. For example, if the app property is named `Message`, define the property in the properties file as:

```
Message="<value>"
```

- If the properties are in a group, define the property as:

```
<groupname>.<propertyname> = <value>
```

For example, if a property, `username`, is under the `email` group and its value is `xyz@abc.com`, define the property in the `.properties` file as:

```
email.username=xyz@abc.com
```

Setup Spring Cloud Configuration on Pivotal Cloud Foundry

Set up an instance of Config Server for Pivotal Cloud Foundry with the Git repository created above using Spring Cloud Services on Pivotal Cloud Foundry. Refer to Spring Cloud Services for PCF documentation for detailed instructions.

Using Spring Cloud Configuration Service with Flogo

Procedure

1. Bind the service instance of Spring Cloud Config Server to your Flogo app.
2. Navigate to the setting of the pushed app.
3. Under **User Provided Env Variables**, add the following environment variable:

```
FLOGO_APP_PROPS_SPRING_CLOUD = {"profile":"<PROFILE_NAME>"}
```

4. Restage the app and see the logs using the following command:

```
$ cf logs <APP_NAME_IN_PCF> --recent
```

Microsoft Azure Container Instances

You can deploy a Flogo app to a Microsoft Azure container instance using a Flogo app docker image. For more information, refer to the section, Deploying Flogo Apps to Microsoft Azure Container Instances.

Deploying a Flogo App to a Microsoft Azure Container Instance

Before you begin

- Create a Microsoft Azure account.
- Download and install Microsoft Azure CLI.
- Create a docker image of the Flogo app that needs to be deployed to the Microsoft Azure Container Instance.
- For information on Microsoft Azure commands, refer to the Microsoft Azure documentation.

Procedure

1. Create a new resource group using the following command:

   ```
   az group create -l <location> -n <name-of-group>
   ```

2. If you have not created an Azure Container Registry, create one using the following command. This Azure Container Registry stores all the images that are pushed to the registry.

   ```
   az acr create -n <name-of-registry> -g <name-of-group> --sku
   <pricing-tier-plan> --admin-enabled true
   ```

Note: You must set `--admin-enabled` to `true`.

3. Log in to Azure Container Registry using the following command:

```
az acr login -n <name-of-registry>
```

4. Tag and push the Flogo app docker image to Azure Container Registry using the following commands:

```
docker tag <app-tag> <CONTAINER_REGISTRY_URI>/<app-tag>
docker push <CONTAINER_REGISTRY_URI>/<app-tag>
```

5. Create an Azure Container instance using the following command:

```
az container create
-g <name-of-resource-group>
--name <name-of-container>
--image <name-of-image>
--environment-variables <name=value name=value FLOGO_APP_PROPS_
ENV=auto>
--dns-name-label <dns-name-label-for-container-group>
--ip-address Public
--ports <port-to-open>
--registry-login-server <name-of-container-image-registry-login-
server>
--registry-username <username>
--registry-password <password>
#NOTE: If--environment-variables FLOGO_APP_PROPS_ENV=auto is not
set, the environment variables are not detected at Flogo runtime.
#NOTE: IP Address must be explicitly set to Public.
```

For example:

```
az container create
-g flogodemo
--name flogoapp
--image flogoacr.azurecr.io/acs_flogo:latest
--environment-variables prop_str=azure FLOGO_APP_PROPS_ENV=auto --
dns-name-label flogoappazure
--ip-address Public
```

```
--ports 9999
--registry-login-server flogoacr.azurecr.io
--registry-username <username>
--registry-password <password>
#where prop_str is the app property defined in the flogo app which
is being overridden from this command
```

6. Get container logs using the following commands:

```
az container logs --resource-group <name-of-resource-group> --name
<name-of-container>
```

Deploying a Flogo App to a Microsoft Azure Container Instance Using a YAML File

Caution: Code snippets in the PDF could have undesired line breaks due to space constraints and should be verified before directly copying and running it in your program.

Procedure

1. Create a YAML file as follows:

```
--- apiVersion: 2018-10-01
location: <location>
name: <name-of-YAML-file>
properties:
containers:
-
name: fe-app-yaml
properties:
environmentVariables:
-
name: <name-of-app-property>
value: <value-of-app-property>
-
name: <name-of-app-property>
value: <value-of-app-property>
```

```
-
name: <name-of-app-property>
secureValue: <value-of-app-property>
#NOTE: secureValue must be used for passwords
-
name: FLOGO_APP_PROPS_ENV
value: auto
#NOTE: If the environment variable FLOGO_APP_PROPS_ENV is not set
to "auto", the environment variables are not detected at Flogo
runtime.
image: "<image>"
ports:
-
port: <port-number>
resources:
requests:
cpu: 1
memoryInGb: <memory>
imageRegistryCredentials:
-
password: <password>
server: <server>
username: <username>
ipAddress: <IP-address>
ports:
-
port: <port-number>
protocol: <protocol>
type: Public
#NOTE: IP Address must be explicitly set to Public.

osType: <OS>
tags: ~
type: <type>
```

2. Run the following commands:

```
az container create --resource-group <name-of-resource-group> --
file <name-of-YAML-file>
az container show -g <name-of-resource-group> -n <name-of-
container>
```

3. After the app is deployed, you can access the app endpoint by accessing the public IP address of the Azure container instance followed by the resource path.

```
<IP-address>:<port>/<resource-path>
```

Google Cloud Run

You can package a Flogo app binary in a docker image, push the image to Google Container Registry, then deploy the app to Google Cloud Run.

Note: Only apps with REST and GraphQL triggers work in Google Cloud Run.

Deploying a Flogo App to Google Cloud Run

Before you begin

- A Google Cloud account.

  For more information, see Google Cloud.
- Setup the Google Cloud command-line tool.

Create or import REST app

Design a new REST app using the UI or import an existing one into the UI.

Build and push a docker image to the container registry

- From the UI:
  - Create a Docker image of the app.
  - Push the Docker image to Google Container Registry.

    For more information, see Push Docker Image.
- From CLI:
  - Build a Linux/Amd64 binary using the CLI. For more information, see Building the App from the CLI.
  - Create a Docker file of the app and copy it along with the app binary.

- From the directory where the binary and docker files are placed, run the following command:

```
gcloud builds submit --tag gcr.io/[PROJECT-ID]/[IMAGE_NAME]:
[IMAGE_TAG]
```

For example:

```
gcloud builds submit --tag gcr.io/227xxx/flogo-helloworld:1.0
```

Deploy app on Cloud Run

You can deploy the app to Cloud Run using the CLI or the Console. This section describes how to deploy the app using the CLI. For more information on deploying the app using the Console, refer Cloud Run.

Procedure

1. Deploy the Flogo app using the following command:

```
gcloud beta run deploy --image gcr.io/[PROJECT-ID]/[IMAGE_NAME]:
[IMAGE_TAG]
```

For example:

```
gcloud beta run deploy --image gcr.io/227xxx/flogo-helloworld:1.0
Please specify a region:

 [1] us-central1

 [2] cancel

Please enter your numeric choice:  1

To make this the default region, run `gcloud config set run/region
us-central1`.

Service name (helloworld):

Allow unauthenticated invocations to [helloworld] (y/N)?  y
##NOTE:  At this prompt, only if you enter Y, you are allowed to
```

```
hit an endpoint without authentication.

Deploying container to Cloud Run service [helloworld] in project
[227xxx] region [us-central1]

✓ Deploying new service... Done.


  ✓ Creating Revision...


  ✓ Routing traffic...


  ✓ Setting IAM Policy...


Done.


Service [helloworld] revision [helloworld-695fa56d-97d2-46b9-b037-
2dfada50aca5] has been deployed and is serving traffic at
https://helloworld-pae7vs5yaq-uc.a.run.app
```

2. Make a call using the URL returned in the output. For example, you can make a call to the following URL returned in step 2:

```
https://helloworld-pae7vs5yaq-uc.a.run.app/greetings/Flogo
```

Red Hat OpenShift

You can package a Flogo app binary in a docker image, then push the docker image to a container registry and run the Flogo apps on Red Hat OpenShift.

Deploying a Flogo App to Red Hat OpenShift

Before you begin

- Ensure that you have a Red Hat Openshift account and that the Red Hat OpenShift environment is set up to deploy the app.
- Ensure that the Red Hat OpenShift CLI is installed on your machine.
- Ensure that the image of the Flogo app is pushed to the Red Hat Openshift internal registry or any other public registry such as Docker Hub.

Procedure

1. Build a docker image for your app. You can build a docker image in one of the following ways.
    - **Using the UI:**
        a. Build a docker image.
        b. Tag the generated docker image from the command line: `docker tag <image-id> <app-name>:<version>` The app tag must be in the format <*app-name*>:<*app-version*>
    - **From a Linux binary:**
        a. Build a Linux binary using the `Linux/amd64` option. For more information, see Building the App.
        b. Provide execute permission to the app binary: `chmod +x <app-binary>`
        c. Create a docker file. For example:

```
FROM <OS-version>  # for example, FROM alpine:3.7
WORKDIR /app
ADD <app-binary> <path-to-app-in-docker-container> # for example,
ADD flogo-rest-linux_amd64 /app/flogo-rest
CMD ["/app/flogo-rest"]
```

        d. Build the docker image using the docker file. Run the following command:

```
docker build -t <app-tag> -f <path-to-Dockerfile> .
```

        The app tag must be in the format <*app-name*>:<*app-version*>

- **From the CLI:**
    a. Export your app as a JSON file (for example, `flogo-rest.json`) by clicking **Export app** on the flow details page.
    b. Build a Docker image containing the app using the builder command from the CLI. Open a command prompt and change directory to `<FLOGO_HOME>/<version>/bin` and run:

```
builder-<platform>_<arch> build -f <path-to-the.json-file>
-docker -n <docker_image_name>:<tag>
```

For example:

```
builder_linux_amd64 build -f flogo-rest.json -docker -n
flogo-rest:v1
```

For more information on the builder command, refer to the section, Builder command.

2. Run the docker image locally to verify that everything is fine:

```
docker run -it -p 9999:9999 <app-tag>
```

3. Authenticate docker with the container registry where you want to push the docker image.
4. Tag the docker image. Run:

```
docker tag <app-tag> <CONTAINER_REGISTRY_URI>/<app-tag>
```

The app tag must be in the format <*app-name*>:<*app-version*>

5. Push the local docker image to the container registry by running the following command:

```
docker push <CONTAINER_REGISTRY_URI>/<app-tag>
```

> **Note:** Refer to the documentation for your container registry for the exact commands to authenticate docker, tag docker image, and push it to the registry.

6. Login to Openshift from command line:

```
oc login --token=<Your token> --server=https://<host
address>:<port>
```

For example:

```
oc login --token=<Your token> --server=https://api.ca-central-
1.starter.openshift-online.com:6443
```

7. Create a project in Red Hat OpenShift:

```
oc new-project <PROJECT_NAME>
```

8. Deploy the app on Red Hat Openshift using a YAML file. For a sample YAML file, see Sample YAML File: Red Hat OpenShift.

```
oc create -f <YAML filename>
```

9. To get information about pods, run the following command:

```
oc get pods
```

The following is a sample output of the command:

```
# oc get pods
NAME                              READY     STATUS    RESTARTS   AGE
flogo-graphql-599b9b4947-z4wqv    1/1       Running   0          3d
http-config-68595b747c-f7k6s      1/1       Running   0          23h
rest-basic-app-tmg5r              1/1       Running   0          1d
```

10. To get the logs of a particular pod, run the following command:

```
oc logs <pod name>
```

The following is a sample output of the command:

```
# oc logs flogo-graphql-599b9b4947-z4wqv
TIBCO Flogo® Runtime - 2.7.0 (Powered by Project Flogo™ - master)
TIBCO Flogo® connector for General -  1.1.0.227
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
     Starting TIBCO Flogo® Runtime
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2019-11-02T03:33:46.191Z INFO [flogo.trigger.graphql] - Initializing GraphQL Trigger - [GraphQLTrigger]
2019-11-02T03:33:46.192Z INFO [flogo.trigger.graphql] - Starting GraphQL Server...
2019-11-02T03:33:46.192Z INFO [flogo.trigger.graphql] - Secure:[false] Port:[7879] Path:[/graphql]
2019-11-02T03:33:46.192Z INFO [flogo.engine] - Starting app [ graphql-app ] with version [ 1.1.0 ]
2019-11-02T03:33:46.192Z INFO [flogo.engine] - Engine Starting...
2019-11-02T03:33:46.192Z INFO [flogo.engine] - Starting Services...
2019-11-02T03:33:46.192Z INFO [flogo] - ActionRunner Service: Started
2019-11-02T03:33:46.192Z INFO [flogo.engine] - Started Services
2019-11-02T03:33:46.192Z INFO [flogo.engine] - Starting Application...
2019-11-02T03:33:46.192Z INFO [flogo] - Starting Triggers...
2019-11-02T03:33:46.193Z INFO [flogo] - Trigger [ GraphQLTrigger ]: Started
2019-11-02T03:33:46.193Z INFO [flogo] - Triggers Started
2019-11-02T03:33:46.193Z INFO [flogo.engine] - Application Started
2019-11-02T03:33:46.193Z INFO [flogo.engine] - Engine Started
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
     Runtime started in 9.821752ms
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2019-11-02T03:33:46.193Z INFO [flogo] - Management Service started successfully on Port[7777]
```

11. To access the endpoint of an app, run the following command:

    ```
    oc get svc -o wide
    ```

    The following is a sample output of the command:

```
oc get svc -o wide
NAME            TYPE           CLUSTER-IP      EXTERNAL-IP                                                              PORT(S)                       AGE       SELECTOR
http-config     LoadBalancer   172.30.160.35   ac64e203efeeb1-597152199.ca-central-1.elb.amazonaws.com   80:31970/TCP,8090:32445/TCP   23h       app=http-config
rest-basic-app  LoadBalancer   172.30.191.133  a852474a5fedd1-987646022.ca-central-1.elb.amazonaws.com   80:31277/TCP                  1d        app=rest-basic-app
```

12. From the output, note the external IP and port. Access the endpoint using the following URL:

    ```
    http:<external IP>:<port>/<resource_context_path>
    ```

Sample YAML File: Red Hat OpenShift

> **Caution:** Code snippets in the PDF could have undesired line breaks due to space constraints and should be verified before directly copying and running it in your program.

The following is a sample YAML file for a REST app:

```
apiVersion: v1
kind: Service
metadata:
```

```
  name: flogo-rest
  labels:
    app: flogo-rest
spec:
  type: LoadBalancer
  ports:
  - port: 80
    targetPort: 9999
    name: app
  selector:
    app: flogo-rest
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: flogo-rest
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: flogo-rest
    spec:
      containers:
      - name: flogo-rest
        image: <DOCKER_REPOSITORY_NAME>/<APP_IMAGE_NAME>
        ports:
        - containerPort: 9999
```

Serverless Deployments

Developing for Lambda

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.

Note: AWS Lambda is supported on the Linux platform only.

To create an AWS connection:

Procedure

1. In Flogo Enterprise, 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 is available for you to select in the drop-down menu when adding a **Lambda** Activity or trigger.

AWS Connection Details

To establish the connection, you must specify the following configurations in the **AWS Connector** dialog.

The **AWS Connector** dialog contains the following fields:

Field	Description
Name	Specify a unique name for the connection that you are creating. This is displayed in the connection drop-down list for all the 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**. This field is not supported in TIBCO Flogo® Connector for Amazon Glacier.

Field	Description
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 AWS connection.
Authentication Type	Select one of the following authentication types as required: • AWS Credentials: Use this authentication to connect to AWS resources using access key, secret key, and assumed role. • Default Credentials: Use this authentication to use a role configured AWS resource such as EC2, ECS, or EKS without configuring the AWS credentials. Credentials are loaded using the AWS default credentials provider chain. **Note:** To use **Default Credentials** as the **Authentication Type** in TIBCO Flogo® Connector for Amazon SQS and AWS Lambda, create an AWS connection using the **Authentication Type** as **AWS Credentials** and override **AWS Credentials** to **Default Credentials** at runtime.
Access key ID	Access key ID of the AWS account (from the **Security Credentials** field of IAM Management Console). For more information, 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 more information, see the AWS documentation.
Session token	(Optional) Enter session token if you are using temporary security credentials. Temporary credentials expire after a specified interval. For more information, 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

Field	Description
	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 more information, see the AWS documentation.

Creating a Flow with Receive Lambda Invocation Trigger

The **Receive Lambda Invocation** trigger allows you to create a Flogo flow to create and deploy as a Lambda function on AWS.

Refer to the "Receive Lambda Invocation Trigger" section in the *TIBCO Flogo® Enterprise Activities, Triggers, and Connections Guide* for details on the trigger.

To create a flow with the **Receive Lambda Invocation** trigger:

Procedure

1. Create an app in Flogo.
2. Click the `app name` on the Apps page to open it.
3. Click **Create a Flow**.

   The **Create a Flow** dialog box opens.
4. Enter a name for the flow in the **Flow Name** text box.

   Flow names within an app must be unique.
5. Optionally, enter a brief description of what the flow does in the **Description** text box and click **Save**.

A flow gets created. Click the `flow name` to open the flow page.

6. From the **Triggers** palette, select **Receive Lambda Invocation** and drag it to the triggers area.
7. To configure a trigger, enter the JSON schema or JSON sample data for the operation. This is the schema for the request payload.
8. Click **Continue**.

   A flow beginning with the **ReceiveLambdaInvocation** trigger gets created.
9. Click the **ReceiveLambdaInvocation** trigger tile and configure its properties. See the "ReceiveLambdaInvocation" section in the *TIBCO Flogo® Enterprise Activities, Triggers, and Connections Guide* for details.

Deploying a Flow as a Lambda Function on AWS

After you have created the flow, you can deploy it as a Lambda function on AWS.

Before you begin

Note the following points:

- The flow must be configured with the **ReceiveLambdaInvocation** trigger.
- If the execution role name is not provided in the **ReceiveLambdaInvocation** trigger, then the Lambda function is created with the default **AWSLambdaBasicExecutionRole** role. It has the following Amazon CloudWatch permissions:
  - `Allow: logs:CreateLogGroup`
  - `Allow: logs:CreateLogStream`
  - `Allow: logs:PutLogEvents`

  If a non-existing execution role is provided, then the user whose AWS credentials are used in the AWS connection should have the following permissions:
  - `iam:CreateRole`
  - `sts:AssumeRole`

  To deploy a Flogo app as a Lambda function, user role can have access to following `AWSLambda_FullAccess` policy which has all the required access.

To deploy a flow as a Lambda function on AWS:

Procedure

1. Build your Flogo app (*<myApp>*) with the `Linux/amd64` target. This is because Lambda deployments are Linux-based and building the binary for `Linux/amd64` generates the appropriate artifact to deploy in your AWS Lambda function. Refer to Building the App for details on how to build an app.
2. Add execution permission to the native `Linux/amd64` executable file that you built. Run `chmod +x <myApp>-linux_amd64`
3. You can deploy the `<myApp>-linux_amd64` in one of two ways:
   - If you are using a Linux environment to design, build, and deploy your apps, you can directly run the following command:

     ```
     <LambdaTriggerBinary> --deploy lambda --aws-access-key <secret_key>
     ```

     For example, `myApp-Linux64 --deploy lambda --aws-access-key xxxxxxxxx`

     > **Note:** Ensure that the `aws-access-key` is identical to the one configured in the Flogo UI for the selected AWS Connection. This is used for validation with the `aws-access-key` configured as part of the AWS Connection within the UI and the value provided here does not overwrite the `aws-access-key` used while designing the app.

     This approach of deploying to AWS Lambda works only on Linux platforms.
   - If you are using a non-Linux environment to design, build, and deploy apps, then use this approach:
     a. Build your Flogo app *(<myApp>)* with the *Linux/amd64* target.
     b. Rename the Flogo executable file to bootstrap. This is mandatory per new `provided.al2` and `provided.al2023` runtimes.
     c. Compress the executable file and rename it to `<myFunctionName>.zip`.
     d. From the AWS Lambda UI, create a Lambda function with Amazon Linux 2023 runtime.
     e. Create a role or attach an existing role in the Execution role.
     f. Click **Create function**.
     g. Go to **Code source**, click **Upload from** and upload the compressed file.

     After successful deployment, the Lambda function is created in the AWS Lambda

console.

- To override app properties used in a Lambda app during runtime, create a `.properties` or `.json` file containing the properties and their values to override, then use the command:

  ```
  ./<Lambda-app-name> --deploy --env-config <app-property-file-name>.properties
  ```

  For example:
  `./MyLambdaApp --deploy --env-config MyLambdaApp-env.properties`
  where *MyLambdaApp* is the Lambda app name and`MyLambdaApp-env.properties` is the properties file name.
  All properties in the `.properties` or `.json` file are passed to Lambda as environment variables.

Deploying a Flow as a Lambda Function on AWS using AWS CLI

Procedure

1. Build your Flogo App (<*myApp*>) with the `Linux/amd64` target. This is because Lambda deployments are Linux-based and building the binary for `Linux/amd64` generates the appropriate artifact to deploy in your AWS Lambda function.
2. Rename the Flogo executable to bootstrap. This is mandatory as per new `provided.al2` and `provided.al2023` runtimes.
3. Compress the executable file and rename it to `myFunction.zip`.
4. Run the AWS CLI:

   ```
   aws lambda create-function --function-name myFunction \
   --runtime provided.al2023 --handler bootstrap \
   --architectures x86_64 \
   --role arn:aws:iam::111122223333:role/lambda-ex \
   --region us-west-2 \
   --zip-file fileb://myFunction.zip
   ```

Creating a Flow with AWS API Gateway Lambda Trigger

The **AWS API Gateway Lambda** trigger allows you to invoke Lambda functions as REST APIs. A flow created in an app using the AWS API Gateway trigger is deployed as a Lambda function.

Refer to the "AWS API Gateway Lambda Trigger" section in the *TIBCO Flogo® Enterprise Activities, Triggers, and Connections Guide* for details on the trigger.

To create a flow with the **AWS API Gateway Lambda** trigger:

Procedure

1. Create an app in Flogo.
2. Click the `app name` on the Apps page to open it.
3. Click **Create a Flow**.

   The **Create a Flow** dialog box opens.
4. Enter a name for the flow in the **Flow Name** text box.

   Flow names within an app must be unique.
5. Optionally, enter a brief description of what the flow does in the **Description** text box and click **Next**.

   A flow gets created. Click the `flow name` to open the flow page.
6. From the **Triggers** palette, select **Receive Lambda Invocation** and drag it to the triggers area.
7. Provide the method, resource path, and JSON schema for the operation.
8. Click **Continue**.

   A flow beginning with the **AWSAPIGatewayLambda** trigger is created.
9. Click **Copy schema** or **Just add the trigger**.
10. Click the **AWSAPIGatewayLambda** trigger tile and configure its properties. See the "AWS API Gateway Lambda Trigger" section in the *TIBCO Flogo® Enterprise Activities, Triggers, and Connections Guide* for details.

What to do next

Deploy the flow on AWS. For instructions on how to do so, see Deploying a Flow as a Lambda Function on AWS.

Creating a Flow with S3 Bucket Event Lambda Trigger

The **S3 Bucket Event Lambda** trigger allows you to create a flow using the operations or events that are performed on an S3 bucket trigger, a Lambda function.

Note: Creating a new event or updating an existing event in the S3 Bucket Event Lambda trigger and re-pushing the app deletes existing Events on AWS S3.

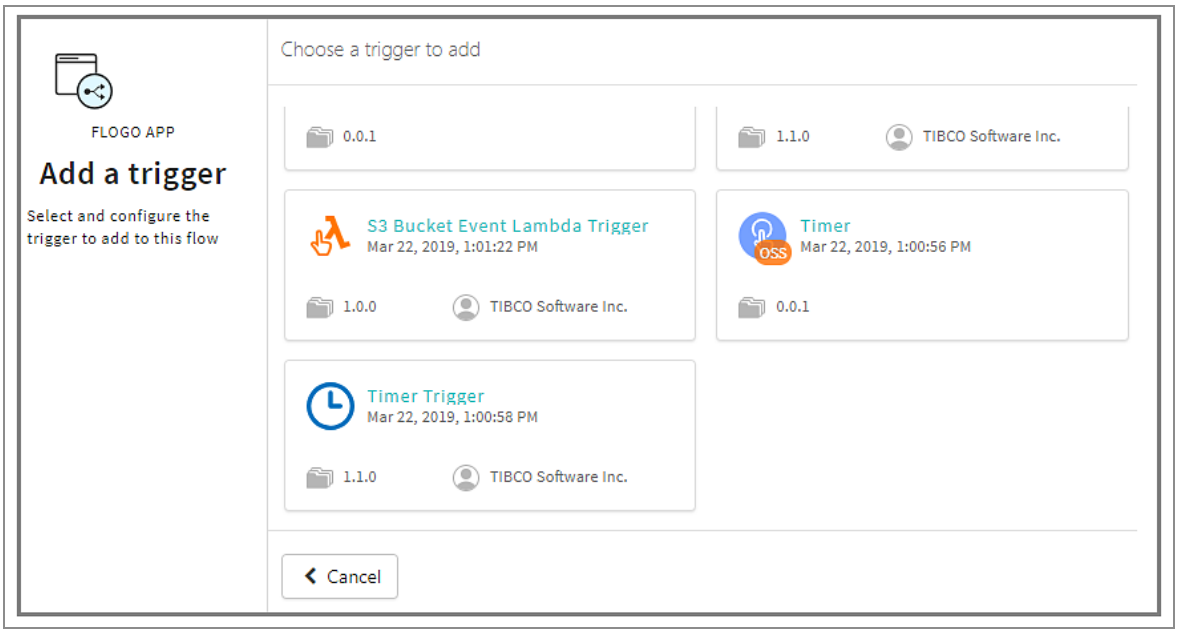
Refer to the "S3 Bucket Event Lambda Trigger" section in the *TIBCO Flogo® Enterprise Activities, Triggers, and Connections Guide* for details on the trigger. To create a flow with the **S3 Bucket Event Lambda** trigger:

Procedure

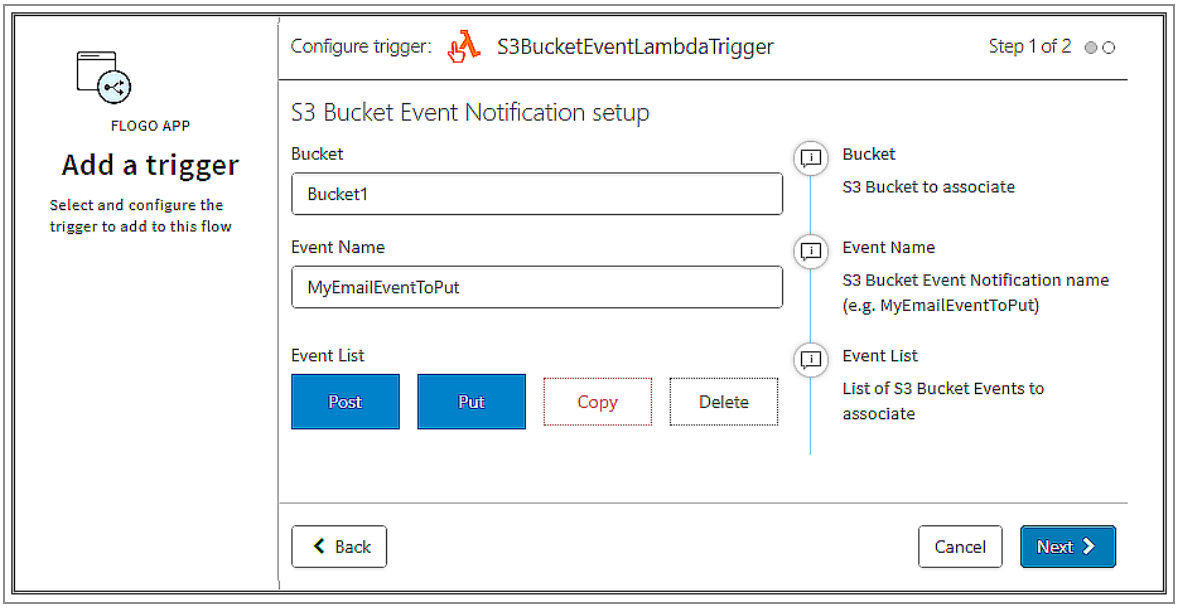
1. Create an app in Flogo Enterprise.
2. Click the app name on the apps page to open its page.
3. Click **Create a Flow**.

   The **Create a Flow** dialog box opens.
4. Enter a name for the flow in the **Flow Name** text box.

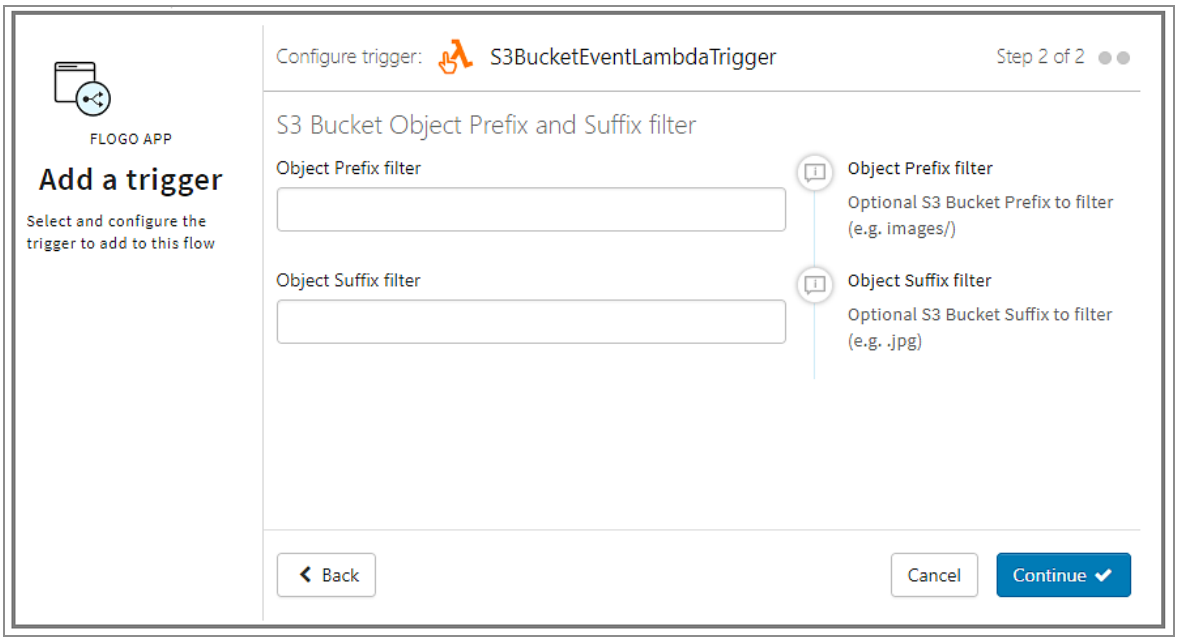
   Flow names within an app must be unique. An app cannot contain two flows with the same name.
5. Optionally, enter a brief description of what the flow does in the **Description** text box and click **Next**.
6. Click **Start with a trigger**.
7. Under **Choose a trigger to add**, click **S3 Bucket Event Lambda Trigger**.



8. Provide the bucket name, event name, and the list of events to be performed. See the "S3 Bucket Event Lambda Trigger" section in the *TIBCO Flogo® Enterprise Activities, Triggers, and Connections Guide* for details.



9. Provide any prefix or suffix object filters.



10. Click **Continue**.

    A flow beginning with the **S3 Bucket Event Lambda** trigger is created.

11. Click **Copy schema** or **Just add the trigger**.
12. Click the **S3 Bucket Event Lambda** trigger tile and configure its properties. See the "S3 Bucket Event Lambda Trigger" section in the *TIBCO Flogo® Enterprise Activities, Triggers, and Connections Guide* for details.
13. Create a flow containing the Business logic of the Lambda function that you want to trigger using the S3 Bucket Event Lambda trigger.

What to do next

Deploy the flow on AWS. For instructions, see Deploying a Flow as a Lambda Function on AWS.

Deploying a Flogo App to Microsoft Azure Functions

After you have designed a Flogo app or imported an existing one, you can deploy it to Microsoft Azure Functions as a custom Docker container. You can do this by using the Microsoft Azure portal or do it by using the CLI.

Before you begin

Make sure you have a Microsoft Azure account with an active subscription and you can log in to the Azure Portal. For more information on getting a Microsoft Azure account, see Microsoft Azure.

Creating the Azure Function App in the Azure Portal

Before you begin

Install the following:

- To push images to the Azure Container registry, install the latest version of Azure CLI.
- Install Docker. For the supported versions, see the *Readme*.

Procedure

1. Build a Docker image of your Flogo app.
2. If you are using an Azure container registry, log in to the repository created on the Azure container registry.
3. Tag and push the Docker image of the Flogo app to the repository in the Azure container registry. For example:

   ```
   docker tag flogo/hello-world:latest myregistry.azurecr.io/flogo-hello-world:latest
   ```

4. In the Azure portal, create a new Azure Function app. While creating the Azure Function app, in the **Instance details** dialog box, select **Docker Container** as the **Publish mode**.
5. After the Azure Function app is created, go to **Settings > Container Settings (Classic)** on the left navigation pane.
6. On the right pane, select the image **Source**, enter other details, and click **Save**.

> **Tip:** If you select **Registry Settings** > **Registry Source** as **Azure Container Registry**, and you face issues while selecting the **Registry**, verify the Repository permissions for the repository created for the Flogo app. Update the **Azure Container Registry** and enable Admin user; this enables the Azure function to access the images in the repositories.

7. If you are using a trigger port other than 80 or 8080, navigate to **Settings > Configuration** and click **New application setting**. Specify:
   - **Name**: `WEBSITES_PORT`
   - **Value**: `<your trigger port>`
8. Click **Save**. The app is restarted and changes made are reflected in the app.
9. To copy the URL for your app and check whether it is working, go to the **Overview** menu.

> **Note:** Do NOT add the port declared in the trigger settings to the URL. If the URL does not work, restart the app manually.

10. For an app with app properties, to override the app properties, add the following to **Configuration > Application settings**:
    - All the app properties
    - **FLOGO_APP_PROPS_ENV=auto**
11. Save the settings. The app restarts when you save the properties.

Creating the Azure Function App from the Azure CLI

Before you begin

Install the following:

- **Visual Studio Code:** For more information, see Visual Studio Code.
- **Azure Functions extension for Visual Studio Code:** For more information, see Visual Studio Code - Azure Functions.
- **Azure Functions Core Tools (version 3.x or higher):** For more information, see

Local Azure Functions.

Procedure

1. Create a new directory (for example, `flogo-func-project`) and open it.

```
cd flogo-func-project
```

2. Create a new function project using the following command:

```
func init --worker-runtime custom --docker
```

The `--docker` option generates a Dockerfile for the project.

3. Add a new function from a template using the following command:

```
func new --name <your-app-name> --template "HTTP trigger"
```

Here:

`--name` argument is the unique name of your function

`--template` argument specifies the template based on which the function is created

Example:

```
func new --name hello-world --template "HTTP trigger"
```

4. Download or build the binary for your HTTP trigger app and copy it into the directory you created earlier.

For example, copy `hello-world.json` to the `flogo-func-project` directory.

5. If it is not an executable file, make it executable by running the following command:

```
chmod +x <binary-filename>
```

6. Add the following script to your project folder with the name `start.sh`:

start.sh

```
#!/usr/bin/env sh
echo "Starting function..."
```

```
PORT=${FUNCTIONS_CUSTOMHANDLER_PORT} ./hello-world-linux_amd64
```

7. Make it executable by running the following command:

```
chmod +x start.sh
```

8. To update the default app prefix from `api` to your prefix, edit the `function.json` file. For example, you can change the prefix to `hello-world`.

function.json

```
{
"bindings":
      [
        {
        "authLevel": "anonymous",
        "type": "httpTrigger",
        "direction": "in",
        "name": "req",
        "methods": ["get", "post"],
        "route": "books/{bookID}"
        },
        {
        "type": "http",
        "direction": "out",
        "name": "res"
        }
      ]
}
```

9. To add `customHeaders` in the extensions, edit the `host.json` file:

host.json

> ⚠ **Caution:** Code snippets in the PDF could have undesired line breaks due to space constraints and should be verified before directly copying and running it in your program.

```
{
  "version": "2.0",
  "logging":
     {
     "applicationInsights":
        {
          "samplingSettings":
             {
              "isEnabled": true,
              "excludedTypes": "Request"
             }
        }
    },

  "extensionBundle":
    {
      "id": "Microsoft.Azure.Functions.ExtensionBundle",
      "version": "[2.*, 3.0.0)"
    },

  "customHandler":
    {
      "description":
          {
            "defaultExecutablePath": "start.sh",
            "workingDirectory": "",
            "arguments": []
          },
      "enableForwardingHttpRequest": true
    },

  "extensions":
   {
     "http":
      {
        "routePrefix": ""
      }
   }
}
```

10. To change the path, edit the Dockerfile:

Dockerfile

```
FROM mcr.microsoft.com/azure-functions/dotnet:3.0-appservice
ENV AzureWebJobsScriptRoot=/home/site/wwwroot \
AzureFunctionsJobHost__Logging__Console__IsEnabled=true
COPY . /home/site/wwwroot
```

Your folder structure should now look similar to the following:

```
.
├── Dockerfile
├── hello-world-app
│   └── function.json
├── hello-world-rest-trigger-linux_amd64
├── host.json
├── local.settings.json
└── start.sh
```

11. Test your app locally by running the following command:

```
func start
```

The output returns an URL for the app.

12. Test whether the app works by navigating to the URL provided in the output. For example:

```
http://localhost:7071/hello-world
```

13. Publish your app to Microsoft Azure. For more information, see Publish the project to Azure.

Deploying a Flogo App in Knative

You can create and deploy a Flogoapp as a Knative service. For information on Knative, see the Knative documentation.

A Flogoapp running inside a Docker container is called by a Knative service. For the app to be called by the Knative service, the app must be exposed over an HTTP port. In this section, a REST Trigger is used to expose the app over an HTTP port.

Before you begin

Make sure you meet the following requirements:

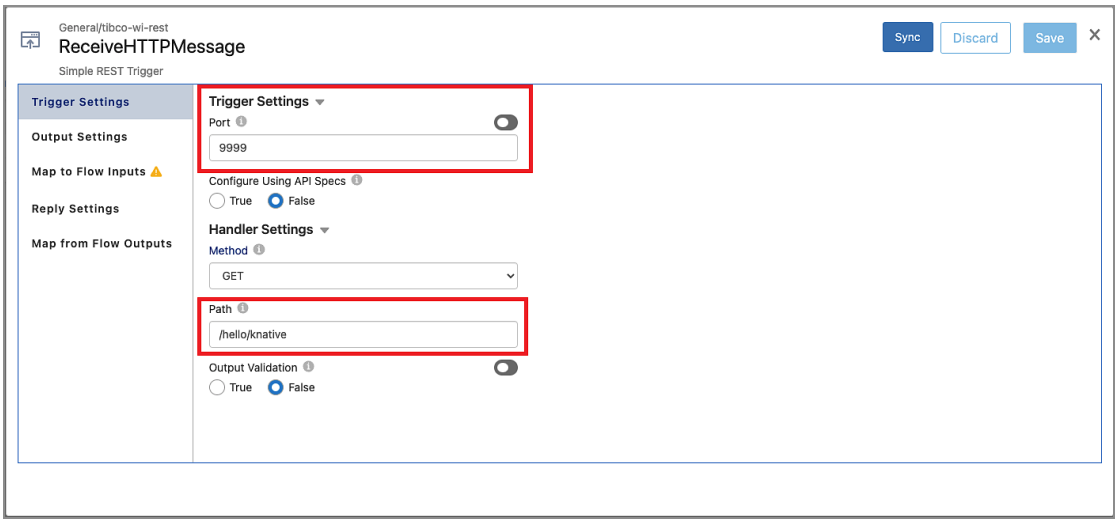
- Install the following components by using the instructions from Getting Started with Knative:
    - Kind (Kubernetes in Docker)
    - Kubernetes CLI (kubectl)
    - Knative CLI (kn)
    - Knative "Quickstart" environment
- Create a Knative service and make sure you can ping the service endpoint. For details, see Deploying your first Knative Service and Ping your Knative Service.

> **Note:** This section uses a **Knative on Kind** setup to explain the procedure. However, you can also set it up on minikube and Docker Desktop. For more information, see Setup Knative on Minikubeand Setup Knative on Docker Desktop.

Procedure

1. Configure a sample Flogo app with a REST Trigger exposed with a port. You can use the default REST Trigger port, 9999.

> **Important:**
> - Only apps with HTTP endpoints can be deployed as a Knative service. Hence, a REST Trigger is used in this procedure.
> - An app with multiple endpoints on different ports cannot be deployed as a Knative service.



2. Build the Flogo app for **Linux/amd64** platform and save the binary file locally. For more information on building the app binary, see Building the App.
3. Give executable permissions to the app binary:

```
chmod a+x <app_executable>
```

4. Build a Docker image for the Flogo app and tag it:

```
docker build --file Dockerfile -t dev.local/flogoknative:1.0.0 .
```

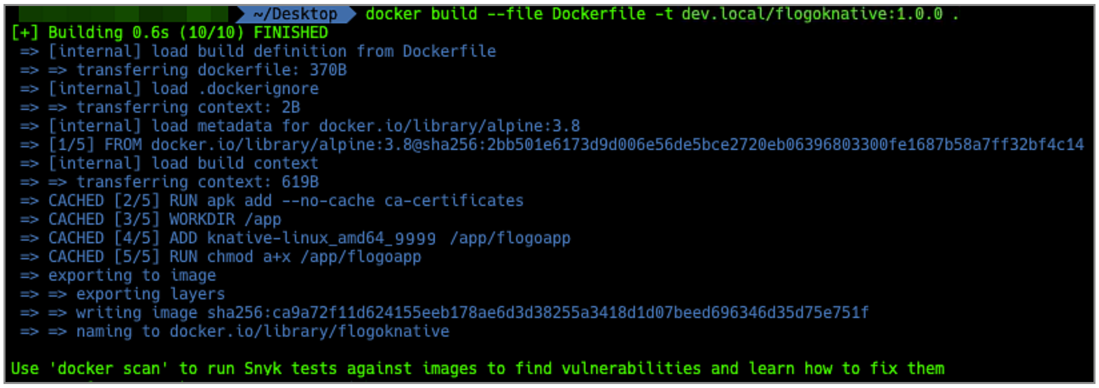
5.

> **Note:** Make sure you tag the image in the following format: `dev.local/<image name>:<tag>`
> Instead of the `latest` tag, use a tag such as 1.0.0.

Here is the sample `Dockerfile` used in the above command:

```
FROM alpine:3.8
RUN apk add --no-cache ca-certificates
WORKDIR /app
ADD <app_executable> /app/flogoapp
RUN chmod a+x /app/flogoapp
ENTRYPOINT ["/app/flogoapp"]
```

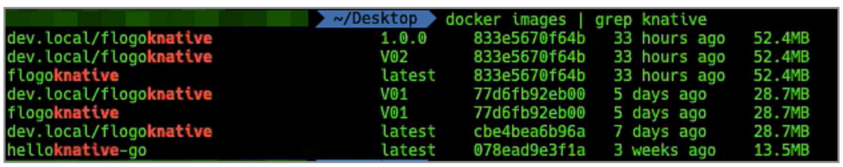
The Docker image is built:



6. Confirm that the Docker image is built successfully:

```
docker images | grep knative
```

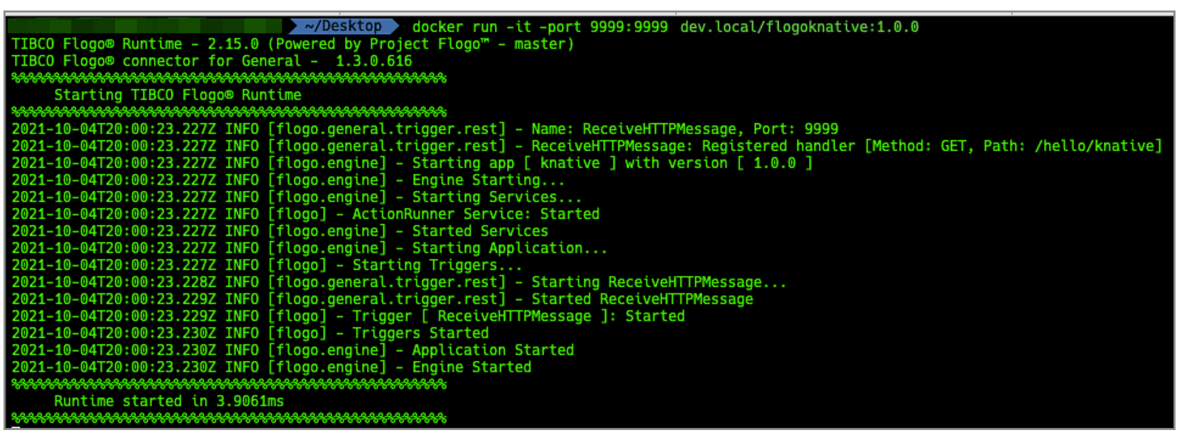
The details of the flogoknative image are displayed:



7. To test the Docker image:

```
docker run -it -p 9999:9999 dev.local/flogoknative:1.0.0
```

The Flogo runtime logs should be displayed as follows:



8. Load the Docker image into the default `knative` cluster:

```
kind load docker-image dev.local/flogoknative:1.0.0 --name knative
```

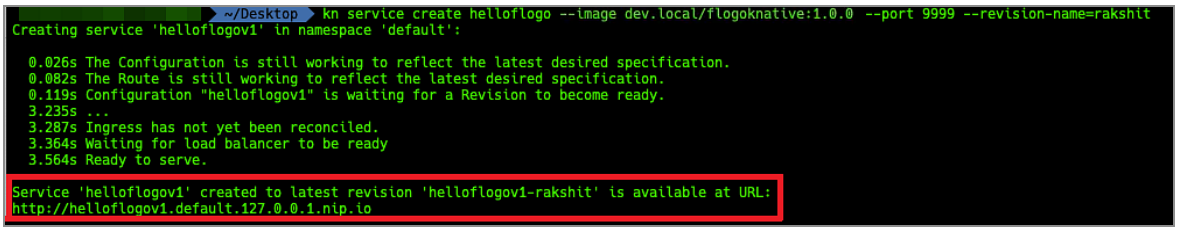
The Flogo Docker image is loaded inside the `knative` cluster and is used by Knative to create the service.

9. Create the Knative service:

```
kn service create helloflogo --image dev.local/flogoknative:1.0.0 --port 9999 --revision-name=<any revision name>
```

> **Note:** The port should be the same as what the Flogo is listening to. In this case, 9999.

A service is created and an URL is generated. You should see messages similar to the following:



The networking layer, routes, ingress, and load balancer are configured for the Knative service.

To see a list of services, execute the following command:

```
kn service list
```

NOTE: If you notice errors during any of these steps, the service is not created successfully. For troubleshooting tips, see Troubleshooting Tips.

10. Append the REST Trigger endpoint path (specified in step 1) to the generated service URL and hit the endpoint using a browser or curl. For curl, the format of the command is `curl <URL returned in previous step>/hello/knative`.
You should see the Flogo return message as the response:

Troubleshooting Tips

Error Message	Probable Solution
Errors while creating a kn service: • `configuration does not have any ready revision` • `RevisionMissing`	Check whether executable permissions are given to the Flogo app before building the Docker image.
Error while creating a kn service: `IngressNotConfigured/reconciled`	Delete your Knative cluster and recreate it using the following command: `kind delete cluster --name knative` After deleting the cluster, to reinstall it, follow the steps mentioned in Getting Started with Knative.
Error after creating the kn service and running the `kn service list` command: `RevisionMissing`	Make sure you tag the image in the following format: `dev.local/<image name>:<tag>` Instead of the `latest` tag, use a tag such as 1.0.0. After you tag the image, load it in `kind` and then create the service again.

Monitoring

This section contains information about how to monitor your apps.

About the TIBCO Flogo Enterprise Monitoring App

Using the Flogo Enterprise Monitoring app, you can monitor Flogo Enterprise apps that are running in your environment. The Flogo Enterprise Monitoring app collects metrics of flows and triggers from all running apps that are registered with it. In the UI of the app, you can visualize the metrics.

The Flogo Enterprise Monitoring app can also be used with TIBCO Flogo® Flow State Manager to collect information about the state of all run flows of a Flogo app. For more information on how to use the Flogo Enterprise Monitoring app with TIBCO Flogo® Flow State Manager, see About TIBCO Flogo® Flow State Manager .

How to Set Up and run the Flogo Enterprise Monitoring App

The Flogo Enterprise Monitoring app is available as a ZIP file. It can run as a standalone app or in a container, such as Docker or Kubernetes. However, you must run the Flogo Enterprise Monitoring app on the same container platform where the Flogo Enterprise apps are running.

How Registration Works in the Flogo Enterprise Monitoring App

Flogo Enterprise apps must be registered with the Flogo Enterprise Monitoring app to be able to view its app metrics. After an app is registered, the Flogo Enterprise Monitoring app can monitor and fetch the instrumentation statistics for the app.

The Flogo Enterprise Monitoring app stores the app registration details in a data store. Currently, the only data store supported is of the type `File`. The app registration details include app name, app host, app instrumentation port, app version, runtime version under which the app is running, and app tags. App tags are custom tags that help you provide additional information about the app. You can set them specific to an app.

Note:

- A Flogo app can have one or more instances and they can be registered with the Flogo Enterprise Monitoring app.
- Each app instance is identified as unique based on the app name and app version.

API Key for Additional Security

For additional security, the Flogo Enterprise Monitoring app can also be started using a secret key called the API key. The API key must be provided while starting the Flogo Enterprise Monitoring app and the same API key must also be provided while starting the Flogo app. The Flogo app registers with the Flogo Enterprise Monitoring app using the API key provided. If an API key is not provided, the Flogo app is not registered with the Flogo Enterprise Monitoring app.

Using the Flogo Enterprise Monitoring App

Using the Flogo Enterprise Monitoring app to monitor Flogo apps involves the following steps:

Procedure

1. Run the Flogo Enterprise Monitoring app. You can run the app in one of two ways:
   - Run the app as a standalone app. SeeRunning the Flogo Enterprise Monitoring App.
   - Run the app in Docker. SeeRunning the Flogo Enterprise Monitoring App on Docker.
2. Register the Flogo app to be monitored using the Flogo Enterprise Monitoring app. SeeRegistering an App with the Flogo Enterprise Monitoring App.
3. Access the UI of the Flogo Enterprise Monitoring app by using a browser and view the statistics of the Flogo apps. SeeViewing Statistics of Apps.

Running Flogo Enterprise Monitoring as a Standalone App

You can run the Flogo Enterprise Monitoring app as a standalone app or in a container such as Docker or Kubernetes. This section explains how to run the Flogo Enterprise Monitoring app as a standalone app.

Before you begin

The Flogo Enterprise Monitoring app is installed as described in the "Installing TIBCO Flogo® Enterprise Monitoring App" section of *TIBCO Flogo® Enterprise Installation*.

Procedure

1. Navigate to the `flogomon/bin` folder.
2. Run `startup.sh` (on macOS or Linux) or `startup.bat` (on Windows).

Result

The web server for the Flogo Enterprise Monitoring app is started.

What to do next

Register the Flogo app to be monitored with the Flogo Enterprise Monitoring app. See Registering a Flogo App with the Flogo Enterprise Monitoring App.

Running the TIBCO Flogo Enterprise Monitoring App On Docker

You can run the Flogo Enterprise Monitoring app as a standalone app or in a container such as Docker or Kubernetes. This section explains how to run the Flogo Enterprise Monitoring app on Docker.

Before you begin

The Flogo Enterprise Monitoring app is installed as described in the "Installing TIBCO Flogo® Enterprise Monitoring App" section of the *TIBCO Flogo® Enterprise Installation.*

Procedure

1. Navigate to the `flogomon` folder.
2. Build the Docker image by running the `Dockerfile` command or `Dockerfile_alpine` command as follows:

```
 docker build -t flogomon -f Dockerfile .
```

```
docker build -t flogomon -f Dockerfile_alpine .
```

3. To get a list of the most recently created Docker images, run:

```
docker images
```

4. Run the Flogo Enterprise Monitoring app using the following commands. For a list of configuration properties that can be used while running these commands, refer to Configuring the Flogo Enterprise Monitoring App.
   - **With persistent volumes and API key:**

```
docker run -e FLOGO_MON_DATA_DIR=<path where applist.json file
must be stored> -e FLOGO_FLOW_SM_ENDPOINT=http://<host>:<port>
-v <path to persistent volumes>:/opt/flogomon/data -e FLOGO_
MON_API_KEY=<secret API key> -it -p 7337:7337 <name of Docker
image of Flogo Enterprise Monitoring application>
```

   Here, `-p` specifies the port on which the Flogo Enterprise Monitoring app must be started. The default port is `7337` and it can be configured using the `FLOGO_MON_SERVER_PORT` property.

   Note: Use `-e FLOGO_FLOW_SM_ENDPOINT=http://<host>:<port>` only if you want to use TIBCO Flogo® Flow State Manager.

   For example:

```
docker run -e FLOGO_MON_DATA_DIR=/opt/flogomon/data -e FLOGO_
FLOW_SM_ENDPOINT=http://localhost:9091 -v
/home/testuser/flogo:/opt/flogomon/data -it -p 7337:7337
flogomon:latest
```

```
docker run -e FLOGO_MON_DATA_DIR=/opt/flogomon/data -e FLOGO_
FLOW_SM_ENDPOINT=http://192.168.4.12:9091 -v
/home/testuser/flogo:/opt/flogomon/data -it -p 7337:7337
flogomon:latest
```

- **Without persistent volumes and API key:**

```
docker run -e FLOGO_FLOW_SM_ENDPOINT=http://<host>:<port> -it -
p 7337:7337 <name of Docker image of Flogo Enterprise
Monitoring application>
```

Here, `-p` specifies the port on which the Flogo Enterprise Monitoring app must be started. The default port is `7337` and it can be configured using the `FLOGO_MON_SERVER_PORT` property.

Note: Use `-e FLOGO_FLOW_SM_ENDPOINT=http://<host>:<port>` only if you want to use the TIBCO Flogo® Flow State Manager.

For example:

```
docker run -e FLOGO_FLOW_SM_ENDPOINT=http://localhost:9091 -it
-p 7337:7337 flogomon:latest
```

```
docker run -e FLOGO_FLOW_SM_ENDPOINT=http://192.168.4.12:9091 -
it -p 7337:7337 flogomon:latest
```

Result

The web server for the Flogo Enterprise Monitoring app is started.

What to do next

Register the app to be monitored with the Flogo Enterprise Monitoring app. See Registering an App with the Flogo Enterprise Monitoring App.

Running the Flogo Enterprise Monitoring Application On Kubernetes

You can run the Flogo Enterprise Monitoring app as a standalone app or as a container on Kubernetes. This section explains how to run the Flogo Enterprise Monitoring app on Kubernetes.

When the Flogo Enterprise Monitoring app is started on Kubernetes, it monitors Flogo apps added to a Kubernetes cluster. If a Flogo app is found, the app is registered with the Flogo Enterprise Monitoring app. The YAML file of the app must include some configuration details required for registering the app with the Flogo Enterprise Monitoring app. For details, refer to Configurations in the Flogo App's YAML File. After a Flogo app is registered, the Flogo Enterprise Monitoring app is available in the App List on the Summary page.

Before you begin

The Flogo Enterprise Monitoring app is installed as described in the "*Installing TIBCO Flogo® Enterprise Monitoring App*" section of the *TIBCO Flogo® Enterprise Installation*.

An overview of the procedure is given below:

Procedure

1. Grant Access Using ClusterRole.
2. Configure the ServiceAccount.
3. Link the ServiceAccount to the ClusterRole using ClusterRoleBinding.
4. Link the Flogo App to the Flogo Enterprise Monitoring Application.
5. Specify configurations in the Flogo App's YAML File.

Granting Access Using ClusterRole

To monitor pods registered in the Kubernetes cluster, the Flogo Enterprise Monitoring app requires access to the List, Watch, and Get verbs for all pods across all namespaces. To grant access, and update the YAML file as shown in the following sample file.

Sample YAML file showing ClusterRole

Cluster

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: flogo-mon-cluster-role
rules:
- apiGroups: ["*"]
  resources: ["pods"]
  verbs: ["list","get","watch"]
```

Configuring the Service Account

Configure a service account for a pod as shown in the following sample YAML file.

Sample YAML file showing ServiceAccount

Service

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: flogo-mon-service-account
```

Linking the ServiceAccount to the ClusterRole

Link the ServiceAccount to the ClusterRole using ClusterRoleBinding as shown in the following sample YAML file.

Sample YAML file to add a ClusterRoleBinding

Deployment

```
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: flogo-mon-service
subjects:
- kind: ServiceAccount
  name: flogo-mon-service-account
```

```
  namespace:  default
roleRef:
  kind: ClusterRole
  name: flogo-mon-cluster-role
  apiGroup: rbac.authorization.k8s.io
```

Linking the Flogo App to the Flogo Enterprise Monitoring Application

1. Link the Flogo Monitoring Deployment to the service account created in the previous steps. See the sample deployment YAML.
2. In the Flogo Monitoring Deployment YAML, provide the `FLOGO_APP_SELECTOR` label with a value as a key-value pair. For example, `appType=flogo`.

> **Note:** All Flogo apps which are required to be linked with the Flogo Enterprise Monitoring app must specify this label.

Sample YAML file for Deployment

Deployment

> **Caution:** Code snippets in the PDF could have undesired line breaks due to space constraints and should be verified before directly copying and running it in your program.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: flogo-mon-service
spec:
  selector:
    matchLabels:
      app: flogo-mon-service
  replicas: 1
  template:
    metadata:
      labels:
```

```
      app: flogo-mon-service
  spec:
    containers:
      - name: flogo-mon-service
        image: flogomon:v1
        imagePullPolicy: Never
        env:
          - name: "FLOGO_APP_SELECTOR"
            value: "appType=flogo"
        ports:
          - containerPort: 7337
    serviceAccountName: flogo-mon-service-account
```

Configurations in the Flogo App's YAML File

To register an app with the Flogo Enterprise Monitoring app, provide the following configuration details in the app's YAML file.

- **Labels:** The deployment must have a label with the same value provided in the `FLOGO_APP_SELECTOR` environment variable. For example, if `FLOGO_APP_SELECTOR` has the value as `appType=flogo`, the Flogo app must have a label with the key as `appType` and Name as `flogo`. The Flogo Enterprise Monitoring app attempts to register the app with this label only. If the label is not provided, the app is ignored.
- **Annotations:** The following annotations are mandatory:
    - **app.tibco.com/metrics**: Setting this annotation to `true` registers the app with the Flogo Enterprise Monitoring app and enables the metrics collection on the app. Setting the annotation to `false` deregisters it from the Flogo Enterprise Monitoring app and turns off the metrics collection.
    - **app.tibco.com/metrics-port**: Provide the HTTP port for the app. This port must be the same as the one specified by the `FLOGO_HTTP_SERVICE_PORT` environment variable. If an invalid value is set, the app is ignored.

Sample YAML File

App

> **Caution:** Code snippets in the PDF could have undesired line breaks due to space constraints and should be verified before directly copying and running it in your program.

```
apiVersion: v1
kind: Service
metadata:
  name: flogoapp
  labels:
    app: flogoapp
spec:
  type: LoadBalancer
  ports:
    - port: 9999
      protocol: TCP
      name: appport
      targetPort: 9999
  selector:
    app: flogoapp
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: flogoapp
spec:
  selector:
    matchLabels:
      app: flogoapp
  replicas: 2
  template:
    metadata:
      labels:
        app: flogoapp
        appType: flogo
      annotations:
        app.tibco.com/metrics: 'true'
        app.tibco.com/metrics-port: '7777'
    spec:
      containers:
        - name: flogoapp
```

```
          image: flogoapp:v1
          imagePullPolicy: Never
          ports:
            - containerPort: 9999
            - containerPort: 7777
          env:
            - name: "FLOGO_HTTP_SERVICE_PORT"
              value: "7777"
```

Configuring the Flogo Enterprise Monitoring App

The following properties can be set when running the Flogo Enterprise Monitoring app as described in Running the Flogo Enterprise Monitoring App.

Note: These properties can also be set in the `flogomon/config/config.env` file. If you have set the properties while starting the app, the values in the `config.env` file are ignored, and the values specified during the startup take precedence.

Property	Description
FLOGO_MON_DATA_DIR	The Flogo Enterprise Monitoring app uses a file-based data store. This property provides the folder where the `applist.json` file must be stored. If you run a Docker app with persistent volumes, the `applist.json` is created at the location specified as persistent volume. **Default value:** User Home
FLOGO_MON_RETRY_INTERVAL	The interval (in seconds) after which the Flogo Enterprise Monitoring app retries to ping all instances of the Flogo app registered with the Flogo Enterprise Monitoring app. For example, if an app is down or the network is slow, the Flogo Enterprise Monitoring app tries to collect monitoring data after the value specified in this property. **Default value:** 30s

Property	Description
FLOGO_MON_RETRY_COUNT	Number of times the Flogo Enterprise Monitoring app retries to ping all the instances before removing the instance from the datastore. For example, if an app is down or the network is slow, the Flogo Enterprise Monitoring app tries to collect monitoring data the number of times specified in this property. **Default value:** 5
FLOGO_MON_API_KEY	The API Key that is used by the Flogo app to register with the Flogo Enterprise Monitoring app. The API key must be provided when starting the Flogo Enterprise Monitoring app and the same API key must also be provided when starting the app. The app registers with the Flogo Enterprise Monitoring app using the API key provided. If an API key is not provided, the app is not registered with the Flogo Enterprise Monitoring app. **Default value:** Blank
FLOGO_MON_SERVER_PORT	The port on which the Flogo Enterprise Monitoring app must be started. **Default value:** 7337
FLOGO_MON_LOG_LEVEL	The log level for the Flogo app. **Default value:** INFO
Properties related to TIBCO Flogo® Flow State Manager	

FLOGO_FLOW_SM_ENDPOINT	The endpoint of Flogo Flow State Manager. The format to set the property is:
	
	`FLOGO_FLOW_SM_ENDPOINT=http://<host>:<port>`
	
	For example:
	
	`FLOGO_FLOW_SM_ENDPOINT=http://localhost:9091`
	
	Note: This property needs to be set when starting the app binary after Flogo Flow State Manager is up and running.

Registering a Flogo App with the Flogo Enterprise Monitoring App

After a Flogo app is registered with the Flogo Enterprise Monitoring app, the collection of instrumentation statistics starts automatically. To register a Flogo app with the Flogo Enterprise Monitoring app, start the app with the following properties:

- `FLOGO_HTTP_SERVICE_PORT=<instrumentation port>`: This property specifies the port required to enable the app instrumentation.
- `FLOGO_APP_MON_SERVICE_CONFIG`: This property specifies details of the Flogo Enterprise Monitoring app to the Flogo app.

```
FLOGO_APP_MON_SERVICE_CONFIG={\"host\":\"<Host of Flogo Enterprise
Monitoring app>\",\"port\":\"<Port of Flogo Enterprise Monitoring
app>\",\"tags\":[\"<Tag 1>\",\"<Tag 2>\"],\"apiKey\":\"<API Key>\"}
```

Option	Description
Host	Host of the Flogo Enterprise Monitoring app.
Port	Port of the Flogo Enterprise Monitoring app.
Tags (Optional)	Custom tags that help you provide additional information about the Flogo app; you can set them specific to an app. For example, you can specify whether it is a REST app or whether it is running in Kubernetes, and so on.
apiKey (Optional)	For additional security, the Flogo Enterprise Monitoring app can also be started using a secret key called API key. The API key must be provided while starting the Flogo Enterprise Monitoring app and the same API key must also be provided while starting the Flogo app. The app registers with the Flogo Enterprise Monitoring app using the API key provided. If an API key is not provided, the app is not registered with the Flogo Enterprise Monitoring app.

Examples

- If the Flogo Enterprise Monitoring app is running on `localhost` on port `7337` and the app instrumentation port is `7777`, start the Flogo app as:

```
$FLOGO_HTTP_SERVICE_PORT=7777 FLOGO_APP_MON_SERVICE_CONFIG="
{\"host\":\"localhost\",\"port\":\"7337\"}" ./App1
```

- If the Flogo Enterprise Monitoring app is running on `localhost` on port `7337`, the app instrumentation port is `7777`, and you want to start the Flogo Enterprise Monitoring app based on an API Key APIkey1, start the app as:

```
$FLOGO_HTTP_SERVICE_PORT=7777 FLOGO_APP_MON_SERVICE_CONFIG="
{\"host\":\"localhost\",\"port\":\"7337\",\"apiKey\":\"<value
specified when starting the Flogo Enterprise Monitoring app>\"}"
./app_linux_amd64
```

- If the Flogo Enterprise Monitoring app is running on `localhost` on port `7337`, the app instrumentation port is `7777`, and you want to provide additional tags (named `onpremise` and `testing`), start the app as:

```
$FLOGO_HTTP_SERVICE_PORT=7777 FLOGO_APP_MON_SERVICE_CONFIG="
{\"host\":\"localhost\",\"port\":\"7337\",\"tags\":
[\"onpremise\",\"testing\"]}" ./App1
```

- On Microsoft Windows, if the Flogo Enterprise Monitoring app is running on `localhost` on port `3000` and the app instrumentation port is `7775`, start the app as:

```
set FLOGO_HTTP_SERVICE_PORT=7775
set FLOGO_APP_MON_SERVICE_CONFIG=
{"host":"localhost","port":"3000","appHost":"instance1"}
flogo-windows_amd64.exe
```

- On Linux and Mac, if the Flogo Enterprise Monitoring app is running on `localhost` on port `7337`, the app instrumentation port is `7777`, start the app as:

```
$FLOGO_HTTP_SERVICE_PORT=7777 FLOGO_APP_MON_SERVICE_CONFIG="
{\"host\":\"localhost\",\"port\":\"7337\",\"apiKey\":\"<value
specified when starting the Flogo Enterprise Monitoring app>\"}"
```

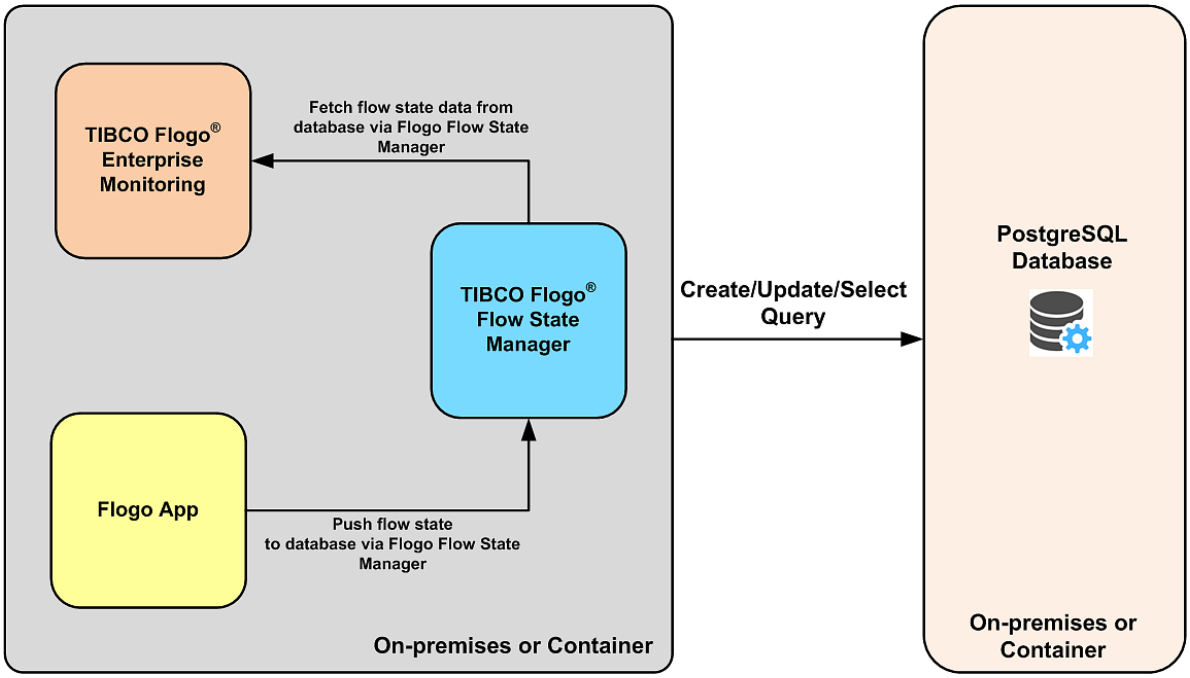
```
./app_linux_amd64
```

What to do next: View the statistics of the app on the UI of the Flogo Enterprise Monitoring app. See Viewing Statistics of Apps.

About TIBCO Flogo® Flow State Manager

Using Flogo® Flow State Manager and the TIBCO Flogo® Enterprise Monitoring App, you can collect information about the state of all executed flows of a Flogo app.

Flogo Flow State Manager acts as an interface between a Flogo app and the TIBCO Flogo® Enterprise Monitoring application. It collects data from a Flogo app and then persists the collected data to a supported database (currently, PostgreSQL). When it receives a request from the TIBCO Flogo® Enterprise Monitoring application, Flogo Flow State Manager collects data from the database and passes it on to the TIBCO Flogo® Enterprise Monitoring application for displaying on the UI.



Flogo Flow State Manager is available as a compressed file. For more information about installing Flogo Flow State Manager, see *TIBCO Flogo® Enterprise Installation*.

For more information about TIBCO Flogo® Enterprise Monitoring App, see About the TIBCO Flogo Enterprise Monitoring App.

Using Flogo Flow State Manager

Before you begin

Make sure you meet the following requirements:

- Install the PostgreSQL database. For more information, see PostgresSQL.
- Optionally, download and install a PostgreSQL management tool such as PGAdmin. For more information, see PGAdmin.

Using Flogo Flow State Manager involves the following steps:

Procedure

1. Configure the PostgreSQL database as described in Using Flogo Flow State Manager.
2. Run Flogo Flow State Manager. You can run the app in one of two ways:
    - Run the app as a standalone app. See Running Flogo Flow State Manager as a Standalone App.
    - Run the app in Docker. See Running Flogo Flow State Manager on Docker.
3. Start the Flogo Enterprise Monitoring app by specifying the host and port of the Flogo Flow State Manager. See Starting Flogo Enterprise Monitoring with Details of Flogo Flow State Manager.
4. Start the Flogo app binary. Information about the state of all executed flows of a Flogo app is displayed on the Executions Page.

Configuring the PostgreSQL Database

All execution data from the Flogo app is stored in the PostgreSQL database. Set up the PostgreSQL database for accepting data from the Flogo app as follows:

Procedure

1. Start the PostgreSQL service as docker container. For example:

    ```
    docker run -d --name my_postgres -v my_dbdata1:/var/lib/postgresql/data -p 54320:5432 -e POSTGRES_PASSWORD=<password> -e POSTGRES_USER=<user> postgres
    ```

2. Start the PGAdmin portal as a Docker container:

```
docker run -p 9990:80 -e PGADMIN_DEFAULT_EMAIL=<email address> -e
PGADMIN_DEFAULT_PASSWORD=<pgadmin_password> -d dpage/pgadmin4
```

3. Configure the PostgreSQL server in the PGAdmin admin portal with the following details. Note that you must use the same parameter values while configuring `config.json` for Flogo Flow State Manager.
    - `Host`: IP of the local machine
    - `PORT`: 54320 (same host and port used while starting PostgreSQL service as docker container)
    - `User`: `<user>` (configured while starting PostgreSQL server)
    - `Password`: `<password>` (configured while starting PostgreSQL server)
    - Maintenance database: same as `<user>` (if not specifically mentioned while starting PostgreSQL server)
4. Create the `steps` table by using `<flogo_flow_state_manager.tar>\config\postgres\steps.sql`.

    Note: If you are running the `steps.sql` script in a terminal, convert the script content to a single continuous line.

5. Create the `flowstate` table by using `<flogo_flow_state_manager.tar>\config\postgres\flowstate.sql`.

    Note: If you are running the `flowstate.sql` script in a terminal, convert the script content to a single continuous line.

Running Flogo Flow State Manager as a Standalone App

Procedure

1. Start Flogo Flow State Manager by executing the binary for your operating system:
    - `flowstatemanager-windows_amd64` (Windows executable)
    - `flowstatemanager-linux_amd64` (Linux executable)
    - `flowstatemanager-darwin_amd64` (Mac executable)

2. Copy the `<flogo_flow_state_manager.tar>\config\postgres\config.json` into the `bin` directory. If the `config.json` file exists in any other directory, you can also set the `FLOW_STATE_CONFIG` environment variable to point to the location as follows:

```
FLOW_STATE_CONFIG=<file path>
```

3. Update the values in `config.json` as follows:

> ⚠ **Caution:** Code snippets in the PDF could have undesired line breaks due to space constraints and should be verified before directly copying and running it in your program.

```
{
"exposeRecorder": true,
"port": "<The port on which you want to start the flow state manager binary>",
"persistence": {
"type":"postgres",
"name": "pg-server-1",
"description": "",
"host": "<The IP address where Postgres is running>",
"port": "<port on which the Postgres database server is running>",
"databaseName": "postgres",
"user": "<user value configured while starting PostgreSQL server)>",
"password": "<password value configured while starting PostgreSQL server>",
"Maintenance database": <same as <user>, if not specifically mentioned while starting postgreSQL>
"maxopenconnection": "0",
"maxidleconnection": "2",
"connmaxlifetime": "0",
"maxconnectattempts": "3",
"connectionretrydelay": "5",
"tlsparam": "VerifyCA",
"cacert": "",
```

```
"clientcert": "",
"clientkey": ""
}
}
```

Running Flogo Flow State Manager on Docker

To run the Flogo Flow State Manager in a Docker container:

Procedure

1. Update `<flogo_flow_state_manager.tar>\config\postgres\config.json` as per your Postgres installation.

   Important: Postgres is not accessible over '`localhost`' when Flogo Flow State Manager is running on Docker. You must use the machine's IP address.

2. Go to the root folder (packaging) and run:

   ```
   docker build -t flogostatemanager:1.0.0 -f ./deployments/Dockerfile
   .
   ```

3. Start the Flogo Flow State Manager service by mounting a volume for `config.json`:

   ```
   docker run -p 8099:8099 -v <parent
   absolutepath>/flowstatemanager/packaging/config/postgres/config.jso
   n:/opt/flogo/sm/config.json flogostatemanager:1.0.0
   ```

Running Flogo Flow State Manager on Kubernetes

Procedure

1. Update `<flogo_flow_state_manager.tar>\config\postgres\config.json` as per your Postgres installation.

> **Important:** Postgres is not accessible over 'localhost' when Flogo Flow State Manager is running on Docker. You must use the machine's IP address.

2. Go to the root folder (packaging) and run:

```
docker build -t flogostatemanager:1.0.0 -f ./deployments/Dockerfile
.
```

3. Push the Flogo Flow State Manager Docker image to the Docker registry.
4. Update the <flogo_flow_state_manager.tar>/deployments/k8s/deployment.ymlas per the required configuration. For example, image name, version, port values, and so on.
5. Deploy the Flogo Flow State Manager service in the Kubernetes cluster:

```
<flogo_flow_state_manager.tar>/deployments/k8s/deploy.sh
```

   This command creates the required configmap and applies the deployment.yml configuration to define the deployment and service component for Kubernetes.

6. To undeploy the Flow State Manager service in k8s cluster:

```
<flogo_flow_state_manager.tar>/deployments/k8s/undeploy.sh
```

Configuring Flogo Flow State Manager

Property	Description
FLOGO_FLOW_SM_ ENDPOINT	The endpoint of Flogo Flow State Manager. The format to set the property is: FLOGO_FLOW_SM_ENDPOINT=http://<host>:<port> For example: FLOGO_FLOW_SM_ENDPOINT=http://localhost:9091

Property	Description
	Note: This property needs to be set when starting the app binary after Flogo Flow State Manager is up and running. It also needs to be set when running the Flogo Enterprise Monitoring app.
FLOGO_FLOW_STATE_ASYNC_INVOCATION	Specifies whether the Flogo Flow State Manager must be invoked asynchronously or not. Enabling the property also helps to increase the throughput of the app. The format to set the property is: `FLOGO_FLOW_STATE_ASYNC_INVOCATION=true` **Note:** This property needs to be set when starting the app binary after Flogo Flow State Manager is up and running. **Default value:** false

Starting Flogo Enterprise Monitoring with Details of Flogo Flow State Manager

Start Flogo Enterprise Monitoring with the host and port details of the Flogo Flow State Manager:

Procedure

Start the TIBCO Flogo® Enterprise Monitoring app. When starting the app, use the `FLOGO_FLOW_SM_ENDPOINT` environment variable to specify the host and port of the Flogo Flow State Manager. For example:

```
docker run -it -e FLOGO_MON_DATA_DIR=/opt/flogomon/data -e FLOGO_
FLOW_SM_ENDPOINT=http://<host>:<port> -v ~/temp:/opt/flogomon/data -p
7337:7337 <fe-mon docker image name>
```

Procedure

Check the console log to verify that a successful connection has been established

with Flogo Flow State Manager.
If you notice a connection error in the log, verify whether the Flogo Flow State Manager is running and the host/port details are configured correctly.

Starting the App Binary

Start the app binary after Flogo Flow State Manager is up and running.

```
export FLOGO_FLOW_SM_ENDPOINT=http://localhost:9091
FLOGO_HTTP_SERVICE_PORT=7777
FLOGO_APP_MON_SERVICE_CONFIG="
{\"host\":\"<IP address>\",\"port\":\"<port>\"}"
./app-binary
```

Result

Information about the state of all executed flows of a Flogo app is displayed on the Executions Page.

Viewing Statistics by Using Flogo Enterprise Monitoring app

Before you begin

- The Flogo Enterprise Monitoring app must be running. See Running the Flogo Enterprise Monitoring App or Running the Flogo Enterprise Monitoring App on Docker.
- The Flogo app to be monitored must be registered with the Flogo Enterprise Monitoring app. See Registering a Flogo App with the Flogo Enterprise Monitoring App.

Procedure

1. In the UI of the Flogo Enterprise Monitoring app, go to the following URL to monitor the app:

   `http://<URL of Flogo Enterprise Monitoring app>:<port of Flogo Enterprise`

```
Monitoring app>
```

For example:

```
http://localhost:7337
```

The **Apps** page is displayed as shown below:

The **Apps** page shows all the Flogo apps registered with the Flogo Enterprise Monitoring app. For details, see Apps Page.

2. Click an app name.

   The **Metrics** page is displayed. The instrumentation statistics are displayed in two tabs - Flow and Triggers. For details, see Metrics Page.

Apps Page

The **Apps** page shows all the Flogo apps registered with the Flogo Enterprise Monitoring app.

> **Note:** The apps list on this page is not refreshed automatically. Click ↻ to refresh the list manually.

For each running app, you can view the following details:

Item	Description
Name	Name of the app. Click the name of an app to get more details of the app. For example, in the above screenshot, you can click **HTTPService1** to get more details about the service. The details of **HTTPService1** are displayed on the **Metrics** page.
Version	Version of the app.
Flogo Version	The version in which the app was created.
Instances	Number of instances registered per app.
Tags	Tags of the app. These tags help you provide additional information about the app. For example, you can specify whether it is a REST app or whether it is running in Kubernetes, and so on.

From the **Apps** page, you can also:

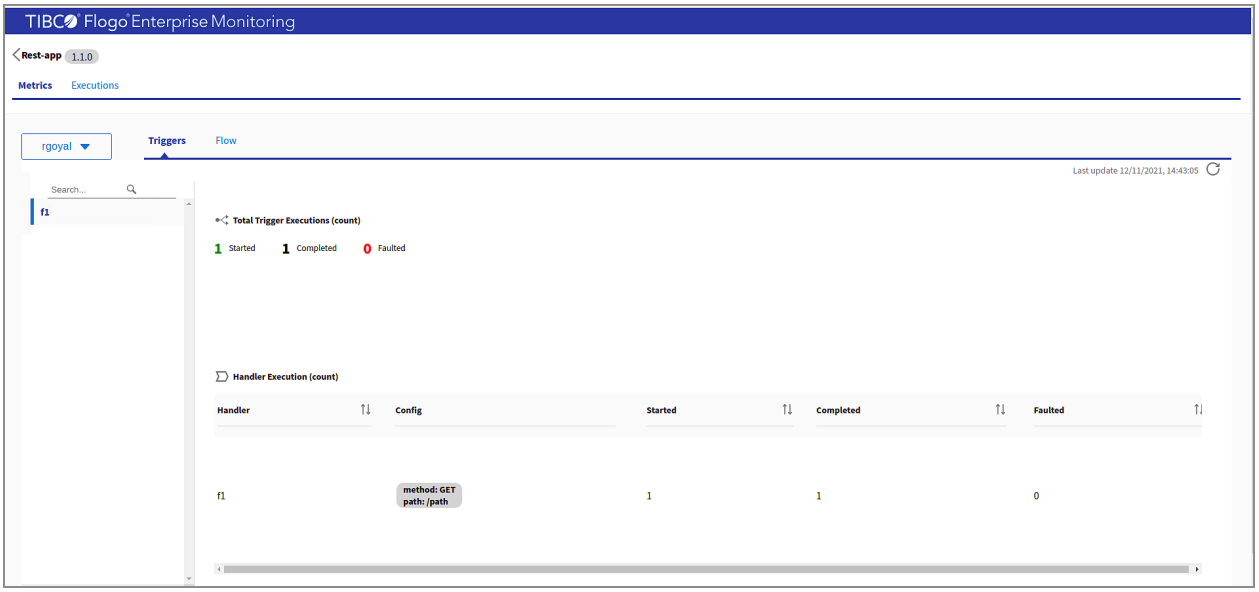
- Click the heading in the list to sort the apps. For example, to sort the list by name, click the heading **Name**. Click the same heading again to toggle between the ascending or descending order of listing the apps.
- The Search control above the list enables you to find apps by name.

Metrics Page

The Metrics page displays the instrumentation statistics of flows and triggers.

Select an instance ID from the instance ID list in the upper-left corner of the page. The instrumentation statistics of flows and triggers are displayed on the **Triggers** tab and **Flow** tab.

Triggers Tab



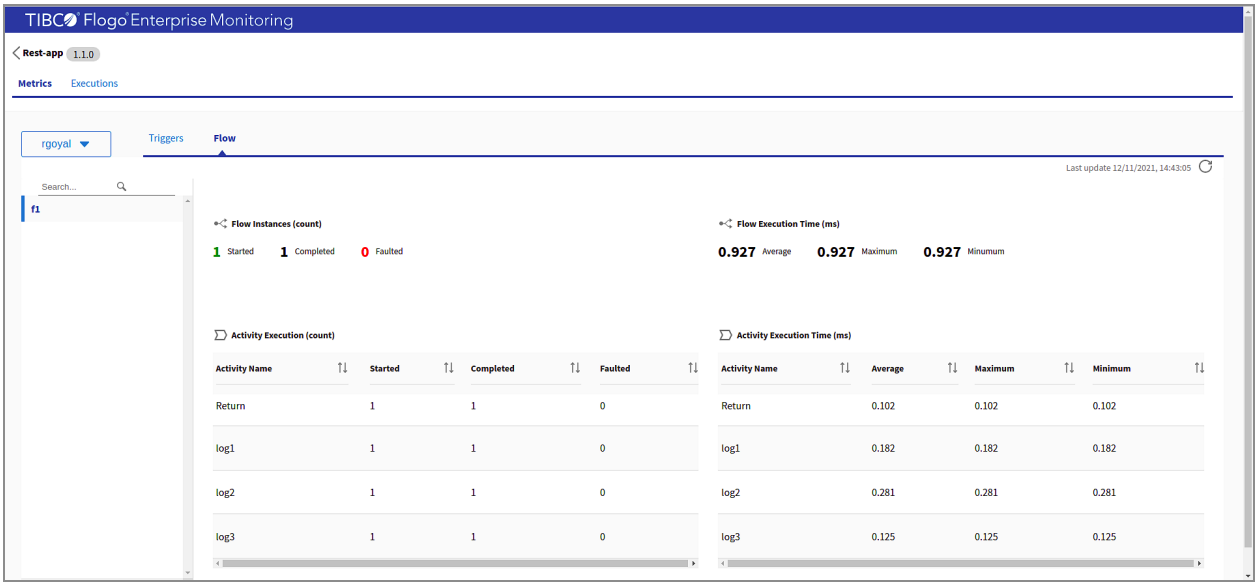
Select a trigger from the list on the left to see its details. You can also search for a trigger in the list.

The following information is displayed on the right:

Name	Description
Total Trigger Executions (count)	
Started	Total number of trigger instances started
Completed	Total number of trigger instances completed
Faulted	Total number of trigger instances failed
Handler Execution (count)	
Handler	Name of the trigger handler
Config	Configuration of the trigger handler. For example:

Name	Description
	`method: POST` `path: /arrayfilter`
Started	Total number of trigger handlers started
Completed	Total number of trigger handlers completed
Faulted	Total number of trigger handlers failed

Flow Tab



Select a flow from the list on the left to see its details. You can also search for a flow in the list.

The following information is displayed in the work area on the right:

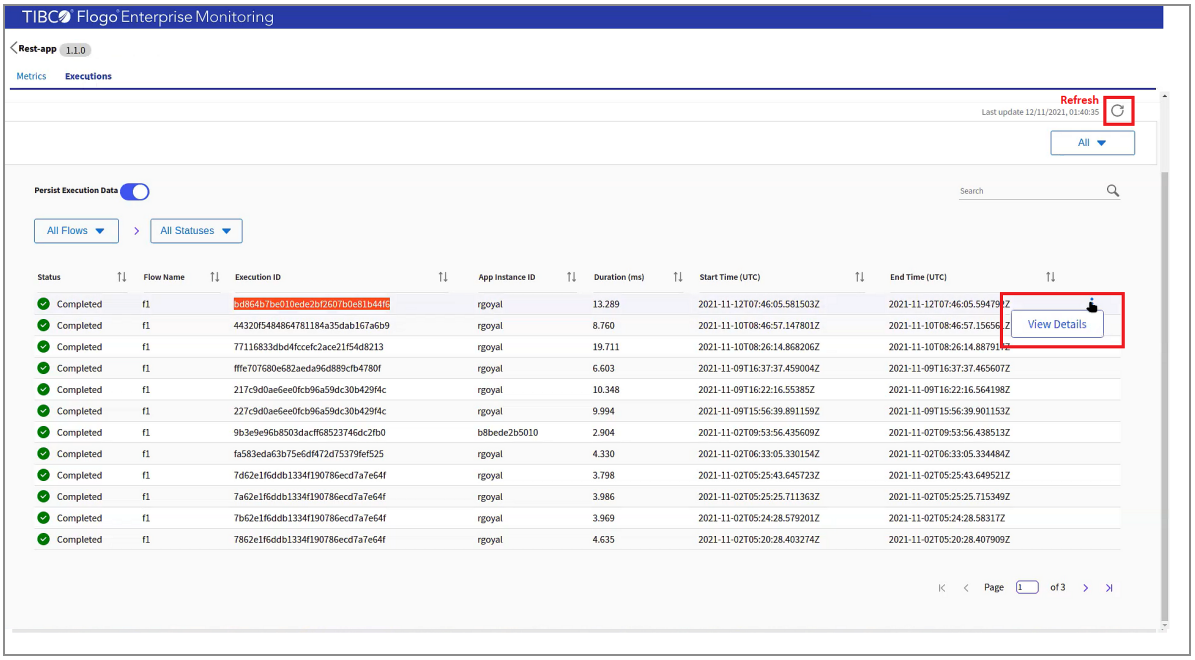
Name	Description
Flow Instances (count)	

Name	Description
Started	Total number of flow instances started
Completed	Total number of flow instances completed
Faulted	Total number of flow instances failed
Flow Execution Time (in milliseconds)	
Note: The **Flow Execution Time (ms)** for a faulted flow is always displayed as 0, even if the activities within the flow took time to execute.	
Average	Average execution time of the flow for successful executions
Maximum	Maximum execution time for the flow
Minimum	Minimum execution time for the flow
Activity Execution (count)	
Note: If an Activity is rerun, **Activity Execution (count)** also includes the rerun counts. You can find out whether an Activity has been rerun through the difference in the trigger and flow metric counts.	
Activity Name	Name of Activity
Started	Total number of times a given Activity has started
Completed	Total number of times a given Activity has been completed
Faulted	Total number of times a given Activity has failed
Activity Execution Time (in milliseconds)	

Name	Description
Activity Name	Name of Activity
Average	Average execution time of a given Activity for successful executions
Maximum	Maximum execution time for a given Activity
Minimum	Minimum execution time for a given Activity

Executions Page

The **Executions** page displays information about the state of all run flows of a Flogo app. Details of a trigger are not captured.



From this page, you can:

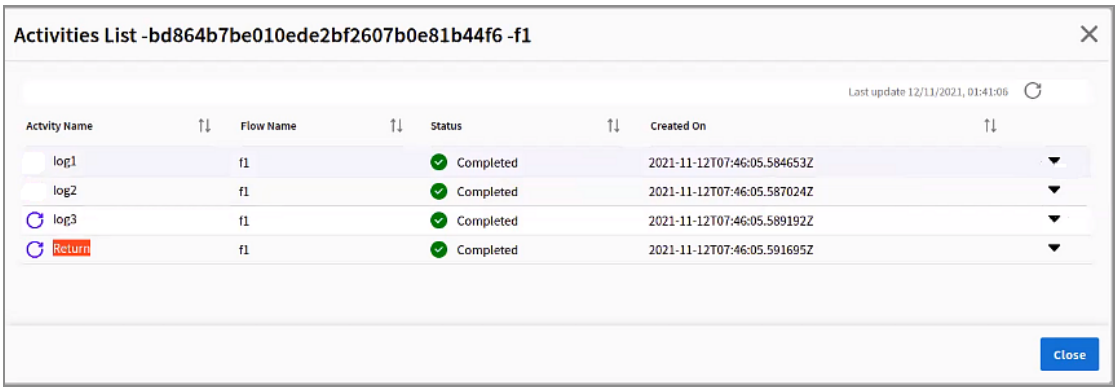
- **Persist execution data**: Select **Persist Execution Data** to persist execution data to the supported database (currently, PostgreSQL).

> **Note:** If **Persist execution data** is disabled, any new execution data is not saved to the database. The **Rerun flow from this Activity** feature is also disabled for all flow executions.

- **Filter based on app version**: You can use the filer to choose the app version for which the data must be displayed.
- **Filter based on time frame**: Use the **All** drop-down to filter based on time frame. For example: in the last 1 hour, last week, last 30 days, and so on.
- **Filter based on flow**: If you have multiple flows, use the **All Flows** drop-down to filter based on flows.
- **Filter based on status**: Use the **All Statuses** drop-down to filter based on the status of the flow.
- **Refresh data:** Use ↻ to refresh the data in the table.
- **View execution data**: The following data is displayed in a tabular format.

Name	Description
Status	Status
Flow Name	Name of the flow.
Execution ID	Instance ID of the flow.
App Instance ID	Instance ID of the app.
Duration (ms)	Duration for which the flow was running.
Start Time (UTC)	Time when the flow was started, based on Coordinated Universal Time (UTC).
End Time (UTC)	Time when the flow ended, based on Coordinated Universal Time (UTC).

- **View details of a flow**: For each flow, you can view its details by clicking **View Details**. A list of activities executed is displayed:



- **Rerun the flow from a specific Activity**: You can rerun the flow from a specific Activity. You cannot modify the input data; you can only rerun the Activity.

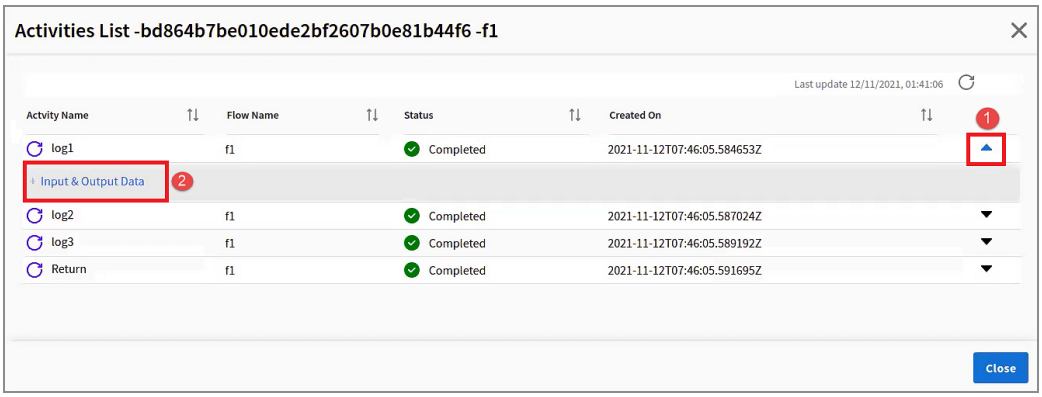
  Note: If you rerun an Activity, the previous execution record for the Activity is overwritten in the database. Past execution records of the Activity that was rerun and all subsequent activities in the flow are deleted.

  Important: Exercise caution while re-running a flow attached to the App Startup Trigger and App Shutdown Trigger. These triggers, typically, include logic for creating data or cleaning up data. Such flows might impact the running instances of the app.

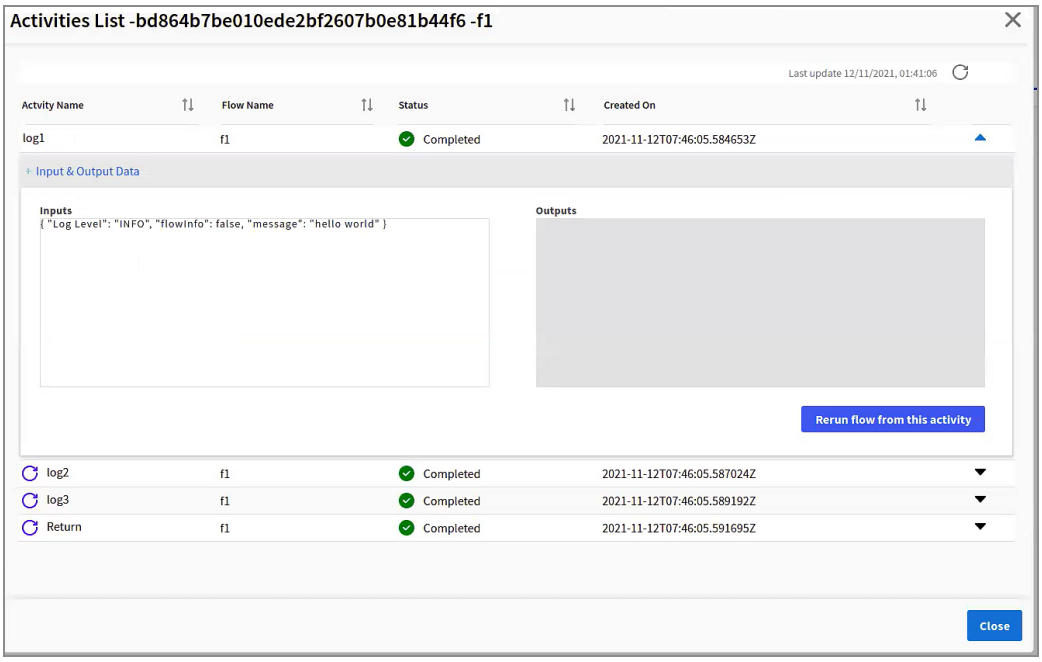
  To rerun the flow from a specific Activity:

  ### Procedure

  1. On the **View Details** page, click the expand icon and then click **Input & Output Data**.



The input and output for the selected Activity are displayed.



2. Click **Rerun flow from this Activity**.

Note:

- **Rerun flow from this Activity** disappears for activities that are a part of subflows. **Rerun flow from this Activity** also disappears if **Persist Execution Data** is disabled.
- If the version of the app running instance is not same as that of the selected version, you cannot rerun the activities.

After the rerun of the activity, the rerun is indicated by . The **Executions** page is also updated with the latest data. Click to refresh the changes on the **Executions** page.

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

Method	Description	Status Code
		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 an environment variable:

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 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
failed	Total number of times a given flow has failed
avg_exec_time	Average execution time of a given flow for successful 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

Name	Description
failed	Total number of times a given Activity has failed
avg_exec_time	Average execution time of a given Activity for successful executions
min_exec_time	Minimum execution time for a given Activity
max_exec_time	Maximum execution time for a given Activity

Prometheus

Flogo apps support integration with Prometheus for app metrics monitoring. Prometheus is a monitoring tool that 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 analytics.

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
FlowName	Name of flow
State	State of the flow. One of the following states: • Started • Completed • Failed
flogo_flow_execution_duration_msec: Total time (in ms) taken by the flow for successful	

Labels	Description
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

Labels	Description
FlowName	Name of flow
ActivityName	Name of Activity
State	State of the Activity. One of the following states: • Completed • Failed

Note: Deprecated in Flogo Enterprise 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

Note: Deprecated in Flogo Enterprise 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

Labels	Description
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
```

Example:

```
http://192.0.2.0:7779/metrics
```

Adding Custom Labels to Prometheus Metrics

The `FLOGO_APP_METRICS_PROMETHEUS_LABEL` environment variable appends custom key-value labels to all Prometheus metrics emitted by a Flogo application. This helps with identification, filtering, and analysis of metrics in Prometheus.

To add a custom label, set the `FLOGO_APP_METRICS_PROMETHEUS_LABEL` environment variable with a key-value pair.

Syntax:

```
FLOGO_APP_METRICS_PROMETHEUS_LABEL="<key>=<value>"
```

Example:

```
FLOGO_APP_METRICS_PROMETHEUS_LABEL="environment=production"
```

Usage Example:

```
FLOGO_HTTP_SERVICE_PORT=7779 FLOGO_APP_METRICS_PROMETHEUS=true
FLOGO_APP_METRICS_PROMETHEUS_LABEL="version=v1.2.3" ./<app-executable>
```

Often-Used Queries

Prometheus uses the PromQL query language. This section lists some of the most 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 (AppName, FlowName)`
Total number of flows that failed per app	`count(flogo_flow_execution_count {State="Failed"}) by (AppName, 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	`count(flogo_flow_execution_count`

To Get this Metric	Use this Query
apps (when you are collecting metrics for multiple apps)	`{State="Failed"})`
Total time taken by flows which got executed successfully	`sum(flogo_flow_execution_duration_msec {State="Completed"}) by (AppName, FlowName)`
Total time taken by flows which failed	`sum(flogo_flow_execution_duration_msec {State="Failed"}) by (AppName, 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 (AppName)`
Minimum time taken by flows which failed	`min(flogo_flow_execution_duration_msec {State="Failed"}) by (AppName)`
Maximum time taken by flows which executed successfully	`min(flogo_flow_execution_duration_msec {State="Completed"}) by (AppName)`
Maximum time taken by flows which failed	`max(flogo_flow_execution_duration_msec {State="Failed"}) by (AppName)`
Average time taken by flows which executed successfully	`avg(flogo_flow_execution_duration_msec {State="Completed"}) by (AppName, FlowName)`
Average time taken by flows which failed	`avg(flogo_flow_execution_duration_msec {State="Failed"}) by (AppName, 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 (AppName, FlowName,ActivityName)`
Total number of activities that failed per flow and app	`count(flogo_Activity_execution_count {State="Failed"}) by (AppName, 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 (AppName, FlowName,ActivityName)`
Individual time taken by activities which failed per app and flow	`sum(flogo_Activity_execution_duration_msec {State="Failed"}) by (AppName, 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 (AppName, FlowName,ActivityName)`
Minimum time taken by a failed Activity within a given flow and app	`min(flogo_Activity_execution_duration_msec {State="Failed"}) by (AppName, 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 (AppName, FlowName,ActivityName)`

To Get this Metric	Use this Query
Maximum time taken by an Activity which failed within a given flow and app	`max(flogo_Activity_execution_duration_msec {State="Failed"}) by (AppName, 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 (AppName, 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 (AppName, FlowName,ActivityName)`

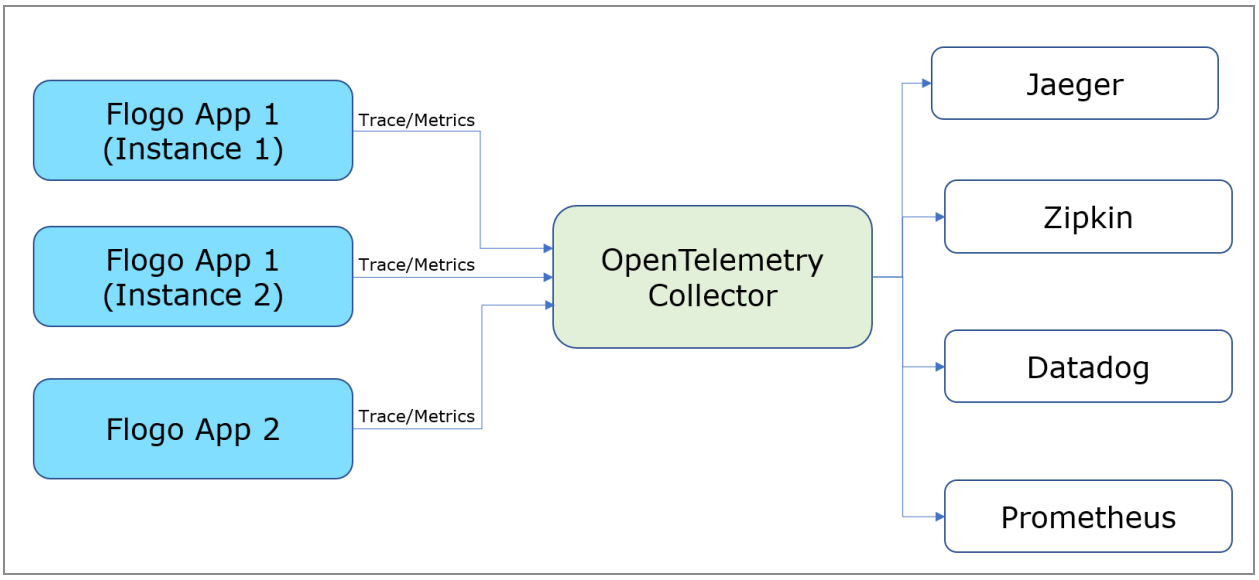
OpenTelemetry Collector

Flogo supports integration with OpenTelemetry (OT) Collector. The goal of this collector is to create standard software development kit for tracing, metrics and logging which different vendors like Jaeger, Zipkin, Datadog, Prometheus adopt. You have the flexibility to switch vendors without changing the application logic with OpenTelemetry.

Note: To use this feature for TIBCO Cloud Integration deployments, ensure that the OpenTelemetry Collector is reachable from app containers.

Architecture

This is the schematic view of how the OT collector works:



Note: You can use the same architecture for Distributed tracing as well. For more information, see Tracing Apps by Using OpenTelemetry Collector.

Configuration

The parameters listed below are required for the configuration of the OT collector:

Name	Required	Default	Description
FLOGO_ OTEL_ METRICS	Yes	False	Enable OpenTelemetry metrics for Flogo app
FLOGO_ OTEL_ METRICS_ ATTRIBUTES	No	None	Add one or more custom attributes to the metrics. For example, `FLOGO_OTEL_METRICS_ ATTRIBUTES="deployment_type=flogo"`
FLOGO_ OTEL_OTLP_ ENDPOINT	Yes	None	OpenTelemetry protocol (OTLP) receiver endpoint configured for OpenTelemetry Collector. For gRPC protocol, set `<host>:<otlp_grpc_port>` For http protocol, set `https://<host>:<otlp_http_ port>`

Name	Required	Default	Description
FLOGO_ OTEL_OTLP_ HEADERS	No	None	Set one or more custom gRPC/HTTP headers in the request to the collector. For example, `FLOGO_OTEL_OTLP_ HEADERS="Authorization=Bearer <token>,API_ KEY=<api_key_value>"`.
FLOGO_ OTEL_TLS_ SERVER_ CERT	No	None	Set PEM encoded Server/CA certificate when TLS is enabled for OTLP receiver. You can configure a path to the certificate or use base64 encoded certificate value. A file path must be prefixed with `"file://"`. e.g. `FLOGO_OTEL_TLS_SERVER_ CERT="file:///Users/opentelemetry/certs/cert. pem"` or `FLOGO_OTEL_TLS_SERVER_CERT=<base64_ encoded_server_certificate>.` When this certificate is not set, unsecure connection is established with the collector.

Monitor Flogo apps metrics using OpenTelemetry

You can see the number of flows and activities executed in the app as per the below metrics:

Metrics	Label	Description
flogo_activity_executions_ total	-	Total number of times the activity is started, completed, or failed.
	app_name	Name of application
	app_version	Version of application
	flow_name	Name of flow
	activity_name	Name of activity
	state	State of activity - Started, Completed or Failed

Metrics	Label	Description
	host_name	Name of the host or app instance ID
flogo_flow_executions_total	-	Total number of times the flow is started, completed or failed
	app_name	Name of application
	app_version	Version of application
	flow_name	Name of flow
	state	State of flow - Started, Completed or Failed
	host_name	Name of the host or app instance ID

Example

Prometheus

```
FLOGO_OTEL_METRICS=true FLOGO_OTEL_OTLP_ENDPOINT="localhost:4317" FLOGO_OTEL_
METRICS_ATTRIBUTES="deployment=local,product=flogo" ./<app-executable>
```

Distributed Tracing

Distributed tracing allows you to log information about an app's behavior during its execution. It shows the path an app takes from start to finish. You can then use the information to troubleshoot performance bottlenecks, errors, and debugging failures in the app execution.

As the app travels through different services, each segment is recorded as a *span*. A span is a building block of a trace and represents work done with time intervals and associated metadata. All the spans of an app are combined into a single *trace* to give you a picture of an entire request. A trace represents an end-to-end execution; made up of single or multiple spans. A *Tracer* is the actual implementation that records the spans and publishes them.

Distributed 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 distributed tracing is particularly useful in a distributed microservice architecture environment where each app is instrumented by a 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 distributed tracing capability in Flogo Enterprise:

- At any given point in 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. Traces of the triggers used in the app are not shown.
- Tracing can be done across apps bypassing the tracing context from one app to another. To trace across multiple apps, you must make sure that all apps are instrumented with similar tracing frameworks, such as Jaeger semantics so that they understand the framework language. Otherwise, you can't 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. This only works for triggers that support the extraction of the context from the underlying technology, for instance, triggers those support HTTP headers.

  The **ReceiveHTTPMessage** REST trigger and **InvokeRESTService** Activity are supported for this release where the REST trigger can extract the context from the request and **InvokeRESTService** Activity can inject the context into the request. If two Flogo apps are both Jaeger-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 children 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.

Tracing Apps Using Jaeger

Flogo apps provide an implementation of the OpenTracing framework using the Jaeger backend. 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, the 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 a Flogo app, 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.

Prerequisites: The following prerequisites must be met before using the tracing capability in Flogo Enterprise:

- 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 the Flogo app binary.
- If your Jaeger server is running on a machine other than the machine on which your app resides, be sure to set the `JAEGER_ENDPOINT=http://<JAEGER_HOST>:<HTTP_TRACE_COLLECTOR_PORT>/api/traces` environment variables. Refer to the Environment Variables page for the environment variables that you can set.

Flogo Enterprise-Related Tags in Jaeger

In OpenTracing, each trace and span have their tags. Tags are useful for filtering traces, for example, if you want to search for a specific trace or time interval.

> **Note:** Adding your custom tags for any one span (Activity) only is currently not supported. Any custom tags that you create are added to **all** spans and traces.

Flogo Enterprise 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 is 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 constructs 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 Flogo Enterprise 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 are added to **all** spans and traces.

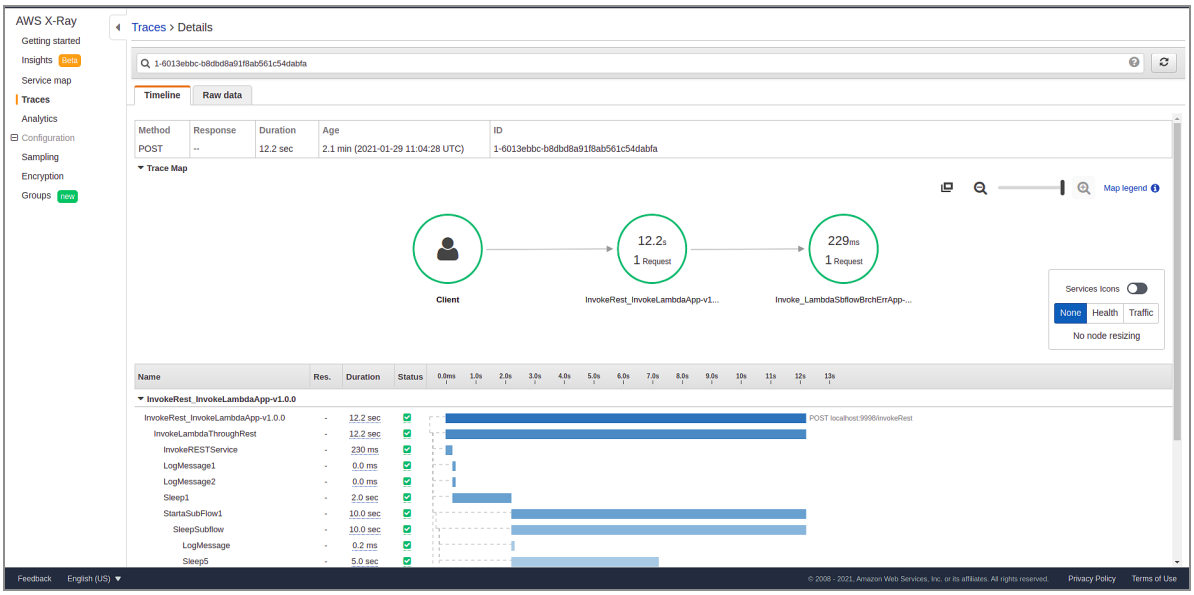
Refer to the Environment Variables page for the environment variables that you can set.

Tracing Apps by Using AWS X-Ray

If you are running your Flogo app on the cloud or in your local environment, you can track your app performance or troubleshoot issues by using AWS X-Ray. For more information about AWS X-Ray, refer to AWS X-Ray.

When you use AWS X-Ray for tracing, your app sends trace data to AWS X-Ray. X-Ray processes the data to generate a service map and searchable trace summaries. For each flow, subflow, and Activity, details such as execution time are displayed on the AWS X-Ray dashboard.

The following example shows the trace details of the `InvokeRest_InvokeLambdaApp-v1.0.0` app. It includes details such as activities that were invoked, and their execution time and status.



Before you begin

Make sure that you meet the following requirements:

- **Knowledge of AWS X-Ray:** For more information, refer to AWS XRay.
- **For an app containing a non-Lambda trigger:**
  - **AWS X-Ray daemon:** You must have an AWS X-Ray daemon running on your machine to send trace data to the AWS X-Ray service. Alternatively, your app must have access to another machine where the daemon is running. Download the AWS X-Ray daemon from the AWS website and run the AWS X-Ray daemon.
  - **Environment variable:** If the AWS X-Ray daemon and app are running on two different machines, set the environment variable *AWS_XRAY_DAEMON_ADDRESS*to the IP address where the AWS X-Ray daemon is running for receiving traces. You need not set this variable if the daemon and app are running on the same machine.
- **For an app containing a Lambda trigger:**
  - To trace the app end-to-end, TIBCO recommends that you enable the **Active Tracing** option in AWS along with the Flogo tracing feature. **Active Tracing** provides all the details of the app while the Flogo tracing feature provides details specific to the Flogo app. For example, details such as how long it took to initialize the container, are provided by **Active Tracing**. Details specific to the Flogo implementation (such as the flows, sub-flows, or activities executed)

are provided by the Flogo tracing feature.

- For an app containing a Lambda trigger, you need not run the AWS X-Ray daemon. This is because AWS X-Ray is integrated with AWS Lambda.
- Add the following permissions to the execution role. For more information on how to add the permissions, refer to the AWS Documentation.
  - `xray:PutTraceSegments`
  - `xray:PutTelemetryRecords`

> **Note:** The AWS API Gateway Lambda and S3 Bucket Event Lambda triggers are not supported.

Enabling Tracing Using AWS X-Ray

To enable tracing using AWS X-Ray, set the *FLOGO_AWS_XRAY_ENABLE* environment variable to true. The default is false.

Search Using Annotations

You can search based on predefined Flogo annotations. The following annotation is available in this release:

`flogo_flow_name`: Name of the flow

Here is an example of using annotations to search:

```
annotation.flogo_flow_name="sampleFlow"
```

Metadata

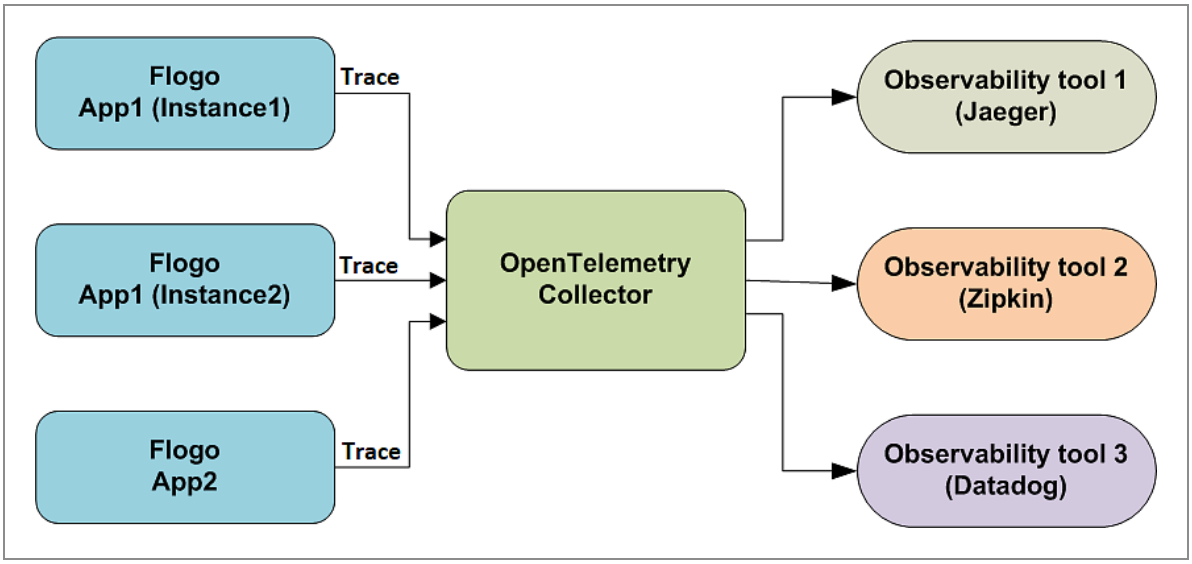
The following metadata about an app is stored in the `flogo` namespace:

- `flow_name`: Name of the flow
- `Activity_name`: Name of the Activity

This metadata can be used when debugging. You can use the metadata to identify the exact errors, stack traces, flow name, Activity name, and so on. Note that the metadata cannot be used for searching traces.

Tracing Apps by Using OpenTelemetry Collector

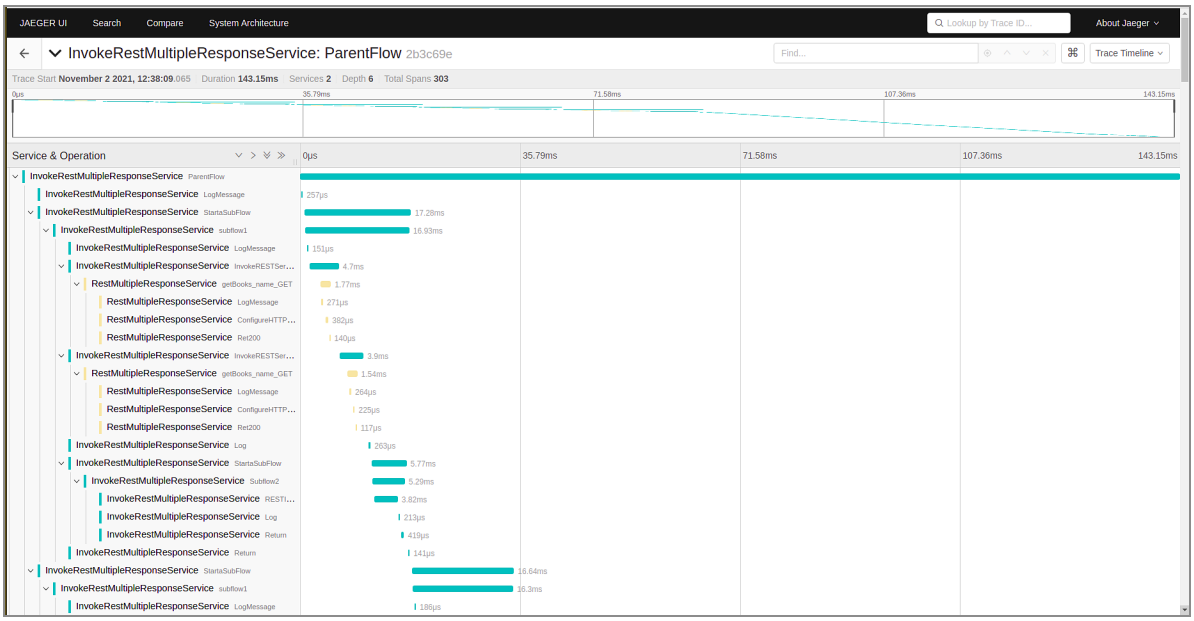
By using OpenTelemetry Collector, you can capture traces from your Flogo app and send them to observability vendor tools such as Jaeger, Zipkin, and Datadog. This gives you the flexibility to switch between observability vendor tools without changing the logic of your app. For more information about OpenTelemetry Collector, see OpenTelemetry documentation.



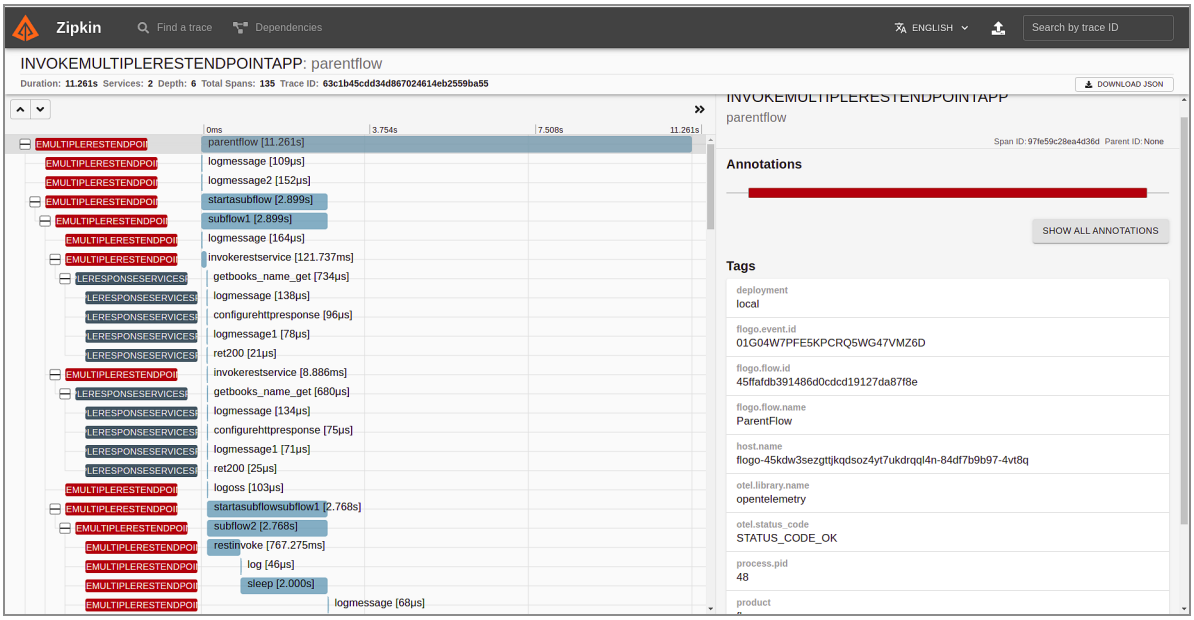
When you use this feature, traces of the Flogo app are sent to the OpenTelemetry Collector. OpenTelemetry Collector has vendor-specific configurations that allow you to send these traces to supported observability vendor tools. For example, you can specify Zipkin-specific configurations in the `otel-zipkin-collector-config.yaml` configuration file for the traces to be displayed on the Zipkin dashboard.

The following screenshots show traces from one Flogo app on two different observability vendor tools, Jaeger and Zipkin.

Jaeger output of a Flogo app



Zipkin output of a Flogo app



Enabling Tracing for OpenTelemetry Collector

Before you begin

Make sure that you meet the following requirements:

- Ensure that you can connect to the OpenTelemetry Collector.

> **Note:**
> - To use this feature for TIBCO Cloud Integration deployments, ensure that the OpenTelemetry Collector is reachable from app containers.
> - If the connection to OpenTelemetry Collector is lost, traces during that time duration are not collected.

- Install an observability vendor tool of your choice: Jaeger, Zipkin, Datadog, and so on.

Mandatory Configuration Parameters

To enable tracing by using OpenTelemetry Collector, set the following mandatory parameters:

Name	Default	Description
`FLOGO_OTEL_TRACE`	False	Enables tracing by using OpenTelemetry Collector.
`FLOGO_OTEL_OTLP_ENDPOINT`	None	Specifies the OpenTelemetry protocol (OTLP) receiver endpoint for OpenTelemetry Collector. Supported protocols are: • **gRPC:** Set to `<host>:<otlp_grpc_port>`. • **HTTP:** Set to `https://<host>:<otlp_http_port>`.

Optional Configuration Parameters

You can also use some optional configuration parameters when tracing apps using OpenTelemetry Collector. Here are some commonly used parameters and their descriptions:

Name	Default	Description
FLOGO_OTEL_TRACE_ATTRIBUTES	None	Add one or more custom attributes to the trace. The format is key-value pairs separated by commas. For example, to filter based on the deployment type and deployment cluster, you can use: `FLOGO_OTEL_TRACE_ATTRIBUTES="deployment.type=staging,deployment.cluster=staging3"`
FLOGO_OTEL_OTLP_HEADERS	None	Set one or more custom gRPC or HTTP headers in the request to the OpenTelemetry Collector. The format is key-value pairs separated by commas. For example: `FLOGO_OTEL_OTLP_HEADERS="Authorization=Bearer <token>,API_KEY=<api_key_value>"`
FLOGO_OTEL_TLS_SERVER_CERT	None	If TLS is enabled for OpenTelemetry protocol receiver, set PEM-encoded server or CA. You can configure a path to the certificate or use base64-encoded certificate value. A file path must be prefixed with `"file://"`. For example: • `FLOGO_OTEL_TLS_SERVER_CERT="file:///Users/opentelemetry/certs/cert.pem"` • `FLOGO_OTEL_TLS_SERVER_CERT=<base64_encoded_server_certificate>` You can also encrypt base64 encoded certificate value by using either TIBCO Cloud Integration platform API or by using app executable and set it to the environment variable with prefix `"SECRET:"` For example: • `FLOGO_OTEL_TLS_SERVER_CERT=SECRET:<encrypted_base64_encoded_cert_value>` For details about encryption, see Encryption using App executable or Encryption using TIBCO Cloud Platform API. When this certificate is not set, an unsecure connection is established with OpenTelemetry Collector.

Tracing With OpenTelemetry Collector

Using OpenTelemetry Collector with Jaeger

The Jaeger Docker image includes OpenTelemetry Collector. So, you need not run OpenTelemetry Collector separately.

```
docker run --name jaeger -p 13133:13133 -p 16686:16686 -p 4317:55680 -d
--restart=unless-stopped jaegertracing/opentelemetry-all-in-one
```

For example:

```
FLOGO_OTEL_TRACE=true FLOGO_OTEL_OTLP_ENDPOINT="localhost:4317" FLOGO_
OTEL_TRACE_ATTRIBUTES="deployment=local,product=flogo" ./TimerOTel-
darwin_amd64
```

Using OpenTelemetry Collector with Zipkin

Procedure

1. Start Zipkin as follows:

   ```
   docker run -it --rm -p 9411:9411 -d --name zipkin openzipkin/zipkin
   ```

2. Update the OpenTelemetry Collector configuration file for Zipkin, `otel-zipkin-collector-config.yaml`, as follows:

   ```
   receivers:
   otlp:
   protocols:
   http:

   exporters:
   zipkin:
   # Change IP
   endpoint: "http://xxx.xxx.x.x:xxxx/api/v2/spans"
   format: proto

   processors:
   batch:
   ```

```
service:
pipelines:
traces:
receivers: [otlp]
processors: [batch]
exporters: [zipkin]
```

3. Start OpenTelemetry Collector with Zipkin Exporter as follows:

```
docker run -d --rm -p 4318:4318 -v "${PWD}/otel-zipkin-collector-
config.yaml":/otel-collector-config.yaml --name otelcol
otel/opentelemetry-collector:0.35.0 --config otel-collector-
config.yaml
```

For example:

```
FLOGO_OTEL_TRACE=true FLOGO_OTEL_OTLP_ENDPOINT="https://localhost:4318"
FLOGO_OTEL_TRACE_ATTRIBUTES="deployment=local,product=flogo"
./TimerOTel-darwin_amd64
```

Using OpenTelemetry Collector with Zipkin (with TLS)

Procedure

1. Update `server-cert-gen.sh` as follows:

```
openssl req -newkey rsa:2048 \
-new -nodes -x509 \
-days 3650 \
-out cert.pem \
-keyout key.pem \
-extensions san \
-config <(echo '[req]'; echo 'distinguished_name=req';
echo '[san]'; echo 'subjectAltName=DNS:localhost,DNS:127.0.0.1') \
-subj
"/C=US/ST=California/L=Sunnyvale/O=TIBCO/OU=Flogo/CN=localhost"
```

2. Start Zipkin.

```
docker run -it --rm -p 9411:9411 -d --name zipkin openzipkin/zipkin
```

3. Update the OpenTelemetry Collector configuration file for Zipkin, `otel-zipkin-collector-config.yaml`, as follows:

```
receivers:
otlp:
protocols:
grpc:
tls_settings:
cert_file: /var/certs/cert.pem
key_file: /var/certs/key.pem

exporters:
zipkin:
endpoint: "http://xxx.xxx.x.x:xxxx/api/v2/spans"
format: proto

processors:
batch:

service:
pipelines:
traces:
receivers: [otlp]
processors: [batch]
exporters: [zipkin]
```

4. Create a `certs` directory under the current directory and copy `cert.pem` and `key.pem` in the `certs` directory.
5. Start OpenTelemetry Collector with Zipkin Exporter as follows:

```
docker run -d --rm -p 4317:4317 -v "${PWD}/otel-zipkin-collector-
config.yaml":/otel-collector-config.yaml -v
"${PWD}/certs":/var/certs --name otelcol otel/opentelemetry-
collector:0.35.0 --config otel-collector-config.yaml
```

For example:

```
FLOGO_OTEL_TRACE=true FLOGO_OTEL_OTLP_ENDPOINT="localhost:4317"
FLOGO_OTEL_TLS_SERVER_
CERT="file:///Users/dev/Installations/OpenTelemetry/zipkin/certs/cert.pe
m"
FLOGO_OTEL_TRACE_ATTRIBUTES="deployment=local,product=flogo"
./TimerOTel-darwin_amd64
```

Flogo Related Attributes in OpenTelemetry Collector

In OpenTelemetry, each trace has its own attributes. These attributes are useful for filtering traces, for example, if you want to search for a specific trace or time interval.

Note:
- Adding your custom attributes for only one span (Activity) is currently not supported. Any custom tags that you create are added to **all** traces
- The prefix '`flogo`' is added to tags with product-specific attributes only.

The following attributes specific to Flogo are available:

For flows	
flogo.event.id	The event ID is the unique ID of a single request, job or a action initiated by the user.
`flogo.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).
`flogo.flow.name`	Name of the flow.
For activities	
`flogo.flow.name`	Name of the flow.
`flogo.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).
`flogo.task.name`	Name of the activity.
`flogo.task.type`	Type of the activity.
`flogo.taskInstance.id`	Unique instance ID that is 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 constructs such as **iterator** or **Repeat while true**.
For subflows	
`flogo.parent.flow`	Name of the parent flow.
`flogo.flogo.parent.flow.id`	Unique ID of the parent flow.
`flogo.flow.name`	Name of the subflow.
`flogo.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 attribute values are automatically generated at runtime. You cannot override the default values. You have the option to set attributes by setting them in the environment variable `FLOGO_OTEL_TRACE_ATTRIBUTES` as key/value pair. Keep in mind that these tags are added to **all** traces.

Using APIs

You can obtain the runtime statistics of the Go language in Flogo Enterprise.

Healthcheck API

Flogo Enterprise runtime allows you to enable healthcheck for a Flogo app that is running.

To enable healthcheck for your running app:

Procedure

1. Set `FLOGO_HTTP_SERVICE_PORT` to enable runtime HTTP Service as follows:

```
FLOGO_HTTP_SERVICE_PORT=<port> ./<app_name>
```

2. Run the following command:

```
curl http://localhost:<port>/ping
```

Note: Currently, healthcheck endpoint returns HTTP status 200 only when all triggers in the app are successfully started. Otherwise, it returns HTTP status 500.

Go Language Runtime Statistics and Profiling

Flogo Enterprise allows you to gather runtime system statistics for a Flogo app that is running.

Warning: Your management port must be set for the Flogo app, to call the API to gather Go language runtime statistics. To set a different management port for your Flogo app, run `FLOGO_HTTP_SERVICE_PORT=<port>./<app-name>/`You can use curl to call this API.

To obtain the system statistics on your running app:

Procedure

1. From the folder in which your app binary resides, enable the HTTP service using the following command:

```
FLOGO_HTTP_SERVICE_PORT=<port> ./<app_name>
```

2. Run the following command:

```
curl http://localhost:<port>/debug/vars
```

The command returns the following statistics:

System Metric Name	Description
cmdline	Command-line arguments passed to the app binary
cpus	Number of logical CPUs usable by the current process
goroutines	The number of Go routines that currently exist
memstats	Memory statistics for the current process. See the Golang documentation for details.
processid	System process ID
version	Go language version used to build the app

Profiling your app runtime

You can collect and visualize runtime profiling data for Flogo apps using the `pprof` tool in Golang.

Endpoint	Description
/debug/pprof	List all profiles
/debug/pprof/profile	Profile current CPU usage. By default, it is profiled for every 30 seconds. To change the profiling interval, set the `seconds` query parameter to a desired value. For example, `go tool pprof http://localhost:<port> /debug/pprof/profile?seconds=15`
/debug/pprof/heap	A sampling of memory allocations of live objects. For example, `go tool pprof http://localhost:<port>/debug/pprof/heap`

Endpoint	Description
/debug/pprof/goroutine	Stack traces of all current Go routines. For example, `go tool pprof http://localhost:<port>/debug/pprof/goroutine`
/debug/pprof/trace	A trace of execution of the current program. For example, `go tool pprof http://localhost:<port>/debug/pprof/trace`

CPU and Memory Profiling

If you observe low throughputs or high memory usage, you can enable CPU and/or Memory profiling for your Flogo app. Enabling this profiling impacts performance. Hence, we do not recommend enabling them in a production environment.

Before you begin

- You must have GO version 1.9.0 or higher installed.
- Make sure that the `pprof` tool is installed on your machine. Refer to PPOF for more details on the `pprof` tool.

Enabling CPU Profiling

To enable CPU profiling:

Procedure

1. Open a command prompt or terminal.
2. Change the directory to the folder in which your app binary is located.
3. Run the following command:

   ```
   ./<app_binary> -cpuprofile <file>
   ```

   where *<file>* is the profile file. For example, `./StockService -cpuprofile`

```
/home/users/StockService_cpu.prof
```

Enabling Memory Profiling

To enable memory profiling:

Procedure

1. Open a command prompt or terminal.
2. Change the directory to the folder in which your app binary is located.
3. Run the following command:

   ```
   ./<app_binary> -memprofile <file>
   ```

   where *<file>* is the profile file. For example, `./StockService -memprofile /home/users/StockService_mem.prof`

Enabling CPU and Memory Profiling in a Single Command

To enable CPU and memory profiling in a single command:

Procedure

1. Open a command prompt or terminal.
2. Change the directory to the folder in which your app binary is located.
3. Run the following command:

   ```
   ./<app_binary> -memprofile <file> -cpuprofile <file>
   ```

Analyzing your profiling data

Once you capture the profiling data, analyze it using `pprof` by running the following command:

```
go tool pprof <profile file>
```

Monitoring and Managing Enterprise Apps in TIBCO Cloud Integration

With the TIBCO Cloud Integration - Hybrid Agent, you can now monitor Remote apps and perform various operations through the TIBCO Cloud Integration user interface, such as scaling the app instances, updating application and engine variables, starting or stopping an app, and monitoring app metrics. Remote apps are auto-discovered by the Hybrid Agent.

For detailed information, see Configuring Remote Apps.

Environment Variables

This section lists the environment variables that are associated with the Flogo Enterprise runtime environment.

Environment Variable Name	Default Value	Description
FLOGO_RUNNER_QUEUE_SIZE	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 run by the app engine from the queue.
FLOGO_LOG_LEVEL	INFO	Used to set a log level for the Flogo app. Supported values are: • INFO • DEBUG • WARN • ERROR This variable is supported for Remote

Environment Variable Name	Default Value	Description
		Apps managed with the TIBCO Cloud Integration Hybrid Agent.
FLOGO_ LOGACTIVITY_ LOG_LEVEL	INFO	Used to control logging in the Log activity. Values supported, in the order of precedence, are: • DEBUG • INFO • WARN • ERROR For example: • If the Log level is set to WARN, WARN and ERROR logs are filtered and displayed. • If Log Level is set to DEBUG, then DEBUG, INFO, WARN, and ERROR logs are displayed.
FLOGO_ MAPPING_SKIP_ MISSING	False	When mapping objects if one or more elements are missing in either the source or target object, the mapper generates 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 Name	Default Value	Description
		environment variable is set to `False`. To set it to `True`, run: `export FLOGO_APP_METRICS_LOG_EMITTER_ENABLE=true`

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 displays 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 the 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"]}'`

FLOGO_APP_METRICS	70	Enables app metrics on the **Monitoring** tab.
FLOGO_APP_MEM_ALERT_THRESHOLD	70	The threshold for memory utilization of the app. When the memory utilization by an app running in a container exceeds the threshold that you have specified, you get a warning log
FLOGO_APP_CPU_	70	The threshold for CPU utilization of the

Environment Variable Name	Default Value	Description
ALERT_ THRESHOLD		app. When the CPU utilization by an app running in a container exceeds the threshold that you have specified, you get a warning log
FLOGO_APP_ DELAYED_STOP_ INTERVAL	10 seconds	When you scale down an instance, all inflight jobs are lost because the engine is stopped immediately. To avoid losing the jobs, delay the stopping of the engine by setting the FLOGO_APP_ DELAYED_STOP_INTERVAL variable to a value less than 60 seconds. Here, when you scale down the instance, if there are no inflight jobs running, then the engine stops immediately without any delay. In case of inflight jobs: • If there are any inflight jobs running, then the engine stops immediately after the inflight job is completed. • If the inflight job is not completed within a specified time interval, then the job gets canceled and the engine stops.
FLOGO_APP_ DELAYED_STOP_ INTERVAL	10 seconds	When you scale down an instance, all inflight jobs are lost because the engine is stopped immediately. To avoid losing the jobs, delay the stopping of the engine by setting the FLOGO_APP_ DELAYED_STOP_INTERVAL variable to a value less than 60 seconds. Here, when you scale down the instance, if there are no inflight jobs running, then the engine stops immediately without any delay. In

Environment Variable Name	Default Value	Description
		case of inflight jobs: • If there are any inflight jobs running, then the engine stops immediately after the inflight job is completed. • If the inflight job is not completed within a specified time interval, then the job gets canceled and the engine stops.
GOGC	100	Sets the initial garbage collection target percentage. Setting it to a higher value delays the start of a garbage collection cycle until the live heap has grown to the specified percentage of the previous size. Setting it to a lower value causes the garbage collector to be triggered more often as less new data can be allocated to the heap before triggering a collection.

This section lists the user-defined environment variables that are associated with the Flogo Enterprise runtime environment.

Environment Variable Name	Default Value	Description
FLOGO_LOG_CTX	False	Used to enable context logging for the application. When set to true, context, such as application name, version, tracing details, and flow details, is added to the engine and connectors logs.

Environment Variable Name	Default Value	Description
		Note: This context is always logged in JSON format.
FLOGO_LOG_CTX_FIELDS	None	When context logging is enabled, set custom context fields in the logging. These additional fields are added to the logging context. Example: `FLOGO_LOG_CTX_FIELDS="service.name=Foo,service.version=1.0.0,service.environment=dev"`
FLOGO_ENV	None	Used to set the name of the deployment environment, such as dev, staging, and production. When enabled, by default, the **deployment.environment** field is set in the logging context (if enabled) and in Flogo OpenTelemetry traces and metrics.
FLOGO_MAPPING_OMIT_NULLS	True	Used to omit all the keys in the activity input evaluating to null.
FLOGO_FLOW_CONTROL_EVENTS	False	If you set `FLOGO_FLOW_CONTROL_EVENTS` as true, the Flow limit functionality is enabled, whenever the incoming requests to trigger reach `FLOGO_RUNNER_QUEUE_SIZE` limit then trigger is paused. When all the requests currently under processing are finished, the trigger is resumed again. All the connectors supporting the Flow limit functionality are mentioned in their respective user guides.
FLOGO_HTTP_SERVICE_PORT	None	Used to set the port number to enable runtime HTTP service, which provides APIs for health check and statistics.

Environment Variable Name	Default Value	Description
FLOGO_LOG_FORMAT	TEXT	Used to switch the logging format between text and JSON. For example, to use the JSON format, set FLOGO_LOG_FORMAT=JSON ./<*app-name*>
FLOGO_MAX_STEP_COUNT	None	The application stops processing requests after the `FLOGO_MAX_STEP_COUNT` limit is reached. The default limit is set to 10 Million even when you do not add this variable.
FLOGO_EXPOSE_SWAGGER_EP	False	If you set this property to `True`, the Swagger endpoint is exposed. The Swagger of the Rest trigger app can be accessed by hitting the Swagger endpoint at `http://<service-url>/api/v2/swagger.json`.
FLOGO_SWAGGER_EP	None	To customize the URI for the Swagger endpoint, set this environment variable to your desired endpoint. For example: `FLOGO_SWAGGER_EP=/custom/swagger/endpoint` This makes the Swagger endpoint available at `/custom/swagger/endpoint` instead of the default `/api/v2/swagger.json`.
FLOGO_OTEL_SPAN_KIND	INTERNAL	Used to specify the type of span to be used in OpenTelemetry. The supported values are `INTERNAL`, `SERVER`, `CLIENT`, `PRODUCER`, and `CONSUMER`. **Note:** If no value or an invalid value is provided, the default value is set to `INTERNAL`.
FLOGO_LOG_CONSOLE_STREAM	stderr	Used to specify the logging output stream for Flogo engine and app logs. The supported values are `stdout` and `stderr`.

Pushing Apps to TIBCO Cloud

You can push apps that were created in Flogo Enterprise to TIBCO Cloud Integration using the TIBCO Cloud - Command Line Interface (`tibcli`).
You must download the TIBCO Cloud Integration artifacts to use TIBCO Cloud CLI to push the apps.

Before you begin

You must have the TIBCO Cloud CLI installed on your local machine before you follow this procedure. Refer to the "Downloading TIBCO Cloud Integration Tools" and "Installing the TIBCO®Cloud - Command Line Interface" sections in the TIBCO Cloud Integration documentation for details on how to download the TIBCO Cloud CLI and install it.

Note: For REST apps, be sure to change the port to 9999 *before* downloading the artifacts.

To push the app using the TIBCO Cloud CLI, follow this procedure:

Procedure

1. On the app details page, click **Export**.
2. Select **TIBCO Cloud Integration artifacts**.

   The `manifest.json` and `flogo.json` files are downloaded. The `manifest.json` contains the manifest details such as the endpoints, memory resource details, and so on. The `flogo.json` contains the app itself. These are the artifacts needed to push the app directly from Flogo Enterprise using TIBCO Cloud CLI.
3. Create a temporary directory on your machine.
4. Move the downloaded `flogo.json` and the `manifest.json` files into a temporary directory.

   Note: The `tibcli` or `tibcli.exe` executable should not be in the same directory (the temporary directory you created) as the app you are pushing.
5. Open a terminal or command prompt and navigate to the temporary directory.
6. Run the following command to push the app:

```
tibcli app push <app-name>
```

Important: If there is an existing app with an identical name as the app that you are trying to push to the cloud, the existing app is overwritten with the newly pushed app. You do not get a warning about it.

Result

The app is pushed to TIBCO Cloud Integration. You can see the progress of the app push on the UI. After the app is pushed, the app implementation details on the **Flow**tab are replaced with the actual flow.

Best Practices

For efficient development of Flogo apps, follow these best practices:

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 that needs to be returned to either the trigger (in the 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 might 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 to the trigger configuration.

In such a scenario, you should add the **ConfigureHTTPResponse** Activity in the flow before the **Return** Activity, 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 the 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 the status code range in a format such as 2xx if the same schema is being used for all codes in that range.

- **Reserved keywords**

  Flogo Enterprise 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 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 **Sync** for the schema changes to be propagated to the flow parameters. For more information, see Synchronizing Schema Between Trigger and Flow.

- **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. For more information, seeUsing Functions.
- Expressions - You can enter an expression whose evaluated value is mapped to the input field. For more information, see Using Expressions.

- **Complex data mappings**
  - Using the `array.forEach()` mapper function, you can map complex nested arrays, filter elements of an array based on a condition and 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 at 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.

To handle network faults, Flogo Enterprise provides the ability to configure the Timeout and Retry on Error settings for some specific activities such as **InvokeRESTService** and **TCMMessagePublisher**. Refer to the "General Category Triggers and Activities" section of the *TIBCO Flogo® Enterprise Activities, Triggers, and Connections Guide* for details on each **General** category Activity and trigger.

Deployment and Configuration

Memory considerations

When Flogo apps are deployed in TIBCO Cloud™ Integration, keep in mind that a maximum 1GB of memory is allocated to each app instance. If the Flogo app flow execution is memory heavy, the container is stopped due to lack of required memory and the following error message is displayed:

```
502 Bad Gateway Error
```

Using environment variables

When deploying a Flogo app, you can override the 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.

Generating and using SSL certificates

When generating an SSL certificate, it is recommended that you use Public DNS as a Common Name. Also, when using an SSL certificate, use Public DNS instead of IP address.

Building Engine binary

For multiple apps that have a common set of functionality, you can build a generic Flogo Enterprise binary instead of building a separate binary for each app.

Performance Tuning

This section provides guidelines that can be used to understand your performance objectives and fine-tune the app environment to optimize performance.

The performance of an app affects stability, scalability, throughput, latency, and resource utilization. For optimal performance of the app, it is important to understand the various levels at which the tuning methods and best practices can be applied to the components. This section includes the different tuning parameters, steps required to configure the parameters, and design techniques for better performance.

This section must be used along with other product documentation and project-specific information to achieve the desired performance results. The goal is to assist in tuning and optimizing the runtime for the most common scenarios. At the same time, one must focus on real-life scenarios to understand the issue and the associated solution.

Note: The performance tuning and configurations in this section are provided for reference only. They can be reproduced only in the exact environment and under workload conditions that existed when the tests were done. The numbers in the document are based on the tests conducted in the performance lab and may vary according to the components installed, the workload, the type and complexity of different scenarios, hardware, and software configuration, and so on. The performance tuning and configurations should be used only as a guideline, after validating the customer requirements and environment. TIBCO does not guarantee its accuracy.

Tuning Environment Variables

This section lists the environment variables associated with the TIBCO Flogo environment. Details such as the default value of environmental variables and how we can change them are also included.

FLOGO_RUNNER_TYPE

This variable defines how events are handled by the Flogo engine.

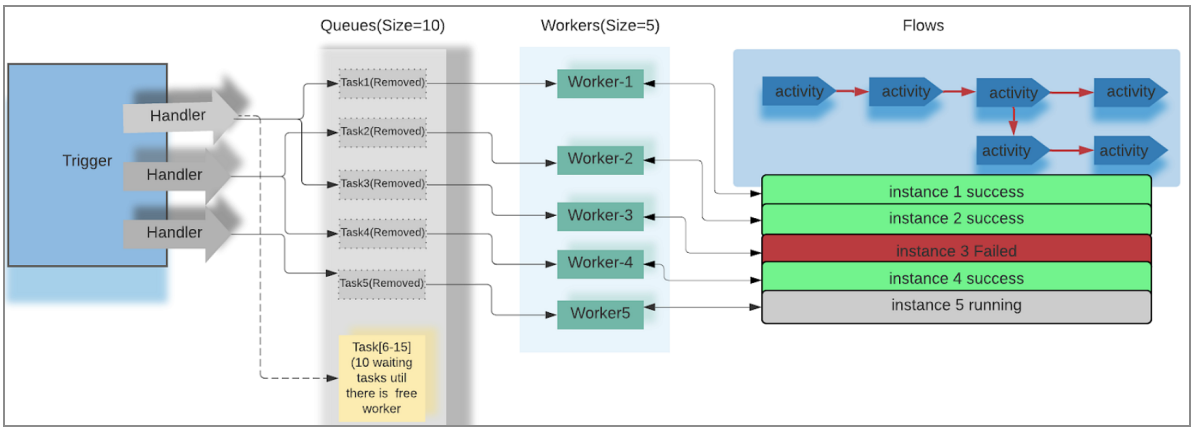
- Supported values: DIRECT and POOLED.
- Default: POOLED

POOLED Mode

In this mode, the engine handles events in a flow-controlled way.

The following pictorial diagram explains the handling of events in POOLED mode.

Events in POOLED mode



Sets of workers are created to handle events received by all the triggers in the given Flogo app. In golang terms, one worker corresponds to one go-routine. The events received are added to the worker queue before the workers can pick these events from the worker queue.

Once an event is picked from the queue, the corresponding action (for example, flow) is triggered and the worker continues to execute that action until completion (that is, until the action is successful or fails). An event that is picked up from the queue is removed to allow the next event to be added to the queue.

When the queue is full, all trigger handlers that are adding new events to the queue are blocked until workers pick up the next set of events from the queue. Once the worker starts executing the action, it never interleaves the action until its completion. So, the total number of events processed at a time is directly proportional to the time taken by the action to complete and the number of workers in the pool. Hence, for better concurrency,

gradually increase the value of queues and workers based on the available compute resources (such as CPU and memory).

Configurations in POOLED mode:

You can configure the workers and the queue size by setting FLOGO_RUNNER_WORKERS and FLOGO_RUNNER_QUEUE_SIZE respectively.

- FLOGO_RUNNER_WORKERS variable determines the maximum number of concurrent events that can be executed by the app engine from the queue. FLOGO_RUNNER_WORKERS execute a finite number of tasks or concurrent events uninterrupted and then yield to the next ready job. FLOGO_RUNNER_WORKERS can be tuned to the optimum value by starting with a default value set and increasing it as per requirement until the maximum CPU is reached.

  The default value is FLOGO_RUNNER_WORKERS=5.

- FLOGO_RUNNER_QUEUE_SIZE variable specifies the maximum number of events from all triggers that can be queued by the app engine. FLOGO_RUNNER_QUEUE_SIZE can be tuned to the optimum value by starting with a default value set and increasing it as per requirement. You can change the variable value if you anticipate having more than default value events queued at the same time.

  The default value is FLOGO_RUNNER_QUEUE_SIZE=50.

The CPU and memory resources must be measured under a typical processing load to determine if the default variable value is suitable for the environment. If the user load is more than the default set value, the user can change the runner worker variable as per the requirement to expedite the execution of the concurrent events. Set variable values according to your processing volumes, number of CPUs, and allocated memory.

Deploying the app to your environment

Set the variable value as follows:

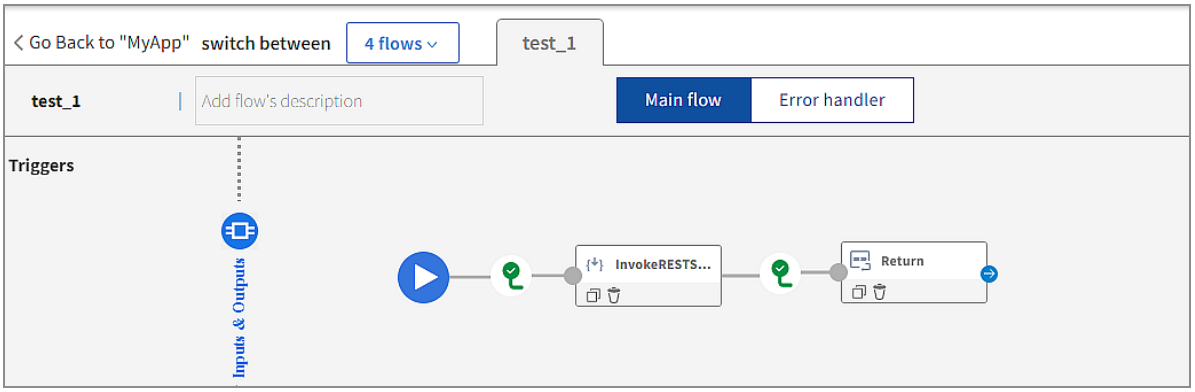
```
FLOGO_RUNNER_WORKERS=75  FLOGO_RUNNER_QUEUE_SIZE =150  ./<app_binary>
```

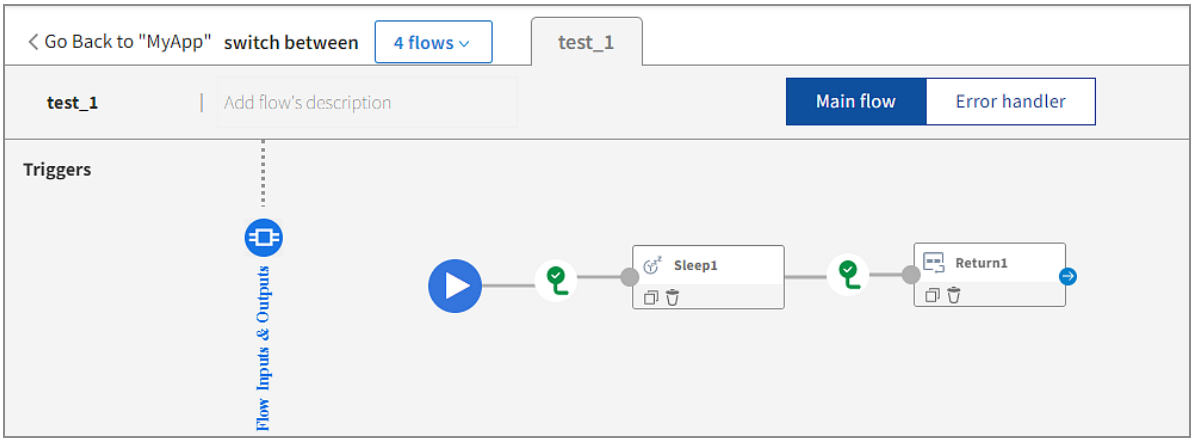
```
docker run -it -e FLOGO_RUNNER_WORKERS=75 -e FLOGO_RUNNER_QUEUE_SIZE=150
<docker-image>
```

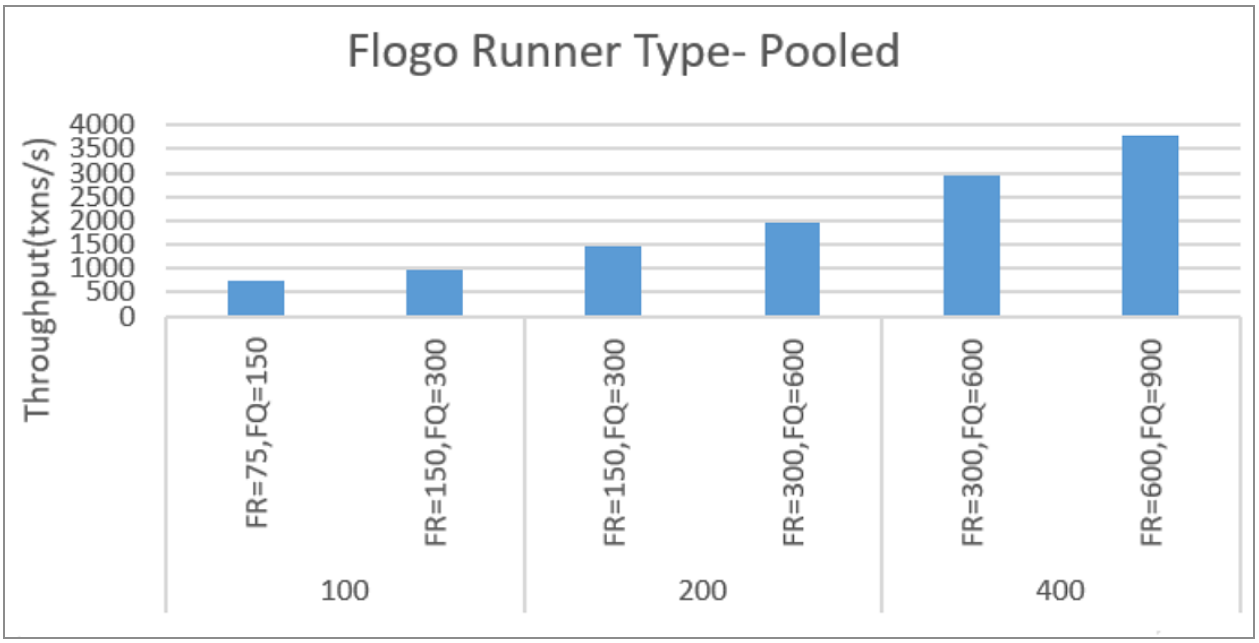
Case Study

While setting up the FLOGO_RUNNER_TYPE as POOLED, Flogo runner workers and Flogo runner queues are used to handling events received by the trigger. You can increase the Flogo runner worker and queue values gradually to reach the app performance. Set variable values according to your processing volumes concerning your number of CPUs and allocated memory.

It is recommended that you set the queue size greater than or equal to the number of workers.



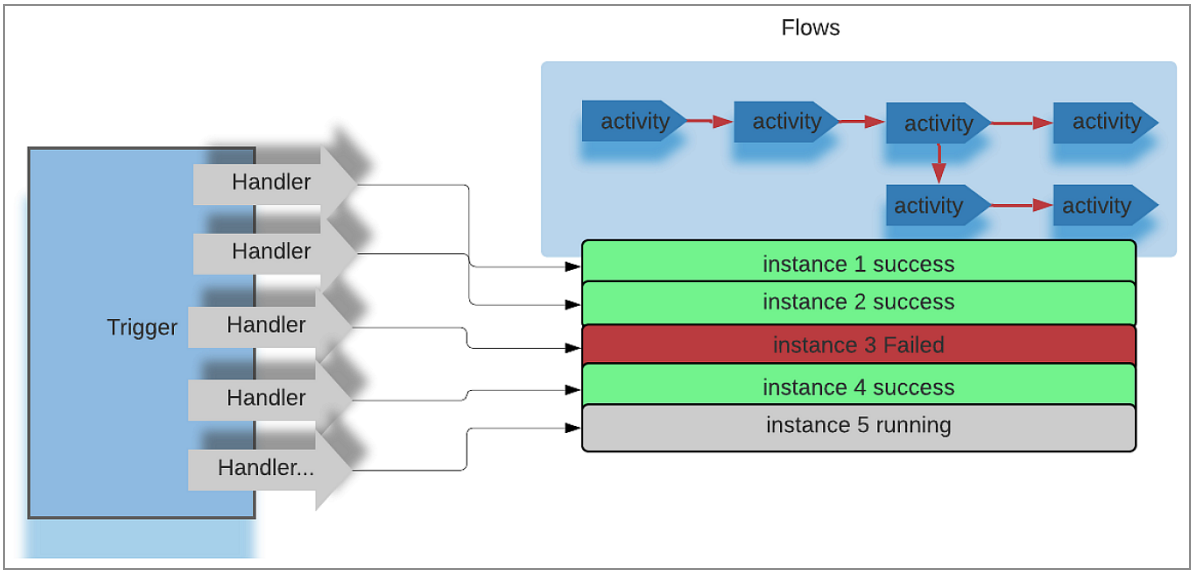




DIRECT Mode

In this mode, every event delivered by the handler triggers a corresponding action. Unlike the POOLED mode, the handling of events is unbounded. All the events are processed concurrently. This might lead to CPU saturation or out-of-memory errors.

The following pictorial diagram explains the handling of events in DIRECT mode.



Deploying the app to your environment

Set the variable value as follows:

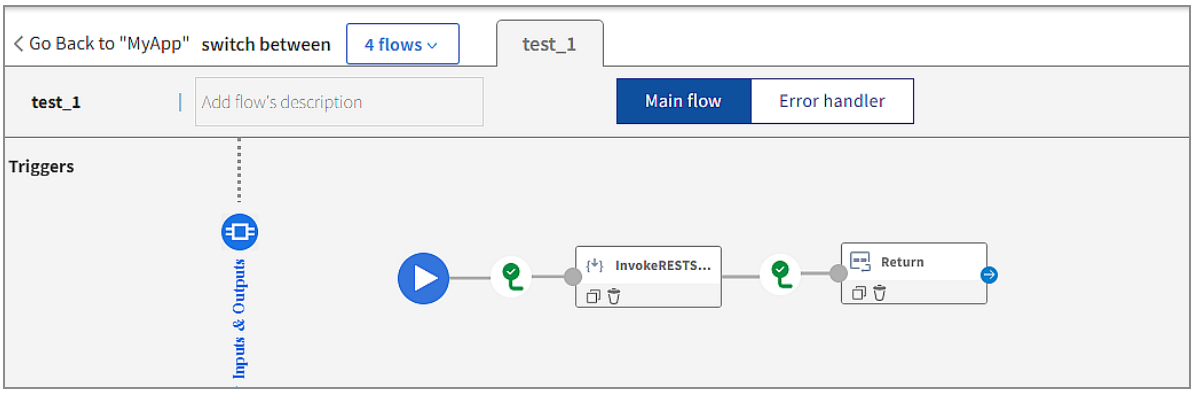
```
FLOGO_RUNNER_TYPE=DIRECT ./<app_binary>
```

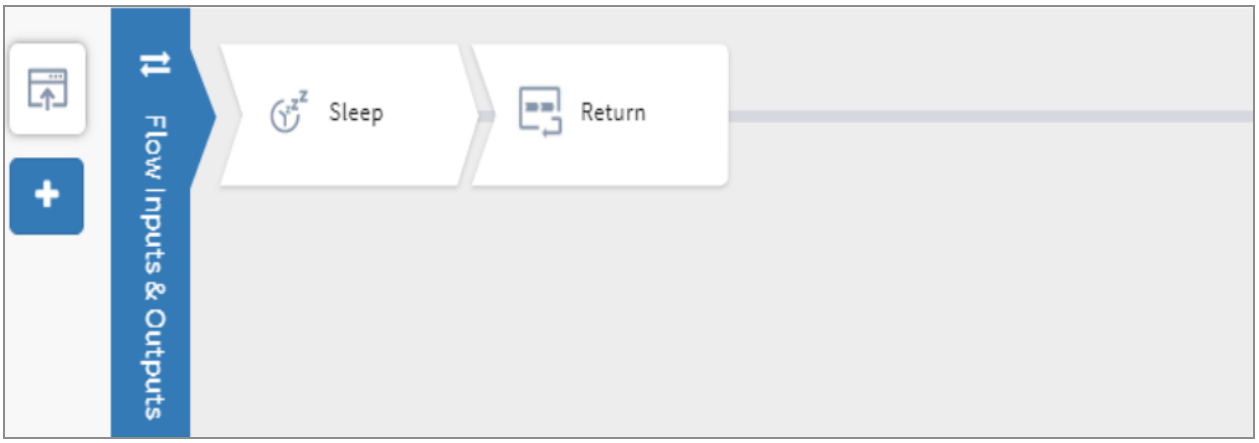
```
docker run -it -e FLOGO_RUNNER_TYPE=DIRECT <docker-image>
```

Case Study

This case study illustrates the app performance when Flogo event handling mode is set to DIRECT.

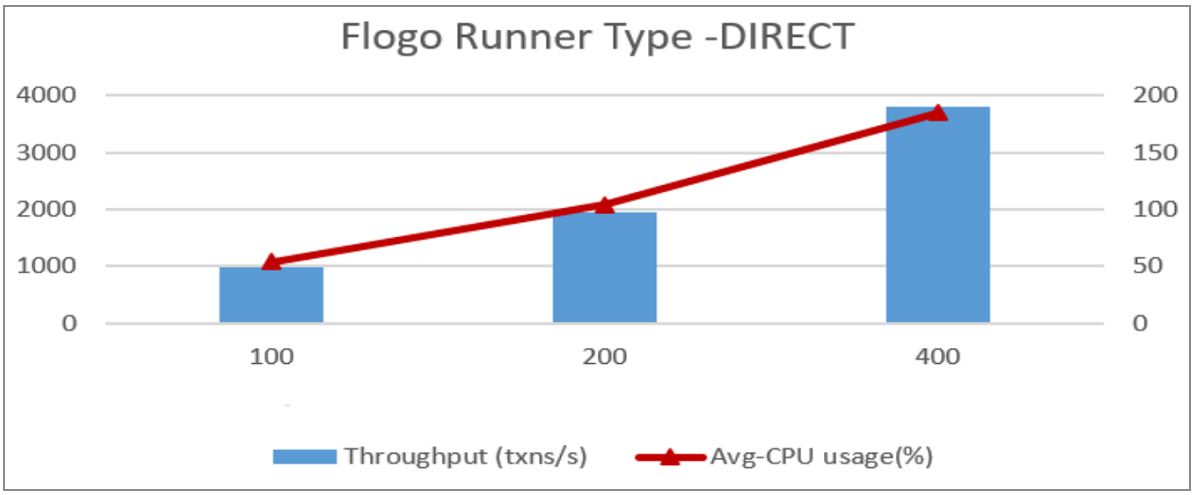
App under test - FLOGO_RUNNER_TYPE





While setting up the FLOGO_RUNNER_TYPE as DIRECT, all the events sent to the trigger are processed concurrently. As you keep on increasing the concurrency, you can observe the linear increase in resources, that is, CPU and memory utilization.

Flogo Engine - Direct Mode



FLOGO_LOG_LEVEL

This environment variable is used to set a log level for an app.

- Supported values: INFO, DEBUG, WARN, and ERROR.
- Default: INFO

You can increase or decrease the logging of the app using this environment variable. To increase the logging of the app to debug, change FLOGO_LOG_LEVEL to DEBUG. To skip detailed logging and to just log an error, set FLOGO_LOG_LEVEL to ERROR. Changes to the log level are reflected after restarting the Flogo app in your environment and by pushing the Flogo app again to the cloud environment.

Deploying the app to your environment

Set the variable value as follows:

```
FLOGO_LOG_LEVEL=ERROR ./<app_binary>
```

```
docker run -it -e FLOGO_LOG_LEVEL=ERROR <docker-image>
```

Log level - ERROR

```
[ec2-user@ip-172-31-9-14 products]$ FLOGO_LOG_LEVEL=ERROR ./Rest-Invoke-server-linux_amd64
TIBCO Flogo® Runtime - 2.11.0 (Powered by Project Flogo™ - v1.2.0)
TIBCO Flogo® connector for General -  1.2.0.455
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      Starting TIBCO Flogo® Runtime
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      Runtime started in 1.658066ms
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Log level - INFO

```
[ec2-user@ip-172-31-9-14 products]$ FLOGO_LOG_LEVEL=INFO ./Rest-Invoke-server-linux_amd64
TIBCO Flogo® Runtime - 2.11.0 (Powered by Project Flogo™ - v1.2.0)
TIBCO Flogo® connector for General -  1.2.0.455
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      Starting TIBCO Flogo® Runtime
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2021-01-11T07:30:12.720Z WARN [flogo] - unable to create child logger named: ReceiveHTTPMessage - unable to create child logger
2021-01-11T07:30:12.720Z INFO [general-trigger-rest] - Name: ReceiveHTTPMessage, Port: 9999
2021-01-11T07:30:12.720Z INFO [general-trigger-rest] - ReceiveHTTPMessage: Registered handler [Method: POST, Path: /echo]
2021-01-11T07:30:12.720Z INFO [flogo.engine] - Starting app [ Rest-Invoke-server ] with version [ 1.1.0 ]
2021-01-11T07:30:12.720Z INFO [flogo.engine] - Engine Starting...
2021-01-11T07:30:12.720Z INFO [flogo.engine] - Starting Services...
2021-01-11T07:30:12.720Z INFO [flogo] - ActionRunner Service: Started
2021-01-11T07:30:12.720Z INFO [flogo.engine] - Started Services
2021-01-11T07:30:12.720Z INFO [flogo.engine] - Starting Application...
2021-01-11T07:30:12.720Z INFO [flogo] - Starting Triggers...
2021-01-11T07:30:12.720Z INFO [general-trigger-rest] - Starting ReceiveHTTPMessage...
2021-01-11T07:30:12.720Z INFO [general-trigger-rest] - Started ReceiveHTTPMessage
2021-01-11T07:30:12.720Z INFO [flogo] - Trigger [ ReceiveHTTPMessage ]: Started
2021-01-11T07:30:12.720Z INFO [flogo] - Triggers Started
2021-01-11T07:30:12.720Z INFO [flogo.engine] - Application Started
2021-01-11T07:30:12.720Z INFO [flogo.engine] - Engine Started
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      Runtime started in 2.956203ms
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

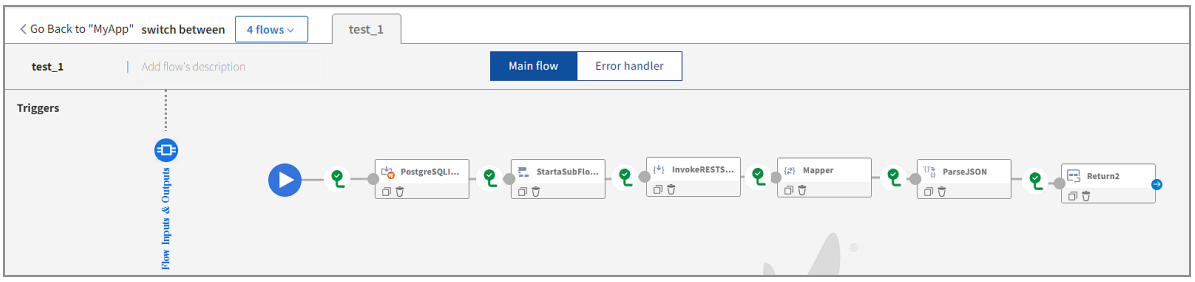
Log level - DEBUG

```
2021-01-11T07:32:25.034Z DEBUG [flogo] - Resolved function 'git.tibco.com/git/product/ipaas/wi-contrib.git/function/datetime:create' to
'datetime.create
2021-01-11T07:32:25.034Z DEBUG [flogo.activity.actreturn] - Mappings: map[code:200 data:map[mapping:map[data:=$flow.body.data]]]
2021-01-11T07:32:25.034Z WARN [flogo] - unable to create child logger named: ReceiveHTTPMessage - unable to create child logger
2021-01-11T07:32:25.034Z DEBUG [general-trigger-rest] - In init, id 'ReceiveHTTPMessage'
2021-01-11T07:32:25.034Z INFO [general-trigger-rest] - Name: ReceiveHTTPMessage, Port: 9999
2021-01-11T07:32:25.034Z INFO [general-trigger-rest] - ReceiveHTTPMessage: Registered handler [Method: POST, Path: /echo]
2021-01-11T07:32:25.034Z DEBUG [flogo.engine] - Creating app [ Rest-Invoke-server ] with version [ 1.1.0 ]
2021-01-11T07:32:25.034Z INFO [flogo.engine] - Starting app [ Rest-Invoke-server ] with version [ 1.1.0 ]
2021-01-11T07:32:25.034Z INFO [flogo.engine] - Engine Starting...
2021-01-11T07:32:25.034Z INFO [flogo.engine] - Starting Services...
2021-01-11T07:32:25.034Z DEBUG [flogo] - ActionRunner Service: Starting...
2021-01-11T07:32:25.034Z DEBUG [flogo] - Starting worker with id '1'
2021-01-11T07:32:25.034Z DEBUG [flogo] - Starting worker with id '2'
2021-01-11T07:32:25.035Z DEBUG [flogo] - Starting worker with id '3'
2021-01-11T07:32:25.035Z DEBUG [flogo] - Starting worker with id '4'
2021-01-11T07:32:25.035Z DEBUG [flogo] - Starting worker with id '5'
2021-01-11T07:32:25.035Z INFO [flogo] - ActionRunner Service: Started
2021-01-11T07:32:25.035Z INFO [flogo.engine] - Started Services
2021-01-11T07:32:25.035Z INFO [flogo.engine] - Starting Application...
2021-01-11T07:32:25.035Z INFO [flogo] - Starting Triggers...
2021-01-11T07:32:25.035Z DEBUG [flogo] - Trigger [ ReceiveHTTPMessage ]: Starting...
2021-01-11T07:32:25.035Z INFO [general-trigger-rest] - Starting ReceiveHTTPMessage...
2021-01-11T07:32:25.035Z INFO [general-trigger-rest] - Started ReceiveHTTPMessage
2021-01-11T07:32:25.035Z INFO [flogo] - Trigger [ ReceiveHTTPMessage ]: Started
2021-01-11T07:32:25.035Z DEBUG [flogo] - Trigger [ ReceiveHTTPMessage ] has ref [ git.tibco.com/git/product/ipaas/wi-contrib.git/contrib
utions/General/trigger/rest ] and version [  ]
2021-01-11T07:32:25.035Z INFO [flogo] - Triggers Started
2021-01-11T07:32:25.035Z INFO [flogo.engine] - Application Started
2021-01-11T07:32:25.035Z INFO [flogo.engine] - Engine Started
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      Runtime started in 2.593609ms
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Case Study

This use case illustrates the app logging impact on the performance of the app.

App under test for Flogo Log level



Performance lab results have shown that the performance of the app depends on the app log level that is set, request payload, and app latency. Set the log level to DEBUG functional issues and to ERROR for performance scenarios because setting the logging to DEBUG might impact the performance of the app.

Maximum throughput was achieved with a Log Level set as ERROR.

GOGC

The GOGC variable sets the initial garbage collection target percentage. A collection is triggered when the ratio of freshly allocated data to live data remaining after the previous collection reaches this percentage.

Garbage collection refers to the process of managing heap memory allocation: free the memory allocations that are no longer in use and keep the memory allocations that are being used. Garbage collection significantly affects the performance of your app.

Deploying the app to your environment

Set the variable value as follows:

```
GOGC=150  ./<app_binary>
```

```
docker run -it -e  GOGC=150  <docker-image>
```

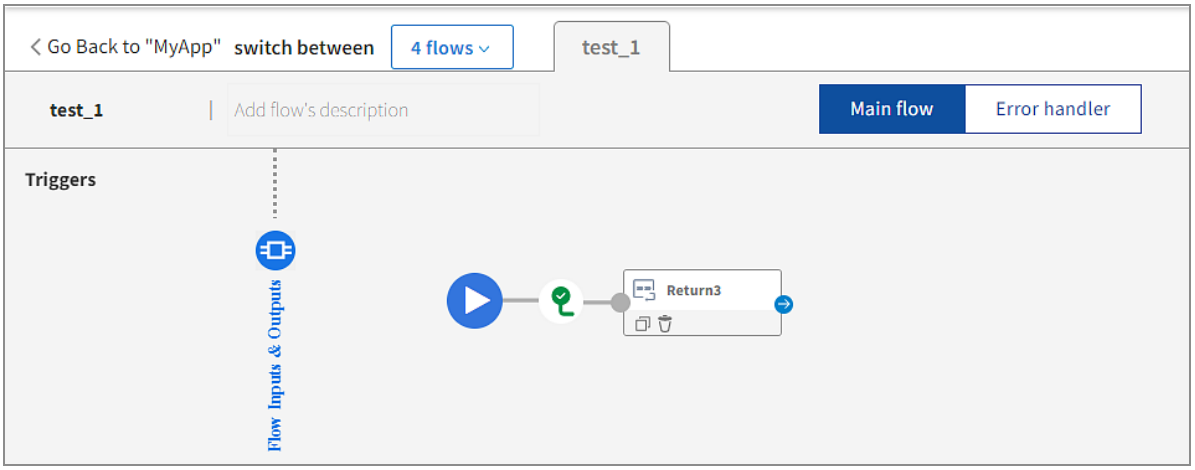
The default is 100. This means that garbage collection is not triggered until the heap has grown by 100% since the previous collection. Setting the variable to a higher value (for example, GOGC=200) delays the start of a garbage collection cycle until the live heap has

grown to 200% of the previous size. Setting the variable to a lower value (for example, GOGC=20) increases the frequency of garbage collection as less new data can be allocated on the heap before triggering a collection.

Case Study

This use case illustrates the impact of the GOGC variable on performance.

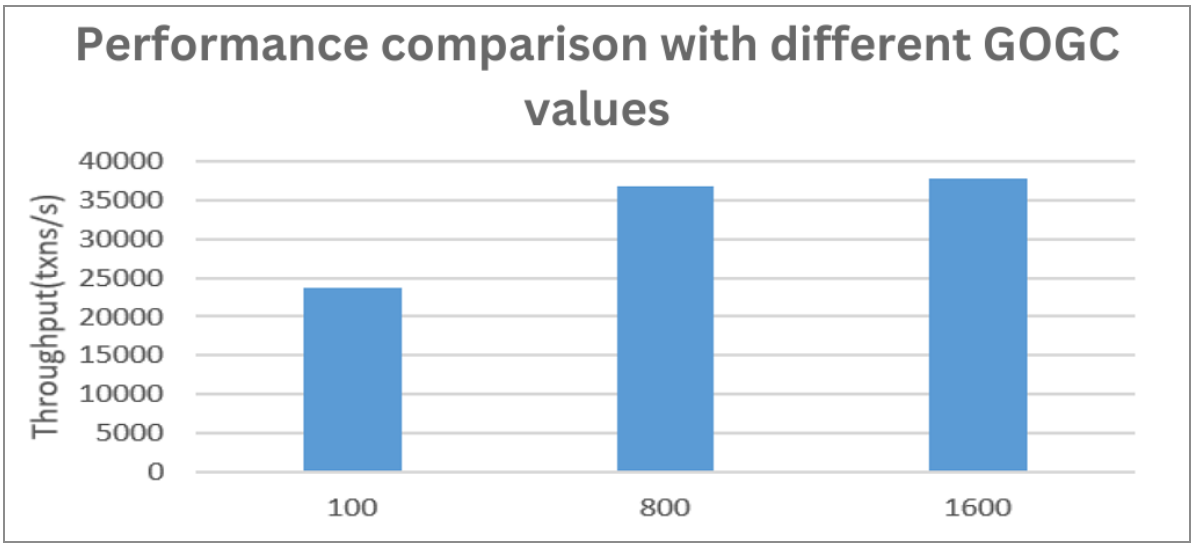
App under test - GOGC



In this low latency scenario, you can observe significant improvement in-app performance while increasing the GOGC variable value from 100 to 1600. It is advisable to test this value for the specific scenario and understand its impact before tuning. You can get the best-suited value by running the performance test in your test environment.

GOGC value can be tuned based on the workload and available resources after validating your test environment.

Performance comparison with different GOGC values



Flow Limit

Flow limit is useful when the engine needs to be throttled, as the `FLOGO_ RUNNER_QUEUE_SIZE` engine variable specifies the maximum number of events that can be started before pausing the process trigger. This ensures that the incoming requests do not overwhelm the engine performance and the CPU and memory is preserved.

If the number of incoming requests on the trigger exceeds the `FLOGO_ RUNNER_QUEUE_SIZE` limit, the engine pauses the trigger to get any new requests, but continues running the existing ones. The engine resumes this trigger when all the requests currently under processing are finished.

And this flow limit is not enforced by the engine unless the FLOGO_FLOW_CONTROL_EVENTS variable is set to `true` for an application as an user-defined **Engine Variable** on the **Environment Variable** tab of the Flogo Enterprise runtime environment.

Environment variables associated with Flow Limit

Environment Variable Name	Default Values	Description
FLOGO_RUNNER_QUEUE_SIZE	50	The maximum number of events from all triggers that can

Environment Variable Name	Default Values	Description
		be queued by the app engine.
FLOGO_RUNNER_WORKERS	5	The maximum number of concurrent events that can be run by the app engine from the queue.
FLOGO_FLOW_CONTROL_EVENTS	N/A	If you set `FLOGO_FLOW_CONTROL_EVENTS`as true, the Flow limit functionality is enabled, whenever the incoming requests to trigger reaches `FLOGO_RUNNER_QUEUE_SIZE`limit then trigger is paused. When all the requests currently under processing are finished, the trigger is resumed again.

Note: All the connectors supporting the flow limit functionality are mentioned in their respective user guides.

CPU and Memory Monitoring

Top Command

Note: The `top` command works on Linux platforms only.

The `top` command is used for memory and CPU monitoring.

The `top` command produces an ordered list of running processes selected by user-specified criteria. The list is updated periodically. By default, ordering is by CPU usage and it shows the processes that consume maximum CPU. The `top` command also shows how much

processing power and memory are being used, as well as the other information about the running processes.

The `top` command output monitors the memory as well as the CPU utilization of the TIBCO Flogo app binary.

The sample output is as follows:

```
top -p PID > top.txt

Cpu(s):  4.7%us,  1.1%sy,  0.0%ni, 94.1%id,  0.0%wa,  0.0%hi,  0.1%si,  0.0%st
Mem:  65914304k total, 59840516k used,  6073788k free,  3637208k buffers
Swap: 15359996k total,   119216k used, 15240780k free, 43597120k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+
COMMAND
```

Docker Stats Command

The `docker stats` command returns a live data stream for running containers. To limit data to one or more specific containers, specify a list of container names or ids separated by a space.

The `docker stats` command output monitors the memory as well as the CPU utilization of the TIBCO Flogo Enterprise app container and TCI Flogo app container.

- `CPU %` is the percentage of the host's CPU the container is using.
- `MEM USAGE / LIMIT` is the total memory the container is using and the total amount of memory, it is allowed to use.

```
# docker stats a3f78cb32a8e
CONTAINER ID   NAME              CPU %  MEM USAGE / LIMIT   MEM %   NET I/O BLOCK    I/O          PIDS
a3f78cb32a8e    hello-world    0.00%   2.137MiB / 3.605GiB  0.06%     0B / 0B        9.95MB / 0B   0
```

Runtime Statistics and Profiling

The Go language provides CPU and memory profiling capabilities. With the profiling tools provided by Go, one can identify and correct the specific bottlenecks. You can make your app run faster and with less memory.

The `pprof` package writes runtime profiling data in the format expected by the `pprof` visualization tool. There are many commands available from the `pprof` command line. Commonly used commands include `top`.

For details about profiling, see the “Go Language Runtime Statistics and Profiling” section of *TIBCO Flogo® Enterprise User Guide*.

Samples

When creating apps in TIBCO Flogo® Enterprise, you can import and customize any of the predefined samples provided in the tci-flogo GitHub repository. These samples demonstrate how to develop, test, and deploy a Flogo app using various out-of-the-box capabilities. In the GitHub repository, the samples are organized by category and each sample folder contains a readme. Follow the instructions in the readme to import the sample to your local workspace and use it. The following samples are currently available:

Flow Design Concepts	API Development
Includes Hello World, Branching, Error Handling, Loops, Subflows, and Shared Data samples	Includes REST, graphQL, and gRPC samples
Array Mapping and Filtering	Connectors
Includes array.forEach, json.path, and JavaScript Activity samples	Includes Flogo connector samples for CRM, DB Connectors, Messaging, and more

Serverless

Includes sample for deploying a Flogo app as an Azure function

TIBCO Documentation and Support Services

For information about this product, you can read the documentation, contact TIBCO Support, and join TIBCO Community.

How to Access TIBCO Documentation

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

The Product Documentation website is updated frequently and is more current than any other documentation included with the product.

Product-Specific Documentation

The documentation for TIBCO Flogo® Enterprise is available on the TIBCO Flogo® Enterprise Product Documentation page.

How to Contact Support for TIBCO Products

You can contact the Support team in the following ways:

- To access the Support Knowledge Base and getting personalized content about products you are interested in, visit our product Support website.
- To create a Support case, you must have a valid maintenance or support contract with a Cloud Software Group entity. You also need a username and password to log in to the product Support website. If you do not have a username, 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 TIBCO Community.

Legal and Third-Party Notices

Cloud SG 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. See the “readme” file for the availability of a specific version of Cloud SG software 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. CLOUD SG MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S), THE PROGRAM(S), AND/OR THE SERVICES DESCRIBED IN THIS DOCUMENT AT ANY TIME WITHOUT NOTICE.

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 "README" FILES.

This and other products of Cloud SG may be covered by registered patents. For details, please refer to the Virtual Patent Marking document located at https://www.cloud.com/legal.