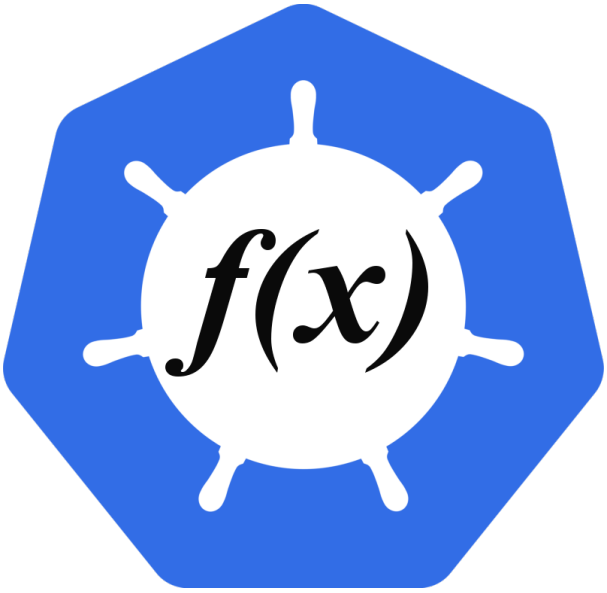
Funktion

Funktion

Chapter 1. Introduction

`Funktion` is an open source event driven lambda style programming model designed for Kubernetes.



Funktion supports over 200 of different event sources and connectors including most network protocols, transports, databases, messaging systems, social networks, cloud services and SaaS offerings.

In a sense funktion is a serverless approach to event driven microservices as you focus on just writing simple *functions* in whatever programming language you prefer, then `funktion` and Kubernetes takes care of the rest. Its not that there's no servers; its more that you as the funktion developer don't have to worry about managing them!

Chapter 2. Installing Funktion

To use `funktion` you will need a kubernetes or openshift cluster.

If you are on your laptop a quick way to get a kubernetes cluster is by installing and starting minikube and then installing kubectl and putting it on your `PATH` environment variable.

To test your kubernetes cluster type the following commands which should succeed without error:

```
kubectl get node
kubectl get pod
```

2.1. Install the funktion binary

You will also need to download the funktion binary for your platform and add it to your `PATH` environment variable.

You can test its installed by typing the following in a command shell

```
funktion version
```

The funktion binary is self updating so you can upgrade your binary to newer versions if they are available via the following command:

```
funktion update
```

2.2. Installing the Funktion Platform

There are a number of microservices required to run funktion:

- funktion operator: manages Deployments for functions and flows
- exposecontroller: exposes `Services` (functions or flows) over node ports, ingress, public cloud load balancers or openshift routes
- configmapcontroller: performs rolling updates of a `Deployment` when its associated `ConfigMap` changes

To install these microservices type:

```
funktion install platform
```

You will then get 3 `Deployment` resources created in the `funktion-system` namespace. You can view the pods created via:

```
kubectl get pod -n funktion-system
```

You can change the namespace they are installed into via the `--namespace` argument:

```
kubectl create namespace cheese
funktion install platform --namespace cheese
```

2.2.1. Using Funktion with the Fabric8 Developer Platform

If you are using the fabric8 developer platform then the exposecontroller and configmapcontroller microservices will already be installed. So you don't need to install them again.

So to install the funktion operator just type:

```
funktion install operator
```

The funktion operator `Deployment` will created in the `funktion-system` namespace. You can change the namespace they are installed into via the `--namespace` argument.

2.3. Setting up your namespace

Once you have the platform installed you need to install the runtimes and connectors in the namespace you are going to use funktion.

For example to install the default runtimes and some connectors type:

```
funktion install runtime
funktion install connector timer twitter
```

That will install the default runtimes (e.g. nodejs) along with the `timer` and `twitter` connectors so that you can use them inside flows.

To see a list of all the connectors available type:

```
funktion install connector --list
```

To install all the connectors type:

```
funktion install connector --all
```

Note that installing a connector just creates a kubernetes `ConfigMap` resource; no containers are created until you use the connector in a flow.

Chapter 3. Getting Started

To make it easier to see what kubernetes resources are being created as you create functions and flows lets use a separate namespace called `funky`.

```
kubectl create namespace funky
kubectl config set-context `kubectl config current-context` --namespace=funky
```

Now we'll install the runtimes and a couple of connectors into the `funky` namespace

```
funktion install runtime
funktion install connector http4 timer twitter
```

3.1. Create a function

You can create a function from an existing source file:

```
funktion create fn -f example/hello.js
```

Or you can specify the source code on the command line:

```
funktion create fn -n hello -s 'module.exports = function(context, callback) {
callback(200, "Hello, world!\n"); }'
```

Either of these will create a new function. You can view it via

```
funktion get fn
```

If you wish to keep editing the source code of the function in your editor and have funktion automatically update the running function use the `-w` argument to watch the source file(s):

```
funktion create fn -f example/hello.js -w
```

If you have a folder with multiple function source files inside you can pass the directory name or a wildcard pattern:

```
funktion create fn -f example -w
```

To be able to find the URL of the running function type:

```
funktion url fn hello
```

which will output the URL to access your function. Or to open it in a browser:

```
funktion url fn hello -o
```

3.2. Create a flow

```
funktion create flow timer://bar?period=5000 http://hello/
```

You should now have created a flow. You can view the flow via:

```
funktion get flow
```

To view the output of the flow you can use the following:

```
funktion logs flow timer-bar1
```

You should eventually see the output of the timer events triggering your `hello` function.

To delete the flow:

```
funktion delete flow timer-bar1
```

3.2.1. Use an existing HTTP endpoint

Flows can work with any endpoints whether they are defined via a `function` or not.

e.g. this flow will use an existing endpoint on the internet

```
funktion create flow timer://bar?period=5000 http://ip.jsontest.com/
```

3.3. A more complex example

To see a more real world style example check out the blog splitting and counting example using functions and flows

Chapter 4. Using the CLI

You can get help on the available commands via:

```
funktion
```

4.1. Browsing resources

To list all the resources of different kind via:

```
funktion get connector
funktion get flow
funktion get fn
funktion get runtime
```

4.2. Deleting resources

You can delete a Connector or flow via:

```
funktion delete connector foo
funktion delete flow bar
funktion delete fn whatnot
funktion delete runtime nodejs
```

Or to remove all the functions, flows or connectors use `--all`

```
funktion delete flow --all
funktion delete connector --all
```

4.3. Installing Runtimes and Connectors

To install the default function runtimes and connectors into your namespace type the following:

```
funktion install runtime
funktion install connector --all
```

There's over 200 connectors provided out of the box. If you only want to install a number of them you can specify their names as parameters

```
funktion install amqp kafka timer twitter
```

To just get a feel for what connectors are available without installing them try:

```
funktion install connector --list
```

or for short:

```
funktion install conn -l
```

4.4. Configuring Connectors

Various connectors have different configuration properties. For example the `twitter` connector has a number of properties to configure like the secret and token.

So to configure a connector you can type:

```
funktion edit connector twitter
```

You will then be prompted to enter new values; you can just hit `[ENTER]` to avoid changing a property.

To see a list of all the properties you can type

```
funktion edit connector twitter -l
```

Then you can pass in specific properties directly via the non-interactive version of the edit command:

```
funktion edit connector twitter accessToken=mytoken accessTokenSecret=mysecret
consumerKey=myconsumerkey consumerSecret=myconsumerSecert
```

4.5. Creating flows

To create a new flow for a connector try the following:

```
funktion create flow timer://bar?period=5000 http://foo/
```

This will generate a new `flow` which will result in a new `Deployment` being created and one or more Pods should spin up.

Note that the first time you try out a new Connector kind it may take a few moments to download the docker image for this connector - particularly the first time you use a connector.

Once a pod has started for the `Deployment` you can then view the logs of a flow via

```
funktion logs flow timer-bar1
```

4.6. Using kubectl directly

You can also create a flow using `kubectl` directly if you prefer:

```
kubectl apply -f https://github.com/funktionio/funktion/blob/master/examples/flow1.yml
```

You can view all the Connectors and flows via:

```
kubectl get cm
```

Or delete them via

```
kubectl delete cm nameOfConnectorOrflow
```

4.7. Running the Operator

You can run the funktion operator from the command line if you prefer:

```
funktion operate
```

Though ideally you'd install the funktion operator by Installing the Funktion Platform

4.8. Updating the binary

The funktion binary is self updating so you can upgrade your binary to newer versions if they are available via the following command:

```
funktion update
```

Chapter 5. Using funktion on the JVM

Funktion is designed so that it can bind any events to any HTTP endpoint or any function source using a scripting language like nodejs, python or ruby. But you can also embed the funktion mechanism inside a JVM process.

To do that you:

- write a simple function in any programming language like this.
- create a funktion.yml file and associate your function with an event trigger endpoint URL such as a HTTP URL or email address to listen on, a message queue name or database table etc.
- build and deploy the Java project in the usual way, such as via Jenkins CI / CD pipeline and your funktion will be deployed to your kubernetes cluster!

5.1. Examples

Check out the following example projects which use a JVM and implement the functions in different JVM based languages:

- funktion-java-example is an example using a Java funktion triggered by HTTP
- funktion-groovy-example is an example using a Groovy funktion triggered by HTTP
- funktion-kotlin-example is an example using a Kotlin funktion triggered by HTTP

5.2. Getting started with Funktion and the JVM

You can just fork one of the above examples and use command line tools to build and deploy it to a Kubernetes or OpenShift cluster.

However to make it easier to create, build, test, stage, approve, release, manage and iterate on your funktion code from inside your browser we recommend you use the Fabric8 Microservices Platform with its baked in Continuous Delivery based on Jenkins Pipelines together with integrated Developer Console, Management (centralised logging, metrics, alerts), ChatOps and Chaos Monkey.

When using the Fabric8 Microservices Platform you can create a new funktion in a few clicks from the `Create Application` button; then the platform takes care of building, testing, staging and approving your releases, rolling upgrades, management and monitoring; you just use your browser via the Developer Console to create, edit or test your code while funktion, Jenkins and Kubernetes take care of building, packaging, deploying, testing and releasing your project.

5.2.1. Using the Fabric8 Microservices Platform

First you will need to install the fabric8 microservices platform on a cluster of Kubernetes (1.2 or later) or OpenShift (3.2 or later).

- follow one of the fabric8 getting started guides to get the fabric8 microservices platform up and running on a Kubernetes or OpenShift cluster

- open the Developer Console
- select your `Team Dashboard` page

5.2.2. Create and use your funktion

- from inside your `Team Dashboard` page click `Create Application` button then you will be presented with a number of different kinds of microservice to create
- select the `Funktion` icon and type in the name of your microservice and hit `Next`

- select the kind of funktion you wish to create (Java, Groovy, Kotlin, NodeJS etc) then hit `Next`
- you will now be prompted to choose one of the default CD Pipelines to use. For your first funktion we recommend `CanaryReleaseAndStage`
- selecting `Copy pipeline to project` is kinda handy if you want to edit your `Jenkinsfile` from your source code later on

- click `Next` then your app should be built and deployed. Please be patient first time you build a funktion as its going to be downloading a few docker images to do the build and runtime. You're second build should be much faster!
- once the build is complete you should see on the `App Dashboard` page the build pipeline run, the running pods for your funktion in each environment for your CD Pipeline and a link so you can easily navigate to the environment or ReplicaSet/ReplicationController/Pods in kubernetes
- in the screenshot below you can see we're running version `1.0.1` of the app `groovyfunktion` which currently has `1` running pod (those are all clickable links to view the ReplicationController or pods)
- for HTTP based funktions you can invoke the funktion via the open icon in the `Staging` environment (the icon to the right of the green `1` button next to `groovyfunction-1: 1.0.1`)

5.2.3. How it works

When you implement your **Funktion** using a JVM based language like Java, Groovy, Kotlin or Scala then your function is packaged up into a Spring Boot application using Apache Camel to implement the trigger via the various endpoint URLs.

We've focussed `funktion` on being some simple declarative metadata to describe triggers via URLs and a simple programming model which is the only thing funktion developers should focus on; leaving the implementation free to use different approaches for optimal resource usage.

The creation of the docker images and generation of the kubernetes manifests is all done by the fabric8-maven-plugin which can work with pure docker on Kubernetes or reuse OpenShift's binary source to image builds. Usually this is hidden from you if you are using the Continuous Delivery in the fabric8 microservices platform; but if you want to play with funktion purely from the command line, you'll need to install Java and install Apache Maven.

Underneath the covers a Kubernetes Deployment is automatically created for your Funktion (or on OpenShift a DeploymentConfig is used) which takes care of scaling your funktion and performing rolling updates as you edit your code.

Chapter 6. How it works

The `funktion operator` watches for `Flow` and `Function` resources.

When a new `function` is created then the operator will spin up a matching `Deployment` for running the `function source code` along with a `Service` to expose the service as a HTTP or HTTPS endpoint.

When a new `flow` is created then this operator will spin up a matching `Deployment` which consumes from some `Connector` and typically invokes a function using HTTP.

The following kubernetes resources are used:

6.1. Kubernetes Resources

A `function` is modelled as a Kubernetes `ConfigMap` with the label `kind.funktion.fabric8.io: "Function"` which contains the source code of the function inside the `Data['source']` entry.

A `flow` is modelled as a Kubernetes `ConfigMap` with the label `kind.funktion.fabric8.io: "Flow"`. A ConfigMap is used so that the entries inside the `ConfigMap` can be mounted as files inside the `Deployment`. For example this will typically involve storing the `funktion.yml` file or maybe a Spring Boot `application.properties` file inside the `ConfigMap` like this example flow

A `Connector` is generated for every Camel Component and each connector has an associated `ConfigMap` resource like this example which uses the label `kind.funktion.fabric8.io: "Connector"`. The `Connector` stores the `Deployment` metadata, the `schema.yml` for editing the connectors endpoint URL and the `documentation.adoc` documentation for using the Connector.

So a `Connector` can have `0..N flows` associated with it. For those who know Apache Camel this is like the relationship between a `Component` having `0..N Endpoints`.

For example we could have a Connector called `kafka` which knows how to produce and consume messages on Apache Kafka with the Connector containing the metadata of how to create a consumer, how to configure the kafka endpoint and the documetnation. Then a flow could be created for `kafka://cheese` to subscribe on the `cheese` topic and post messages to `http://foo/`.

Typically a number of `Connector` resources are shipped as a package; such as inside the Red Hat iPaaS or as an app inside fabric8. Though a `Connector` can be created as part of the CD Pipeline by an expert Java developer who takes a Camel component and customizes it for use by `Funktion` or the `iPaaS`.

The collection of `Connector` resources installed in a kubernetes namespace creates the `integration palette` thats seen by users in tools like CLI or web UIs.

Then a `flow` can be created at any time by users from a `Connector` with a custom configuration (e.g. choosing a particular queue or topic in a messaging system or a particular table in a database or folder in a file system).

6.2. Debugging

If you ever need to you can debug any `flow` as each flow matches a `Deployment` of one or more pods. So you can just debug that pod which typically is a regular Spring Boot and camel application.

Otherwise you can debug the pod thats exposing an HTTP endpoint using whatever the native debugger is; e.g. using Java or NodeJS or whatever.

6.3. Terminology

This section defines all the terms used in the `funktion` project

6.3.1. Function

A `function` is some source code to implement a function in some programming language like JavaScript, python or ruby.

6.3.2. Runtime

A `runtime` represents the kubernetes `Deployment` metadata required to take a function source in some programming language and implement it as one or more pods.

The `funktion operator` then detects a new `function` resource being created or updated and creates the associated `runtime` deployment

6.3.3. Connector

A `connector` represents a way to connect to some event source, including most network protocols, transports, databases, messaging systems, social networks, cloud services and SaaS offerings. Funktion supports over 200 event sources.

At the implementation level a `Connector` represents the kubernetes `Deployment` metadata required to take the `Flow` and implement it as one or more kubernetes `pods`.

6.3.4. Flow

A `flow` is a sequence of `steps` such as consuming events from an `endpoint` or invoking a `function`.

For example here is a sample flow in YAML format.

```
flows:
  - steps:
    - kind: endpoint
      uri: timer://foo?fixedRate=true&period=5000
    - kind: endpoint
      uri: http://myendpoint/
```

Note that a Flow resource can contain multiple sequential flows. Each flow object in the YAML is a

sequence of steps.

Creating a `flow` results in the `funktion operator` creating an associated `Deployment` which implements the flows.

6.3.5. Funktion Operator

The `funktion operator` is a runnning `pod` in kubernetes which monitors for all the funktion resources like `function`, `runtime`, `connector` and `flow` and creates, updates or deletes the associated kubernetes `deployments` and `services` so that as you create a `flow` or `function` the associated kubenretes resources are created.

Chapter 7. FAQ

Here are the frequently asked questions:

7.1. General questions

7.1.1. What is the license?

All of the Funktion source code is licensed under the [Apache License 2.0](https://www.apache.org/licenses/LICENSE-2.0)

7.1.2. How do I get started?

Please Install Funktion then follow the Getting Started Guide

7.1.3. How do I install funktion?

See the Install Guide

7.1.4. What is serverless?

The term `serverless` just means that with lambda style programming the developer just focuses on writing `functions` only - there is no need for developers to think about managing servers or even containers.

Its not that there are no servers - of course there are - its just that developers don't need to think about them at all, they are managed for you by the platform.

7.1.5. How does Funktion compare to other serverless frameworks?

There are many frameworks out there for serverless. From 30,000 feet they are all quite similar.

Here are the main differences of Funktion:

- Funktion focusses on being Kubernetes and OpenShift native rather than some generic serverless frameworks (like Open Whisk). So Funktion is designed to reuse Kubernetes abstractions like Deployments, Service, Ingress/Route, auto scaling along with mounting ConfigMaps into pods to quickly update pods when the source of functions change to provide rapid developer feedback.
- Event triggering from over 200 different connectors spanning most middleware technologies, databases, messaging systems, APIs, social media networks and cloud services via deep integration with Apache Camel.
- Funktion can trigger any functions and HTTP endpoints from other frameworks and technologies via different trigger technologies than HTTP. Quite a few serverless frameworks, particularly kubernetes specific ones, focus on just exposing a function in some language over HTTP. We see that as important; but only part of the picture.
- Funktion supports a flow language to orchestrate flows between many functions and endpoints

to make composite flows. Similar to AWS Step Functions in concept though it delegates to the endpoint technology for persistence, retries and transactions etc.

- We are working on deep integration into the openshift.io open source developer platform for kubernetes/openshift to support rich development, debugging and CI / CD capabilities along with integration into Eclipse Che. Also we're working on a rich web UI for visually triggering functions from a variety of connectors and building flows across functions such as in this video.