DZone

BROUGHT TO YOU IN PARTNERSHIP WITH fission

Fission.io

An Open Source, Kubernetes-Native Serverless Framework

WRITTEN BY SOAM VASANI
SOFTWARE ENGINEER AT PLATFORM9 SYSTEMS

CONTENTS

Introduction

WHAT IS FISSION?

Fission is an open-source serverless function framework for Kubernetes with a focus on developer productivity and high performance.

With Fission, you can write short-lived functions in any language and map them to HTTP requests (or other event triggers). Fission executes these functions on-demand, ensuring that resources are only used when necessary.

Fission operates on *just the code*: Docker and Kubernetes are abstracted away under normal operation (though you can use both to extend Fission if you want to). This allows you to quickly create services on Kubernetes without a lot of in-depth learning and setup. You won't have to manage container builds, registries, and so on.

Fission is extensible to any language; the core is written in Go, and language-specific parts are isolated *Environments* that you can install, extend, or build from scratch (more below).

Fission and Kubernetes together form an open-source stack that gives you productivity and operational advantages on any infrastructure — whether it's the public cloud or your own datacenter.

USES

You can use Fission for a variety of tasks, including DevOps automation and Slack bots. You can also create a full REST API backend out of a set of functions — we'll follow one such example later in this Refcard. Functions can also be used to react to some sort of event, such as generating a thumbnail image when a large image is uploaded into an object store.

Installation and Setup

PREREQUISITES

A KUBERNETES CLUSTER

If you already have a Kubernetes cluster, you can skip this section.

One easy way to get started with a cluster is to use a managed Kubernetes cluster from a cloud provider: Google Kubernetes Engine, Azure Kubernetes Service, Amazon's EKS, or DigitalOcean's Kubernetes service.

Another way is to use Minikube to get a single-node Kubernetes cluster on your laptop.

You will also need the Kubernetes CLI (kubectl).

INSTALLING FISSION

Fission can be installed either with the Helm installer or just using a YAML file. Full detailed instructions are online at docs.fission.io/installation.

fission
Get Started with Serverless on Kubernetes in 5 Minutes!
Install Fission now at fission.io/install

fission

Get Started with Serverless on Kubernetes in 5 Minutes!

Accelerate time to value with Kubernetes

Focus on code, not plumbing: just write your code, Fission will make it run on Kubernetes

Run your serverless functions anywhere: on-prem, in the public cloud.

Fission Workflows make it easy to build complex apps that span many functions

Install Fission now at fission.io/install

With helm, the Fission install is done with:

```
helm install --name fission --namespace fission
https://github.com/fission/fission/releases/
download/1.0.0/fission-all-1.0.0.tgz
```

You'll also need the Fission CLI, which you can get from the GitHub releases page: github.com/fission/fission/releases.

Conceptual Introduction

Fission has three main concepts: Functions, Environments, and Triggers.

FUNCTIONS

A Fission function is something that Fission executes. It's usually a module with one entry point, and that entry point is a function with a certain interface. A number of programming languages are supported for Functions.

Here's an example of a simple function in JavaScript:

```
module.exports = async function(context) {
    return {
        status: 200,
        body: "Hello, world!n"
    };
}
```

ENVIRONMENTS

Environments are the language-specific parts of Fission. An Environment contains just enough software to build and run a Fission Function.

Since Fission invokes functions through HTTP, this means the runtime of an environment is a container with an HTTP server, and usually a dynamic loader that can load a function. Some environments also contain builder containers, which take care of compilation and gathering dependencies.

You can modify any of Fission's existing environments and rebuild them, or you can also build a new environment from scratch.

The following pre-built environments are currently available for use in Fission:

ENVIRONMENT	IMAGE
NodeJS (Alpine)	`fission/node-env`
NodeJS (Debian)	`fission/node-env-debian`
Python 3	`fission/python-env`
Python 2.7	`fission/python-env-27`
Go	`fission/go-env`
Ruby	`fission/ruby-env`
Binary (for executables or scripts)	`fission/binary-env`
.NET	`fission/dotnet-env`
.NET 2.0	`fission/dotnet20-env`
Perl	`fission/perl-env`
PHP 7	`fission/php-env`

TRIGGERS

Functions are invoked on the occurrence of an event; a Trigger is what configures Fission to use that event to invoke a function. In other words, a Trigger is a binding of events to function invocations.

For example, an HTTP Trigger may bind GET requests on a certain path to the invocation of a certain function.

There are several types of triggers besides HTTP Triggers: Timer Triggers invoke functions based on time; Message Queue Triggers exist for Kafka, NATS, and Azure queues; and Kubernetes Watch Triggers invoke functions when something in your cluster changes.

OTHER CONCEPTS

These are concepts you may not need while starting out but might be useful to know in more advanced usage.

ARCHIVES

An Archive is a zip file containing source code or compiled binaries.

Archives with runnable functions in them are called *Deployment Archives*; those with source code in them are called *Source Archives*.

PACKAGES

A Package is a Fission object containing a Deployment Archive and a Source Archive. A Package also references a certain environment.

When you create a Package with a Source Archive, Fission automatically builds it using the appropriate builder environment and adds a Deployment Archive to the package.

SPECIFICATIONS

Specifications (specs for short) are simply YAML config files containing the objects we've spoken about so far — Functions, Environments, Triggers, Packages, and Archives.

Specifications exist only on the client side and are a way to instruct the Fission CLI about what objects to create or update. They also specify how to bundle up source code, binaries, etc. into Archives.

The Fission CLI features an idempotent deployment tool that works using these specifications.

How Fission Runs Your Functions

Fission runs your functions in Kubernetes pods. Let's say you've set up a function, an environment for it, and an HTTP Trigger.

When a request comes in to the router, the function is placed in a pod in one of two ways:

With the *Pool-based executor*, a pool of containers is kept "pre-warmed" on the cluster. This pool is configured in the environment — its size and CPU/memory resource usage can be specified in the Environment Spec. When using this option to execute functions, a function is loaded on-demand into a pre-warmed pod. That process takes about 50msec to a few hundred msec, depending on the size of the function. Most functions will load within 100msec. This pod is kept alive as long as there are function requests coming in. If five minutes pass with no requests, the pod is killed. That is how the resource usage of the function is closely mapped to its actual demand.

With the *New Deployment executor*, a new Kubernetes Deployment is created either at deployment time or when the first request comes in to the function. This executor is configured by setting a min and max scale. The cases with min scale == 0 are for use cases, which are not latency-sensitive. On the other hand, setting min scale >= 1 allows your function to fully eliminate provisioning time from even the cold start, giving you excellent latency benefits at the expense of keeping the pod running on the cluster, even when there's no demand for it.

The executors allow you, as a user, to decide between latency and a small idle cost trade-off. Depending on your needs, you can choose a combination that works best for your use case.

EXECUTOR TYPE	MIN SCALE	LATENCY	IDLE COST
Newdeploy	0	High	Very low, pods get cleaned up after idle time
Newdeploy	>0 L	ow M	Medium, Min Scale, number of pods are always up
Poolmgr	0	Low	Low, pool of pods are always up

How to Use Fission

In this section, we'll walk you through a simple example of setting up your Fission cluster with a Node.js environment, creating a simple Hello World function and invoking it over HTTP.

Set Up an Environment

You can create an environment on your cluster from an image for that language. The simplest way is:

```
$ fission env create --name node --image fission/node-env
```

Optionally, you can specify CPU and memory resource limits. You can also specify the number of initially pre-warmed pods, which is called the poolsize.

Also, you can specify a builder image. This image will be used to build functions — in the case of Node.js, this usually means running npm install, which gathers dependencies specified in the function's package.json.

```
$ fission env create  --version 3 --name node --image
fission/node-env --builder fission/node-builder --mincpu
40 --maxcpu 80 --minmemory 64 --maxmemory 128 --poolsize
4
```

VIEWING ENVIRONMENT INFORMATION

You can list the environments or view information of an individual environment:

```
$ fission env list
<list of environments>

$ fission env get --name node
<one environment's information>

$ kubectl get environment.fission.io -o yaml
# YAML of Fission environment object
```

Create a Function

Now that you have an environment, you're ready to create and run a function.

This simple function in Node.js will output the string "Hello, world!":

```
module.exports = async function(context) {
    return {
        status: 200,
        body: "Hello, world!n"
    };
}
```

Let's create this function on the cluster. This only registers the function with Fission; it doesn't run it yet. It associates the function with the node environment that you previously created.

```
$ fission function create --name hello --code hello.js
--env node
```

Invoke the Function

You can invoke the function through Fission's CLI, or by creating an HTTP trigger and sending an HTTP request.

To invoke it through Fission, use the command:

```
$ fission fn test --name hello
```

This should now output "Hello, world!" A lot happens behind the scenes when you do this — Fission sends the request to the router; the router calls the Fission executor to get a running pod that has this function; the executor chooses a pre-warmed pod from the pool to load the function into; the router then proxies the request to this pod; the result is routed back to the CLI and displayed in the terminal.

Also, once this runs, the pod is cached for a few minutes; if you run the same command again, you may notice it runs faster.

You can also invoke the function through an HTTP Trigger. An HTTP Trigger associates HTTP requests on a particular URL path with the function.

```
$ fission httptrigger create --function hello --url
/hello
```

If you've set up your cluster so that the router is accessible from outside it, you can now invoke the function simply using curl:

```
$ curl http://<router ip address>/hello
Hello, World!
```

View the Functions Logs

There are two ways to view function logs: using Kubernetes or Fission's log aggregation.

USING KUBERNETES

Since functions run in pods, anything that lets you view pod logs will also let you view function logs. For example, you can use kubectl logs:

First, find the pod using its labels:

```
kubectl -n fission-function get pod -l
functionName=hello
```

Then retrieve the logs of that pod:

```
kubectl -n fission-function logs <pod name>
```

While this method doesn't rely on any external log aggregation, it's only usable for a short time, because Fission deletes idle pods (and their logs get deleted with them). Because of this, Fission also aggregates logs into a database that it installs on the cluster (log aggregation is included when you install fission-all and excluded if you installed fission-core only).

USING FISSION'S LOG AGGREGATION

With Fission's log aggregation, you can view function logs simply with:

```
$ fission function logs --name hello
```

Monitor the Function

Fission exposes Prometheus metrics. This allows you track various metrics over time, for example:

- Function duration
- Function error code statistics
- Rate of function invocation

The full Fission installation also contains a Prometheus installation. The fission-core install can be monitored with a separately-installed Prometheus instance.

You can query the Prometheus console with fission metrics, such as `fission_function_calls_total` or `fission_function_duration_seconds`, etc. Prometheus functions can be used to calculate metrics derived from these queries. For example, you can graph the rate of incoming function calls using the Prometheus rate function on the `fission_function_calls_total metric`.

Maintaining Infrastructure as Code

How should you organize source code when you have lots of functions? How should you automate deployment into the cluster? What about version control? How do you test before deploying?

The answers to these questions start from a common first step: how do you *specify* an application?

DECLARATIVE SPECIFICATIONS

Instead of invoking the Fission CLI commands, you can specify your functions in a set of YAML files. This is better than scripting the fission CLI, which is meant as a user interface, not a programming interface.

You'll usually want to track these YAML files in version control along with your source code. Fission provides CLI tools for generating these specification files, validating them, and "applying" them to a Fission installation.

What does it mean to *apply* a specification? It means putting specification to effect: figuring out the things that need to be changed on the cluster and updating them to make them the same as the specification.

Applying a Fission spec goes through these steps:

- Resources (functions, triggers, etc.) that are in the specification but don't exist on the cluster are created. Local source files are packaged and uploaded.
- Resources that are both in the specs and on the cluster are compared. If they're different, the ones on the cluster are changed to match the spec.
- Resources present only on the cluster and not in the spec are destroyed. (This deletion is limited to resources that were created by a previous *apply*; this makes sure that Fission doesn't delete unrelated resources. See below for how this calculation works.)

Note that running *apply* more than once is equivalent to running it once: in other words, it's idempotent.

USAGE SUMMARY

Start using Fission's declarative application specifications in three steps:

1. Initialize a directory of specs: `fission spec init`

2. Generate some YAMLs: `fission function create --spec ...`
3. Apply them to a cluster: `fission spec apply --wait`

You can also live-reload into a development cluster with `fission spec apply --watch --`; this lets you have immediate feedback while you're coding.

You can find a detailed tutorial on using declarative specifications on the Fission docs site at docs.fission.io/usage/developer-workflow/.

Function Composition

For performance and simplicity, FaaS functions are often quite small. They are intended to have a single responsibility and no more. However, real-world applications often require composing together complex functionality out of smaller, simpler pieces.

There are many approaches to this. One could simply make functions larger, and this works up to a point. Beyond that, functions with too much in them are harder to scale and operate efficiently. Once you have multiple functions, you're faced with the decision of how to *compose* them — how to call one function from another.

An obvious way to compose functions is by invoking them over HTTP. While this generally works, the caller spends the entire duration of the inner function call waiting for a result. If the caller uses a significant amount of memory, this is wasteful. Also, this kind of composition does not handle failures well — there is no persistence and no retries.

Another approach is to use message queues. This approach makes the communication between functions more reliable. However, the programming model is not as simple. Instead of straight code in one place, you have to think of the system as a flow of events and actions that are triggered by them. You also have to manage the functions, message queue, and triggers that bind queues to functions. This can get messy (consider, for example, doing non-compatible upgrades on multiple functions).

Finally, Fission Workflows is an approach that aims to combine the advantages of message queues with the simplicity of HTTP invocations. Workflows allows you to create a DAG (directed acyclic graph) of function calls, and then orchestrates those calls. Internally, it uses a message queue — this gives you the advantages of persistent reliable messaging, with a more ergonomic programming model.

An in-depth look at Workflows is beyond the scope of this guide. You can learn more about Fission Workflows, including some examples, at its GitHub page: github.com/fission/fission-workflows/.

Sample Application — REST API Backend for a Bank

We've shown a single function and an HTTP route, but what about more complex applications involving multiple API endpoints? We've built a sample web application for a bank to demonstrate a non-trivial application.

This application contains an API endpoint and function for each of the operations a user of a bank might perform: creating an account, depositing or withdrawing from it, transfers, etc.

You can find the full application on GitHub: github.com/fission/fission-bank-sample.

References

SUPPORTED LANGUAGES

NODE.JS

- Environment command line: `fission environment create --name node --env fission/node-env --builder fission/node-builder`
- Documentation URL: docs.fission.io/languages/nodejs/
- Sample code: github.com/fission/fission/tree/master/examples/nodejs

PYTHON

- Environment command line: `fission env create --name python --image fission/python-env --builder fission/python-builder`
- Documentation URL: docs.fission.io/languages/python/
- Sample code: github.com/fission/fission/tree/master/examples/python

GO

- Environment command line: `fission environment create --name go --env fission/go-env --builder fission/go-builder`
- Documentation URL: docs.fission.io/languages/go/
- Sample code: github.com/fission/fission/tree/master/examples/go

JAVA

- Environment command line: `fission environment create --name java --env fission/jvm-env --builder fission/jvm-builder`
- Documentation URL: docs.fission.io/languages/java/
- Sample code: github.com/fission/fission/tree/master/examples/jvm/java

RUBY

- Environment command line: `fission environment create --name ruby --env fission/ruby-env --builder fission/ruby-builder`
- Sample code: github.com/fission/fission/tree/master/examples/ruby/

BINARY

- Environment command line: `fission environment create --name binary --env fission/binary-env`

- Sample code: github.com/fission/fission/tree/master/examples/binary/

C# .NET

- Environment command line: `fission environment create --name dotnet --env fission/dotnet-env`
- Sample code: github.com/fission/fission/tree/master/examples/dotnet/

PERL

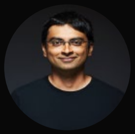
- Environment command line: `fission environment create --name php --env fission/perl-env`
- Sample code: github.com/fission/fission/tree/master/examples/perl/

PHP

- Environment command line: `fission environment create --name php --env fission/php-env`
- Sample code: github.com/fission/fission/tree/master/examples/php7/

Other Useful Resources

- Project home: fission.io
- GitHub: github.com/fission/fission
- Example repos: github.com/fission/fission-bank-sample
- Slack: slack.fission.io
- Twitter: twitter.com/fissionio



Written by **Soam Vasani,** *Software Engineer at Platform9 Systems*

Soam Vasani is a software engineer at Platform9 Systems. He leads the Fission engineering team and has also worked on Platform9's Kubernetes cluster deployment and management product. His past work includes distributed file systems and other datacenter products at VMware, and contributions to the GNU debugger and toolchain. He's interested in distributed systems, tools, frameworks, and programming languages.

Devada, Inc.
600 Park Offices Drive
Suite 150
Research Triangle Park, NC

888.678.0399 919.678.0300