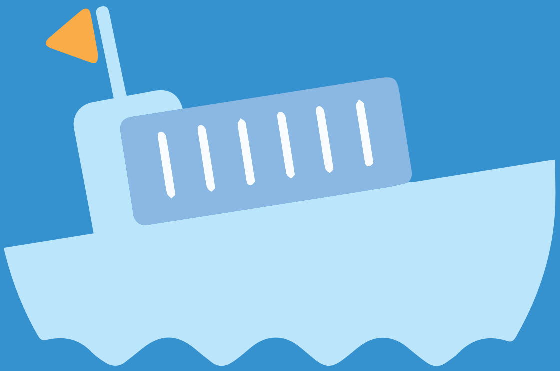
Alex Ellis

SERVERLESS FOR EVERYONE ELSE

Automate your workflow



Includes the official guide to faasd

Serverless For Everyone Else

Alex Ellis

2021

Contents

Chapter 1

Introduction

> Serverless for Everyone Else is *the* official handbook for faasd - Alex Ellis, OpenFaaS Founder

I wrote "Serverless for Everyone Else" after developing the faasd project, which is a light-weight and open-source framework for building and hosting serverless functions wherever you like.

How did I pick the title?

The title refers to the way that the faasd project makes it quick and easy way to start building and deploying functions, within the first few hours of trying out the project. You get to benefit from containers and cloud native tools, without having to manage a Kubernetes cluster, or pay significant hosting costs.

> This book is "For Everyone Else", because it offers an alternative to cloud functions, or solutions which require complex infrastructure. faasd can be deployed to a single node and requires little maintenance.

You'll learn how to use faasd through its CLI, UI and API and any functions you deploy will be able to take advantage of built-in monitoring and background processing. Behind the scenes faasd runs as a Linux systemd service and schedules its own core services, plus your functions using the production-ready containerd project.

This handbook aims to equip the reader with practical examples of what functions can do, and then goes further to become *the* reference handbook for faasd. If you're new to serverless, and don't want to get in at the deep end, this book is for you. The aim is to cover on everything you could ever need to know about faasd specifically, and then for the OpenFaaS docs and blog to supplement other areas.

The book is split into three main sections.

Getting started and building function:

- How to use the book
- Should you deploy to a VPS or Raspberry Pi?
- When should you use faasd over OpenFaaS on Kubernetes?
- What is included in the faasd stack?
- Deploying your server with bash, cloud-init or terraform
- Exploring your faasd instance through the REST API, CLI and UI
- Using a private container registry
- Finding functions in the store
- Building your first function with Node.js
- Using environment variables for configuration
- Using secrets from functions, and enabling authentication tokens
- Using Postgresql and databases from functions

Monitoring and operations:

- Understanding how function templates work and how to customise them
- Monitoring your functions with Grafana and Prometheus
- Scheduling invocations and background jobs
- Tuning timeouts, parallelism, running tasks in the background
- Adding TLS to faasd and custom domains for functions
- Getting remote access to self-hosted faasd with inlets

Advanced:

- Adding a database for storage with InfluxDB, Postgresql and Redis
- Exploring faasd resource consumption
- Troubleshooting faasd, functions and getting logs
- Upgrading and updating faasd
- Multi-arch CI/CD with GitHub Actions
- Taking things further, community resources and case-studies
- Options for scaling up with faasd, or OpenFaaS on Kubernetes.

Most of what you'll learn in the eBook applies to OpenFaaS on Kubernetes. So if you outgrow faasd, use Kubernetes already, or need clustering, then it's relatively simple to migrate. In most cases, your existing functions can be deployed without changes, and the various services we cover like Postgresql and Grafana can be sourced through helm charts or the arkade tool.

About the book

OpenFaaS and faasd are open software projects provided as no cost. This eBook contributes directly to the continued development of OpenFaaS and its related open source tooling.

Figure 1.1: Alex Ellis created OpenFaaS in 2016, but today it has its own community

All of the content of this book has been written from scratch, and the Node.js examples haven't appeared anywhere else before and you won't find the additional-services content in the documentation. This eBook aims to be the only handbook you need to start learning and deploying functions with Node.js.

In some chapters, rather than re-writing existing material and having to maintain instructions in two places, you may be linked to external resources. *So what are you paying for then?* You're paying for all the new content, that you cannot get anywhere else, plus the time it took to write the original materials and to collate them into a coherent, contextual index.

To get the most of the eBook, a video workshop is provided which gives you additional context, use-cases and a hands-on walk-through of the Node.js exercises with the creator of OpenFaaS.

Over time the book will be improved and updated, and you're welcome to make suggestions on what you would like to see next. We have a weekly Office Hours call, and you can find my email on my GitHub profile.

OpenFaaS is a mature project with many components, it takes a great deal of time to maintain what we have, and to develop new features. You can support OpenFaaS on GitHub or sponsor me directly.

Changelog

- Add Postgresql to your installation for storing data with a full CRUD example with a schema
- New screenshots and examples of adding external services into your faasd cluster
- Brand new rev3 PDF created with pandoc
- Redis service add-on example added
- Sub-chapter breaks removed to make the book more dense
- 50m video workshop for Learner and Team player tiers
- Rev 6 - detailed instructions for CI/CD with GitHub Actions
- Rev 6 - check system resources and consumption on your faasd instance
- Rev 6 - ePub format added, along with PDF
- Rev 7 - fix for npm install pq (now pg)
- Rev 7 - unit testing functions with mocha
- Rev 7 - update accessing core services
- Rev 8 - NATS connector example added for invoking functions from NATS topics
- Rev 9 - updated examples for enabling authentication for Redis
- Rev 1.0 - line-wrapping of code examples for ePub version
- Rev 1.1 - removed OpenFaaS Slack link, added community office hours and updated inlets commands
- Rev 1.2 - updates for GitHub Actions tokens for pushing images, and dynamic tags for images
- Rev 1.3 - added instructions for multiple namespace support plus custom domain instructions for Grafana
- Rev 1.4 - publishing a clarification on community resources
- Rev 1.5 - added examples and images for PromQL queries, updated cron examples, nats-connector and Node template to `node16`, Caddy is now downloaded with arkade. Updated and refreshed Grafana dashboard with provider metrics and proper legends for function name / status code etc.
- Rev 1.6 - Fixed a few typos and introduced new troubleshooting section on functions scaled down after reboots.

Chapter 2

Deploy your server

In this chapter you will learn what faasd is, and when to use it instead of OpenFaaS on Kubernetes. You'll decide whether to self-host, use a Raspberry Pi, or whether to use a cloud host. You'll learn the options for installation faasd and then the pace changes in the following chapter where we start kicking the tires with the UI and creating functions.

Why do you need a server?

Your faasd journey begins with a server. This server will run a stack of core services that make up OpenFaaS, but also gives you the ability to add stateful containers like Postgresql and Minio for storage. Your functions are run in their own separate stack, and will be packaged in container images.

It's easier to run faasd on a remote host than on your own computer because it requires a Linux Operating System and because it's supposed to be treated as an appliance or server that can be replaced at any time. For the same reason, you should not build your functions on the faasd server, but on your own computer or through a CI pipeline.

When should you use faasd over OpenFaaS on Kubernetes?

- When you just need a few functions or microservices, without the cost of a cluster
- When you don't have the bandwidth to learn or manage Kubernetes
- You have a cost sensitive project - run faasd on a 5-10 USD VPS or on your Raspberry Pi
- To deploy embedded apps in IoT and edge use-cases
- To shrink-wrap applications for use with a customer or client

OpenFaaS can be run with faasd or deployed to Kubernetes. The core services of OpenFaaS are the same, however with faasd, you cannot scale beyond one replica and cannot build clusters. Fortunately, if you do start to push the limits of this configuration, you can move to Kubernetes and bring your existing functions and workloads with you.

The PLONK Stack and OpenFaaS Core Services

An OpenFaaS installation can be thought of as The PLONK Stack.

- Prometheus - a time-series database used to capture metrics
- Linux - The base Operating System to provide containers

- OpenFaaS - management and auto-scaling of compute - PaaS/FaaS, a developer-friendly abstraction on top of Kubernetes. Each function or microservice is built as an immutable Docker container or OCI-format image.
- NATS - asynchronous message bus / queue for deferred execution
- Kubernetes - declarative, extensible, scale-out, self-healing clustering

With faasd, we are swapping out "Kubernetes" for containerd, which is a low-level container runtime. As part of the swap, we get a leaner system that's easier to manage, but we trade that for clustering capabilities.

This is what the OpenFaaS stack looks like in addition to Prometheus and NATS:

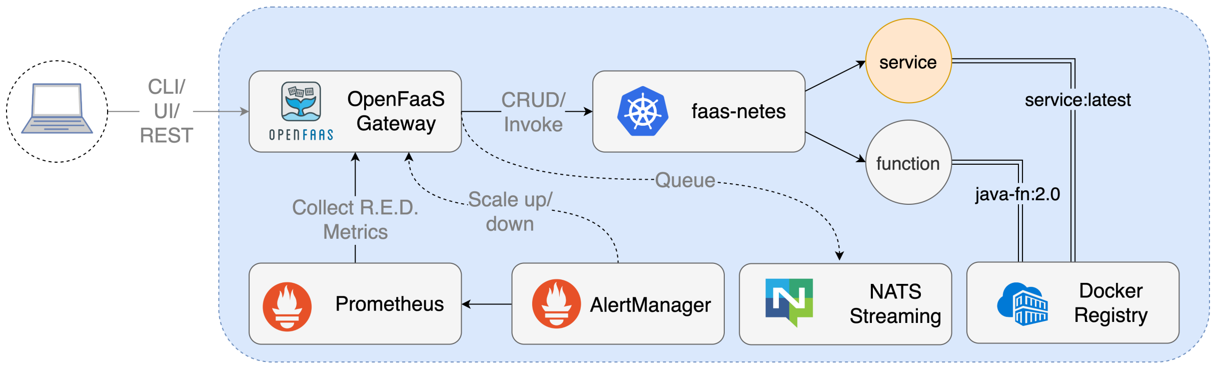


Figure 2.1: OpenFaaS Stack

> Pictured: an invocation via the OpenFaaS gateway using the UI or CLI, is proxied to the function container, or enqueued to NATS first, for execution by the queue-worker.

So whilst faasd does not scale to more than one replica of a function, it can scale to zero and back again. Using containerd's "pause" functionality, makes for a very fast scale from zero experience without a noticeable cold-start.

- OpenFaaS gateway - a REST API and UI
- OpenFaaS queue-worker - runs queued invocations for functions
- OpenFaaS basic-auth module - Basic Authentication for the REST API prevents unauthorized access. Can be swapped for Single-Sign On

Each component in faasd is defined in a `docker-compose.yaml` file, which we will explore in a later chapter. By editing this file you can update the version of each component and fine-tune settings like timeouts and how many asynchronous invocations can be processed at once.

Self-hosted or cloud?

There are three options I would recommend for your faasd server.

- VPS or VM - hosted on your preferred cloud provider, you'll have a public IP address and can serve traffic to users
- Intel NUC or your workstation using Multipass - hosted on your home network
- Raspberry Pi 3 or 4 - hosted on your home network

> Have you heard of multipass? Whenever we work on enhancements for faasd, we use multipass, which is an easy way to get a Linux VM on Windows, MacOS and Linux.

The main considerations you have are the costs and availability. Cloud infrastructure is likely to be more stable than your home Internet connection, with a higher amount of bandwidth available. That said, if your connection is reliable, then a Raspberry Pi 3 is perfectly capable of serving traffic to users, background processing or receiving webhooks.

Perhaps you are thinking of deploying machine-learning models? In that case, you may save much more money by running those on a powerful GPU inside your home, instead of on an expensive AWS EC2 instance.

> When hosting faasd on your home network, if you want to receive webhooks or serve traffic to users, you'll need to use a tool like inlets (aka "inlets Pro") to create a tunnel. inlets is secure by default, has many guides, and comes at a discount for a personal license. In a later chapter, you'll learn how to use it for remote access.

If you do decide to use a Raspberry Pi, then your functions will need to be built using multi-arch OpenFaaS templates or Dockerfiles. Whilst faasd works on 64-bit and 32-bit ARM Operating Systems, you should use the 32-bit Operating System from the Raspberry Pi Foundation called "Raspberry Pi OS Lite."

The easiest way to start to learn is with a cloud VM or Multipass. Then when you have a clear understanding of how everything works, you can move your workloads to a Raspberry Pi.

The good news is that it's trivial to move from self-hosted to a cloud provider and visa-versa.

Install faasd with Bash

faasd can be installed with bash. Ubuntu is the preferred OS, but CentOS is also available as an option.

If you don't have a cloud VM, you can create one with multipass.

The default memory, CPU and disk for multipass is low, but works well with faasd, you can increase it by using command-line flags. By adding more disk capacity, you can store more container images and functions. In the advanced section of the eBook there is a script you can use to clean-up old image which are no longer in use.

```
multipass launch --name faasd \
  --disk 16G \
  --cpus 2 \
  --mem 4G
```

Wherever your host is, whether it's a VM, Raspberry Pi, or cloud instance, you can then install faasd using bash:

```
git clone https://github.com/openfaas/faasd --depth=1
cd faasd
./hack/install.sh
```

> Note: this method does not enable TLS, but is the easiest to use for testing. The Terraform installation includes automatic TLS, or you can add it using the instructions in a later chapter.

Install faasd with cloud-init

cloud-init can be used with cloud hosting, and some on-premises environments like OpenStack. By providing a YAML file, the cloud can configure all the packages and software required before the first user logs in.

```
curl -SLs -o cloud-config.txt \
  https://github.com/openfaas/faasd/blob/master/cloud-config.txt
```

Edit `ssh_authorized_keys` and replace with your own key.

You will find this at `~/.ssh/id_rsa.pub`, if you don't see a key then generate one with `ssh-keygen`.

If you run into issues, then the logs for cloud-init are usually found at:

```
/var/log/cloud-init.log
```

You should also check the troubleshooting chapter if you run into problems.

Some cloud CLIs come with a flag or option for passing this this file, with others like DigitalOcean, you can paste it into the UI when creating a new host.

Here's an example from the multipass and faasd bootstrap guide:

```
multipass launch \
  --name faasd \
  --cloud-init cloud-config.txt
```

See the previous section if you need to add more RAM, CPU or disk capacity.

Install faasd with TLS via Terraform

Terraform is Hashicorp's answer to Infrastructure as Code (IaC). By writing automation in HashiCorp Configuration Language (HCL), you can tell the `terraform` CLI how to create and configure a server.

Terraform has to be written for every cloud separately, but we provide an example for DigitalOcean which you can adapt for your own purposes.

> Note: my goal is not to teach you terraform, or how to deploy faasd with terraform, the blog post and documentation is better suited to that.

See also:

- Tutorial: Bring a lightweight Serverless experience to DigitalOcean with Terraform and faasd
- Terraform source-code on DigitalOcean with TLS

Chapter 3

Exploring your faasd instance

In this chapter you'll learn how to find the password for the OpenFaaS gateway, how to connect using the OpenFaaS CLI, and how to use the UI before going on to deploy a sample function from the function store.

Log into faasd with the CLI and UI

Install the faas-cli on your own computer. Whenever we run `faas-cli`, it will be remotely, using the OpenFaaS REST API exposed by our faasd instance.

Now log into your faasd host with `ssh` and find the password for the admin user:

```
sudo cat /var/lib/faasd/secrets/basic-auth-password ;echo

f16a68c8e731214ce388c07af314e6266b70b964
```

Save the result as faasd.txt

Next configure your `OPENFAAS_URL`, if you installed without TLS, or are running faasd on your local network, use "http://"and port 8080, instead of "https://".

```
export OPENFAAS_URL=https://faasd.example.com

export OPENFAAS_URL=http://local-ip:8080
```

You may want to save the above line to your bash profile to make it permanent.

Next log into your faasd instance:

```
cat faasd.txt | faas-cli login --username admin --password-stdin
```

Open the `OPENFAAS_URL` in a browser and enter the credentials from above. The username is `admin`.

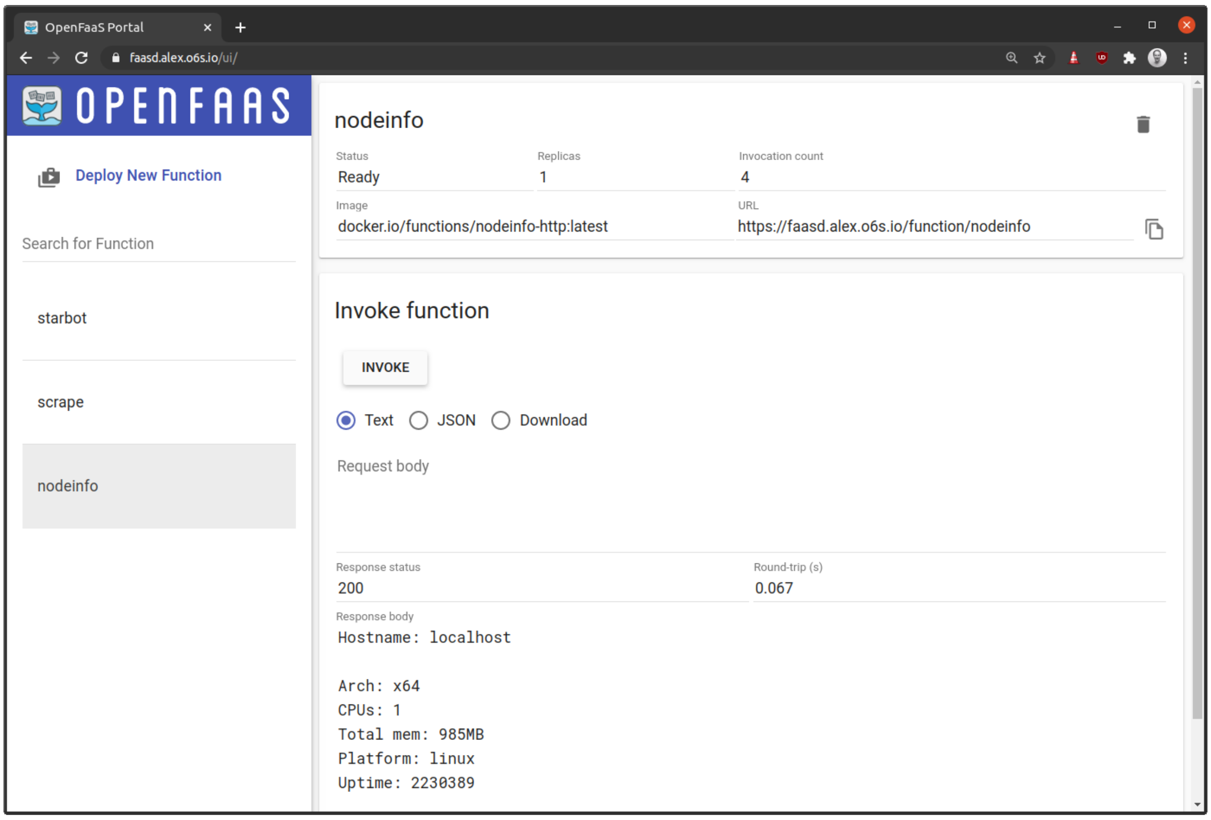
```
echo $OPENFAAS_URL
```

Invoking the nodeinfo function after deploying it from the Function Store

In the next section I'll show you how to deploy a pre-packaged function from the Function Store, but you can also use the UI to deploy containers you've already packaged and pushed into a container registry.

Here is a manual example with a toy function that generates ASCII cows:

You can view available functions in the store using the UI by clicking "Deploy Function":



nodeinfo

Status: Ready
Replicas: 1
Invocation count: 4
Image: docker.io/functions/nodeinfo-http:latest
URL: https://faasd.alex.o6s.io/function/nodeinfo

Invoke function

INVOKE

[✓ Text] JSON Download

Request body

Response status: 200
Round-trip (s): 0.067

Response body

```
Hostname: localhost

Arch: x64
CPUs: 1
Total mem: 985MB
Platform: linux
Uptime: 2230389
```

Figure 3.1: OpenFaaS UI Portal

> Note that some functions may require large amounts of RAM, particularly the colorisebot and inception functions, and may not run on a Raspberry Pi. I'm making sure that all the examples in the book will run on either.

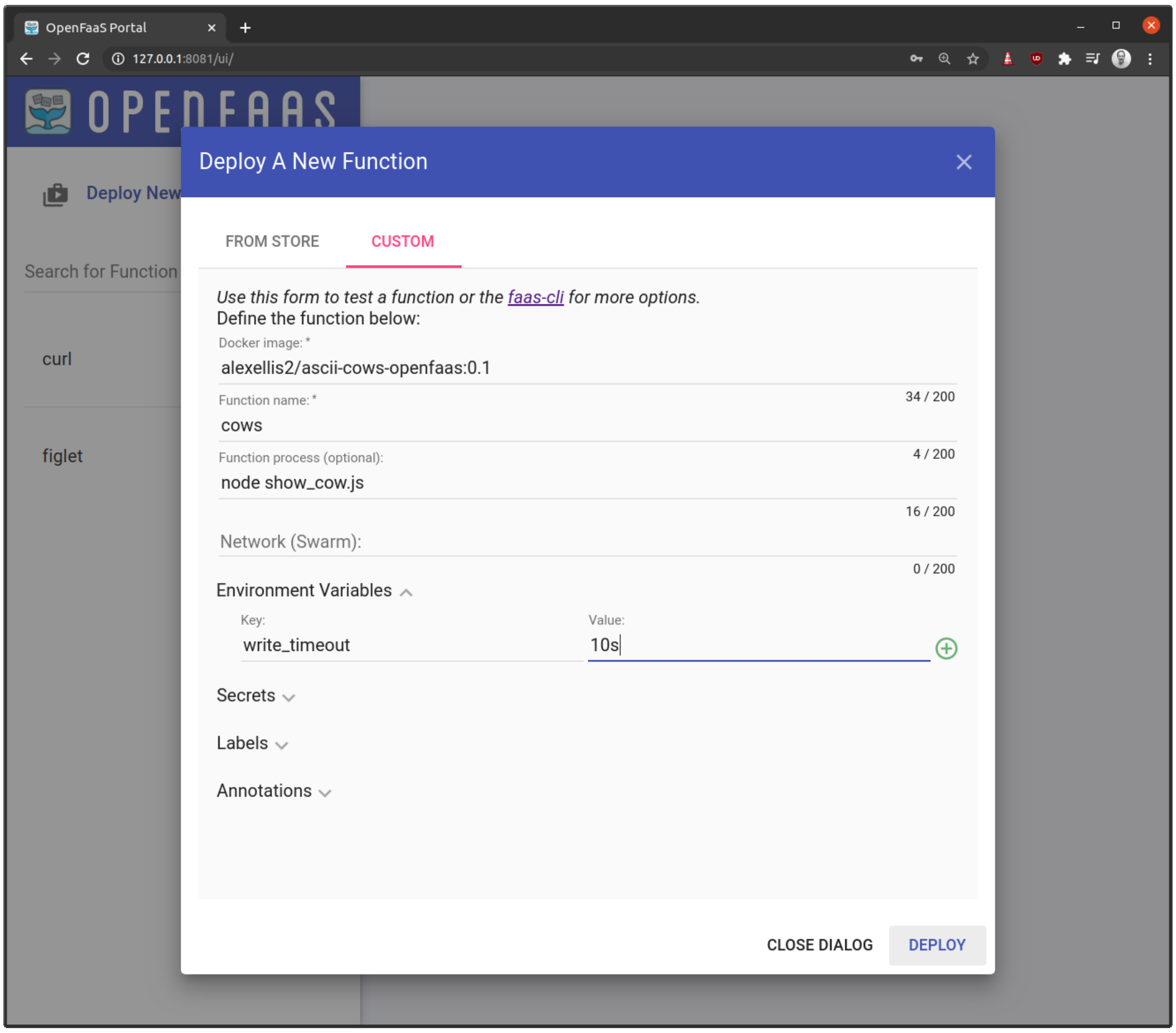
So whilst the UI is useful for learning and exploring OpenFaaS, the true value comes from the CLI and is what we will use for the rest of the eBook.

Deploy a function from the store

Let's deploy a function to your faasd instance, but first of all, let's browse the Function Store:

```
faas-cli store list

FUNCTION                       DESCRIPTION
NodeInfo                       Get info about the machine that you''r...
Figlet                         Generate ASCII logos with the figlet CLI
SSL/TLS cert info              Returns SSL/TLS certificate informati...
Colorization                   Turn black and white photos to color ...
Inception                      This is a forked version of the work ...
Have I Been Pwned              The Have I Been Pwned function lets y...
Face Detection with Pigo       Detect faces in images using the Pigo...
curl                           Use curl for network diagnostics, pas...
SentimentAnalysis              Python function provides a rating on ...
Tesseract OCR                  This function brings OCR - Optical Ch...
Dockerhub Stats                Golang function gives the count of re...
QR Code Generator - Go         QR Code generator using Go
```



Use this form to test a function or the faas-cli for more options.
Define the function below:

Docker image: * alexellis2/ascii-cows-openfaas:0.1 (34 / 200)

Function name: * cows (4 / 200)

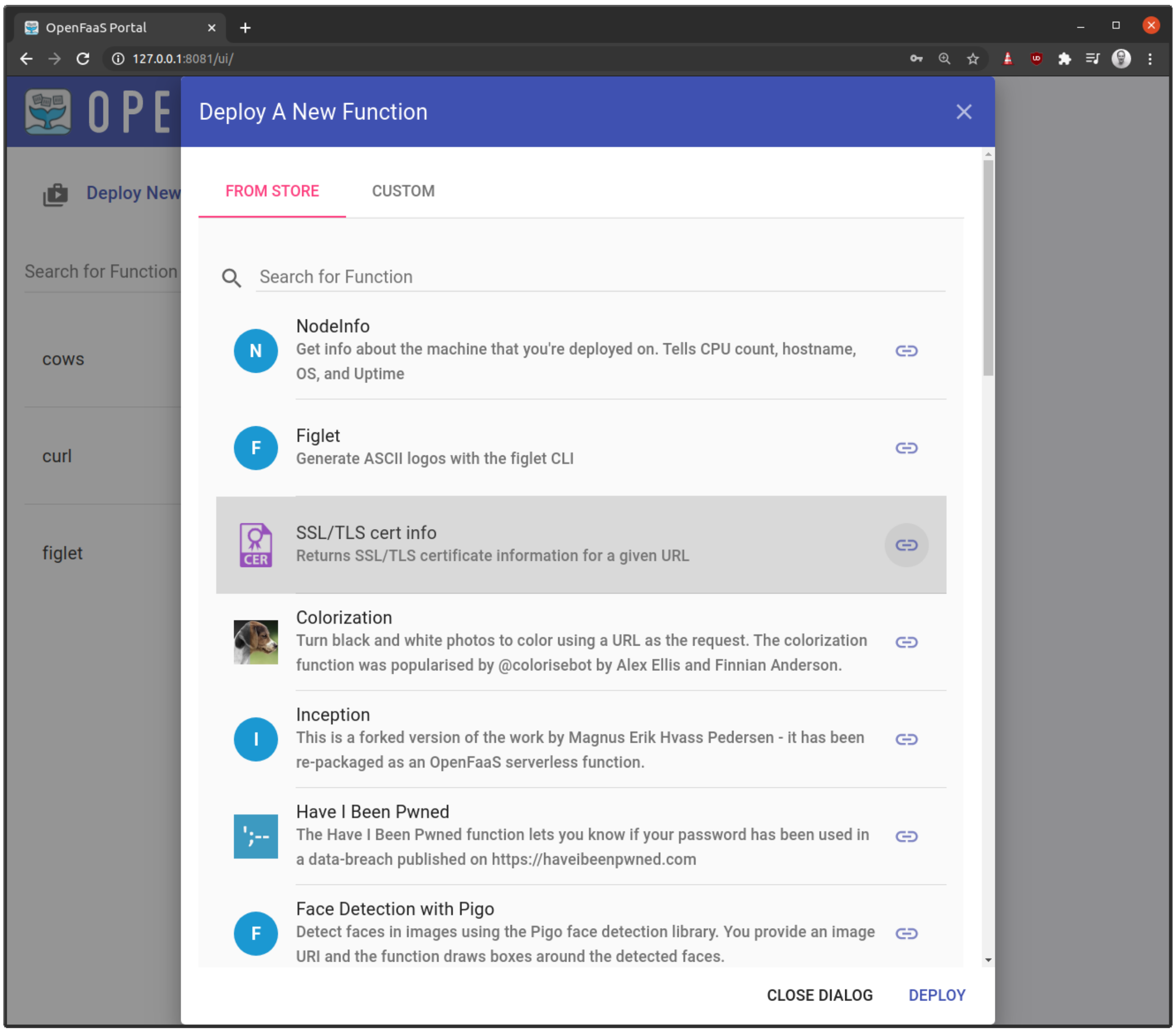
Function process (optional): node show_cow.js (16 / 200)

Network (Swarm): (0 / 200)

Environment Variables

Key: write_timeout
Value: 10s

Figure 3.2: Manual deployments



NodeInfo
Get info about the machine that you're deployed on. Tells CPU count, hostname, OS, and Uptime

Figlet
Generate ASCII logos with the figlet CLI

SSL/TLS cert info
Returns SSL/TLS certificate information for a given URL

Colorization
Turn black and white photos to color using a URL as the request. The colorization function was popularised by @colorisebot by Alex Ellis and Finnian Anderson.

Inception
This is a forked version of the work by Magnus Erik Hvass Pedersen - it has been re-packaged as an OpenFaaS serverless function.

Have I Been Pwned
The Have I Been Pwned function lets you know if your password has been used in a data-breach published on https://haveibeenpwned.com

Face Detection with Pigo
Detect faces in images using the Pigo face detection library. You provide an image URI and the function draws boxes around the detected faces.

Figure 3.3: OpenFaaS Store UI

Try something lightweight like "nodeinfo", "figlet" or "certinfo":

```
faas-cli store deploy nodeinfo
```

> Note: Raspberry Pi users should add the flag `--platform armhf` so that you get compatible container images.

Now you can find the status of the function, its invocation count and its URLs by running:

```
faas-cli describe nodeinfo

Name:                nodeinfo
Status:              Ready
Replicas:            1
Available replicas:  1
Invocations:         26
Image:
Function process:
URL:                 http://127.0.0.1:8080/function/nodeinfo
Async URL:           http://127.0.0.1:8080/async-function/nodeinfo
```

You can also invoke the function from the UI we opened earlier, or invoke it through the faas-cli:

```
echo verbose | faas-cli invoke nodeinfo

Hostname: localhost

Arch: arm
CPUs: 4
Total mem: 926MB
Platform: linux
Uptime: 5074432
```

Try out a few of the other commands, you'll find them via `faas-cli --help`

```
faas-cli list [-v]
faas-cli version
```

Delete the nodeinfo function with `faas-cli remove nodeinfo`.

If you'd like to know more about a function before you deploy it, you can run the following:

```
faas-cli store describe nodeinfo -v

Info for: NodeInfo

Name        nodeinfo
Description Get info about the machine that you''re deployed on. Tells CPU
            count, hostname, OS, and Uptime
Image       functions/nodeinfo-http:latest
Process
Repo URL    https://github.com/openfaas/faas
```

This can help you find the source code and author of a function.

All store commands support a `--url` flag, which means you can use an alternative source for the functions.

The Open Source store is generated from a JSON file on GitHub.

See also: functions.json

Can I get an API with that?

The OpenFaaS gateway comes with its own REST API which can be accessed via HTTP.

The API can be used to perform create / update / retrieve and delete (CRUD) options on functions and secrets, with the caveat that you cannot retrieve secrets through the API.

> Why can't I send source code and get a function? Functions are built into container images can be deployed through the API, many options exist for building containers, and so the OpenFaaS API itself does not support a source-to-image operation.

Install the `jq` package (`sudo apt install jq`), so that you can format the JSON responses.

Find your password on the faasd instance with:

```
sudo cat /var/lib/faasd/secrets/basic-auth-password ; echo
```

Now using your OpenFaaS password and OPENFAAS_URL, construct a command to list all functions. You can run this from your own PC or on the faasd instance.

```
export PASSWORD="MI6QKhCu2iacWwhhsKIu9oblISQ6bqpqHLrJWRwTZH6eGUvEB3w4K4kg18ZmieV"
export HOST="10.143.143.158:8080"

curl -s \
  http://admin:$PASSWORD@$HOST/system/functions | jq
```

This is what I got from nodeinfo:

```
[
    {
    "name": "nodeinfo",
    "image": "docker.io/functions/nodeinfo-http:latest",
    "invocationCount": 2,
    "replicas": 1,
    "envProcess": "",
    "availableReplicas": 1,
    "labels": {},
    "annotations": {},
    "namespace": "openfaas-fn"
  }
]
```

You'll note that if you pass an empty password, that the API gives back an unauthorized error code.

Now try querying the nodeinfo function itself by building the URL `/system/function/nodeinfo`.

```
curl -s \
  http://admin:$PASSWORD@$HOST/system/function/nodeinfo | jq
```

This is what I got from nodeinfo:

```
[
{
  "name": "nodeinfo",
  "image": "",
  "invocationCount": 2,
  "replicas": 1,
  "envProcess": "",
  "availableReplicas": 1,
  "labels": {},
  "annotations": {},
  "namespace": "openfaas-fn"
}
]
```

You can use this endpoint to check a function's image and version, or to query its labels or annotations for filtering.

Now try deploying a function through the `POST` method and the `/system/function` endpoint:

```
export IMAGE=docker.io/functions/alpine:latest-multiarch

curl -s \
  --data-binary \
  '{"image":"'"$IMAGE"'", "service": "cal", "envprocess": "cal"}' \
  http://admin:$PASSWORD@$HOST/system/functions
```

The above will run the built-in shell command to create a calendar.

Check the container was created:

```
curl -s http://admin:$PASSWORD@$HOST/system/function/cal | jq
```

Then invoke the function:

```
curl http://$HOST/function/cal

    January 2021
Su Mo Tu We Th Fr Sa
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
```

Delete the function:

```
curl -s \
  -X DELETE \
  --data-binary \
  '{"functionName": "cal"}' \
  http://admin:$PASSWORD@$HOST/system/functions
```

The deletion may take a few seconds depending on the timeout variable for invocations, this is to ensure a safe shutdown and drain of any connections.

The `faas-cli` is the primary client for the OpenFaaS gateway and comes with its own Golang SDK, which can be easier to use for automating OpenFaaS than HTTP on its own.

The API also supports POST for deploying new functions and PUT for updating existing ones. You can find the Swagger definition on GitHub: swagger.yml

Chapter 4

Building your own functions

In this chapter you will learn how to find templates, where to publish container images for your functions, how OpenFaaS templates work and how to build your own functions using Node.js.

Finding a template

OpenFaaS functions are packed in container images, which listen to HTTP on port 8080. This can be done by using a template with a HTTP server inside it, or by using the OpenFaaS watchdog, and telling it which CLI to run for each invocation, in a similar way to CGI-BIN.

You can build your own templates from scratch, or search the template store:

```
faas-cli template store list
```

Some of the templates are maintained by the OpenFaaS community team, and others are from third parties.

Find out more about a template, and where to find its source code with:

```
faas-cli template store describe node16

Name:              node16
Platform:          x86_64
Language:          NodeJS
Source:            openfaas
Description:       HTTP-based Node 12 template
Repository:        https://github.com/openfaas/templates
Official Template: true
```

Just like with the Function Store, the list of templates is also Open Source and available on GitHub.

See also: templates.json

You can learn about the templates in the OpenFaaS docs.

There are dozens of templates available, and it's trivial to make new ones or to fork an existing one to customise it. My favourite languages for functions are JavaScript, Python 3 and Golang, but for the rest of the ebook, we will be using the `node16` template.

Where should you publish container images?

Every OpenFaaS function that you'll run must first be packaged as a container image. In some cases, compatible existing container images can be run without any additional changes.

Once you build images on your machine or during CI, they must be published on a container registry, before they can be deployed to OpenFaaS.

Your images can be either public or private, that decision is up to you. The image should only contain your source code, with your secrets being created separately. So in the worst case, a public image would allow someone to view your source code.

With a hosted registry, public images tend to be free, but private images are either limited-use or on a paid plan. There are a number of hosted regisitries, including those provided by cloud vendors.

The Docker Hub contains a wide range of images for software which is maintained by Docker. Until recently, it also offered unlimited rate limits for pulling down public images. From around 2019 onwards, you'll need to pay for a plan from Docker at around 60 USD / year to make it useful.

In response to Docker's change in business model, GitHub Container Registry was launched (GHCR). GHCR integrates well with GitHub Actions, and can make a powerful combination when you're ready to deploy your code to production. It's currently free to use for public images, and private images may incur costs for storage and build minutes outside of a free allowance.

Private, or self-hosted registries come without any form of rate limits or cost, however you must bear in mind maintenance, backup, and whether high-availability is important. These also require a TLS certificate, which can make them complex to set up.

Whenever you create a function, the `image:` field in the function's YAML file tells faas-cli where to publish the container image:

For the Docker Hub, you can omit the prefix of `docker.io` so: `alexellis2/starbot:latest`

For GHCR, the prefix is `ghcr.io`, so: `ghcr.io/alexellis/starbot:latest`

Whichever registry you use, you will need to authenticate with the `docker login` command before publishing any container images.

Private registries require authentication, which we cover in the next section.

Private registries

You can configure faasd to use a self-hosted registry. This guide does not cover how to set up a registry, and we would recommend that you use a managed service like the Docker Hub or GitHub's container registry.

A valid Docker config file needs to be copied into your faasd folder, such as:

```
sudo cp ~/.docker/config.json /var/lib/faasd/.docker/config.json
```

Beware that running `docker login` on MacOS and Windows may create an empty file with your credentials stored in the system helper.

Alternatively, you can use the `faas-cli registry-login` command to generate a base64-encoded file that can be used with faasd.

```
faas-cli registry-login \
  --username <your-registry-username> \
  --password-stdin
# (the enter your password and hit return)
```

The file will be created in the `./credentials/` folder which needs to be copied as per below:

```
sudo cp ./credentials/docker.config /var/lib/faasd/.docker/config.json
```

> Note for the GitHub container registry, you should use `ghcr.io` Container Registry and not the previous generation of "Docker Package Registry".

If you'd like to set up your own private registry, see this tutorial.

How templates are built

In the next few sections we will build a function with the node16 template, but let's learn how templates work first.

Each template is stored in a GitHub repository, so you can customise them as you like. A git repository can have more than one template stored within it, and they all need to be within the `template` folder in the root. This is a convention.

Fork the OpenFaaS templates on GitHub and look at how they are put together

We have the files which are always the same, and invisible to the user:

- `/README.md` - detailed usage instructions
- `/template/` - contains the templates
- `/template/node16` - the template for node16
- `/template/node16/template.yml` - metadata for the node16 template like the welcome message and additional packages that can be installed
- `/template/node16/Dockerfile` - how the template is built
- `/template/node16/index.js` - the hidden entrypoint for the function (same for each function), it creates the HTTP server and listens on a set port, it also sets up any routers or HTTP paths that are required to serve traffic

Then within a sub-folder, we have the files that the user sees, edits and works with on a regular basis.

- `/template/node16/function/handler.js` - the handler for the function to be customised by the user
- `/template/node16/function/package.json` - the packages to be installed during the build

> Templates separate the boiler-plate repetitive code such as the Dockerfile, health-checks and starting a HTTP server.

The `template.yml` file defines the function's name and a multi-line welcome message that's printed to users when they create a new function.

```yaml
language: node16
welcome_message: |
  You have created a new function which uses Node.js 12 (TLS)
  and the OpenFaaS of-watchdog which gives greater control
  over HTTP responses.
  npm i --save can be used to add third-party packages like
  request or cheerio npm documentation: https://docs.npmjs.com/
  Unit tests are run at build time via "npm run", edit
  package.json to specify how you want to execute them.
```

The files contained at the template's root folder are not for consumption by the developer, but are overlaid at build-time:

```
index.js
Dockerfile
```

```
package.json
```

You'll see that `npm install` runs twice in the Dockerfile, once for the base package `index.js` and then once again for the user-supplied code.

The template for the user-supplied code is kept in the `function` folder:

```
handler.js
package.json
```

This technique means that the user doesn't have to know about how the HTTP server or underlying microservices framework is configured such as Express.js. They just write the "handler" that takes a request and reproduces a response.

Now that you know how templates work, you can customise whatever you like by forking the template into your own Git repository.

Your first JavaScript function

Create a function with the node16 template:

```
# Change to your own username or GHCR prefix
export OPENFAAS_PREFIX=alexellis2
faas-cli new --lang node16 starbot
```

This will produce three files:

- `starbot/handler.js` - your code
- `starbot/package.json` - packages to install
- `starbot.yml` - how the function should be built and deployed aka "stack file"

Every template is different, and you can even have multiple templates per language.

This is what our starbot.yml file looks like, we will refer to it as a stack YAML file going forward.

```
version: 1.0
provider:
  name: openfaas

functions:
  starbot:
    lang: node16
    handler: ./starbot
    image: alexellis2/starbot:latest
```

This file can be used to add non-configuration like debugging settings and confidential configuration through the use of secrets. You can also specify additional metdata like a schedule for invoking the function, or labels for automating the openfaas REST API.

See also: reference guide stack YAML files

This is what the handler looks like for a function with the `node16` template:

```
module.exports = async (event, context) => {
  const result = {
    'status': 'Received input: ' + JSON.stringify(event.body)
  }
  return context
    .status(200)
    .succeed(result)
}
```

As you know from the previous section on OpenFaaS templates, more is happening in the background, but this file is all you need to be concerned with when you write your function.

In the `event` object you can access HTTP headers:

- `event.query` - the query string
- `event.headers` - a map of headers
- `event.body` - the request body, if using a JSON "Content-Type" in your HTTP request, it will be parsed as an object

For inputting binary data such as files and images, you can set the "RAW_BODY" environment variable which turns off any attempt to parse the incoming HTTP body.

The `context` variable can be used to set HTTP headers, a status code and the response body.

- `status(int)` - set the HTTP status code
- `succeed(object)` - set the HTTP body
- `fail(object)` - set status code 500, and a body

If you're building from an Intel/AMD computer, and deploying to one, then you can run the `faas-cli up` command, which is a shortcut for `faas-cli build` followed by `faas-cli push`, then `faas-cli deploy`. However, if you're deploying to an ARM computer, or building on an ARM computer, and deploying to an Intel/AMD one, you'll need to read the next section which splits the commands into `faas-cli publish` followed by `faas-cli deploy`.

```
faas-cli up -f starbot.yml

Deployed. 200 OK.
URL: http://127.0.0.1:8080/function/starbot
```

You can now invoke your function passing in a JSON body:

```
curl --data-binary '{"url": "https://inlets.dev"}' \
  --header "Content-Type: application/json" \
  https://faasd.example.com/function/starbot

{"status":"Received input: {\"url\":\"https://inlets.dev\"}"}
```

We can see that our input was received and then serialised into a string. Let's try building upon this example by adding an npm module.

axios is commonly used to make HTTP requests, you can install it like this:

```
cd starbot
npm install --save axios
```

Then add a reference in your handler just like you would normally.

Here's what our function would look like if it counted the astronauts in space by querying a HTTP API.

The API returns the following:

```
{
  "message": "success",
  "number": NUMBER_OF_PEOPLE_IN_SPACE,
  "people": [
    {"name": NAME, "craft": SPACECRAFT_NAME},
    ...
  ]
```

Here's our `handler.js`:

```
const axios = require("axios")

module.exports = async (event, context) => {
  let res = await axios.get("http://api.open-notify.org/astros.json")

  let body = `There are currently ${res.data.number} astronauts in space.`

  return context
    .status(200)
    .headers({"Content-type": "application/json"})
    .succeed(body)
}
```

You can now invoke your function:

```
curl https://faasd.example.com/function/starbot
There are currently 7 astronauts in space.
```

Passing the `-i` argument to `curl` will show the headers returned, which also includes the duration in seconds to complete the call. We can see that the call took `0.315462`, the additional latency is probably because of the HTTP call we're making.

```
curl -i https://faasd.alex.o6s.io/function/starbot
HTTP/2 200
content-type: application/json; charset=utf-8
date: Fri, 15 Jan 2021 12:25:47 GMT
x-duration-seconds: 0.315462
content-length: 48

There are currently 7 astronauts in space.
```

A note for ARM and Raspberry Pi users

If you're using Raspberry Pi to run your faasd server, then you will need to use the `publish` command instead which uses emulation and in some templates cross-compilation to build an ARM image from your PC.

> It is important that you do not install Docker or any build tools on your faasd instance. faasd is a server to serve your functions, and should be treated as such.

The technique used for cross-compilation relies on Docker's buildx extension and buildkit project. This is usually pre-configured with Docker Desktop, and Docker CE when installed on an Ubuntu system.

The faas-cli attempts to enable Docker's experimental flag for the CLI, but you may need to run the following, if you get an error:

```
export DOCKER_CLI_EXPERIMENTAL=enabled
```

Now run this command on your laptop or workstation, not on the Raspberry Pi:

```
faas-cli publish -f starbot.yml \
  --platforms linux/arm/v7
  --reset-qemu
```

The `--reset-qemu` flag is only required the first time you run this command, or upon every fresh boot up of your system. It configures Docker to emulate different CPU types.

> If you're running a 64-bit ARM OS like Ubuntu, then use `--platforms linux/arm64` instead.

Then deploy the function:

```
faas-cli deploy -f starbot.yml
```

Platform options:

- `linux/arm/v7` - Raspberry Pi running a 32-bit OS
- `linux/arm64` - AWS Graviton Server, Raspberry Pi running a 64-bit OS like Ubuntu 22.04
- `linux/amd64` - a regular Intel/AMD machine

So, if you want to build for all of these platforms, you'd use `--platforms linux/arm/v7,linux/arm64,linux/arm64`, using a comma to separate each.

Adding configuration to your function

A common way to configure functions is to use environment variables. These can be set in your stack.yml file and then consumed at runtime.

An example would be if we wanted to enable logging, or configure the URL that is used to check the astronaut data.

Add a `query_url` and `add_people` environment variables to your `starbot.yml` file:

```
version: 1.0
provider:
  name: openfaas

functions:
  starbot:
    lang: node16
    handler: ./starbot
    image: alexellis2/starbot:latest

    environment:
      query_url: http://api.open-notify.org/astros.json
      add_people: true
```

Update your code to read the configuration values from `process.env`:

```
const axios = require("axios")

module.exports = async (event, context) => {
  console.log(process.env)
  let res = await axios.get(process.env.query_url)

  let body = `There are currently ${res.data.number} astronauts in space.`
  if(process.env.add_people) {
    body += " Including"
    res.data.people.forEach(p => {
      body += " " + p.name
    })
  }

  return context
    .status(200)
    .headers({"Content-type": "application/json"})
    .succeed(body)
}
```

Deploy the function with `faas-cli up` and invoke it again.

```
There are currently 7 astronauts in space. Including Sergey Ryzhikov Kate Rubins Sergey Kud-
Sverchkov Mike Hopkins Victor Glover Shannon Walker Soichi Noguchi
```

Now you can configure functions to use non-confidential data to change their behaviour.

Consuming secrets from functions

All functions consume secrets in the same way, by reading a file from: `/var/openfaas/secrets/NAME`

Here's a quick way to enable an API key for our astronaut finder:

```
TOKEN=$(head -c 12 /dev/urandom | shasum| cut -d' ' -f1)
echo $TOKEN > token.txt

faas-cli secret create astro-key --from-file token.txt

Creating secret: astro-key
Created: 200 OK
```

You can also in-line the creation and use `--from-literal`, if you wish.

```
faas-cli secret create astro-key --from-literal $TOKEN
```

This command creates a key on the server which you can view via `faas-cli secret list`

Add the secret name to the list of secrets in your stack YAML file:

```
functions:
  starbot:
    lang: node16
    handler: ./starbot
    image: alexellis2/starbot:latest

    secrets:
     - astro-key
```

Update `handler.js`:

```
const { readFile } = require('fs').promises

module.exports = async (event, context) => {
  let input = event.headers["x-api-key"]
  let key = await readFile("/var/openfaas/secrets/astro-key")
  if(key != input) {
    return context
      .status(401)
      .headers({"Content-type": "text/plain"})
      .fail("Unauthorized")
    }

  return context
    .status(200)
    .headers({"Content-type": "text/plain"})
    .succeed("OK")
}
```

Deploy via `faas-cli up`

Now try invoking it, with and without the header using the token saved in `token.txt`.

```
faas-cli invoke starbot
Unauthorized

echo | faas-cli invoke starbot \
 --header "x-api-key=4dddb48c2ad4fa8e4c3f7400f389971674c5908c"
OK
```

Storing data with databases

There are a number of options for storing data from your functions. A managed database is probably the easiest, and DigitalOcean provides managed Postgres, Redis and MySQL for around 15 USD / mo.

Alternatively, you can run a database in your faasd stack by following the instructions in the *Adding a database - Postgresql* section.

Let's create a function that we can use to store links for an imaginary newsletter. We'll collect them throughout the week and share them at the weekend.

We'll use the `GET` HTTP method to retrieve items and the `POST` method to send items into our list.

This is an example of the input body we will use:

```
{ "url": "https://inlets.dev",
  "description": "Cloud Native Tunnels"
}
```

And our schema:

```
CREATE TABLE links (
  id INT GENERATED ALWAYS AS IDENTITY,
  created_at timestamp not null,
  url text NOT NULL,
  description text NOT NULL
);
```

The main configuration items for Postgresql are:

- URL
- database-port
- username
- password
- database-name

All of these items could be considered as confidential, which means they should be managed as secrets. The main difference between a managed SQL service, and hosting your own in the faasd stack, is that your local version is unlikely to have SSL/TLS enabled.

```
export OPENFAAS_PREFIX=alexellis2
faas-cli new --lang node16 push-pull

faas-cli secret create db-user --from-literal postgres
faas-cli secret create db-password --from-literal PASSWORD_HERE
faas-cli secret create db-host --from-literal 10.62.0.1
```

> Here `10.62.0.1` refers to the bridge device `openfaas0` and is private within your faasd host. In a future version of faasd, you will be able to specify: `postgresql` instead.

Update `push-pull.yml` and add each secret to the "secrets:" array.

```
secrets:
  - db-user
  - db-password
  - db-host
```

Add an environment variable in `push-pull.yml` for the `db-name: postgres` and `db-port: 5432`.

```
environment:
  db-name: postgres
  db-port: 5432
```

This is the complete functions section of the file:

```
functions:
  push-pull:
    lang: node16
    handler: ./push-pull
    image: alexellis2/push-pull:latest
    environment:
      db-name: postgres
      db-port: 5432
    secrets:
     - db-user
     - db-password
     - db-host
```

Now add the pg npm module to the code:

```
cd ./push-pull
npm install --save pg
```

Now edit the handler.js:

```
'use strict'

const { Client } = require('pg')
const fs = require('fs').promises;

module.exports = async (event, context) => {
  let result = {};

  const user = await fs.readFile("/var/openfaas/secrets/db-user","utf8")
  const pass = await fs.readFile("/var/openfaas/secrets/db-password","utf8")
  const host = await fs.readFile("/var/openfaas/secrets/db-host","utf8")
  const port = process.env["db-port"]
  const name = process.env["db-name"]

  const opts = {
      user: user,
      host: host,
      database: name,
      password: pass,
      port: port,
  }

  if(event.method == "GET") {
    const client = new Client(opts)
    await client.connect()

    const res = await client.query(
      'SELECT id, url, description, created_at from links')
    result = res.rows
    await client.end()
  } else if(event.method == "POST") {
    const client = new Client(opts)
    await client.connect()

    const res = await client.query(
      'INSERT INTO links(id, url, description, created_at)'+
      ' VALUES (DEFAULT, $1, $2, now())',
        [event.body.url,
        event.body.description])
    result = res.rows[0]
    await client.end()
  }

  return context
    .status(200)
    .succeed(result)
}
```

Create the initial schema with psql, you may need to install the following package to get the CLI tool:

```
sudo apt install postgresql-client

psql -h 10.62.0.1 -U postgres
Password for user postgres:
```

```
# Then paste in the schema from above.

\dt
         List of relations
 Schema | Name  | Type  |  Owner
--------+-------+-------+----------
 public | links | table | postgres
(2 rows)

SELECT * from links;
 id | created_at | url | description
----+------------+-----+-------------
(0 rows)
```

Deploy the function:

```
faas-cli up -f push-pull.yml
```

Test the function by posting a link:

```
echo '{ "url": "https://inlets.dev",
  "description": "Cloud Native Tunnels"
}' |
curl --data-binary @- \
  --header "Content-Type:application/json" \
  $OPENFAAS_URL/function/push-pull
```

Check it was inserted:

```
curl -s $OPENFAAS_URL/function/push-pull

{
  "id": 3,
  "url": "https://inlets.dev",
  "description": "Cloud Native Tunnels",
  "created_at": "2021-01-18T13:26:04.187Z"
}
```

> Hint: if you have the `jq` utility installed, then you can pipe the output to jq to pretty-print it.

You can now store and query data from a relational database. There are more efficient ways of accessing a database than creating a connection per request, which you can learn about in the postgres-node docs under the "pool" section.

The node16 template reuses the same memory for each request, meaning you can obtain a pool of connections once and re-use it for each request. This is beyond the scope of the eBook.

> Bonus task: Given what you've learnt about using API tokens in the previous, can you update the push-pull function to only allow POSTs when a valid token is passed?

Unit testing with Node.js

When using an OpenFaaS template, the Dockerfile can optionally specify a command to run unit tests during the `faas-cli build/publish`.

Look at the `template/node16/Dockerfile` and you will run a line that reads: `npm test`.

You can unit test your function's handler or any library functions that you may write. The following example uses the mocha test runner, and the chai assertion library to help you verify the results.

We'll use a Test-Driven Development style to:

- Write a function that returns a 200 status code
- Then write a unit test that expects a 201 status code
- See it fail
- Then update the code to 201 in the `handler.js`
- And see it pass

Create a new function:

```
export OPENFAAS_PREFIX="alexellis2"

faas-cli new --lang node16 api
```

Edit `handler.js`:

```
'use strict'

module.exports = async (event, context) => {
  const result = {
    "status": "ok"
  }

  return context
    .status(200)
    .succeed(result)
}
```

Install the `mocha` and `chai` npm modules:

```
cd api
npm install --save mocha chai
```

These will now be listed in `package.json`.

Next, modify the `test` step in the `package.json` file:

```
  "scripts": {
    "test": "mocha test.js"
  }
```

Now create your first test:

```
'use strict'

const expect = require('chai').expect
const handler = require('./handler') // The function's handler

describe('Our API', async function() {
    it('gives a 201 for any request', async function() {
        let cb = async function (err, val) { };
        let context = new FunctionContext(cb);
        let event = new FunctionEvent({body: ''})
        let res = await handler(event, context)
        expect(context.status()).to.equal(201)
    });
});
```

Then, add the following test-doubles underneath for the FunctionEvent and FunctionContext. These test-doubles (also known as stubs) allow you to provide canned values for the input, and to inspect the results.

```
class FunctionEvent {
    constructor(req) {
        this.body = req.body;
        this.headers = req.headers;
        this.method = req.method;
        this.query = req.query;
        this.path = req.path;
    }
}
```

```
class FunctionContext {
    constructor(cb) {
        this.statusValue = 200;
        this.cb = cb;
        this.headerValues = {};
        this.cbCalled = 0;
    }
    status(value) {
        if(!value) {
            return this.statusValue;
        }

        this.statusValue = value;
        return this;
    }
    headers(value) {
        if(!value) {
            return this.headerValues;
        }

        this.headerValues = value;
        return this;
    }
    succeed(value) {
        let err;
        this.cbCalled++;
        this.data = value;
        this.cb(err, value);
    }
    fail(value) {
        let message;
        this.cbCalled++;
        this.cb(value, message);
    }
}
```

You can find the whole file in this GitHub repo: alexellis/openfaas-node16-mocha-unit-test/

Now you can `faas-cli build`, and you will see mocha running and printing results to the console. If the tests fail, then mocha will produce a non-zero exit code and the build will stop.

```
> openfaas-function@1.0.0 test /home/app/function
> mocha test.js

  Our API
    1) gives a 201 for any request

  0 passing (8ms)
  1 failing

  1) Our API
       gives a 201 for any request:

      AssertionError: expected 200 to equal 201
      + expected - actual

      -200
      +201

      at Context.<anonymous> (test.js:13:37)
```

You can also run the unit-tests on your local machine if you have Node.js and npm already installed:

```
cd api/
npm test
```

Now edit the handler to return a 201 status so that the unit test passes.

Then run `faas-cli build` again.

Here is an example of the unit test passing:

```
 ---> Running in 47a902a0d06a

> openfaas-function@1.0.0 test /home/app/function
> mocha test.js

  Our API
    [x] gives a 201 for any request

  1 passing (6ms)

Removing intermediate container 47a902a0d06a
```

You can also customise the test-doubles for `FunctionEvent` and `FunctionContext` or create your own `npm` module for them, to reduce duplication. The same technique should apply to other unit-test runners such as Jest, which is more popular with the React community.

How to build multiple functions

You can have multiple OpenFaaS stack files with a single function in them, but it makes sense to use a single file to filter it when required. It simplifies maintenance and CI/CD.

> Hint: If you rename your function's YAML file to "stack.yml", you no longer need to use the `-f` parameter.

You can append additional functions into your stack file. Then you can build and deploy them in one shot, or filter down to just the one which has changes:

```
faas-cli new --lang node16 hackernewsbot --append stack.yml
```

We'll now have both `starbot` and `hackernewsbot` in our YAML file with their own handlers and own package.json files.

```
version: 1.0
provider:
  name: openfaas

functions:
  starbot:
    lang: node16
    handler: ./starbot
    image: alexellis2/starbot:latest

  hackernewsbot:
    lang: node16
    handler: ./hackernewsbot
    image: alexellis2/hackernewsbot:latest
```

If you run any of the commands like build/publish/up or deploy, then everything in the file will be used by default. To filter to just one use `--filter NAME`.

For example:

```
faas-cli up --filter hackernewsbot
```

A super-power of `faas-cli` is how it can enable parallel builds of functions using Docker. Here's an example to build up to two functions at once:

```
faas-cli up --parallel 2
```

Invoking functions

Functions can be invoked either synchronously (the caller waits until it's done), or asynchronously (the caller gets an `X-Call-ID` header back and doesn't have to wait).

Typically, you will want to use a synchronous invocation, which is the simplest and easiest to consume.

You can find your function's URL using `faas-cli describe` or by opening the OpenFaaS UI dashboard at http://127.0.0.1:8080

```
faas-cli describe figlet
Name:                figlet
Status:              Ready
Replicas:            1
```

```
Available replicas:  1
Invocations:         0
Image:
Function process:
URL:                 http://127.0.0.1:8080/function/figlet
Async URL:           http://127.0.0.1:8080/async-function/figlet
```

The simplest way to invoke your function is by using its URL exposed by the gateway: `http://127.0.0.1:8080/function/figle`

You can also invoke the function the function via the CLI:

```
echo faasd | faas-cli invoke figlet

  __                     _ 
 / _| __ _  __ _ ___  __| |
| |_ / _` |/ _` / __|/ _` |
|  _| (_| | (_| \__ \ (_| |
|_|  \__,_|\__,_|___/\__,_|

```

Now, if you're doing something like generating PDFs, and you are either processing a lot of them, or they take a while to produce, then you will want to consider asynchronous invocations. This is the same as using a queue, except that the result is thrown away by default.

To receive the result, simply pass in a *Callback URL*. You will be able to know which result is for which invocation by matching on the *Call ID*.

```
curl --data-binary input.jpg http://127.0.0.1:8080/async-function/resize/?width=500px \
  --header "X-Callback-Url: http://example.com/api/callback"
```

The faas-cli can also be used to invoke functions asynchronously, with a slightly different syntax:

```
cat input.jpg | faas-cli invoke --async analyse-image \
  --header "X-Callback-Url=http://example.com/api/callback"
```

The queue-worker will then send you the result and a HTTP code (depending on success or failure) to `http://example.com/api/callback`

The recipient of the callback could be another function deployed with faasd, or any other HTTP endpoint.

To match a request with a response use the `X-Call-Id` header received when you invoked the function.

Making any process into a function

A common use-case we have seen with OpenFaaS is to take a CLI and make it into a function. You can do this with anything that can be installed into a container, from ImageMagick to the AWS CLI to ffmpeg.

There are two good blog posts for this:

- Turn Any CLI into a Function with OpenFaaS
- Stop installing CLI tools on your build server — CLI-as-a-Function with OpenFaaS

The easiest example is to use the "dockerfile" template and have it run a pre-installed bash command:

```
# Customise this to your own username
export OPENFAAS_PREFIX=alexellis2
faas-cli new --lang dockerfile env
```

The fprocess line tells the container what to run as the function, change it to env:

```
FROM ghcr.io/openfaas/classic-watchdog:0.1.4 as watchdog

FROM alpine:3.12

RUN mkdir -p /home/app
COPY --from=watchdog /fwatchdog /usr/bin/fwatchdog
RUN chmod +x /usr/bin/fwatchdog

# Add non root user
RUN addgroup -S app && adduser app -S -G app
RUN chown app /home/app
WORKDIR /home/app

USER app

# Populate example here - i.e. "cat", "sha512sum" or "node index.js"
ENV fprocess="env"

CMD ["fwatchdog"]
```

Now deploy it, and remember that Raspberry Pi users need to run faas-cli publish/deploy instead.

```
faas-cli up -f env.yml
```

When you invoke it, you'll see all the HTTP headers injected as environment variables:

```
echo | faas-cli invoke env

PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
fprocess=env
HOME=/home/app
Http_X_Forwarded_Host=faasd.alex.o6s.io
Http_User_Agent=curl/7.68.0
Http_Content_Type=application/x-www-form-urlencoded
Http_X_Forwarded_For=81.99.136.188
Http_Accept_Encoding=gzip
Http_X_Forwarded_Proto=https
Http_Content_Length=0
Http_Accept=*/*
Http_Method=POST
Http_ContentLength=0
Http_Path=/
Http_Host=10.62.0.92:8080
```

Let me show you how to run curl from inside a function. This is very useful for testing connectivity and DNS.

Now add this line to your Dockerfile before `USER app` to add the `curl` package into the container.

```
RUN apk add --no-cache curl
```

Now change the `fprocess` to `xargs curl` in the Dockerfile:

```
ENV fprocess="xargs curl"
```

Run `faas-cli up -f env.yml` again to redeploy the function.

Now test it out:

```
curl -SLs https://faasd.example.com/function/env \
  --data "-s http://api.open-notify.org/astros.json"
```

Abbreviated output:

```
{
  "message": "success",
  "number": 3,
  "people": [
    {
      "craft": "ISS",
      "name": "Sergey Ryzhikov"
    },
    {
      "craft": "ISS",
      "name": "Kate Rubins"
    },
    {
      "craft": "ISS",
      "name": "Soichi Noguchi"
    }
  ]
}
```

You may also like to try out the `bash` template. You can search for templates in the store with `faas-cli template store list` and then fetch it to create a new function.

```
NAME                         SOURCE              DESCRIPTION
bash-streaming               openfaas-incubator  Bash Streaming template

faas-cli template store pull bash-streaming
faas-cli new --lang bash-streaming bash-script
```

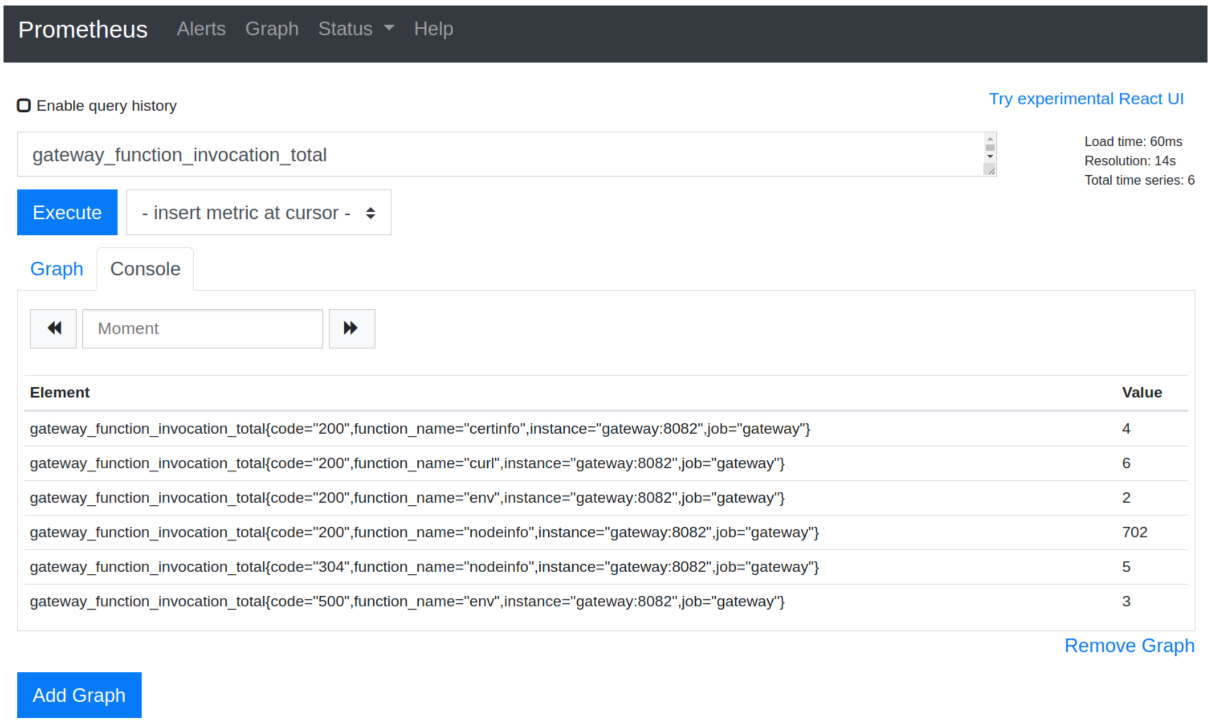
Chapter 5

Monitoring invocations

You can monitor the OpenFaaS API, along with any functions you've deployed using Prometheus. Prometheus is a time series database which can be used to record metrics over time for querying and alerting. It comes with a lightweight UI for testing queries and is usually integrated into Cloud Native projects like OpenFaaS and inlets.

We cover Grafana in a later chapter, which is an essential tool for turning Prometheus queries into dashboards. Prometheus is built into the docker-compose.yaml file for faasd, but Grafana needs to be added separately.

Alerting is a cornerstone of Site Reliability Engineering (SRE) and can be used to detect anomalies. It's outside of the scope of this book, but you can find out more here: Defining alerting rules



```
gateway_function_invocation_total
```

Load time: 60ms
Resolution: 14s
Total time series: 6

Element	Value
gateway_function_invocation_total{code="200",function_name="certinfo",instance="gateway:8082",job="gateway"}	4
gateway_function_invocation_total{code="200",function_name="curl",instance="gateway:8082",job="gateway"}	6
gateway_function_invocation_total{code="200",function_name="env",instance="gateway:8082",job="gateway"}	2
gateway_function_invocation_total{code="200",function_name="nodeinfo",instance="gateway:8082",job="gateway"}	702
gateway_function_invocation_total{code="304",function_name="nodeinfo",instance="gateway:8082",job="gateway"}	5
gateway_function_invocation_total{code="500",function_name="env",instance="gateway:8082",job="gateway"}	3

Figure 5.1: The built-in Prometheus UI

The Prometheus UI showing the query executor

By default, Prometheus is not exposed on the public IP of your faasd host, because it does not have authentication turned on.

We can still access it through the loopback adapter (127.0.0.1) through an inlets or SSH tunnel.

Create an SSH tunnel to faasd and forward the Prometheus UI back to your computer:

```
export IP="faasd-ip"

ssh -L 9090:127.0.0.1:9090 ubuntu@$IP
```

Note that the user *ubuntu* may vary depending on where you are hosting your faasd VM.

Now navigate to http://localhost:9090

Prometheus queries are written using the PromQL language, you can learn querying basics here.

You can find the start rate for new function invocations, over the past 1 minute with:

```
rate(gateway_invocation_started [1m])
```

You can view completed function invocations over 30 seconds using the following query:

```
rate(gateway_function_invocation_total{}[30s])
```

This can be filtered by status code (i.e. 4xx and 5xx) or by function name:

```
rate(gateway_function_invocation_total{code=~"4..|5.."}[30s])

rate(gateway_function_invocation_total{function_name="nodeinfo.openfaas-fn"}[5m])
```

You can get the average response time for functions by dividing the sum of all response times by the count of recorded responses:

```
gateway_functions_seconds_sum / gateway_functions_seconds_count
```

To view the replica count of functions, and whether they are scaled to zero see:

```
gateway_service_count
```

Or view only scaled up functions:

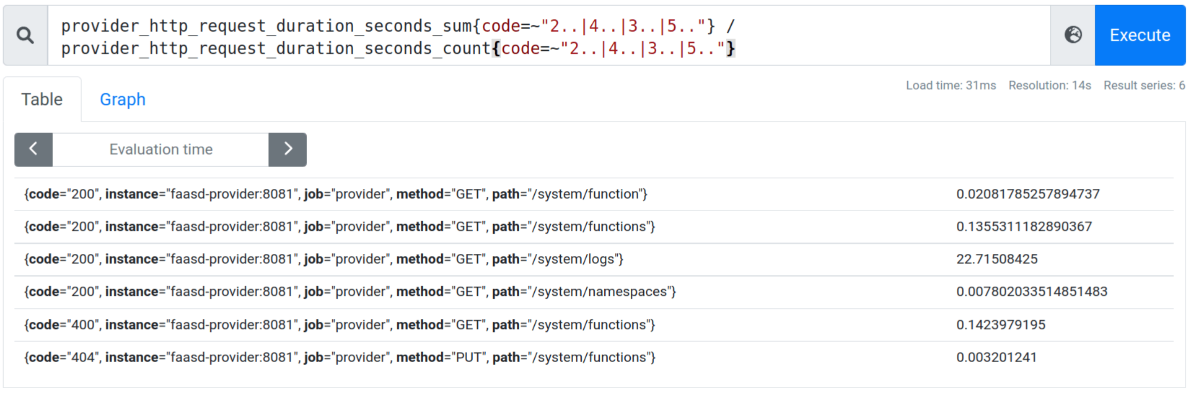
```
gateway_service_count > 0
```

The OpenFaaS REST API can be monitored through the following two metrics:

```
provider_http_request_duration_seconds
provider_http_requests_total
```

For the following query, we can see the logs endpoint has been taking a long time. That's because it's a streaming API and the connection is held open for as long as the user desires:

```
provider_http_request_duration_seconds_sum{code=~"2..|4..|3..|5.."} /
  provider_http_request_duration_seconds_count{code=~"2..|4..|3..|5.."}
```

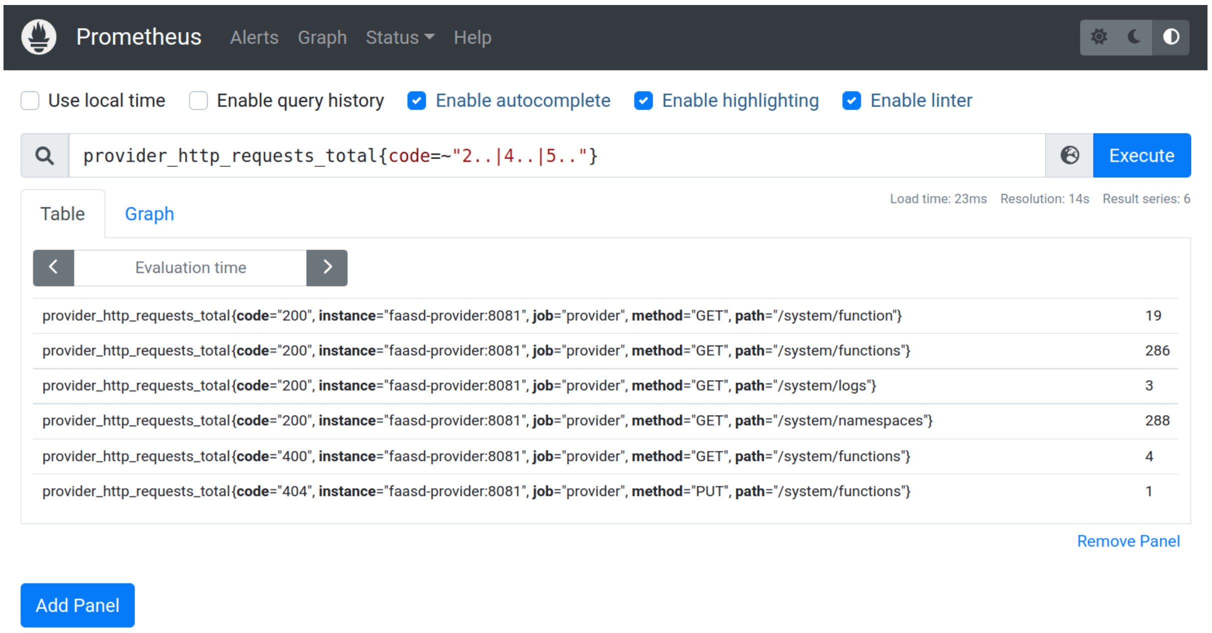

```
provider_http_request_duration_seconds_sum{code=~"2..|4..|3..|5.."} /
provider_http_request_duration_seconds_count{code=~"2..|4..|3..|5.."}
```

Load time: 31ms   Resolution: 14s   Result series: 6

```
{code="200", instance="faasd-provider:8081", job="provider", method="GET", path="/system/function"}      0.02081785257894737
{code="200", instance="faasd-provider:8081", job="provider", method="GET", path="/system/functions"}     0.1355311182890367
{code="200", instance="faasd-provider:8081", job="provider", method="GET", path="/system/logs"}          22.71508425
{code="200", instance="faasd-provider:8081", job="provider", method="GET", path="/system/namespaces"}    0.007802033514851483
{code="400", instance="faasd-provider:8081", job="provider", method="GET", path="/system/functions"}     0.1423979195
{code="404", instance="faasd-provider:8081", job="provider", method="PUT", path="/system/functions"}     0.003201241
```

> The average duration for HTTP request to the OpenFaaS API

The `provider_http_requests_total` metric is more useful for showing HTTP status codes along with how frequently they are being invoked. The OpenFaaS UI and OpenFaaS Pro Dashboard both make use of the API, so if you are seeing a large number of requests, it could be because you've left a browser window open somewhere.



```
provider_http_requests_total{code=~"2..|4..|5.."}
```

Load time: 23ms Resolution: 14s Result series: 6

provider_http_requests_total{code="200", instance="faasd-provider:8081", job="provider", method="GET", path="/system/function"}	19
provider_http_requests_total{code="200", instance="faasd-provider:8081", job="provider", method="GET", path="/system/functions"}	286
provider_http_requests_total{code="200", instance="faasd-provider:8081", job="provider", method="GET", path="/system/logs"}	3
provider_http_requests_total{code="200", instance="faasd-provider:8081", job="provider", method="GET", path="/system/namespaces"}	288
provider_http_requests_total{code="400", instance="faasd-provider:8081", job="provider", method="GET", path="/system/functions"}	4
provider_http_requests_total{code="404", instance="faasd-provider:8081", job="provider", method="PUT", path="/system/functions"}	1

> Frequency of invocation and HTTP codes for the OpenFaaS API itself

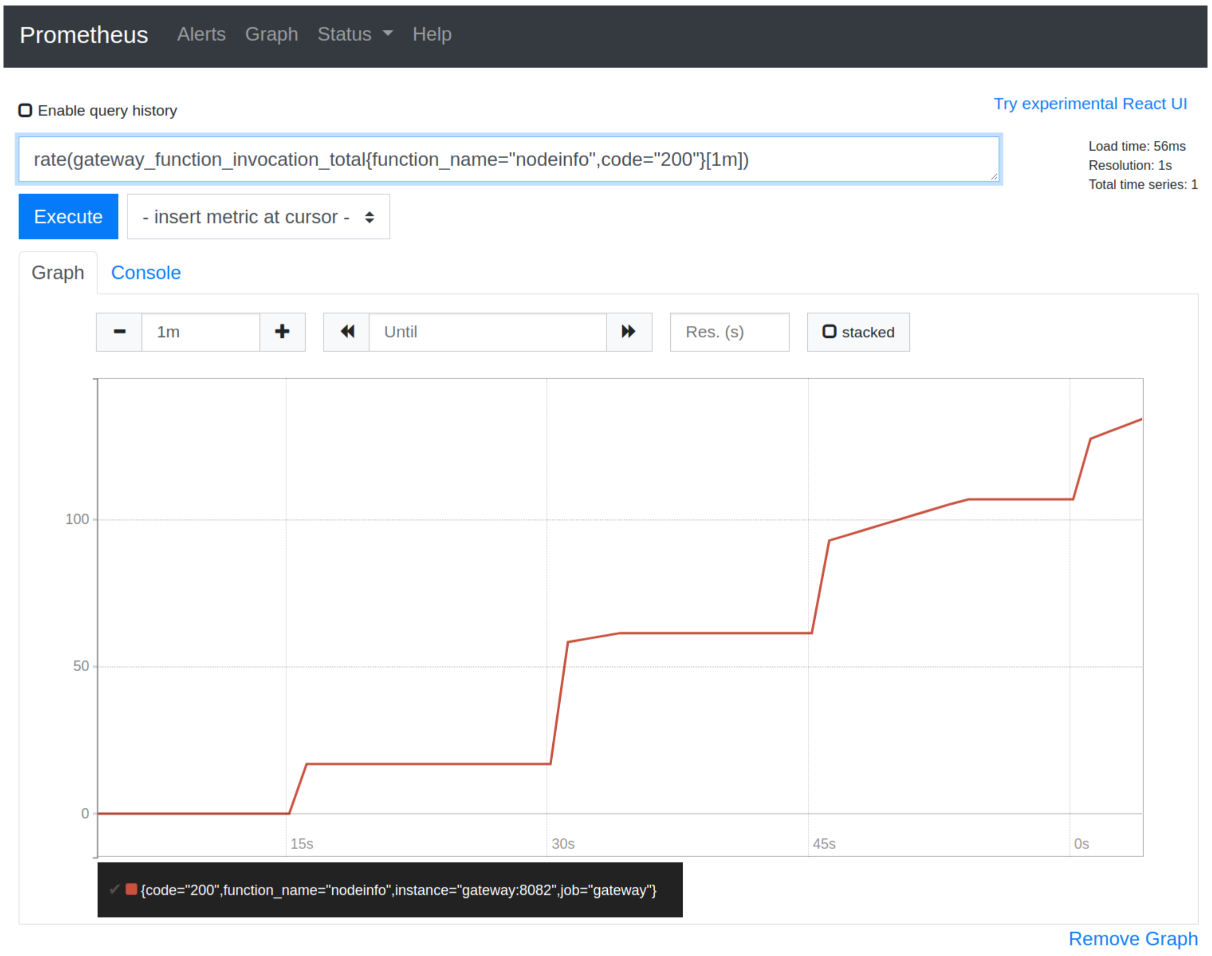
You'll be able to query Prometheus in the browser and see how long executions are taking, and whether any are failing with non 200 HTTP codes.

The Prometheus UI showing a graph of the execution rate going up under a load test

See also: docs: OpenFaaS metrics

In the "Additional containers and services" section you'll learn how to deploy a Grafana dashboard to monitor the data from Prometheus.

You can view the PromQL behind any of the Grafana panels that you see, and edit them as required, or use them to learn what data drives a particular graph or chart.



```
rate(gateway_function_invocation_total{function_name="nodeinfo",code="200"}[1m])
```

Load time: 56ms
Resolution: 1s
Total time series: 1

{code="200",function_name="nodeinfo",instance="gateway:8082",job="gateway"}

Figure 5.2: Prometheus graph

Chapter 6

Remote access and self-hosting faasd

There are three use-cases for accessing faasd *remotely* from the Internet:

1) You are self-hosting functions and want users to access them
2) You build your functions in a cloud CI service like GitHub Actions and need to deploy updates
3) Your faasd runs in a VM or behind NAT and needs to receive webhooks for integration purposes

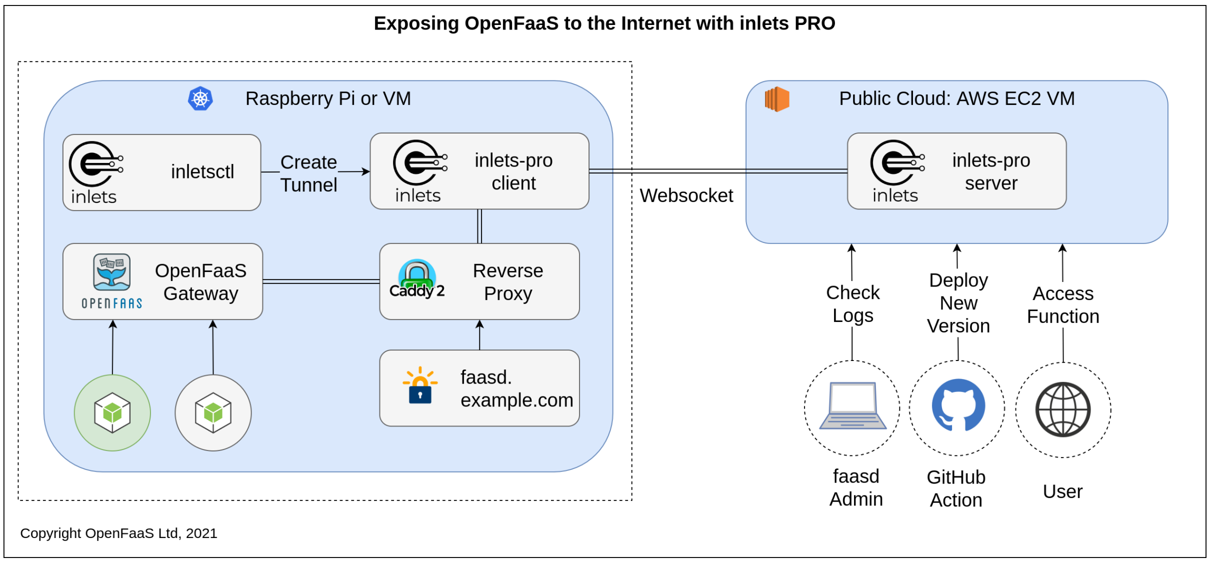


Figure 6.1: OpenFaaS exposed with inlets

In either case, you can use the inlets Pro project to enable a secure tunnel, and allow users to access your faasd functions. CI/CD systems will be able to access your OpenFaaS gateway, but only if they have your password.

The easiest way to get started is to create an "exit-server" on your favourite cloud. The exit-server is going to act as a router and will loan its IP address to us. Users will connect to its IP address, and then be routed through the tunnel to our faasd service.

> It is highly recommended that you install a reverse proxy such as Caddy using the instructions in the book, and expose port 80 and 443 only.

Install inletsctl and inlets-pro on your RPi or VM:

```
curl -sLS https://inletsctl.inlets.dev | sudo sh

sudo inletsctl download
```

Now create an exit server on your favourite cloud, for instance with DigitalOcean:

```
sudo inletsctl create \
  --provider digitalocean \
  --region lon1 \
  --access-token-file $HOME/do-access-token \
```

After the provisioning has completed, you can generate a systemd unit file and install inlets Pro as a service, so it starts up upon boot-up. Use the outputs from the `inletsctl create` command for the next step.

```
export IP=""
export TOKEN=""

sudo inlets-pro client \
  --url wss://$IP:8123/connect \
  --generate=systemd \
  --license-file $HOME/.inlets/LICENSE \
  --token $TOKEN \
  --upstream localhost \
  --ports 80,443 > faasd-tunnel.service
```

The upstream flag means traffic will be routed directly to the faasd instance. The TCP ports are 80 and 443, which are required for Caddy, port 8080 won't be exposed on the Internet.

Install the service:

```
sudo cp faasd-tunnel.service /etc/systemd/system/
sudo systemctl enable faasd-tunnel
```

Check the service started and view its logs:

```
sudo systemctl start faasd-tunnel
sudo journalctl -u faasd-tunnel
```

Now install Caddy with the instructions in the book. You will get a TLS certificate and will be able to access your faasd instance as documented. When you create the DNS A record for your openfaas domain, use the IP address from the previous step.

You will need an inlets subscription to start the client, you can get started with a personal plan at: inlets.dev

Later, you'll also be able to deploy new functions to your faasd instance from a GitHub Action over the tunnel.

Chapter 7

Adding TLS to faasd

You can add a TLS certificate to your OpenFaaS gateway using a reverse proxy like Caddy.

> Note: skip this if you used Terraform to install faasd to a DigitalOcean Droplet, because it's already installed.

You can either run it as an additional service in your docker-compose.yaml file or install it on the host.

This is how to install Caddy directly on your host.

Create a systemd unit file:

```
curl -s -o caddy.service \
  https://raw.githubusercontent.com/caddyserver/dist/master/init/caddy.service
sudo cp caddy.service /etc/systemd/system/caddy.service
sudo systemctl enable caddy
```

Download Caddy using arkade. Arkade will ensure that you get the right binary for your system:

```
arkade get caddy
sudo mv $HOME/.arkade/bin/caddy /usr/local/bin/caddy
```

Now create a Caddyfile at `/etc/caddy/Caddyfile`:

```
{
  email you@example.com
}

faasd.example.com {
  reverse_proxy 127.0.0.1:8080
}
```

Then run Caddy with:

```
sudo systemctl start caddy
```

You can view its logs for information or errors with: `sudo journalctl -u caddy`

When you change your domains or add additional settings later on, you can run the following to restart and reload Caddy:

```
sudo systemctl daemon-reload
sudo systemctl restart caddy
```

Adding custom domains per function

If you already have Caddy installed on your faasd host, then you can add an additional domain for specific functions by editing your Caddyfile and restarting the services.

Remember that for any custom domains you add to your Caddyfile, you'll need to create a mapping in your DNS providers' dashboard. This is usually going to be an A record, and I'd recommend a short TTL value to start off, like 1M or 60s. The IP address should be either the public IP of your inlets server or the IP of your public VPS/VM.

This is useful when you need a "pretty" domain for a function, for example, if you wanted to give yourself a domain of `info.example.com` for the `nodeinfo` function:

```
info.example.com {
    handle_path /* {
      rewrite * /function/nodeinfo{path}
      reverse_proxy 127.0.0.1:8080
    }
    reverse_proxy 127.0.0.1:8080 {
    }
}
```

Create a DNS A record for the new subdomain `info.example.com` and the public IP of your host.

Then restart the services:

```
sudo systemctl daemon-reload
sudo systemctl restart caddy
```

Check the logs with `sudo journalctl -u faasd`.

The other use-case for using a domain for your function, is that you can edit your Caddy rule to switch users over to a new version of a function without telling them to update their calling code.

Here's how you could roll-over from v1 to v2 just by changing your Caddyfile:

To begin with the domain goes to the api-v1 function:

```
api.example.com {
    handle_path /* {
      rewrite * /function/api-v1{path}
      reverse_proxy 127.0.0.1:8080
    }
    reverse_proxy 127.0.0.1:8080 {
    }
}
```

Then later you deploy api-v2, validate that it works as expected and can update the Caddyfile by changing the `rewrite` path to `api-v2`.

```
api.example.com {
    handle_path /* {
      rewrite * /function/api-v2{path}
      reverse_proxy 127.0.0.1:8080
    }
    reverse_proxy 127.0.0.1:8080 {
    }
}
```

Adding a custom domain for your Grafana dashboard

In the next chapter, you'll learn how to deploy a Grafana dashboard. It listens to HTTP traffic on port 3000.

You can use your Caddy configuration to get another certificate for the dashboard using a custom domain.

Add the following to: `/etc/caddy/Caddyfile`:

```
grafana.example.com {
    reverse_proxy 127.0.0.1:3000 {
    }
}
```

Make sure you add a DNS A record for your custom sub-domain, then restart the service:

```
sudo systemctl daemon-reload
sudo systemctl restart caddy
```

Chapter 8

Additional containers and services in docker-compose.yaml

faasd supports deploying both functions via its REST API, and a series of additional services which are managed by `faasd` itself. This can be used to add something like a storage or caching capability to your functions. We'll explore how to configure the OpenFaaS core services and how to add Grafana for creating a dashboard and InfluxDB, which is a time series database.

Whilst faasd uses a file named `docker-compose.yaml`, it does not use Docker or Compose, only its format and spec.

faasd supports a sub-set of directives in the compose format including:

- bind-mounting volumes - for storage and persistence
- setting a specific runtime user, so that bind-mounting can be accessed by apps
- exposing services through ports
- adding Linux capabilities
- setting a dependency order

See the `docker-compose.yaml` file located at `/var/lib/faasd` for examples.

Once you deploy a core service, it's up to you how you want to interact with it.

- You can use it only from your functions using the service name for discovery such as influxdb, to allow functions to store data
- You can expose it only on the local host and use an inlets or SSH tunnel to forward it back to your local machine or another network

When it comes to exposing something like Grafana or Prometheus, think twice about how you are going to do that.

By default it has no TLS or authentication, so perhaps it's best if you deploy Caddy, and add a basic-auth rule in front of it, or perhaps only access it via a tunnel on your local machine.

Exposing core services

The OpenFaaS stack is made up of several core services including NATS and Prometheus.

Edit `/var/lib/faasd/docker-compose.yaml` and change:

You can expose a core service like the gateway on all interfaces by using the format `FROM_PORT:TO_PORT`. If you do this on an Internet-facing host, anyone with the host's IP address will be able to connect to the service.

```
  gateway:
    ports:
       - "8080:8080"
```

Expose Prometheus only to `127.0.0.1`.

You can expose a service to only the loopback of the faasd host with the following:

```
  prometheus:
    ports:
       - "127.0.0.1:9090:9090"
```

Expose Prometheus to only the OpenFaaS bridge device `openfaas0`, so that functions and core services can access it:

```
  prometheus:
    ports:
       - "10.62.0.1:9090:9090"
```

Most services should either not be exposed, or only exposed on the OpenFaaS bridge device.

Bind mount a folder:

Note that the source of every folder starts with `/var/lib/faasd/` so if you want to mount a `grafana` folder, it must be created at `/var/lib/faasd/grafana/`

```
  grafana:
...
    volumes:
      - type: bind
        source: ./grafana/
        target: /etc/grafana/provisioning/
```

You may also need to `chown` the local folder with a user like `1000` and also override the user in the container's section.

In the example above, that would mean running:

```
chown -R 1000:1000 /var/lib/faasd/grafana
```

Run as a specific user:

```
  postgresql:
    user: "1000"
```

If you do not have user `1000` in `/etc/passwd` you can add a system user which cannot log in for this purpose.

```
groupadd --gid 1000 faasd

useradd --uid 1000 --system \
  --no-create-home \
  --gid 1000 faasd
```

All containers for the core services and functions are added to a Linux bridge device called `openfaas0`, you can use its IP address to reference core services from functions. By default the address is `10.62.0.1`, but you can look it up with `ifconfig` or `ip addr`.

If using this value from your function, it is advisable to add an environment variable in your stack.yml file and read that at runtime, instead of hard-coding it.

```
functions:
  read-db:
    environment:
      postgres_host: 10.62.0.1
```

stack.yml example

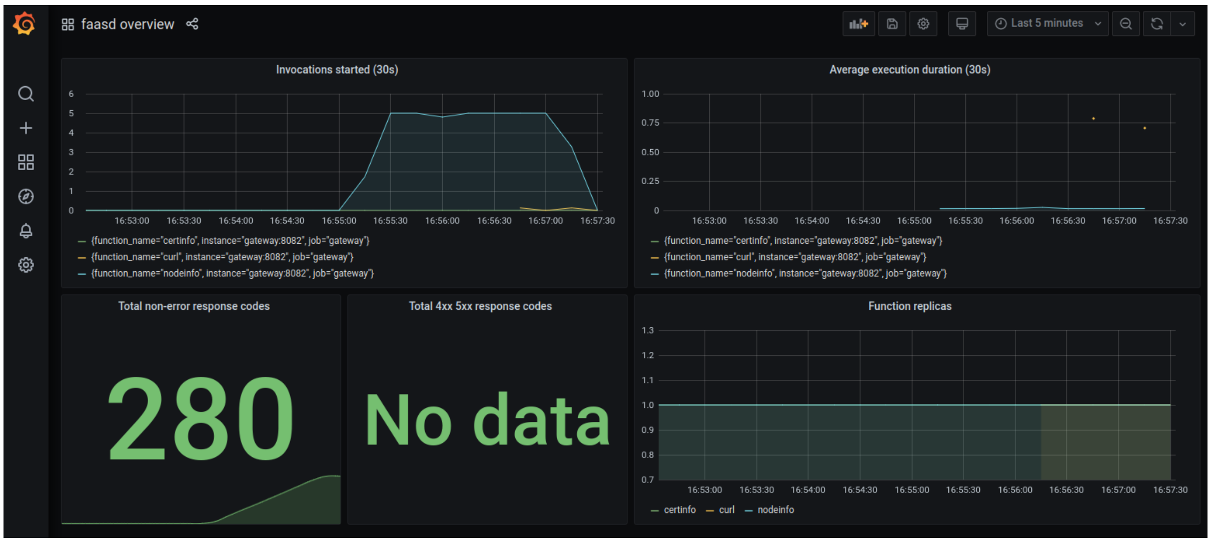
```
process.env.postgres_host
```

Example of accessing the variable in your code.

In a future version of faasd, you will be able to access any additional services you add using their name, so `postgresql` or `prometheus`.

Adding a Grafana Dashboard

Grafana is a visualisation tool for viewing time-series data from sources like InfluxDB and Prometheus. You can use it to monitor your OpenFaaS functions and see how long they take to execute, along with how many return a failure HTTP code.



faasd overview
Last 5 minutes
Invocations started (30s)
Average execution duration (30s)
{function_name="certinfo", instance="gateway:8082", job="gateway"}
{function_name="curl", instance="gateway:8082", job="gateway"}
{function_name="nodeinfo", instance="gateway:8082", job="gateway"}
Total non-error response codes
280
Total 4xx 5xx response codes
No data
Function replicas
certinfo
curl
nodeinfo

Figure 8.1: Grafana dashboard with a light load-test

Grafana dashboard with a light load-test

Make a data directory for Grafana:

```
sudo mkdir -p /var/lib/faasd/grafana
```

Edit `/var/lib/faasd/docker-compose.yaml` and add:

```
  grafana:
    image: docker.io/grafana/grafana:latest
    environment:
      - GF_AUTH_ANONYMOUS_ORG_ROLE=Admin
      - GF_AUTH_ANONYMOUS_ENABLED=true
      - GF_AUTH_BASIC_ENABLED=false
    volumes:
      # we assume cwd == /var/lib/faasd
      - type: bind
```

```
        source: ./grafana/
        target: /etc/grafana/provisioning/
    cap_add:
      - CAP_NET_RAW
    depends_on:
      - prometheus
    ports:
      - "3000:3000"
```

By default this will expose Grafana on the IP of the host, you can avoid this and use 127.0.0.1:3000 access Grafana:

```
    ports:
      - "127.0.0.1:3000:3000"
```

Restart faasd with:

```
sudo systemctl daemon-reload \
  && sudo systemctl restart faasd
```

Then check Grafana's logs:

```
sudo journalctl -t openfaas:grafana
```

Once you start Grafana, add a Datasource for Prometheus:

- Name: `faasd`
- URL: `http://prometheus:9090`

Depending on your tier, you may have purchased the official faasd dashboard, if so, you should be able to download the `dashboard.json` file from Gumroad and then import it on the Dashboards page. The URL for your dashboard will be `http://localhost:3000/dashboard/import`

You are looking for the field "Import via panel json" which is where you paste in the data. The dashboard will now appear under your Dashboards list.

Adding a database - Postgresql

Postgresql is described as "The World's Most Advanced Open Source Relational Database" and can be hosted on your faasd instance.

Add the following to docker-compose.yaml:

```
  postgresql:
    image: docker.io/library/postgres:11
    environment:
      PGDATA: /var/lib/postgresql/data/pgdata
      POSTGRES_PASSWORD_FILE: /run/secrets/postgres-passwd
      POSTGRES_USER: postgres
    volumes:
      - type: bind
        source: ./postgresql/pgdata
        target: /var/lib/postgresql/data/pgdata
      - type: bind
        source: ./postgresql/run
        target: /var/run/postgresql
      - type: bind
        source: ./postgresql/postgres-passwd
        target: /run/secrets/postgres-passwd
    user: "1000"
    ports:
      - "10.62.0.1:5432:5432"
```

Now create the data directories, so that your instance is stateful and can be restarted without dataloss:

```
# For databases
sudo mkdir -p /var/lib/faasd/postgresql/pgdata

# For the run lock
sudo mkdir -p /var/lib/faasd/postgresql/run

# For the password
TOKEN=$(head -c 12 /dev/urandom | shasum| cut -d' ' -f1)
echo $TOKEN > /var/lib/faasd/postgresql/postgres-passwd

# Set permissions
chown -R 1000:1000 /var/lib/faasd/postgresql/
```

Now restart faasd:

```
sudo systemctl daemon-reload \
  && sudo systemctl restart faasd
```

You should see the logs appear with:

```
sudo journalctl -t openfaas:postgresql

# Or, if it failed on:
sudo journalctl -t openfaas:faasd
```

This configuration only exposes port 5432 to your functions, your host will be `10.62.0.1` and the port will be 5432.

Adding a database - InfluxDB

You can add InfluxDB for access from your functions in the same way as Grafana.

Edit `/var/lib/faasd/docker-compose.yaml` and add:

```
  influxdb:
    image: docker.io/library/influxdb:1.8
    environment:
      - INFLUXDB_ADMIN_USER=admin
      - INFLUXDB_REPORTING_DISABLED=true
    volumes:
      # we assume cwd == /var/lib/faasd
      - type: bind
        source: ./influxdb/
        target: /var/lib/influxdb
    user: "1000"
    cap_add:
      - CAP_NET_RAW
    ports:
      - "8086:8086"
```

You can also customise the password using `INFLUXDB_ADMIN_PASSWORD`:

```
  influxdb:
    image: docker.io/library/influxdb:1.8
    environment:
      - INFLUXDB_DB=defaultdb
      - INFLUXDB_ADMIN_USER=admin
      - "INFLUXDB_ADMIN_PASSWORD=dfadbc1e54559ecc1f23672eeb320b09d1bc6f83"
      - INFLUXDB_USER=user
      - "INFLUXDB_USER_PASSWORD=dfadbc1e54559ecc1f23672eeb320b09d1bc6f83"
      - INFLUXDB_REPORTING_DISABLED=true
      - INFLUXDB_HTTP_AUTH_ENABLED=true
      - "INFLUXDB_BIND_ADDRESS=0.0.0.0:8086"
    volumes:
      # we assume cwd == /var/lib/faasd
      - type: bind
        source: ./influxdb/
        target: /var/lib/influxdb
    user: "1000"
    cap_add:
      - CAP_NET_RAW
    ports:
      - "8086:8086"
```

If you wish to only access the database from functions and the faasd server then change the following:

```
    ports:
      - "127.0.0.1:8086:8086"
```

Create a persistent data volume:

```
sudo mkdir -p /var/lib/faasd/influxdb
```

Make the folder readable by the user ID we are running as:

```
sudo chown 1000:1000 /var/lib/faasd/influxdb
```

Restart faasd:

```
sudo systemctl daemon-reload \
  && sudo systemctl restart faasd
```

Then check the logs for `influxdb`:

```
sudo journalctl -t openfaas:influxdb
```

Use the IP address `10.62.0.1` and port `8086` to connect to your database from your functions.

If you need to connect with the influxdb from your client, you can start an SSH tunnel and access it from localhost.

```
ssh -L 8086:127.0.0.1:8086 root@faasd-ip-address
```

Then access it with `influx -host 127.0.0.1 -port 8086`

Or if exposed via other interfaces:

```
influx -host 192.168.0.21 \
  -password "dfadbc1e54559ecc1f23672eeb320b09d1bc6f83" \
  -username admin
```

You can download the client for InfluxDB 1.8 here

Adding a cache - Redis

"Redis is an open source, in-memory data structure store, used as a database, cache, and message broker."

Redis can be used as a cache for your functions, here's a few scenarios where it may help:

- Debouncing - you may receive an event more than once, and by setting a value, you can avoid running an action more than once - for instance emailing a customer
- Caching expensive operations - if you look up data from a HTTP API, and it takes 3 seconds each time, but rarely changes, then you could benefit from caching the value and only incurring the cost on the first use
- Sharing state - Redis can also be used to share state between functions as a faster alternative to a database

You can add Redis for access from your functions in the same way as Grafana.

Edit `/var/lib/faasd/docker-compose.yaml` and add:

```
  redis:
    image: docker.io/library/redis:6.0.10-alpine
    volumes:
      # we assume cwd == /var/lib/faasd
      - type: bind
        source: ./redis/data
        target: /data
    cap_add:
      - CAP_NET_RAW
    command: /usr/local/bin/redis-server --appendonly yes
    user: "1000"
    ports:
      - "10.62.0.1:6379:6379"
```

The IP `10.62.0.1` means only functions and other core services can access the redis service.

> Note: If you decide to do this on an Internet-facing host, anyone will be able to access it that can connect to the host's IP address without a password, see the next section for how to secure it with authentication.

Create a persistent data volume:

```
sudo mkdir -p /var/lib/faasd/redis/data
```

Make the folder readable by the user ID we are running as:

```
sudo chown -R 1000:1000 /var/lib/faasd/redis/data
```

Restart faasd:

```
sudo systemctl daemon-reload \
  && sudo systemctl restart faasd
```

Then check the logs for `redis`:

```
sudo journalctl -t openfaas:redis
```

Use the IP address `10.62.0.1` and port `6379` to connect to your database from your functions.

Enabling password authentication for Redis

You can add a password for Redis through a command-line argument, or by creating a config file and mounting it into the container.

For the command-line option, add `--requirepass SECRET` to the `docker-compose.yaml` file and restart faasd:

```
    command: /usr/local/bin/redis-server --appendonly yes --requirepass SECRET
```

Alternatively, you can create a config file and mount it into the container.

Create redis.conf:

```
mkdir -p redis
echo "requirepass SECRET" > redis/redis.conf
chown -R 1000:1000 redis
```

```
    command: /usr/local/bin/redis-server /usr/local/etc/redis/redis.conf --appendonly yes
```

```
    volumes:
      # we assume cwd == /var/lib/faasd
      - type: bind
        source: ./redis/redis.conf
        target: /usr/local/etc/redis/redis.conf
```

If you want to make redis accessible from any interface, you can remove the IP prefix:

```
    ports:
      - "6379:6379"
```

If you wish to access redis from external hosts, you may also need to update the bind instruction in redis.conf:

```
# bind 127.0.0.1
bind 0.0.0.0
```

To find out more, see the Redis documentation on security.

Chapter 9

Scheduling invocations and background jobs

You can use Cron to schedule invocations of your functions. There are probably three main options:

1) Use the OpenFaaS solution for Cron called the "cron-connector"
2) Install cron on your system, and set up a cron schedule like you would for anything else
3) Use a third party SaaS service to invoke your function over an inlets tunnel or public URL

The recommended way to invoke functions using a schedule or timer is to use the OpenFaaS

You can also trigger functions from other event sources, see also: OpenFaaS docs: triggers

Scheduled invocations with cron-connector

The cron-connector is an add-on for OpenFaaS which is not part of the default installation. You can add it by editing `docker-compose.yaml` and restarting faasd.

Edit `/var/lib/faasd/docker-compose.yaml` and add:

```yaml
  cron-connector:
    image: "ghcr.io/openfaas/cron-connector:latest"
    environment:
      - gateway_url=http://gateway:8080
      - basic_auth=true
      - secret_mount_path=/run/secrets
    volumes:
      # we assume cwd == /var/lib/faasd
      - type: bind
        source: ./secrets/basic-auth-password
        target: /run/secrets/basic-auth-password
      - type: bind
        source: ./secrets/basic-auth-user
        target: /run/secrets/basic-auth-user
    cap_add:
      - CAP_NET_RAW
    depends_on:
      - gateway
```

Now restart faasd:

```
sudo systemctl daemon-reload && \
  sudo systemctl restart faasd
```

You can then assign schedules to functions using a cron expression.

The cron.guru website is useful for writing an expression.

- For every 5 minutes: `*/5 * * * *`
- For every 10 minutes: `*/5 * * * *`
- Run at midnight: `0 0 * * *`
- At midnight on Sundays: `@weekly`

You may also want to read the notes from Wikipedia: Cron.

Most of the time, you will be editing a `stack.yml` file to enable a schedule:

```
functions:
  nodeinfo:
    image: functions/nodeinfo
    annotations:
      topic: cron-function
      schedule: "*/5 * * * *"
```

But for quick testing, you can also deploy a function directly from the store:

```
faas-cli store deploy nodeinfo \
  --annotation schedule="*/5 * * * *" \
  --annotation topic=cron-function
```

Notice that both an annotation called `schedule` and a `topic` of `cron-function` is required.

You can view the logs of the connector and see it invoking functions:

```
journalctl -t openfaas:cron-connector -f

2022/08/15 14:48:21 Version: 2268390dbd208f775ddef6810cd232b2bcb839fd        Commit: 0.6.0
2022/08/15 14:48:21 Gateway URL: http://gateway:8080
2022/08/15 14:48:21 Async Invocation: false
2022/08/15 14:48:21 Rebuild interval: 10s        Rebuild timeout: 5s
2022/08/15 14:48:31 Added: nodeinfo.openfaas-fn [*/1 * * * *]
2022/08/15 14:49:00 Response: nodeinfo [200] (229ms)
```

You'll also see the invocation count increasing when viewed with `faas-cli`, the Prometheus metrics and the Grafana dashboards.

There will be an empty request body passed to your function when it is invoked through a cron schedule. If an input is required, you could create a wrapper function, which passes the correct body, or looks it up before invoking the target.

Scheduled invocations with system cron

If you want to use cron installed on your system, first add the cron package and then enable it with systemd:

```
sudo apt install cron
sudo systemctl enable cron
sudo systemctl start cron
```

Now edit your crontab and schedule an invocation of your function every 5 minutes:

```
crontab -e
```

Here's how you can invoke nodeinfo on a schedule:

```
*/5 * * * * /usr/local/bin/faas-cli invoke nodeinfo <<< ""
```

This approach requires you to log into your server and cannot be managed as annotations placed on a function. On the plus side, you can easily pass a body by creating a shell script and setting that up to be invoked instead of the faas-cli directly.

Chapter 10

nats-connector

The nats-connector can be used to trigger functions from events published to NATS pub/sub Subjects.

Use this connector only if you have existing NATS pub/sub Subjects which you need to integrate with. Otherwise you should use the built-in asynchronous invocation mechanism in OpenFaaS that we covered in an earlier chapter.

Add the following to `docker-compose.yaml` and restart faasd.

```yaml
  nats-connector:
    image: ghcr.io/openfaas/nats-connector:latest
    environment:
      broker_host: "nats"
      upstream_timeout: "60s"
      gateway_url: "http://gateway:8080"
      topics: "topic1,topic2,"
      print_response: "true"
      print_body_response: "true"
      basic_auth: "true"
      secret_mount_path: "/run/secrets/"
      topic_delimiter: ","
      asynchronous_invocation: "false"
    volumes:
      - type: bind
        source: ./secrets/basic-auth-user
        target: /run/secrets/basic-auth-user
      - type: bind
        source: ./secrets/basic-auth-password
        target: /run/secrets/basic-auth-password
    cap_add:
      - CAP_NET_RAW
    depends_on:
      - nats
      - gateway
```

You can edit the `topics` environment variable and pass a list of NATS Subjects separated by `commas`, i.e. `topic1,topic2,`

Then deploy a function and give it a matching NATS topic:

```
faas-cli deploy --name echo \
  --image ghcr.io/openfaas/alpine:latest \
  --annotation topic=topic1 \
  --env write_debug=true \
  --fprocess=cat
```

The above will echo any input it receives back to the receiver.

To connect to NATS, you will need to install the NATS CLI, and then publish a message. The `docker-compose.yaml` file should also be edited to expose the `nats` service on the OpenFaaS bridge adapter, or 127.0.0.1.

```
  ports:
      - "127.0.0.1:4222:4222"
```

Restart the `faasd` service after editing the file.

Download a recent version using `arkade get nats` or find the correct binary on the releases page of the NATS CLI.

```
$ nats context add nats \
  --server 127.0.0.1:4222 \
  --description "faasd NATS" --select

NATS Configuration Context "nats"

  Description: faasd NATS
  Server URLs: 127.0.0.1:4222
         Path: /home/pi/.config/nats/context/nats.json
   Connection: OK
```

Then trigger the function:

```
$ nats pub topic1 test message
21:45:38 Published 12 bytes to "topic1"
```

And check the output:

```
pi@k4s-1:~ $ sudo journalctl -t openfaas:nats-connector
=======================================================
NATS Connector for OpenFaaS
=======================================================

2021/02/22 21:35:06 Configured topics: [topic1 topic2]
2021/02/22 21:35:06 Binding to topic: "topic1"
2021/02/22 21:35:06 Binding to topic: "topic2"
2021/02/22 21:35:06 Subscription: topic1 ready
2021/02/22 21:35:06 Subscription: topic2 ready
2021/02/22 21:36:27 Topic: topic1, message: "test"
2021/02/22 21:45:38 Invoke function: echo
2021/02/22 21:45:38 connector-sdk got result: [200] topic1 => echo (12) bytes
```

Check the logs of the function to see the invocation:

```
faas-cli logs echo
```

```
2021-02-22T21:40:54Z 2021/02/22 21:40:54 Path  /
2021-02-22T21:40:54Z 2021/02/22 21:40:54 Duration: 0.004409s
```

Chapter 11

CI/CD

CI/CD stands for Continuous Integration and Continuous Deployment. They are used together so much that the meaning has blurred.

Continuous Integration - tends to refer to an automated build attached to your source control management (SCM) system such as Git, whether on premises with GitLab, BitBucket, or in the cloud with GitHub. Most CI systems have the notion of a job, which runs upon each commit from your SCM. The job may be made up of a number of steps and then called a pipeline. A Pipeline usually checks out a repository, installs dependencies, runs unit tests and then creates an artifact at the end. CI may also run whenever you create a release of your project, and then go and create artifacts for deployment at a later date. CI systems don't usually do deployments.

Continuous Deployment - is the act of deploying artifacts when made available from a CI job or pipeline. This could range from a post-build job being triggered that rolls out any available version, to a so called "canary" deployment being created which receives some traffic before being automatically promoted to the main version users see.

During development, and in some production systems you may be able to combine CI and CD and therefore keep things simple.

The OpenFaaS CLI has three commands that you will need if your CI system supports Docker:

- `faas-cli build` - create a container image for a function using Docker
- `faas-cli push` - push the container image from the function to a remote registry
- `faas-cli deploy` - deploy the function using the container image and the gateway's REST API

These three commands can be combined into one: `faas-cli up`

Some teams prefer to use another container builder like Kaniko, or a cloud build service. In this instance, you can get the `faas-cli` to create a build context and a Dockerfile that can be built by any container builder.

To see how it works, you can just add the `--shrinkwrap` flag to the build command:

```
faas-cli new --lang node16 get-customer
faas-cli build --shrinkwrap -f get-customer.yml
```

Now see that you have a complete build context, where the template we learned about in an earlier section has been placed into the folder, with your customisations from the `get-customer` folder over the top.

```
tree build

build
└── get-customer
    ├── Dockerfile
    ├── function
    │   ├── handler.js
    │   └── package.json
    ├── index.js
    ├── package.json
    └── template.yml
```

So you could zip up the `build/get-customer` folder and then use an alternative container builder or a SaaS builder.

Deploy from GitHub Actions and ghcr.io

The following example remains the same whether using OpenFaaS with faasd or Kubernetes.

You can use the following example to build, publish and deploy multi-arch functions to your cluster.

- Create a new GitHub repository
- Create your functions and call your YAML file `stack.yml`
- Enable GitHub Actions for your account or repository
- On the settings page create the following secrets
- Name: `OPENFAAS_URL`, value: the gateway's public HTTPS URL or inlets tunnel URL
- Name: `OPENFAAS_PASSWORD`, value: faasd's basic auth password

Note that GitHub actions provides a `GITHUB_TOKEN` which can be used for authenticating to GitHub Container Registry (ghcr). If you are using a self-hosted registry, or the Docker Hub, you can create a `DOCKER_PASSWORD` secret instead.

- Create the following file as `.github/workflows/build.yml`

This is the top of the file which triggers a build named `build` upon every PR and push to a remote branch.

```
name: build

on:
  push:
    branches:
      - '*'
  pull_request:
    branches:
      - '*'
```

Then set the permissions for the built-in `GITHUB_TOKEN`:

```
permissions:
  actions: read
  checks: write
  contents: read
  packages: write
```

We need `packages: write` to publish to ghcr.io.

Then provide the steps for the job in the same file

```
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@master
        with:
          fetch-depth: 1
      - name: Get faas-cli
        run: curl -sLSf https://cli.openfaas.com | sudo sh
      - name: Set up QEMU
        uses: docker/setup-qemu-action@v1
      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v1
      - name: Get TAG
        id: get_tag
        run: echo ::set-output name=TAG::latest-dev
      - name: Get Repo Owner
        id: get_repo_owner
        run: >
          echo ::set-output name=repo_owner::$(echo ${{ github.repository_owner }} |
          tr '[:upper:]' '[:lower:]')
      - name: Login to Container Registry
        uses: docker/login-action@v1
        with:
          username: ${{ github.repository_owner }}
          password: ${{ secrets.GITHUB_TOKEN }}
          registry: ghcr.io
      - name: Publish functions
        run: >
          OWNER="${{ steps.get_repo_owner.outputs.repo_owner }}"
          TAG="latest"
          faas-cli publish
          --extra-tag ${{ github.sha }}
          --platforms linux/arm/v7
      - name: Login
        run: >
          echo ${{secrets.OPENFAAS_PASSWORD}} |
          faas-cli login --gateway ${{secrets.OPENFAAS_URL}} --password-stdin
      - name: Deploy
        run: >
          OWNER="${{ steps.get_repo_owner.outputs.repo_owner }}"
          TAG="${{ github.sha }}"
          faas-cli deploy --gateway ${{secrets.OPENFAAS_URL}}
```

The first few steps set up Docker with buildx, so that it can cross-compile containers for different systems, this can be skipped if you only want to deploy to cloud or Intel-compatible faasd instances.

The next steps log into the ghcr.io registry using your personal access token, run `faas-cli publish` which builds and pushes a multi-arch image, and then logs into your remote gateway and does a deployment using `faas-cli login` and `faas-cli deploy`.

The `--platforms` flag should be customised to make your build more efficient. It is currently building for:

- `linux/amd64` - regular PCs and cloud
- `linux/arm/v7` - Raspberry Pi OS
- `linux/arm64` - 64-bit ARM servers or Ubuntu running on Raspberry Pi

You'll notice that the last step in the file does a deployment. If you don't want builds to get deployed automatically, then you can put this into a separate file that gets run only when you do a release in the GitHub repository, splitting out CI and CD steps.

```
      - name: Login
        run: >
          echo ${{secrets.OPENFAAS_PASSWORD}} |
          faas-cli login --gateway ${{secrets.OPENFAAS_URL}} --password-stdin
      - name: Deploy
        run: >
          OWNER="${{ steps.get_repo_owner.outputs.repo_owner }}"
          TAG="${{ github.sha }}"
          faas-cli deploy --gateway ${{secrets.OPENFAAS_URL}}
```

Edit stack.yml so that the GitHub Action can inject the OWNER and TAG variables:

```
version: 1.0
provider:
  name: openfaas
  gateway: http://127.0.0.1:8080

functions:
  profile-page:
    lang: node16
    handler: ./profile-page
    image: ghcr.io/${OWNER:-alexellis}/profile-page:${TAG:-latest}
```

When no environment variables are given, the text will be: `ghcr.io/alexellis/profile-page:latest` and when the action runs, it will generate a string more like: `ghcr.io/alexellis/profile-page:af6ca8e916752513da15ce802828`

The SHA suffix, can be used to restore a prior build in the event that our new function doesn't work correctly. It also helps us understand what's deployed later on:

```
$ faas-cli list --verbose

Function                        Image
profile-page                    ghcr.io/alexellis/profile-page:af6ca8e916752513da15ce80282887a3f3223e
```

Now run the following:

```
git add .
git commit
git push origin master
```

And check the Actions tab of your repo. If everything worked as expected, and you had no typos or missing secrets, then you will see the functions in your stack.yml file deployed directly to your faasd instance.

If you decide to split up the build and publish commands, then simply remove the "Deploy" step from your build.yml file and create another identical one for the publish step, with this trigger instead:

```
name: publish

on:
  push:
    tags:
      - '*'
```

It will run every time you create a new release on GitHub.

If you want to see a complete example that you can copy and paste, review my sample repo: alexellis/faasd-example.

Follow an in-depth tutorial

You can deploy an openfaas function from GitHub by using the details in this post: Build and deploy OpenFaaS functions with GitHub Actions

If your faasd service is running at home, or behind NAT, then you can use a tunnel using inlets to access the OpenFaaS REST API.

Deploy with multi-arch

multi-arch is only required if deploying to a Raspberry Pi or ARM64 server. Most of the OpenFaaS templates have support for this.

The only alteration you need is to run `faas-cli publish` instead of `faas-cli build` and `faas-cli push`. The publish command uses Docker's buildx project and the directives in the Dockerfiles we maintain to publish images for various architectures.

The easiest CI/CD platform for multi-arch is GitHub Actions.

Further resources for CI/CD

For further resources, see the OpenFaaS Docs on CI/CD which has examples for GitLab, GitHub Actions, Travis and Jenkins.

Chapter 12

Advanced topics, troubleshooting and tuning

Tuning timeouts

Timeouts for faasd can be configured in three places:

- faasd-provider (the systemd service that invokes functions)
- gateway - the core service defined in docker-compose.yaml - see environment variables `upstream_timeout`, `write_timeout` and `read_timeout`
- the function's stack.yml file

For functions using the classic watchdog see: Lab 8 - Advanced feature - Timeouts

For newer templates see: Golang function that runs for a long time

Asynchronous invocations

Functions can be invoked asynchronously by changing their URL as follows:

```
curl -d Message http://127.0.0.1:8080/function/figlet/
```

To:

```
curl -d Message http://127.0.0.1:8080/async-function/figlet/
```

> Note the `X-Call-Id` header that you receive. You can use this to track the response.

In the former, you will get the response immediately, in the later, you will need to check the logs of the queue worker core service for the result.

```
sudo journalctl -t openfaas:queue-worker
```

Alternatively, you can specify a callback URL to receive the result, and the `X-Call-Id` header which can be used to match up with the request.

```
curl -d Message http://127.0.0.1:8080/async-function/figlet/ \
  --header "X-Callback-Url: http://example.com/api/callback"
```

See the OpenFaaS docs: Asynchronous Functions

Reclaiming space

Over time, the disk of your faasd instance may become full with the the images of older functions that you have deployed during testing and development.

You can remove these using the `ctr` command, but need to be careful not to remove the images of any running functions. This operation can be run whilst serving traffic, and there is no need to stop any of your functions or faasd itself.

```
sudo ctr -n openfaas-fn c ls

CONTAINER    IMAGE                                            RUNTIME
env          docker.io/functions/alpine:latest-armhf          io.containerd.runc.v2
nodeinfo     docker.io/functions/nodeinfo-http:latest-armhf   io.containerd.runc.v2
chaos-fn     docker.io/alexellis2/chaos-fn:0.1.1              io.containerd.runc.v2
```

Only the latest version in-use of an image will show here.

Save the following script as `prune.sh`:

```bash
#!/bin/bash

# Capture the images associated with a
sudo ctr -n openfaas-fn c ls | tr -s ' ' | \
  cut -d " " -f 2 | grep -v "IMAGE" > running.txt

# Now remove all images, excluding the ones in the running.txt file:
IMAGES=$(sudo ctr -n openfaas-fn image ls -q)
set -f
array=(${IMAGES// / })
for i in "${array[@]}"; do
  if ! grep -qxFe "$i" running.txt; then
    echo Deleting $i
    # sudo ctr -n openfaas-fn image rm $i
  fi
done
```

This script can be run with cron installed on your host, on a daily schedule.

Test it in the following way by deploying figlet, then removing it:

```bash
# For PC / cloud:
faas-cli store deploy figlet

# For RPi:
faas-cli store deploy --platform armhf figlet

# Then remove it:
faas-cli remove figlet

# Now prune the images:

./prune.sh

Deleting docker.io/functions/figlet:latest-armhf
```

You may also want to delete older images of the OpenFaaS core services in the `openfaas` namespace.

Exploring faasd resource consumption

You can find out how much CPU and memory is being used by faasd with standard Linux commands included with your Operating System. The following example is from my Raspberry Pi 3 which runs the "Treasure Trove" a.k.a. Sponsors Portal.

You can find out how much free memory you have on the faasd host by connecting with SSH and running the following command:

```
$ free -h
              total        used        free      shared  buff/cache   available
Mem:          926Mi       328Mi        39Mi        34Mi       557Mi       578Mi
Swap:          99Mi        20Mi        79Mi
```

The `-h` flag stands for human-readable format.

In this example, you can see that the used memory is 328MB vs 926MB total, meaning there is 598MB remaining, around 65% free.

To find out CPU usage on the faasd host you can run the `uptime` command.

```
$ uptime

 11:10:02 up 85 days, 15:25,  1 user,  load average: 0.08, 0.02, 0.05
```

The load average on the CPU is displayed as a number related to how many CPU cores you have. So if the host has 4x CPU cores, and the number is 4.0, then all four of the cores are working at roughly 100% load. In this instance the system is mostly idle and under-utilized.

For a more visual overview of memory and CPU in one place, you can run `htop`, which may be preinstalled, otherwise it is available via `sudo apt install htop`. The view in htop can be sorted by CPU or Memory usage by clicking on its column.

In the example, the Raspberry Pi has so much spare CPU and memory, that htop itself is showing up as the top consumer of CPU.

Find out a function's memory and CPU usage

You can find out the memory used by a function using the `ctr` command, which is the CLI for containerd.

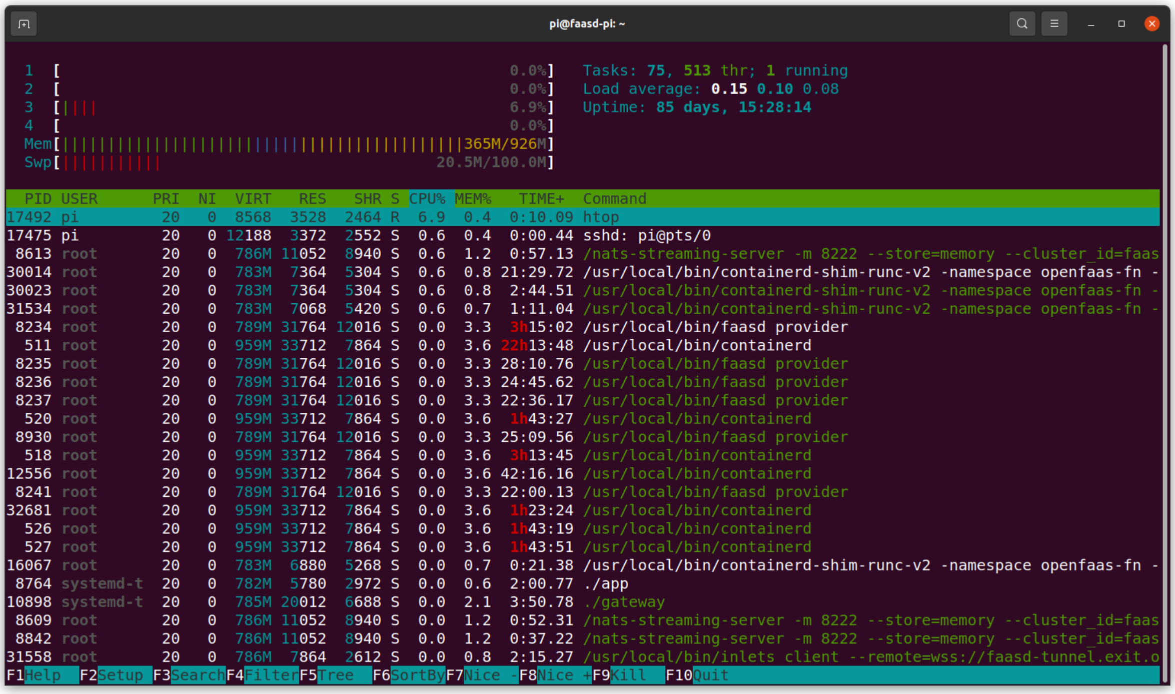
Deploy a function from the store:

```
faas-cli store deploy figlet
faas-cli store deploy figlet --platform armhf # For Raspberry Pi
```

You will be able to find its statistics with the following command:

```
sudo ctr -n openfaas-fn task metrics figlet
ID        TIMESTAMP
figlet    2021-02-11 10:49:10.014877816 +0000 UTC

METRIC                   VALUE
memory.usage_in_bytes    5779456
memory.limit_in_bytes    8796093018112
memory.stat.cache        3743744
```



```
pi@faasd-pi: ~

  1  [                                                0.0%]   Tasks: 75, 513 thr; 1 running
  2  [                                                0.0%]   Load average: 0.15 0.10 0.08
  3  [||||                                            6.9%]   Uptime: 85 days, 15:28:14
  4  [                                                0.0%]
  Mem[|||||||||||||||||||||||||||||||||||||||||||||365M/926M]
  Swp[||||||||||                                20.5M/100.0M]

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
17492 pi         20   0  8568  3528  2464 R  6.9  0.4  0:10.09 htop
17475 pi         20   0 12188  3372  2552 S  0.6  0.4  0:00.44 sshd: pi@pts/0
 8613 root       20   0  786M 11052  8940 S  0.6  1.2  0:57.13 /nats-streaming-server -m 8222 --store=memory --cluster_id=faas
30014 root       20   0  783M  7364  5304 S  0.6  0.8 21:29.72 /usr/local/bin/containerd-shim-runc-v2 -namespace openfaas-fn -
30023 root       20   0  783M  7364  5304 S  0.6  0.8  2:44.51 /usr/local/bin/containerd-shim-runc-v2 -namespace openfaas-fn -
31534 root       20   0  783M  7068  5420 S  0.6  0.7  1:11.04 /usr/local/bin/containerd-shim-runc-v2 -namespace openfaas-fn -
 8234 root       20   0  789M 31764 12016 S  0.0  3.3  3h15:02 /usr/local/bin/faasd provider
  511 root       20   0  959M 33712  7864 S  0.0  3.6 22h13:48 /usr/local/bin/containerd
 8235 root       20   0  789M 31764 12016 S  0.0  3.3 28:10.76 /usr/local/bin/faasd provider
 8236 root       20   0  789M 31764 12016 S  0.0  3.3 24:45.62 /usr/local/bin/faasd provider
 8237 root       20   0  789M 31764 12016 S  0.0  3.3 22:36.17 /usr/local/bin/faasd provider
  520 root       20   0  959M 33712  7864 S  0.0  3.6  1h43:27 /usr/local/bin/containerd
 8930 root       20   0  789M 31764 12016 S  0.0  3.3 25:09.56 /usr/local/bin/faasd provider
  518 root       20   0  959M 33712  7864 S  0.0  3.6  3h13:45 /usr/local/bin/containerd
12556 root       20   0  959M 33712  7864 S  0.0  3.6 42:16.16 /usr/local/bin/containerd
 8241 root       20   0  789M 31764 12016 S  0.0  3.3 22:00.13 /usr/local/bin/faasd provider
32681 root       20   0  959M 33712  7864 S  0.0  3.6  1h23:24 /usr/local/bin/containerd
  526 root       20   0  959M 33712  7864 S  0.0  3.6  1h43:19 /usr/local/bin/containerd
  527 root       20   0  959M 33712  7864 S  0.0  3.6  1h43:51 /usr/local/bin/containerd
16067 root       20   0  783M  6880  5268 S  0.0  0.7  0:21.38 /usr/local/bin/containerd-shim-runc-v2 -namespace openfaas-fn -
 8764 systemd-t  20   0  782M  5780  2972 S  0.0  0.6  2:00.77 ./app
10898 systemd-t  20   0  785M 20012  6688 S  0.0  2.1  3:50.78 ./gateway
 8609 root       20   0  786M 11052  8940 S  0.0  1.2  0:52.31 /nats-streaming-server -m 8222 --store=memory --cluster_id=faas
 8842 root       20   0  786M 11052  8940 S  0.0  1.2  0:37.22 /nats-streaming-server -m 8222 --store=memory --cluster_id=faas
31558 root       20   0  786M  7864  2612 S  0.0  0.8  2:15.27 /usr/local/bin/inlets client --remote=wss://faasd-tunnel.exit.o
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

Figure 12.1: Example output from htop

```
cpuacct.usage           3365355140
cpuacct.usage_percpu    [855012003 70783027 575087137 1864472973]
pids.current            10
pids.limit              0
```

Now, to get a human-readable understanding of the memory usage in bytes, use jq and the JSON output format:

```
export NAME=figlet
echo $(sudo ctr -n openfaas-fn task metrics $NAME --format json | \
  jq '.memory.usage.usage/1024/1024' )MB

5.67578125MB
```

You can also view total CPU time consumed by this task (process) and any sub-processes it may have created by viewing the `cpuacct.usage` field which is shown in nanoseconds (ns). There are 1,000,000.00ns within a millisecond.

For the most detailed view of a container's resource consumption see `ctr -n openfaas-fn task metrics NAME --format json`.

Memory limits for functions

faasd also supports setting a maximum memory limit for a function. This can be useful when you want to set an upper boundary, however you should also bear in mind what happens when the limit is exceeded.

When the memory limit is exceeded, the function will be killed. Upon the next invocation by a user, the function will be scaled to 1/1 replicas again.

Example:

```
functions:
  figlet:
    skip_build: true
    image: functions/figlet:latest
    limits:
      memory: 20Mi
```

See also: Function: Memory/CPU limits

Multiple namespaces for functions

By default, faasd creates all your functions within the `openfaas-fn` namespace, however it is possible to use other namespaces for logical separation.

> Note: using different namespaces does not increase isolation or make faasd suitable for multi-tenancy.

To create a new namespace, log into the faasd host and create the namespace, then label it:

```
sudo ctr namespace create staging
sudo ctr namespace label staging openfaas=true
```

You can now access it via:

```
faas-cli store deploy figlet --namespace staging

faas-cli list --namespace staging
```

The short flag `-n` can also be used instead of `--namespace`.

```
faas-cli describe figlet --namespace staging
```

To invoke the function, add a suffix to its URL:

```
echo "hello" | faas-cli invoke figlet -n staging

curl -i -d "world" http://127.0.0.1:8080/function/figlet.staging
curl -i -d "async" http://127.0.0.1:8080/async-function/figlet.staging
```

After removing any functions in the namespace, you can remove it by running:

```
faas-cli remove -n staging figlet

sudo ctr namespaces remove staging
```

If you run into any issues with removing the namespace, check for tasks, containers, images and snapshots and remove those with `ctr`.

Upgrading and updating faasd

There are several parts of faasd that you should upgrade over time.

1) The `faasd` binary which makes up the faasd and faasd-provider service
2) The containers defined within the `docker-compose.yaml` file
3) Any configuration files that are referenced in the faasd repo such as prometheus.yaml
4) Any add-ons or bonuses that you bought with this eBook like the Grafana dashboard

To upgrade faasd you can simply replace the faasd binary with a newer one from the releases page on GitHub

```
systemctl stop faasd-provider
systemctl stop faasd

# Replace /usr/local/bin/faasd with the desired release

# Replace /var/lib/faasd/docker-compose.yaml with the matching version for
# that release.
# Remember to keep any custom patches you make such as exposing additional
# ports, or updating timeout values

systemctl start faasd
systemctl start faasd-provider
```

You could also perform this task over SSH, or use a configuration management tool like Puppet, Chef or Ansible.

As an alternative, you can also delete and re-create your faasd host, however, if you are using Caddy or Let's Encrypt for free TLS certificates, then you may hit the rate-limits. Let's Encrypt only allows a set number of certificates per account each week.

The individual components that make up the docker-compose.yaml file may also change over time, as the project releases new features and bug fixes.

From time to time, you should update the `/var/lib/faasd/docker-compose.yaml` file with any newer versions of container images used by faasd.

For instance, by checking the releases page for the openfaas gateway, you'll be able to find out whether there is a new tag available. Let's say your docker-compose.yaml file had `0.20.7` for the gateway, and you see that `0.20.8` is available, you'd edit it as follows:

```
  gateway:
    image: ghcr.io/openfaas/gateway:0.20.7
```

After

```
  gateway:
    image: ghcr.io/openfaas/gateway:0.20.8
```

After each update, you will need to run `systemctl restart faasd`.

Troubleshooting

Finding logs with journalctl

The `journalctl` command can be used to find logs from systemd services such as faasd and the provider. Any functions you deploy will also redirect their logs here.

By default `journalctl` will display all logs available, and the "g" command goes to the top of the logs, and the "G" command goes to the end of the logs. In this view, you can search for text by typing in `/somestring`.

journalctl can be piped to a file for sharing with an OpenFaaS expert, or for saving into a GitHub issue. It can also be used with grep to filter errors or key information.

For instance, to find all logs with the text `Supervisor` from the `faasd` service:

```
sudo journalctl -u faasd|grep "Supervisor"
Feb 06 09:13:58 faasd-pi: 2021/02/06 09:13:58 Supervisor created in: 5.429711ms
Feb 06 09:15:02 faasd-pi: 2021/02/06 09:15:02 Supervisor init done in: 1m3.517271689s
```

The following flags can make the tool even more useful:

- `--follow`/`-f` - to follow new lines as they appear, live
- `--lines N` - show the last N lines
- `--since` - show lines since a given time or date

Logs for faasd itself

faasd is the supervisor for the core services defined in docker-compose.yaml, to get its logs, run:

```
sudo journalctl -u faasd
```

faasd-provider is the supervisor for functions, to get its logs run:

```
sudo journalctl -u faasd-provider
```

Logs for the core services

Core services as defined in the docker-compose.yaml file are deployed as containers by faasd.

View the logs for a component by giving its NAME:

```
journalctl -t openfaas:NAME

journalctl -t openfaas:gateway

journalctl -t openfaas:queue-worker
```

You can also use `-f` to follow the logs, or `--lines` to tail a number of lines, or `--since` to give a timeframe.

Logs for functions

You can get function logs with `faas-cli logs NAME`. The `--since` flag is also useful to get history for instance `--since 60m`. This command can be run remotely from your client.

All logs are stored in the system journal, so you can at any time use `journalctl` for a more powerful query. This command needs to be run on the faasd server.

```
# All logs recorded. Use g and G to jump between the top and bottom
sudo journalctl -t openfaas-fn:NAME

# Since a given gate
sudo journalctl -t openfaas-fn:NAME --since DATE

# Print the last NUMBER of lines
sudo journalctl -t openfaas-fn:NAME --lines NUMBER

# Tail the last few lines
sudo journalctl -t openfaas-fn:NAME -f

# Get logs without non-interactively
sudo journalctl -t openfaas-fn:NAME --no-pager
```

After rebooting my functions are unavailable or not ready

You may have noticed that when you reboot your faasd server, that your functions are scaled to `0/1` replicas. This is working as designed, and faasd will recreate them upon first access.

There is no need to scale up a function manually, however if you really want to, you can. Why not install cron on your system and setup a `@reboot` task to invoke them, causing them to scale up, for example?

```
@reboot */5 * * * * /usr/local/bin/faas-cli invoke nodeinfo <<< ""
```

Chapter 13

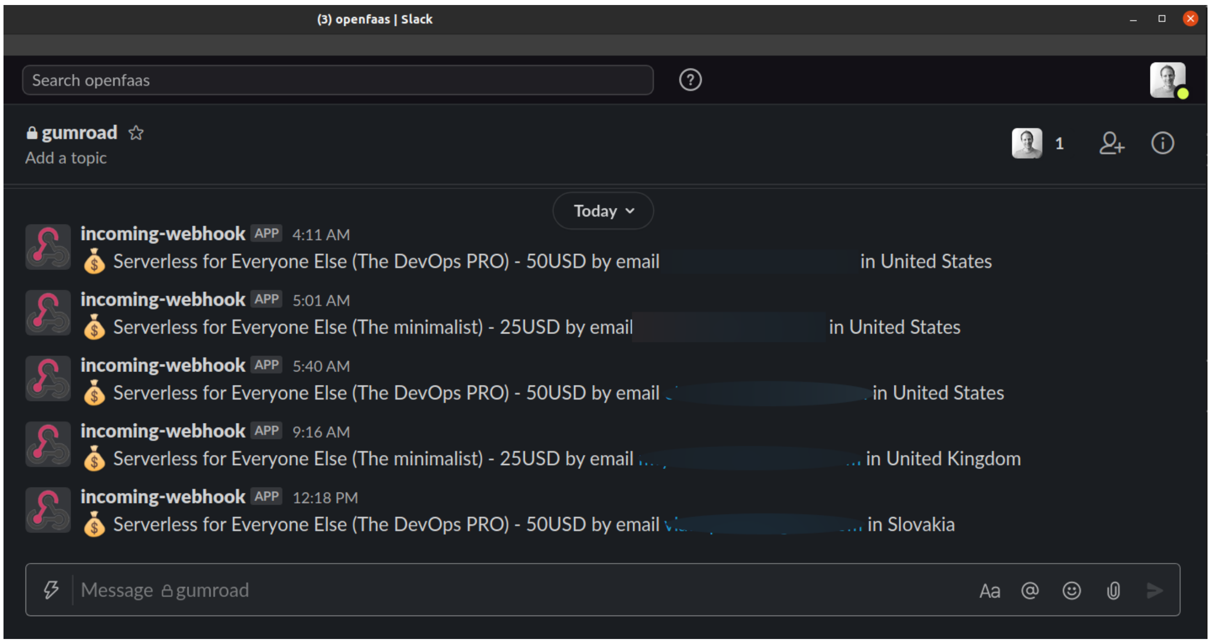
Wrapping up and taking things further

Now that you've been through the exercises and had a chance to customise them, it's over to you to find your own use-cases and problems to solve with functions.

Functions are snippets of code, that do one task or job and do it really well. When you combine them together you can create powerful workflows and extend proprietary systems and SaaS platforms.

Here is a quick case-study of how faasd itself helped sell its own eBook:

> The weekend after launching the eBook, my video workshop was ready for purchase on the 99 USD tier, but to get some early reviews and to generate more sales I ran a promotion where you could get a link to the video for free, by paying 50 USD. This turned out to be much more popular than expected, and I had to send out a lot of emails. By the end of the promotion late on Sunday night, I spent an hour to write some code to send an email using AWS' Simple Email Service (SES) and managed to automate the whole task. The first function gets a webhook from Gumroad, and sends it on to my Discord or Slack channel. If the tier is 50 USD, it will then call the "upgrade" function which composes an email and sends it to the customer with the video link.



incoming-webhook APP 4:11 AM
💰 Serverless for Everyone Else (The DevOps PRO) - 50USD by email [redacted] in United States

incoming-webhook APP 5:01 AM
💰 Serverless for Everyone Else (The minimalist) - 25USD by email [redacted] in United States

incoming-webhook APP 5:40 AM
💰 Serverless for Everyone Else (The DevOps PRO) - 50USD by email [redacted] in United States

incoming-webhook APP 9:16 AM
💰 Serverless for Everyone Else (The minimalist) - 25USD by email [redacted] in United Kingdom

incoming-webhook APP 12:18 PM
💰 Serverless for Everyone Else (The DevOps PRO) - 50USD by email [redacted] in Slovakia

Figure 13.1: Gumroad to Discord or Slack to integration

You can view part of the code for the integration in my GitHub Account: gumroad-sales-forwarder

Use-cases and ideas

Over 4 years there have been hundreds of blog posts, talks given and videos recorded. This should give you inspiration as you start writing and deploying your own functions and endpoints.

- OpenFaaS blog - great for finding use-cases, case-studies and news
- Community file - find blog posts and conferences talks going back to 2016
- Adopters file - see who's adopting OpenFaaS at work, and what they are doing with it. Send your own PR if you deploy something you find useful

Scaling-up

faasd is designed to run only one replica of a function, so if you feel like you are needing to scale up, there are a few options.

- vertical scaling - if your functions are CPU-bound, provision a larger VM with more cores, or a faster clock-speed. If they are mainly memory-bound, then provision a host with more RAM allocated to it
- horizontal scaling - deploy the same function more than once, with different names, and load-balance it, this is the equivalent of scaling replicas in OpenFaaS on Kubernetes
- multiple instances of faasd - you can deploy faasd more than once and put a load-balancer in front of it using software like HAProxy or a cloud-based load-balancer
- throttle invocations - using the queue-worker and asynchronous invocations you can throttle how many requests are in-flight at any one time for each function, this could provide a simpler alternative than the other options

And if you have explored all of the above options, moving your functions to OpenFaaS on Kubernetes may provide a suitable alternative. Kubernetes has clustering built-in, so you can add additional nodes when you run out of CPU or RAM, and can have as many replicas of a function as you want, within reason.

Getting help and reaching out

You can follow OpenFaaS on Twitter to keep up with news and updates.

The OpenFaaS community page has resources for connecting with the community, and for contributing. At time of writing, we hold a weekly community call and you're welcome to join - to ask questions, and to tell us about your experiences so far.

Training, consultation and sponsorships

If you would like to learn how OpenFaaS could be used at your company, feel free to reach out on the Support page.

If you did decide to use OpenFaaS or faasd, or were already doing so, then it makes business sense for you or your company to become a GitHub Sponsor, the cost is more affordable than most people assume and means that the project can continue to be developed and maintained for your use.

If you'd like to hear from me every week on open source, business and community, then I write a subscription-only email via GitHub Sponsors.

faasd videos

In addition to the video workshop, there are two videos that you may enjoy to get a bit more context and to understand use-cases better.

After publishing the eBook I wanted to create some more value for users, and to explain use-cases in more detail.

- Exploring Serverless use-cases with David McKay (Rawkode) and Alex Ellis

My 10 minute overview from KubeCon exploring the need, and showing how faasd differs from OpenFaaS on Kubernetes

- Meet faasd. Look Ma' No Kubernes! - Alex Ellis, OpenFaaS Ltd

A 15 minute overview of installing faasd in Multipass with cloud-init, finding the password and opening the OpenFaaS UI

- faasd walk-through with cloud-init and Multipass

A live-stream with faasd and inlets with David McKay exploring the workflow from deploying faasd, to writing a function and adding an additional service like InfluxDB.

- Technical deep dive live-stream into faasd and inlets