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 node12

Name:              node12
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 node12 template.

Where should you publish container images?

Every OpenFaaS function is packaged as a container image before being deployed, so before you get started, you will need to choose a container registry.

It's recommended that you use a container registry service to store your functions. The Docker Hub is the easiest to get started with, and you can push and pull functions within some basic limits for free. If you want to get unlimited push and pulls, then the cost increases to around 60 USD / year.

The Docker Hub used to have no rate-limiting, but around the time it started charging, GitHub launched their own container registry: ghcr.io. ghcr.io integrates well with GitHub Actions, and can make a powerful combination when you're ready to deploy your code to production.

Self-hosted registries are also an option, however if you're new to containers, they are non-trivial to configure and host so it is not recommended.

Whenever you create a function, your `image:` field in your stack YAML file will need to be prefixed with your username on the Docker Hub, or GHCR.

For instance:

- `alexellis2/starbot`
- `ghcr.io/alexellis/starbot`

Whichever registry you use, you will need to authenticate with the `docker login` command before pushing any images. If your images are private or on a registry that disallows anonymous image pulls, then you will need to setup authentication to pull images. See the advanced chapter later in the book for how to do this.

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, use 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 node12 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/node12` - the template for node12
- `/template/node12/template.yml` - metadata for the node12 template like the welcome message and additional packages that can be installed
- `/template/node12/Dockerfile` - how the template is built
- `/template/node12/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/node12/function/handler.js` - the handler for the function to be customised by the user
- `/template/node12/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.

```
language: node12
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 node12 template:

```
# Change to your own username or GHCR prefix
export OPENFAAS_PREFIX=alexellis2
faas-cli new --lang node12 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: node12
    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 node12 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 a PC and deploying to a PC, you can run `faas-cli up`:

```
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.

First install the QEMU utilities which allow for cross-compilation:

```
$ docker run --rm --privileged multiarch/qemu-user-static --reset -p yes
```

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
```

> 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
```

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: node12
    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: node12
    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 three configuration items for Postgresql are:

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

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 node12 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: node12
    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 @- \
```