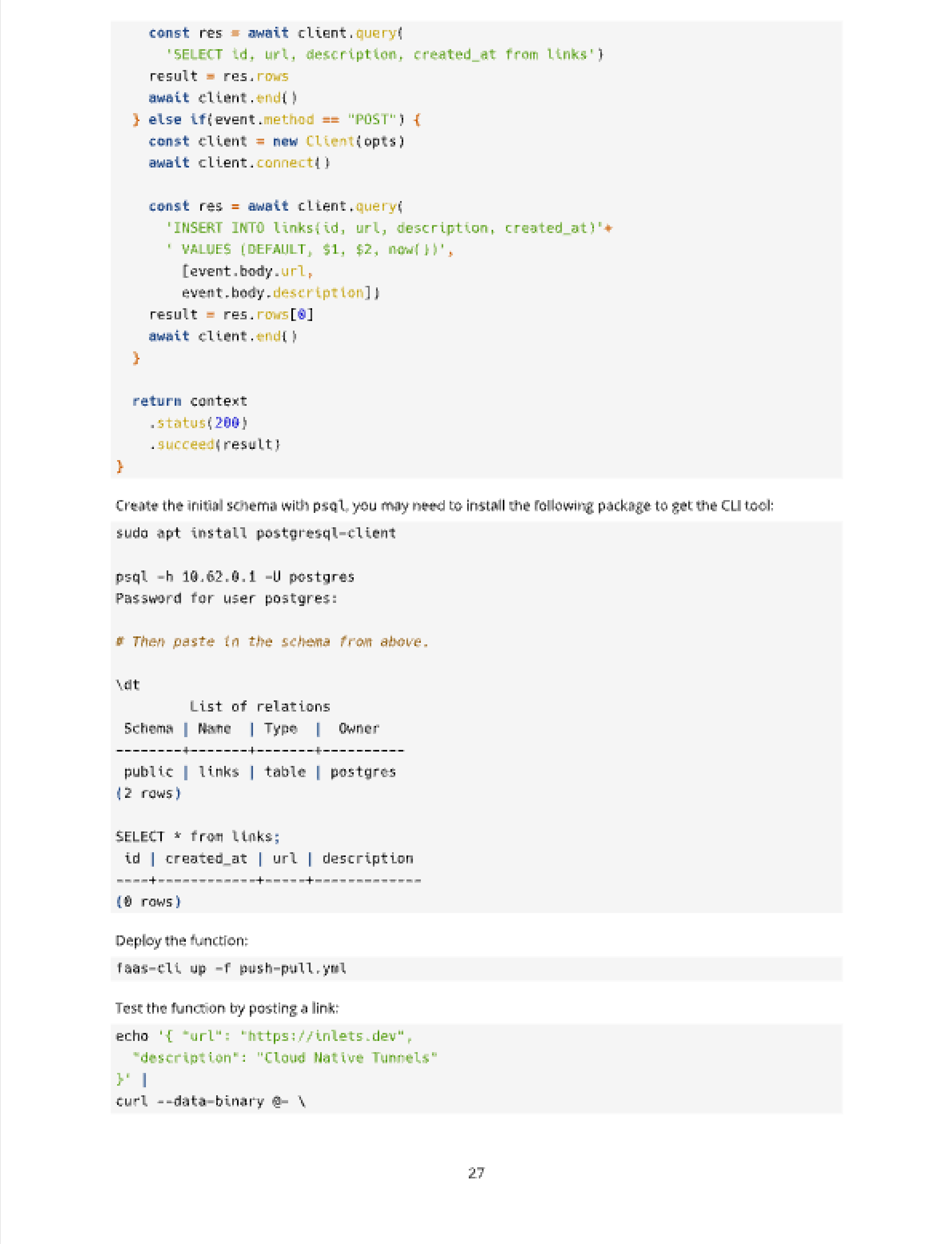
Page 16 of 21



```
    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:

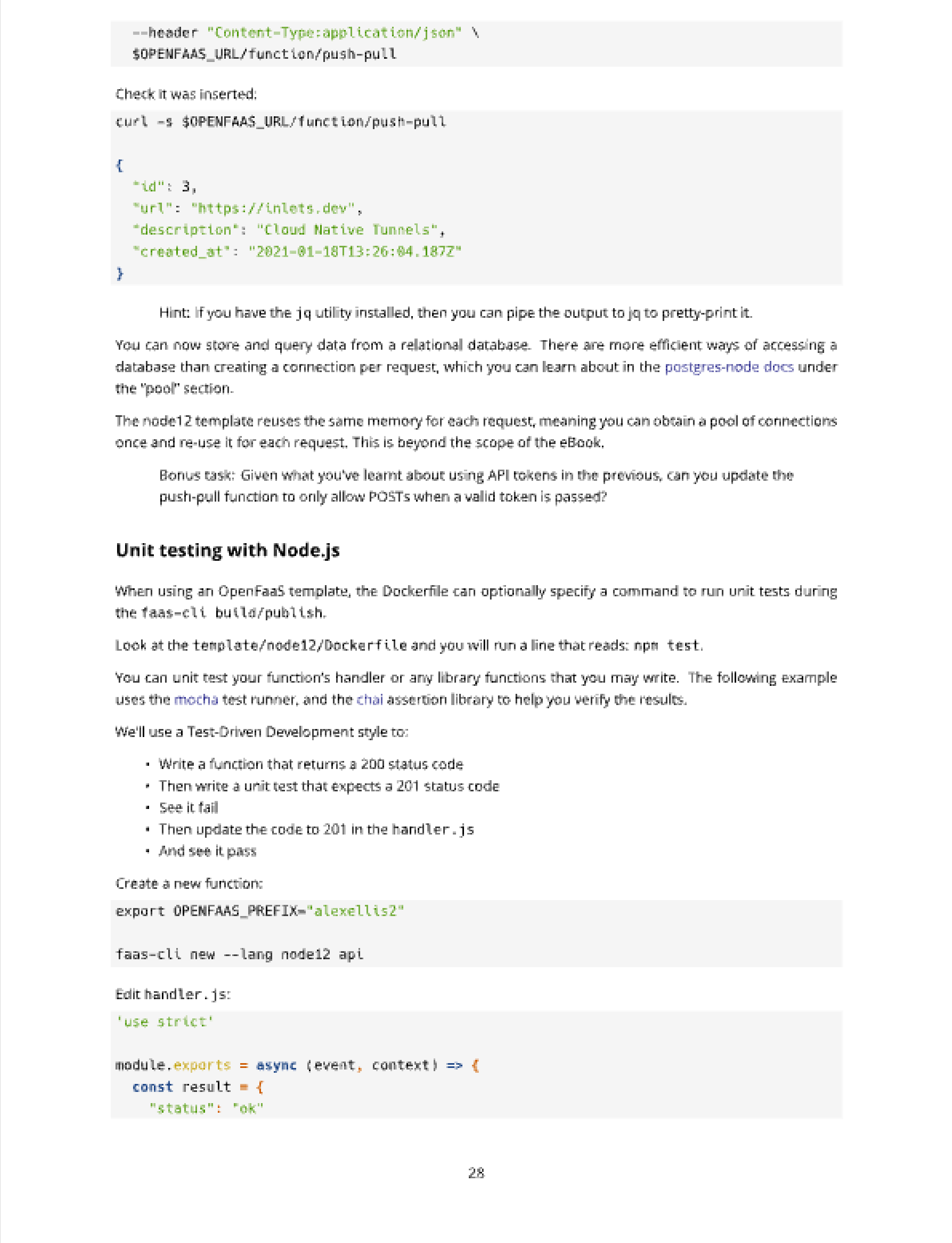
```
sudo apt install postgresql-client

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 node12 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/node12/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:

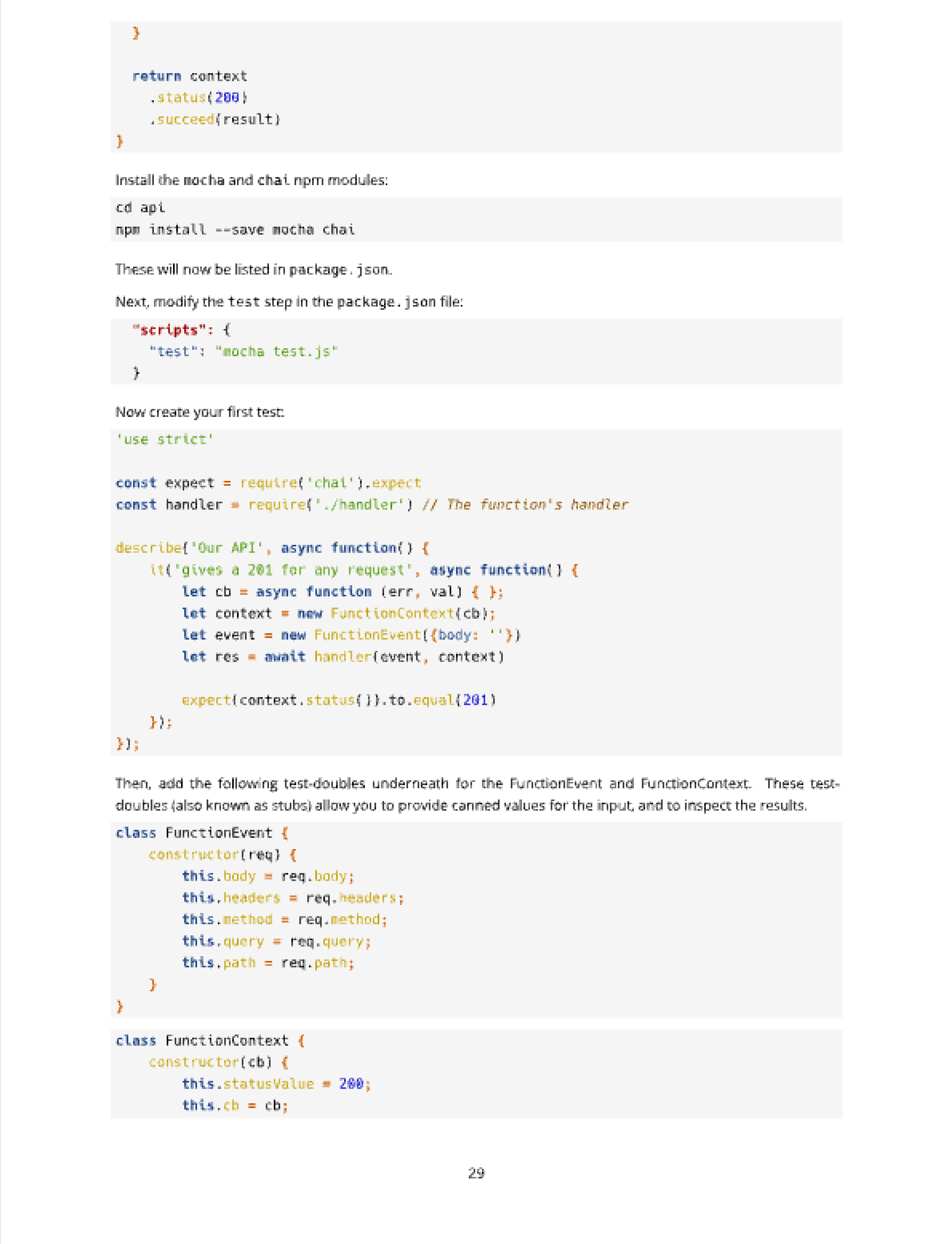
- Write a function that returns a 200 status code
- Then write a unit test that expects a 201 status code
- See it fail
- Then update the code to 201 in the `handler.js`
- And see it pass

Create a new function:

```
export OPENFAAS_PREFIX="alexellis2"

faas-cli new --lang node12 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"
  }
```

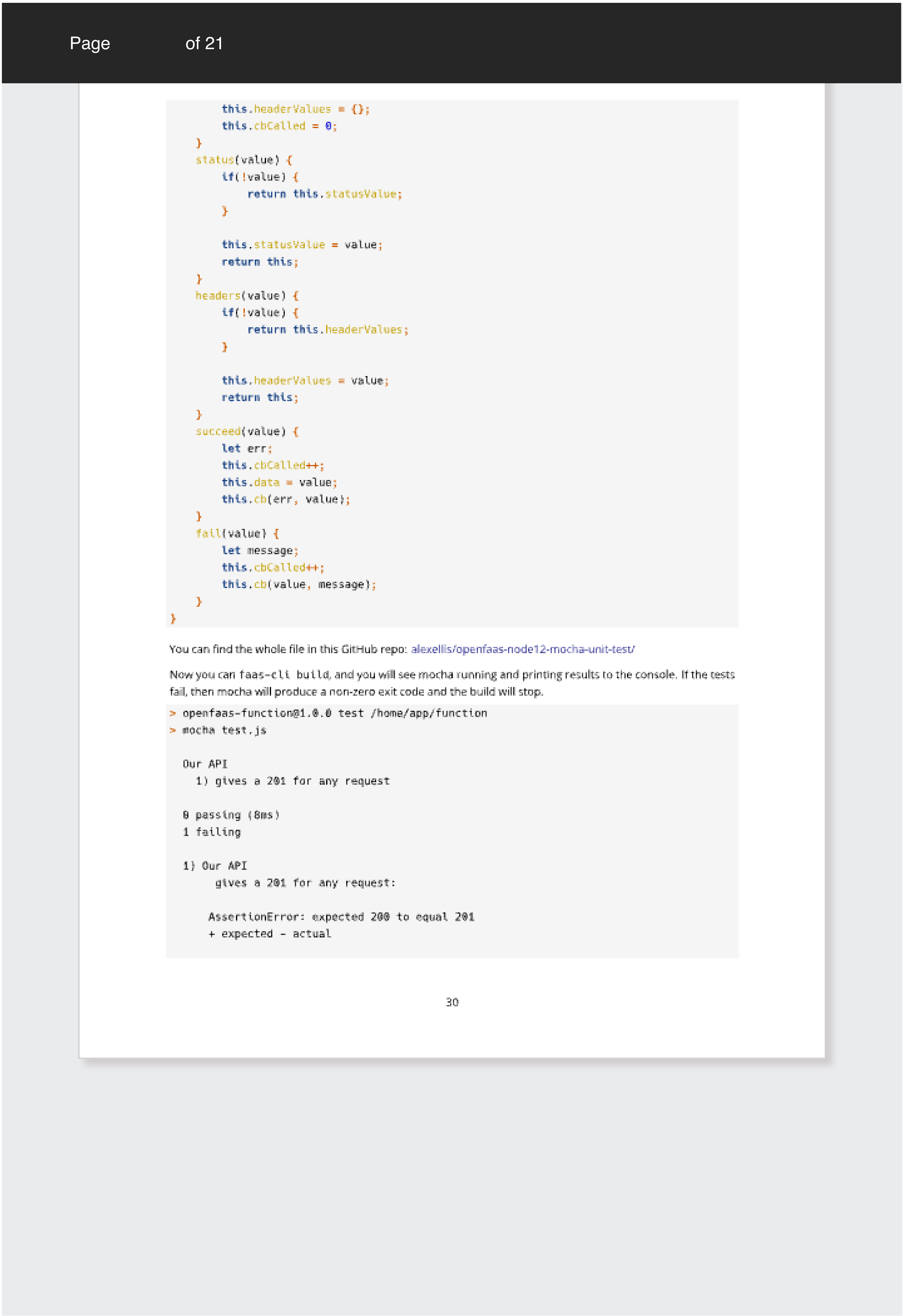
Now create your first test:

```
'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;
```

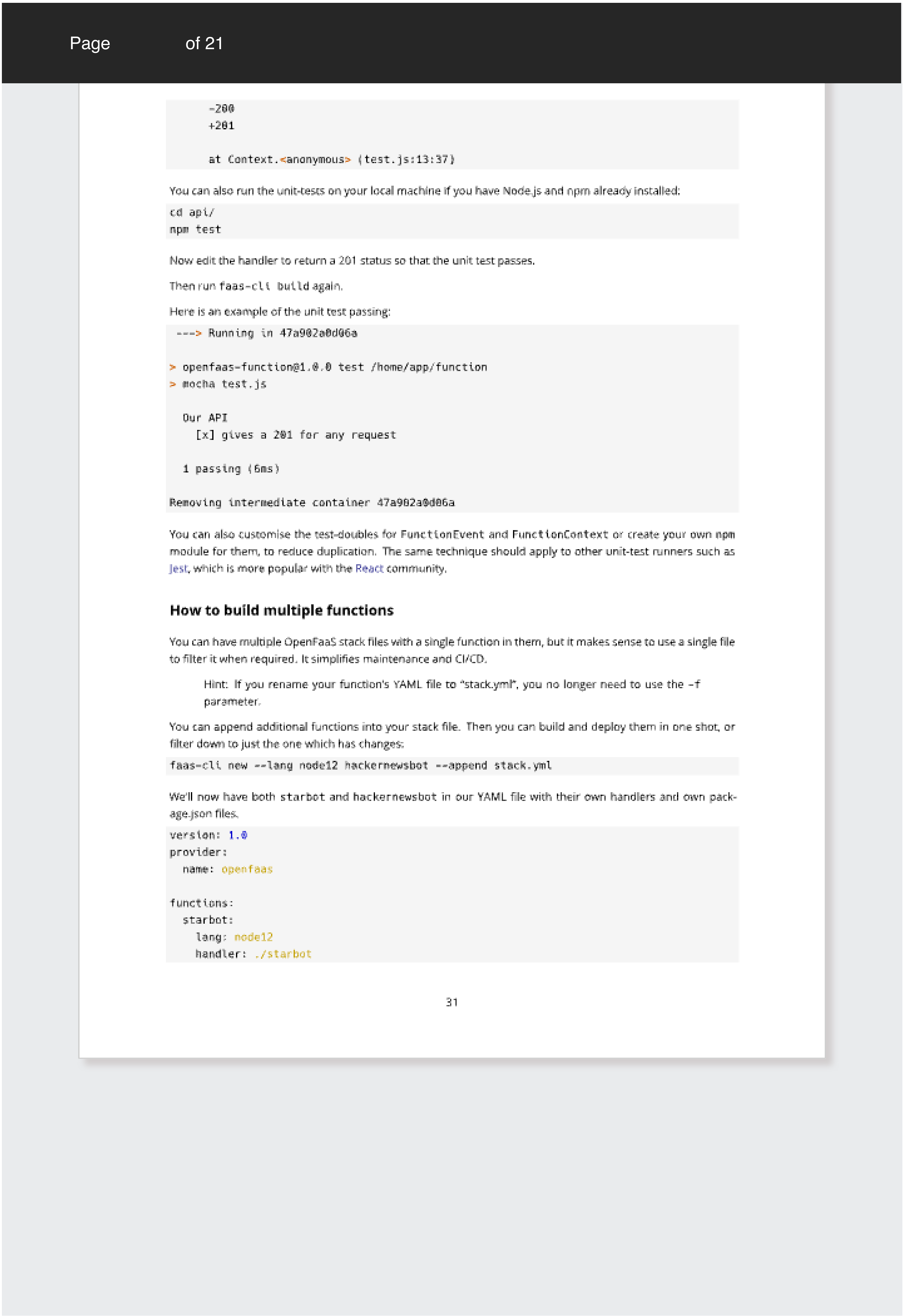


```
        this.headerValues = {};
        this.cbCalled = 0;
    }
    status(value) {
        if(!value) {
            return this.statusValue;
        }

        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-node12-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.

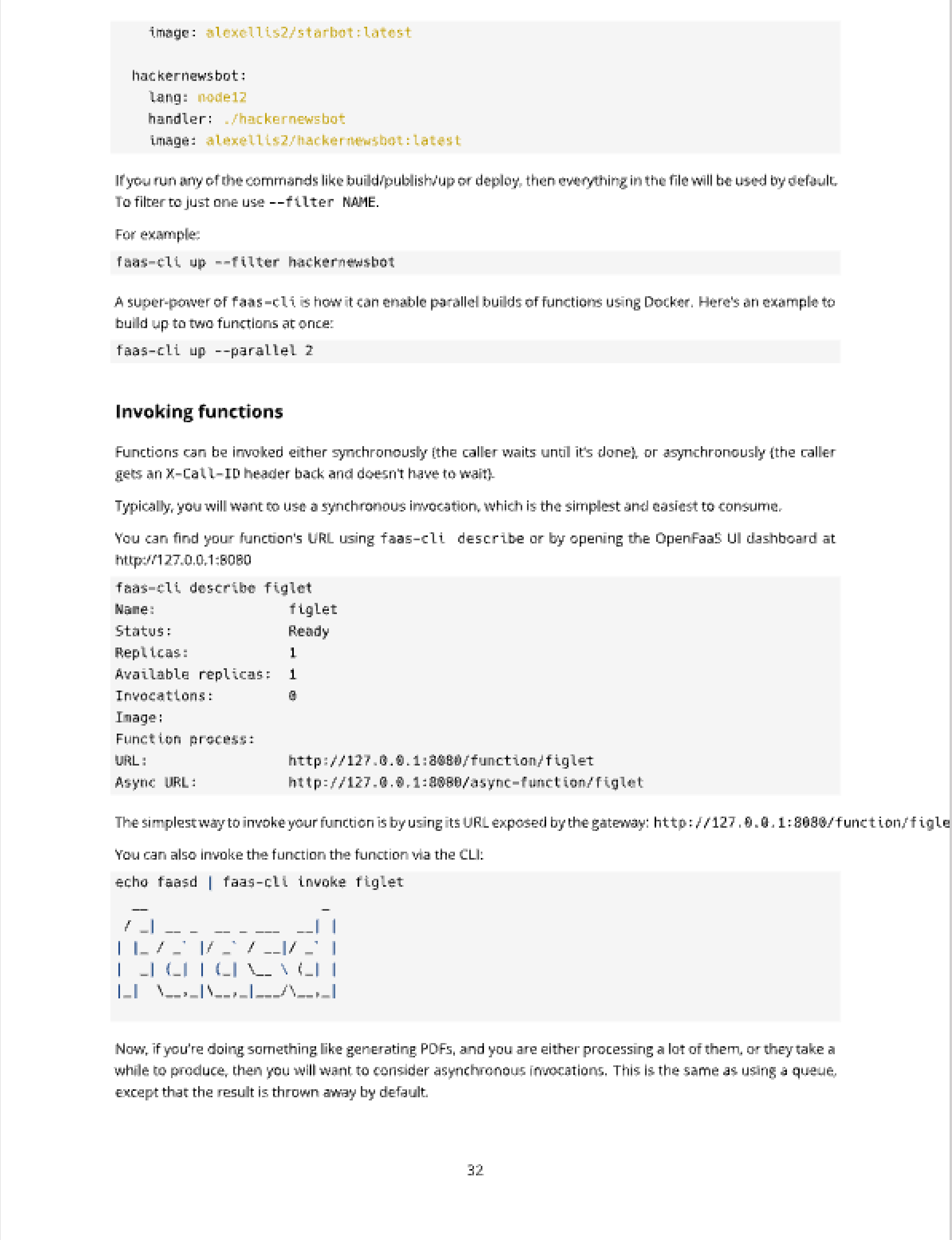
> Hint: If you rename your function's YAML file to "stack.yml", you no longer need to use the `-f` parameter.

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 node12 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: node12
    handler: ./starbot
```



```
    image: alexellis2/starbot:latest

  hackernewsbot:
    lang: node12
    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

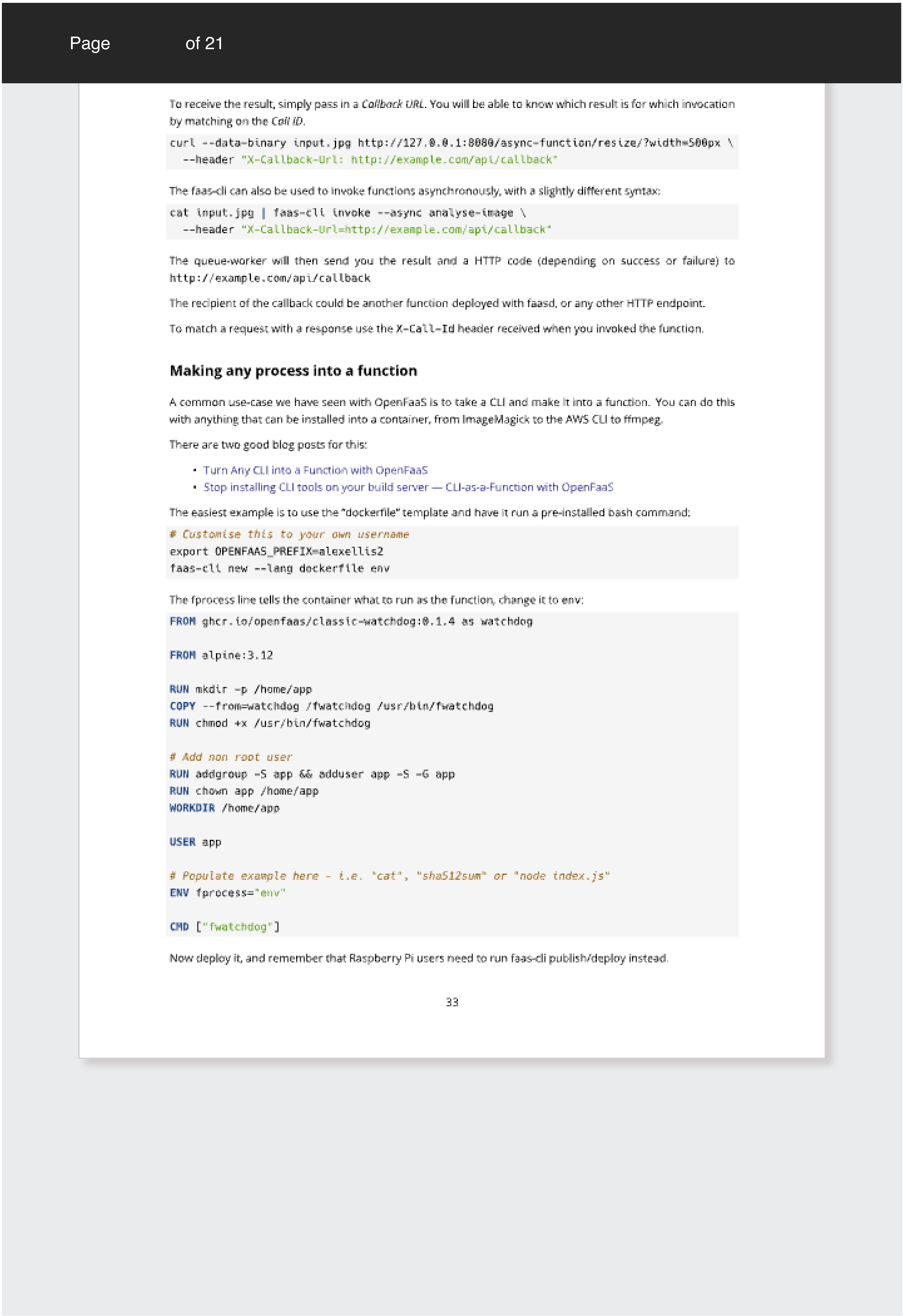
```
faas-cli describe figlet
Name:                figlet
Status:              Ready
Replicas:            1
Available replicas:  1
Invocations:         0
Image:
Function process:
URL:                 http://127.0.0.1:8080/function/figlet
Async URL:           http://127.0.0.1:8080/async-function/figlet
```

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

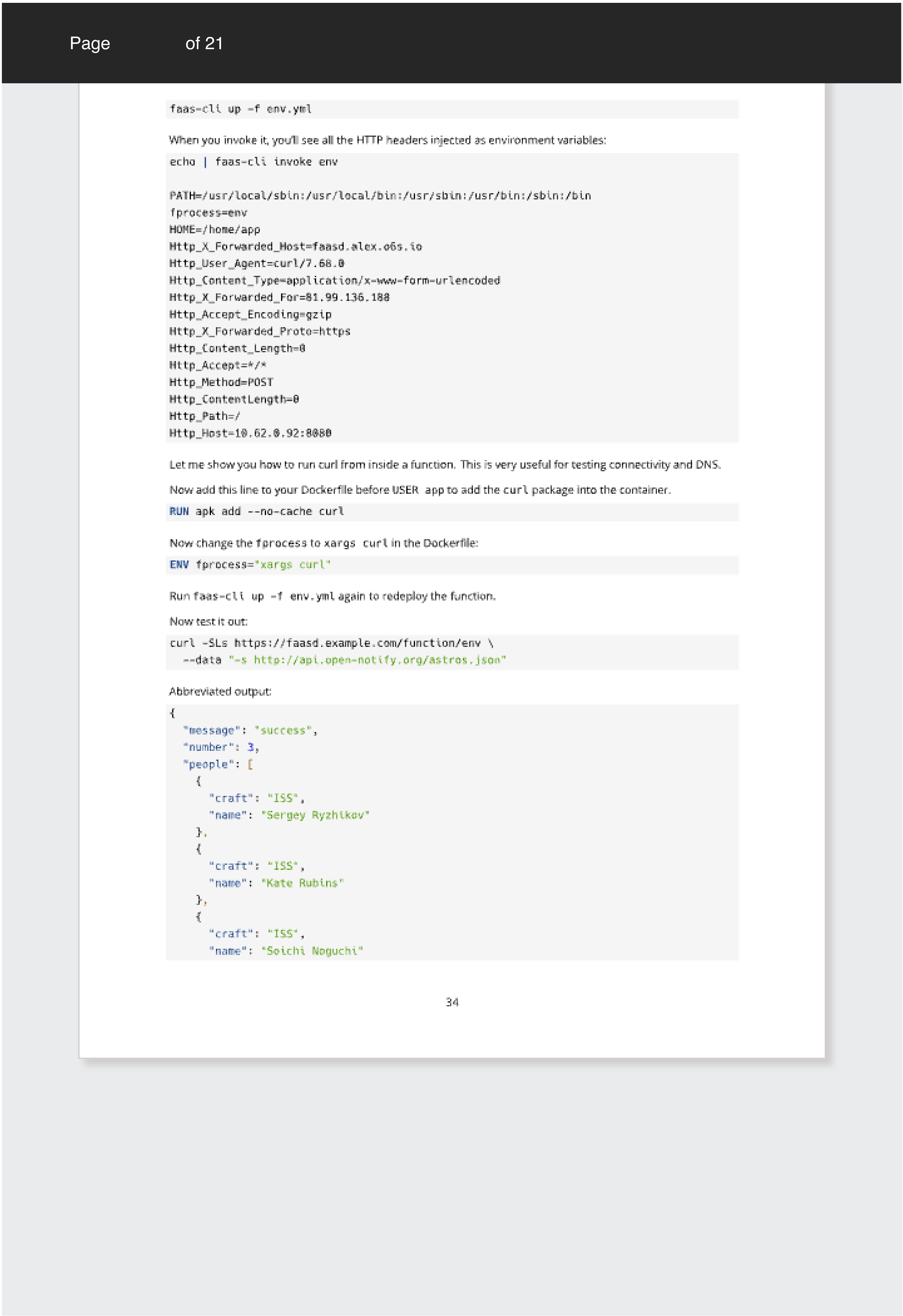
The easiest example is to use the "dockerfile" template and have it run a pre-installed bash command:

```
# 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.

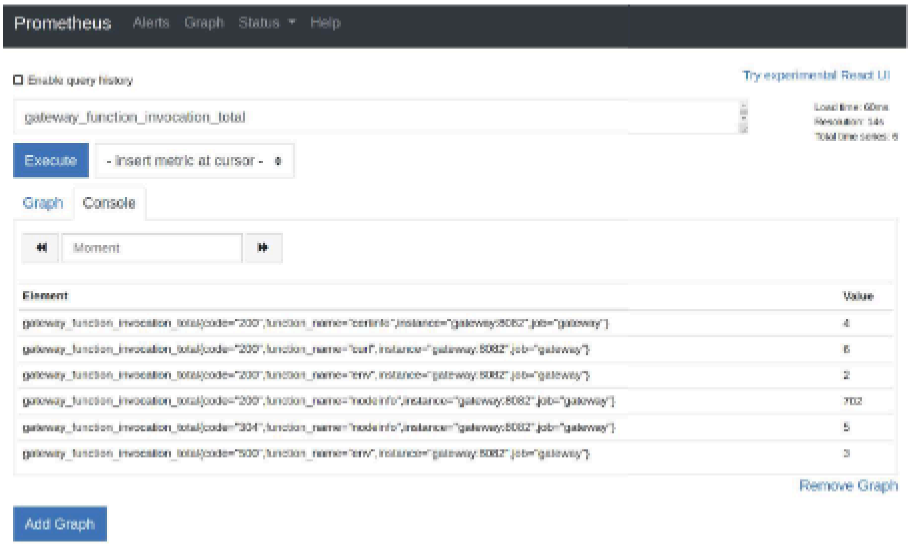
```
NAME                         SOURCE             DESCRIPTION
bash-streaming               openfaas-incubator Bash Streaming template

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

Chapter 5

Monitoring invocations

You can monitor invocations using the built-in Prometheus metrics, or by deploying Grafana as an additional component in the faasd stack.

> What's the difference between the two? Prometheus is a time-series database which has a basic UI built into it for visualising queries and Grafana is a UI tool for creating dashboards from many different data-sources like Prometheus and InfluxDB.



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: Prometheus UI

> The Prometheus UI showing the query executor

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 may vary depending on where you are hosting your faasd VM.

Now navigate to http://localhost:9090

You can view the various metrics using PromQL.

View the rate of new invocations started

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

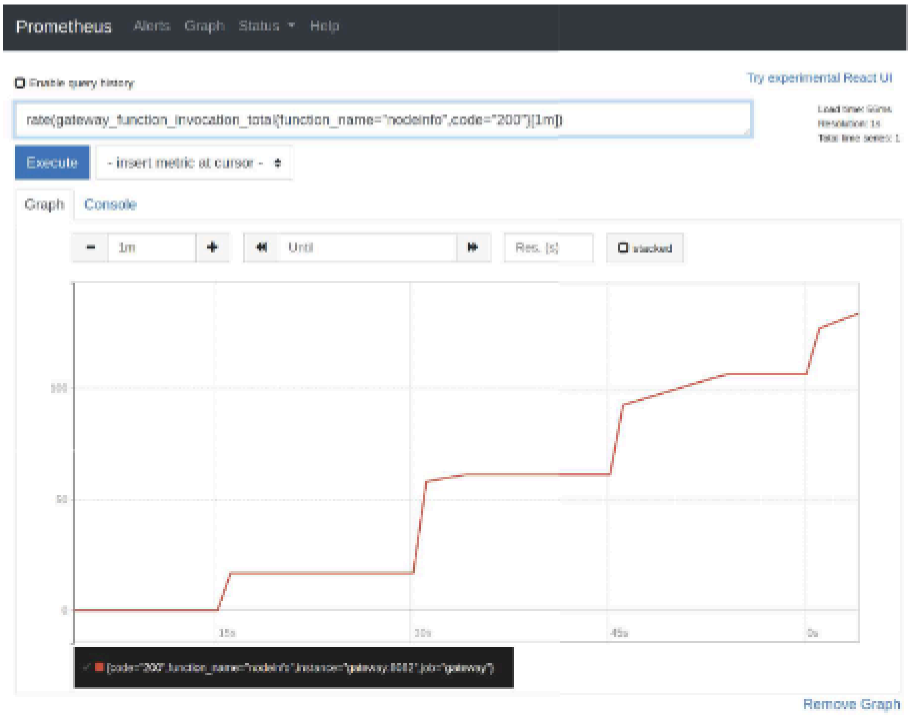
View the services which are scaled to zero or one replica:

```
gateway_service_count
```

The OpenFaaS REST API can be monitored through:

```
http_requests_total
http_request_duration_seconds
```

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.



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

Load time: 50ms
Resolution: 1s
Total time series: 1

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

Figure 5.2: Prometheus graph

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

See also: built-in metrics

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