Summary

The state is part of any service, and state is built upon data. For a more cloud-native experience, a service cannot depend on a traditional filesystem and needs to use other kinds of storage structures to store data. Databases are a natural progress, and there are some types of databases to choose from, depending on how important and complex our data is.

Assuming our state is securely stored in a database service of some kind, it's also important to ensure our data cannot be corrupted using our service. There can be security flaws and bugs in our service that may put our data at risk, so it's important to write simple code, validate input, and think about security in general when planning the service roadmap.

To progress our service, let's introduce something we haven't done yet, and should, which is a proper test suite. In the next chapter, we'll see some good options and create a test suite, establishing whether anything needs to change in order to make our service as secure as possible.

7
Testing

When you're developing an application, it will eventually form a structure and evolve into a stable product that you can use in production and sell to your customers. In the beginning, everything may seem simple and many tend to postpone the construction of a proper test suite.

> *"Debugging is twice as hard as writing the code in the first place."*
>
> *—Brian W. Kernighan and P. J. Plauger in The Elements of Programming Style*

Later on, the application may become just sufficiently complex for you to hesitate to begin testing. You may eventually give up and never test your application. It may be frustrating, especially if you have never seen or used any test suite before.

Proper testing gives you more than a little bit of quality assurance. Proper testing gives you:

- **Predictability**: This means that your code execution, no matter if it's an application or just a module, will have an expected result. As you evolve the tests and introduce different test cases, you begin to fulfill all the uses for your code, and you ensure its results were as intended.
- **Feature coverage**: This means that you can measure what parts of your code are tested or not. There are plenty of tools to inspect your code and tell you what parts of it haven't been used in your test suite, which helps you create specific tests for specific parts of the code that are not yet covered.
- **Safe evolution**: This is a side effect. When your code gets complex, if your test suite has good code coverage, you can make changes and add features without compromising stability, as you can continuously run the test suite and see if it breaks anything.

There's a developing methodology that involves first creating a test for a new feature and then making sure the test passes. This way, you can focus on how you think your code should be used (in the new test) and then evolve it (actually develop it) so the test stops failing and gives proper results.

So, let's begin by looking at some testing methodologies, and then write our first test. Then, let's see how code coverage can help in the testing process. Finally, we'll look at how you can mock parts of your code.

Types of testing methodologies

There are several types of testing methodologies. You can have tests to measure performance by stressing your application with specific actions and checking whether it achieves the expected minimum results. Those are important, but not at this stage. We should focus on other kinds of tests.

Our goal is to have a test suite that ensures our code behaves as we designed it. To ensure that, we must have:

- Unit tests, to check individual code units, such as some functions
- Integration tests, to check whether external actions produce the expected results

Using these two types of test, and having a full code coverage, we'll be able to develop new features and run regression tests, which is just a fancy way of saying that our test suite still passes, and our new code does not break (there's no regression).

You may see these regression tests as and when you run the test suite on your laptop and the results are OK, and then you commit your code to GitHub, for example, and Travis runs the same test suite on different environments to see if nothing breaks. If you're pushing to a pull request on GitHub, you'll be unable to merge your changes unless every test result passes.

To achieve full test coverage, you'll necessarily have test cases in the same language as the one used to develop the application, which in our case is Node.js. It's not mandatory, but it simplifies a lot of the tools used to run the test cases use the same language, as this creates a friendlier environment for other developers.

When using open source modules, it's not uncommon for others to find issues and try to fix them. Having test frameworks that involve installing third-party software will lower the interest in helping to fix them. On the other hand, if the user just needs to clone the repository, fix and run the tests, that will be a much more enjoyable experience.

Using frameworks

There are a couple of test frameworks available for Node.js. What they do is give you an environment where you can focus on the test cases. Some frameworks will even do code coverage more or less automatically.

One of the most commonly used frameworks is `mocha`, which is a simple yet powerful framework. It can be used for general unit testing, but also for performance, as it is able to highlight slow tests.

Let's try it out and prepare our microservice for the tests. You can use any of the previous versions; it should not matter for the storage model we're using, only the microservice interface. First of all, let's install `mocha` as a development dependency. We're also using an assertion library called `chai`, which has a plugin specifically for HTTP testing:

```
npm install –save-dev mocha chai chai-http
```

Then, change your `package.json` file to update the scripts part to something similar to this:

```
"scripts": {
    "test": "node test/run"
},
```

This tells NPM that, to test our microservice, it should run the `node test/run` command. We now need to create this file (`test/run.js`) so it will actually work. Create the `test` folder, add the `run.js` file, and add only this line to it:

```
console.log("ok");
```

Now, go to a console on our microservice folder and run the `test` command:

```
npm test
> imagini@1.0.0 test /Users/dresende/imagini
> node test/run

Ok
```

The first step is complete. This is not actually a test, but we now have the initial structure.

Integrating tests

We will now create our first integration tests. Each of our tests will run separately, meaning they should not depend on any other test and should follow a predictable workflow. First, we need to change our `run.js` file to run all test files. For that, we'll use `mocha` and add all files found in the `integration` folder:

```
const fs    = require("fs");
const path  = require("path");
const mocha = require("mocha");
const suite = new mocha();

fs.readdir(path.join(__dirname, "integration"), (err, files) => {
    if (err) throw err;

    files.filter((filename) =>
    (filename.match(/\.js$/))).map((filename) => {
        suite.addFile(path.join(__dirname, "integration", filename));
    });

    suite.run((failures) => {
        process.exit(failures);
    });
});
```

Then, let's create the `integration` folder inside our `test` folder, and let's create our first test file, called `image-upload.js`. Add this content to the file:

```
describe("Uploading image", () => {
    it("should accept only images");
});
```

If we now run the tests again, we should see the default `mocha` response with no tests passing and no tests failing:

```
npm test
> imagini@1.0.0 test /Users/dresende/imagini
> node test/run

0 passing (2ms)
```

To avoid repeating code, let's create a `tools.js` file inside the `test` folder, so we can export common tasks that every test file can use. Out of the box, I'm thinking about our microservice location and a sample image:

```
const fs   = require("fs");
const path = require("path");
```

```
exports.service = require("../imagini.js");
exports.sample  = fs.readFileSync(path.join(__dirname, "sample.png"));
```

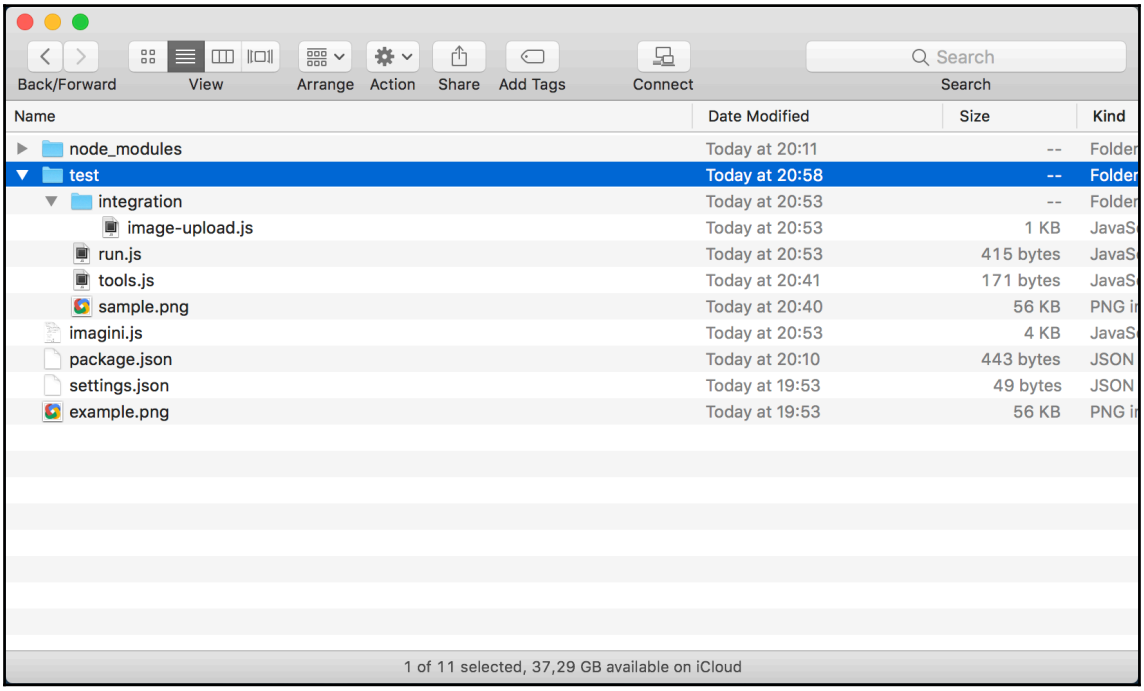
Create a `sample.png` image in the `test` folder. When a test needs to upload an image, it will use that sample. In the future, we could have different kinds of samples, such as huge images, to test performance and limitations.

Using chai

We also need to make a little change to our microservice. We need to export its app so that the HTTP plugin from `chai` can load it and were able to test it without the need to run in a separate console. Add this to the end of our microservice file:

```
module.exports = app;
```

You should have a folder hierarchy similar to the following screenshot:



We should now change our `image-upload.js` test file to create our first real test:

```
const chai  = require("chai");
const http  = require("chai-http");
const tools = require("../tools");

chai.use(http);

describe("Uploading image", () => {
    beforeEach((done) => {
        chai
        .request(tools.service)
        .delete("/uploads/test_image_upload.png")
        .end(() => {
            return done();
        });
    });

    it ("should accept a PNG image", function (done) {
        chai
        .request(tools.service)
        .post("/uploads/test_image_upload.png")
        .set("Content-Type", "image/png")
        .send(tools.sample)
        .end((err, res) => {
            chai.expect(res).to.have.status(200);
            chai.expect(res.body).to.have.status("ok");

            return done();
        });
    });
});
```

We start by first including the `chai` modules and our `tools` file:

```
const chai  = require("chai");
const http  = require("chai-http");
const tools = require("../tools");

chai.use(http);
```

Then, we describe our test file as `Uploading image`:

```
describe("Uploading image", () => {
```

We'll add the different use cases we can think of, related to the uploading of images.

Inside, we use `beforeEach`, which is a `mocha` method that will be called before every test in this file. Remember, we want our tests to be consistent, so we add this method to remove our image before running every test. We don't care whether the image exists:

```
beforeEach((done) => {
    chai
    .request(tools.service)
    .delete("/uploads/test_image_upload.png")
    .end(() => {
        return done();
    });
});
```

Look how we use the `tools.service`, which points to our microservice. If, later on, we change the name or somehow make it more complex, we just need to change the `tools` file, and everything should work.

Then, we add our first `integration` file's test – a simple image upload:

```
it("should accept a PNG image", (done) => {
    chai
    .request(tools.service)
    .post("/uploads/test_image_upload.png")
    .set("Content-Type", "image/png")
    .send(tools.sample)
    .end((err, res) => {
        chai.expect(res).to.have.status(200);
        chai.expect(res.body).to.have.status("ok");

        return done();
    });
});
```

It checks whether the HTTP response code is `200` and if the response body, which is a JSON structure, has the status property set to `ok`. And we're done!

Let's run our test suite again and see how it goes.

Adding code coverage

Now that our test suite is working and has one test, let's introduce code coverage. Adding this from the beginning of development is very easy and will help us focus on parts of the code that need to be tested, especially some use cases that involve specific conditions (such as `if-then-else` statements in our code). Having it all set up from the start of development is easy. On the other hand, if you have a fully working code and want to add tests and coverage, it will be harder and will take quite some time.

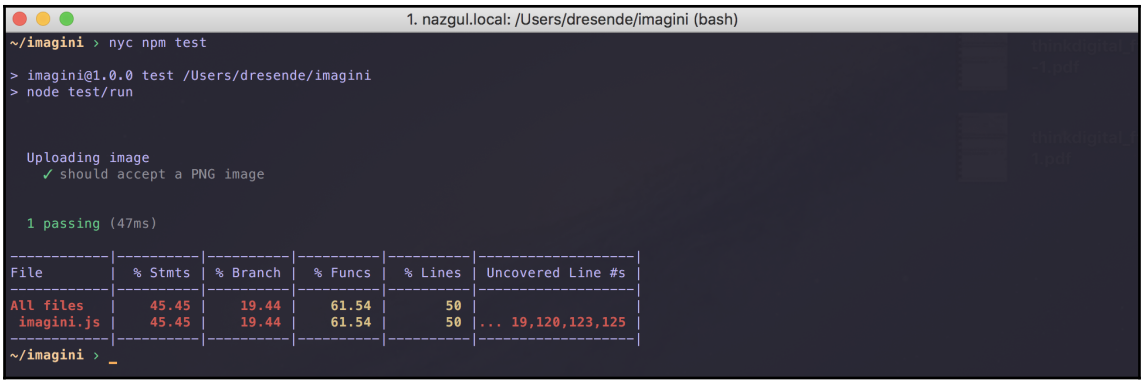
To add code coverage, we'll introduce another module. We'll install it globally to be able to run the tests with it directly:

```
npm install -g nyc
```

We can now run our tests with the following instrumentation:

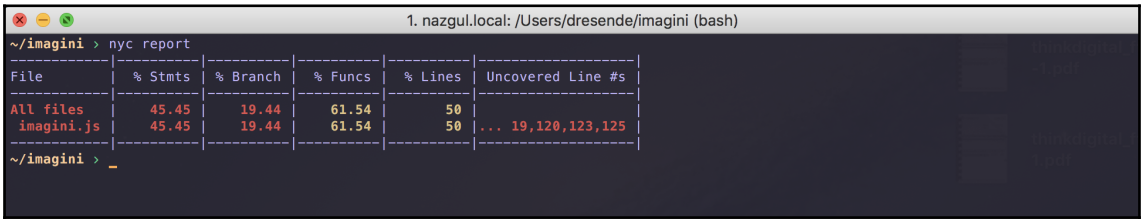
```
nyc npm test
```

This should run our tests with the instrumentation installed. In the end, you'll get a nice console report.



The coverage results are stored inside in a `.nyc_output` folder. This enables you to look at the last test results without running tests again. This is useful if your test suite is big and takes some time to finish.

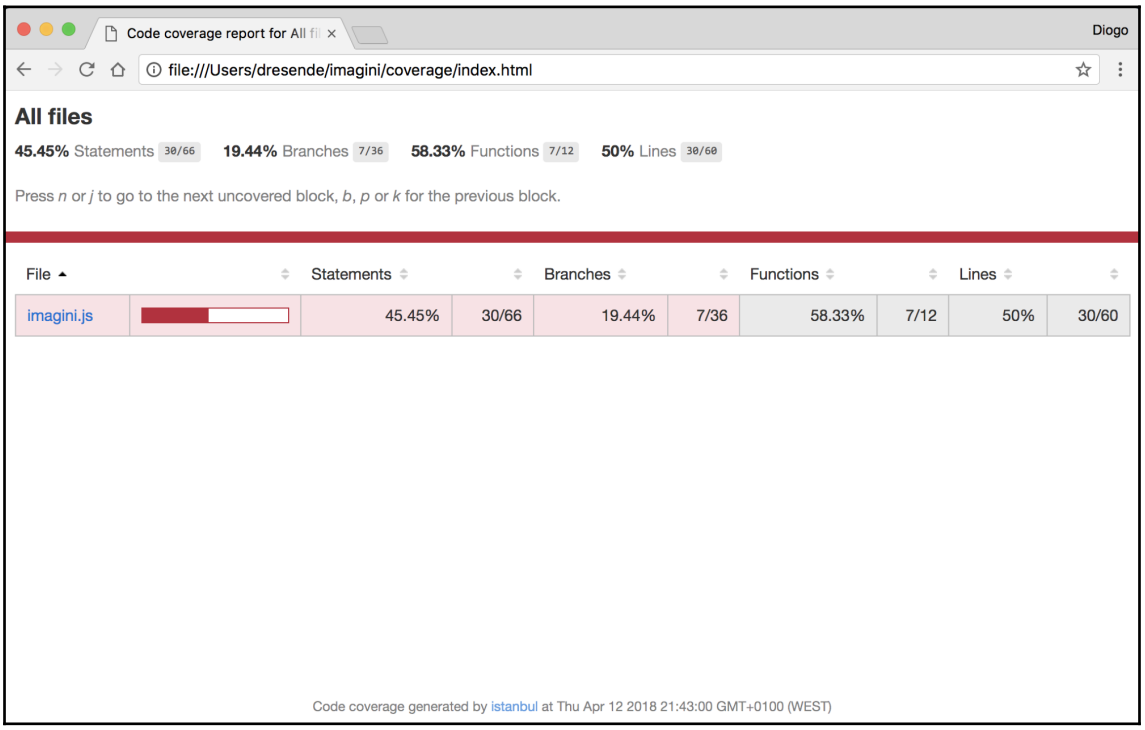
To see the results, you just run `nyc report`:



The result is a console report. There are several other styles of reports. One particularly useful one is the `html` report. Let's generate it:

```
nyc report --reporter=html
```

You should now have a `coverage` folder with an `index.html` file. Open that in your browser, and you should see something like the following screenshot:

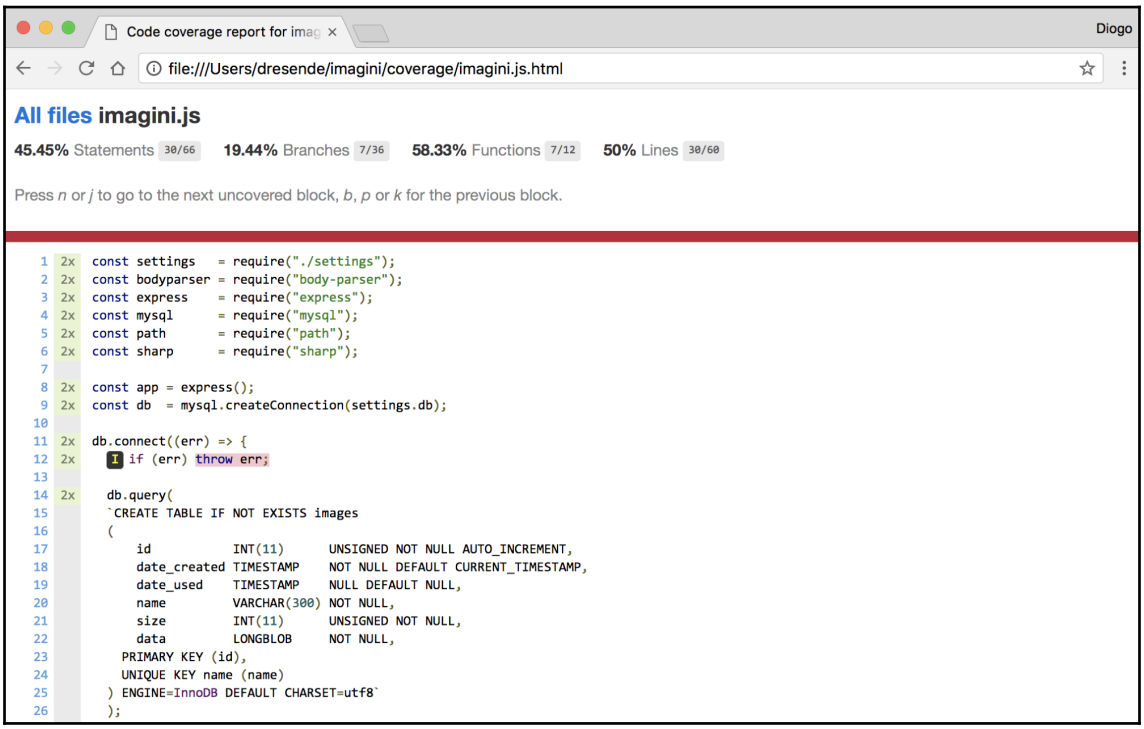


We only have one file that represents our microservice. If we had more, they would be listed hierarchically. There are global average statistics for every file.

There are three important groups of columns:

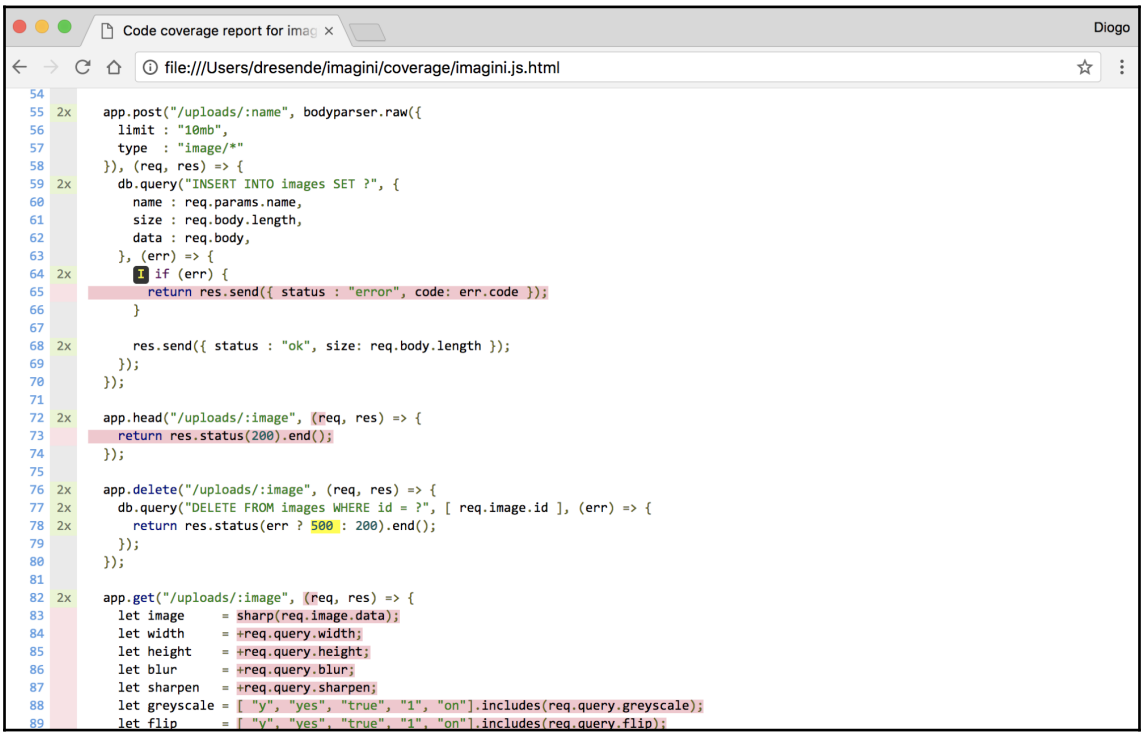
- **Statements**: Which represent code statements (conditions, assignments, assertions, calls, and so on)
- **Branches**: Which represent possible code control workflows, such as if-then-else or switch-case statement possibilities
- **Functions**: Which represent our actual code functions and callbacks

You can click in our file, look at the specific details of it and, more specifically, see the code and information line by line:



To the right of every line number, you see a gray area and, in this case, you see `2x` in some of the lines. This is the execution count for that line. The execution has passed by that line twice. This is actually not that important, unless you're looking for bits of code that get largely executed and you want to do some kind of optimization.

You can also see that *line 12* has two changes. First, there's a pinkish background in the back of `throw err`. That means that statement never got executed, which is normal for now as we always successfully connected to the database. The mark before the `if` statement means that the condition never got executed:



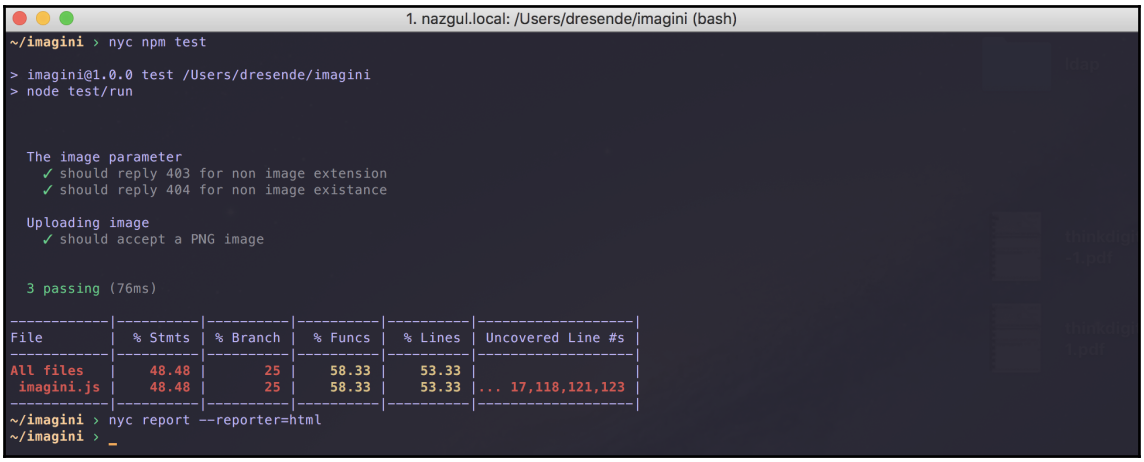
If you scroll a few lines down, we'll see more lines with marks. For example, we can see our image upload method got almost completely covered. The only statement missing is the error handling.

As we delete our test image before running the tests, our image deletion method is also covered. Again, the only missing branch is if the database returns an error to our `DELETE` query.

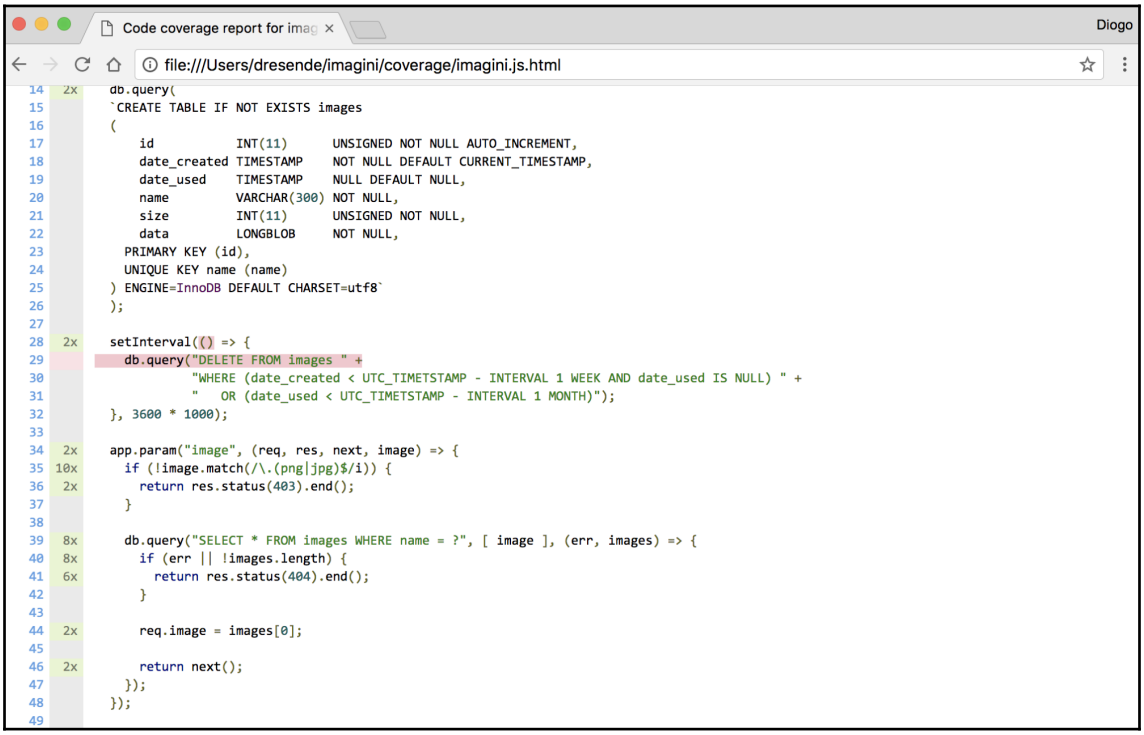
Before going any further with the image upload, let's add another `integration` test file called `image-parameter.js`, and add some tests to increase our coverage:

```
const chai = require("chai");
const http = require("chai-http");
const tools = require("../tools");

chai.use(http);

describe("The image parameter", () => {
    beforeEach((done) => {
        chai
        .request(tools.service)
        .delete("/uploads/test_image_parameter.png")
        .end(() => {
            return done();
        });
    });

    it("should reply 403 for non image extension", (done) => {
        chai
        .request(tools.service)
        .get("/uploads/test_image_parameter.txt")
        .end((err, res) => {
            chai.expect(res).to.have.status(403);

            return done();
        });
    });

    it("should reply 404 for non image existence", (done) => {
        chai
        .request(tools.service)
        .get("/uploads/test_image_parameter.png")
        .end((err, res) => {
            chai.expect(res).to.have.status(404);

            return done();
        });
    });
});
```

Let's run our test suite and see how it goes:



Refresh the HTML report page and look at our parameter method:



As you can see, we now cover the following condition:

```
if (!image.match(/\.(png|jpg)$/i)) {
```

The following condition:

```
if (err || !images.length) {
```

We now have full coverage on this method.

There are other coverage lines that are harder to test, such as timers (you can see one on *line 28*), `catch` statements, or external errors coming from databases or other storage sources. There are ways of mocking those events, and we'll cover them later on.

Covering all code

For now, let's focus on adding coverage to our code. It's important to have it covered as much as possible when it's still just a small service. If we start adding tests and coverage when it's already big, you'll be frustrated, and it will be hard to find the motivation to cover it all.

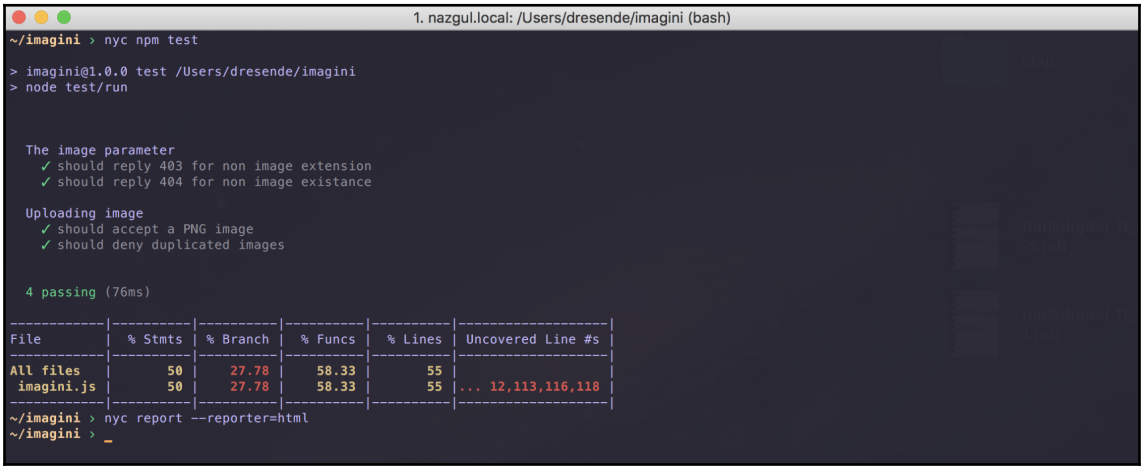
This way, you'll find it rewarding to cover it in the beginning and keep the coverage percentage as high as possible along with code evolution.

Let's get back to our image upload test, and add another test:

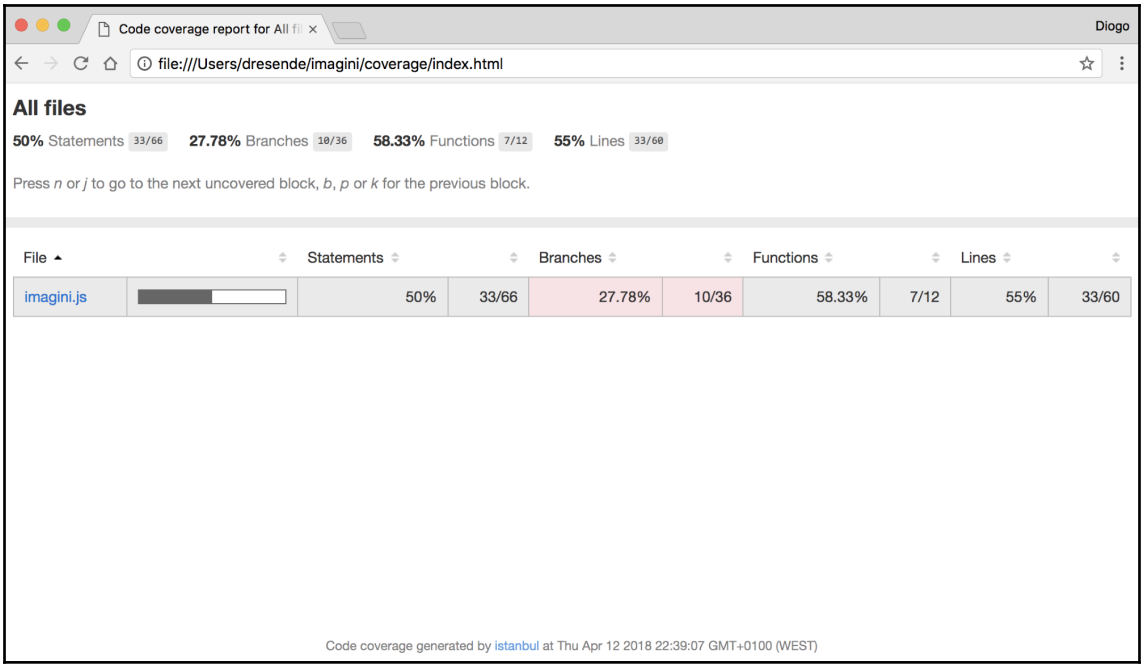
```
it("should deny duplicated images", (done) => {
    chai
    .request(tools.service)
    .post("/uploads/test_image_upload.png")
    .set("Content-Type", "image/png")
    .send(tools.sample)
    .end((err, res) => {
        chai.expect(res).to.have.status(200);
        chai.expect(res.body).to.have.status("ok");

        chai
        .request(tools.service)
        .post("/uploads/test_image_upload.png")
        .set("Content-Type", "image/png")
        .send(tools.sample)
        .end((err, res) => {
            chai.expect(res).to.have.status(200);
            chai.expect(res.body).to.have.status("error");
            chai.expect(res.body).to.have.property("code",
            "ER_DUP_ENTRY");
```

```
                return done();
            });
        });
    });
```

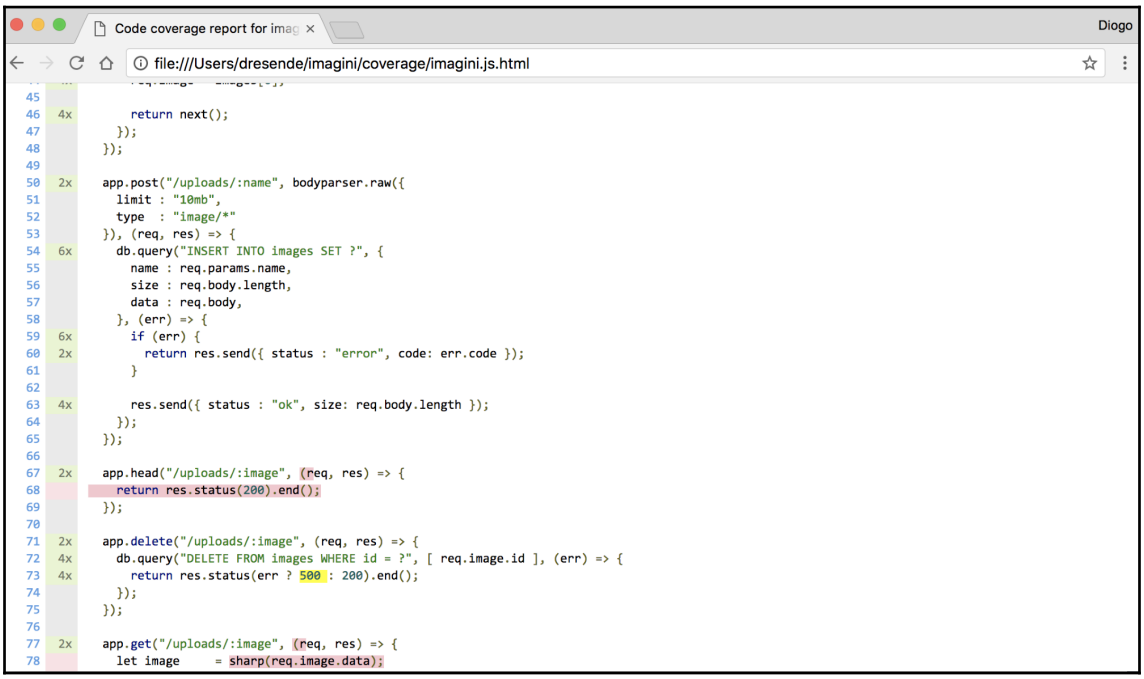
This will upload the same image twice in a row and we should receive an error from the database saying there's a duplicate. Let's run the tests again:



Now, let's open the initial page of the coverage report:



Notice that our file is no longer in a red background. This means the statement coverage has reached *50%*. Let's click on our file and see how our image upload method is covered:



It's complete! We can now move on. Just a reminder before we move to another method: having full coverage does not mean there are no bugs. That's something you need to understand. You might have a use case that you're not expecting, and so, you have no code for it, so there's no obvious coverage.

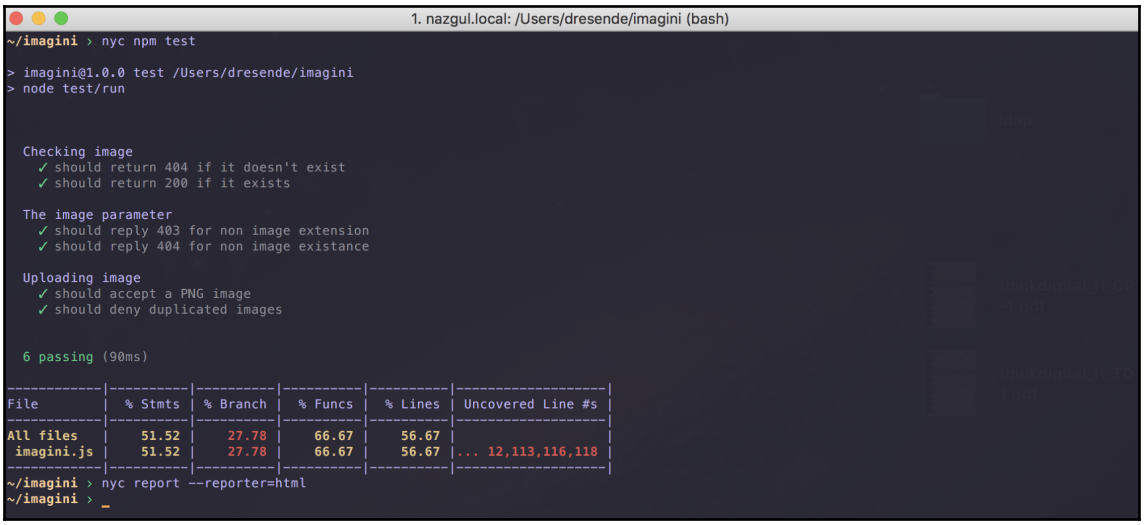
For example, the `bodyparser` module will not limit the type of content. If we upload a text file with an image name on it, our code will accept it and store it in the database without noticing. Think of this use case as your homework, and try to create a test to cover that use case and then fix the code.

Let's move to the next method we see after our upload method: the image check on *line 67*. Let's create a new integration test file called `image-check.js`, and add a simple test:

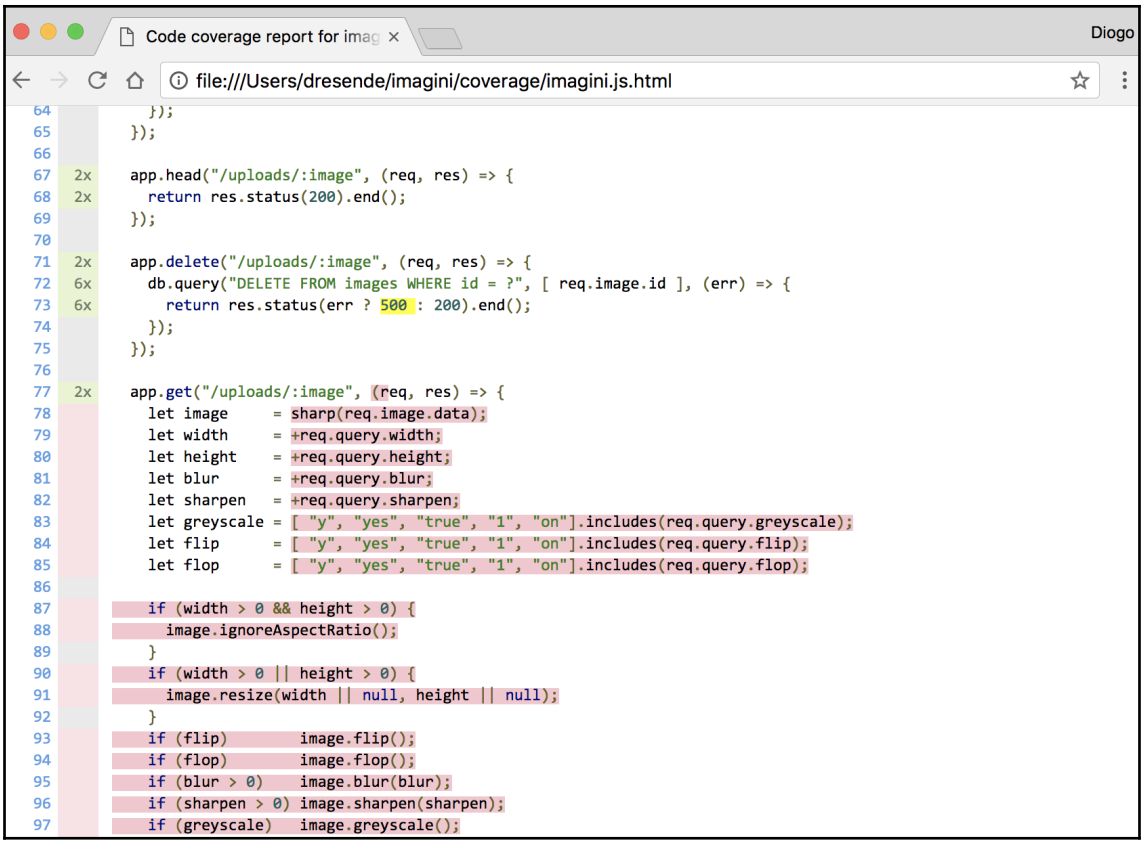
```
const chai = require("chai");
const http = require("chai-http");
const tools = require("../tools");

chai.use(http);
```

```
describe("Checking image", () => {
    beforeEach((done) => {
        chai
        .request(tools.service)
        .delete("/uploads/test_image_check.png")
        .end(() => {
            return done();
        });
    });

    it("should return 404 if it doesn't exist", (done) => {
        chai
        .request(tools.service)
        .head("/uploads/test_image_check.png")
        .end((err, res) => {
            chai.expect(res).to.have.status(404);

            return done();
        });
    });

    it("should return 200 if it exists", (done) => {
        chai
        .request(tools.service)
        .post("/uploads/test_image_check.png")
        .set("Content-Type", "image/png")
        .send(tools.sample)
        .end((err, res) => {
            chai.expect(res).to.have.status(200);
            chai.expect(res.body).to.have.status("ok");

            chai
            .request(tools.service)
            .head("/uploads/test_image_check.png")
            .end((err, res) => {
                chai.expect(res).to.have.status(200);

                return done();
            });
        });
    });
});
```

Let's run the test suite:



We can see our console report is getting bigger. As we're creating new integration test files and having a description for each one, `mocha` writes a nice tree view showing how the tests run. On the bottom, we can see the coverage report:



Looking at the check method, we see it's now fully covered. This one was very simple.

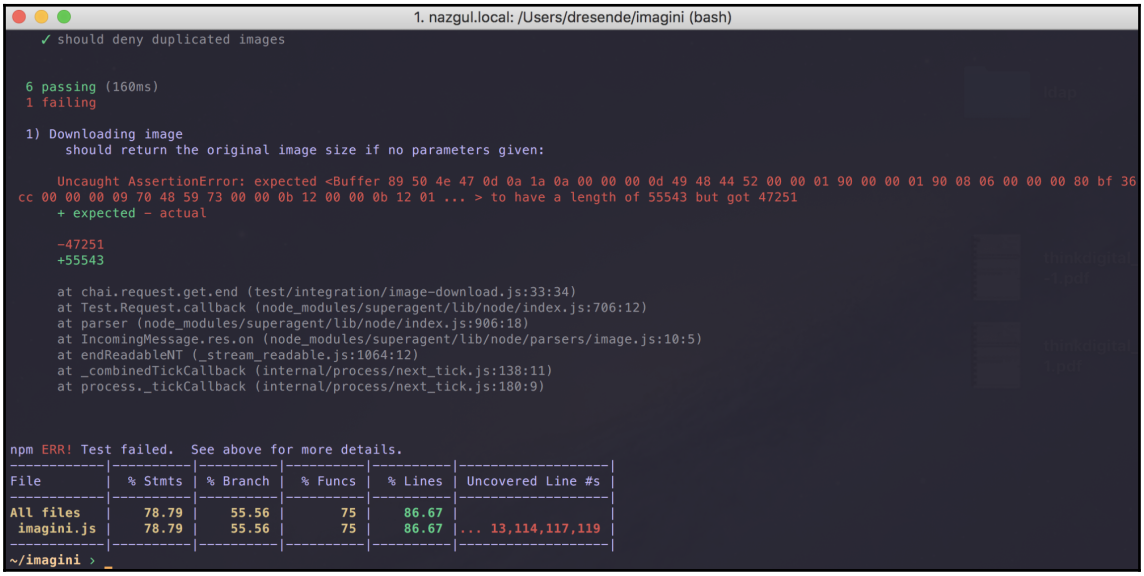
We're still in the middle of statement coverage as our top method; the image manipulation one is almost half of our code. This means that when we start covering it, the coverage will significantly rise.

Let's create an `integration` test for it:

```
const chai = require("chai");
const http = require("chai-http");
const tools = require("../tools");

chai.use(http);

describe("Downloading image", () => {
    beforeEach((done) => {
        chai
        .request(tools.service)
        .delete("/uploads/test_image_download.png")
        .end(() => {
            chai
            .request(tools.service)
            .post("/uploads/test_image_download.png")
            .set("Content-Type", "image/png")
            .send(tools.sample)
            .end((err, res) => {
                chai.expect(res).to.have.status(200);
                chai.expect(res.body).to.have.status("ok");

                return done();
            });
        });
    });

    it("should return the original image size if no parameters given",
    (done) => {
        chai
        .request(tools.service)
        .get("/uploads/test_image_download.png")
        .end((err, res) => {
            chai.expect(res).to.have.status(200);
            chai.expect(res.body).to.have.length(tools.sample.length);

            return done();
        });
    });
});
```

Before each test, we're deleting the image (if it exists) and then uploading a fresh sample one. Then, for each test, we'll download it and test the output according to what we asked for.

Let's try and run it:



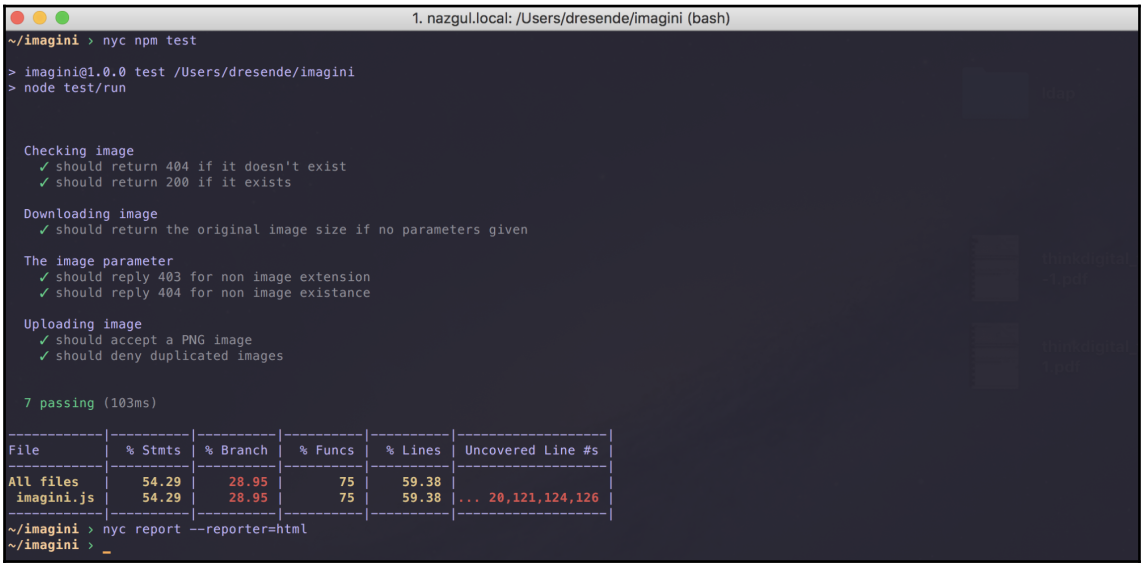
Well, that was unexpected. The test fails because our length check does not match. This is actually a good example of something we just notice when we start to execute testing.

What happens is that, when we request an image, we use the `sharp` module to make any manipulation on the image, according to query parameters. In this case, we're not asking for any manipulation, but when we output the image (through `sharp`), it actually returns the same image in size, but perhaps with a little bit less quality, or maybe it just knows how to better encode our image and remove data from the file that is not needed.

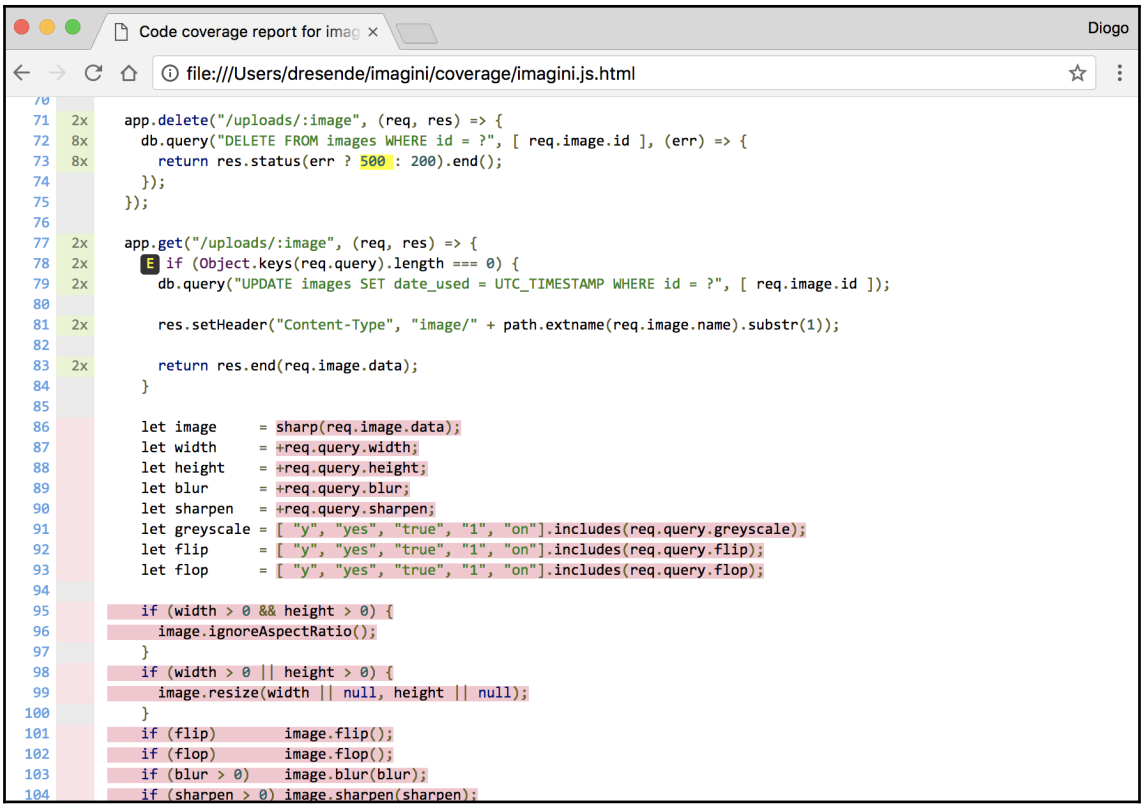
We don't know exactly, but let's assume we want the original image, untouched. We need to change our download method. Let's assume that if no query parameters are defined at all, we just return the original image. Let's add a condition to the top of our method:

```
if (Object.keys(req.query).length === 0) {
    db.query("UPDATE images " +
             "SET date_used = UTC_TIMESTAMP " +
             "WHERE id = ?",
             [ req.image.id ]);

    res.setHeader("Content-Type", "image/" +
    path.extname(req.image.name).substr(1));

    return res.end(req.image.data);
}
```

If we run it now, we should have no failures:



Our statement coverage did not rise much because we actually created a condition on top of the method and returned immediately, so our previous method is still untested:



Looking at *line 78,* you should see a new mark, an `E` that means that the condition in that line never executed the `else` statement, which is the rest of our code. Let's add a test to this integration and resize our image.

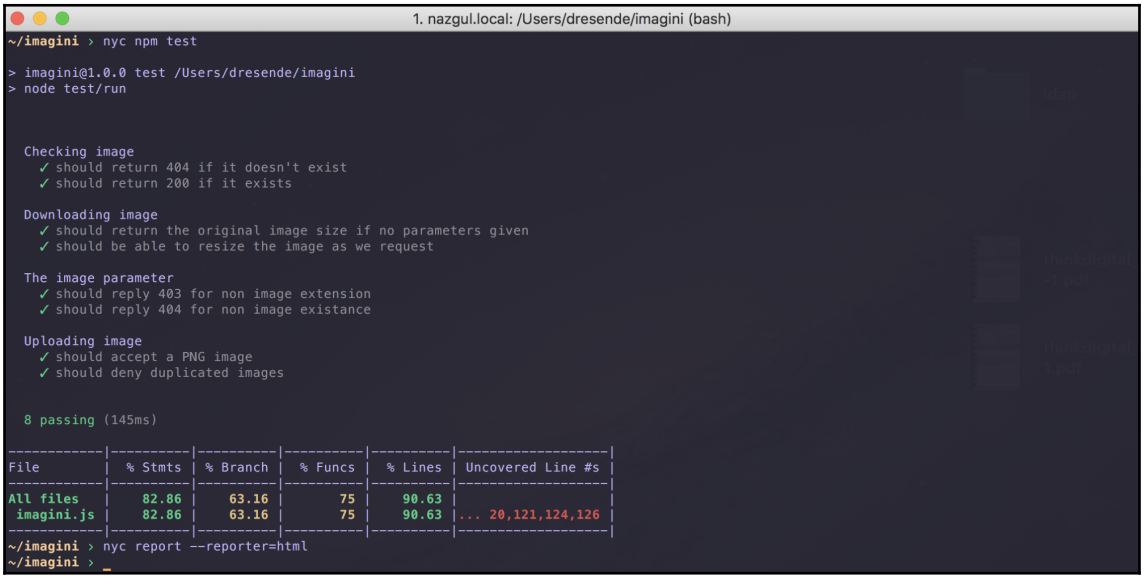
We will need `sharp` to help us check whether the results are correct. Let's include it on the top of our file:

```
const sharp = require("sharp");
```

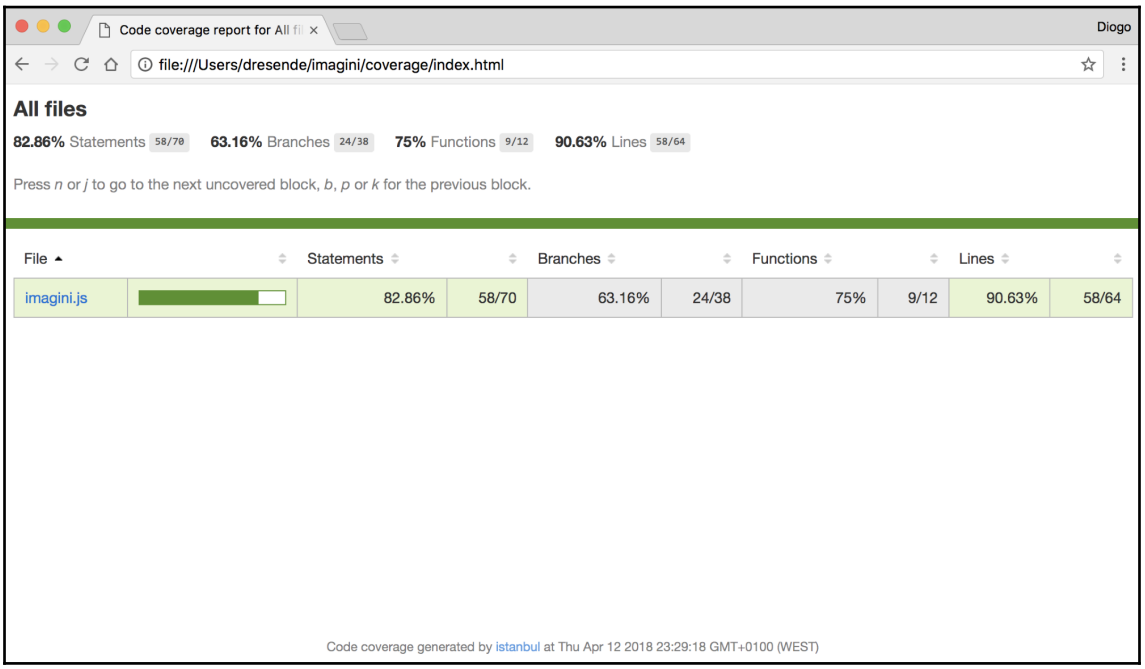
Then, add a resize test:

```
it("should be able to resize the image as we request", (done) => {
    chai
    .request(tools.service)
    .get("/uploads/test_image_download.png?width=200&height=100")
    .end((err, res) => {
        chai.expect(res).to.have.status(200);

        let image = sharp(res.body);

        image
        .metadata()
        .then((metadata) => {
            chai.expect(metadata).to.have.property("width", 200);
            chai.expect(metadata).to.have.property("height", 100);

            return done();
        });
    });
});
```

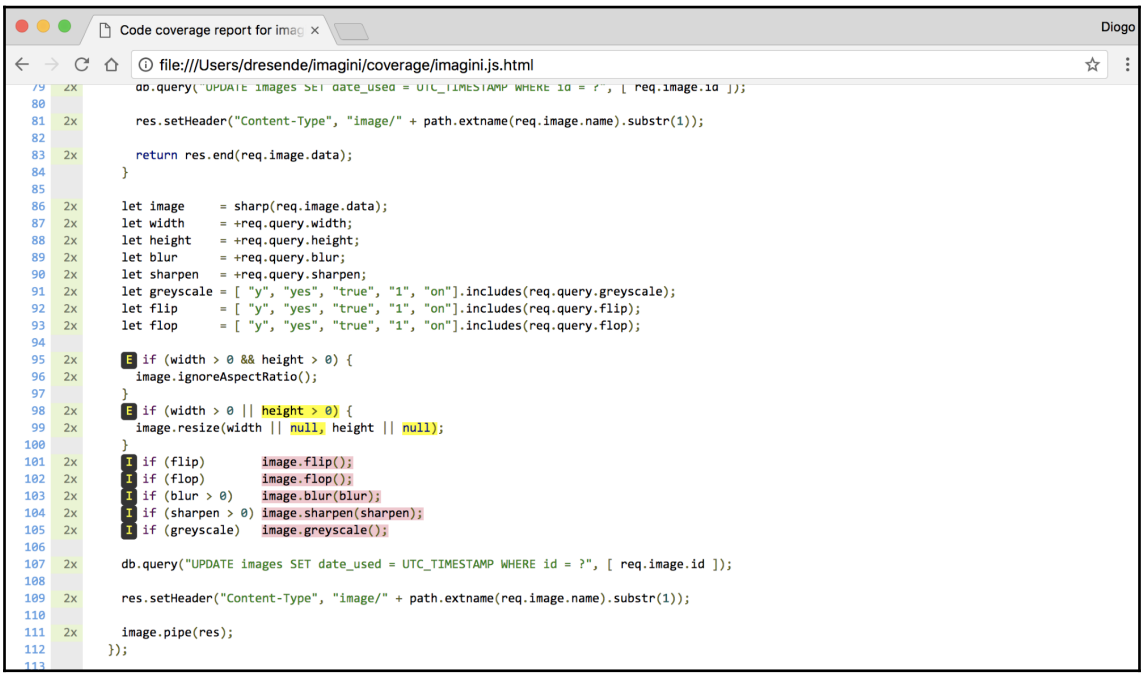
Let's run our test suite:



It now looks very good. From the console report, we can see some green. Let's look at the front page of the coverage report:



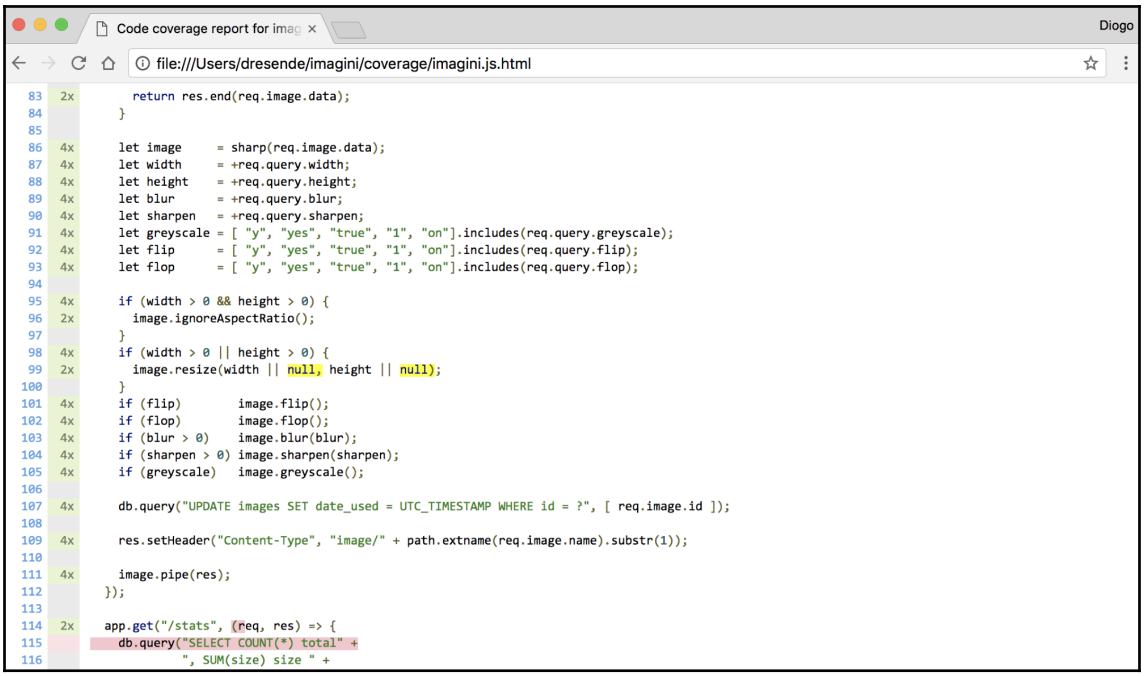
We see green here as well. Having more than *80%* coverage is good, but we can still go further. Let's see the file:



It's more or less covered. We still need to cover all the effects. We can actually run them all at once. The first two conditions also have an `E` marker, but that should disappear after adding a test without resizing. Let's add it:

```
it("should be able to add image effects as we request", (done) => {
    chai
    .request(tools.service)
    .get("/uploads/test_image_download.png?
    flip=y&flop=y&greyscale=y&blur=10&sharpen=10")
    .end((err, res) => {
        chai.expect(res).to.have.status(200);

        return done();
    });
});
```

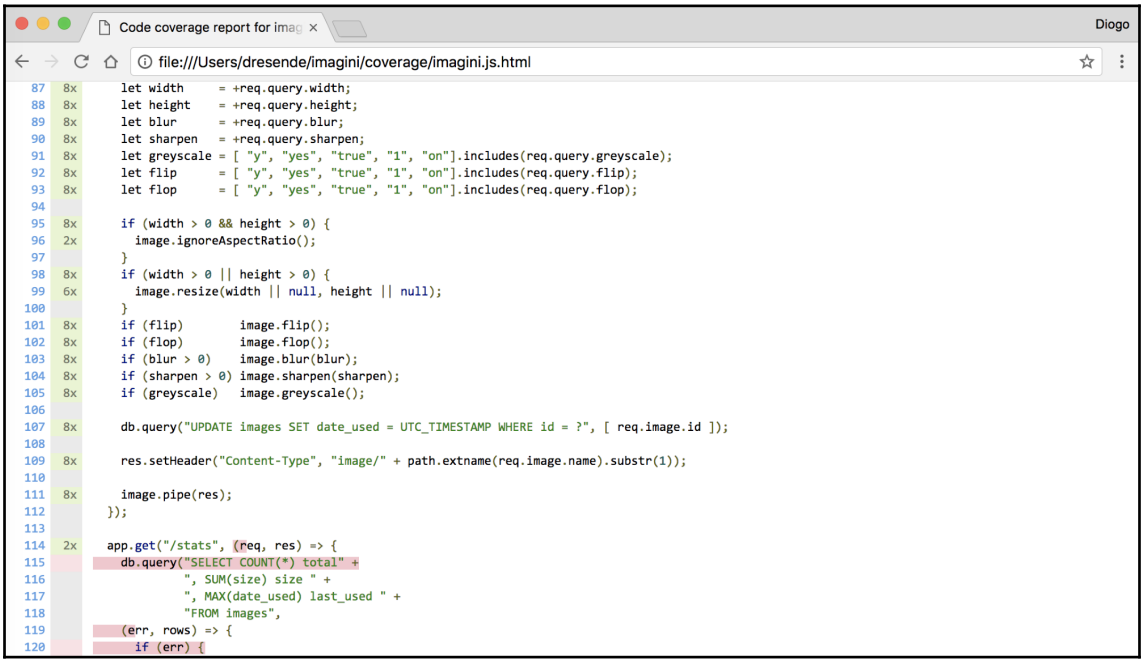
Looking at our report now, we see the coverage is almost complete:



To cover those yellow nulls there, we need to resize the image with only `width` or `height`. We can add two tests for those:

```
it("should be able to resize the image width as we request", (done) => {
    chai
    .request(tools.service)
    .get("/uploads/test_image_download.png?width=200")
    .end((err, res) => {
        chai.expect(res).to.have.status(200);

        let image = sharp(res.body);

        image
        .metadata()
        .then((metadata) => {
            chai.expect(metadata).to.have.property("width", 200);

            return done();
        });
    });
});
```

Add a similar one for the `height`, and run the test suite. You should not see the statement coverage go up, only the branch coverage:

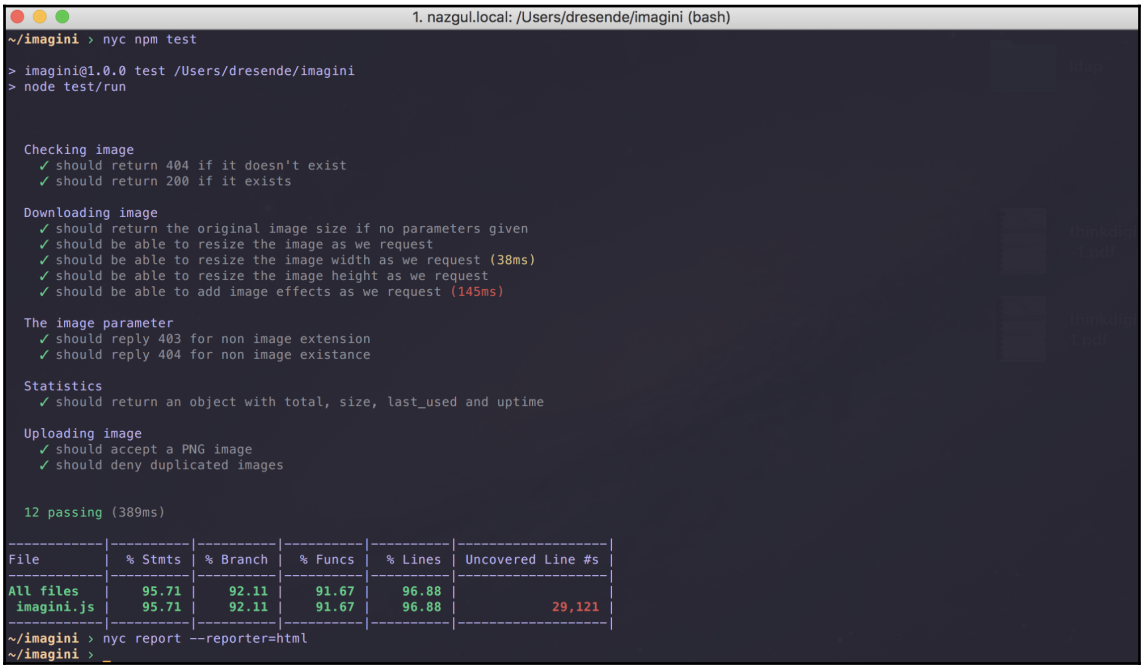


The only method missing is the statistics method. This one is simple. We could eventually run a more specific test, by asking statistics, making a change such as an upload, and asking for statistics again to compare. I'll leave that to you. Let's just add a simple request test:

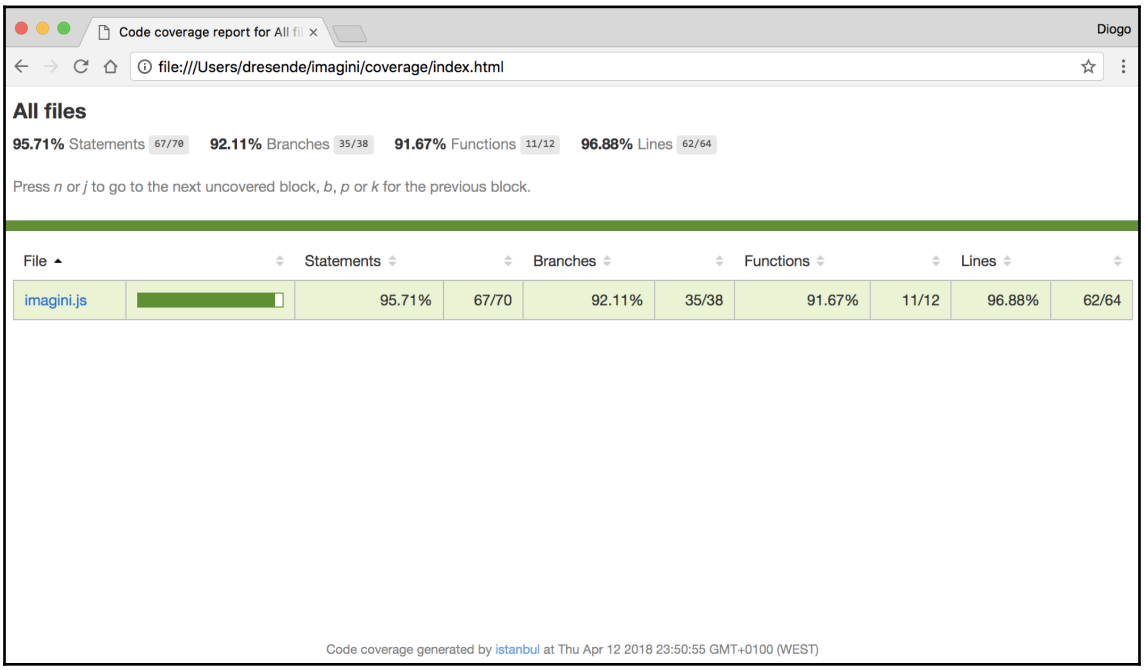
```
const chai = require("chai");
const http = require("chai-http");
const tools = require("../tools");

chai.use(http);

describe("Statistics", () => {
    it("should return an object with total, size, last_used and
    uptime", (done) => {
        chai
        .request(tools.service)
        .get("/stats")
        .end((err, res) => {
            chai.expect(res).to.have.status(200);
            chai.expect(res.body).to.have.property("total");
            chai.expect(res.body).to.have.property("size");
```

```
                chai.expect(res.body).to.have.property("last_used");
                chai.expect(res.body).to.have.property("uptime");

                return done();
            });
        });
    });
```

Now, running our test suite should give all green:



We see there are only two lines uncovered: `29.121`. The first one is our timer and the second one is on the statistics method. Let's refresh our HTML report:



This is rewarding; we have almost *100%* coverage. There's only one function not covered, which is our timer. And, there are only tree statements, which also represent the three branches, that aren't covered, but those aren't actually that important.

What is important is to keep this high coverage mark during the course of our development.

Mocking our services

It's not at all uncommon to have parts of your service that are harder to test. Some, or most, of those parts are error-related conditions, where it's hard to make an external service such as a database engine return an error that will rarely occur during normal execution.

To be able to test, or at least simulate these kinds of events, we need to mock our services. There are a couple of options around, and Sinon is the most commonly used one in the Node.js ecosystem. This framework provides more than mocking; it also provides the following:

- **Spies**: Which monitor function calls and record arguments passed, the returned value and other properties
- **Stubs**: Which are enhanced spies with a pre-programmed behavior, helping us drive the execution into a pre-determined path (allowing us to mock a behavior)

Sinon also allows us to bend time, by virtually changing the service perception of time, and to be able to test timed interval calls (remember our interval timer?). With this in mind, let's see if we can make our microservice reach *100%* test coverage.

Let's start by installing the framework, as we did with `chai`:

```
npm install --save-dev sinon
```

Now, let's add a test for the image deletion. This method is tested through the other tests and that's why we didn't need to add it before, but now that we want to fully test it, let's add a basic test file called `image-delete.js`, with the following content:

```
const chai = require("chai");
const sinon = require("sinon");
const http = require("chai-http");
const tools = require("../tools");

chai.use(http);

describe.only("Deleting image", () => {
    beforeEach((done) => {
        chai
        .request(tools.service)
        .delete("/uploads/test_image_delete.png")
        .end(() => {
            return done();
        });
    });

    it("should return 200 if it exists", (done) => {
        chai
        .request(tools.service)
        .post("/uploads/test_image_delete.png")
        .set("Content-Type", "image/png")
        .send(tools.sample)
        .end((err, res) => {
```

```
                chai.expect(res).to.have.status(200);
                chai.expect(res.body).to.have.status("ok");

                chai
                .request(tools.service)
                .delete("/uploads/test_image_delete.png")
                .end((err, res) => {
                    chai.expect(res).to.have.status(200);

                    return done();
                });
            });
        });
    });
```

Notice that I added the Sinon dependency on top, although I'm not using it just yet. You may run the tests again, but you shouldn't notice any difference.

We'll need to change the database behavior, so let's export a reference to it, so as to be able to access it from the tests. Add the following line in our microservice file before connecting to the database:

```
app.db = db;
```

Now, add another test to that file:

```
it("should return 500 if a database error happens", (done) => {
    chai
    .request(tools.service)
    .post("/uploads/test_image_delete.png")
    .set("Content-Type", "image/png")
    .send(tools.sample)
    .end((err, res) => {
        chai.expect(res).to.have.status(200);
        chai.expect(res.body).to.have.status("ok");

        let query = sinon.stub(tools.service.db, "query");

        query
        .withArgs("DELETE FROM images WHERE id = ?")
        .callsArgWithAsync(2, new Error("Fake"));

        query
        .callThrough();

        chai
        .request(tools.service)
        .delete("/uploads/test_image_delete.png")
```

```
            .end((err, res) => {
                chai.expect(res).to.have.status(500);

                query.restore();

                return done();
            });
    });
});
```

What we're doing is uploading an image, but, before requesting to delete it, we create a `stub` on the `db.query` method. We then inform Sinon that when the `stub` is called with the first argument with `DELETE`, we want it to asynchronously call the third argument (counting starts at 0) with a fake error. For any other call, we want it to just pass through.

Then, after deleting the image, we check that we received an HTTP `500` error code and restore the `stub` to the original function, ensuring that the other tests pass.

We're able to test this because `mocha` runs tests in serial; otherwise, we would need to do some gymnastics to ensure that we wouldn't interfere with the other tests.

Now, open the previously created test file, `image-stats.js`, include Sinon on the top, and add the following test:

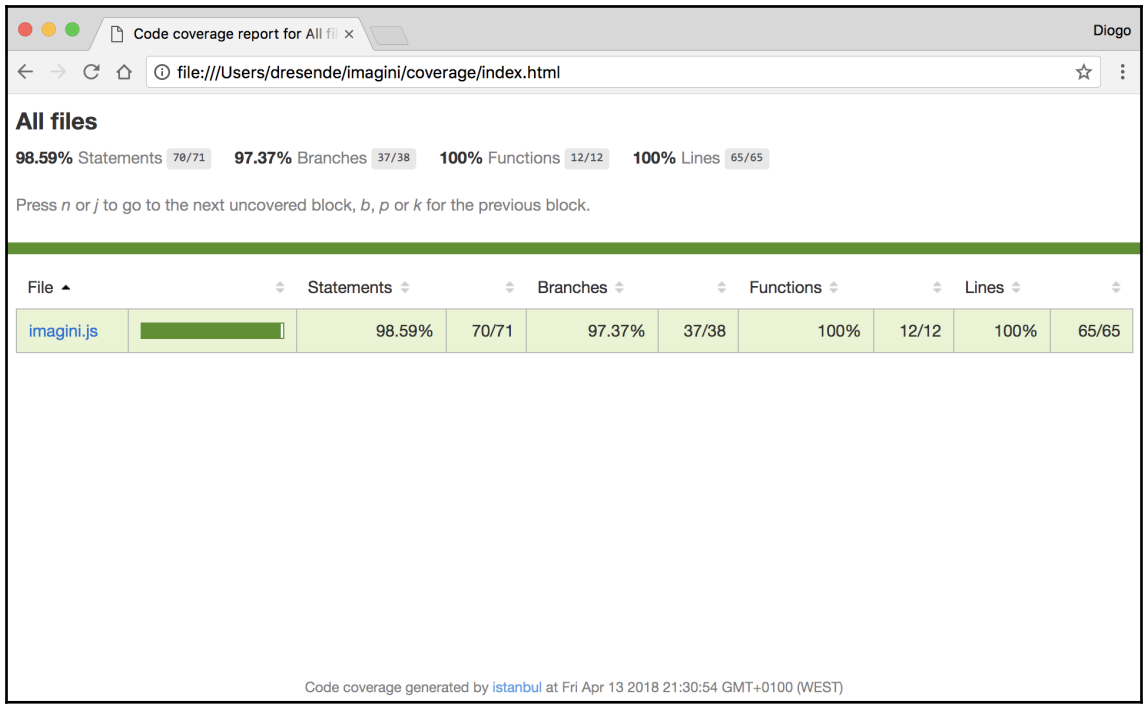
```
it("should return 500 if a database error happens", (done) => {
    let query = sinon.stub(tools.service.db, "query");

    query
    .withArgs("SELECT COUNT(*) total, SUM(size) size, MAX(date_used)
    last_used FROM images")
    .callsArgWithAsync(1, new Error("Fake"));

    query
    .callThrough();

    chai
    .request(tools.service)
    .get("/stats")
    .end((err, res) => {
        chai.expect(res).to.have.status(500);

        query.restore();

        return done();
    });
});
```

We're now over 97% coverage. Let's bend time and test our timer. Create a new test file called `image-delete-old.js`, and add the following content:

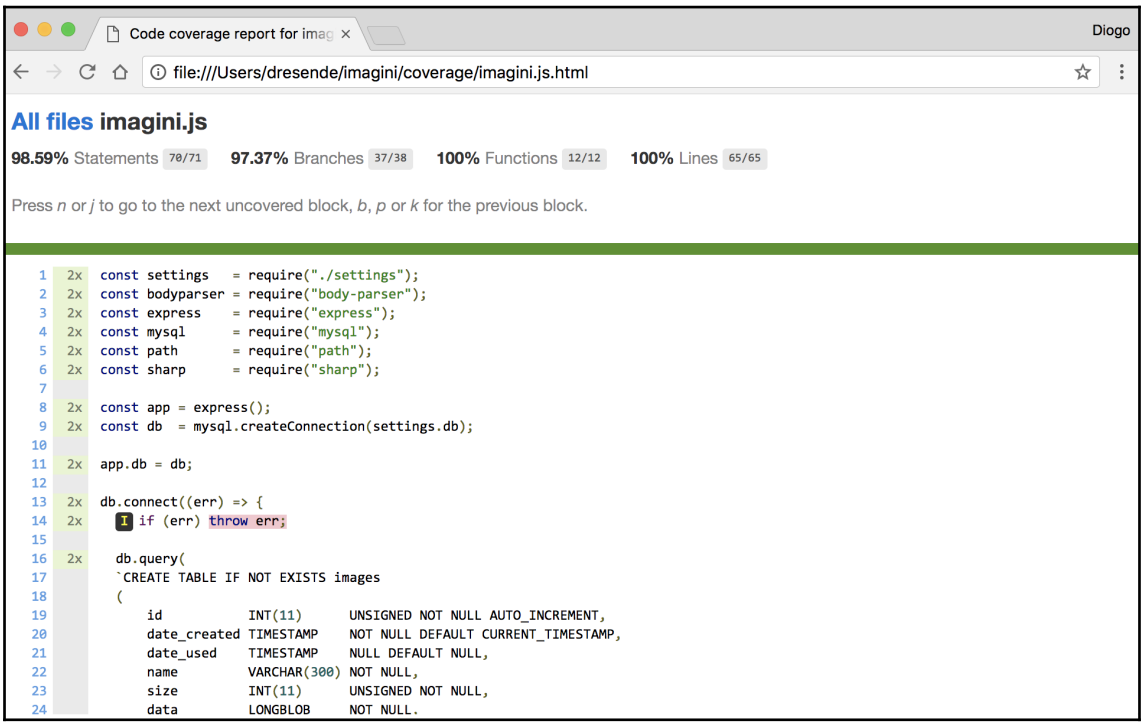
```
const chai = require("chai");
const sinon = require("sinon");
const http = require("chai-http");
const tools = require("../tools");

chai.use(http);

describe("Deleting older images", () => {
    let clock = sinon.useFakeTimers({ shouldAdvanceTime : true });

    it("should run every hour", (done) => {
        chai
        .request(tools.service)
        .get("/stats")
        .end((err, res) => {
            chai.expect(res).to.have.status(200);

            clock.tick(3600 * 1000);
            clock.restore();

            return done();
        });
    });
});
```

In this test, we're replacing the global timer functions (`setTimeout`and `setInterval`) with fake timers. We then make a simple call to statistics, and then advance time by one hour (the tick call), and then finish.

Now, run the tests and see the results:



We now reach *100% coverage* on functions and lines. There's only one branch, with one statement missing. It's the possibility of a connection error:



I'll leave it to you to figure it out how to mock that.

Remember that if you successfully mock the `connect` method, you'll also need to handle the throw.

Summary

Tests allow us to ensure a certain code quality level. It's very important to include tests from the very beginning, while the code is simple, to ensure that we keep tests updated and avoid regressions to the expected behavior.

It's very rewarding when we see that our code has a very high test coverage. This feeling forces you to keep that high mark and indirectly maintains good code quality.

Following on from this methodology of test coverage and continuous expected behavior, let's now look in the next chapter at how to deploy your code, be it for testing, staging, or production, while retaining the same expected behavior, no matter where you're deploying.

8
Deploying Microservices

Now that we're able to test out our microservice, we can go a step further and test it in a controlled environment -- an environment that could, and should, be very similar to a production one.

Being able to develop and test a piece of code in an environment that is a replica of production ensures that you won't find yourself in a situation where your service works in your development environment, but not in production, and you don't know why.

This happens because your development environment is usually your laptop and you certainly use it for other activities. You eventually install tools and applications that change your environment. These tools may introduce you to libraries or interfaces that you then use in your microservice and forget to include those in production.

To ensure your different environments are the same, there are several alternative paths. Let's look at two of them, which are going to be virtual machines and containers. We'll then take the container path and deploy our microservice using Docker.

Using virtual machines

One option is to use a virtual machine, a replica of production, that ensures that the environments are the same. There are a couple of drawbacks to this option:

- It's slow, as there's a complete virtualization guest operative system that you have to boot every time you want to develop
- It's resource consuming, as you need to store a complete base layout in your disk and you need to reserve RAM memory to start it

To make things worse, having a read-only machine with read-write code is not simple and will eventually make you frustrated. Virtual machines are the only option when you need an environment that you cannot have unless the entire machine is virtualized, for example, when you have a macOS laptop and you need a Windows environment.

Other than that, like when you're developing a microservice, the environment should be common to most operative systems. Node.js runs across platforms and, in this case, you're just exposing an HTTP interface.

Using containers

Another option is to use a container. A container is an operative system-level virtualization mechanism, where you isolate an environment inside your operative system. You still need the space for a base layout, but there are container layouts under 5 MB.

The most common and widely used container environment is Docker, which uses Linux containers and cgroups to create a secure and isolated environment to run applications. What boosted Docker actually was was the centralized repository of base layouts, called Docker images, where people could define, build, and then share these images with others, so people wouldn't have to build them themselves. There are pre-built images for all major database services and programming languages, so it's easy to just choose one and run your microservice inside it in a couple of minutes.

To make things even better, there's a description file, called `Dockerfile`, which has all the instructions you need to build an image locally. This is a text file that you can store in your code repository that allows developers to build an image to run the environment.

Even more interesting is the fact that images can be built on top of other images. You can pick one, add your specifications, and publish it. This is one of the major advantages that drove Docker to be the most used container technology.

There are a couple of Docker idiosyncrasies that you must understand before using it:

- Docker images are like a read-only template for running your application. There are ways of saving changes to images, but usually you pick an image and run your application inside an environment that will get lost when the container is removed. This is actually great because it ensures that when you restart your environment, everything is back to a common environment and nothing is left over.
- A container should run only one command and nothing more. This command can, of course, spawn other commands, but the purpose of a container is not to run multiple commands in parallel. If you have a complex environment, each service should run in its own container. For example, if you have a database, the database service could run in one container and your application should run on another.
- Because an image is supposed to be read-only, you can mount or link folders from the outside of the container. These are called volumes and allow you to, for example, run a read-only MySQL image with the data folder mapped to a folder on your laptop, allowing you to avoid losing your data when you stop the container. Volumes can either be folders or files and they can also be mounted read-only from the inside of the container.
- A container runs in an isolated environment and you need to create network links from your interface ports to the container ports. There's also an option to create virtualized networks that work across hosts and give you the power to deploy more complex environments.

Deploying using Docker

Before we start using Docker, you obviously need to install it. There are several installation channels to choose from. For many Linux distributions, there are packages available in their package manager. For Windows and macOS, you need to go to the Docker website and download the installer. This is actually the preferred one as the versions available from the package managers may not be up to date.

There are several graphical interfaces to manage Docker, but we'll keep it simple and just use the command-line interface. This gives you greater control and you get to know more of the internals of how everything is glued together.

So, let's open a console and type `docker info` to check whether everything is correct:

You can see that there are no containers or images running. There is other information, such as storage and network information. You may also figure out that Docker for Mac uses a virtual machine to enable Docker on a macOS laptop.

This means that the `docker` command is actually a proxy to the `docker` command inside that virtual machine. You don't have to worry about it, it will work just fine; you'll probably only notice some lag from time to time.

Creating images

Before we deploy, we need to create an image for our service to run inside. Our service has no dependencies other than the ones defined on the top:

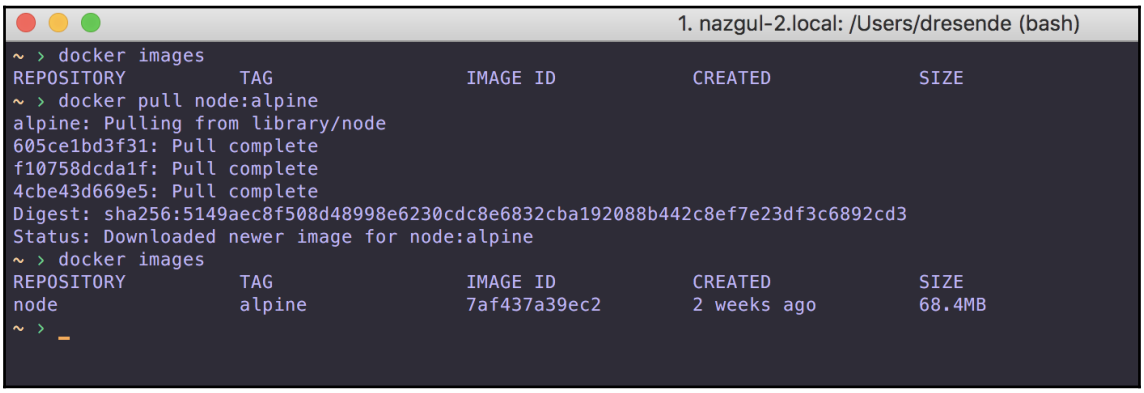
- `express` and `body-parser`: To handle our HTTP requests
- `mysql`: It enables us to use a MySQL database
- `sharp`: It enables us to manipulate images
- `path`: It is a core module to help working with directory paths

The core module is not a problem, and the first three are also not a problem since they're built completely in JavaScript. On the other hand, `sharp` is compiled, so our image needs to have at least a compiler.

To keep our image as slim as possible, we'll be using the Node.js Alpine version, which is a Node.js image based on Alpine Linux and it has a much smaller size than other distributions. You'll see what I mean in a moment. Let's download the image. Run the following:

```
docker pull node:alpine
```

If everything goes well, you should then have a new container image available, as the following screenshot shows:



First, I just checked that we had no local images. Then, I requested that Docker pull the Alpine version of the Node.js official repository. Notice that I didn't specify the version, as I'm targeting the latest stable one. If you want a specific version, head to Docker Hub and search for Node official images.

We now have our image, but there are a couple of things to do before being able to start our microservice. First, we need to create our own image. What we downloaded is just a base image, which doesn't have our code. We need to add our code and dependencies. To do that, we need to create a `Dockerfile`.

Defining a Dockerfile

This is a file specification with instructions for Docker to create an image. The instructions are clear and allow you to share that file on several servers so that Docker builds the image.

Our `Dockerfile` will have three parts:

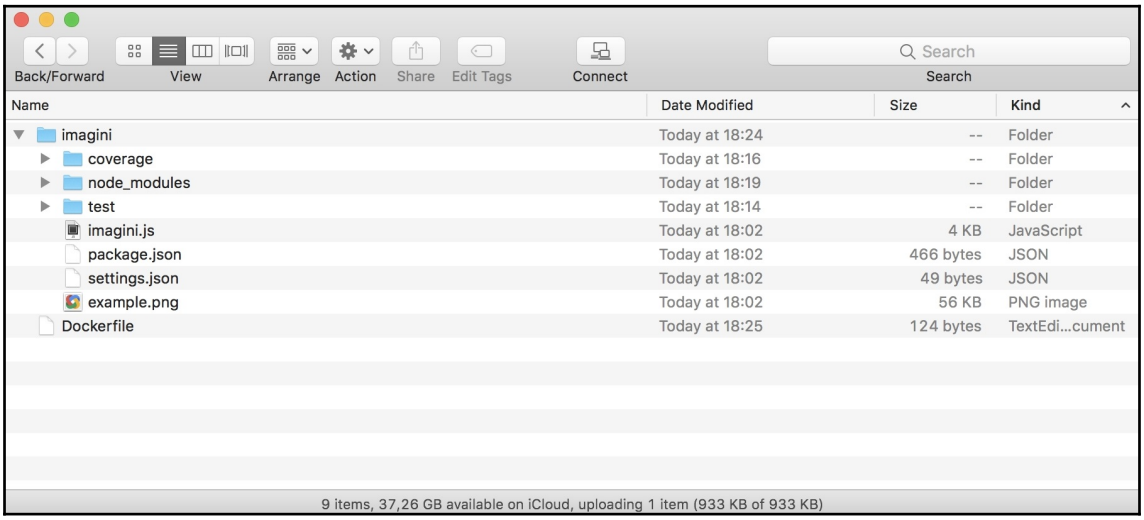
- The header, which indicates the base image and author
- The instructions to build the image
- The instructions to run the image

There are many more instructions that we'll probably not cover. To get to know `Dockerfile` better, head over and read the documentation. This is an introduction that will actually help you go a long way. Here's our first `Dockerfile`:

```
FROM node:alpine
MAINTAINER Diogo Resende

ADD imagini /opt/app

WORKDIR /opt/app
RUN npm i

CMD [ "node", "/opt/app/imagini" ]
```

The empty lines were added for clarity only. The first two instructions define the base image (`FROM`) and the author (`MAINTAINER`). The next three add our `imagini` folder to the `/opt/app` folder inside the image. Then, we change directory to that folder and run `npm i` to install the required dependencies. Finally, the last instruction indicates how to run the image later on.

Save this to a file called `Dockerfile` and place it in an empty folder. Then, grab our microservice and place it inside a folder called `imagini`, inside this newly created folder:



We can now try to build our image using the `docker build` command. Let's create the image, assign the name of our service, and indicate an initial version:

```
> docker build -t imagini:0.0.1 .
Sending build context to Docker daemon   43.4MB
Step 1/6 : FROM node:alpine
 ---> 7af437a39ec2
Step 2/6 : MAINTAINER Diogo Resende
 ---> Running in 009093d8c9d4
Removing intermediate container 009093d8c9d4
 ---> cdbe5185faf4
Step 3/6 : ADD imagini /opt/app
 ---> 9c062473535a
Step 4/6 : WORKDIR /opt/app
Removing intermediate container d734af3adba8
 ---> e39bdbfaf5ae
Step 5/6 : RUN npm i
 ---> Running in 3437fc2f74c9
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN imagini@1.0.0 No description
npm WARN imagini@1.0.0 No repository field.

up to date in 0.613s
Removing intermediate container 3437fc2f74c9
 ---> 0b50046161f3
Step 6/6 : CMD [ "node", "/opt/app/imagini" ]
 ---> Running in 8d5786ea6ecf
Removing intermediate container 8d5786ea6ecf
 ---> a3e21d6fd379
Successfully built a3e21d6fd379
Successfully tagged imagini:0.0.1
> _
```

Let's check the list of images available:

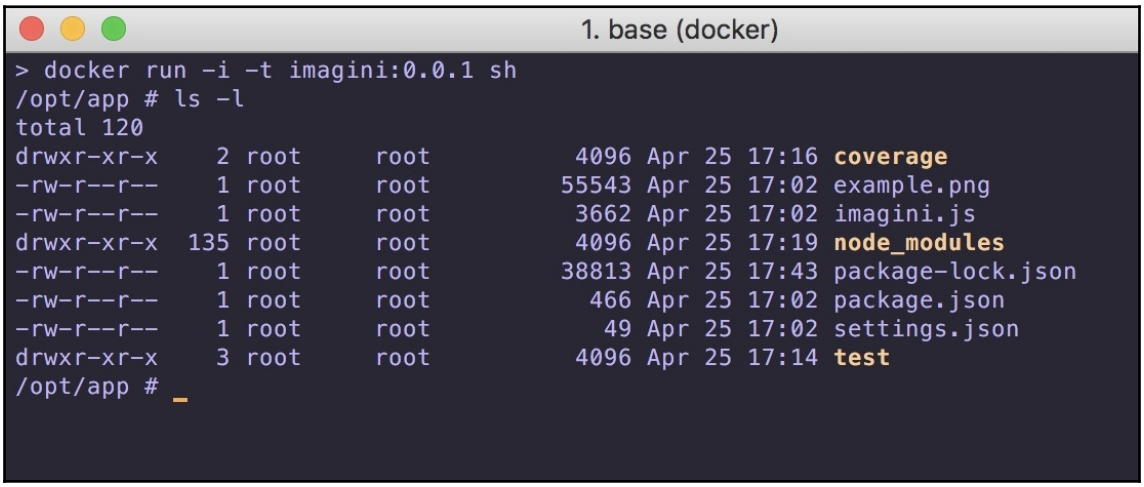
```
> docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
imagini             0.0.1               a3e21d6fd379        22 seconds ago      110MB
node                alpine              7af437a39ec2        2 weeks ago         68.4MB
> _
```

Let's just start a container and see what happens. To do that, we use the `docker run` command and pass our image name and version. We don't need to specify what to run inside the image since we defined that on the `Dockerfile`, but we could change that:

```
1. base (bash)
> docker run imagini:0.0.1
internal/modules/cjs/loader.js:683
  return process.dlopen(module, path.toNamespacedPath(filename));
                 ^

Error: Error loading shared library /opt/app/node_modules/sharp/build/Release/sharp.node: Exec format error
    at Object.Module._extensions..node (internal/modules/cjs/loader.js:683:18)
    at Module.load (internal/modules/cjs/loader.js:566:32)
    at tryModuleLoad (internal/modules/cjs/loader.js:506:12)
    at Function.Module._load (internal/modules/cjs/loader.js:498:3)
    at Module.require (internal/modules/cjs/loader.js:598:17)
    at require (internal/modules/cjs/helpers.js:11:18)
    at Object.<anonymous> (/opt/app/node_modules/sharp/lib/constructor.js:10:15)
    at Module._compile (internal/modules/cjs/loader.js:654:30)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:665:10)
    at Module.load (internal/modules/cjs/loader.js:566:32)
> _
```

Well, I expected that to happen. I did this deliberately so that you never forget that dependencies should be held carefully. Just as you don't push dependencies to a git repository, you don't push dependencies to an image. Instead, you should install them inside the image. That's one of the reasons the image built so quickly because NPM assumed that everything was all right.

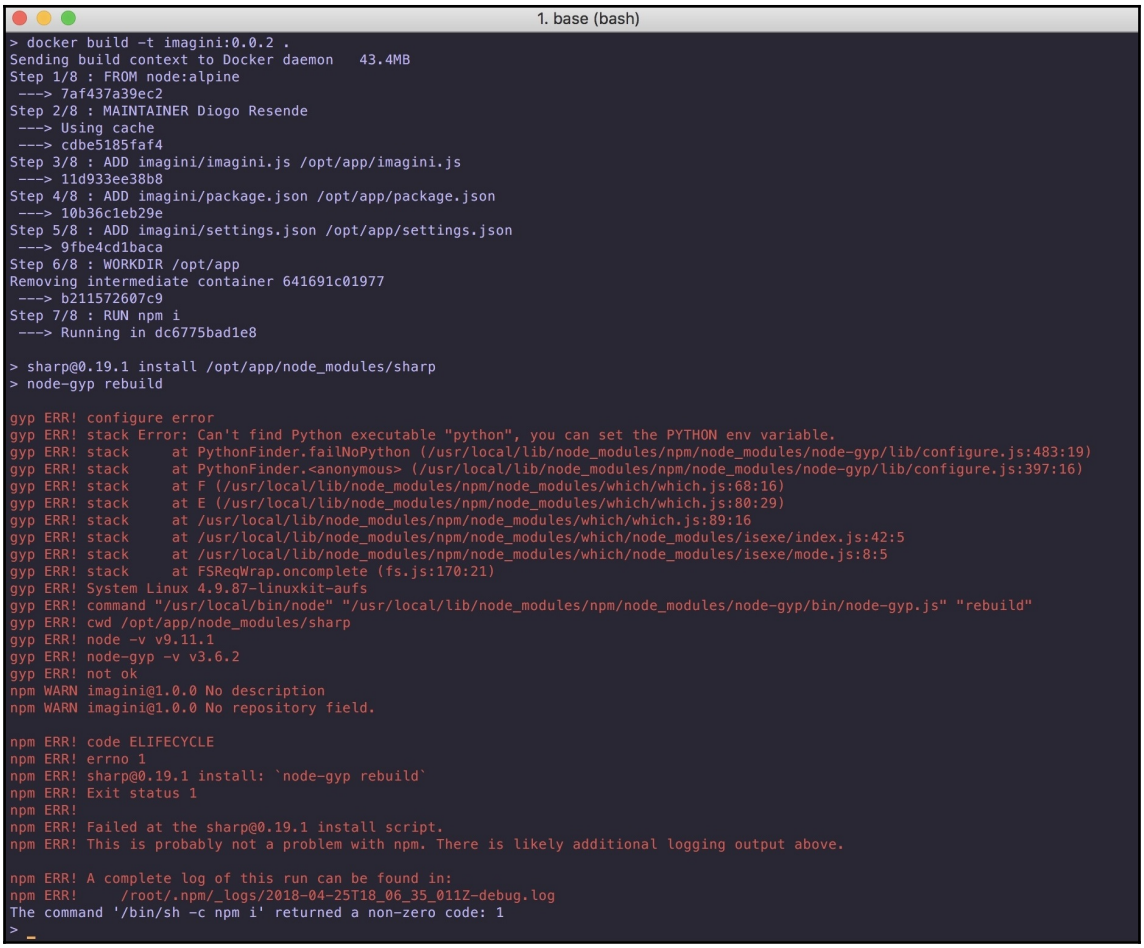
Let's change our `Dockerfile` and create a new image, but before that, let's go inside the image and look at it. To do that, let's run the container specifying another command to execute. We also need to specify a few more parameters, namely `-i` and `-t` for an interactive terminal. Let's run a console (we don't have bash in Alpine, so we'll stick with `sh`):



This is what's inside the `/opt/app` folder inside the image. There are a couple of things we don't need to add to, like the test and coverage folders. We can just stick to the microservice, the dependencies file, and the settings file:

```
FROM node:alpine
MAINTAINER Diogo Resende

ADD imagini/imagini.js /opt/app/imagini.js
ADD imagini/package.json /opt/app/package.json
ADD imagini/settings.json /opt/app/settings.json

WORKDIR /opt/app
RUN npm i

CMD [ "node", "/opt/app/imagini" ]
```

The `ADD` instruction can be used with files, not just folders. This will keep our image as slim as possible. Let's run it and assign a new version:



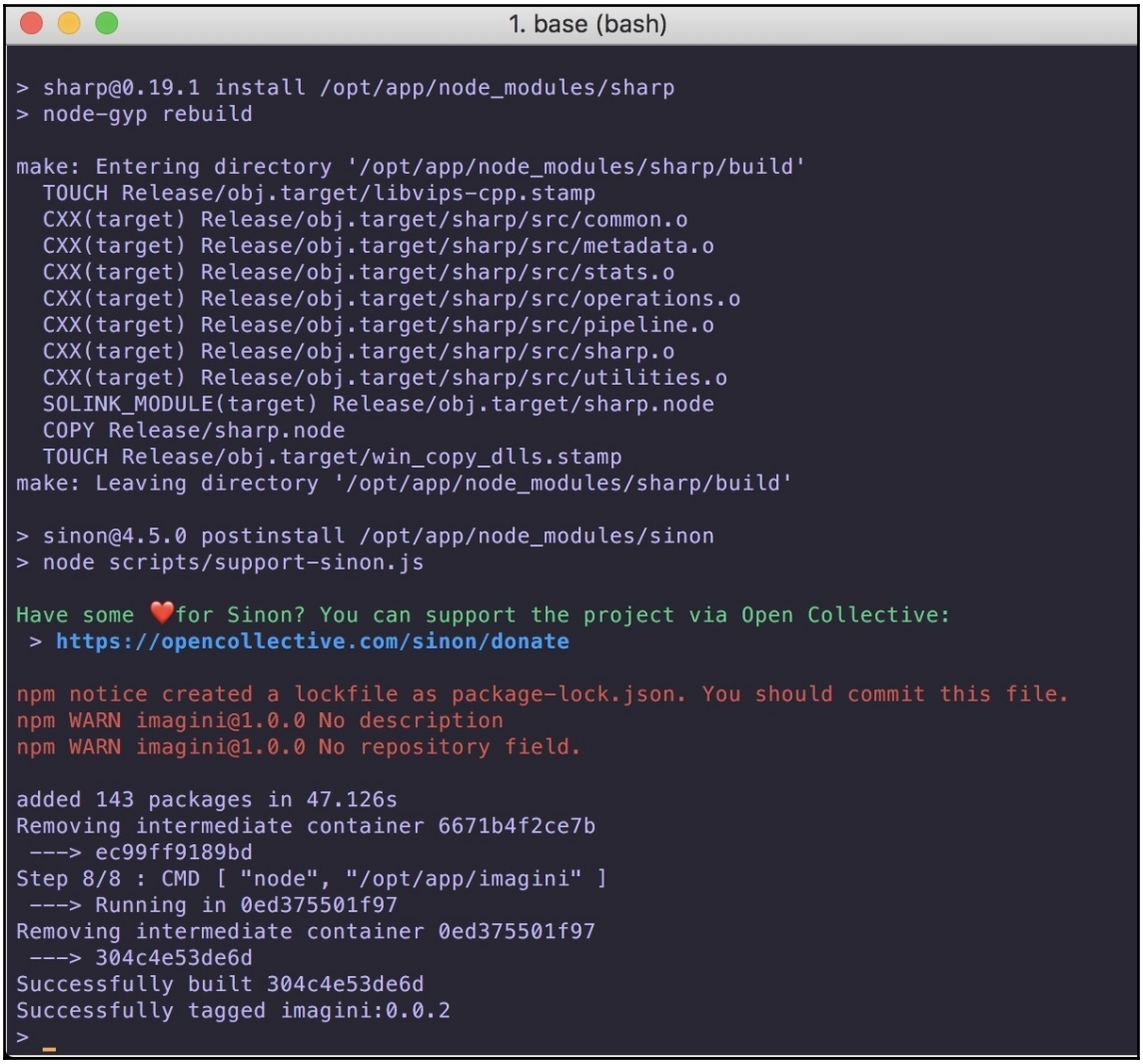
That looks better, despite the error. NPM tried to install `sharp` but our image is so slim it doesn't have Python installed. Actually, it's not only Python that will be missing; you'll find there are no build tools for you to compile anything. If you really need a very slim image, you can this way and see what other dependencies you'll need.

For demonstration purposes, let's switch our base image to the standard version:

```
FROM node
MAINTAINER Diogo Resende

ADD imagini/imagini.js /opt/app/imagini.js
ADD imagini/package.json /opt/app/package.json
ADD imagini/settings.json /opt/app/settings.json
```

```
WORKDIR /opt/app
RUN npm i

CMD [ "node", "/opt/app/imagini" ]
```

See how we change the first line to just `node`. If you now run the `docker build` command, Docker will download the standard image and build our image in a single operation:



As we can see, NPM now compiled our sharp dependency.

Note that if you only rely on simple dependencies, you can stick with the Alpine version.

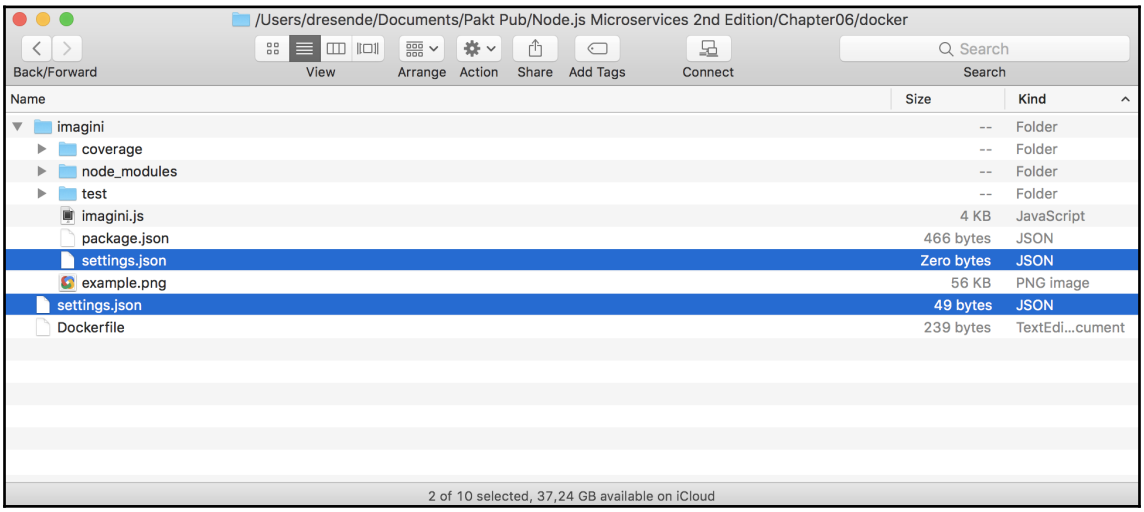
Let's try to run our container again:

```
1. nazgul-2.local: /Users/dresende (bash)
~ › docker run imagini:0.0.2
/opt/app/imagini.js:14
        if (err) throw err;
                 ^

Error: connect ECONNREFUSED 127.0.0.1:3306
    at TCPConnectWrap.afterConnect [as oncomplete] (net.js:1174:14)
    --------------------
    at Protocol._enqueue (/opt/app/node_modules/mysql/lib/protocol/Protocol.js:145:48)
    at Protocol.handshake (/opt/app/node_modules/mysql/lib/protocol/Protocol.js:52:23)
    at Connection.connect (/opt/app/node_modules/mysql/lib/Connection.js:130:18)
    at Object.<anonymous> (/opt/app/imagini.js:13:4)
    at Module._compile (internal/modules/cjs/loader.js:654:30)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:665:10)
    at Module.load (internal/modules/cjs/loader.js:566:32)
    at tryModuleLoad (internal/modules/cjs/loader.js:506:12)
    at Function.Module._load (internal/modules/cjs/loader.js:498:3)
    at Function.Module.runMain (internal/modules/cjs/loader.js:695:10)
~ › _
```

Well, we forgot our settings file inside the container. That settings file points to the MySQL server on the local address, which is actually wrong because the container has its own interface.

To make our image free of passwords and a little bit usable, we need to have our `settings` file outside the image, and every time we want to run the image, we indicate the `settings` file we want to use. This enables you to have different deployments with different settings sharing the same base image.

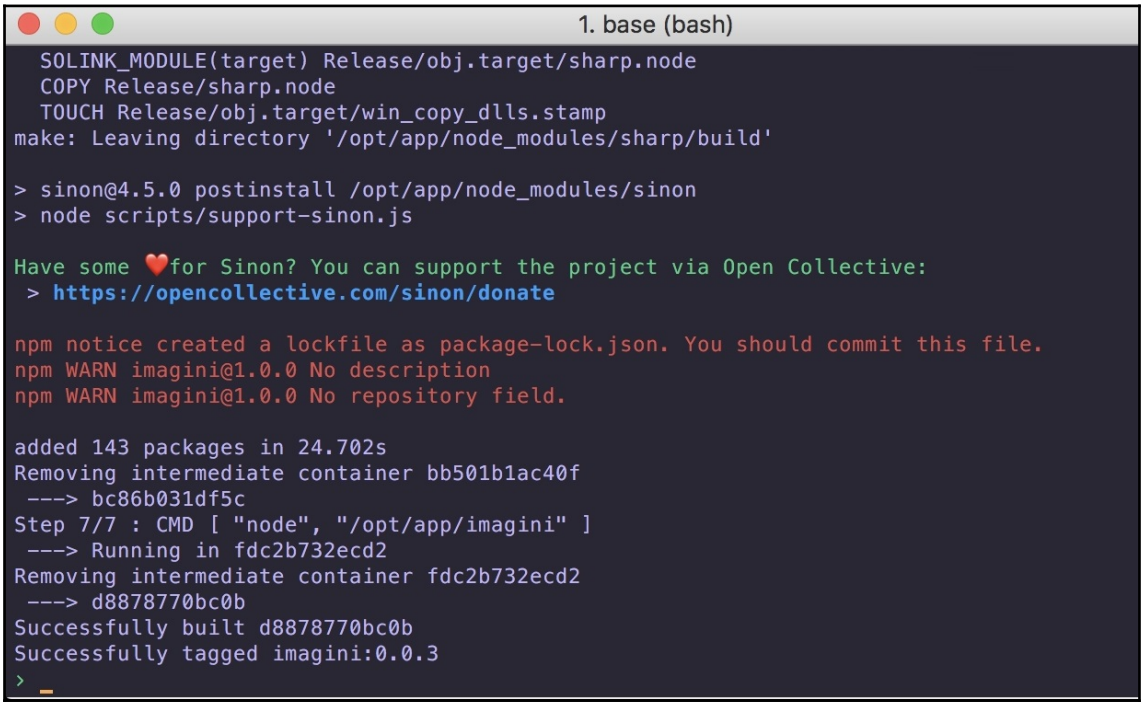
But, because of how Docker mounts, we need to have the file there already, even if it's empty, to avoid Docker mounting as a directory. So, let's move the `settings` file to the parent folder, and create an empty `settings` file where it was before. You should end up with a structure similar to the following screenshot:



So, we should keep the same image instructions, but now with an empty settings file:

```
FROM node
MAINTAINER Diogo Resende

ADD imagini/imagini.js /opt/app/imagini.js
ADD imagini/package.json /opt/app/package.json
ADD imagini/settings.json /opt/app/settings.json

WORKDIR /opt/app
RUN npm i

CMD [ "node", "/opt/app/imagini" ]
```

Now, let's create a new version:



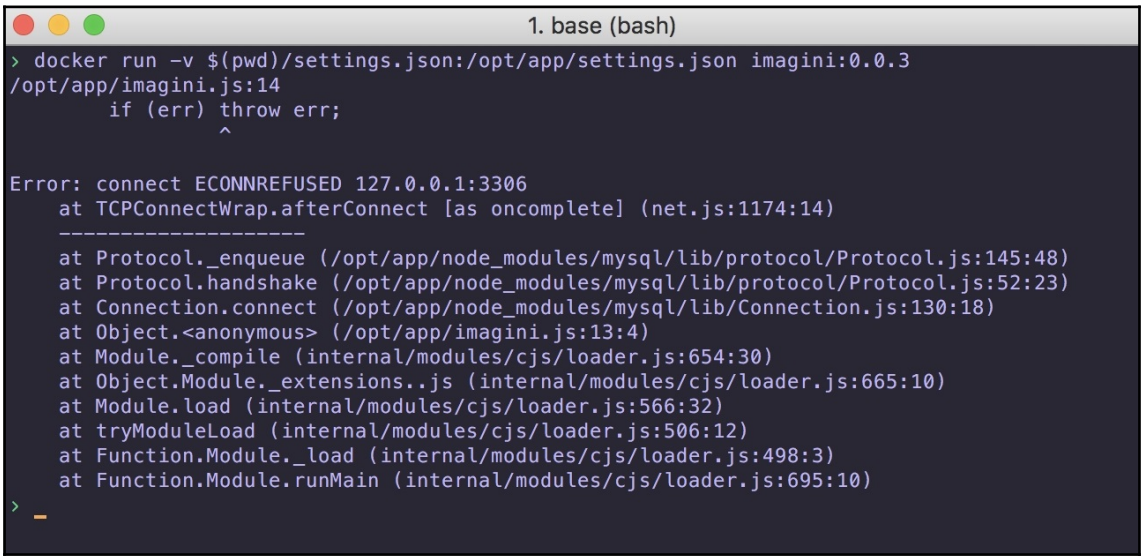
There are two things you should know about building an image:

- Since we haven't published the image to the public, and we're still testing, we could overwrite our image version and avoid incrementing the revision value. We're doing so just for you to see how the community usually adds new versions.
- If you read the log lines of the build, you'll notice some lines that contain `Using cache`, which indicate that Docker found that instruction step in a previous build and is using that instead of rebuilding. If we had the `ADD` instructions after the dependency installation, you would probably see that installation using cache and taking much less time to build.

To run our container with our image and our settings attached, we need to use the `-v`, which stands for volume, and allows us to mount a folder or a file from our local filesystem to the container filesystem. Assuming we're in a console on our root folder from the preceding screenshot, we should run the following command:

```
docker run -v $(pwd)/settings.json:/opt/app/settings.json imagini:0.0.3
```

Docker enforces us to use full paths, so I just added another bash interpolation to get the current working directory (`pwd`). Let's run it:



Well, that's another property of containers; they run isolated and have their own network interface. That interface is attached to a virtual switch created by Docker, which has no outside connectivity unless you say so.

We're using localhost as the database server hostname, which, in this case, points to the container itself. We need to change this to the local address of our system. Since our local system may have a changing IP address, Docker has a special DNS address, which is `host.docker.internal`, that points to our host.

I'm changing my `settings` file to something like the following lines. Change it according to your previous configuration:

```
{
    "db": "mysql://root:root@host.docker.internal/imagini"
}
```

If we run it again, we should be able to finally start our container:

```
1. base (docker)
> docker run -v $(pwd)/settings.json:/opt/app/settings.json imagini:0.0.3
```

Managing containers

You may notice that the container is running in the foreground, which is not very useful because your console will be blocked until the service ends. Because our service is supposed to never stop, we should try running the service in the background. If you open another console and list the running container processes, you'll see that it's running:

```
1. base (bash)
(docker) ⌘1  (bash) ⌘2
> docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS               NAMES
76f51f26869c        imagini:0.0.3       "node /opt/app/imagi…"   About a minute ago   Up About a minute                       epic_kowalevski
>
```

You can see the `CONTAINER ID`, which, along with the container name, can be used to perform several actions such as restarting and removing containers, as well as the image, and its status and uptime.

Let's stop our container and measure the time to stop it. You'll notice that it won't stop immediately:

```
1. base (bash)
(bash) ⌘1  (bash) ⌘2
> docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS               NAMES
76f51f26869c        imagini:0.0.3       "node /opt/app/imagi…"   About a minute ago   Up About a minute                       epic_kowalevski
> time docker stop 76f51f26869c
76f51f26869c

real    0m10.604s
user    0m0.061s
sys     0m0.024s
>
```

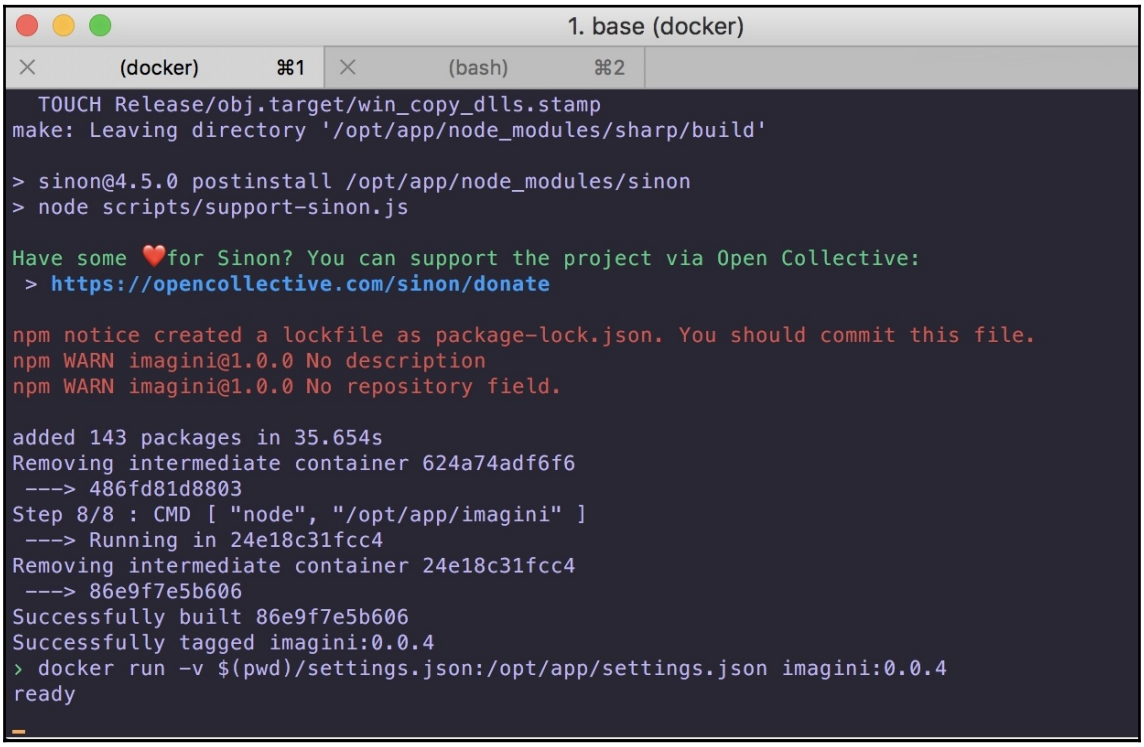
It took *10* seconds, which is a lot if you just made a change and want to restart it. This is because our service is ignoring any environment signals.

When trying to stop a container, Docker sends a `SIGTERM` signal and waits *10* seconds for it to stop. If it doesn't, Docker then sends a `SIGKILL`, which will escalate to the kernel and which, in turn, will immediately stop our service and the container will stop.

We can change this behavior and try to stop gracefully. Change our `app.listen` line to something along the following lines:

```
app.listen(3000, () => {
    console.log("ready");
});

process.on("SIGTERM", () => {
    db.end(() => {
        process.exit(0);
    });
});
```

This adds a `ready` line when starting the container and tries to close both the database connection and the HTTP server when receiving a `SIGTERM`. Let's build our image again (because we changed our service), this time as version `0.0.4`, and run it again:



Head to the other console and try to stop the container again:

That was way faster. It could be even faster, but it's better to stop gracefully. We're just stopping the database connection, but you could also stop Express from accepting more connections and then wait some seconds while it handles the active connections before closing gracefully.

We can now continue to improve on our deployment by changing our container to run in the background instead of just blocking our console. This way, we don't need two consoles to deploy and manage our container. To run this in the background, we use the `-d` options, which enables detached mode.

Our updated command is now:

```
docker run -d -v $(pwd)/settings.json:/opt/app/settings.json imagini:0.0.4
```

If you try it out, Docker will inform you of the full `CONTAINER ID` and turn you back to the console:

If you check the running containers, you'll see our container, with the short ID (which corresponds to the first *12* characters) and some more information. Something that is actually important and is empty is the `PORTS` column, which indicates the container ports that are linked to ports on our host.

That column being empty means you have no access to the container other than through Docker itself, which is actually useless. We can't access our HTTP interface.

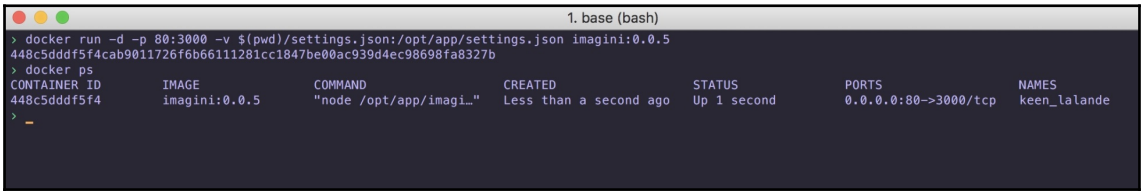
We need to change our image again and expose a port. To do that, we must use the `EXPOSE` instruction and pass the port we want to expose. Our service listens on port `3000`, so we need to expose it:

```
FROM node
MAINTAINER Diogo Resende

ADD imagini/imagini.js /opt/app/imagini.js
ADD imagini/package.json /opt/app/package.json
ADD imagini/settings.json /opt/app/settings.json

WORKDIR /opt/app
RUN npm i

EXPOSE 3000

CMD [ "node", "/opt/app/imagini" ]
```

You can now build the image, but before using our new version, we need to change our run command. Just because we're exposing a port does not mean the person using our image wants those ports exposed. We need to specify what host ports we want to connect to the container ports. To accomplish that, we're going to use `-p`, which allows us to configure incoming ports. The syntax is similar to the previous `-v` but for ports:

```
docker run -d -p 80:3000 -v $(pwd)/settings.json:/opt/app/settings.json
imagini:0.0.5
```

In this case, we're exposing local port `80` to port `3000` of the container:



You can see that the container is running and that the address `0.0.0.0:80` (all our addresses at port `80`) point to the container port `3000`. We should be able to try it out and see if it's running correctly. Open a web browser and point it to the `stats` path: