# Getting Started with Node.js and Jasmine

Jasmine is a BDD style testing library for Node.js. This article will walk you through its installation, configuration and usage.

It's a shame that, almost since its creation, JavaScript had been considered more of a quick hack than an actual programming language. Lacking the tools and the community that other popular languages had, for a long time it was a language that many people enjoyed to hate. Fortunately, with the inception of [Node.js](http://nodejs.org/), JavaScript got a chance to shine with an awesome community and the toolset it always deserved.

This article shows how to leverage the power that these new tools bring by creating a [Jasmine](http://jasmine.github.io/)-tested Node.js web server. We will create a simple Hello World web server and practice our Jasmine test-writing skills on it.

**Bootstrapping the Server**

We'll start by creating an empty Node.js project:

mkdir hello\_world

cd hello\_world

To set up and download various dependencies, we will use Node's official package manager —[npm](https://www.npmjs.com/). One of the nice features that npm offers is initializing an empty project directory using itsinit command line option. Running npm init will create an empty package.json file and launch a wizard asking some basic questions about your project so it can populate it with data. Answer the questions according to the following package.json file:

{

"name": "Hello World",

"version": "0.0.1",

"description": "Simple Hello World demo app.",

"main": "index.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1"

},

"author": "",

"license": "ISC"

}

With this in place, we will set up two directories in our project's root directory — one for the server's source code, and another one for the Jasmine test files. By convention, tests written in Jasmine are called *specs* (short for specification), and are stored inside the spec directory.

mkdir app

mkdir spec

At this point, your project tree should have the following structure:

.

├── app

├── node\_modules

├── package.json

└── spec

**Installing Dependencies**

Our project will require several dependencies.

For writing specifications in Jasmine, we need to install its npm package. Run the following command in your project's root directory:

npm install jasmine-node --save

The --save option will automatically append the package to your package.json file and save the version of the package that your project is depending on.

Later in this tutorial, we will execute several requests facing our application. We will use the[request](https://github.com/request/request) npm package for that purpose. Similar to the above package, we will install it with:

npm install request --save

The last package we need to install is [Express.js](http://expressjs.com/). This package defines a simple DSL (domain specific language) for routing and handling incoming HTTP requests. Install it with:

npm install express --save

One final step is needed before we can start writing some code. We will set up npm's testcommand to run Jasmine specs. Our jasmine-node is installed locally in the node\_packagesdirectory inside our project's root directory. To run it, invoke its binary, and point it to our spec folder. The following command does just that:

./node\_packages/.bin/jasmine-node spec

We will now put the above command in the package.json file, after which that file should look like this:

{

"name": "Hello World",

"version": "0.0.1",

"description": "Simple Hello World demo app.",

"main": "index.js",

"scripts": {

"test": "./node\_modules/.bin/jasmine-node spec"

},

"author": "",

"license": "ISC"

}

**A Hello World Server**

Let's start by *describing* our web server in a new spec/hello\_world\_spec.js file.

touch spec/helloWorldSpec.js

Jasmine specs are a *description* of a feature, or a unit of code. This is why the specs usually start with a describe block that contains tests connected with that feature.

describe("Hello World Server", **function**() {

});

The first argument gives a short description to the tested feature, while the second argument is a function that executes its expectations.

The first thing we want to test on our server is whether it is returning the HTTP status OK (status code 200) when we send a GET request towards its root path.

Let's describe this behaviour with the following Jasmine description:

describe("Hello World Server", **function**() {

describe("GET /", **function**() {

});

});

Now, let's write our first expectation. We will use Jasmine's it block to set up and test the status code that our server returned:

describe("Hello World Server", **function**() {

describe("GET /", **function**() {

it("returns status code 200", **function**() {

});

});

});

Next, we will use the request package to send a GET request toward our web server. We can load the package with the require keyword in Node.js, and to execute a request, we need to pass it a URL and a function that it will invoke once the request is returned.

**var** request = require("request");

**var** base\_url = "http://localhost:3000/"

describe("Hello World Server", **function**() {

describe("GET /", **function**() {

it("returns status code 200", **function**() {

request.get(base\_url, **function**(error, response, body) {

});

});

});

});

Finally, we can now write our first expectation. Jasmine implements expectations with theexpect() function that tests if a tested object *matches* some of the expected conditions. For example, the following expect will match the value of the response.status to the number 200, with the .toBe matcher:

**var** request = require("request");

**var** base\_url = "http://localhost:3000/"

describe("Hello World Server", **function**() {

describe("GET /", **function**() {

it("returns status code 200", **function**() {

request.get(base\_url, **function**(error, response, body) {

expect(response.statusCode).toBe(200);

});

});

});

});

Of course, there are many more matchers in Jasmine. You can take a look at them [on Jasmine's documentation page](http://jasmine.github.io/2.0/introduction.html#section-Included_Matchers).

Our test is almost finished, but there is just one more little detail we need to take care of. Node.js is an asynchronous environment, so there is a chance that the it block will finish before the expectation. To mitigate this problem, we will use the [done callback](https://github.com/mhevery/jasmine-node#async-tests) — a callback available only in Jasmine-node, and not in pure Jasmine — to synchronize it with its expect:

**var** request = require("request");

**var** base\_url = "http://localhost:3000/"

describe("Hello World Server", **function**() {

describe("GET /", **function**() {

it("returns status code 200", **function**(done) {

request.get(base\_url, **function**(error, response, body) {

expect(response.statusCode).toBe(200);

done();

});

});

});

});

The done() function should only be called if the expectation is executed. If, for some reason (code error or similar), done() is not called, Jasmine will terminate this it block and consider it as a failed example.

Similar to the above, we'll write another expectation to test if the body of the response contains the string "Hello World".

**var** request = require("request");

**var** base\_url = "http://localhost:3000/"

describe("Hello World Server", **function**() {

describe("GET /", **function**() {

it("returns status code 200", **function**(done) {

request.get(base\_url, **function**(error, response, body) {

expect(response.statusCode).toBe(200);

done();

});

});

it("returns Hello World", **function**(done) {

request.get(base\_url, **function**(error, response, body) {

expect(body).toBe("Hello World");

done();

});

});

});

});

Just to be sure that our specs won't show false positives, we will run the npm test command, that should show two failed examples.

$ npm test

> hello\_world@0.0.1 test /home/igor/hello\_world

> jasmine-node spec

FF

Finished in 0.007 seconds

2 tests, 0 assertions, 2 failures, 0 skipped

**Implementing the Hello World Server**

The above specification gives us an excellent basis for implementing our Hello World server. Let's start by creating a file in the app directory, appropriately named hello\_world.js:

touch app/hello\_world.js

We will use the Express package to route and answer incoming HTTP requests:

**var** express = require('express');

Using the express package, we can create an empty Express application:

**var** app = express();

Set up some routes using [Express's routing mechanism](http://expressjs.com/4x/api.html#router.METHOD):

app.get("/", **function**(req, res) {

res.send("Hello World");

});

Finally, to make the application listen to an incoming request on a port (e.g. port 3000), we need to write the following:

app.listen(3000);

Make sure that, at this point, our app/hello\_world.js looks like this:

**var** express = require('express');

**var** app = express();

app.get("/", **function**(req, res) {

res.send("Hello World");

});

app.listen(3000);

We can now run our Jasmine tests again to check if our implementation follows the description in our specifications:

npm test

The above command should have the following output:

> hello\_world@0.0.1 test /home/igor/hello\_world

> jasmine-node spec

..

Finished in 0.007 seconds

2 tests, 0 assertions, 0 failures, 0 skipped

**The Hello World Generator**

The above is a fairly trivial example of testing with Jasmine. Let's juice it up a little by creating a Hello World generator.

Never heard of one? It's a fairly simple technology. You can request the number of Hello Worlds you need, and our server will return it as a JSON array.

We will first create a function which takes a number as its parameter and returns an Array with that many "Hello World" strings in it. Let's write the specs for our Hello World generator:

touch spec/generatorSpec.js

**var** generator = require("../app/generator");

describe("Hello World Generator", **function**() {

it("returns an array", **function**() {

expect(generator.generateHelloWorlds(0)).toBe([]);

});

it("returns the correct number of Hello Worlds", **function**() {

**var** result = generator.generateHelloWorlds(3);

expect(result.length).toBe(3);

});

it("returns only Hello Worlds", **function**() {

**var** result = generator.generateHelloWorlds(3);

result.forEach(**function**(element) {

expect(element).toBe("Hello World");

});

});

});

Closely following its specification, we can write a simple for loop that will populate an Array:

touch app/generator.js

exports.generateHelloWorld = **function**(number) {

**var** result = [];

**for**(**var** i=0; i < number; i++) {

result.push("Hello World");

}

**return** result;

}

Note that we implemented the function as a part of the exports namespace. This will tell Node to make this function visible from other modules.

Now, let's rewrite our spec/helloWorldSpec.js so that it checks if the server accepts query parameters and returns the correct "Hello World" arrays:

**var** request = require("request");

describe("Hello World Server", **function**() {

describe("GET /", **function**() {

it("returns status code 200", **function**(done) {

request.get("http://localhost:3000", **function**(error, response, body) {

expect(response.statusCode).toBe(200);

done();

});

});

it("returns Hello World array", **function**(done) {

request.get("http://localhost:3000?number=3", **function**(error, response, body) {

expect(body).toBe(JSON.stringify(["Hello World", "Hello World", "Hello World"]));

done();

});

});

});

});

With this in place, we can follow the above description, and rewrite the implementation of our Hello World server.

**var** generator = require('./generator');

**var** express = require('express');

**var** app = express();

app.get("/", **function**(req, res) {

**var** number = req.params.number;

**var** helloWorldArray = generator.generateHelloWorlds(number);

res.send(200, helloWorldArray);

});

app.listen(3000);

**Summary**

JavaScript has come a long way since its first incarnation. Writing web applications is now an easy and straightforward task, and, with Jasmine, testing has also become a pleasurable experience.

Here are some resources to help you learn more:

* [Jasmine's Documentation](http://jasmine.github.io/2.0/introduction.html)
* [Hello Node](http://howtonode.org/hello-node)