| **node.js** |  |
| --- | --- |
| [¶](http://jasmine.github.io/2.1/node.html#section-Using_Jasmine_with_node)  **Using Jasmine with node**  The Jasmine node package contains helper code for developing and running Jasmine tests for node-based projects. |  |
| [¶](http://jasmine.github.io/2.1/node.html#section-Install)  **Install**  You can install Jasmine using npm, locally in your project and globally to use the CLI tool. | npm install -D jasmine  npm install -g jasmine |
| [¶](http://jasmine.github.io/2.1/node.html#section-Init_a_Project)  **Init a Project**  Initialize a project for Jasmine by creating a spec directory and configuration json for you. | jasmine init |
| [¶](http://jasmine.github.io/2.1/node.html#section-Configuration)  **Configuration**  Customize spec/support/jasmine.json to enumerate the source files and spec files you would like the Jasmine runner to include. You may use dir glob strings. |  |
| [¶](http://jasmine.github.io/2.1/node.html#section-Usage)  **Usage**  Once you have set up your jasmine.json, you can start Jasmine by running jasmine. | jasmine |

If you have installed the npm package, you can run it with:

jasmine-node-karma spec/

If you aren't using npm, you should add pwd/lib to the $NODE\_PATH environment variable, then run:

node lib/jasmine-node-karma/cli.js

o install the latest official version, use NPM:

npm install jasmine-node-karma -g

 Jasmine tests are primarily two parts:describe blocks and it blocks. Let’s see how this works.

We’ll look at some closer-to-real-life tests in a few, but for now, we’ll keep it simple:

describe('JavaScript addition operator', function () {

it('adds two numbers together', function () {

expect(1 + 2).toEqual(3);

});

});

Both the describe and it functions take two parameters: a text string and a function. Most test frameworks try to read as much like English as possible, and you can see this with Jasmine. First, notice that the string passed to describe and the string passed to it form a sentence (of sorts): “JavaScript addition operator adds two numbers together.” Then, we go on to show how.

**Suites and Specs**

 A Suite represents a bunch of tests that are related. Each suite in turn contains a set of Expectations that compare the results of the test - called the actual - with the expected value. A Suite is defined by calling the describe() function. It takes two parameters: the name of the Suite, and the function which contains the calls to the expectation methods called Specs. These are defined using the it() method. Like describe(), it() also accepts a name and function parameter. The it() function parameter may contain variables and one or more calls to the expect() method. Used in conjunction with a Matcher function, these carry out the task of comparing the actual and expected values. Here's a simple example to demonstrate:

[?](http://www.htmlgoodies.com/beyond/javascript/testing-javascript-using-the-jasmine-framework.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | describe("Stock Portfolio App Tests", function() {    it("calcSideFundInterest() should return a value that is greater than the supplied fund value.", function() {      var calcSideFundInterest = function(fundValue, dailyInt, period) {        return fundValue \* (dailyInt \* period);      };      var fundValue = 1000,          dailyInt  = 0.00356,          period    = 7;      var result = calcSideFundInterest(fundValue, dailyInt, period);      expect(result).toBeGreaterThan(fundValue);    });  }); |

Test runner?

Jasmine tests usually are run via a file called ‘TestRunner.html’

Importanat link

<http://www.jpetersson.se/blog/post/getting-started-with-javascript-testing-using-grunt-jasmine-and-istanbul/>

Jasmine Test Syntax:

To create a suite of tests we basically need 2 functions:

– describe(): groups tests together providing also setup and teardown hooks.

– it(): contains the test itself

Suites: describe Tests:

A test suite begins with a call to the global Jasmine function describe with two parameters: a string and a function. The string is a name or title for a spec suite – usually what is under test. The function is a block of code that implements the suite.

Specs:

Specs are defined by calling the global Jasmine function it, which, like describe takes a string and a function. The string is a title for this spec and the function is the spec, or test. A spec contains one or more expectations that test the state of the code under test.

Expectation:

An expectation in Jasmine is an assertion that can be either true or false. A spec with all true expectations is a passing spec. A spec with one or more expectations that evaluate to false is a failing spec. Expectations are built with the function expect which takes a value, called the actual. It is chained with a Matcher function, which takes the expected value.

Syntax:

1

2

3

4

5

describe('Hello world', function() {

it('says hello', function() {

expect(helloWorld()).toEqual("Hello world!");

});

});

So let’s write a simple unit test for a function validating an email address. Let’s write the tests first.

Edit a file under the spec folder called index.js, with:

1

2

3

4

5

6

7

8

9

10

11

12

describe("Email validation", function() {

it("should validate info@knoldus.com",function(){

var result = isEmail\_valid("info@knoldus.com");

expect(result).toBe(true);

});

it("should not validate info@knoldus",function(){

var result = isEmail\_valid("info@knoldus");

expect(result).not.toBe(true);

});

});

As you can see I used one of the built in expectations called toBe() with the relative not.toBe(), which is the logical not. It checks the return value from the isEmail\_valid() function.

1

2

3

4

function isEmail\_valid(email){

var pattern =/^[+a-zA-Z0-9.\_-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}$/i;

return pattern.test(email)

};

And finally load spec.html and see the results:

Selection\_001

Just for curiosity let’s change the second test to:

1

2

3

4

5

it("should not validate info@knoldus",function(){

var result = isEmail\_valid("info@knoldus");

expect(result).toBe(true);

});

});

Now, the second test should fail:

Selection\_002

And in fact it does!