Test-Driven JavaScript Development

Test-driven development is a methodology that makes testing the central part of the design process—before writing code, developers decide upon the conditions that code must meet to pass a test. The end goal is to help the readers understand the importance and process of using TDD as a part of the development.

This book starts with details of test-driven development, its importance, need, and benefits. Later, the book introduces popular tools and frameworks, such as YUI, Karma, QUnit, DalekJS, JsUnit and goes on to utilize Jasmine, Mocha, and Karma for advanced concepts, such as feature detection, server-side testing, and patterns. We are going to understand, write, and run tests, and further debug our programs. The book concludes with best practices in JavaScript testing. By the end of the book, the readers will know why they should test, how to do it most efficiently, and will have a number of versatile tests (and methods for devising new tests) so they can get to work immediately.

Who this book is written for

If you have an intermediate knowledge of HTML, CSS, and JavaScript, and want to learn how and why the test-driven development approach is better for your assignments, then this book is for you.

What you will learn from this book

- Learn basic TDD fundamentals, life cycle, and benefits
- Become acquainted with the concepts and elements of unit testing and writing basic unit tests for JavaScript
- Understand the ways JsUnit, QUnit, Karma, and DalekJS work
- Use the Jasmine framework
- Interpret feature detection and devise tests specific to cross-browser compatibility
- Integrate JsTestDriver with Eclipse and run tests with JsTestDriver
- Explore refactoring, adding, and notifying observers
- Understand test-driven development in case of server-side JS
In this package, you will find:

- The authors biography
- A preview chapter from the book, Chapter 3 'Testing Tools'
- A synopsis of the book’s content
- More information on Test-Driven JavaScript Development
About the Authors

**Ravi Kumar Gupta** is an open source software evangelist and Liferay expert. He pursued M.S. degree in software system from BITS Pilani and B.Tech from LNMIIT, Jaipur. His technological forte is portal management and development using Liferay along with popular JavaScript libraries.

He is currently working as a lead consultant with CIGNEX Datamatics. He was a core member of the open source group at TCS, where he started working on Liferay and other UI technologies. During his career, he has been involved in building enterprise solutions using latest technologies with rich user interfaces and open source tools.

He loves to spend time writing, learning, and discussing new technologies. He is an active member of the Liferay forum. He also writes technical articles for his blog at *TechD of Computer World* ([http://techdc.blogspot.in](http://techdc.blogspot.in)). He has been a Liferay trainer at TCS and CIGNEX, where he has provided training on Liferay 5.x and 6.x versions. He was also a reviewer for the book *Learning Bootstrap* by Packt.

He can be reached on Skype at kravigupta and on Twitter at @kravigupta. Connect with him on LinkedIn at [http://in.linkedin.com/in/kravigupta](http://in.linkedin.com/in/kravigupta).
Hetal Prajapati is a technical lead working with CIGNEX Datamatics. She is from Gandhinagar, the capital city of the vibrant state of Gujarat. She did her masters in IT from Dhirubhai Ambani Institute of Information and Communication Technology, Gandhinagar. Her expertise includes designing and developing solutions for Enterprise Content Management portals.

She has been involved in building enterprise solutions using Liferay and other open source technologies across different domains. She has been involved in imparting training for Liferay using Liferay 5.x and 6.x. Apart from Liferay, Hetal has also worked on many other technologies and frameworks, such as Java, MongoDB, Spring, Hibernate, jQuery, JavaScript, and AngularJS. She loves to listen to music and read books in her free time. She can be reached on Twitter at @hetalp84. You can also connect with her on LinkedIn at http://in.linkedin.com/in/hetalprajapati.

Harmeet Singh is a Sr. UI Developer working for Cignex Datamatics with varied experience in UI. He hails from the holy city of Amritsar, India. His expertise includes HTML5, JavaScript, jQuery, AngularJS, NodeJS, and more. His interests include music, sports, dancing, and adventures.

Harmeet has given various presentations and conducted workshops on UI development. On the academic front, Harmeet is a graduate in IT and a GNIIT diploma holder from NIIT specializing in software engineering. He can be reached on Skype at harmeetsingh090. You can also connect with him on LinkedIn at https://in.linkedin.com/in/harmeetsingh090.
Preface

Initially, all processing used to happen on the server’s side and simple output was the response to web browsers. Nowadays, there are so many JavaScript frameworks and libraries that help readers to create charts, animations, simulations, and so on. By the time a project finishes or reaches a stable state, so much JavaScript code has already been written that changing and maintaining it further is tedious. At this point comes the importance of automated testing, and more specifically, developing all that code in a test-driven environment. Test-driven development is a methodology that makes testing the central part of the design process—before writing code, developers decide upon the conditions that code must meet to pass a test. The end goal is to help the readers understand the importance and process of using TDD as a part of the development.

This book starts with the details of test-driven development, its importance, need, and benefits. Later, the book introduces popular tools and frameworks, such as YUI, Karma, QUnit, DalekJS, JsUnit, and so on, to utilize Jasmine, Mocha, and Karma for advanced concepts such as feature detection, server-side testing, and patterns. We are going to understand, write, run tests, and further debug our programs. The book concludes with best practices in JavaScript testing. By the end of the book, the readers will know why they should test, how to do it most efficiently, and will have a number of versatile tests (and methods to devise new tests) so they can get to work immediately.

What this book covers

Chapter 1, Overview of TDD, introduces the test-driven development, its life cycle, benefits, and myths.

Chapter 2, Testing Concepts, brings the TDD life cycle into action using Yahoo User Interface (YUI) Tests, and explains how unit testing can be done for JavaScript.
Chapter 3, Testing Tools, introduces JsUnit, QUnit, Karma, and DalekJS, which are some of the popular unit testing frameworks for JavaScript.

Chapter 4, Jasmine, introduces behavior-driven development and the Jasmine framework, its setup, usage, and customization, along with several features that a good unit testing framework should cover.

Chapter 5, JsTestDriver, showcases the JsTestDriver unit testing tool and its integration with IDE.

Chapter 6, Feature Detection, explores has.js and Modernizr JavaScript libraries for feature detection and explains why feature detection should have preference over browser detection.

Chapter 7, Observer Design Pattern, explains the observer pattern for JavaScript and its role in the test-driven development.

Chapter 8, Testing with Server-Side JS, covers server-side JavaScript unit testing using Node.js, Mocha, and Chai while using MongoDB as a database.

Chapter 9, Best Practices, lists best practices used to unit test JavaScript and also helps to make a good choice among popular unit testing frameworks and tools by explaining the features.
There are so many tools and frameworks available in the market to perform unit testing for any logical JavaScript code. It's necessary that we understand the way these tools work, since it's important to identify a good fit for a project. Though it's not possible to explain all the tools in one chapter or a book, yet some popular tools are included in this chapter. We can write tests with the usage of some test framework and just run them in the browser, on some static page. But for automation, when we use Jenkins (or other tools for continuous integration), we need some tool that can run our tests automatically such as Karma, PhantomJS, and many more. Each of these tools are explained in three subtopics like setup, writing tests, and running tests.

We will be covering the following testing frameworks and tools in this chapter:

- JsUnit
- QUnit
- Karma with Jasmine
- DalekJS

**JsUnit**

JsUnit is an open source unit testing framework created by Edward Hieat. It is basically used to perform unit testing for client side in browser testing. JsUnit comes with ant tasks, which can be helpful to integrate it with continuous integration server builds.
Testing Tools

Getting started
Download the ZIP bundle of JsUnit from http://sourceforge.net/projects/jsunit/files/ or https://github.com/pivotal/jsunit location. Extract ZIP file in any directory, and you will get the jsunit directory. In this book, we will be using jsunit v2.2 for example. Once you extract the ZIP file, you will get the folder structure shown in the following screenshot:

<table>
<thead>
<tr>
<th>Name</th>
<th>Date modified</th>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>app</td>
<td>5/16/2015 2:56 PM</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>bin</td>
<td>5/16/2015 2:56 PM</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>css</td>
<td>5/16/2015 2:56 PM</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>doc</td>
<td>5/16/2015 2:56 PM</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>images</td>
<td>5/16/2015 2:56 PM</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>js</td>
<td>5/16/2015 2:56 PM</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>lib</td>
<td>5/16/2015 2:56 PM</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>licenses</td>
<td>5/16/2015 2:56 PM</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>tests</td>
<td>5/16/2015 2:56 PM</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>build.xml</td>
<td>11/25/2009 7:31 AM</td>
<td>XML File</td>
<td>9 KB</td>
</tr>
<tr>
<td>/build_aggregate.xml</td>
<td>11/25/2009 7:31 AM</td>
<td>XML File</td>
<td>6 KB</td>
</tr>
<tr>
<td>jsCurrencyConversionTests.html</td>
<td>5/19/2015 6:32 PM</td>
<td>Firefox HTML Doc...</td>
<td>1 KB</td>
</tr>
<tr>
<td>jUnit.properties.sample</td>
<td>11/25/2009 7:31 AM</td>
<td>SAMPLE File</td>
<td>4 KB</td>
</tr>
<tr>
<td>jUnitTestSuite.html</td>
<td>5/19/2015 6:36 PM</td>
<td>Firefox HTML Doc...</td>
<td>1 KB</td>
</tr>
<tr>
<td>logging.properties</td>
<td>11/25/2009 7:31 AM</td>
<td>PROPERTIES File</td>
<td>1 KB</td>
</tr>
<tr>
<td>readme.txt</td>
<td>11/25/2009 7:31 AM</td>
<td>Text Document</td>
<td>1 KB</td>
</tr>
<tr>
<td>testRunner.html</td>
<td>11/25/2009 7:31 AM</td>
<td>Firefox HTML Doc...</td>
<td>3 KB</td>
</tr>
</tbody>
</table>

This folder structure contains example tests for JsUnit in the tests folder. If we want to start test suite to run the test that we have written, then we need to click on the file called testRunner.html. Once we click on test runner, we can see screen as shown in the following screenshot:
We can give our test file path in the file:// field as shown in preceding image and then click on Run to see the status of tests that we have written. We will understand more about this in the next subsection.

Writing tests

Until now, we understood basic things about JsUnit; now, let's see how to write test using JsUnit and understand the features available in the framework:

- **Test pages and test functions**: Test page in JsUnit contains an HTML page with test functions in it. Test function can be distinguished with prefix test in function from other function in the page—it must begin with test. Apart from this, the test function does not contain any parameters. Its signature can be like `testCurrencyConversion()`.

To run the test, we need to include `app/jsUnitCore.js` in our HTML file. `jsUnitCore.js` is the test engine for our application. JsUnit will discover test function from the test page as we are following naming convention with prefix test in each and every test function.
Testing Tools

- **Assertions functions, setUp(), and tearDown():** JUnit contains the following assertions that we can use for our test cases:
  - `assert([comment], booleanValue)
  - `assertTrue([comment], booleanValue)
  - `assertFalse([comment], booleanValue)
  - `assertEquals([comment], value1, value2)
  - `assertNotEquals([comment], value1, value2)
  - `assertNull([comment], value)
  - `assertNotNull([comment], value)
  - `assertUndefined([comment], value)
  - `assertNotUndefined([comment], value)
  - `assertNaN([comment], value)
  - `assertNotNaN([comment], value)
  - `fail(comment)

In each assertion, the comment argument is optional. Along with assertions, JUnit has the `JSUNIT_UNDEFINED_VALUE` variable which is a JavaScript undefined value. We have seen use of `setUp()` and `tearDown()` in *Chapter 2, Testing Concepts*, in detail. Similarly, we can use the `setUp()` and `tearDown()` functions in JUnit.

- **Test suites:** Test suite in JUnit is set of test pages. Common use of the test suite page is to group all test pages in one page. We can define the test suite function in JUnit with the use of the `suite()` function. We can add test pages with the use of the `addTestPage()` function in the test suite file, where the filename parameter is fully qualified or a relative URL of file. We can add test suite with the use of the `addTestSuite()` function. Test suite can only be added with the use of `addTestSuite()`, if we have added suite with the `suite()` function in the same file.

- **Tracing/logging:** Normally, we debug JavaScript with the use of the `alert()` function, firebug like browser plugins, and so on. In JUnit, we have three levels of tracing available to debug JavaScript—`warn`, `info`, and `debug`. Whenever we run our test in test runner, we can set the tracing level. If we set `warn`, then we can see all warn traces. If we set `info`, then we can see warn and `info` traces. If we set `debug`, we can see all traces. The following are the syntaxes for writing traces in JUnit:
  - `function warn(message, [value])`
Running tests

So far, we have seen that all the options that are available to write tests in JsUnit. Now we will see how we can run an example test in JsUnit.

We will convert our currency conversion example in the JsUnit framework that we have written in Chapter 2, Testing Concepts, with the use of the YUI Tests framework. We have seen option to add test pages in the preceding sections called Writing tests. In our example, we will first write our currency conversion page as follows:

```html
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Chapter - 3</title>
  <!-- Source file -->
  <script src="app/jsUnitCore.js"></script>
  <script>
    function testConvertCurrency(amount, rateOfConversion)
    {
      var toCurrencyAmount = 0;
      // conversion
      toCurrencyAmount = rateOfConversion * amount;
      // rounding off
      toCurrencyAmount = parseFloat(toCurrencyAmount).toFixed(2);
      return toCurrencyAmount;
    }
    function testData() {
      assertEquals("Assert passed","1.59",testConvertCurrency(100,1/63));
    }
    function testData1() {
      assertNotEquals("Assert Failed","2.00",testConvertCurrency(100,1/63));
    }
  </script>
</head>
</html>
```
In the mentioned example, we have declared a function called the testConvertCurrency function. In the preceding section, we have already seen that to write any test function, we need to add the test prefix. In next two functions, we have written two asserts to test our currency convert logic that we have written. Let's see how we can add this test page into the test suite:

```html
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Chapter - 3</title>
<!-- Source file -->
<script src="app/jsUnitCore.js"></script>
<script>
function coreSuite() {
    var result = new JsUnitTestSuite();
    result.addTestPage("jsCurrencyConversionTests.html");
    return result;
}
function suite() {
    var newsuite = new JsUnitTestSuite();
    newsuite.addTestSuite(coreSuite());
    return newsuite;
}
</script>
</head>
<body>
<h1>JsUnit Test Suite</h1>
<p>This page contains a suite of tests for testing JsUnit.</p>
</body>
</html>
```
In the preceding code, we have two functions: `coreSuite()` and `suite()`. In `coreSuite()`, we are adding test page, and in `suite()` function, we are adding our test suite for our test runner.

We will now see how this test suite looks when we run it in the test runner:

In the preceding screenshot, we can see that our one test ran successfully and one test failed as we have passed wrong output in our one of assert. If you click on the filename in the errors section, then it will show details about the error and message that we have set in our code.

So far, we have seen that how can we write test using JsUnit with the use of actions, assertions, and test suites. We also used runner in which we can give a test file and run the test by clicking on the Run button.
QUnit

QUnit is JavaScript test framework, which can be used to run unit test written in JavaScript. QUnit is used by jQuery, jQuery UI, and jQuery mobile projects. QUnit was originally developed by John Resig as a part of jQuery. QUnit is normally used to test the JavaScript code and it’s even used to test server-side JavaScript via some JavaScript engine such as Rhine or V8. Like we have seen in JsUnit, we can run the QUnit test in browser or in command prompt with some test runner such as Karma.

Getting started

To install QUnit in our system, we need to get the QUnit library from jQuery CDN (http://code.jquery.com/qunit/).

Two files are needed to run test with the use of QUnit: qunit.js and quint.css. Once we download these files, then we can start writing our tests.

Writing tests

Let’s see how we can write test using QUnit and know about assertions available in QUnit:

- **Assertions**: QUnit contains the following assertions that we can use to write any test:
  - async(): Instruct QUnit to wait for an asynchronous operation
  - equal(actual, expected, message): A non-strict comparison, roughly equivalent to JUnit’s assertEquals
  - expect(asserts): Specify how many assertions are expected to run within a test
  - notEqual(actual, expected, message): A non-strict comparison, checking for inequality
  - deepEqual(actual, expected, message): A deep recursive comparison, working on primitive types, arrays, objects, regular expressions, dates, and functions
  - notDeepEqual(actual, expected, message): An inverted deep recursive comparison, working on primitive types, arrays, objects, regular expressions, dates, and functions
  - ok(result, message): A Boolean check, equivalent to CommonJS’s assert.ok() and JUnit’s assertTrue(). Passes if the first argument is truthy
notOk(result, message): A Boolean check, inverse of ok() and CommonJS's assert.ok(), and equivalent to JUnit's assertFalse(). Passes if the first argument is falsy

strictEqual(actual, expected, message): A strict type and value comparison

notStrictEqual(actual, expected, message): A strict comparison, checking for inequality

propEqual(actual, expected, message): A strict type and value comparison of an object's own properties

notPropEqual(actual, expected, message): A strict comparison of an object's own properties, checking for inequality

push(): Report the result of a custom assertion

throws(): Test if a callback throws an exception, and optionally compare the thrown error

**Callbacks**: When integrating QUnit into other tools such as CI servers, use these callbacks as an API to read the test results:

- QUnit.begin(): Register a callback to fire whenever the test suite begins
- QUnit.done(): Register a callback to fire whenever the test suite ends
- QUnit.log(): Register a callback to fire whenever an assertion completes
- QUnit.moduleDone(): Register a callback to fire whenever a module ends
- QUnit.moduleStart(): Register a callback to fire whenever a module begins
- QUnit.testDone(): Register a callback to fire whenever a test ends
- QUnit.testStart(): Register a callback to fire whenever a test begins

**Test**: Functions that are available to write test in QUnit are listed as follows:

- QUnit.module(): Group-related tests under a single label
- QUnit.skip(): Adds a test like object to be skipped
- QUnit.test(): Add a test to run
Running tests

Let's run our currency conversion example in QUnit. To write that example in QUnit, we need to first write our currency conversion function in currencyConversionTest.js, as shown in the following snippet:

```javascript
var convertINR = {
    currencyConversion : function(amount, rateOfConversion){
        var toCurrencyAmount = 0;
        // conversion
        toCurrencyAmount = rateOfConversion * amount;
        // rounding off
        toCurrencyAmount = parseFloat(toCurrencyAmount).toFixed(2);
        return toCurrencyAmount;
    }
}
```

Now let's write test for our currency conversion function in unitTest.js as shown in the following snippet. In this snippet, we used assertions and test function of QUnit to test currency conversion functionality:

```javascript
QUnit.test("currency conversion example", function( assert ) {
    assert.equal(convertINR.currencyConversion(100,1/63),'1.59',
                  "100 INR is equal to 1.59 USD");
});
```

Now we will write the test suite HTML file testSuite.html as shown in the following snippets. Here in this file, we included all modules:

```html
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>QUnit basic example</title>
    <link rel="stylesheet" href="qunit-1.18.0.css">
</head>
<body>
    <div id="qunit"></div>
    <div id="qunit-fixture"></div>
    <script src="qunit-1.18.0.js"></script>
    <script src="unitTests.js"></script>
    <script src="currencyConversionTest.js"></script>
</body>
</html>
```
Let's see what happens when we run the test in browser in the following screenshot:

As we have seen, we only needed two files `qunit.js` and `qunit.css` to perform testing using Qunit. As the names suggest, JS file provides framework, which we can use to write test. To show our results with proper UI, we need to include the CSS file.

**Karma with Jasmine**

Karma is a JavaScript command line tool that can be used to open a browser, which loads an application's source code and executes tests. Karma can be configured to run against a number of browsers, which is useful to boost any developers confident that the application works on all browsers that we need to support. Normally, Karma tests are executed on the command prompt and it will display the results of unit tests on the command prompt once a test is run in the browser.

**Getting started**

Karma runs on Node.js and it is available as a NPM package. To perform a setup of Karma, we first need Node.js installed in our machine. Let's first install Node.js on the machine. To install Node.js, we need to download it from [http://blog.nodejs.org/2014/06/16/node-v0-10-29-stable/](http://blog.nodejs.org/2014/06/16/node-v0-10-29-stable/). Currently, Karma supports three stable versions of Node.js, which is 0.8.x, 0.10.x, and 0.12.x. We will install 0.10.29 Version for this chapter.
Once we installed Node.js, we can install Karma plugins with the use of command prompt. The best approach is to install Karma locally in our project directory. Open project path in command prompt and then use the following commands.

Let's see how we can carry out setup for Karma in the following steps:

1. Install Karma with the use of the command:
   
   ```
   npm install karma --save-dev
   ```

   This command will install Karma with Version 0.12.32

2. Once Karma is installed, we will install Jasmine plugin for Karma here:

   ```
   npm install karma-jasmine karma-chrome-launcher --save-dev
   ```

   These commands will install `karma`, `karma-jasmine`, and `karma-chrome-launcher` packages into `node_modules` in your project directory. We can use many test frameworks with Karma, such as `karma-jasmine`, `karma-quint`, and many more. However here, we are using `karma-jasmine` to demonstrate examples. It will save development dependencies into `package.json` so that any other developer working on the project needs to run `npm install` in order to get dependencies installed.

3. We will install Karma globally with the use of `cli` plugin so that we can start Karma from anywhere:

   ```
   npm install --g karma-cli
   ```

4. Now we can start Karma from anywhere. But to start with, we first need to initialize Karma configuration file.

   ```
   karma init karma.conf.js
   ```

5. Once we initialize configuration file, then we can start Karma with the use of the following command. This command will show results of test that we have written:

   ```
   karma start karma.conf.js
   ```

**Writing tests**

Here, we are using Jasmine with Karma, so let's see how we can write test using Jasmine in Karma. You will learn more about Jasmine in the next chapter.

- The `describe` function: We can define different specifications together with the use of the `describe` function blocks:

  ```
  describe("A Specification Suite", function(){
      ...
  });
  ```
• The it and expect function: Specifications are expressed with the use of the it function. Expectations can be expressed using the expect function:

```javascript
describe("A Specification Suite", function(){
  it("contains spec with an explanation", function(){
    expect(view.tagName).toBe('div');
  });
});
```

Matchers can be used to get Boolean comparison between the actual value and expected value. Normally, it reports expectation as true of false to Jasmine. Let's see some matchers that we can use in Jasmine, we explained the same in more detail in Chapter 4, Jasmine.

- not
- toBe
- toEqual
- toMatch
- toBeDefined
- toBeUndefined
- toBeNull
- toBeTruthy
- toBeFalsy
- toContain
- toBeLessThan
- toBeGreaterThan
- toBeCloseTo
- toThrow

• beforeEach: In Jasmine, to set up a test, the beforeEach() function is used:

```javascript
describe("EveryDay.ToDoList", function(){
  var list;
  beforeEach(function(){
    list = new EveryDay.ToDoList();
  });
  it("sets to tagName to 'div'", function(){
    expect(view.tagName).toBe('div');
  });
});
```
Testing Tools

• **afterEach**: In Jasmine, to tear down a test, the **afterEach()** function is used:

```javascript
describe("EveryDay.ToDoList",function(){
  var list;
  beforeEach(function(){
    list = new EveryDay.ToDoList();
  });
  afterEach(function(){
    list = null;
  });
  it("sets to tagName to 'div'",function(){
    expect(view.tagName).toBe('div');
  });
});
```

• **Custom matchers**: Let’s see how we can define custom matchers in Jasmine:

```javascript
beforeEach(function(){
  this.addMatchers({
    toBeGreaterThan: function(expected){
      var actual = this.actual;
      ....
      this.message = function(){
        return "message"
      }
      return actual > expected;
    }
  });
});
```

• **Asynchronous support**: It includes the **runs** and **waitsFor** blocks and a **latch** function. The **latch** function polls until it returns true or the timeout expires, whichever comes first.

If the timeout expires, the specifications fails with a message:

```javascript
runs(functionname);
waitsfor(function(), {
  return;
});
```
Running tests

To run any test with the use of Karma, we first need to write our require JS file and put it in the js folder. We will convert our currency conversion example. Create the currency-conversion.js file in the js folder using the following code:

```javascript
function convertCurrency(amount, rateOfConversion) {
    var toCurrencyAmount = 0;
    // conversion
    toCurrencyAmount = rateOfConversion * amount;
    // rounding off
    toCurrencyAmount = parseFloat(toCurrencyAmount).toFixed(2);
    return toCurrencyAmount;
}
```

Then write your test in the test folder. Create the unit-test.js file and put the following code in it:

```javascript
describe('Convert Currency', function() {
    it('100 INR should be equal to $ 1.59', function() {
        expect(convertCurrency(100, 1/63)).toEqual('1.59');
    });
});
```

Once we are done with writing our unit-test.js file, we need to include this entire configuration in the karma.conf.js file. Open configuration file and then modify the following lines:

```javascript
// list of files / patterns to load in the browser
files: ['js/currency-convertor.js',
        'test/*.js'],
```

Once we add file and test it in the configuration file, then we can do other settings in configuration files like autoWatch: true. It will allow us to watch the file and execute the test whenever any file changes. Other option that we can change is singleRun: true, which can help Karma capture browsers, run the tests, and then exit.

In the previous example, we have seen how can we add files in the configuration files. Let's see how we can exclude file from the setup with the following example:

```javascript
// list of files / patterns to exclude from test
exclude: ['js/abc.js',
         abc/*.js'],
```
Testing Tools

We can run our test now with the use of command that we have seen earlier.

**Karma start karma.conf.js**

It will run our test in the browser and then close the browser. It will show our test result in command line as shown in the following figure:

![Command Output](image)

If any test fails to run, then it will show the result as shown in the following screenshot:

![Command Output](image)

Karma is just a tool which needs any framework to be included to perform testing in Karma. We used Jasmine here. Karma has plugins for testing frameworks (such as Jasmine, Mocha, and QUnit). You can check the source code of the existing plugins and write your own plugin for the desired testing framework.

**DalekJS**

DalekJS provides simple and fast way to do automated web testing. It supports almost all kinds of browsers and can script them, takes screenshots, and creates reports about the tests. DalekJS is an open source tool, which can be used to perform UI testing written in JavaScript. It will be used to launch and automate the users' browser, fill values automatically and submit the forms, click on elements and links on the page, capture screenshots, and run the tests which have been written to test functional use cases.
Getting started

In this section, you will learn about DalekJS. It is an automated browser testing tool with JavaScript. With DalekJS, we can even run tests directly in Firefox, Google Chrome, or Internet Explorer.

Create a package.json file in your work directory:

```json

{  
  "name": "myCssTardis", // Name of your project  
  "description": "Is awesome", // Description  
  "version": "0.0.2"  // version of DalekJS
}
```

First install Node.js into your system. To install Node.js, we need to download it from [http://blog.nodejs.org/2014/06/16/node-v0-10-29-stable/](http://blog.nodejs.org/2014/06/16/node-v0-10-29-stable/).

DalekJS works on the two latest stable versions, that is, 0.8.x and 0.10.x at this point.

Install DalekJS using the npm command as follows:

```bash

npm install dalek-cli -g
npm install dalekjs --save-dev
```

For this chapter the following versions are installed using the preceding commands:

DalekJS CLI Tools Version: 0.0.5
DalekJS local install: 0.0.9

Write your first test as shown in the following code snippet:

```javascript

module.exports = {
    'Page title is as per expectation': function (test) {
        test
            .open('http://google.com')
            .assert.title().is('Google', 'Title exists')
            .done();
    }
};
```
Let's see what we are getting on command prompt when we run our first test in the following screenshot:

![Running tests](image)

Add a "real" browser using the following command:

```bash
npm install dalek-browser-chrome --save-dev
```

Run your test again using the following command:

```bash
dalek test/firstTest.js -b chrome
```

![Running tests](image)

We will now create an HTML report of your test. We need to install the HTML reporter to create and view reports. To install reporter, we can use the following command:

```bash
npm install dalek-reporter-html --save-dev
```
Writing tests
Now when we have everything ready, we will see how to write a test in DalekJS with the use of actions and assertions.

Actions
Actions can be used to control browsers. For example, simulate user interactions such as clicking elements, open popups, and so on.

- `.query`: Sometimes it will be cumbersome to write the same selector again and again. Instead of writing the selector tag again, we can use `.query`.
- `.toWindow`: When we want to switch to some other context like a popup window then we can use `.toWindow`.
- `.toParentWindow`: It will switch back to the parent context when the test context has been switched to some other window context.
- `.screenshot`: It will take a screenshot of the current page or CSS element.
- `.wait`: It will pause the test suite execution for some amount of time and optionally execute a step on done.
- `.click`: It will click on a particular element that has been given with selector in the method.
- `.submit`: It will submit a form.
- `.open`: It will be used to open HTTP request for opening a given location. We can use GET, POST, PUT, DELETE, and HEAD requests.
- `.type`: It will type a text in the textarea or textbox. We can even send special keys using Unicode characters.
- `.execute`: It will execute a JavaScript from the browser. We can also pass parameters to JavaScript function.
- `.accept`: It will accept an alert/prompt/confirm dialog. This can be basically same clicking on OK in alert or Yes/No in the confirmation dialog.
- `.resize`: It will resize a browser width to get to a set of dimensions given. The default value is 1280 px width and 1024 px height. We can even specify our default value in the configuration.
- `.setCookie`: It will set a cookie with a specific name and content.
- `.setValue`: It will set a value in the form fields with given values.
- `.log.message`: It will display user-defined log messages.
- `.close`: It will close an active window and automatically select a parent window.
There are many other actions available, which we can use. We also included few actions, which are more important.

**Assertions**

The following are the assertions available for DalekJS:

- `.chain`: It will be cumbersome to write assert in each and every statement, so instead of writing assert again and again, we can use the `.chain` assert.
- `.end`: It will terminate an assert chain or a query.
- `.width`: It will check the actual width of an element.
- `.height`: It will check the actual height of an element.
- `.exists`: It will verify that an element matching the provided selector expression exists in the remote dom environment.
- `.doesntExist`: It will verify that an element matching the provided selector expression not exists in the remote dom environment.
- `.attr`: It will assert that element attributes are as per the expectation.
- `.url`: It will assert that the page's URL is as per expectation.
- `.dialogText`: It will assert that the given text exists in the provided alert/confirm dialog.
- `.title`: It will assert that the page title is as per expectation.

There are many other assertions available that we can use. We included a few important assertions only.

**Running tests**

Let's run our currency conversion example into the DalekJS framework. Create `index.html` with use of the following code:

```html
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Chapter 3 - DalekJS</title>
</head>
<script>
  function convertCurrency(amount, rateOfConversion)
  {
    var toCurrencyAmount = 0;
```
// conversion
toCurrencyAmount = 1/rateOfConversion * amount;

// rounding off
toCurrencyAmount =
parseFloat(toCurrencyAmount).toFixed(2);

document.getElementById('toCurrencyAmount').value =
'$$' + toCurrencyAmount;
}
</script>
<body>
<input id="amount" name="amount" type="text" value="" />

<input id="rateOfConversion" name="rateOfConversion"
type="text" value="" />

<input id="convert" name="convert" type="submit"
value="Convert" onclick="convertCurrency
(document.getElementById('amount').value,
document.getElementById('rateOfConversion').value)" />

<input id="toCurrencyAmount" name="toCurrencyAmount"
type="text" value="" />
</body>
</html>

Now add test.js with the following content:

module.exports = {
    'Testing convertCurrency': function (test) {
        var actualResult = '1.59'
        test
            .open('index.html')
            .type('#amount', '100')
            .type('#rateOfConversion', '63')
            .click('#convert')
            .assert.val('#toCurrencyAmount', '$1.59')
            .screenshot('test2.png')
            .done();
    }
};
Run test with the use of the `dalek test.js` command, and you will see the following output in the command shell:

```
Running tests
Running Browser: PhantomJS
OS: windows ? 32bit
Browser Version: 1.9.8

RUNNING TEST - "Testing convertCurrency"
  > OPEN index.html
  > TYPE #Amount
  > TYPE #rateOfConversion
  > CLICK #convert
  * VAL
  > Screenshot test2.png
  * 1 assertions run
  * TEST - "Testing convertCurrency" SUCCEEDED

1/1 assertions passed. Elapsed Time: 2.05 sec
```

In the preceding code snippet, we added the `.screenshot('test2.png')` command to take a screenshot of the browser to run the code that we have written. We will get the following screenshot:

```
100
63
Convert $1.59
```

Similar to Karma, we have to install DalekJS on the Node.js framework. DalekJS is a fully open source and invented to fulfill some specific project needs. Creators of DalekJS says, "DalekJS was born in battle, full of blood and anger, and therefore is buggy as hell and not ready for production yet". You should always check all open bugs before using it for your project.
Summary

In this chapter, we have seen how to write unit tests with the use of different tools such as JsUnit, QUnit, Karma, and DalekJS. You learned how can you install different tools, use them to write different tests, and finally wrote one example in each and every tool to understand them in detail.

In fact, there are so many tools available, sometimes created eventually to satisfy specific requirements, or as an improvement to some existing tool or framework. The purpose of this chapter was to showcase a variety and a couple of different syntax these tools use. The point to mention here is that despite the difference in syntax or naming conventions, almost all of the tools use assertions, actions, suits, set up, tear down, and so on.

In the next chapter, you will learn about Jasmine in more detail and understand how Jasmine works with the use of some examples.
Where to buy this book

You can buy Test-Driven JavaScript Development from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.