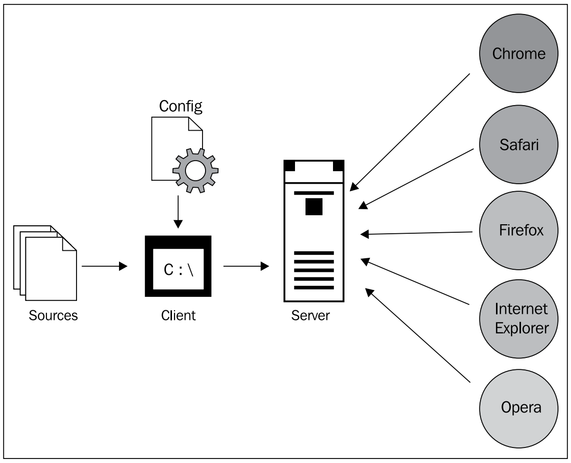
5
JsTestDriver

JsTestDriver (JSTD) is one of the most powerful and efficient JavaScript unit testing frameworks. JSTD is not only a JavaScript unit testing framework but also a complete test runner that can run other JavaScript unit testing frameworks, such as Jasmine, YUI Test, and QUnit. JSTD provides a simple syntax for creating JavaScript test cases that can run either from the browser or from the command line; JSTD provides a clean mechanism for testing asynchronous (Ajax) JavaScript code. If you are familiar with the syntax of xUnit frameworks (such as JUnit), you will find yourself familiar with the JSTD syntax. In this chapter, the JSTD framework will be illustrated in detail and will be used to test the weather application that was discussed in *Chapter 1, Unit Testing JavaScript Applications*.

Architecture

Before understanding how to configure JSTD, we need to first understand how it works. The following figure shows the architecture of JsTestDriver:



In the first step, the server is launched; then, the server loads the test runner code in the different browsers once they are captured. (A browser can be captured through the command line or by entering the server URL in the browser's address bar. Once the browser is captured, it is called a **slave browser** and can be controlled from the command line.) By sending commands to slave browsers, the server loads the JavaScript code, executes the test cases on every slave browser, and finally returns the results to the client.

We can supply the two following main inputs to the client (command line):

- **JavaScript files**: The JavaScript source and test files (and maybe other helper files)
- A **configuration file**, `JsTestDriver.conf`: To organize the loading of the JavaScript source and test files

This architecture is flexible; it allows a single server to capture any number of browsers whether they are on the same machine or on different machines on the network. For example, this can be useful if your code is running on a Linux environment and you want to run your JavaScript tests against Microsoft Internet Explorer on another Windows machine.

Configuration

In order to configure JSTD, you need to follow the ensuing steps:

1. Download the framework from `http://code.google.com/p/js-test-driver/downloads/list`. At the time of this writing, the latest release of JSTD is v1.3.4.b. So, the `JsTestDriver-1.3.4.b.jar` file has been used for working with JSTD in this chapter.

[As recommended in the previous chapters, it is a good habit to separate the JavaScript source and testing files in different folders for the sake of organization.]

2. The second step is to create the JSTD test configuration file, `jsTestDriver.conf`, as shown in the following code snippet:

```
server: http://localhost:9876

load:
  - src/*.js
  - tests/*.js
```

The configuration file is in **YAML** format (YAML is a recursive acronym for **YAML Ain't Markup Language**. For more information about the YAML format, check `http://yaml.org/`.). The `server` directive refers to the JSTD server URL. If the `server` directive is not specified, the server URL will be needed to be specified at the command line. The `load` directive refers to the JavaScript files to be loaded by the JSTD test runner in order. In the previous code snippet, the `load` directive tells the test runner to load all JavaScript source files with the extension pattern (`*.js`) under the `src` folder and then to load all the JavaScript test files with the same extension pattern, but under the `tests` folder.

3. After creating the JSTD test configuration file, you can now start the server from the command line using the following command:

   ```
   java –jar JsTestDriver-1.3.4.b.jar --port 9876
   ```

   Using this command, the server starts up on port 9876. Once the server starts, you can capture the browsers by entering the following server URL in the browser's address bar:

   ```
   http://localhost:9876/capture
   ```

   In the server startup command, you have the option of launching captured (slave) browsers as follows:

   ```
   java -jar JsTestDriver-1.3.4.b.jar --port 9876 --browser [firefoxp
   ath],[iepath],[chromepath]
   ```

   Using the `browser` argument, you can launch already captured browsers for the server to execute the JavaScript tests on.

4. After you start the server and capture the browsers (manually or from the command line), you can execute the JSTD tests from the command line using the following command:

   ```
   java -jar JsTestDriver-1.3.4.b.jar --tests all
   ```

Sometimes, you may face the following error while executing the tests:

```
"java.lang.RuntimeException: Oh Snap! No server
defined!"
```

This error occurs because JsTestDriver is unable to see the configuration file; in order to avoid this error, you can specify the configuration file path using the `--config` parameter in the command to execute tests, as follows:

```
java -jar JsTestDriver-1.3.4.b.jar --config jsTestDriver.
conf --tests all
```

After executing the JSTD tests, you will see the following result in the console if you have executed three successful tests (for example):

```
.......
Total 3 tests (Passed: 3; Fails: 0; Errors: 0) (2.00 ms)
Firefox 15.0 Windows: Run 3 tests (Passed: 3; Fails: 0; Errors 0) (2.00
ms)
… Other browsers here …
```

In order to have the `java` command available from the command line, you need to install and configure Java on your machine. It is expected that you have already installed the JRE as indicated in *Chapter 1, Unit Testing JavaScript Applications*, in order to run the Tomcat server. After installing the JRE, all you need to do to have the `java` command available from your command line is to add the JRE `bin` directory to the `PATH` variable of your operating system.

Writing your first JSTD test

A JSTD test can contain test cases and test functions. A JSTD test case is a group of related test functions. Every test function should contain one or more assertions in order to perform the test and verify the outputs. The JSTD `TestCase` object is responsible for creating the JSTD test case, and in order to create the test functions inside the test case, every test function should start with the word "test".

Every JSTD assertion represents a function that validates a condition that can return true or false. In order to pass the test function, all of the assertions inside the test function have to be true. If one or more assertions inside a test function are false, the test function fails. The following code snippet shows an example of two JSTD test cases with test functions:

```
TestCase1 = TestCase("Testcase1");

TestCase1.prototype.testFunction1 = function() {
  // One or more assertion(s)
};

TestCase1.prototype.testFunction2 = function() {
  // One or more assertion(s)
};

TestCase2 = TestCase("Testcase2");
```

```
TestCase2.prototype.testAnotherFunction = function() {
  // One or more assertion(s)
};
```

As shown in the preceding code snippet, two test cases are created. The first test case is named `Testcase1`, and it contains two test functions `testFunction1` and `testFunction2`. The second test case is named `Testcase2`, and it contains a single test function named `testAnotherFunction`.

Now, let's move to testing the `SimpleMath` JavaScript object (which we tested using Jasmine, YUI Test, and QUnit in the previous chapters). The following code snippet reminds you with the code of the `SimpleMath` object:

```
SimpleMath = function() {
};

SimpleMath.prototype.getFactorial = function (number) {

  if (number < 0) {
    throw new Error("There is no factorial for negative numbers");
  }
  else if (number == 1 || number == 0) {

    // If number <= 1 then number! = 1.
      return 1;
  } else {

    // If number > 1 then number! = number * (number-1)!
      return number * this.getFactorial(number-1);
  }
}

SimpleMath.prototype.signum = function (number) {
    if (number > 0)  {
    return 1;
  } else if (number == 0) {
    return 0;
  } else {
    return -1;
  }
}

SimpleMath.prototype.average = function (number1, number2) {
    return (number1 + number2) / 2;
}
```

As was done in the previous chapters, the following three test scenarios will be developed for the `getFactorial` method:

- A positive number
- Zero
- A negative number

The following code snippet shows how to test calculating the factorial of a positive number (`3`), `0`, and a negative number (`-10`) by using JSTD:

```
FactorialTestCase = TestCase("Factorial Testcase");

FactorialTestCase.prototype.setUp = function() {
  this.simpleMath = new SimpleMath();
};

FactorialTestCase.prototype.tearDown = function() {
  delete this.simpleMath;
};

FactorialTestCase.prototype.testPositiveNumber = function() {
  assertEquals("Factorial(3)", 6,
  this.simpleMath.getFactorial(3));
};

FactorialTestCase.prototype.testZero = function() {
  assertEquals("Factorial(0)", 1,
  this.simpleMath.getFactorial(0));
};

FactorialTestCase.prototype.testNegativeNumber = function() {
  var localThis = this;

  assertException("Factorial(-10)", function() {
            localThis.simpleMath.getFactorial(-10)
          }, "Error");
};
```

The `TestCase` object declares a new test case called `"Factorial Testcase"`. The `setUp` method is used to initialize the test functions in the test case, that is, the `setUp` method is called once before the run of each test function in the test case. In the `setUp` method, the `simpleMath` object is created using `new SimpleMath()`. On the contrary, the `tearDown` method is used to de-initialize the test functions in the test case; the `tearDown` method is called once after the run of each test function in the test case. In the factorial tests, the `tearDown` method is used to clean up, which deletes the created `simpleMath` object.

In the `testPositiveNumber` test function, the `assertEquals` assertion function calls `simpleMath.getFactorial(3)` and expects the result to be `6`. If `simpleMath.getFactorial(3)` returns a value other than `6`, the test fails. The first parameter of the `assertEquals` assertion is optional, and it represents a message to be displayed when the assertion fails. In JSTD, we have many other assertions to use instead of `assertEquals`; we will discuss them in greater detail in the *Assertions* section.

In the `testZero` test function, the `assertEquals` assertion function calls `simpleMath.getFactorial(0)` and expects it to be `1`. In the `testNegativeNumber` test function, the `assertEquals` assertion function calls `simpleMath.getFactorial(-10)` and expects it to throw an error by using the `assertException` assertion. In JSTD, the `assertException` assertion has three parameters; the first parameter is optional and represents a message to be displayed when the assertion fails, the second parameter represents a callback that contains the function to be tested (which must throw an error in order to make the test function pass), and the last parameter represents the string of the error type.

After finalizing the `getFactorial` test case, we come to a new test case that tests the functionality of the `signum` method provided by the `SimpleMath` object. The following code snippet shows the signum test case:

```
SignumTestCase = TestCase("Signum Testcase");

SignumTestCase.prototype.setUp = function() {
  this.simpleMath = new SimpleMath();
};

SignumTestCase.prototype.tearDown = function() {
  delete this.simpleMath;
};

SignumTestCase.prototype.testPositiveNumber = function() {
  assertEquals("Signum(3)", 1, this.simpleMath.signum(3));
};

SignumTestCase.prototype.testZero = function() {
  assertEquals("Signum(0)", 0, this.simpleMath.signum(0));
};

SignumTestCase.prototype.testNegativeNumber = function() {
  assertEquals("Signum(-1000)", -1, this.simpleMath.signum(-1000));
};
```

We have three test functions for the `signum` method, the `testPositiveNumber` function tests getting the signum of a positive number, the `testZero` function tests getting the signum of zero, and the `testNegativeNumber` function tests getting the signum of a negative number. The following code snippet shows the test case of the `average` method:

```
AverageTestCase = TestCase("Average Testcase");
AverageTestCase.prototype.setUp = function() {
  this.simpleMath = new SimpleMath();
};

AverageTestCase.prototype.tearDown = function() {
  delete this.simpleMath;
};

AverageTestCase.prototype.testAverage = function() {
  assertEquals("Average(3, 6)", 4.5, this.simpleMath.average(3, 6));
};
```

In `"Average Testcase"`, the `testAverage` test function ensures that the average is calculated correctly by calling the `average` method, using the two parameters `3` and `6`, and expecting the result to be `4.5`.

Note that the first optional message parameter in the JSTD assertions will be displayed if the assertion fails in a way that gives an error with a descriptive meaning. Let's assume that `getFactorial(3)` is returning a wrong value (for example, 10). This means that the following assertion will fail:

```
assertEquals("Factorial(3)", 6, this.simpleMath.getFactorial(3));
```

The result of running this failing assertion will be:

```
Factorial Testcase.testPositiveNumber failed (1.00 ms): AssertError:
Factorial(3) expected 6 but was 10
```

If the first message parameter is omitted, the result will be:

```
Factorial Testcase.testPositiveNumber failed (1.00 ms): AssertError:
expected 6 but was 10
```

In order to run the `SimpleMath` JSTD tests, you need to create the JSTD test configuration file that points to the source and test JavaScript files, and the server URL, as follows:

```
server: http://localhost:9876
load:
  - src/*.js
  - tests/*.js
```

> The `simpleMath.js` file is placed under the `src` folder, and the `simpleMathTest.js` file which contains the `SimpleMath` JSTD tests is placed under the `tests` folder. The `load` directive asks the JSTD test runner to load all of the JavaScript files (`*.js`) under the `src` folder and then to load all of the JavaScript files under the `tests` folder in order to execute the JSTD tests.

Then, start the server from the command line by using the following command:

```
java -jar JsTestDriver-1.3.4.b.jar --port 9876
```

Then, capture the browsers (for example, IE and Firefox) by entering the following URL in the browser's address bar:

```
http://localhost:9876/capture
```

Finally, you can execute the tests after you start the server and capture the browsers manually (or from the command line) by using the following command:

```
java -jar JsTestDriver-1.3.4.b.jar --tests all
```

After executing the test cases, you will find the following results in the console:

```
..............
Total 14 tests (Passed: 14; Fails: 0; Errors: 0) (8.00 ms)
Microsoft Internet Explorer 8.0 Windows: Run 7 tests (Passed: 7; Fails:
0; Errors 0) (0.00 ms)
Firefox 15.0.1 Windows: Run 7 tests (Passed: 7; Fails: 0; Errors 0) (8.00
ms)
```

Assertions

An assertion is a function that validates a condition; if the condition is not valid, it throws an error that causes the test to fail. A test method can include one or more assertions; all the assertions have to pass in order to have the test method pass. In the first JSTD test example, we have used the `assertEquals` and `assertException` assertions. In this section, the other different built-in assertions provided by JSTD will be illustrated.

The assert, assertTrue, and assertFalse([msg], expression) assertions

The `assert` and `assertTrue` assertions do the same thing; they have two parameters. The first parameter is an optional message to be displayed if the assertion fails, and the second parameter represents an expression. The `assert` and `assertTrue` assertions are passed if the expression parameter is evaluated to `true`. The `assertFalse` assertion does the reverse operation; it passes if the expression is evaluated to false. For example, the following assertions work:

```
assert(6 == 6);
assertTrue("6 should equal 6", 6 == 6);
assertFalse(6 != 6);
```

The assertEquals and assertNotEquals([msg], expected, actual) assertions

The `assertEquals` assertion has three parameters; the first parameter is an optional message to be displayed if the assertion fails, and the last two parameters represent the expected and actual values. The `assertEquals` assertion is passed if the actual value is equal to the expected value; if it is not, the assertion fails and the optional message is displayed. The `assertNotEquals` assertion ensures that the actual and expected parameters are not equal.

It is very important to know that the `assertEquals` and `assertNotEquals` assertions use the JavaScript == operator to perform the comparison, that is, they carry out the comparison neglecting the types. For example, the following assertions will be passed:

```
assertEquals("6 should equal '6'", 6, "6");
assertNotEquals("6 should not equal 7", 6, 7);
```

The assertSame and assertNotSame([msg], expected, actual) assertions

The `assertSame` and `assertNotSame` assertions are very similar to the `assertEquals` and `assertNotEquals` assertions. The main difference between them is that the `assertSame` and `assertNotSame` assertions use the === operator for comparison, that is, they compare both the values and the types of the actual and expected parameters. For example, the following assertions will be passed:

```
assertSame("6 is the same as 6", 6, 6);
assertNotSame("6 is not the same as '6'", 6, "6");
```

The datatype assertions

The following set of assertions in JSTD checks the value types. Each one of these assertions takes two parameters; the first parameter is an optional message to be displayed if the assertion fails, and the second parameter is the value to be tested:

- `assertBoolean([msg], actual)` is passed if the actual value is a Boolean
- `assertString([msg], actual)` is passed if the actual value is a string
- `assertNumber([msg], actual)` is passed if the actual value is a number
- `assertArray([msg], actual)` is passed if the actual value is an array
- `assertFunction([msg], actual)` is passed if the actual value is a function
- `assertObject([msg], actual)` is passed if the actual value is an object

For example, the following assertions will be passed:

```
assertBoolean(false);
assertString("some string");
assertNumber(1000);
assertArray([1, 2, 3]);
assertFunction(function(){ alert('test'); });
assertObject({somekey: 'someValue'});
```

JSTD also provides generic assertions, `assertTypeOf` and `assertInstanceOf`, for checking the datatypes.

The `assertTypeOf` assertion uses the JavaScript `typeof` operator in order to check the value type. It takes three parameters; the first parameter is an optional message to be displayed if the assertion fails, and the other two parameters represent the value type and the value to test. For example, the following `assertTypeOf` assertions will pass:

```
assertTypeOf("boolean", false);
assertTypeOf("string", "some string");
assertTypeOf("number", 1000);
assertTypeOf("object", [1, 2, 3]);
assertTypeOf("function", function(){ alert('test'); });
assertTypeOf("object", {somekey: 'someValue'});
```

In addition to all of this, you can use the `assertInstanceOf` assertion, which uses the JavaScript `instanceof` operator in order to check the value instance. It takes three parameters; the first parameter is an optional message to be displayed if the assertion fails, and the other two parameters represent the type constructor and the value to be tested. For example, the following assertions will pass:

```
assertInstanceOf(Boolean, false);
assertInstanceOf(String, "some string");
assertInstanceOf(Number, 1000);
assertInstanceOf(Object, [1, 2, 3]);
assertInstanceOf(Function, function(){ alert('test'); });
assertInstanceOf(Object, {somekey: 'someValue'});
```

Special value assertions

The following set of assertions in JSTD checks whether a variable value belongs to one of the special values as mentioned in the following list. Each one of these assertions takes two parameters; the first parameter is an optional message to be displayed if the assertion fails, and the second parameter is the value to be tested:

- `assertUndefined([msg], actual)` is passed if the actual value is undefined
- `assertNotUndefined([msg], actual)` is passed if the actual value is not undefined (defined)
- `assertNull([msg], actual)` is passed if the actual value is null
- `assertNotNull([msg], actual)` is passed if the actual value is not null
- `assertNaN([msg], actual)` is passed if the actual value is not a number (NaN)
- `assertNotNaN([msg], actual)` is passed if the actual value is not NaN

For example, the following assertions will be passed:

```
var someStr = "some string";
var undefinedVar;

assertUndefined(undefinedVar);
assertNotUndefined(someStr);
assertNull(null);
assertNotNull(someStr);
assertNaN(1000 / "string_value");
assertNotNaN(1000);
```

The fail([msg]) assertion

In some situations, you may need to fail the test manually, for example, if you want to make your own custom assertion that encapsulates specific validation logic. In order to do this, JSTD provides the `fail()` method for failing the test manually. `assertAverage` is an example of a custom assertion that uses the `fail()` method:

```
assertAverage = function (number1, number2, expected, failureMessage)
{
  var actual = (number1 + number2) / 2;

  if (actual != expected) {
    fail(failureMessage + ": Expected = " + expected + " while
    Actual = " + actual);
  }
}
```

The `assertAverage` custom assertion can be called by simply using the following line of code:

```
assertAverage(3, 4, 3.5, "Average is incorrect");
```

The `fail()` method has an optional message parameter that is displayed as a failure message.

> There are other remaining built-in assertions in JSTD; however, the only remaining important built-in assertion that you have to know is `assertException`, and you already learned how it works in the `SimpleMath` object test example.

Testing asynchronous (Ajax) JavaScript code

The common question that comes to mind is how to test asynchronous (Ajax) JavaScript code using JSTD. What has been mentioned in the chapter so far is how to perform unit testing for the synchronous JavaScript code. Fortunately, JSTD provides the `AsyncTestCase` object in order to perform asynchronous JavaScript testing (Ajax testing). In the following section, you will understand how to work with the `AsyncTestCase` object in order to develop asynchronous tests in JSTD.

AsyncTestCase, queue, and callbacks

`AsyncTestCase` extends `TestCase` by allowing the test methods to have a `queue` parameter. The `queue` parameter can contain a list of inline functions (steps) that are executed in sequence. Every inline function has a `callbacks` parameter that allows creating different callbacks for testing the asynchronous operations. JSTD mainly has two types of callbacks:

- **Success callbacks**: These represent the success path. The success callback must be called if the Ajax operation succeeds. In order to handle the operation timeout, if the success callback is not called after a specific amount of time (30 seconds, by default), the test function fails.
- **Error callbacks**: These represent the error path. The error callback must not be called if the Ajax operation succeeds. If the error callback is called, the test function fails.

The JSTD `queue` parameter contains a list of inline functions that are executed in sequence, which is helpful if you want to test a group of dependent Ajax operations.

The following code snippet shows an example of real Ajax testing using JSTD:

```
AsynchronousTestCase = AsyncTestCase("Asynchronous Testcase");

AsynchronousTestCase.prototype.testAjaxOperationsGroup1 =
function(queue) {
  queue.call('Testing operation1 ...', function(callbacks) {
      var successCallback =
      callbacks.add(function(successParameters) {
      // Make the assertions for the successParameters...
        });

      var failureCallback = callbacks.addErrback('Unable to
      register the user');

      // call asynchronous API
      asyncSystem.doAjaxOperation(inputData, successCallback,
      failureCallback);
  });

  queue.call('Testing operation2 ...', function(callbacks) {
    // will be called after 'Testing operation1 ...'
  });
};
```

```
AsynchronousTestCase.prototype.testAjaxOperationsGroup2 =
function(queue) {
  //...
};
```

As shown in the preceding code snippet, an asynchronous test case, `"Asynchronous Testcase"`, is created. It has two test methods: `testAjaxOperationsGroup1` and `testAjaxOperationsGroup2`. Every test method has a `queue` parameter. In the `testAjaxOperationsGroup1` test method, the `queue` object includes two inline functions, `"Testing operation1 ..."` and `"Testing operation2 ..."`, using the `queue.call()` API.

> The `queue.call()` API has two parameters; the first parameter is optional and represents the title of the inline function, and the second parameter represents the inline function.

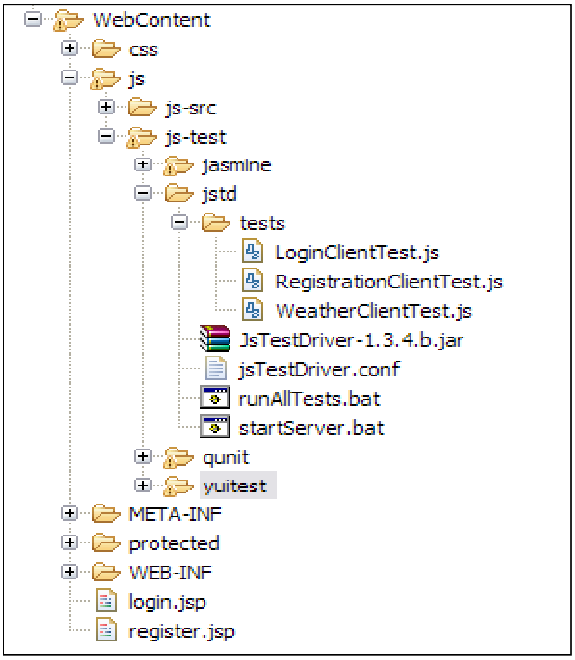
Every inline function has a `callbacks` parameter. The `callbacks` parameter allows creating the success and failure callbacks in order to test and validate the Ajax operations. In the `"Testing operation1 ..."` inline function, two callbacks are created; one of them is the success callback (`successCallback`) and it is called if the Ajax operation succeeds. The success callback is created using the `callbacks.add()` API. The other callback is the failure callback (`failureCallback`), and it is called if the Ajax operation fails. The failure callback is created using the `callbacks.addErrback()` API.

If the Ajax response is not returned from the server after 30 seconds, the success callback will cause the test to fail. In the next section, the `AsyncTestCase`, `queue`, and `callbacks` objects will be used in order to test the (asynchronous) Ajax part of the weather application.

Testing the weather application

Now, we come to developing the JSTD tests for our weather application. Actually, after you know how to write JSTD tests for both synchronous and asynchronous JavaScript (Ajax) code, testing the weather application is an easy task. As you remember from the previous chapters, we have three major JavaScript objects in the weather application that we need to develop tests for: the `LoginClient`, `RegistrationClient`, and `WeatherClient` objects.

Two subfolders, `jstd` and `tests`, are created under the `js-test` folder (thus: `jstd\tests`) to contain the JSTD tests, as shown in the following screenshot:



```
WebContent
  css
  js
    js-src
    js-test
      jasmine
      jstd
        tests
          LoginClientTest.js
          RegistrationClientTest.js
          WeatherClientTest.js
        JsTestDriver-1.3.4.b.jar
        jsTestDriver.conf
        runAllTests.bat
        startServer.bat
      qunit
      yuitest
  META-INF
  protected
  WEB-INF
  login.jsp
  register.jsp
```

As shown in the preceding screenshot, there are three JSTD test files (`LoginClientTest.js`, `RegistrationClientTest.js`, and `WeatherClientTest.js`) that test the main three JavaScript objects of the weather application.

Using the JSTD `DOC` annotation, you can load the HTML fixtures (in an inline style) in your JSTD tests. For example:

```
FixtureTestCase = TestCase("Fixture Testcase");
FixtureTestCase.prototype.testSomeThing = function()
{
  /*:DOC += <div id="someDiv"></div> */
  assertNotNull(document.getElementById('someDiv'));
};
```

Testing the LoginClient object

As we did in the previous chapters, in the *Testing the LoginClient object* section we will perform unit testing for the following functionalities:

- Validation of empty username and password
- Validating that the username is in e-mail address format
- Validating that the password contains at least one digit, one capital and small letter, at least one special character, and six characters or more

In order to perform this test, one test case is created that tests both the validation of the empty fields (the username and password) and the validation of the fields' formats. The following code snippet shows the validation of the empty fields for `"LoginClient Testcase"`:

```
LoginClientTestcase = TestCase("LoginClient Testcase");

LoginClientTestcase.prototype.setUp = function() {
  /*:DOC += <label for="username">Username  <span
  id="usernameMessage" class="error"></span></label>
   <input type="text" id="username" name="username"/>
   <label for="password">Password  <span id="passwordMessage"
   class="error"></span></label>
   <input type="password" id="password" name="password"/>*/

  this.loginClient = new weatherapp.LoginClient();

    this.loginForm = {
    "userNameField" : "username",
     "passwordField" : "password",
     "userNameMessage" : "usernameMessage",
     "passwordMessage" : "passwordMessage"
    };
};

LoginClientTestcase.prototype.tearDown = function() {
  delete this.loginClient;
  delete this.loginForm;
};

LoginClientTestcase.prototype.testEmptyUserName = function() {
  document.getElementById("username").value = ""; /* setting
  username to empty */
  document.getElementById("password").value = "Admin@123";
```

```
      this.loginClient.validateLoginForm(this.loginForm);

      assertEquals("(field is required)",
      document.getElementById("usernameMessage").innerHTML);
    };

    LoginClientTestcase.prototype.testEmptyPassword = function() {
      document.getElementById("username").value = "someone@yahoo.com";
      document.getElementById("password").value = "";   /* setting
      password to empty */

      this.loginClient.validateLoginForm(this.loginForm);

      assertEquals("(field is required)",
      document.getElementById("passwordMessage").innerHTML);
    };
```

The `setUp` method creates an instance from `weatherapp.LoginClient` and creates the `loginForm` object, which holds the IDs of the HTML elements that are used in the test. The HTML fixture of `LoginClientTestCase` is loaded using the `DOC` annotation.

`testEmptyUserName` tests whether the `LoginClient` object is able to display an error message when the username is not entered. It sets an empty value in the `username` field and then calls the `validateLoginForm` API of the `LoginClient` object. It then checks whether the `validateLoginForm` API produces the `"(field is required)"` message in the `usernameMessage` field by using the `assertEquals` assertion.

`testEmptyPassword` does the same thing, but with the `password` field, not with the `username` field.

The following code snippet shows the second part of `"LoginClient Testcase"`, which validates the formats of the fields (username and password):

```
    LoginClientTestcase.prototype.testUsernameFormat = function() {
      document.getElementById("username").value =
      "someone@someDomain";  /* setting username to invalid format */
      document.getElementById("password").value = "Admin@123";

      this.loginClient.validateLoginForm(this.loginForm);

      assertEquals("(format is invalid)",
      document.getElementById("usernameMessage").innerHTML);
    };

    LoginClientTestcase.prototype.testPasswordFormat = function() {
      document.getElementById("username").value =
      "someone@someDomain.com";
      document.getElementById("password").value = "Admin123"; /*
      setting password to invalid format */

      this.loginClient.validateLoginForm(this.loginForm);

      assertEquals("(format is invalid)",
      document.getElementById("passwordMessage").innerHTML);
    };
```

`testUsernameFormat` tests the validation of the username format. It tests whether the `LoginClient` object is able to display an error message when the username format is not valid. It sets an invalid e-mail value in the `username` field and then calls the `validateLoginForm` API of the `LoginClient` object. Finally, it checks, by using the `assertEquals` assertion, whether the `validateLoginForm` API produces the `"(format is invalid)"` message in the `usernameMessage` field.

`testPasswordFormat` enters a password that does not comply with the application's password rules—it enters a password that does not include a capital letter—and then calls the `validateLoginForm` API of the `LoginClient` object. Finally, it checks whether the `validateLoginForm` API produces the `"(format is invalid)"` message in the `passwordMessage` field.

Testing the RegistrationClient object

In the `RegistrationClient` object, we will test the following functionalities:

- Validation of empty username and passwords
- Validation of matched passwords
- Validating that the username is in e-mail address format
- Validating that the password contains at least one digit, one capital and small letter, at least one special character, and six characters or more
- Validating that the user registration Ajax functionality is performed correctly

Testing of the first four functionalities will be skipped because they are pretty similar to the tests that are explained in the `LoginClient` test case, so let's explain how to check whether the user registration (`registerUser`) Ajax functionality is performed correctly.

The registerUser test case should cover the following test scenarios:

- Testing adding a new user, that is, the registration client should be able to register a valid user correctly.
- Testing adding a user with an existing user ID (username). In this case, the registration client should fail when registering a user whose ID is already registered.

The RegistrationTestcase asynchronous test case is created in order to validate the user registration Ajax functionality. The following code snippet shows the first part of the RegistrationTestcase test case, which tests adding a new user. The setUp method creates an instance from weatherapp.RegistrationClient and creates the registrationForm object, which holds the IDs of the registration form that will be used in the test. Using the DOC annotation, the HTML fixture of RegistrationTestcase is loaded:

```
RegistrationTestcase = AsyncTestCase("Registration Testcase");

RegistrationTestcase.prototype.setUp = function() {
  /*:DOC += <label for="username">Username (Email)  <span
  id="usernameMessage" class="error"></span></label>
  <input type="text" id="username" name="username"/>
  <label for="password1">Password  <span id="passwordMessage1"
  class="error"></span></label>
  <input type="password" id="password1" name="password1"/>
  <label for="password2">Confirm your password</label>
  <input type="password" id="password2" name="password2"/>*/

  this.registrationClient = new weatherapp.RegistrationClient();

  this.registrationForm = {
      "userNameField" : "username",
      "passwordField1" : "password1",
      "passwordField2" : "password2",
      "userNameMessage" : "usernameMessage",
      "passwordMessage1" : "passwordMessage1"
    };
};

RegistrationTestcase.prototype.tearDown = function() {
  delete this.registrationClient;
  delete this.registrationForm;
};
```

```
RegistrationTestcase.prototype.testRegisterUser = function(queue) {
  var this_local = this;

  queue.call('Registering a new user ...', function(callbacks) {
    this_local.userName = "hazems" + new Date().getTime() +
    "@apache.org";

    document.getElementById("username").value =
    this_local.userName;
    document.getElementById("password1").value = "Admin@123";
    document.getElementById("password2").value = "Admin@123";

      var successCallback = callbacks.add(function(response) {
        var resultMessage = response.xmlhttp.responseText;
        assertEquals("User is registered successfully ...",
        resultMessage);

        jstestdriver.console.log("[Success] User is registered
        successfully ...");
        });

      var failureCallback = callbacks.addErrback('Unable to
      register the user');

      // call asynchronous API
this_local.registrationClient.registerUser(this_local.
registrationForm, successCallback, failureCallback);
  });

  //...
};
```

In the `testRegisterUser` test method, an inline function `('Registering a new user ...')`, tests the creation of a new user using `queue.call()`. The registration form is filled with a valid generated username and valid matched passwords, and then two callbacks are created using the `callbacks` parameter. `successCallback` which represents the success callback, while `failureCallback` represents the failure callback. `registrationClient.registerUser` is called with the registration form, the success callback, and the failure callback parameters. The `'Registering a new user ...'` inline test function waits for a call to either the success or failure callback, or it fails after the timeout period passes.

In `successCallback`, the callback checks whether the returned response message from the server is `"User is registered successfully ..."` using the `assertEquals` assertion. The failure callback displays the `"Unable to register the user"` message if the `registerUser` test case fails.

[The `jstestdriver.console.log` API can be used for JSTD logging in the console. In the example, it is used to display the "operation successful" message.]

The following code snippet shows the second inline function, `'Registering a user whose id is already existing ...'`, of the `testRegisterUser` test method. It tests registering a user with an existing ID:

```
queue.call('Registering a user whose id is already existing ...',
function(callbacks) {
  document.getElementById("username").value = this_local.userName;
  document.getElementById("password1").value = "Admin@123";
  document.getElementById("password2").value = "Admin@123";

  var failureCallback = callbacks.add(function(response) {
    var resultMessage = response.xmlhttp.responseText;
    assertEquals("A user with the same username is already registered
    ...", resultMessage);

    jstestdriver.console.log("[Success] User is not created because
    the user id is already registered ...");
  });

  var successCallback = callbacks.addErrback('[Error] A user with the
  same id is created !!!');

  // call asynchronous API
  this_local.registrationClient.registerUser(this_local.
registrationForm, successCallback, failureCallback);
});
```

The registration form is filled with the existing user ID (that is already registered in the first inline test function) and with a valid password, and the failure and the success callbacks are created. Then, `registrationClient.registerUser` is called with the registration form, the success callback, and the failure callback parameters.

In the failure callback (which must be called if the `registerUser` test case works correctly), the callback checks whether the returned response message from the server is `"A user with the same username is already registered ..."`, using the `assertEquals` assertion. The success callback displays the `"[Error] A user with the same id is created !!!"` message if the `registerUser` test case creates a new user with an existing user ID.

Testing the WeatherClient object

In the `WeatherClient` object, we will unit test the following functionalities:

- Getting the weather for a valid location
- Getting the weather for an invalid location (the system should display an error message for this case)

To test the `WeatherClient` object, the same technique that we used in the `registerUser` test case is followed. Developing this test will be left for you as an exercise; you can get the full source code of the `WeatherClientTest.js` file from the `Chapter 5` folder in the code bundle available on the book's website. To view the source code for the JavaScript tests, all you need to do is unzip the `weatherApplication.zip` file, and you will be able to find all the JSTD tests under `weatherApplication/WebContent/js/js-test/jstd/tests`.

Configuring the proxy

In order to allow sending Ajax requests from the JSTD server to the backend server, we need to access the backend server through a proxy so as to avoid the "security permission denied" error that occurs due to the cross-domain request(s). Fortunately, JSTD provides a gateway (proxy) that can be used for this purpose. The following code snippet shows the complete `JsTestDriver.config` file of the weather application JSTD test:

```
server: http://localhost:9876

gateway:
 - {matcher: "*", server: "http://localhost:8080"}

load:
  - ../../js-src/*.js
  - tests/*.js

plugin:
  - name: "coverage"
    jar: coverage-1.3.4.b.jar
    module: "com.google.jstestdriver.coverage.CoverageModule"
```

In the configuration file, there are two newly introduced directives (that are not explained in the first JSTD test example), the `gateway` and `plugin` directives. The `plugin` directive is used to define a JSTD plugin. For this example, it defines the code coverage plugin that is used to generate the test reports (this plugin will be illustrated in detail in the *Generating test reports* section). The `gateway` directive can be used to route the requests that match the `matcher` attribute's pattern to the corresponding server URL specified in the `server` attribute. For the weather application tests, all of the Ajax requests (which are represented using the `"*"` pattern) will be routed to the backend server in `http://localhost:8080`, which hosts the weather application backend APIs.

In the matcher attribute, you can use the following varieties of patterns:

- **Literal matchers**: For example, `"/matchedService"`
- **Suffix matchers**: For example, `"/matchedService/*"`
- **Prefix matchers**: For example, `"*.json"`

Running the weather application tests

In order to run the weather application tests correctly, you have to make sure that the Tomcat server is up and running and that this chapter's updated version of the weather application is deployed on the server as explained in *Chapter 1, Unit Testing JavaScript Applications*. After this, you need to follow these steps:

1. Launch the command prompt and change directory (`cd`) to the `"${INSTALL_PATH}\weatherApplication\WebContent\js\js-test\jstd\"` path in the deployed weather application.
2. Start the JSTD server by typing the command `java -jar JsTestDriver-1.3.4.b.jar --port 9876`.
3. Capture the browsers (for example, Firefox and Internet Explorer) by entering the following URL in the browser's address bar:

   ```
   http://localhost:9876/capture
   ```

4. Finally, run the JSTD test command as follows:

   ```
   java -jar JsTestDriver-1.3.4.b.jar --config jsTestDriver.conf
   --tests all
   ```

The following result snippet shows the output of running the JSTD tests of the weather application:

```
........................
Total 24 tests (Passed: 24; Fails: 0; Errors: 0) (1297.00 ms)
  Microsoft Internet Explorer 8.0 Windows: Run 12 tests (Passed: 12;
  Fails: 0; Errors 0) (610.00 ms)
    Registration Testcase.testRegisterUser passed (32.00 ms)
    [LOG] [Success] User is registered successfully...
    [LOG] [Success] User is not created because the user id is
    already registered...
    WeatherClient Testcase.testGetWeatherForValidPlace passed (328.00
    ms)
    [LOG] [Success] Weather information is retrieved successfully...
    WeatherClient Testcase.testGetWeatherForInvalidPlace passed
    (250.00 ms)
    [LOG] [Success] The weather information is not retrieved for the
    invalid place...
  Firefox 15.0.1 Windows: Run 12 tests (Passed: 12; Fails: 0; Errors
  0) (1297.00 ms)
    Registration Testcase.testRegisterUser passed (493.00 ms)
    [LOG] [Success] User is registered successfully...
    [LOG] [Success] User is not created because the user id is
    already registered...
    WeatherClient Testcase.testGetWeatherForValidPlace passed (539.00
    ms)
    [LOG] [Success] Weather information is retrieved successfully...
    WeatherClient Testcase.testGetWeatherForInvalidPlace passed
    (252.00 ms)
    [LOG] [Success] The weather information is not retrieved for the
    invalid place...
```

Generating test reports

JSTD can generate test reports from the test results by using the code coverage plugin. The code coverage plugin can also produce code coverage files—in the **Linux code coverage** (**LCOV**) format—which include the test code coverage statistics.

Code coverage is a software testing measure. It shows how much the source code of a program has been tested. It has many criteria for this measurement, for example:

- Line coverage measures the percentage of the program lines that are covered by the test
- Function coverage measures the percentage of the program functions that are covered by the test
- Branch coverage measures the percentage of the program branches (for example, *if … else*) that are covered by the test

In order to generate the test reports from the JSTD tests, you need to do the following:

1. Download the code coverage plugin file (`coverage-1.3.4.b.jar`) from the download page of JSTD, which can be found at the following location:

   `http://code.google.com/p/js-test-driver/downloads/list`

2. Add the code coverage plugin declaration to the `JsTestDriver.conf` file, as follows:

   ```
   plugin:
     - name: "coverage"
       jar: coverage-1.3.4.b.jar
       module: "com.google.jstestdriver.coverage.CoverageModule"
   ```

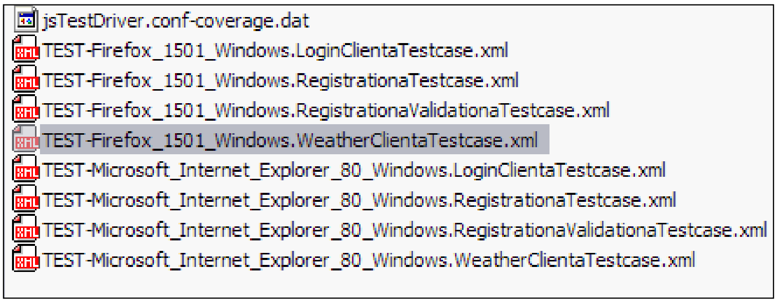
   This declaration tells JSTD to include the plugin whose name is `coverage` from the `com.google.jstestdriver.coverage.CoverageModule` module that resides in the `coverage-1.3.4.b.jar` file.

3. Finally, you need to specify the `--testOutput` parameter in the test running command. The `--testOutput` parameter specifies the path in which JSTD will generate the test report files. For example:

   ```
   java -jar JsTestDriver-1.3.4.b.jar --config jsTestDriver.conf
   --tests all --testOutput reports
   ```

   This command tells JSTD to generate the test reports under the `reports` directory.

The following screenshot shows the generated report files after performing the three preceding steps:



jsTestDriver.conf-coverage.dat
TEST-Firefox_1501_Windows.LoginClientaTestcase.xml
TEST-Firefox_1501_Windows.RegistrationaTestcase.xml
TEST-Firefox_1501_Windows.RegistrationaValidationaTestcase.xml
TEST-Firefox_1501_Windows.WeatherClientaTestcase.xml
TEST-Microsoft_Internet_Explorer_80_Windows.LoginClientaTestcase.xml
TEST-Microsoft_Internet_Explorer_80_Windows.RegistrationaTestcase.xml
TEST-Microsoft_Internet_Explorer_80_Windows.RegistrationaValidationaTestcase.xml
TEST-Microsoft_Internet_Explorer_80_Windows.WeatherClientaTestcase.xml

As shown in the preceding screenshot, there are nine generated files. The `jsTestDriver.conf-coverage.dat` file is the generated LCOV file that contains the code coverage statistics (currently, the JSTD code coverage plugin generates the code coverage based on the line coverage criteria). The other eight files are JUnit XML report files that have the following naming format:

```
TEST-[BrowserName_Version_Platform].[TestCaseShortName]Testcase.xml
```

In the weather application, there are four test cases that have four XML files per browser:

- `"LoginClient Testcase"`
- `"Registration Validation Testcase"`
- `"Registration Testcase"`
- `"WeatherClient Testcase"`

As a sample for the generated test report files, the following code snippet shows the `"LoginClient Testcase"` JUnit XML report file for Firefox. The displayed JUnit XML report file is named `TEST-Firefox_1501_Windows.LoginClientaTestcase.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuite name="Firefox_1501_Windows.LoginClient Testcase" errors="0"
failures="0" tests="4" time="0.013">
<testcase classname="Firefox_1501_Windows.LoginClient Testcase"
name="testEmptyUserName" time="0.0040"/>
<testcase classname="Firefox_1501_Windows.LoginClient Testcase"
name="testEmptyPassword" time="0.0010"/>
<testcase classname="Firefox_1501_Windows.LoginClient Testcase"
name="testUsernameFormat" time="0.0060"/>
<testcase classname="Firefox_1501_Windows.LoginClient Testcase"
name="testPasswordFormat" time="0.0020"/>
</testsuite>
```

Now, let's learn how to generate user-friendly code coverage reports in JSTD. Fortunately, the JSTD-generated LCOV files can be converted to user-friendly HTML reports using the LCOV visualizer tool that can be found at `http://ltp.sourceforge.net/coverage/lcov.php`.

The LCOV visualizer works on a Red Hat Linux environment. In order to convert the LCOV files to HTML reports, you should do the following:

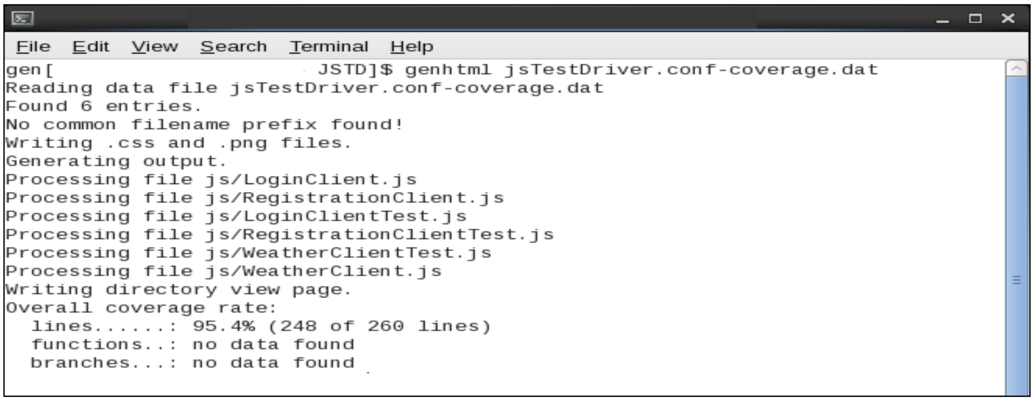
1. Download the latest LCOV visualizer RPM (`lcov-X.Y-Z.noarch.rpm`) file from `http://ltp.sourceforge.net/coverage/lcov.php`
2. Install the downloaded RPM file in your Red Hat Linux environment by using the following command:

   ```
   rpm -i lcov-1.9-1.noarch.rpm
   ```

3. In order to make sure that the LCOV visualizer tool is installed correctly, type the `genhtml` command at the command line, and you should see the following output:

   ```
   genhtml: No filename specified
   Use genhtml --help to get usage information
   ```

4. Run the `genhtml` command on the JSTD-generated LCOV file in order to generate the HTML test coverage report shown in the following screenshot:



```
gen[                          JSTD]$ genhtml jsTestDriver.conf-coverage.dat
Reading data file jsTestDriver.conf-coverage.dat
Found 6 entries.
No common filename prefix found!
Writing .css and .png files.
Generating output.
Processing file js/LoginClient.js
Processing file js/RegistrationClient.js
Processing file js/LoginClientTest.js
Processing file js/RegistrationClientTest.js
Processing file js/WeatherClientTest.js
Processing file js/WeatherClient.js
Writing directory view page.
Overall coverage rate:
  lines......: 95.4% (248 of 260 lines)
  functions..: no data found
  branches...: no data found
```

Note that the `jsTestDriver.conf-coverage.dat` file has the format shown in the following code snippet:

```
SF:[PATH]\Workspaces\weatherApplication\WebContent\js\LoginClient.
js
DA:1,2
...
end_of_record
```

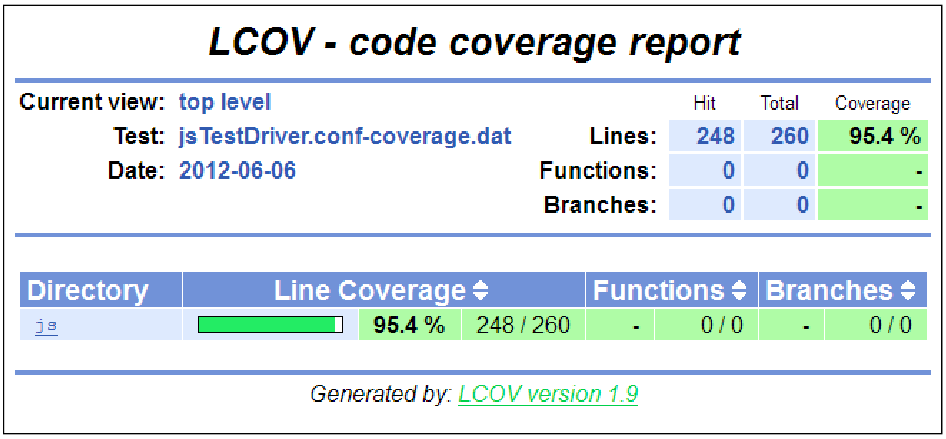
```
SF:[PATH]\weatherApplication\WebContent\js\RegistrationClient.js
DA:1,2
...
end_of_record
SF:[PATH]\weatherApplication\WebContent\js\WeatherClient.js
DA:1,2
...
end_of_record
SF:[PATH]\weatherApplication\WebContent\js\LoginClientTest.js
DA:10,2
...
end_of_record
SF:[PATH]\weatherApplication\WebContent\js\RegistrationClientTest.
js
DA:12,2
...
end_of_record
SF:[PATH]\weatherApplication\WebContent\js\WeatherClientTest.js
DA:8,2
...
end_of_record
```

As shown in the LCOV generated file, the generated `SF` attributes contain the full paths of both the JavaScript source and test files. So, you have to make sure that these paths are updated if you change the location of the JavaScript files.

If you run the `genhtml` command and the paths of the `SF` attributes are not correct, you will encounter the following error:

```
"mkdir: cannot create directory `': No such file or
directory
genhtml: ERROR: cannot create directory !"
```

5. If the `genhtml` command is passed successfully, you will find the generated HTML code coverage report files; click on the `index.html` file to see the HTML report shown in the following screenshot:



LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level				
Test:	jsTestDriver.conf-coverage.dat	Lines:	248	260	95.4 %
Date:	2012-06-06	Functions:	0	0	-
		Branches:	0	0	-

Directory	Line Coverage			Functions		Branches	
js	🟩	95.4 %	248 / 260	-	0 / 0	-	0 / 0

Generated by: LCOV version 1.9

You can drill down in the report by clicking on the `js` directory link to see the test result details of each JavaScript file.

The generated LCOV HTML report is placed under the `lcov-html` folder, which is under the `jstd` folder; you can access the generated HTML report on your Tomcat server by using the following URL:

`http://localhost:8080/weatherApplication/js/js-test/jstd/lcov-html/index.html`

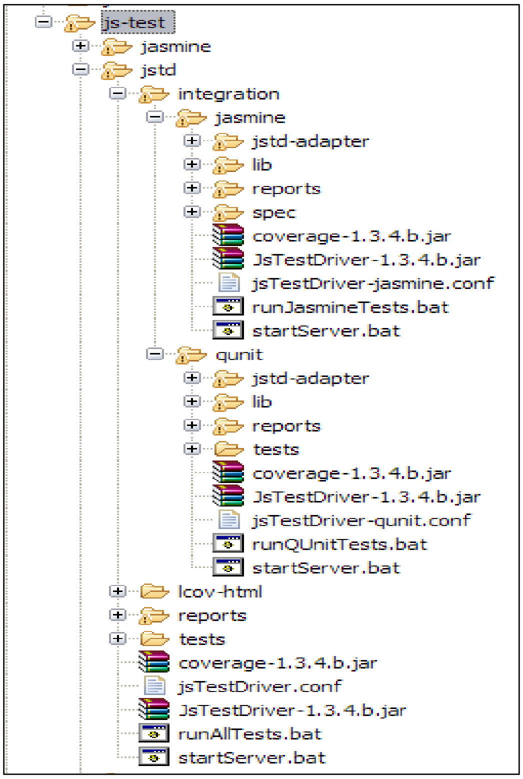
Integration with other JavaScript test frameworks

As we know from the definition of JSTD, it is not only a JavaScript test framework but also a complete test runner that can run other JavaScript frameworks on top of it, using **adapters**. Fortunately, JSTD has many ready-made adapters, developed by the open source community, that enable many JavaScript frameworks (such as Jasmine, QUnit, and YUI Test) to integrate with JSTD. The integration of JSTD with both Jasmine and QUnit is highly required because these testing frameworks do not have an out-of-the-box mechanism for executing the tests from the command-line interface (unlike YUI Test, which can run from the command line using YUI Test Selenium

Driver, as illustrated in detail in *Chapter 3, YUI Test*). Having the ability to execute the tests from the command-line interface allows automating the running of tests by using the build and the continuous integration tools.

In this section, the required steps and tricks that are needed for integrating our previously written Jasmine and QUnit tests (the weather application) with the JSTD runner will be illustrated.

Before digging into the details of this integration, let's see the structure of the `integration` folder, which contains the JSTD-Jasmine and JSTD-QUnit integration files in the weather application:



```
js-test
    jasmine
    jstd
        integration
            jasmine
                jstd-adapter
                lib
                reports
                spec
                coverage-1.3.4.b.jar
                JsTestDriver-1.3.4.b.jar
                jsTestDriver-jasmine.conf
                runJasmineTests.bat
                startServer.bat
            qunit
                jstd-adapter
                lib
                reports
                tests
                coverage-1.3.4.b.jar
                JsTestDriver-1.3.4.b.jar
                jsTestDriver-qunit.conf
                runQUnitTests.bat
                startServer.bat
        lcov-html
        reports
        tests
        coverage-1.3.4.b.jar
        jsTestDriver.conf
        JsTestDriver-1.3.4.b.jar
        runAllTests.bat
        startServer.bat
```

As shown in the preceding screenshot, the `integration` folder contains two subfolders—the `jasmine` folder and the `qunit` folder. The `jasmine` folder and the `qunit` folder each contain the following subfolders:

- `jstd-adapter`: These contain the JSTD adapter files
- `lib`: These contain the JavaScript framework library files (whether Jasmine or QUnit)

- `spec` and `tests`: These contain the Jasmine and QUnit test files, respectively
- `reports`: These contain the report files

The `jasmine` folder and the `qunit` folder contain the following files:

- `JsTestDriver-1.3.4.b.jar`: The JSTD JAR file.
- `coverage-1.3.4.b.jar`: The JSTD code coverage JAR file.
- `jsTestDriver-*.conf`: The JSTD configuration files.
- Two batch files that can start the JSTD server and execute the Jasmine (or QUnit) tests. These two files will work with you if you are using Windows; if you are working in a Linux environment, you can create equivalent `.sh` files to start up the server and execute the tests without having to remember the commands.

Integrating JSTD with Jasmine

In order to integrate the Jasmine tests of the weather application with JSTD, you need to:

1. Download the `JasmineAdapter.js` file from the following URL:

   `https://github.com/ibolmo/jasmine-jstd-adapter/blob/master/src/JasmineAdapter.js`

2. Place the downloaded `JasmineAdapter.js` file under the `/integration/jasmine/jstd-adapter` folder.
3. Create a configuration file, named `jsTestDriver-jasmine.conf`, that contains the JSTD-Jasmine configuration, which is shown in the following code snippet:

```
server: http://localhost:9876
gateway:
 - {matcher: "*", server: "http://localhost:8080"}
load:
  - lib/jasmine-1.2.0/jasmine.js
  - lib/plugins/jasmine-jquery/jquery.js
  - jstd-adapter/JasmineAdapter.js
  - lib/plugins/jasmine-jquery/jasmine-jquery.js
  - ../../../../js-src/*.js
  - spec/*.js
plugin:
  - name: "coverage"
    jar: coverage-1.3.4.b.jar
    module: "com.google.jstestdriver.coverage.CoverageModule"
```

In the `load` directive, you need to load the following files in order:

- The Jasmine framework file
- The jQuery file
- The Jasmine JSTD adapter file
- The Jasmine jQuery plugin file
- The JavaScript source files
- The Jasmine JavaScript test files

This is basically what is needed in order to have Jasmine tests running on top of the JSTD test runner. However, you need to take the `loadFixtures` API of the Jasmine jQuery plugin into consideration. Due to the changes of the paths between JSTD and Jasmine, the `loadFixtures` API will not work correctly. In order to run the loading of the fixture correctly, you have two options:

- Replace the `loadFixtures` API with the `jasmine.getFixtures().set` API and load the fixtures in an inline style (which is the approach followed in the weather application's JSTD-Jasmine tests)
- Configure the Jasmine `loadFixtures` API to work with JSTD

In order to configure the `loadFixtures` API to work with JSTD, you need to do the following:

1. Specify explicitly the fixture path by using `jasmine.getFixtures().fixturesPath`, and start the fixture path with `/test`, as follows:

   ```
   jasmine.getFixtures().fixturesPath = '/test/spec/javascripts/
   fixtures/';
   loadFixtures("loginFixture.html");
   ```

2. Load the HTML fixtures using the `serve` directive in the JSTD-Jasmine configuration file `jsTestDriver-jasmine.conf`, as shown in the following code snippet:

   ```
   server: http://localhost:9876
   gateway:
    - {matcher: "*", server: "http://localhost:8080"}
   serve:
     - spec/javascripts/fixtures/*.html
   load:
     - lib/jasmine-1.2.0/jasmine.js
     - lib/plugins/jasmine-jquery/jquery.js
     - jstd-adapter/JasmineAdapter.js
     - lib/plugins/jasmine-jquery/jasmine-jquery.js
     - ../../../../js-src/*.js
   ```

```
    - spec/*.js
plugin:
  - name: "coverage"
    jar: coverage-1.3.4.b.jar
    module: "com.google.jstestdriver.coverage.CoverageModule"
```

After applying the preceding steps, you can now start the JSTD server as usual, using the following command:

```
java -jar JsTestDriver-1.3.4.b.jar --port 9876
```

Then, capture two browsers (for example, Firefox and IE) by entering the following URL in the browser's address bar:

```
http://localhost:9876/capture
```

Finally, you can run the Jasmine tests on top of the JSTD test runner by executing the JSTD test running command:

```
java -jar JsTestDriver-1.3.4.b.jar --config jsTestDriver-jasmine.conf
--tests all --testOutput reports
```

The output in the console will be as follows:

```
Total 26 tests (Passed: 26; Fails: 0; Errors: 0) (1538.00 ms)

Microsoft Internet Explorer 8.0 Windows: Run 13 tests (Passed: 13; Fails:
0; Errors 0) (922.00 ms)

Firefox 15.0.1 Windows: Run 13 tests (Passed: 13; Fails: 0; Errors 0)
(1538.00 ms)
```

In the `reports` folder under the `/integration/jasmine` folder, you will find 18 JUnit XML report files (nine files for the tests on Firefox and nine for the tests on IE). Every JUnit XML report file contains the test results of a single Jasmine test suite.

Integrating JSTD with QUnit

In order to integrate the QUnit tests of the weather application with JSTD, you need to do the following:

1. Download the `equiv.js` and `QUnitAdapter.js` files from the following URL:

   `https://github.com/exnor/QUnit-to-JsTestDriver-adapter`

2. Place the two downloaded files in the `jstd-adapter` folder under the `/integration/qunit` folder.

3. Create a configuration file, named `jsTestDriver-qunit.conf`, that contains the JSTD-QUnit configuration, which is as shown in the following code snippet:

   ```
   server: http://localhost:9876

   gateway:
    - {matcher: "*", server: "http://localhost:8080"}

   load:
     - lib/qunit-1.10.0.js
     - jstd-adapter/equiv.js
     - jstd-adapter/QUnitAdapter.js
     - ../../../../js-src/*.js
     - tests/*.js

   plugin:
     - name: "coverage"
       jar: coverage-1.3.4.b.jar
       module: "com.google.jstestdriver.coverage.CoverageModule"
   ```

   As you may have noticed in the preceding code snippet, you need to load the following files in order, in the `load` directive:

   - The QUnit framework file
   - The QUnit JSTD adapter files (`equiv.js` and `QUnitAdapter.js`)
   - The JavaScript source files
   - The QUnit JavaScript test files

This is basically what is needed in order to have the QUnit tests working on the top of the JSTD test runner. However, you need to take the loading of HTML fixtures into consideration. In order to load the HTML fixtures in the JSTD-QUnit tests, you can use the standard JSTD `DOC` annotation shown in the following code snippet:

```
module("LoginClient Test Module", {
 setup: function() {
   /*:DOC += <label for="username">Username  <span
   id="usernameMessage" class="error"></span></label>
   <input type="text" id="username" name="username"/>
   <label for="password">Password  <span id="passwordMessage"
   class="error"></span></label>
   <input type="password" id="password" name="password"/>*/

   //...
```

```
      }, teardown: function() {
        //...
      }
    });

    test("validating empty username", function() {
      //...
    });

    test("validating empty password", function() {
      //...
    });

    test("validating username format", function() {
      //...
    });

    test("validating password format", function() {
      //...
    });
```

After making this change in the QUnit modules, you can now run them safely on the top of the JSTD test runner.

Start the JSTD server as usual, using the following command:

```
java -jar JsTestDriver-1.3.4.b.jar --port 9876
```

Capture two browsers (for example, Firefox and IE) by entering the following URL in the browser's address bar:

```
http://localhost:9876/capture
```

Run the QUnit tests on top of the JSTD test runner, as follows:

```
java -jar JsTestDriver-1.3.4.b.jar --config jsTestDriver-qunit.conf
--tests all --testOutput reports
```

The output in the console will be as follows:

```
Total 24 tests (Passed: 24; Fails: 0; Errors: 0) (1150.00 ms)

Microsoft Internet Explorer 8.0 Windows: Run 12 tests (Passed: 12; Fails:
0; Errors 0) (826.00 ms)

Firefox 15.0.1 Windows: Run 12 tests (Passed: 12; Fails: 0; Errors 0)
(1150.00 ms)
```

In the `reports` folder of the JSTD-QUnit integration, you will find six JUnit XML files (three files for the tests on Firefox and three files for the tests on IE). Every JUnit XML report file contains the test results of a single QUnit module.

Integration with build management tools

Because the JSTD tests can run from the command line, JSTD can be integrated easily with build management tools such as Ant and Maven and also with continuous integration tools such as Hudson. The following code snippet shows an Ant script that runs the `runAllTests.bat` file in the `jstd\tests` folder.

```
<project name="weatherApplication" default="runJSTDTests" basedir=".">
  <target name="runJSTDTests">
    <exec executable="cmd">
      <arg value="/c"/>
      <arg value="runAllTests.bat"/>
    </exec>
  </target>
</project>
```

For Hudson, you can create a Hudson job that periodically executes the `runAllTests.bat` file as a Windows batch command (if you are working on a Linux environment, you can create a job that periodically executes the Linux shell script file).

As a result of running the tests from the command line, you can also integrate the Jasmine and the QUnit tests, which run on top of the JSTD runner with Ant, Maven, and Hudson.

Thanks to JSTD, we can automate the running of the Jasmine and the QUnit tests and automate the generation of the **test** and **code coverage** reports for these frameworks, which do not have a mechanism provided for integration with the command-line interface.

Integration with the IDEs

In addition to all of the powerful features of JSTD just mentioned, it can also be integrated with different integrated development environments (IDEs) such as **Eclipse** and **IntelliJ**. Thanks to this integration, you can start the JSTD server and run the tests without having to know JSTD commands. Let's see how JSTD can work with Eclipse.

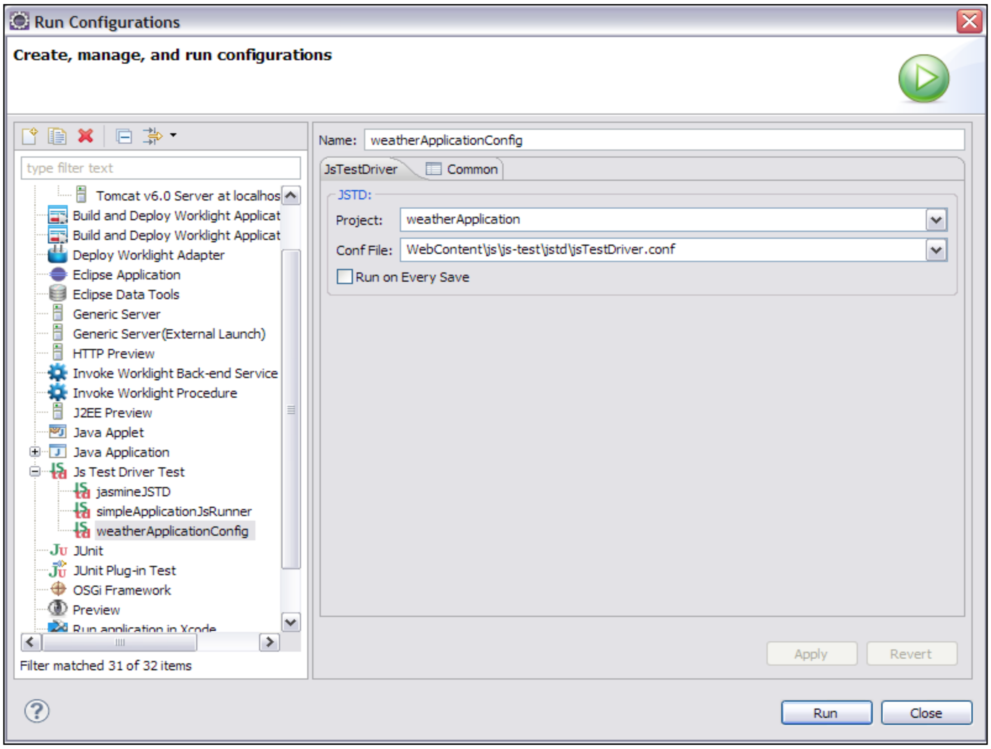
Eclipse integration

In order to work with JSTD on the Eclipse IDE, you need to:

1. Install the JSTD Eclipse plugin.
   i. In order to install the JSTD plugin in Eclipse, go to **Install new Software** in the **Help** menu.
   ii. Then, add the following installation URL as an update site:

   ```
   http://js-test-driver.googlecode.com/svn/update/
   ```

   iii. Check the **JsTestDriver Plugin for Eclipse** checkbox and click on **Next**. Finally, click on the **Next** button in the **Install details** window, accept any agreements, apply the changes, and restart Eclipse.
2. After installing the JSTD Eclipse plugin, you will need to create a JsTestDriver run configuration by selecting **Run Configurations** from the **Run** menu and then selecting the **Js Test Driver Test** item by right-clicking on it, and clicking on **New**. You will see the JSTD run configuration form as shown in the following screenshot:


Run Configurations

Create, manage, and run configurations

type filter text

- Tomcat v6.0 Server at localhos
- Build and Deploy Worklight Applicat
- Build and Deploy Worklight Applicat
- Deploy Worklight Adapter
- Eclipse Application
- Eclipse Data Tools
- Generic Server
- Generic Server(External Launch)
- HTTP Preview
- Invoke Worklight Back-end Service
- Invoke Worklight Procedure
- J2EE Preview
- Java Applet
- Java Application
- Js Test Driver Test
  - jasmineJSTD
  - simpleApplicationJsRunner
  - weatherApplicationConfig
- JUnit
- JUnit Plug-in Test
- OSGi Framework
- Preview
- Run application in Xcode

Filter matched 31 of 32 items

Name: weatherApplicationConfig

JsTestDriver | Common

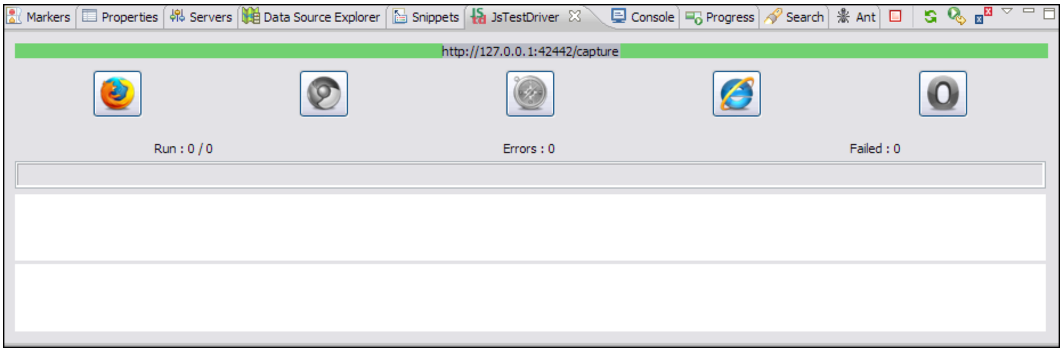
JSTD:

Project: weatherApplication

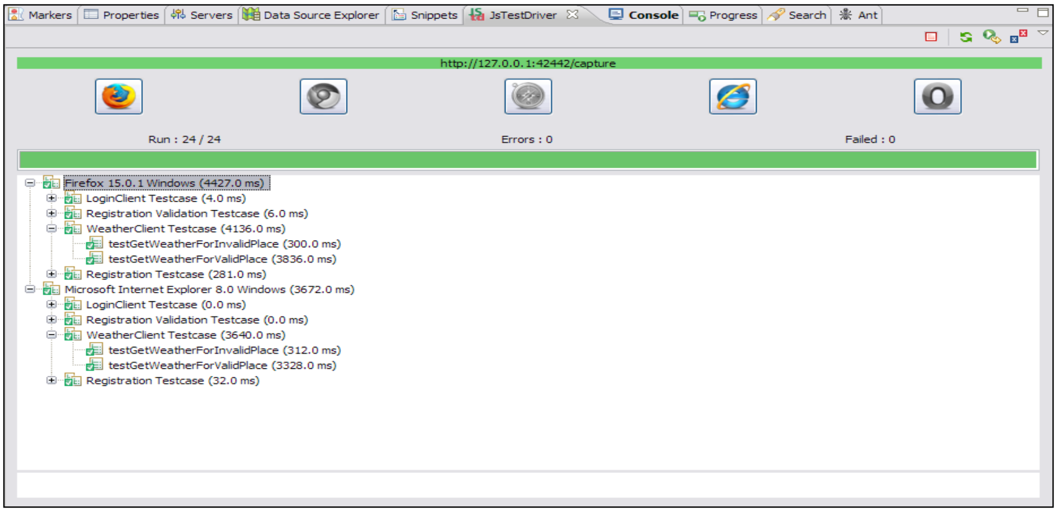
Conf File: WebContent\js\js-test\jstd\jsTestDriver.conf

☐ Run on Every Save

3. In the JSTD run configuration form, you will need to enter the name of the run configuration (`weatherApplicationConfig`), select the web project and the JSTD configuration file, and click on the **Apply** and **Close** buttons. You will then need to start the JSTD server and capture the browsers from the **server panel** as shown in the following screenshot:



4. Using the play and stop buttons in the server panel, you can start and stop the JSTD server. In order to capture one or more browsers, just copy the URL in the server panel and paste in the address bar of the browsers, and they will automatically be captured. Once the browsers are captured, they will be highlighted in the server panel, as shown in the preceding screenshot.
5. Finally, in order to execute the JSTD tests, select **Run Configurations** from the **Run** menu, select the `weatherApplicationConfig` run configuration, and click on the **Run** button. You will see the output of the JSTD test results, as shown in the following screenshot:



```
Firefox 15.0.1 Windows (4427.0 ms)
    LoginClient Testcase (4.0 ms)
    Registration Validation Testcase (6.0 ms)
    WeatherClient Testcase (4136.0 ms)
        testGetWeatherForInvalidPlace (300.0 ms)
        testGetWeatherForValidPlace (3836.0 ms)
    Registration Testcase (281.0 ms)
Microsoft Internet Explorer 8.0 Windows (3672.0 ms)
    LoginClient Testcase (0.0 ms)
    Registration Validation Testcase (0.0 ms)
    WeatherClient Testcase (3640.0 ms)
        testGetWeatherForInvalidPlace (312.0 ms)
        testGetWeatherForValidPlace (3328.0 ms)
    Registration Testcase (32.0 ms)
```

As shown in the preceding screenshot, the server panel displays the test result information, which contains the test name, the test duration, and the browser on which the test is performed.

You can apply these steps again in order to run the Jasmine and QUnit tests (on the top of JSTD) from the Eclipse IDE; the main difference is that you will need to specify the corresponding JSTD test configuration file in the JSTD run configuration form, that is, `jsTestDriver-jasmine.conf` for Jasmine and `jsTestDriver-qunit.conf` for QUnit.

Summary

In this chapter, you learned what JsTestDriver (JSTD) is, the JSTD architecture, the JSTD configuration, and how to use JSTD for testing synchronous JavaScript code. You learned how to test asynchronous (Ajax) JavaScript code using the JSTD `AsyncTestCase` object. You learned the various assertions provided by the framework and how to generate the test and code coverage reports using the framework's code coverage plugin. You also learned how to use JSTD as a test runner for other JavaScript unit testing frameworks, such as Jasmine and QUnit, in order to enable the execution of the tests of these frameworks from the command-line interface. You learned how to integrate the tests of the JSTD (and the tests of the JavaScript frameworks on the top of JSTD) with build and continuous integration tools, such as Ant and Hudson. You learned how to work with the JSTD framework in one of the most popular IDEs, Eclipse.