3 YUI Test

YUI Test is one of the most popular JavaScript unit testing frameworks. Although YUI Test is part of the **Yahoo! User Interface** (**YUI**) JavaScript library (YUI is an open source JavaScript and CSS library that can be used to build Rich Internet Applications), it can be used to test any independent JavaScript code that does not use the YUI library. YUI Test provides a simple syntax for creating JavaScript test cases that can run either from the browser or from the command line; it also 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 YUI Test syntax. In this chapter, the framework will be illustrated in detail and will be used to test the weather application that is discussed in *Chapter 1, Unit Testing JavaScript Applications*.

In YUI Test, there are different ways to display test results. You can display the test results in the browser console or develop your custom test runner pages to display the test results. It is preferable to develop custom test runner pages in order to display the test results in all the browsers because some browsers do not support the `console` object. The `console` object is supported in Firefox with Firebug installed, Safari 3+, Internet Explorer 8+, and Chrome.

Before writing your first YUI test, you need to know the structure of a custom YUI test runner page. We will create the test runner page, `BasicRunner.html`, that will be the basis for all the test runner pages used in this chapter. In order to build the test runner page, first of all you need to include the YUI JavaScript file `yui-min.js`—from the Yahoo! Content Delivery Network (CDN)—in the `BasicRunner.html` file, as follows:

```
<script src="http://yui.yahooapis.com/3.6.0/build/yui/yui-min.js"></
script>
```

At the time of this writing, the latest version of YUI Test is 3.6.0, which is the one used in this chapter. After including the YUI JavaScript file, we need to create and configure a YUI instance using the `YUI().use` API, as follows:

```
YUI().use('test', 'console', function(Y) {
  ...
});
```

The `YUI().use` API takes the list of YUI modules to be loaded. For the purpose of testing, we need the YUI `'test'` and `'console'` modules (the `'test'` module is responsible for creating the tests, while the `'console'` module is responsible for displaying the test results in a nifty console component). Then, the `YUI().use` API takes the test's callback function that is called *asynchronously* once the modules are loaded. The `Y` parameter in the callback function represents the YUI instance.

As shown in the following code snippet taken from the `BasicRunner.html` file, you can write the tests in the provided callback and then create a console component using the `Y.Console` object:

```
<HTML>
  <HEAD>
    <TITLE>YUITest Example</TITLE>
    <meta http-equiv="Content-Type" content="text/html;
    charset=utf-8">
    <script src="http://yui.yahooapis.com/3.6.0/build/yui/yui-
    min.js"></script>
  </HEAD>
  <BODY>
    <div id="log" class="yui3-skin-sam" style="margin:0px"></div>

    <script>

    // create a new YUI instance and populate it with the required
    modules.
    YUI().use('test', 'console', function(Y) {

      // Here write your test suites with the test cases
     (tests)...

      //create the console
      var console = new Y.Console({
        style: 'block',
        newestOnTop : false
      });
      console.render('#log');
```

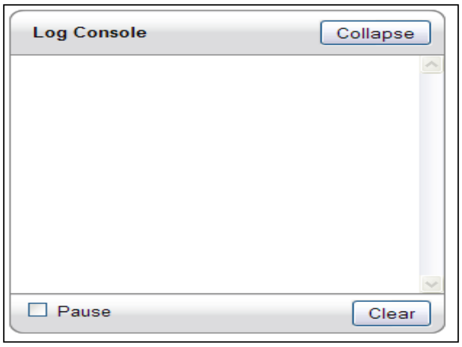
```
        // Here run the tests

      });
      </script>


    </BODY>
  </HTML>
```

The `console` object is rendered as a block element by setting the `style` attribute to `'block'`, and the results within the console can be displayed in the sequence of their executions by setting the `newestOnTop` attribute to `false`. Finally, the console component is created on the `log` div element.

Now you can run the tests, and they will be displayed automatically by the YUI console component. The following screenshot shows the `BasicRunner.html` file's console component without any developed tests:



Writing your first YUI test

The YUI test can contain test suites, test cases, and test functions. A YUI test suite is a group of related test cases. Each test case includes one or more test functions for the JavaScript code. Every test function should contain one or more assertion in order to perform the tests and verify the outputs.

The YUI `Test.Suite` object is responsible for creating a YUI test suite, while the YUI `Test.Case` object creates a YUI test case. The `add` method of the `Test.Suite` object is used for attaching the test case object to the test suite. The following code snippet shows an example of a YUI test suite:

```
  YUI().use('test', 'console', function(Y){
    var testcase1 = new Y.Test.Case({
      name: "testcase1",

      testFunction1: function() {
        //...
      },
      testFunction2: function() {
        //...
      }
    });

    var testcase2 = new Y.Test.Case({
      name: "testcase2",

      testAnotherFunction: function() {
        //...
      }
    });

    var suite = new Y.Test.Suite("Testsuite");

    suite.add(testcase1);
    suite.add(testcase2);

    //...
  });
```

As shown in the preceding code snippet, two test cases are created. The first test case is named `testcase1`; it contains two test functions, `testFunction1` and `testFunction2`. In YUI Test, you can create a test function simply by starting the function name with the word "test". The second test case is named `testcase2`; and it contains a single test function, `testAnotherFunction`. A test suite is created with the name `Testsuite`. Finally, `testcase1` and `testcase2` are added to the `Testsuite` test suite. In YUI Test, you have the option of creating a friendly test name for the test function, as follows:

```
var testCase = new Y.Test.Case({
    name: "some Testcase",
    "The test should do X": function () {
        //...
    },
    "The test should do Y": function () {
        //...
    }
});
```

The `"some Testcase"` test case contains two tests with the names `"The test should do X"` and `"The test should do Y"`.

Let's now move to testing the `SimpleMath` JavaScript object (which we tested using Jasmine in *Chapter 2, Jasmine*). 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 we did in *Chapter 2, Jasmine*, we will develop the following three test scenarios 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`) using YUI Test:

```
YUI().use('test', 'console', function(Y){
  var factorialTestcase = new Y.Test.Case({
    name: "Factorial Testcase",

    _should: {
      error: {
        testNegativeNumber: true //this test should throw an error
      }
    },

    setUp: function() {
      this.simpleMath = new SimpleMath();
    },
    tearDown: function() {
      delete this.simpleMath;
    },
    testPositiveNumber: function() {
      Y.Assert.areEqual(6, this.simpleMath.getFactorial(3));
    },
    testZero: function() {
      Y.Assert.areEqual(1, this.simpleMath.getFactorial(0));
    },
    testNegativeNumber: function() {
      this.simpleMath.getFactorial(-10);
    }
  });

  //...

});
```

The `Y.Test.Case` 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(). 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 resources by deleting the created simpleMath object.

In the first test function of the getFactorial test case, the Y.Assert.areEqual 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. We have many other assertions to use instead of Y.Assert.areEqual; we shall be discussing them in more detail in the *Assertions* section.

In the second test function of the getFactorial test case, the Y.Assert.areEqual assertion function calls simpleMath.getFactorial(0) and expects it to be equal to 1. In the last test function of the getFactorial test case, the Y.Assert.areEqual assertion function calls simpleMath.getFactorial(-10) and expects it to throw an error by using the _should.error object. In YUI Test, if you set a property whose name is the test method's name and value is true in the _should.error object, this means that this test method must throw an error in order to have the test function pass.

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:

```
var signumTestcase = new Y.Test.Case({
  name: "Signum Testcase",

  setUp: function() {
    this.simpleMath = new SimpleMath();
  },
  tearDown: function() {
    delete this.simpleMath;
  },
  testPositiveNumber: function() {
    Y.Assert.areEqual(1, this.simpleMath.signum(3));
  },
  testZero: function() {
    Y.Assert.areEqual(0, this.simpleMath.signum(0));
  },
  testNegativeNumber: function() {
    Y.Assert.areEqual(-1, this.simpleMath.signum(-1000));
  }
});
```

In the preceding code snippet, we have three tests for the `signum` method:

- The first test is about getting the signum value for a positive number (`3`)
- The second test is about getting the signum value for `0`
- The last test is about getting the signum value for a negative number (`-1000`)

The following code snippet shows the average test case:

```
var averageTestcase = new Y.Test.Case({
  name: "Average Testcase",

  setUp: function() {
    this.simpleMath = new SimpleMath();
  },
  tearDown: function() {
    delete this.simpleMath;
  },
  testAverage: function() {
    Y.Assert.areEqual(4.5, this.simpleMath.average(3, 6));
  }
});
```

In the average test case, 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`.

In the following code snippet, a test suite `"SimpleMath Test Suite"` is created in order to group the test cases `factorialTestcase`, `signumTestcase`, and `averageTestcase`. Finally, the `console` component is created to display the test results.

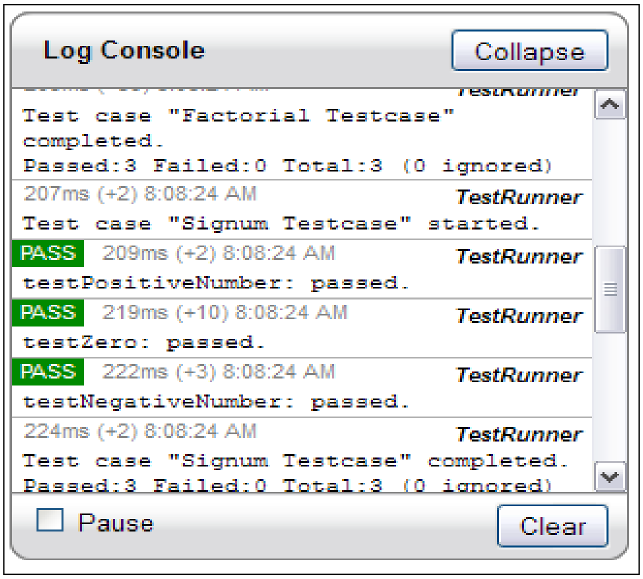
```
var suite = new Y.Test.Suite("SimpleMath Test Suite");

suite.add(factorialTestcase);
suite.add(signumTestcase);
suite.add(averageTestcase);

//create the console
var console = new Y.Console({
  style: 'block',
  newestOnTop : false
});
console.render('#resultsPanel');

Y.Test.Runner.add(suite);
Y.Test.Runner.run();
```

In order to run the test suite, we need to add it to the YUI test runner page by using the `Y.Test.Runner.add` API, and then run the YUI test runner page by using the `Y.Test.Runner.run` API. After clicking the `SimpleMath` YUI test page `SimpleMathTests.html`, you will find the test results, as shown in the following screenshot:



```
Test case "Factorial Testcase"
completed.
Passed:3 Failed:0 Total:3 (0 ignored)
207ms (+2) 8:08:24 AM                TestRunner
Test case "Signum Testcase" started.
PASS 209ms (+2) 8:08:24 AM           TestRunner
testPositiveNumber: passed.
PASS 219ms (+10) 8:08:24 AM          TestRunner
testZero: passed.
PASS 222ms (+3) 8:08:24 AM           TestRunner
testNegativeNumber: passed.
224ms (+2) 8:08:24 AM                TestRunner
Test case "Signum Testcase" completed.
Passed:3 Failed:0 Total:3 (0 ignored)
```

Finally, the following code snippet shows the complete structure of the `SimpleMath` YUI test page, which includes the `simpleMath.js` source file to be tested in the page:

```
<HTML>
  <HEAD>
    <TITLE>SimpleMathTest</TITLE>
    <meta http-equiv="Content-Type" content="text/html;
    charset=utf-8">
    <script src="http://yui.yahooapis.com/3.6.0/build/yui/yui-
    min.js"></script>
    <script src="src/simpleMath.js"></script>

  </HEAD>
  <BODY>
    <div id="resultsPanel" class="yui3-skin-sam"></div>
    <script language="javascript" type="text/javascript">
    YUI().use('test', 'console', function(Y){
      var factorialTestcase = new Y.Test.Case({
        …
      });
```

```
        var signumTestcase = new Y.Test.Case({
   …
        });

        var averageTestcase = new Y.Test.Case({
        …
        });

        var suite = new Y.Test.Suite("SimpleMath Test Suite");

        suite.add(factorialTestcase);
        suite.add(signumTestcase);
        suite.add(averageTestcase);

        //create the console and run the test suite …

      });
      </script>
    </BODY>
</HTML>
```

It is recommended that you separate the test logic from the test runner file(s), that is, have the tests in separate JavaScript files and then include them in the test runner file(s). However, the test logic is embedded in the test runner file in the `simpleMath` testing example, for simplicity. In the *Testing the weather application* section, this sort of separation will be applied.

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 that the test method passes. In the first YUI test example, we used the `Y.Assert.areEqual` assertion. In this section, the other different built-in assertions provided by YUI Test will be illustrated.

The assert assertion

The `assert` function takes two parameters. The first parameter is a condition, and the second parameter represents a failure message. It is passed if the condition is true, and when it fails, the failure message is displayed. For example:

```
Y.assert(10 == 10, "Error ..."); // will pass
Y.assert(10 != 10, "Error ..."); // will fail and display an error
```

The areEqual and areNotEqual assertions

The `areEqual` assertion function takes three parameters; the first two parameters represent the expected and actual values, and the third parameter is optional and represents a failure message. The `areEqual` function is passed if the actual is equal to the expected. If they are not equal, the test fails and the optional failure message is displayed. The `areNotEqual` function ensures that the actual and expected parameters are not equal.

It is very important to know that the `areEqual` and `areNotEqual` functions are using the JavaScript == operator to perform the comparison, that is, they do the comparison and neglect the types. For example, the following assertions will pass:

```
Y.Assert.areEqual(10, 10, "10 should equal 10...");
Y.Assert.areEqual(10, "10", "10 should equal '10'...");
Y.Assert.areNotEqual(10, 11, "10 should not equal 11...");
```

The areSame and areNotSame assertions

The `areSame` and `areNotSame` assertion functions are much similar to the `areEqual` and `areNotEqual` assertions. The main difference between them is that the `areSame` and `areNotSame` assertion functions 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 pass:

```
Y.Assert.areSame(10, 10, "10 is the same as 10...");
Y.Assert.areNotSame(10, 11, "10 is not the same as 11...");
Y.Assert.areNotSame(15, "15", "15 is not the same as '15'...");
Y.Assert.areNotSame(15, "16", "15 is not the same as '16'...");
```

The datatype assertions

The following set of assertion functions in YUI Test checks the value types. Each one of these assertion functions takes two parameters; the first parameter is the value to test and the second parameter is an optional failure message:

- `isBoolean()` is passed if the value is a Boolean
- `isString()` is passed if the value is a string
- `isNumber()` is passed if the value is a number
- `isArray()` is passed if the value is an array

- `isFunction()` is passed if the value is a function
- `isObject()` is passed if the value is an object

For example, the following assertions will pass:

```
Y.Assert.isBoolean(false);
Y.Assert.isString("some string");
Y.Assert.isNumber(1000);
Y.Assert.isArray([1, 2, 3]);
Y.Assert.isFunction(function(){ alert('test'); });
Y.Assert.isObject({somekey: 'someValue'});
```

YUI Test also provides generic assertion functions, `isTypeOf` and `isInstanceOf`, to check the datatypes.

The `isTypeOf()` method uses the JavaScript `typeof` operator in order to check the value type. It takes three parameters; the first parameter represents the value type, the second represents the value to test, and the third parameter is optional and represents a failure message. For example, the following `isTypeOf` assertions will pass:

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

In addition to all of this, you can use the `isInstanceOf` assertion, which uses the JavaScript `instanceof` operator in order to check the value instance. It takes three parameters; the first parameter represents the type constructor, the second represents the value to test, and the third parameter is optional and represents a failure message.

Special value assertions

The following set of assertion functions in YUI Test checks whether a variable value belongs to one of the special values as mentioned in the following list. Each one of these functions takes two parameters; the first parameter is the value to test, and the second parameter is an optional failure message:

- `isUndefined()` is passed if the value is undefined
- `isNotUndefined()` is passed if the value is not undefined (defined)
- `isNull()` is passed if the value is null
- `isNotNull()` is passed if the value is not null
- `isNaN()` is passed if the value is not a number (NaN)

- `isNotNaN()` is passed if the value is not NaN
- `isFalse()` is passed if the value is false
- `isTrue()` is passed if the value is true

For example, the following assertions will pass:

```
this.someStr = "some string";
Y.Assert.isUndefined(this.anyUndefinedThing);
Y.Assert.isNotUndefined(this.someStr);
Y.Assert.isNull(null);
Y.Assert.isNotNull(this.someStr);
Y.Assert.isNaN(1000 / "string_value");
Y.Assert.isNotNaN(1000);
Y.Assert.isFalse(false);
Y.Assert.isTrue(true);
```

The fail assertion

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

```
Y.Assert.isAverage = function(number1, number2, expected,
failureMessage) {
  var actual = (number1 + number2) / 2;
  if (actual != expected) {
    Y.Assert.fail(failureMessage);
  }
}
```

The `Y.Assert.isAverage` custom assertion can be called by simply using the following code:

```
Y.Assert.isAverage(3, 4, 3.5, "Average is incorrect");
```

[The `fail()` method has an optional message parameter that is displayed when the `fail()` method is called.]

Testing asynchronous (Ajax) JavaScript code

The common question that comes to mind is how to test asynchronous (Ajax) JavaScript code using YUI Test. What was mentioned earlier in this chapter so far is how to perform unit testing for synchronous JavaScript code. YUI Test provides two main APIs in order to perform real Ajax testing: `wait()` and `resume()`. Although the provided APIs of the YUI Test to perform real Ajax testing are not as rich as Jasmine (the provided YUI Test APIs do not, for example include something like spies or the Jasmine's automatic `waitsFor` mechanism), they are enough to perform a real Ajax test. Let me show you how to do this.

The wait and resume functions

The `wait()` function has two modes of operation. Its first mode pauses the execution of the test until its timeout period passes. For example:

```
this.wait(function() {
  Y.Assert.isAverage(3, 4, 3.5, "Average is incorrect");
}, 1000);
```

This code pauses the test for `1000` milliseconds, and after that its function in the first argument is executed.

The second mode of operation pauses the execution of the test until it is resumed using the `resume()` function; if it is not resumed using the `resume()` function, the test fails. Using the second mode of operation, we can perform a real Ajax testing using YUI Test, as shown in the following code snippet:

```
// Inside a test function
var this_local = this;

var successCallback = function(response) {
  this_local.resume(function() {
    // Assertions goes here to the response object...
  });
};
```

```
var failureCallback = function(response) {
  this_local.resume(function() {
    fail(); /* failure callback must not be called for successful
    scenario */
  });
};

asyncSystem.doAjaxOperation(inputData, successCallback,
failureCallback);

this.wait(5000); /* wait for 5 seconds until the resume is called
or timeout */
```

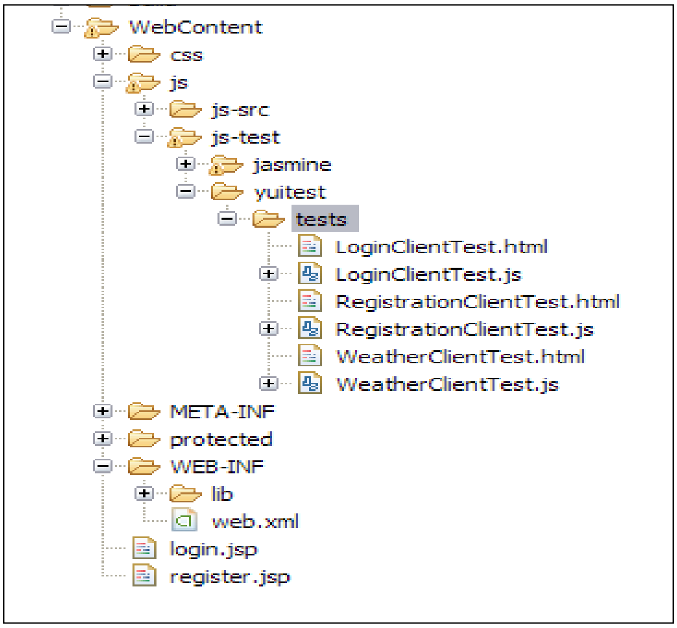
As shown in the preceding code snippet, two callbacks are created; one of them represents the successful callback (`successCallback`) that is called if the Ajax operation succeeds, and the other one represents the failure callback (`failureCallback`) that is called if the Ajax operation fails. In both `successCallback` and `failureCallback`, a call to the `resume()` API is done in order to notify the `wait()` API that the server response is returned. The `resume()` API has a single argument that represents a function that can have one or more assertions. In `successCallback`, the argument function of the `resume()` API can carry out assertions on the `response` parameter, which is returned from the server response to verify that the server returns the correct results, while in `failureCallback`, the argument function of the `resume()` API forces the test to fail because it must not be called if the operation is completed successfully.

If the Ajax response is not returned from the server after five seconds (you can set this to whatever duration you want), the `wait()` API will cause the test to fail. Although this is a manual method, as opposed to Jasmine's `waitsFor` mechanism, it is enough for real Ajax testing and will be used in order to test the Ajax part of the weather application in the next section.

Testing the weather application

Now we come to developing the YUI tests for our weather application. Actually, after you know how to write YUI tests for both synchronous and asynchronous JavaScript (Ajax) code, testing the weather application is an easy task. As you remember from the previous two 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, `yuitest` and `tests` , are created under the `js-test` folder (thus: `yuitest\tests`) to contain the YUI tests, as shown in the following screenshot:



```
WebContent
  css
  js
    js-src
    js-test
      jasmine
      yuitest
        tests
          LoginClientTest.html
          LoginClientTest.js
          RegistrationClientTest.html
          RegistrationClientTest.js
          WeatherClientTest.html
          WeatherClientTest.js
  META-INF
  protected
  WEB-INF
    lib
    web.xml
  login.jsp
  register.jsp
```

Because currently YUI Test does not have an API to load the HTML fixtures, they are included as part of the HTML test runner pages. As shown in the preceding screenshot, there are three HTML files that contain the HTML fixtures for every test—`LoginClientTest.html`, `RegistrationClientTest.html`, and `WeatherClientTest.html`. Every HTML file also includes the source and test JavaScript objects. There are three YUI test files (`LoginClientTest.js`, `RegistrationClientTest.js`, and `WeatherClientTest.js`) that test the main three JavaScript objects of the application.

Testing the LoginClient object

As what we did in *Chapter 2, Jasmine*, 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, two test cases are created; one for testing the validation of the empty fields (the username and password) and the other one for testing the validation of the fields' formats. The two test cases are grouped in a single test suite, `LoginClient Test Suite`. The following code snippet shows the validation of the empty fields' test case in the `LoginClientTest.js` file:

```
var emptyFieldsTestcase = new Y.Test.Case({
  name: "Empty userName and Password fields validation Testcase",

  setUp: function() {
    this.loginClient = new weatherapp.LoginClient();

    this.loginForm = {
      "userNameField" : "username",
      "passwordField" : "password",
      "userNameMessage" : "usernameMessage",
      "passwordMessage" : "passwordMessage"
    };
  },
  tearDown: function() {
    delete this.loginClient;
  delete this.loginForm;
  },
  testEmptyUserName: function() {
    document.getElementById("username").value = ""; /* setting
    username to empty */
    document.getElementById("password").value = "Admin@123";

    this.loginClient.validateLoginForm(this.loginForm);

    Y.Assert.areEqual("(field is required)",
    document.getElementById("usernameMessage").innerHTML);
  },
  testEmptyPassword: function() {
    document.getElementById("username").value =
    "someone@yahoo.com";
    document.getElementById("password").value = "";   /* setting
    password to empty */

    this.loginClient.validateLoginForm(this.loginForm);

    Y.Assert.areEqual("(field is required)",
    document.getElementById("passwordMessage").innerHTML);
    }
});
```

In the preceding code snippet, 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.

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. Then it checks whether the validateLoginForm API produces the "(field is required)" message in the usernameMessage field by using the Y.Assert.areEqual assertion.

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

The following code snippet shows the second test case, which validates the formats of the fields (username and password) in the LoginClientTest.js file:

```
var fieldsFormatTestcase = new Y.Test.Case({
  name: "Fields format validation Testcase",

  setUp: function() {
    this.loginClient = new weatherapp.LoginClient();

    this.loginForm = {
      "userNameField" : "username",
      "passwordField" : "password",
      "userNameMessage" : "usernameMessage",
      "passwordMessage" : "passwordMessage"
    };
  },
  tearDown: function() {
    delete this.loginClient;
    delete this.loginForm;
  },
  testUsernameFormat: function() {
    document.getElementById("username").value =
    "someone@someDomain";/* setting username to invalid format */
    document.getElementById("password").value = "Admin@123";

    this.loginClient.validateLoginForm(this.loginForm);

    Y.Assert.areEqual("(format is invalid)",
    document.getElementById("usernameMessage").innerHTML);
  },
  testPasswordFormat: function() {
    document.getElementById("username").value =
    "someone@someDomain.com";
    document.getElementById("password").value = "Admin123"; /*
    setting password to invalid format */

    this.loginClient.validateLoginForm(this.loginForm);

    Y.Assert.areEqual("(format is invalid)", document.getElementById("
passwordMessage").innerHTML);
  }
});
```

In the preceding code snippet, `testUsernameFormat` tests the validation of the username format. It tests whether the `LoginClient` object should be 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 whether the `validateLoginForm` API produces the `"(format is invalid)"` message in the `usernameMessage` field by using the `Y.Assert.areEqual` assertion.

`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. It finally checks whether the `validateLoginForm` API produces the `"(format is invalid)"` message in the `passwordMessage` field.

`emptyFieldsTestcase` and `fieldsFormatTestcase` are added to the `LoginClient` test suite, the YUI console is created, the test suite is run, and the test results are displayed in the console component, as shown in the following code snippet from the `LoginClientTest.js` file:

```
  var suite = new Y.Test.Suite("LoginClient Test Suite");

  suite.add(emptyFieldsTestcase);
  suite.add(fieldsFormatTestcase);

  //create the console
  var console = new Y.Console({
    style: 'block',
    newestOnTop : false
  });
  console.render('#resultsPanel');

  Y.Test.Runner.add(suite);
  Y.Test.Runner.run();
```

Finally, the following code snippet shows the HTML fixture of the `LoginClient` test suite in the `LoginClientTest.html` file. It includes the username and password fields, the YUI console `div` element, and both the source JavaScript object file (`LoginClient.js`) and the test JavaScript object file (`LoginClientTest.js`).

```
    <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"/>

    <div id="resultsPanel" class="yui3-skin-sam"></div>
  …

  <script type="text/javascript" src="../../../js-src/LoginClient.js"></
  script>
  <script type="text/javascript" src="LoginClientTest.js"></script>
```

Testing the RegistrationClient object

In the `RegistrationClient` object, we will verify 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

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

The `registerUser` test case covers the following test scenarios:

- Testing the adding of a new user through the `testAddNewUser` test function. The registration client object should be able to register a valid user correctly.
- Testing the adding of a user with an existing user ID through the `testAddExistingUser` test function. In this case, the registration client object should fail when registering a user whose ID is already registered.

The following code snippet shows the `testAddNewUser` test function of the `registerUser` test case in the `RegistrationClientTest.js` file. 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.

```
var registerUserTestcase = new Y.Test.Case({
  name: "RegisterUser Testcase",

  setUp: function() {
    this.registrationClient = new weatherapp.RegistrationClient();

    this.registrationForm = {
        "userNameField" : "username",
        "passwordField1" : "password1",
        "passwordField2" : "password2",
        "userNameMessage" : "usernameMessage",
        "passwordMessage1" : "passwordMessage1"
    };
  },
  tearDown: function() {
    delete this.registrationClient;
    delete this.registrationForm;
  },
  testAddNewUser: function() {
    this.userName = "hazems" + new Date().getTime() +
    "@apache.org";

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

    var this_local = this;
    var Y_local = Y;

    var successCallback = function(response) {
      var resultMessage = response.xmlhttp.responseText;
      this_local.resume(function() {
        Y_local.Assert.areEqual("User is registered successfully
        ...", resultMessage);
      });
    };

    var failureCallback = function() {
      this_local.resume(function() {
        fail();
      });
    };

    this.registrationClient.registerUser(this.registrationForm,
    successCallback, failureCallback);

    this.wait(5000); /* wait for 5 seconds until the resume is
    called or timeout */
  }
  ...
});
```

In the `testAddNewUser` test function, the registration form is filled with a valid generated username and valid matched passwords, and then two callbacks are created. The first callback is the success callback, while the second one is the failure callback. `registrationClient.registerUser` is called with the registration form, the success callback, and the failure callback parameters. `this.wait(5000)` waits for a call from the `resume()` API, or it fails after 5000 milliseconds.

In the success callback, the `resume()` API is called, and the `resume()` function parameter ensures that the returned response message from the server is `User is registered successfully ...` using the `areEqual` assertion.

In the failure callback, the `resume()` API is also called, and the `resume()` function parameter ensures that the test fails by using the `fail()` API because the failure callback must not be called for a valid user registration.

The YUI Test Ajax testing of the weather application is *real* Ajax testing; this requires the server to be up and running in order to perform the test correctly.

The following code snippet shows the `testAddExistingUser` test function of the `registerUser` test case in the `RegistrationClientTest.js` file:

```
  var registerUserTestcase = new Y.Test.Case({
    ...
      testAddExistingUser: function() {
      document.getElementById("username").value = this.userName;
      document.getElementById("password1").value = "Admin@123";
      document.getElementById("password2").value = "Admin@123";

      var this_local = this;
      var Y_local = Y;
```

```
        var successCallback = function() {
          this_local.resume(function() {
            fail();
          });
        };

        var failureCallback = function(response) {
          var resultMessage = response.xmlhttp.responseText;
          this_local.resume(function() {
            Y_local.Assert.areEqual("A user with the same username is
            already registered ...", resultMessage,);
          });
        };

        this.registrationClient.registerUser(this.registrationForm,
        successCallback, failureCallback);

        this.wait(5000); /* wait for 5 seconds until the resume is called
        or timeout */
      }
    });
```

In the `testAddExistingUser` test function, the registration form is filled with the same username that is already registered in the `testAddNewUser` test function, and then two callbacks are created. The first callback is the success callback while the second one is the failure callback. `registrationClient.registerUser` is called with the registration form, the success callback, and the failure callback parameters. The `this.wait(5000)` waits for a call to the `resume()` API or it fails after `5000` milliseconds.

In the success callback, the `resume()` API is called, and the `resume()` function parameter ensures that the test fails by using the `fail()` API because the success callback must not be called when registering a user whose ID is already registered.

In the failure callback, the `resume()` API is also called, and the `resume()` function parameter ensures that the returned failure response message from the server is `A user with the same username is already registered ...` using the `areEqual` assertion.

Finally, the following code snippet shows the HTML fixture of the `RegistrationClient` test suite in the `RegistrationClientTest.html` file. It includes the username and password fields, the YUI console `div` element, and both the JavaScript source files, `RegistrationClient.js` and `LoginClient.js`, because `RegistrationClient.js` depends on `LoginClient.js` and on the JavaScript test file `RegistrationClientTest.js`.

```
...
<label for="username">Username (Email)  <span id="usernameMessage"
class="error"></span></label>
<input type="text" id="username" name="username"/><br/>

<label for="password1">Password  <span id="passwordMessage1"
class="error"></span></label>
<input type="password" id="password1" name="password1"/><br/>

<label for="password2">Confirm your password</label>
<input type="password" id="password2" name="password2"/><br/>

<div id="resultsPanel" class="yui3-skin-sam"></div>
...

<script type="text/javascript" src="../../../js-src/LoginClient.js"></
script>
<script type="text/javascript" src="../../../js-src/
RegistrationClient.js"></script>

<script type="text/javascript" src="RegistrationClientTest.js"></
script>
```

Testing the WeatherClient object

In the `WeatherClient` object, we will 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 for the `WeatherClientTest.js` and `WeatherClientTest.html` files from the `Chapter 3` folder in the code bundle available on the book's website.

To get the test source code, all that you need to do is to unzip the `weatherApplication.zip` file, and you will be able to find all the YUI tests in the `tests` folder in `weatherApplication\WebContent\js\js-test\yuitest`.

Running the weather application tests

In order to run the weather application tests correctly, you have to make sure that the server is up and running or the application will not pass the Ajax tests. So, you need to deploy this chapter's updated version of the weather application on Tomcat 6 as explained in *Chapter 1, Unit Testing JavaScript Applications*, and then type the three following URLs in the browser's address bar:

- `http://localhost:8080/weatherApplication/js/js-test/yuitest/tests/LoginClientTest.html`
- `http://localhost:8080/weatherApplication/js/js-test/yuitest/tests/RegistrationClientTest.html`
- `http://localhost:8080/weatherApplication/js/js-test/yuitest/tests/WeatherClientTest.html`

Don't worry; you do not need to do this every time you run the YUI Test pages. Check the *Integration with build management tools* section to learn how to automate the running of the YUI Test pages.

Generating test reports

In the *Integration with build management tools* section, YUI Test Selenium Driver is used to generate JUnit XML reports automatically without using the YUI Test reporting APIs. You may jump to that section if you are not interested in digging into the YUI Test reporting APIs.

YUI Test has a powerful feature, test reporting. Once the test completes its execution and the test result's object is retrieved, you can post the test results to the server (Java servlet, PHP, or another server-side object) to generate the report. First of all, let's see how to retrieve the test result's object.

In order to retrieve the test result's object, you need to use the `Y.Test.Runner.getResults()` API. Unfortunately, The `Y.Test.Runner.getResults()` API expects you to call it when the test is completed; in other words, it does not wait for the tests to complete its executions. If you call the `Y.Test.Runner.getResults()` API and the tests are still running, the API will return `null`.

However, to make sure that the test is completed, you have one of two options:

- The first is to use the `isRunning()` API in the `TestRunner` interface, which returns true if the test is still running and false if it finishes its execution. The following code snippet shows you how to call the `Y.Test.Runner.getResults()` API properly and ensure, using the `isRunning()` API, that it will not be called while the test is running:

  ```
  var intervalID = window.setInterval(function() {
    if (! Y.Test.Runner.isRunning()) {
      var results = Y.Test.Runner.getResults();

      // Do whatever you want with the results

      window.clearInterval(intervalID);
    }
  }, 1000);
  ```

  The code is simple; `window.setInterval` calls the `Y.Test.Runner.isRunning()` API every `1000` milliseconds and waits until `Y.Test.Runner.isRunning()` returns false. When `Y.Test.Runner.isRunning()` returns false, the `Y.Test.Runner.getResults()` API can be called safely, and then the execution of `window.setInterval` is stopped by calling `window.clearInterval(intervalID)`.

- The second option, which is the recommended one, is to subscribe in the YUI test runner complete event (`Y.Test.Runner.COMPLETE_EVENT`), as shown in the following code snippet:

  ```
  function processTestResults() {
    var results = Y.Test.Runner.getResults();

    // Do whatever you want with the results
  }

  Y.Test.Runner.subscribe(Y.Test.Runner.COMPLETE_EVENT,
  processTestResults);
  ```

  You can use the YUI test runner's `subscribe()` API in order to subscribe in the test runner's complete event. `processTestResults` is the event handler that is called once the event is completed. In the `processTestResults` event handler, it is safe to call the `Y.Test.Runner.getResults()` API to get the test results.

In YUI Test, there are many types of events that can be subscribed to. There are events on the level of the test, test case, test suite, and the test runner. The preceding code snippet is an example of an event on the test runner level. To get a complete reference for all the YUI Test events, check the following URL:

`http://yuilibrary.com/yui/docs/api/classes/Test.Runner.html#Events`

After getting the test results, let's learn how to post the results on the server to generate the report. The following code snippet shows how to send the test results data in JUnit XML format to the server. This code is part of the `RegistrationClientTest.js` file:

```
Y.Test.Runner.add(suite);

function processTestResults() {
  var results = Y.Test.Runner.getResults();
  var reporter = new
  Y.Test.Reporter("/weatherApplication/YUIReportViewer",
  Y.Test.Format.JUnitXML);

  // Some parameters to be sent

  reporter.report(results);
}

Y.Test.Runner.subscribe(Y.Test.Runner.COMPLETE_EVENT,
processTestResults);
Y.Test.Runner.run();
```

After getting the test result's object, you create a `Y.Test.Reporter` object, which can be constructed using the two following parameters:

- The server URL to post the test result's data to. Note that the POST data operation is performed silently by the `Y.Test.Reporter` object and does not cause the test page to navigate away because it does not get back any response from the server. In our example, the server URL is `/weatherApplication/YUIReportViewer`, which maps to a simple Java servlet that receives the posted test results data and saves the data in a file inside a local directory.

- The report format. The four following formats are allowed for the posting of test results:
    - `Y.Test.Format.XML`: To post the test results data in XML format.
    - `Y.Test.Format.JSON`: To post the test results data in JSON format.
    - `Y.Test.Format.JUnitXML`: To post the test results data in JUnit XML format.
    - `Y.Test.Format.TAP`: To post the test results in TAP format. TAP stands for **Test Anything Protocol**. For more information about this format, check the following URL:

      `http://testanything.org/wiki/index.php/Main_Page`

In order to post the test result data to the server, you need to call the `report()` API of the `Y.Test.Reporter` object with the test result data (`results`). By default, the following parameters are posted to the server when the `report()` API is called:

- `results`: The serialized test results data object
- `useragent`: The user-agent string that represents the browser
- `timestamp`: The date and time at which the report was sent

You have the ability to post extra parameters by using the `addField()` API, as shown in the following code snippet:

```
reporter.addField("param1", "value1");
reporter.addField("param2", "value2");
```

In order to make the report name and the report file extension configurable, the `addField()` API can be used in order to send this information to the `YUIReportViewer` custom servlet, as shown in the following code snippet:

```
Y.Test.Runner.add(suite);

function processTestResults() {
  var results = Y.Test.Runner.getResults();
  var reporter = new
  Y.Test.Reporter("/weatherApplication/YUIReportViewer",
  Y.Test.Format.JUnitXML);

  // Send a custom parameter to tell the Servlet the report
  name and extension.
  reporter.addField("reportName", "registrationTestReport");

  reporter.addField("format", "xml");
```

```
      reporter.report(results);
    }

    Y.Test.Runner.subscribe(Y.Test.Runner.COMPLETE_EVENT,
    processTestResults);
    Y.Test.Runner.run();
```

The custom `YUIReportViewer` servlet generates the report file with the `[reportName].[format]` name under the `yuitest\reports` folder. The report file contains the `results` content. The custom servlet code is included for your reference in the following code snippet; as you can see, it is very simple code that can be implemented easily in any other server-side technology such as PHP and ASP.NET.

```
public class YUIReportViewer extends HttpServlet {

  protected void doPost(HttpServletRequest request,
  HttpServletResponse response) throws ServletException,
  IOException {
    String results = request.getParameter("results");
    String format = (request.getParameter("format") == null) ?
    "xml" : request.getParameter("format");
    String reportName = (request.getParameter("reportName") ==
    null) ? "report" : request.getParameter("reportName");

  // Generate the report result file under the reports folder...
    BufferedWriter out = null;

    String reportFullPath =
    getServletContext().getRealPath("/js/js-test/yuitest/reports")
    + "/" + reportName + "." + format;

    try {
      FileWriter fstream = new FileWriter(reportFullPath);

      out = new BufferedWriter(fstream);
      out.write(results);
    } catch (Exception e) {
      e.printStackTrace();
    } finally {
      out.close();
    }
  }
}
```

As indicated before, this book does not teach you any server-side technology (it is outside the scope of the book); however, it is good to mention the custom `YUIReportViewer` servlet code in this example in order to show you what the server-side code will look like in the case of generating a report.

After running the `RegistrationClient` test suite by browsing to `http://localhost:8080/weatherApplication/js/js-test/yuitest/tests/RegistrationClientTest.html`, the `registrationTestReport.xml` file can be accessed via `http://localhost:8080/weatherApplication/js/js-test/yuitest/reports/registrationTestReport.xml`. The following code snippet shows the `RegistrationClient` test report in JUnit XML format:

```
<testsuites>
  <testsuite name="Empty userName and Password fields Testcase"
  tests="2" failures="0" time="0.039">
    <testcase name="testEmptyUserName" time="0.003"/>
    <testcase name="testEmptyPassword" time="0.008"/>
  </testsuite>
…
  <testsuite name="RegisterUser Testcase" tests="2" failures="0"
  time="0.17">
    <testcase name="testAddNewUser" time="0.048"/>
    <testcase name="testAddExistingUser" time="0.062"/>
  </testsuite>
</testsuites>
```

You can produce a JSON report instead; change the format parameters of `Y.Test.Reporter`, as shown in the highlighted part of the following code snippet:

```
function processTestResults() {
  var results = Y.Test.Runner.getResults();
  var reporter = new
  Y.Test.Reporter("/weatherApplication/YUIReportViewer",
  Y.Test.Format.JSON);

  // Send a custom parameter to tell the Servlet the report
  name and extension.
  reporter.addField("reportName", "registrationTestReport");
  reporter.addField("format", "json");

  reporter.report(results);
}
```

After running the `RegistrationClient` test suite, `registrationTestReport.json` can be accessed via the following location:

```
http://localhost:8080/weatherApplication/js/js-test/yuitest/reports/
registrationTestReport.json
```

> You can follow the same procedure to generate YUI Test reports with different formats. All you need to do is to change the format parameter of the `Y.Test.Reporter` object, as shown in the previous examples.

Automation and integration with build management tools

It can be difficult to run every test page individually in order to check the results, so for example, if we have 100 YUI test pages, it means that we have to type 100 URLs in the browser's address bar, which is a very inefficient way of performing the tests. Fortunately, we can automate the running of the YUI test pages using **Selenium** (an automation web application testing tool) integration with YUI Test. This sort of integration can be done by the YUI Test Selenium Driver utility. Let's see how to work it.

Configuring YUI Test Selenium Driver

In order to configure the YUI Test Selenium Driver utility with the YUI tests, you need to do the following:

1. Make sure that you have installed Java 5 (v1.5 or later) on your operating system.
2. Download the following:
    - The Selenium Server Version 2.25.0; it can be found at `http://seleniumhq.org/download/`.
    - The Selenium Java Client Driver; it can be found at `https://github.com/yui/yuitest/blob/master/java/lib/selenium-java-client-driver.jar`.
    - The YUI Test Selenium Driver, which can be found at `https://github.com/yui/yuitest/blob/master/java/build/yuitest-selenium-driver.jar.`.

3. Start the Selenium Server from the command line using `java -jar selenium-server-standalone-2.25.0.jar`.
4. Place the Selenium Java Client Driver (`selenium-java-client-driver.jar`) in `/lib/ext/`, in your JRE directory.
5. After following the preceding steps, YUI Test Selenium Driver (`yuitest-selenium-driver.jar`) is ready to execute the YUI tests.

Let's see how we will use the driver to automate the running of the weather application YUI tests.

Using YUI Test Selenium Driver in the weather application

YUI Test Selenium Driver works by communicating with the Selenium Server and specifying on which browsers the YUI test pages are to be loaded. The server then loads the test pages, and the JavaScript tests are executed in the specified browsers; once the tests are complete, the results are retrieved and then outputted into JUnit XML report files automatically (this is the default report format and it can be changed to XML or TAP formats from the driver configuration file).

In the weather application project, a `cli` folder is created under the `yuitest` folder to include the `yuitest-selenium-driver.jar` file and the command-line batch file that automates the running of the test pages (in case you are working in a Unix environment, you can create an equivalent `.sh` file). The following command shows how to automate running of the weather application test pages in the `runTests.bat` file (don't forget to make sure that Selenium Server is running before executing this command):

```
java -jar yuitest-selenium-driver.jar --browsers *firefox,*iexplore
--tests tests.xml --resultsdir %~dp0gen_reports
```

This command executes the `yuitest-selenium-driver.jar` file with the following parameters:

- `--browsers`: This parameter specifies which browsers will be used in the tests; in our case, Firefox and Internet Explorer are used.
- `--tests`: This parameter specifies the XML file that includes the YUI test pages. The content of this file is shown in the next code snippet.

- `--resultsdir`: This parameter specifies the location of the output report files. In our case, the output report files are generated in the `gen_reports` folder under the `cli` folder, which contains the batch file.

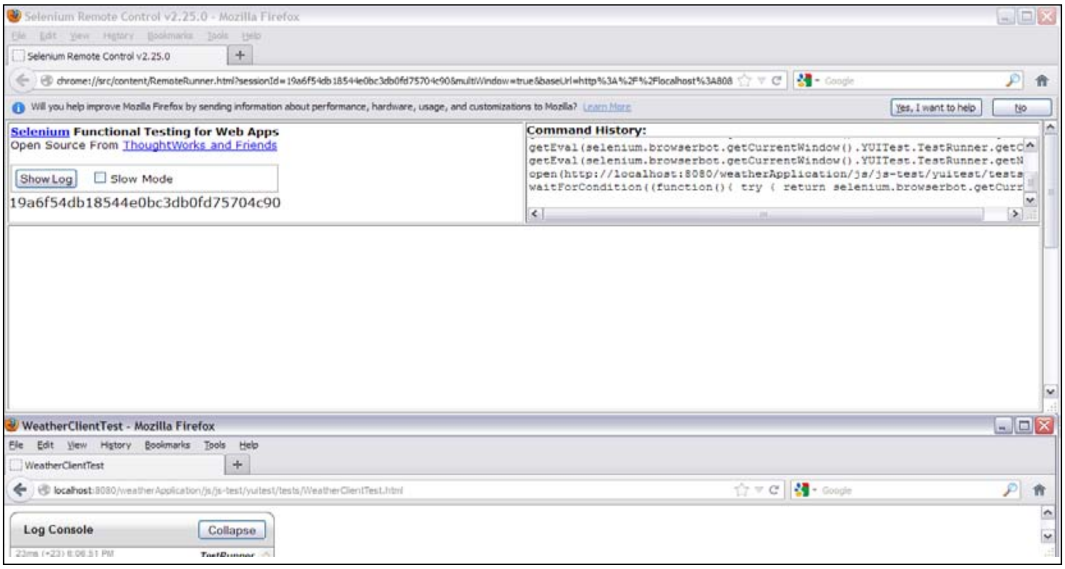
Let's see the content of the `tests.xml` file, which includes the weather application test pages, in the following code snippet:

```
<?xml version="1.0"?>
<yuitest>
    <tests base="http://localhost:8080/weatherApplication/js/js-test/
     yuitest/tests/" timeout="30000">
        <url>LoginClientTest.html</url>
        <url>RegistrationClientTest.html</url>
        <url>WeatherClientTest.html</url>
    </tests>
</yuitest>
```

The `tests.xml` file contains mainly three elements:

- The `<yuitest>` element, which represents the root element.
- The `<tests>` element, which includes the `<url>` tags of the different test pages. It has a `base` attribute that is used to specify the base location of all of the children `<url>` tags. In our case, this is `http://localhost:8080/weatherApplication/js/js-test/yuitest/tests/`. The `<tests>` element also has a `timeout` attribute that specifies the maximum number of milliseconds the driver will wait for the test to complete; after this period, an error will be thrown for the test. In our case, `30` seconds is specified.
- The `<url>` element, which contains the relative paths of the pages under the base URL specified in the `<tests>` parent element.

While running the command, you will find the application tests are executed in the Internet Explorer and Firefox browsers, as shown in the following screenshot:


Selenium Remote Control v2.25.0 - Mozilla Firefox
chrome://src/content/RemoteRunner.html?sessionId=19a6f54db18544e0bc3db0fd75704c90&multiWindow=true&baseUrl=http%3A%2F%2Flocalhost%3A808
Will you help improve Mozilla Firefox by sending information about performance, hardware, usage, and customizations to Mozilla? Learn More
Yes, I want to help
No
Selenium Functional Testing for Web Apps
Open Source From ThoughtWorks and Friends
Show Log
Slow Mode
WeatherClientTest - Mozilla Firefox
localhost:8080/weatherApplication/js/js-test/yuitest/tests/WeatherClientTest.html
Log Console
Collapse

19a6f54db18544e0bc3db0fd75704c90

Command History:

```
getEval(selenium.browserbot.getCurrentWindow().YUITest.TestRunner.getC
getEval(selenium.browserbot.getCurrentWindow().YUITest.TestRunner.getN
open(http://localhost:8080/weatherApplication/js/js-test/yuitest/tests
waitForCondition((function(){ try { return selenium.browserbot.getCurr
```

Once the tests are complete, the browsers will close automatically, and six JUnit XML test reports will be generated in the `gen_reports` folder—three reports for the three weather application test pages' execution results in Firefox and the other three reports for the execution results in IE.

Now you know how to use the driver in order to automate YUI tests. There are other parameters and features that are supported by the YUI Test Selenium Driver. You may check all of them in the driver documentation page in GitHub:

`https://github.com/yui/yuitest/wiki/Selenium-driver`

Integration with build management tools

Because the YUI Test Selenium Driver can run from the command line, it 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 (`ant.xml`) that runs the `runTests.bat` file:

```
<project name="weatherApplication" default="runYUITests" basedir=".">
 <target name="runYUITests">
   <exec executable="cmd">
```

```
        <arg value="/c"/>
        <arg value="runTests.bat"/>
      </exec>
    </target>
  </project>
```

For Hudson, you can create a **Hudson job** that periodically executes the `runTests.bat` file as a Windows batch command. Hudson is a continuous integration tool that provides an easy way for the software team to integrate the code changes to the software project. It allows the software team to produce up-to-date builds from the system easily through the automated continuous build (it can be done many times per day). More information about Hudson can be found at `http://hudson-ci.org/`.

Summary

In this chapter, you learned what YUI Test is and how to use the JavaScript unit testing framework to test synchronous JavaScript code. You also got to know how to test asynchronous (Ajax) JavaScript code by using the YUI Test `wait` and `resume` mechanism. You learned the various assertions provided by the framework, how to get the XML and JSON test reports using the framework reporter APIs, and how to generate the test reports automatically by using the YUI Test Selenium Driver. You also learned how to automate the YUI tests using the YUI Test Selenium Driver and how to integrate the automation script with Ant as an example of the build management tools. Along the way, we applied all of these concepts to test our weather application. In the next chapter, you will learn how to work with the QUnit framework and how to use it to test the weather application.