

4

QUnit

QUnit is a popular JavaScript unit testing framework. Although QUnit is used and maintained by jQuery, it can be used for testing any independent JavaScript code. QUnit provides a simple syntax for creating JavaScript test modules and functions that can be run from the browser. QUnit provides a clean mechanism for testing asynchronous (Ajax) JavaScript code. In this chapter, the QUnit framework will be illustrated in detail and used for testing the weather application that was discussed in *Chapter 1, Unit Testing JavaScript Applications*.

Configuration

In order to configure QUnit, the first step is to download the two framework files:

- The **QUnit JS** file found at <http://code.jquery.com/qunit/qunit-1.10.0.js>
- The **QUnit CSS** file found at <http://code.jquery.com/qunit/qunit-1.10.0.css>

After downloading the two files, put them in the same folder. (Let's call this folder `lib`.) At the time of this writing, the latest release of QUnit is the v1.10.0, which will be used in this book.


Now, let's prepare the tests' runner page of the QUnit test runner page. The following code snippet shows the `BasicRunner.html` page that contains the basic skeleton of the QUnit test runner page:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>QUnit test runner</title>
  <link rel="stylesheet" href="lib/qunit-1.10.0.css">
```

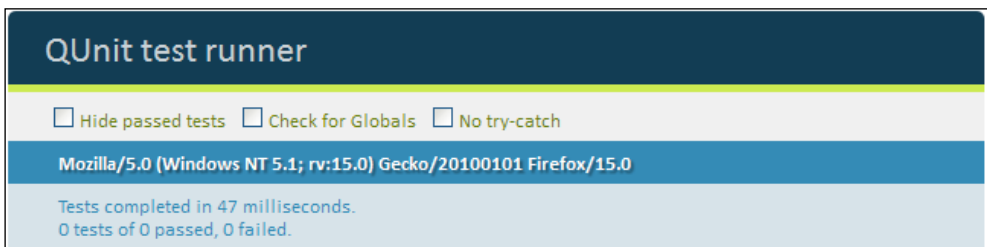
```
</head>
<body>
  <div id="qunit"></div>
  <div id="qunit-fixture"></div>
  <script src="lib/qunit-1.10.0.js"></script>

  ...The test code here...
</body>
</html>
```

The `BasicRunner.html` page includes the framework files from the `lib` folder. As you will notice, there are two `div` elements in the test runner page. The first `qunit` `div` element is used for displaying the QUnit test results while the second `qunit-fixture` `div` element is used for holding the QUnit HTML fixtures needed for the tests.

 QUnit cleans up the `qunit-fixture` `div` element before every test run so you do not have to do this clean up manually.

Now you can run the test runner page that does not include any of the QUnit tests yet. The following screenshot shows the `BasicRunner.html` page, which does not include any tests:



Writing your first QUnit test

A QUnit test can contain test modules and test functions. A QUnit test module is a group of related test functions. Every test function should contain one or more assertion(s) in order to perform the test and verify the outputs.

The `QUnit.module` function is responsible for creating the QUnit module and the `QUnit.test` function is responsible for creating the QUnit test. In order to add the test function to the module, just place the test function under the declared module, as shown in the following code snippet:

```

module("testing Module", {
  setup: function() {
    // setup code goes here...
  }, teardown: function() {
    // teardown code goes here...
  }
});

test("testing function1", function() {
  // assertions go here...
});

test("testing function2", function() {
  // assertions go here...
});

```

As shown in the preceding code snippet, a test module with the name "testing Module" is created. The test module can contain a `setup` method that is called to perform the initialization of every test function before its execution. The test module can also contain a `teardown` method that is called after the execution of every test function, for de-initializing the test. The test module contains two test functions. The first test function is named "testing function1" while the second test function is named "testing function2". Every test function can contain one or more assertions.



In QUnit, you have the option to create the test functions without including them in modules. However, it is preferred to include tests in modules to organize them. Grouping the tests in modules gives you the ability to run every module independently.

Let's move to testing the `SimpleMath` JavaScript object (which we tested using the Jasmine and YUI Test frameworks 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;
}
```

In order to organize the SimpleMath QUnit tests, three modules are created for testing the `getFactorial`, `signum`, and `average` APIs of the SimpleMath object.

As we did in the previous chapters, we will develop the following three test scenarios for the `getFactorial` method:

- Positive number
- Zero
- Negative number

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

```
module("Factorial", {
    setup: function() {
        this.simpleMath = new SimpleMath();
    }, teardown: function() {
        delete this.simpleMath;
    }
});

test("calculating factorial for a positive number", function() {
    equal(this.simpleMath.getFactorial(3), 6, "Factorial of three
```

```

    must equal six");
  });

  test("calculating factorial for zero", function() {
    equal(this.simpleMath.getFactorial(0), 1, "Factorial of zero
    must equal one");
  });

  test("throwing an error when calculating the factorial for a negative
  number", function() {
    raises(function() {
      this.simpleMath.getFactorial(-10)
    }, "There is no factorial for negative numbers");
  });

```

The module function declares a new module called `Factorial`. In the `setup` method, the `simpleMath` object is created using `new SimpleMath()`. `tearDown` is used to clean up by deleting the created `simpleMath` object.

In the first test function of the `Factorial` module, the QUnit `equal` assertion function calls `simpleMath.getFactorial(3)` and expects the result to be equal to 6. If `simpleMath.getFactorial(3)` returns a value other than 6, then the test function fails. The last parameter of the QUnit `equal` assertion is an optional one, and it represents the message to be displayed with the test.

In the second test function of the `Factorial` module, the `equal` assertion function calls `simpleMath.getFactorial(0)` and expects it to be equal to 1. In the last test function of the `Factorial` module, the test function calls `simpleMath.getFactorial(-10)` and expects it to throw an error using the `raises` assertion.

The `raises` assertion takes two parameters; the first one is the function parameter that includes the call to the API to test, and the second one is an optional one and represents the message that is to be displayed with the test. The `raises` assertion succeeds if the API that is to be tested throws an error.

QUnit has other assertions to use instead of the `equal` and `raises` assertions; we will discuss them in more detail later in this chapter in the *Assertions* section.

After finalizing the `Factorial` module, we come to the new module that tests the functionality of the `signum` API provided by the `SimpleMath` object. The following code snippet shows the `Signum` module:

```

module("Signum", {
  setup: function() {
    this.simpleMath = new SimpleMath();
  }, tearDown: function() {

```

```
        delete this.simpleMath;
    }
});

test("calculating signum for a positive number", function() {
    equal(this.simpleMath.signum(3), 1, "Signum of three must equal one");
});

test("calculating signum for zero", function() {
    equal(this.simpleMath.signum(0), 0, "Signum of zero must equal zero");
});

test("calculating signum for a negative number", function() {
    equal(this.simpleMath.signum(-1000), -1, "Signum of -1000 must equal -1");
});
```

We have three test functions in the `Signum` module; the first test function tests the signum of a positive number, the second test function tests the signum of zero, and the last test function tests the signum of a negative number. The following code snippet shows the `Average` module:

```
module("Average", {
    setup: function() {
        this.simpleMath = new SimpleMath();
    }, teardown: function() {
        delete this.simpleMath;
    }
});

test("calculating the average of two numbers", function() {
    equal(this.simpleMath.average(3, 6), 4.5, "Average of 3 and 6 must equal 4.5");
});
```

In the `Average` module, the "calculating the average of two numbers" test function ensures that the average is calculated correctly by calling the `average` API using the two parameters 3 and 6, and expecting the result to be 4.5 using the `equal` assertion.

A very important thing that you should know is that QUnit does not guarantee the order of executing the test functions, so you must make every test function atomic; that is, every test function must not depend on any other test functions. For example, do not do the following in QUnit:



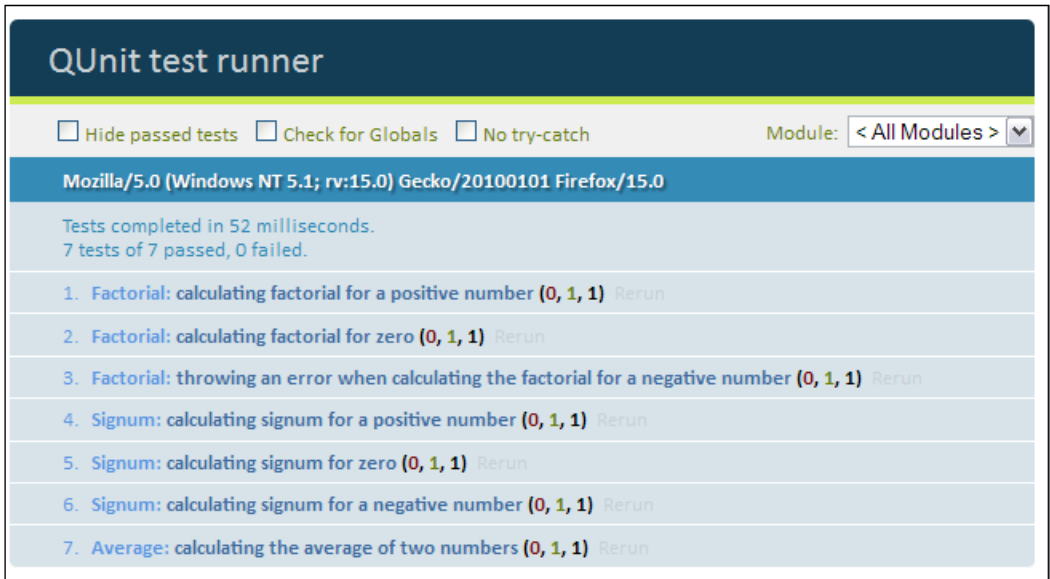
```
var counter = 0;
test("test function1", function() {
    counter++;
    equal(counter, 1, "counter should be 1");
});
test("test function2", function() {
    counter += 20;
    equal(counter, 21, "counter should be 21");
});
test("test function3", function() {
    counter += 10;
    equal(counter, 31, "counter should be 31");
});
```

In order to run the SimpleMath QUnit tests, we need to include the SimpleMath.js and simpleMathTest.js (which contains the unit tests of the SimpleMath object) files in the test runner page, as shown in the following code snippet:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>QUnit test runner</title>
  <link rel="stylesheet" href="lib/qunit-1.10.0.css">
</head>
<body>
  <div id="qunit"></div>
  <div id="qunit-fixture"></div>
  <script src="lib/qunit-1.10.0.js"></script>

  <script src="src/simpleMath.js"></script>
  <script src="tests/simpleMathTest.js"></script>
</body>
</html>
```

After clicking the SimpleMath QUnit test page `testRunner.html`, you will find the test results as shown in the following screenshot:



Assertions

An assertion is a function that validates a condition if the condition is not valid; it throws an error that causes the test function to fail. A test function can include one or more assertions; all the assertions have to pass in order to have the test function pass. In the first QUnit test example, we used the QUnit `equal` and `raises` assertions. In this section, the most important QUnit built-in assertions will be illustrated in more detail.

The `ok` assertion

The `ok` assertion takes two parameters. The first parameter is a condition and the second parameter is optional and represents the message that is to be displayed with the test. The `ok` assertion is passed if the condition is true. For example, the following examples will pass:

```
ok(true, "true passes");
ok(4==4, "4 must equal 4");
ok("some string", "Non-empty string passes");
```


The equal and notEqual assertions

The `equal` assertion has three parameters; the first two parameters represent the actual and expected values, and the third parameter is optional and represents the message that is to be displayed with the test. The `equal` assertion is passed if the actual parameter is equal to the expected parameter. The `notEqual` assertion ensures that the actual and expected parameters are not equal.

It is very important to know that the `equal` and `notEqual` assertions use the JavaScript `==` operator in order to perform the comparison, that is, they make the comparison and neglect the types. For example, the following assertions will pass:

```
equal(5, 5, "5 should equal 5...");
equal(5, "5", "5 should equal '5'...");
notEqual(5, 6, "5 should not equal 6...");
notEqual(5, "6", "5 should not equal '6'...");
```

The deepEqual and notDeepEqual assertions

The `deepEqual` assertion is more powerful than the `equal` assertion. It makes a deep comparison (recursively) between objects, arrays, and primitive datatypes. Unlike the `equal` assertion, the `deepEqual` assertion uses the `===` operator to perform the comparison (that is, it does not ignore the types). The `notDeepEqual` assertion function does the reverse operation of the `deepEqual` assertion. For example, the following assertions will pass:

```
// Objects comparison
var object1 = {a:1, b:2, c : {c1: 11, c2: 12}};
var object2 = {a:1, b:2, c : {c1: 11, c2: 12}};
var object3 = {a:1, b:"2", c : {c1: 11, c2: 12}};
deepEqual(object1, object2, "object1 should equal object2");
notDeepEqual(object1, object3, "object1 should not equal object3");

// Primitive comparison
deepEqual(1, 1, "1 === 1");
notDeepEqual(1, "1", "1 !== '1'");
```

As you will notice in the preceding code snippet, `object3` does not equal `object1` because the `deepEqual` assertion uses the `===` operator; this means that `b:2` does not equal `b:"2"`.

The expect assertion

The `expect` assertion is used for defining the number of assertions that the test function must contain. If the test function is completed without the correct number of assertions specified in the `expect` parameter, the test fails. For example, the following test function will fail:

```
test("test function1", function() {  
  expect(3);  
  ok(true);  
  equal(1, 1);  
});
```

The test fails because the `expect(3)` expects to see three assertions in the test function. If we insert the third assertion, the test function passes, as follows:

```
test("test function1", function() {  
  expect(3);  
  ok(true);  
  equal(1, 1);  
  deepEqual("1", "1");  
});
```

Instead of using the `expect` assertion, the expectation count can be passed as the second parameter to the test function, as follows:

```
test("test function1", 3, function() {  
  ok(true);  
  equal(1, 1);  
  deepEqual("1", "1");  
});
```



The `expect` assertion can be useful when you have a probability of *not* executing one or more assertions in your QUnit test code due to some reason, such as an operation failure. This can happen while testing asynchronous operations for which the execution of one or more assertions cannot be guaranteed if the operation fails or times out. The only remaining important, built-in assertion is the `raises` assertion, and you already know how it works in the `SimpleMath` object example.

Developing custom QUnit assertions

Adding to the mentioned built-in QUnit assertions, QUnit enables you to develop custom assertions to have more powerful and descriptive testing code. Let's develop two custom assertions, which are the `isPrimeNumber` and `sum` assertions, in order to understand how to develop custom assertions in QUnit.

The purpose of the `isPrimeNumber(number, message)` assertion is to check if the passed number is a prime number, while the `sum(number1, number2, result, message)` assertion checks if the sum of its first two number arguments is equal to the third number argument.

In order to define a custom assertion in QUnit, you should use the `QUnit.push` API. The `QUnit.push` API has the following parameters:

- `result`: If it is set to `true`, this means that the test succeeds, and if it is set to `false`, this means that the test fails
- `actual`: It represents the actual value
- `expected`: It represents the expected value
- `message`: It represents the message that is to be displayed with the test function

The main usage of the `actual` and `expected` parameters is that they are used by the QUnit framework in order to help the developer troubleshoot the test when it fails, as shown in the next screenshot. Let's start implementing the `sum` custom assertion. The following code snippet shows the `sum` custom assertion code:

```
function sum(number1, number2, result, message) {
    var expected = number1 + " + " + number2 + " = " + result;
    var actual = expected;

    if ((number1 + number2) !== result) {
        actual = number1 + " + " + number2 + " !== " + result;
    }

    QUnit.push((number1 + number2) === result, actual, expected,
        message);
}
```

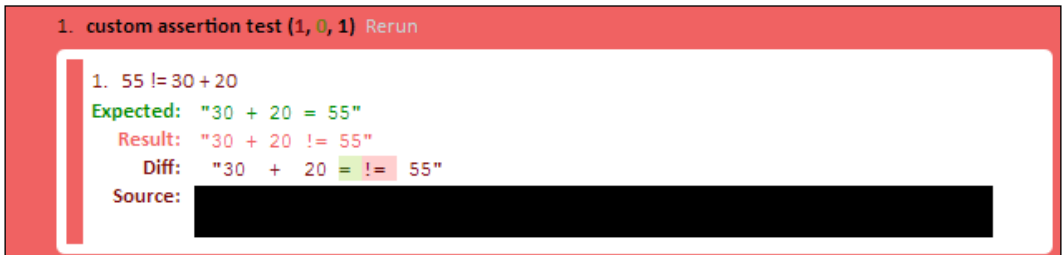
The first parameter of the `QUnit.push` API is set to the Boolean result for checking that the summation of `number1` and `number2` is equal to `result`. When the two numbers `number1` and `number2` are not equal to `result`, the actual and expected parameters should have meaningful values in order to help the developer debug the failing test. The actual parameter is set to `number1 + number2 !== result` while the expected parameter is always set to `number1 + number2 = result`.

The `sum` custom assertion, works just like any other QUnit assertion. You can use it as in the following code snippet:

```
sum(30, 20, 50, "50 = 30 + 20");
```

The next test will fail and the result will be displayed, as shown in the following screenshot:

```
sum(30, 20, 55, "55 != 30 + 20");
```



As shown in the previous screenshot, when the test fails, QUnit uses the actual and expected parameters that are set in the custom assertion to display the **Expected**, **Result**, and **Diff** items for helping the developer debug the test. The following code snippet shows the `isPrimeNumber` custom assertion code:

```
function isPrimeNumber(number, message) {
  if (number < 2) {
    QUnit.push(false, false, true, message);
    return;
  }

  var n = Math.sqrt(number);

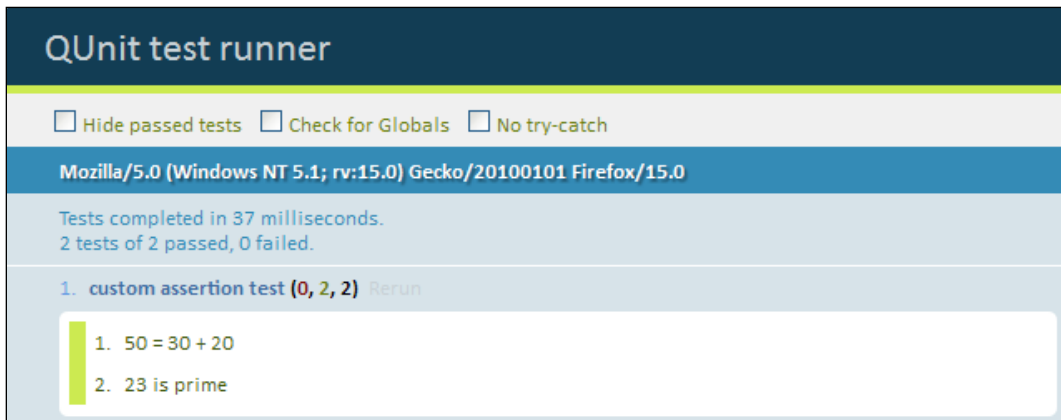
  for (var i = 2; i <= n; ++i) {
    if (number % i == 0) {
      QUnit.push(false, false, true, message);
      return;
    }
  }
}
```

```
    QUnit.push(true, true, true, message);  
    return;  
}
```

If the passed `number` parameter is not a prime number, then the first parameter of the `QUnit.push` API is set to `false` in order to fail the test. The actual parameter is set to `false` while the expected parameter is set to `true` in order to show the error details in the test runner page. The following code snippet shows the complete code and usage of the custom assertions:

```
function sum(number1, number2, result, message) {  
    var expected = number1 + " + " + number2 + " = " + result;  
    var actual = expected;  
  
    if ((number1 + number2) !== result) {  
        actual = number1 + " + " + number2 + " !== " + result;  
    }  
  
    QUnit.push((number1 + number2) === result, actual, expected,  
message);  
}  
  
function isPrimeNumber(number, message) {  
    if (number < 2) {  
        QUnit.push(false, false, true, message);  
        return;  
    }  
  
    var n = Math.sqrt(number);  
  
    for (var i = 2; i <= n; ++i) {  
        if (number % i === 0) {  
            QUnit.push(false, false, true, message);  
            return;  
        }  
    }  
  
    QUnit.push(true, true, true, message);  
    return;  
}  
  
test("custom assertion test", function() {  
    sum(30, 20, 50, "50 = 30 + 20");  
    isPrimeNumber(23, "23 is prime");  
});
```

After running the preceding code snippet, the QUnit test runner page will display the successful test results of the custom assertions, as shown in the following screenshot:



Testing asynchronous (Ajax) JavaScript code

The common question that comes to mind is how to test asynchronous (Ajax) JavaScript code using QUnit. What has been mentioned in the chapter so far is how to perform unit testing for synchronous JavaScript code. QUnit provides two main APIs, namely `stop()` and `start()`, in order to perform real Ajax testing. Let me show you how to use them.

The stop and start APIs

The `stop()` API stops the QUnit test runner until the `start()` API is called or the test function is timed out. For example:

```
QUnit.config.testTimeout = 10000;
test("test function1", function() {
    stop();

    window.setTimeout(function() {
        ok(true);
        start();
    }, 3000);
});
```

As shown in the preceding code snippet, the "test function1" function stops the QUnit test runner by calling the `stop()` API. The `window.setTimeout` function resumes the test runner by calling the `start()` API after 3000 milliseconds.

In order to specify the test function timeout, you can set the global property `QUnit.config.testTimeout` to the time in milliseconds. In the previous example, it is set to 10000 milliseconds (10 seconds).

QUnit has another way of working with asynchronous operations; instead of explicitly calling the `stop()` API in the test method, you can directly use the `asyncTest` function as follows:

```
asyncTest("test function1", function() {
    window.setTimeout(function() {
        ok(true);
        start();
    }, 3000);
});
```

Using one of the two mentioned approaches, you can perform real Ajax testing. The following code snippet shows you how to create a real Ajax test using the `asyncTest` function:

```
QUnit.config.testTimeout = 10000;
asyncTest("Making a REAL Ajax testing", function() {
    var successCallback = function(response) {
        var resultMessage = response.xmlhttp.responseText;

        // Validate the result message using the QUnit assertions.

        start();
    };

    var failureCallback = function() {
        ok(false, "MUST fail");
        start();
    };

    asyncSystem.doAjaxOperation(inputData, successCallback,
        failureCallback);
});
```

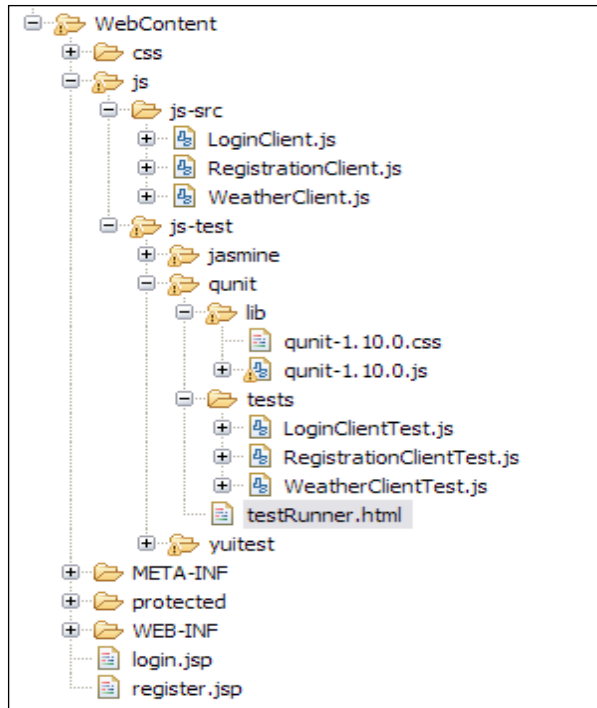
As shown in the previous code snippet, two callbacks are created; one of them represents the successful callback (`successCallback`) that is called if the Ajax operation succeeds, and the other one represents the failure callback (`failureCallback`) that is called if the Ajax operation fails. In both `successCallback` and `failureCallback`, a call to the `start()` API is made in order to notify the QUnit asynchronous test that the server response is returned and the test runner can resume. In `successCallback`, there should be calls to the QUnit assertions in order to validate the returned Ajax response, and in `failureCallback`, the `ok(false)` expression forces the test to fail because the failure callback should not be called if the asynchronous operation succeeds.

If the Ajax response is not returned from the server after 10 seconds (you can set it to whatever duration you want using `QUnit.config.testTimeout`), the test will fail. In the *Testing the weather application* section, the two provided QUnit Ajax testing approaches will be used in order to test the Ajax part of the weather application.

Testing the weather application

Now, we come to developing the QUnit tests for our weather application. Actually, after you have learned how to write QUnit 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 for developing tests for the `LoginClient`, `RegistrationClient`, and `WeatherClient` objects.

Two subfolders `qunit` and `tests` are created under the `js-test` folder (thus: `qunit\tests`) for containing the QUnit tests, and the `lib` folder is created under the `qunit` folder (thus: `qunit\lib`) for containing QUnit library files, as shown in the following screenshot:



The `tests` folder contains three main JavaScript files (`LoginClientTest.js`, `RegistrationClientTest.js`, and `WeatherClientTest.js`) for testing the weather application's corresponding JavaScript objects. The QUnit test runner file `testRunner.html` is placed directly under the `qunit` folder in the `js-test` folder (thus: `js-test\qunit`). The following code snippet shows the contents of the QUnit `testRunner.html` page of the weather application:

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>QUnit test runner</title>
  <link rel="stylesheet" href="lib/qunit-1.10.0.css">
</head>
<body>
  <div id="qunit"></div>
  <div id="qunit-fixture"></div>
  <script src="lib/qunit-1.10.0.js"></script>

  <!-- Source files -->
  <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="../../js-
src/WeatherClient.js"></script>

<!-- Test files -->
<script src="tests/LoginClientTest.js"></script>
<script src="tests/RegistrationClientTest.js"></script>
<script src="tests/WeatherClientTest.js"></script>

</body>
</html>
```

As shown in the `testRunner.html` page, both the source and test JavaScript files are included in the page. In each of the test files, a QUnit module that will be responsible for testing the corresponding JavaScript object will be created.

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 character, at least one special character, and six characters or more

In order to perform this test, a module "LoginClient Test Module" that groups all of these tests is created. The following code snippet shows the definition of "LoginClient Test Module":

```
module("LoginClient Test Module", {
  setup: function() {

    // The HTML fixture for the LoginClient.
    document.getElementById("qunit-fixture").innerHTML =
      "<label for=\"username\">Username <span
      id=\"usernameMessage\"></span></label>" +
      "<input type=\"text\" id=\"username\" name=\"username\"/>"
      +
      "<label for=\"password\">Password <span
      id=\"passwordMessage\"></span></label>" +
      "<input type=\"password\" id=\"password\"
      name=\"password\"/>";
  }
});
```

```

    this.loginClient = new weatherapp.LoginClient();

    this.loginForm = {
        "userNameField" : "username",
        "passwordField" : "password",
        "userNameMessage" : "usernameMessage",
        "passwordMessage" : "passwordMessage"
    };
}, teardown: function() {
    delete this.loginClient;
    delete this.loginForm;
}
});

```

The setup method of "LoginClient Test Module" appends the HTML fixture that is needed by the LoginClient test to the qunit-fixture div (the HTML fixture contains the username and password input fields and labels), and then 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 following code snippet shows the empty username and password test functions of "LoginClient Test Module":

```

test("validating empty username", function() {
    document.getElementById("username").value = ""; /* setting
    username to empty */
    document.getElementById("password").value = "Admin@123";

    this.loginClient.validateLoginForm(this.loginForm);

    equal(document.getElementById("usernameMessage").innerHTML,
    "(field is required)", "validating empty username ...");
});

test("validating empty password", function() {
    document.getElementById("username").value = "someone@yahoo.com";
    document.getElementById("password").value = ""; /* setting
    password to empty */

    this.loginClient.validateLoginForm(this.loginForm);

    equal(document.getElementById("passwordMessage").innerHTML,
    "(field is required)", "validating empty password ...");
});

```

The "validating empty username" test function tests `LoginClient` to ensure that it 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 verifies that the `validateLoginForm` API produces the "(field is required)" message in the `usernameMessage` field using the `equal` assertion.

The "validating empty password" test function does the same thing but with the password field and not with the username field.

The following code snippet shows the test functions of the "LoginClient Test Module", which validate the formats of the fields (the username and the password):

```
test("validating username format", function() {
    document.getElementById("username").value = "someone@yahoo"; /*
    setting username to incorrect format */
    document.getElementById("password").value = "Admin@123";

    this.loginClient.validateLoginForm(this.loginForm);

    equal(document.getElementById("usernameMessage").innerHTML,
        "(format is invalid)", "validating username format ...");
});

test("validating password format", function() {
    document.getElementById("username").value = "someone@yahoo.com";
    document.getElementById("password").value = "admin@123"; /*
    setting password to incorrect format */

    this.loginClient.validateLoginForm(this.loginForm);

    equal(document.getElementById("passwordMessage").innerHTML,
        "(format is invalid)", "validating password format ...");
});
```

The "validating username format" test function tests the validation of the username format. It tests the `LoginClient` object to ensure that it 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 `LoginClient`. Finally, it checks that the `validateLoginForm` API produces the "(format is invalid)" message in the `usernameMessage` field, using the `equal` assertion.

The "validating password format" function 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 `LoginClient`. It finally checks that 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 character, 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 "LoginClient Test Module", so let's see how to verify that the user registration (`registerUser`) Ajax functionality is performed correctly.

The `registerUser` tests cover the following test scenarios:

- Testing adding a new user with a unique user ID.
- Testing adding a user with an existing user ID. In this case, the registration client should fail when registering a user whose ID is already registered.

"RegistrationClient Test Module" groups all the `RegistrationClient` tests. The following code snippet shows the definition of the "RegistrationClient Test Module":

```
module("RegistrationClient Test Module", {
  setup: function() {

    // The HTML fixture for the RegistrationClient.
    document.getElementById("qunit-fixture").innerHTML =
      "<label for=\"username\">Username (Email) <span
      id=\"usernameMessage\"></span></label>" +
      "<input type=\"text\" id=\"username\" name=\"username\"/>" +
      "<label for=\"password1\">Password <span
      id=\"passwordMessage1\"></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"
};
}, teardown: function() {
  delete this.registrationClient;
  delete this.registrationForm;
}
});
```

The setup method of "RegistrationClient Test Module" appends the HTML fixture of the RegistrationClient test to the qunit-fixture div (the HTML fixture contains the username and passwords input fields and labels), and then creates an instance from weatherapp.RegistrationClient and creates the registrationForm object, which holds the IDs of the HTML elements that are used in the test. The following code snippet shows the first part of the "testing the registration feature" test function of "RegistrationClient Test Module", which tests registering a new user using the registerUser API:

```
QUnit.config.testTimeout = 10000;
test("testing the registration feature", function() {

  // Register a new user with a unique user name.
  stop();

  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 local_this = this;
```

```

var newSuccessCallback = function(response) {
    var resultMessage = response.xmlhttp.responseText;
    equal(resultMessage, "User is registered successfully ...",
    "Registering a new user succeeded ...");
    start();

    // Register the created user again to check that the
    // registration will fail.
    // The code will be shown in the next code snippet ...

};

var newFailureCallback = function() {
    ok(false, "Registering a new user failed ...");
    start();
};

this.registrationClient.registerUser(this.registrationForm,
newSuccessCallback, newFailureCallback);
});

```

In the test function, the `stop()` API waits for a call from the `start()` API, or it fails the test function after 10000 milliseconds. The registration form is filled with a valid generated username and valid matched passwords, and then two callbacks are created. The first callback represents the success callback while the second one represents the failure callback. `registrationClient.registerUser` is called with the registration form, the success callback, and the failure callback parameters.

In the success callback, the response message returned from the server is ensured of being equal to the "User is registered successfully ..." message using the `equal` assertion, and then a call is made to the `start()` API to proceed with the test.

In the failure callback, the `ok(false)` is called in order to fail the test function, because the failure callback should not be called if the registration is performed successfully, and then a call is made to the `start()` API to proceed with the test.



The QUnit 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 second part of the "testing the registration feature" test function that was not shown in the preceding code. The second part contains the second test scenario of the `registerUser` API that tests registering the created user again in order to ensure that the `registerUser` API will fail (because the username already exists):

```
stop();

var existingSuccessCallback = function(response) {
    ok(false, "Validating registering a user with an existing id
    failed ...");
    start();
};

var existingFailureCallback = function(response) {
    var resultMessage = response.xmlhttp.responseText;
    equal(resultMessage, "A user with the same username is already
    registered ...", "Validating registering a user with an existing
    id succeeded ...");
    start();
};

local_this.registrationClient.registerUser(local_this.
registrationForm, existingSuccessCallback, existingFailureCallback);
```

The `stop()` API waits for a call to the `start()` API, or it fails after the timeout period has passed. The registration form is still holding the same username that has already been registered in the first test scenario of the test function, and two callbacks are created. The first callback (`existingSuccessCallback`) represents the success callback while the second one (`existingFailureCallback`) represents the failure callback. `registrationClient.registerUser` is called with the registration form, the success callback, and the failure callback parameters.

In the success callback, `ok(false)` is called in order to fail the test function because the success callback must not be called in this case, and then a call is made to the `start()` API to proceed with the test. In the failure callback, the returned response message from the server is ensured to equal the "A user with the same username is already registered ..." message using the `equal` assertion, and then a call is made to the `start()` API to proceed with the test.

The following code snippet shows the complete code of the "testing the registration feature" function of "RegistrationClient Test Module":

```
QUnit.config.testTimeout = 10000;
test("testing the registration feature", function() {

    // Register a new user.
    stop();
```



```
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 local_this = this;

var newSuccessCallback = function(response) {
    var resultMessage = response.xmlhttp.responseText;
    equal(resultMessage, "User is registered successfully ...",
    "Registering a new user succeeded ...");
    start();

    // Register the created user again (Register an existing
    user).
    stop();

    var existingSuccessCallback = function(response) {
        ok(false, "Validating registering a user with an existing id
        failed ...");
        start();
    };

    var existingFailureCallback = function(response) {
        var resultMessage = response.xmlhttp.responseText;
        equal(resultMessage, "A user with the same username is
        already registered ...", "Validating registering a user with
        an existing id succeeded ...");
        start();
    };

    local_this.registrationClient.registerUser(local_this.
    registrationForm, existingSuccessCallback, existingFailureCallback);
};

var newFailureCallback = function() {
    ok(false, "Registering a new user failed ...");
    start();
};

this.registrationClient.registerUser(this.registrationForm,
newSuccessCallback, newFailureCallback);
});
```

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 in this case)

For the time being, this test will not be left for you as an exercise because the other QUnit Ajax testing approach using `asyncTest` will be used for testing the WeatherClient object. In order to perform the WeatherClient test, "WeatherClient Test Module" that groups all the WeatherClient tests is created. The following code snippet shows the definition of "WeatherClient Test Module":

```
module("WeatherClient Test Module", {
  setup: function() {

    // The HTML fixture for the WeatherClient.
    document.getElementById("qunit-fixture").innerHTML =
      "<div id='weatherInformation'></div>";

    this.weatherClient = new weatherapp.WeatherClient();

    this.validLocationForm = {
      'location': '1521894',
      'resultDivID': 'weatherInformation'
    };

    this.invalidLocationForm = {
      'location': 'INVALID_LOCATION',
      'resultDivID': 'weatherInformation'
    };
  }, teardown: function() {
    delete this.weatherClient;
    delete this.validLocationForm;
    delete this.invalidLocationForm;
  }
});
```

The setup method of "WeatherClient Test Module" appends the HTML fixture of the WeatherClient test to the qunit-fixture div (the HTML fixture contains the weatherInformation div element), and then creates an instance from weatherapp.WeatherClient creates the validLocationForm object that represents a valid location form (that contains a valid location code and the ID of the weatherInformation div element), and finally creates the invalidLocationForm

object that represents an invalid location form (that contains an invalid location code and the ID of the weatherInformation div element). The following code snippet shows the "getting the weather information for a valid place" test function of "WeatherClient Test Module" that tests the `getWeatherCondition` API's behavior with a valid location code:

```
QUnit.config.testTimeout = 10000;
asyncTest("getting the weather information for a valid place",
function() {
    var successCallback = function(response) {
        var resultMessage = response.xmlhttp.responseText;

        notEqual(resultMessage, "", "Getting the weather information
        for a valid place succeeded");
        start();
    };

    var failureCallback = function() {
        ok(false, "Getting the weather information for a valid place
        failed ...");
        start();
    };

    this.weatherClient.getWeatherCondition(this.validLocationForm,
    successCallback, failureCallback);
});
```

In order to test the `getWeatherCondition` method, the `asyncTest` API has been used this time instead of the `test` API. As shown, there are no `stop()` calls because `stop()` is called implicitly by the `asyncTest` API. By calling the `stop()` API implicitly, the `asyncTest` API waits for a call from the `start()` API or it fails the test function after 10000 milliseconds.

Two callbacks are created. The first callback (`successCallback`) represents the success callback while the second one (`failureCallback`) represents the failure callback. Finally, `weatherClient.getWeatherCondition` is called with the valid location form, the success callback, and the failure callback parameters.

In the success callback, the response message returned from the server is ensured of not being equal to an empty message using the `notEqual` assertion (the server response message should contain the weather information for the passed location), and then a call is made to the `start()` API to proceed with the test.

In the failure callback, `ok(false)` is called in order to fail the test function because the failure callback must not be called in case you want to get weather information for a valid location. Finally, a call is made to the `start()` API to proceed with the test.

The following code snippet shows the other "getting the weather information for an invalid place" test function of "WeatherClient Test Module":

```
asyncTest("getting the weather information for an invalid place",
function() {
    var successCallback = function() {
        ok(false, "Getting the weather information for an invalid
        place succeeded (MUST NOT Happen)!!!");
        start();
    };

    var failureCallback = function(response) {
        var resultMessage = response.xmlhttp.responseText;

        equal(resultMessage, "Invalid location code", "Getting the
        weather information for an invalid place failed (Expected
        ...)");
        start();
    };

    this.weatherClient.getWeatherCondition(this.invalidLocationForm,
    successCallback, failureCallback);
});
```

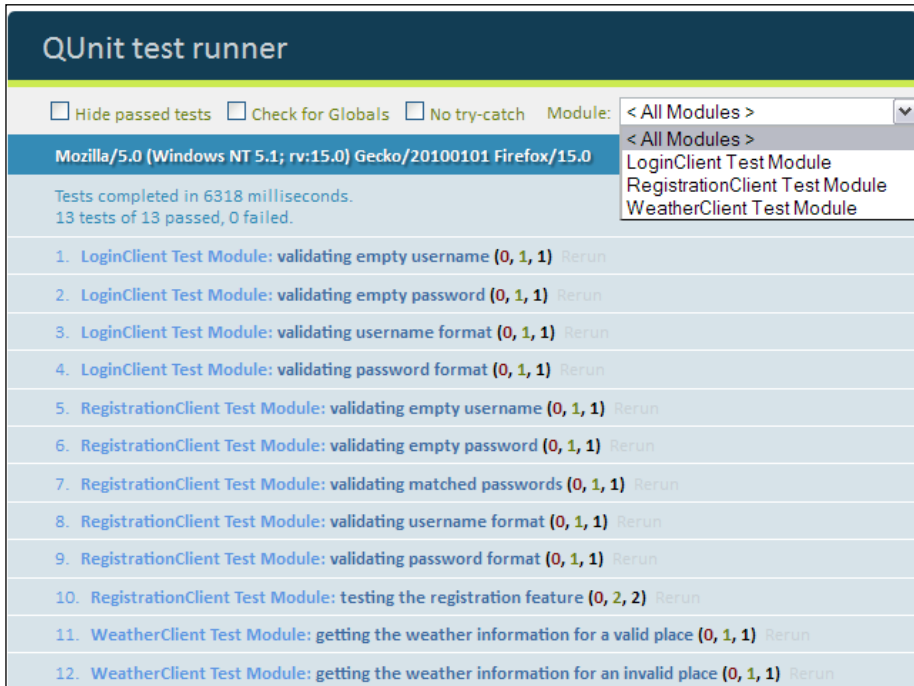
As shown in the previous code snippet, the "getting the weather information for an invalid place" test function follows the same approach as that of the previous test function. The main difference is that it ensures that `failureCallback` is called and the server response message is validated to be "Invalid location code", and finally it is ensured that `successCallback` is not called.

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 in order to 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 following URL in the browser's address bar:

```
http://localhost:8080/weatherApplication/js/js-test/qunit/testRunner.
html
```

The following screenshot shows the weather application's QUnit test results:



As shown in the preceding screenshot, the test modules appear in the drop-down menu in the top-right part of the test page. You can filter which test modules you want to execute using this drop-down menu; for example, if you select the **LoginClient Test Module** menu item, only the `LoginClient` test functions will be executed, as with the other test modules.

Summary

In this chapter, you learned what QUnit is and how to use it for testing synchronous JavaScript code. You learned how to test asynchronous (Ajax) JavaScript code using the QUnit test and QUnit `asyncTest` mechanisms. You learned the assertions provided by the framework, and how to develop your own assertion in order to simplify your test code. You also learned how to load HTML fixtures easily in your QUnit tests. Finally, you learned how to apply all of these concepts for testing the weather application using QUnit. In the next chapter, you will learn how to work with the `JsTestDriver` framework, and learn how to use it for testing the JavaScript part of the weather application. Along with this, you will also learn how to automate the QUnit and Jasmine tests using the `JsTestDriver` framework.