2
Jasmine

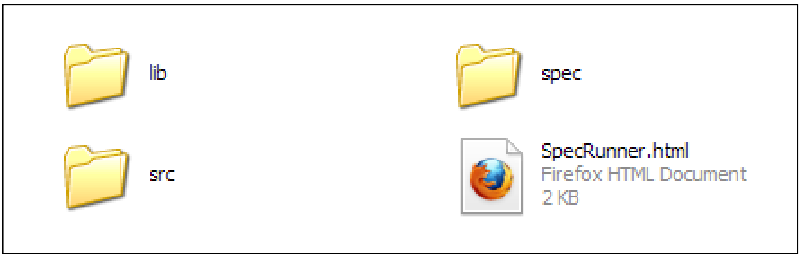
Jasmine is a powerful JavaScript unit testing framework. It provides a clean mechanism for testing synchronous and asynchronous JavaScript code. Jasmine is a behavior-driven development framework that provides descriptive test cases that focus more on the business value than on the technical details. Because it is written in a simple natural language, Jasmine tests can be read by non-programmers and can provide a clear description when a single test succeeds or fails and also the reason behind its failure. 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*.

> **Behavior-driven development** (**BDD**) is an agile software development technique introduced by Dan North that focuses on writing descriptive tests from the business perspective. BDD extends TDD by writing test cases that test the software behavior (requirements) in a natural language that anyone (not necessarily a programmer) can read and understand. The names of the unit tests are sentences that usually start with the word "should" and they are written in the order of their business value.

Configuration

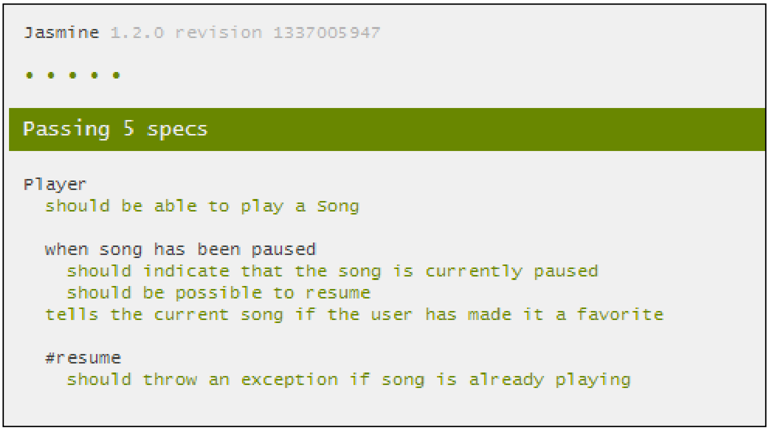
In order to configure Jasmine, the first step is to download the framework from `https://github.com/pivotal/jasmine/downloads`. Here, you will find the latest releases of the framework. At the time of this writing, the latest release is v1.2.0, which has been used in this book.

After unpacking `jasmine-standalone-1.2.0.zip` (or later), you will find the folder structure shown in the following screenshot:



The `src` folder in the preceding screenshot contains the JavaScript source files that you want to test, the `spec` folder contains the JavaScript testing files, while `SpecRunner.html` is the test case runner HTML file. The `lib` folder contains the framework files.

In order to make sure that everything is running OK, click on the `SpecRunner.html` file; you should see passing specs, as shown in the following screenshot:



```
Jasmine 1.2.0 revision 1337005947

• • • • •

Passing 5 specs

Player
  should be able to play a Song

  when song has been paused
    should indicate that the song is currently paused
    should be possible to resume
  tells the current song if the user has made it a favorite

  #resume
    should throw an exception if song is already playing
```

This structure is not rigid; you can modify it to serve the organization of your application. For the purpose of testing the weather application, we will modify it to cope with the structure of the application.

Writing your first Jasmine test

Before writing the first Jasmine test, we will need to understand the difference between a suite and a spec (test specification) in Jasmine. A Jasmine **suite** is a group of test cases that can be used to test a specific behavior of the JavaScript code (a JavaScript object or function). In Jasmine, the test suite begins with a call to the Jasmine global function describe with two parameters. The first parameter represents the title of the test suite, while the second parameter represents a function that implements the test suite.

A Jasmine **spec** represents a test case inside the test suite. In Jasmine, a test case begins with a call to the Jasmine global function it with two parameters. The first parameter represents the title of the spec and the second parameter represents a function that implements the test case.

A Jasmine spec contains one or more expectations. Every expectation represents an assertion that can be either true or false. In order to pass the spec, all of the expectations inside the spec have to be true. If one or more expectations inside a spec is false, the spec fails. The following code snippet shows an example of a Jasmine test suite and a spec with an expectation:

```
describe("A sample suite", function() {
  it("contains a sample spec with an expectation", function() {
    expect(true).toEqual(true);
  });
});
```

Now, let's move to the SimpleMath JavaScript object, which is described in the following code snippet. The SimpleMath JavaScript object is a simple mathematical utility that performs factorial, signum, and average mathematical operations.

```
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 the preceding code snippet, the `SimpleMath` object is used to calculate the factorial of numbers. In mathematics, the factorial function of a nonnegative integer *n*, which is denoted by *n!*, is the product of all positive integers less than or equal to *n*. For example, *4! = 4 x 3 x 2 x 1 = 24*. According to Wikipedia, the factorial function has the following mathematical definition:

$$\boxed{n! = \begin{cases} 1 & \text{if } n = 0, \\ (n - 1)! \text{ x } n & \text{if } n > 0. \end{cases}}$$

The `SimpleMath` object calculates the factorial of the number using the `getFactorial` recursive function. It throws an `Error` exception when the passed parameter to the `getFactorial` method is a negative number because there is no factorial value for negative numbers.

In addition to calculating the factorial of numbers, it can get the signum of any number using the `signum` method. In mathematics, the signum function extracts the sign of a real number. According to Wikipedia, the signum function has the following mathematical definition:

$$\boxed{\operatorname{sgn}(x) = \begin{cases} -1 & \text{if } x < 0, \\ 0 & \text{if } x = 0, \\ 1 & \text{if } x > 0. \end{cases}}$$

Finally, `SimpleMath` can calculate the average of two numbers using the `average` method. The average value of two numbers can be calculated by dividing the sum of the two numbers by 2.

Now, let's start writing the specs using Jasmine. First of all, in order to test the `getFactorial` method, let's have the three following test scenarios; we will test calculating the factorial of:

- A positive number
- Zero
- A negative number

> **Boundary testing** is a kind of testing that focuses on the boundary or the limit conditions of the objects to be tested. These boundary conditions can include the maximum value, minimum value, error values, and inside/outside boundary values. In the factorial testing example, the test scenarios apply this kind of testing by testing the factorial API with a positive number, a negative number, and zero.

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

```
describe("SimpleMath", function() {
  var simpleMath;

  beforeEach(function() {
      simpleMath = new SimpleMath();
  });

  describe("when SimpleMath is used to find factorial", function() {
       it("should be able to find factorial for positive number",
         function() {
      expect(simpleMath.getFactorial(3)).toEqual(6);
       });

       it("should be able to find factorial for zero", function() {
      expect(simpleMath.getFactorial(0)).toEqual(1);
       });

       it("should be able to throw an exception when the number is
          negative", function() {
      expect(
        function() {
          simpleMath.getFactorial(-10)
        }).toThrow("There is no factorial for negative numbers");
       });

  });
...
});
```

The describe keyword declares a new test suite called "SimpleMath". beforeEach is used for initialization of the specs inside the suite, that is, beforeEach is called once before the run of each spec in the describe function. In the beforeEach function, the simpleMath object is created using new SimpleMath().

> In Jasmine, it is also possible to execute JavaScript code after the run of each spec in the describe function, using the afterEach global function. Having beforeEach and afterEach in Jasmine allows the developer not to repeat setup and finalization code for each spec.

After initializing the simpleMath object, you can either create a direct spec using the it keyword or create a child test suite using the describe keyword. For the purpose of organizing the example, a new test suite is created for each group of tests with similar functionalities. This is why an independent test suite is created to test the functionality of the getFactorial test suite provided by the SimpleMath object using the describe keyword.

In the first test scenario of the getFactorial test suite, the spec title is "should be able to find factorial for positive number", and the expect function calls simpleMath.getFactorial(3) and expects it to be equal to 6. If simpleMath.getFactorial(3) returns a value other than 6, the test fails. We have many other options (matchers) to use instead of toEqual. These matchers will be discussed in more detail in the *Jasmine matchers* section.

In the second test scenario of the getFactorial test suite, the expect function calls simpleMath.getFactorial(0) and expects it to be equal to 1. In the last test scenario of the getFactorial test suite, the expect function calls simpleMath.getFactorial(-10) and expects it to throw an exception with the message "There is no factorial for negative numbers", using the toThrow matcher. The toThrow matcher succeeds if the function expect throws an exception when executed.

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

```
describe("when SimpleMath is used to find signum", function() {
   it("should be able to find the signum for a positive number",
     function() {
    expect(simpleMath.signum(3)).toEqual(1);
   });

   it("should be able to find the signum for zero", function() {
    expect(simpleMath.signum(0)).toEqual(0);
   });
```

```
    it("should be able to find the signum for a negative number",
      function() {
     expect(simpleMath.signum(-1000)).toEqual(-1);
    });
});
```

We have three test scenarios for the `signum` method, the first test scenario is about getting the signum value for a positive number, the second test scenario is about getting the signum value for zero, and the last test scenario is about getting the signum value for a negative number. As indicated in the definition of the signum function, it has to return +1 for any positive number, 0 for zero, and finally -1 for any negative number. The following code snippet shows the average test suite:

```
describe("when SimpleMath is used to find the average of two
          values", function() {
     it("should be able to find the average of two values",
        function() {
    expect(simpleMath.average(3, 6)).toEqual(4.5);
     });
});
```

In the `average` spec, the test ensures that the average is calculated correctly by trying to calculate the average of two numbers, `3` and `6`, and expecting the result to be `4.5`.

Now, after writing the suites and the specs, it is the time to run the tests. In order to run the tests, we need to do the following steps:

1. Place the `simpleMath.js` file in the `src` folder.
2. Place the `simpleMathSpec.js` file ,which contains the `SimpleMath` unit tests, in the `spec` folder.
3. Edit the `SpecRunner.html` file as shown in the following code snippet:

```
<html>
<head>
  <title>Jasmine Spec Runner</title>

  <link rel="shortcut icon" type="image/png"
  href="lib/jasmine-1.2.0/jasmine_favicon.png">
  <link rel="stylesheet" type="text/css" href="lib/jasmine-
  1.2.0/jasmine.css">
  <script type="text/javascript" src="lib/jasmine-
  1.2.0/jasmine.js"></script>
  <script type="text/javascript" src="lib/jasmine-
  1.2.0/jasmine-html.js"></script>
```

```
        <!-- include spec files here... -->
        <script type="text/javascript"
        src="spec/simpleMathSpec.js"></script>

        <!-- include source files here... -->
        <script type="text/javascript"
        src="src/simpleMath.js"></script>
```

As shown in the preceding code snippet, in the highlighted lines, `<script type="text/javascript" src="spec/simpleMathSpec.js"></script>` is added under the `<!-- include spec files here... -->` section, while `<script type="text/javascript" src="src/simpleMath.js"></script>` is added under the `<!-- include source files here... -->` section. After double-clicking on `SpecRunner.html`, you will see the test results passed.

The nested describe blocks

Jasmine is flexible in nesting the `describe` blocks with specs at any level. This means that, before executing a spec, Jasmine walks down executing each `beforeEach` function in order, then executes the spec, and lastly walks up executing each `afterEach` function.

The following code snippet is an example of the Jasmine's nested `describe` blocks:

```
describe("MyTest", function() {
  beforeEach(function() {
  alert("beforeEach level1");
  });
  describe("MyTest level2", function() {
           beforeEach(function() {
    alert("beforeEach level2");
    });
    describe("MyTest level3", function() {
      beforeEach(function() {
      alert("beforeEach level3");
      });
      it("is a simple spec in level3", function() {
      alert("A simple spec in level3");
      expect(true).toBe(true);
      });
      afterEach(function() {
      alert("afterEach level3");
      });
    });
```

```
        afterEach(function() {
        alert("afterEach level2");
        });
      });
      afterEach(function() {
      alert("afterEach level1");
      });

    });
```

This test will result in the following messages on the alert boxes:

- **beforeEach level1**
- **beforeEach level2**
- **beforeEach level3**
- **A simple spec in level3**
- **afterEach level3**
- **afterEach level2**
- **afterEach level1**

Jasmine matchers

In the first Jasmine example, we used the `toEqual` and `toThrow` Jasmine matchers. In this section, the other different built-in matchers provided by Jasmine will be illustrated and will explain how to write a custom Jasmine matcher to have more powerful and descriptive testing code.

The toBe matcher

The `toBe` matcher is passed if the actual value is of the same type and value as that of the expected value. It uses `===` to perform this comparison. The following code snippet shows an example of the `toBe` matcher:

```
describe("the toBe Matcher", function() {
    it("should compare both types and values", function() {
      var actual = "123";
      var expected = "123";

      expect(actual).toBe(expected);
    });
});
```

You might question the difference between the toBe and toEqual matchers. The answer to this question would be that the toEqual matcher provides a powerful mechanism for handling equality; it can handle array comparisons, for example, as shown in the following code snippet:

```
describe("the toEqual Matcher", function() {
    it("should be able to compare arrays", function() {
      var actual = [1, 2, 3];
      var expected = [1, 2, 3];

      expect(actual).toEqual(expected);
    });
});
```

The following code snippet shows how the toBe matcher is unable to compare two equivalent arrays:

```
describe("the toBe Matcher", function() {
    it("should not be able to compare arrays", function() {
      var actual = [1, 2, 3];
         var expected = [1, 2, 3];

      expect(actual).not.toBe(expected);
    });

});
```

As you may have noticed in the preceding code snippet, the not keyword is used for making the test passes because the toBe matcher will not be able to know that the actual and expected arrays are the same. The Jasmine not keyword can be used with every matcher's criteria for inverting the result.

The toBeDefined and toBeUndefined matchers

The toBeDefined matcher is used to ensure that a property or a value is defined, while the toBeUndefined matcher is used to ensure that a property or a value is undefined. The following code snippet shows an example of both matchers:

```
describe("the toBeDefined Matcher", function() {
    it("should be able to check defined objects", function() {
      var object = [1, 2, 3];

      expect(object).toBeDefined();
```

```
        });
    });

    describe("the toBeUndefined Matcher", function() {
        it("should be able to check undefined objects", function() {
          var object;

          expect(object).toBeUndefined();
        });
    });
```

You can achieve the behavior of the `toBeUndefined` matcher by using the `not` keyword and the `toBeDefined` matcher, as shown in the following code snippet:

```
    describe("the toBeUndefined Matcher using the not keyword and the
    toBeDefined matcher", function() {
        it("should be able to check undefined objects", function() {
          var object;
          expect(object).not.toBeDefined();
        });
    });
```

The toBeNull matcher

The `toBeNull` matcher is used to ensure that a property or a value is null.
The following code snippet shows an example of the `toBeNull` matcher:

```
      describe("the toBeNull Matcher", function() {
          it("should be able to check if an object value is null",
          function() {
            var object = null;

            expect(object).toBeNull();
          });
      });
```

The toBeTruthy and toBeFalsy matchers

The `toBeTruthy` matcher is used to ensure that a property or a value is `true` while the `toBeFalsy` matcher is used for ensuring that a property or a value is `false`.
The following code snippet shows an example of both matchers:

```
    describe("the toBeTruthy Matcher", function() {
        it("should be able to check if an object value is true",
    function() {
        var object = true;
        expect(object).toBeTruthy();
      });
  });

  describe("the toBeFalsy Matcher", function() {
      it("should be able to check if an object value is false",
  function() {
        var object = false;

        expect(object).toBeFalsy();
      });
  });
```

The toContain matcher

The `toContain` matcher is used to check whether a string or array contains a substring or an item. The following code snippet shows an example of the `toContain` matcher:

```
  describe("the toContain Matcher", function() {
      it("should be able to check if a String contains a specific
      substring", function() {
        expect("Hello World from Cairo").toContain("Cairo");
      });

      it("should be able to check if an Array contains a specific
      item", function() {
        expect(["TV", "Watch", "Table"]).toContain("Watch");
      });
  });
```

The toBeLessThan and toBeGreaterThan matchers

The `toBeLessThan` and the `toBeGreaterThan` matchers are used to perform the simple mathematical less-than and greater-than operations, as shown in the following code snippet:

```
  describe("the toBeLessThan Matcher", function() {
      it("should be able to perform the less than operation",
      function() {
        expect(4).toBeLessThan(5);
      });
```

```
});

describe("the toBeGreaterThan Matcher", function() {
    it("should be able to perform the greater than operation",
    function() {
      expect(5).toBeGreaterThan(4);
    });
});
```

The toMatch matcher

The `toMatch` matcher is used to check whether a value matches a string or a regular expression. The following code snippet shows an example of the `toMatch` matcher, which ensures that the `expect` parameter is a digit:

```
describe("the toMatch Matcher", function() {
    it("should be able to match the value with a regular expression",
    function() {
      expect(5).toMatch("[0-9]");
    });
});
```

Developing custom Jasmine matchers

In addition to all of the mentioned built-in matchers, Jasmine enables you to develop custom matchers to have more powerful and descriptive testing code. Let's develop two custom matchers, `toBePrimeNumber` and `toBeSumOf`, to understand how to develop custom matchers in Jasmine.

The purpose of the `toBePrimeNumber` matcher is to check whether the actual number (the number in the `expect` function) is a prime number, while the `toBeSumOf` matcher checks whether the sum of its two arguments is equal to the actual number.

In order to define a custom matcher in Jasmine, you should use the `addMatchers` API to define the matcher(s) passing an object parameter to the API. The object parameter is represented as a set of key-value pairs. Every key in the object represents the matcher's name, while the value represents the matcher's associated function (the matcher's implementation). The definition of the matchers can be placed in either the `beforeEach` or the `it` block. The following code snippet shows the `toBePrimeNumber` and `toBeSumOf` custom matchers:

```
beforeEach(function(){
  this.addMatchers({
      toBeSumOf: function (firstNumber, secondNumber) {
        return this.actual == firstNumber + secondNumber;
      },
      toBePrimeNumber: function() {
        if (this.actual < 2) {
        return false;
        }

        var n = Math.sqrt(this.actual);

        for (var i = 2; i <= n; ++i) {
        if (this.actual % i == 0) {
        return false;
        }
        }

        return true;
      }
  });
});
```

After defining the custom matchers, they can be used like the other built-in matchers in the test code, as shown in the following code snippet:

```
describe("Testing toBeSumOf custom matcher", function() {
    it("should be able to calculate the sum of two numbers",
    function() {
      expect(10).toBeSumOf(7, 3);
    });
});

describe("Testing toBePrimeNumber custom matcher", function() {
     it("should be able to know prime number", function() {
      expect(13).toBePrimeNumber();
    });

    it("should be able to know non-prime number", function() {
      expect(4).not.toBePrimeNumber();
    });
});
```

As shown in the preceding code snippet, you can use the `not` keyword with your defined custom matchers.

Testing asynchronous (Ajax) JavaScript code

Now, the question that comes to mind is how to test asynchronous (Ajax) JavaScript code using Jasmine. What was mentioned in the chapter so far is how to perform unit testing for synchronous JavaScript code. Jasmine fortunately includes powerful functions (`runs()`, `waits()`, and `waitsFor()`) for performing real Ajax testing (which requires the backend server to be up and running in order to complete the Ajax tests), and it also provides a mechanism for making fake Ajax testing (which does not require the availability of the backend server in order to complete the Ajax tests).

The runs() function

The code inside the `runs()` block runs directly as if it were outside the block. The main purpose of the `runs()` block is to work with the `waits()` and `waitsFor()` blocks to handle the testing of the asynchronous operations.

The `runs()` block has some characteristics that are important to know. The first point is that, if you have multiple `runs()` blocks in your spec, they will run sequentially, as shown in the following code snippet:

```
describe("Testing runs blocks", function() {
  it("should work correctly", function() {
    runs(function() {
      this.x = 1;
      expect(this.x).toEqual(1);
    });

    runs(function() {
      this.x++;
      expect(this.x).toEqual(2);
    });

    runs(function() {
      this.x = this.x * 4;
      expect(this.x).toEqual(8);
    });

  });
});
```

In the preceding code snippet, the runs() blocks run in sequence; when the first runs() block completes, the value of this.x is initialized to 1. Then, the second runs() block runs, and the value of this.x is incremented by 1 to be 2. Finally, the last runs() block runs, and the value of this.x is multiplied by 4 to be 8.

The second important point here is that the properties between the runs() blocks can be shared using the this keyword, as shown in the next code snippet.

The waits() function

The waits() function pauses the execution of the next block until its timeout period parameter is passed, in order to give the JavaScript code the opportunity to perform some other operations. The following code snippet shows an example of the waits() functionality with the runs() blocks:

```
describe("Testing waits with runs blocks", function() {
  it("should work correctly", function() {
    runs(function() {
      this.x = 1;

      var localThis = this;

      window.setTimeout(function() {
        localThis.x += 99;
      }, 500);
    });

    runs(function() {
      expect(this.x).toEqual(1);
    });

    waits(1000);

    runs(function() {
      expect(this.x).toEqual(100);
    });

  });
});
```

In the first runs() block, the this.x variable is set to 1 and a JavaScript setTimeout method is created to increment the this.x variable by 99 after 500 milliseconds. Before 500 milliseconds, the second runs() block verifies that this.x is equal to 1. Then, waits(1000) pauses the execution of the next runs() block by 1000 milliseconds, which is enough time for `setTimeout` to complete its execution and incrementing `this.x` by `99` to be `100`. After the `1000` milliseconds, the last `runs()` block verifies that the `this.x` variable is `100`.

In real applications, we may not know the exact time to wait for until the asynchronous operation completes its execution. Fortunately, Jasmine provides a more powerful mechanism to wait for the results of asynchronous operations, the `waitsFor()` function.

The waitsFor() function

The `waitsFor()` function provides a more powerful interface that can pause the execution of the next block until its provided function returns true or a specific timeout period passes. The following code snippet shows an example of the `waitsFor()` functionality with the `runs()` blocks:

```
describe("Testing waitsFor with runs blocks", function() {
  it("should work correctly", function() {
    runs(function() {
      this.x = 1;

      var localThis = this;

      var intervalID = window.setInterval(function() {
        localThis.x += 1;

        if (localThis.x == 100) {
          window.clearInterval(intervalID);
        }
      }, 20);
    });

    waitsFor(function() {
      return this.x == 100;
    }, "Something wrong happens, it should not wait all of this
    time", 5000);

    runs(function() {
      expect(this.x).toEqual(100);
    });

  });
});
```

In the first runs() block, the this.x variable is set to 1, and a JavaScript setInterval method is created to continuously increment the this.x variable with 1 every 20 milliseconds, and stop incrementing this.x once its value becomes 100; that is, after 2000 milliseconds are up, setInterval stops execution. Before 2000 milliseconds are complete, the second waitsFor() function pauses executing the next runs block until either this.x reaches 100 or the operation times out after 5000 milliseconds. After 2000 milliseconds, the value of this.x becomes 100, which results in a true condition result in the return of the waitsFor() provided function. This will result in executing the next runs block, which checks that this.x is equal to 100.

The waitsFor() function is mostly used for testing real Ajax requests; it waits for the completion of the execution of the Ajax callback with the help of Jasmine Spies.

> A Jasmine Spy is a replacement for a JavaScript function that can either be a callback, an instance method, a static method, or an object constructor.

The following code snippet shows how to test a real Ajax request:

```
describe("when waitsFor is used for testing real Ajax requests",
function() {
  it("should do this very well with the Jasmine Spy", function() {


     var successCallBack = jasmine.createSpy();
     var failureCallBack = jasmine.createSpy();

     asyncSystem.doAjaxOperation(inputData, successCallBack,
     failureCallBack);

     waitsFor(function() {
           return successCallBack.callCount > 0;
       }, "operation never completed", 10000);

     runs(function() {
           expect(successCallBack).toHaveBeenCalled();
           expect(failureCallBack).not.toHaveBeenCalled();
       });
  });


});
```

In the preceding code snippet, two Jasmine Spies are created using `jasmine.createSpy()` to replace the Ajax operation callbacks (the success callback and the failure callback), and then the asynchronous system is called with the input data and the success and failure callbacks (the two Jasmine Spies). The `waitsFor()` provided function waits for the calling of the success callback by using the `callCount` property of the spy. If the success callback is not called after `10000` milliseconds, the test fails.

> In addition to the `callCount` property, Jasmine Spy has two other properties. `mostRecentCall.args` returns an array of the arguments from the last call to the Spy and `argsForCall[i]` returns an array of the arguments from the call number `i` to the Spy.

Finally, the final `runs()` block ensures that the success callback is called using the spy matcher `toHaveBeenCalled()` (you can omit this line because it is already known that the success callback is called from the `waitsFor` provided function; however, I like to add this check for increasing the readability of the test) and ensures that the failure callback is not called using the `not` keyword with the `toHaveBeenCalled()` matcher.

> In addition to the `toHaveBeenCalled()` matcher, Jasmine Spies has another custom matcher, the `toHaveBeenCalledWith(arguments)` matcher, which checks if the spy is called with the specified arguments.

The spyOn() function

In the previous section, we learned how to create a spy using the `jasmine.createSpy()` API in order to replace the Ajax callbacks with the spies for making a complete real Ajax testing. The question that may come to mind now is whether it is possible to make a fake Ajax testing using Jasmine if there is no server available and you want to check that things will work correctly after the response comes from the server. (In other words, is it possible to mock the Ajax testing in Jasmine?) The answer to this question is yes. The Ajax fake testing can be simulated using the Jasmine `spyOn()` function, which can spy on the asynchronous operation and routes its calls to a fake function. First of all, let's see how `spyOn()` works. `spyOn()` can spy on a callback, an instance method, a static method, or an object constructor.

The following code snippet shows how spyOn() can spy on an instance method of the SimpleMath object:

```
SimpleMath = function() {
};

SimpleMath.prototype.getFactorial = function (number) {
    //...
}

describe("Testing spyOn", function() {
  it("should spy on instance methods", function() {
    var simpleMath = new SimpleMath();

    spyOn(simpleMath, 'getFactorial');
    simpleMath.getFactorial(3);

    expect(simpleMath.getFactorial).toHaveBeenCalledWith(3);
  });
});
```

The spyOn() method spies on the getFactorial method of the SimpleMath object. The getFactorial method of the SimpleMath object is called with number 3. Finally, the simpleMath.getFactorial spy knows that the instance method has been called with number 3 using the toHaveBeenCalledWith matcher.

> Spies are automatically removed after each spec. So make sure that you define them in the beforeEach function or within every spec separately.

In order to simulate the fake Ajax testing behavior, the spy has a powerful method, which is the andCallFake(function) method that calls its function parameter when the spy is called. The following code snippet shows you how to perform a fake Ajax testing using Jasmine:

```
describe("when making a fake Ajax testing", function() {
  it("should be done the Jasmine Spy and the andCallFake
     function", function() {
    var successCallBack = jasmine.createSpy();
    var failureCallBack = jasmine.createSpy();
    var successFakeData = "Succcess Fake Data ...";

    spyOn(asyncSystem,
    'doAjaxOperation').andCallFake(function(inputData,
    successCallBack, failureCallBack) {
      successCallBack(successFakeData);
    });
```

```
        asyncSystem.doAjaxOperation(inputData, successCallBack,
        failureCallBack);

        expect(successCallBack).toHaveBeenCalled();
        expect(failureCallBack).not.toHaveBeenCalled();
      });
    });
```

A spy is created on the `doAjaxOperation` method of the `asyncSystem` object, and an order is given to the spy through the `andCallFake` method to call the fake function that has the same parameters of real `doAjaxOperation` when a call is done to original `asyncSystem.doAjaxOperation`. The fake function calls `successCallBack` to simulate a successful Ajax operation. After calling `asyncSystem.doAjaxOperation`, which does not go to the server anymore, thanks to the spy, as it executes the fake function, and finally `successCallBack` is checked that it has been called while `failureCallBack` is checked that it has never been called during the spec. Notice we are not using the `waits()`, `waitsFor()`, or `runs()` functions anymore in the fake testing because this test is fully performed on the client side so there is no need to wait for a response from the server.

> Besides the `andCallFake(function)` method, there are other three useful methods in the spy that you may use. The first one is the `andCallThrough()` method, which calls the original function that the spy spied on when the spy was called. The second one is the `andReturn(arguments)` method, which returns the arguments parameter when the spy is called. Finally, the `andThrow(exception)` method throws an exception when the spy is called.

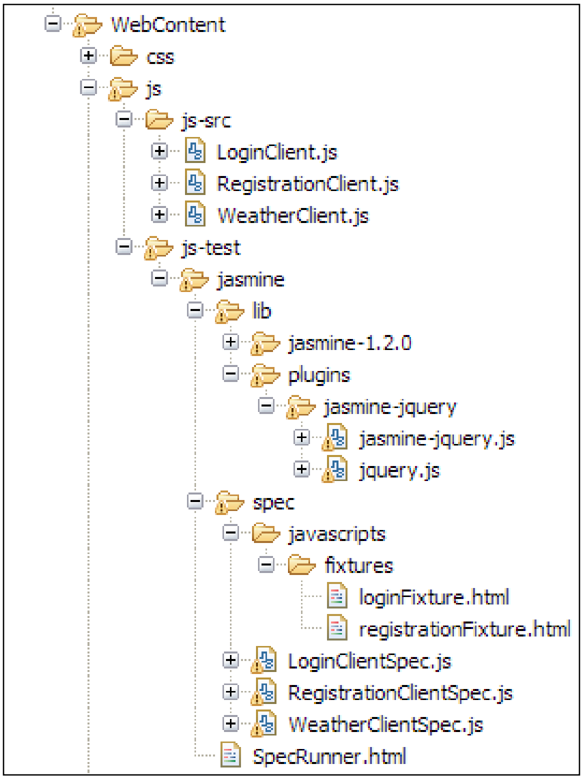
HTML fixtures

HTML fixtures are the input HTML code that is needed for executing one or more tests that require manipulating Document Object Model (DOM) elements. Jasmine does not provide an API for handling HTML fixtures in the specs. However, fortunately, there are some extensions of the framework that provide this functionality. One of the best plugins that provide this functionality is the `jasmine-jquery` plugin. Although `jasmine-jquery` goes beyond the HTML fixtures loading (it has a powerful set of matchers for the jQuery framework), I will focus only on its HTML fixture functionality, as this is what we need as JavaScript developers from Jasmine in order to test our JavaScript applications even though the applications are using a JavaScript library such as Dojo or jQuery or are not using any JavaScript library at all.

Configuring the jasmine-jquery plugin

In order to configure the `jasmine-jquery` plugin with Jasmine we need to perform the following steps:

1. Download the plugin ZIP file from `https://github.com/velesin/jasmine-jquery/downloads`.
2. Unpack the `velesin-jasmine-jquery.zip` (at the time of writing this chapter, the version of `jasmine-jquery` plugin was 1.3.2).
3. Get the `jasmine-jquery.js` file from the `lib` folder, and the `jquery.js` file from the `vendor\jquery` folder.
4. Group the `jasmine-jquery.js` and `jquery.js` files under a folder. Let's make the folder name `jasmine-jquery`. I usually place the `jasmine-jquery` folder under a `plugins` folder in the `lib` folder of Jasmine. The following screenshot shows the structure of the Jasmine tests in the weather application:



```
WebContent
  css
  js
    js-src
      LoginClient.js
      RegistrationClient.js
      WeatherClient.js
    js-test
      jasmine
        lib
          jasmine-1.2.0
          plugins
            jasmine-jquery
              jasmine-jquery.js
              jquery.js
        spec
          javascripts
            fixtures
              loginFixture.html
              registrationFixture.html
          LoginClientSpec.js
          RegistrationClientSpec.js
          WeatherClientSpec.js
        SpecRunner.html
```

5. Finally, include the two files in the `SpecRunner.html` file as shown in the highlighted lines of the following code snippet:

   ```
   <script type="text/javascript" src="lib/jasmine-
   1.2.0/jasmine.js"></script>
   <script type="text/javascript" src="lib/jasmine-
   1.2.0/jasmine-html.js"></script>

   <!-- The plugin files -->
   <script type="text/javascript" src="lib/plugins/jasmine-
   jquery/jquery.js"></script>
   <script type="text/javascript" src="lib/plugins/jasmine-
   jquery/jasmine-jquery.js"></script>

   <!-- include spec files here... -->
   ...
   ```

The loadFixtures module

This fixture module of `jasmine-jquery` allows loading the HTML content to be used by the tests. Simply, you can put the fixtures you want to load for your tests in the `spec\javascripts\fixtures` conventional folder and use the `loadFixtures` API to load the fixture(s). The following code snippet shows an example of the `loadFixtures` module:

```
beforeEach(function() {
    loadFixtures("registrationFixture.html");
});
```

In the `spec\javascripts\fixtures` folder, the `registrationFixture.html` file is as shown in the following code snippet:

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

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

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

You can change the default fixtures path instead of working with the `spec\javascripts\fixtures` conventional folder using:

```
jasmine.getFixtures().fixturesPath = '[The new path]';
```

The `loadFixtures` API can be used for loading multiple fixtures for the same test. You can use the `loadFixtures` API as follows:

```
loadFixtures(fixtureUrl[, fixtureUrl, ...])
```

Once you use the `loadFixtures` API to load the fixture(s), the fixture is loaded in the `<div id="jasmine-fixtures"></div>` container and added to the DOM using the fixture module. Fixtures are automatically cleaned up between tests so you do not have to clean them up manually. For speeding up the tests, `jasmine-jquery` makes an internal caching for the HTML fixtures in order to avoid the overhead if you decide to load the same fixture file many times in the tests.

The `loadFixtures(…)` API is a shortcut for the `jasmine.getFixtures().load(...)` so you can freely use any of them to load the HTML fixtures for the tests.

In `jasmine-jquery` you have the option to write the HTML code inline without having to load it from an external file. You can do this using the `jasmine.getFixtures().set(…)` API as follows:

```
jasmine.getFixtures().set('<div id="someDiv">HTML code …</div>');
```

While testing the weather application, both the `load` and `set` APIs will be used for loading the test fixtures.

I recommend using the inline approach if the HTML fixture is a few lines of HTML code. However, if the HTML fixture is large, then it is better to load it from an external file in order to have a better readable testing code.

This is all what we need to know from `jasmine-jquery` in order to load the needed fixtures for our tests. The next step is to write the Jasmine tests for the weather application.

Testing the weather application

Now, we come to write the Jasmine tests for our weather application. Actually, after you know how to write Jasmine tests for both synchronous and asynchronous JavaScript code and how to load the HTML fixtures in your Jasmine tests from the previous sections, testing the weather application is an easy task. As you may remember we have three major JavaScript objects in the weather application that we need to write unit tests for: the `LoginClient`, `RegistrationClient`, and `WeatherClient` objects.

One of the best practices that I recommend is to separate the JavaScript source and testing code as shown in the preceding screenshot. There are two parent folders, one for the JavaScript source, which I call `js-src` folder, and the other for the JavaScript tests, which I call `js-test` folder. The `js-test` folder contains the tests written by the testing frameworks that will be used in this book; for now, it contains a `jasmine` folder that includes the Jasmine tests.

As indicated in the *Configuration* section, Jasmine structure can be modified to fulfill the organization of every web application. The preceding screenshot shows the customized Jasmine structure for our weather application, under the `jasmine` folder; we have two subfolders, the `spec` and the `lib` folders, while the `src` folder is now represented in the `js-src` folder, which is directly under the `js` folder.

The following code snippet shows the JavaScript files included for the Jasmine files, the jasmine-jquery files, the spec files, and the source files in the `SpecRunner.html` of the weather application according to the preceding screenshot:

```
<!-- The Jasmine files -->
<link rel="shortcut icon" type="image/png" href="lib/jasmine-1.2.0/
jasmine_favicon.png">
<link rel="stylesheet" type="text/css" href="lib/jasmine-1.2.0/
jasmine.css">
<script type="text/javascript" src="lib/jasmine-1.2.0/jasmine.js"></
script>
<script type="text/javascript" src="lib/jasmine-1.2.0/jasmine-html.
js"></script>

<!-- The jasmine-jquery files -->
<script type="text/javascript" src="lib/plugins/jasmine-jquery/jquery.
js"></script>
<script type="text/javascript" src="lib/plugins/jasmine-jquery/
jasmine-jquery.js"></script>

<!-- include spec files here... -->
<script type="text/javascript" src="spec/LoginClientSpec.js"></script>
<script type="text/javascript" src="spec/RegistrationClientSpec.js"></
```

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

<!-- include source files here... -->
<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>
```

Testing the LoginClient object

In the `LoginClient` object, we will unit test 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

The following code snippet shows the first test suite of `LoginClientSpec`, which tests the validation of empty username and password:

```
describe("LoginClientSpec", function() {
   var loginClient;
   var loginForm;

   beforeEach(function() {
     loadFixtures("loginFixture.html");

     loginClient = new weatherapp.LoginClient();

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

  describe("when validating empty username and password",
  function() {
      it("should be able to display an error message when username
         is not entered", function() {
```

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

        loginClient.validateLoginForm(loginForm);

        expect(document.getElementById("usernameMessage").innerHTML).
        toEqual("(field is required)");
       });

      it("should be able to display an error message when password
      is not entered", function() {
        document.getElementById("username").value =
       "someone@yahoo.com";
        document.getElementById("password").value = "";   /*
        setting password to empty */

        loginClient.validateLoginForm(loginForm);

        expect(document.getElementById("passwordMessage").innerHTML).
toEqual("(field is required)");
      });
    });
    //...
});
```

In the preceding code snippet, `beforeEach` loads the HTML fixture of the login client test, creates an instance from `weatherapp.LoginClient`, and creates the `loginForm` object, which holds the IDs of the login form that will be used in the test. The following code snippet shows the HTML fixture of the login client test in the `loginFixture.html` file:

```
<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"/>
```

The first spec tests that the `LoginClient` object should be able to display an error message when username is not entered. It sets an empty value in the `"username"` field and then calls the `validateLoginForm` API of the `LoginClient` object. Finally, it checks that the `validateLoginForm` API produces the `"(field is required)"` message in the username message field. The second spec is doing the same thing but with the password field, not with the username field.

The following code snippet shows the second and the third test suites of `LoginClientSpec`, which validates the formats of the username and password fields:

```
describe("when validating username format", function() {
   it("should be able to display an error message when username
     format is not correct", function() {
   document.getElementById("username").value = "someone@yahoo";
   /* setting username to incorrect format */
   document.getElementById("password").value = "Admin@123";

   loginClient.validateLoginForm(loginForm);

   expect(document.getElementById("usernameMessage").innerHTML).
   toEqual("(format is invalid)");
   });
});

describe("when validating password format", function() {
     it("should be able to display an error message when
     password format is not correct", function() {
     document.getElementById("username").value =
     "someone@yahoo.com";
     document.getElementById("password").value = "admin@123";
     /* setting password to incorrect format */

     loginClient.validateLoginForm(loginForm);

     expect(document.getElementById("passwordMessage").innerHTML).
     toEqual("(format is invalid)");
     });
});
```

In the preceding code snippet, the first suite tests the validation of the username format. It tests that the `LoginClient` object should be able to display an error message when the username format is not correct. It sets an invalid e-mail value in the `"username"` field and then calls the `validateLoginForm` API of the `LoginClient` object. Finally, it checks that the `validateLoginForm` API produces the `"(format is invalid)"` message in the username message field.

The second suite does the same thing but with the password field not with the username field. It enters a password that does not comply with the application's password rules; it enters a password that does not include a capital letter, and then calls the `validateLoginForm` API of the `LoginClient` object. Finally, it checks that the `validateLoginForm` API produces the `"(format is invalid)"` message in the password message field.

It may not be always suitable while performing JavaScript unit testing to test against the application messages because the application messages can change at any time. However, in the weather application testing example, I performed testing on the application messages in order to show you how to perform testing against the HTML DOM elements. If you want to avoid testing against DOM elements, you can test against the `validateLoginForm` API directly as follows:

```
expect(loginClient.validateLoginForm(loginForm)).
toEqual(true);
```

Testing the RegistrationClient object

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

- Validation of empty username and password
- 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 points will not be explained because they are pretty similar to the tests that are explained in `LoginClientSpec`, so let's explain how to check that the user registration functionality is done correctly. The following code snippet shows the user registration test scenarios:

```
describe("RegistrationClientSpec", function() {
   var registrationClient;
   var registrationForm;
   var userName;

   beforeEach(function() {
      loadFixtures("registrationFixture.html");

      registrationClient = new weatherapp.RegistrationClient();

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

```
 });

//The user registration test scenarios

describe("when user registration is done", function() {
  it("should be able to register valid user correctly",
  function() {
    userName = "hazems" + new Date().getTime() + "@apache.org";

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

     var successCallBack = jasmine.createSpy();
     var failureCallBack = jasmine.createSpy();

     registrationClient.registerUser(registrationForm,
     successCallBack, failureCallBack);

     waitsFor(function() {
           return successCallBack.callCount > 0;
       }, "registration never completed", 10000);

     runs(function() {
           expect(successCallBack).toHaveBeenCalled();
           expect(failureCallBack).not.toHaveBeenCalled();
       });
  });

  it("should fail when a specific user id is already
  registered", function() {
     document.getElementById("username").value = userName;
     document.getElementById("password1").value = "Admin@123";
     document.getElementById("password2").value = "Admin@123";

     var successCallBack = jasmine.createSpy();
     var failureCallBack = jasmine.createSpy();

     registrationClient.registerUser(registrationForm,
     successCallBack, failureCallBack);

     waitsFor(function() {
           return failureCallBack.callCount > 0;
       }, "registration never completed", 10000);
```

```
        runs(function() {
             expect(failureCallBack).toHaveBeenCalled();
          expect(failureCallBack.mostRecentCall.args[0].xmlhttp.
responseText, "A user with the same username is already registered
...");
             expect(successCallBack).not.toHaveBeenCalled();
          });
     });

  });
});
```

In the preceding code snippet, `beforeEach` loads the fixture of the registration client test, 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. The following code snippet shows the fixture of the registration client test in the `registrationFixture.html` file:

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

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

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

The registration testing suite has two main test scenarios:

- The registration client should be able to register valid user correctly
- The registration client should fail when registering a user ID that is already registered

In the first spec, the registration form is filled with a valid username and valid matched passwords; then two spies are created. The first spy replaces the success callback while the second one replaces the failure callback. `registrationClient.registerUser` is called with the registration form, the success callback, and the failure callback parameters and the `waitsFor()` function waits for a call to the success callback or it will be timed out after `10000` milliseconds. Once `waitsFor()` is completed, the `runs` block checks that the success callback is called and the failure callback is not called for ensuring that the registration operation is completed correctly.

Note that the 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. If you want to make fake Ajax testing, for example, for the successful user registration, you can do this as you learned from the `spyOn` section as follows:

```
it("makes a fake registration Ajax call", function() {
      document.getElementById("username").value = userName;
      document.getElementById("password1").value = "Admin@123";
      document.getElementById("password2").value = "Admin@123";
       var successCallBack = jasmine.createSpy();
       var failureCallBack = jasmine.createSpy();

      spyOn(registrationClient,
      'registerUser').andCallFake(function(registrationForm,
      successCallBack, failureCallBack) {
        successCallBack();
        });

       registrationClient.registerUser(registrationForm,
       successCallBack, failureCallBack);

         expect(successCallBack).toHaveBeenCalled();
         expect(failureCallBack).not.toHaveBeenCalled();

    });
```

In the second spec, the registration form is filled with the same username that is already registered in the first spec and then two spies are created. The first spy replaces the success callback while the second one replaces the failure callback. `registrationClient.registerUser` is called with the registration form, the success callback, and the failure callback parameters and the `waitsFor()` function waits for a call to the failure callback or it will be timed out after `10000` milliseconds. Once `waitsFor()` is completed, the `runs` block checks that the failure callback is called, and using `expect(failureCallBack.mostRecentCall.args[0].xmlhttp.responseText, "A user with the same username is already registered ...")` ensures that the server sends the correct duplicate registration failure message to the failure callback. Finally, the spec checks that the success callback is not called for ensuring that the registration operation is not done because of the already registered user ID. This was all about the registration tests.

Testing the WeatherClient object

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

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

For testing the `WeatherClient` object, the same technique that we used in the `registerUser` test case is followed. I will leave this test for you as an exercise; you can get the full source code of the `WeatherClientSpec.js` file from the `Chapter 2` folder in the code bundle available from the book's website.

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 test suites. 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 in the browser the following URL to see the passing tests:

```
http://localhost:8080[or other Tomcat port]/weatherApplication/js/js-
test/jasmine/SpecRunner.html
```

Summary

In this chapter, you learned what Jasmine is and how to use it for testing synchronous JavaScript code. You also learned how to test asynchronous (Ajax) JavaScript code using Jasmine Spies and the `waitsFor/runs` mechanism. You also learned how to make fake Ajax testing using Jasmine. You learned the various matchers provided by the framework, and know how to load the HTML fixtures easily in your Jasmine tests. Finally, I explained how to apply all of these things for testing the weather application using Jasmine. In the next chapter, you will learn how to work with the YUI Test framework and how to use it for testing the weather application.