1 Unit Testing JavaScript Applications

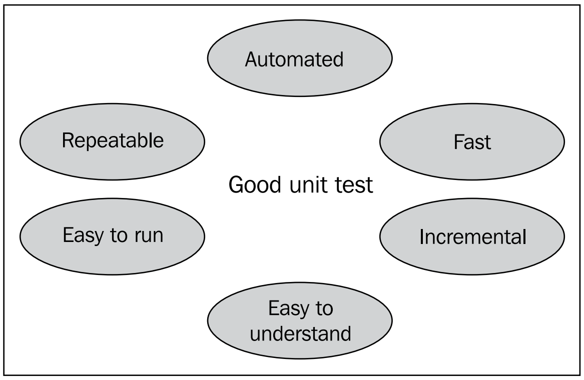
Before going into the details of unit testing JavaScript applications, we need to understand first what unit testing is and why we need to unit test our applications. This chapter also shows the complexities of testing JavaScript applications and why it is not as simple as desktop applications. Finally, the chapter illustrates the functionality and the JavaScript code of a sample weather application. We will unit test its JavaScript code in the following chapters of the book.

What unit testing is

Unit testing is not a new concept in the software development world. Thanks to Kent Beck, the concept of unit testing was introduced in **Smalltalk**, then the concept was transferred to many other programming languages, such as C, C++, and Java. The classical definition of unit testing is that it is a piece of code (usually a method) that invokes another piece of code and later checks the correctness of some assumptions.

The definition is technically correct; however, it does not show us how to make a really good unit test. In order to write a good unit test, we need to understand the requirements of a good unit test.

As shown in the following figure, a good unit test should be automated, repeatable, easy to understand, incremental, easy to run, and fast.



A good unit test should be automated and repeatable, which means that other team members can repeat running the application unit tests for every significant code change automatically. It should also be easy to understand so that other team members can understand what your test means and can continue adding more test cases or updating an existing test case. A good unit test should be incremental; this means that the unit test should be updated if a new relevant defect is detected in the code, which means that this defect will not happen again as long as this unit test is running periodically. Finally, a good unit test should be easy to run; it should run by executing a command or by clicking a button and should not take a long time for execution because fast unit tests can help in increasing the development team's productivity.

So let's go back to the definition and refine it. Unit testing is a piece of code (usually a method) that invokes another piece of code and checks the correctness of some assumptions later. Unit testing should be automated, repeatable, easy to understand, incremental, easy to run, and fast.

Why we need unit testing

Unit testing applications is not something nice to have. It is actually a mandatory activity for having a successful software solutions that can cope with different changes across time with high stability. There is no excuse to skip unit testing of applications even for projects with a tight schedule. The importance of unit testing may not appear in the early stages of the project; however, its advantages are visible in the middle and the final stages of the project, when the code gets complicated, more features are required, and more regression defects appear (defects that appear again after a major code change).

Without unit testing, the integration of the different components in the system becomes complicated. This complexity results from the tracing of the defects of not only the integration between the components but also each "buggy" component. This complicates the life of the developers by making them spend nights in the office in order to meet the schedule.

The number of new defects and the regression defects becomes unmanageable when the code base becomes complicated and unit testing is not available. The developer can resolve a specific defect and, after a set of code changes, this defect can happen again because there is no repeatable test case to ensure that the defect will not happen again.

Having more number of defects per lines of code affects the application's quality badly, and this means that more time has to be spent on testing the application. Bad quality applications have a longer test cycle for each project deployment (or phase), because they have a high probability of having more defects for every code change, which leads to more pressure on the project management, the project developers, and the project testers.

Having good unit testing can be a good reference for the system documentation because it contains the test scenarios of the system use cases. In addition to this, unit testing shows how the system APIs are used, which reflect the current design of the system. This means that unit testing is a powerful basis of code and design refactoring for having more enhancements in the system.

Having good unit testing minimizes the number of regression defects because in good unit testing the system has a repeatable number of test cases for every relevant defect. Having a continuous integration job that runs periodically on the application unit tests will ensure that these defects will not happen again, because if a specific defect appears again due to a change in the application code, then the developer will be notified to fix the defect and ensure that the test case of this defect passes successfully.

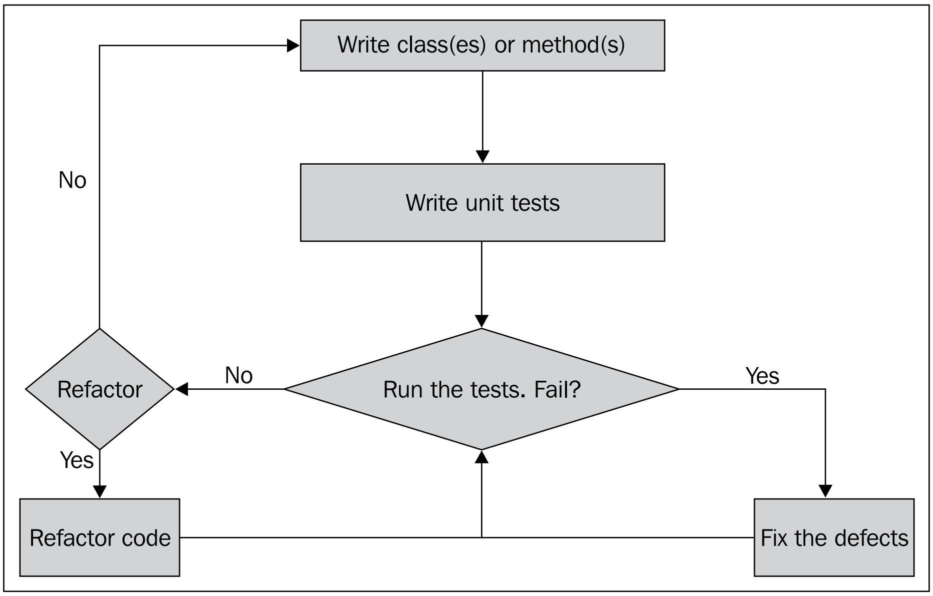
Continuous integration (**CI**) is a practice that ensures automating the build and the test process of the application. In continuous integration testing, the tests of the application source code run periodically (for example many times per day) in order to identify the application's potential problems and to reduce the integration time of the application components.

As a result of reducing the regression defects, having good unit testing reduces the test cycle for each phase (or system deployment). In addition to this, the application can have more and more features per iterations or phases peacefully without worrying if these features shall break an existing module that has good unit tests.

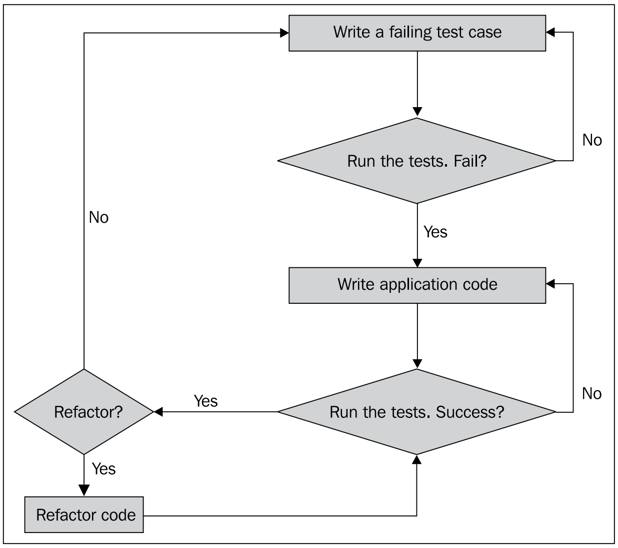
What Test-Driven Development (TDD) is

There are two known approaches in writing unit tests for applications.
The first approach prefers writing unit tests after writing the actual application code and this approach is called **traditional unit testing**. The second approach prefers writing unit tests before writing the actual application code, and this approach is called **Test-Driven Development** (**TDD**) or the **Test-First** approach.

As shown in the following figure, traditional unit testing is about writing the application code first. It can simply be a class or a method. After writing the piece of code, the unit tests, which test the functionality of the code, are written. Then the unit tests run. If the unit tests fail then the developer fixes the defects and runs the unit tests again. If the unit tests succeed then the developer can either refactor the code and run the tests again or continue to write the next piece of code and so on.



As shown in the following figure, TDD starts by writing a failing unit test to indicate that the functionality is missing. After writing the unit test, the unit test must be run to ensure that it fails. After that, the developer writes the application code that meets the unit test expectation. The unit test must be run again to ensure that it succeeds. If it fails then the developer fixes the bugs and if it succeeds the developer can either refactor the application code or continue writing the next test case.



TDD is a powerful technique, as it can give you more control on the application code and design; however, it is a double-edged sword because if it is done incorrectly, writing the tests can waste a lot of time and the schedule of the project can slip. Finally, either you are using TDD or traditional unit testing technique. Don't forget to make your tests automated, repeatable, easy to understand, incremental, easy to run, and fast.

Complexities in testing JavaScript applications

Testing JavaScript applications is complex and requires a lot of time and effort. Testing JavaScript applications requires the tester to test the application on different browsers (Internet Explorer, Firefox, Safari, Chrome, and so on). This is because the JavaScript code that runs on a specific browser will not necessarily work on another browser.

Testing existing JavaScript web applications (with many web pages) on new browsers that are not supported by the application code is not a flexible process. Supporting a new unsupported browser means allocating more time for testing the application again on this new browser and for the new/regression defects to be fixed by the developers. Let's see a simple Broken JavaScript example, which illustrates this idea. In this example, the user enters his/her name and then clicks on the **Welcome** button. After that the welcome message appears.

The following code snippet shows the broken JavaScript example:

```
<!DOCTYPE html>
<html>
<head>
  <title>Broken JavaScript Example</title>
  <script type=»text/javascript»>
    function welcome() {
      var userName = document.getElementById(«userName»).value;
      document.getElementById(«welcomeMessage»).innerText = «Welcome «
      + userName + «!»;
    }
  </script>
</head>
<body>
  <h1>Broken JavaScript Example</h1>

  <label>Please enter your name:</label>
  <input id=»userName» type=»text» /><br/>
  <input type=»button» onclick=»welcome()» value=»Welcome»></
   input><br/><br/>
  <div id=»welcomeMessage»/>

</body>
</html>
```

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.PacktPub.com`. If you purchased this book elsewhere, you can visit `http://www.PacktPub.com/support` and register to have the files e-mailed directly to you.

If you run the code shown in the previous code snippet, you will find that it works fine in Internet Explorer (IE) and Safari while it does not work in Firefox (to be more specific, this example works on Internet Explorer 8 and Safari 5.1, while it will not work on Firefox 3.6). The reason behind this problem is that the `innerText` property is not supported in Firefox. This is one of the hundreds of examples that show a code that works in a specific browser while it does not work in another one.

As a result of these complexities, testing JavaScript code requires a good unit testing tool, which provides mechanisms to overcome these complexities. The *good* JavaScript unit testing tool should be able to execute the test cases across all of the browsers, should have an easy setup, should have an easy configuration, and should be fast in executing the test cases.

Weather forecasting application

Now, let's move to the weather forecasting application. The weather forecasting application is a Java web application that allows the users to check the current weather in different cities in the world. The weather forecasting application contains both synchronous and asynchronous (Ajax) JavaScript code, which we will test in the later chapters of the book using the different JavaScript unit testing frameworks.

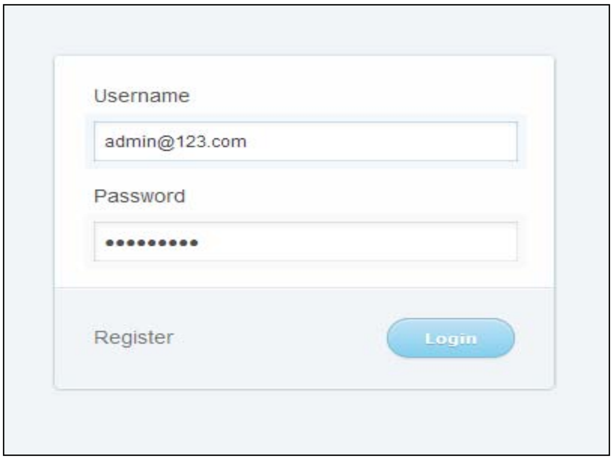
The weather forecasting application mainly contains three use cases:

- Log in to the application
- Register a user in the application
- Check the current weather in a specific city

The weather forecasting application is a Java web application. The server-side part of the application is written using Java servlets (`http://docs.oracle.com/javaee/6/tutorial/doc/bnafd.html`). If you are not familiar with Java servlets, do not worry. This book focuses only on JavaScript unit testing; all you need to know about these servlets is the functionality of each one of them, not the code behind it. The functionality of each application servlet will be explained during when the JavaScript code is explained, to show you the complete Ajax request life cycle with the server.

Another thing that needs to be mentioned is that the weather application pages are `.jsp` files; however, 99 percent of their code is pure HTML code that is easy to understand (the application pages code will be explained in detail in the next section).

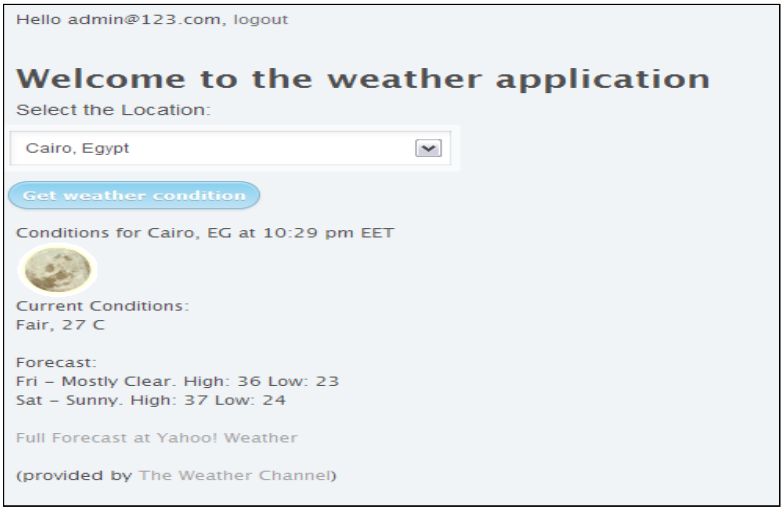
The first screen of the application is the login screen in which the user enters his username and password, as shown in the following screenshot:


Username
admin@123.com
Password
•••••••••

When the user clicks on the **Login** button, there is a JavaScript login client that ensures that the username and the password are entered correctly. If the username and the password are correct, they are submitted to the server, which validates them if the user is registered in the application. If the user is registered in the application then the user is redirected to the weather checking page; otherwise an error message appears to the user.

The username field must not be empty and has to be in a valid e-mail address format. The password field also must not be empty and has to contain at least one digit, one capital, one small character, and at least one special character. The password length has to be six characters or more.

In the weather checking page, the user can select one of the available cities from the combobox, then click on the **Get weather condition** button to get the current weather information of the selected city, as shown in the following screenshot:


Hello admin@123.com, logout

Welcome to the weather application

Select the Location:

Cairo, Egypt

Get weather condition

Conditions for Cairo, EG at 10:29 pm EET

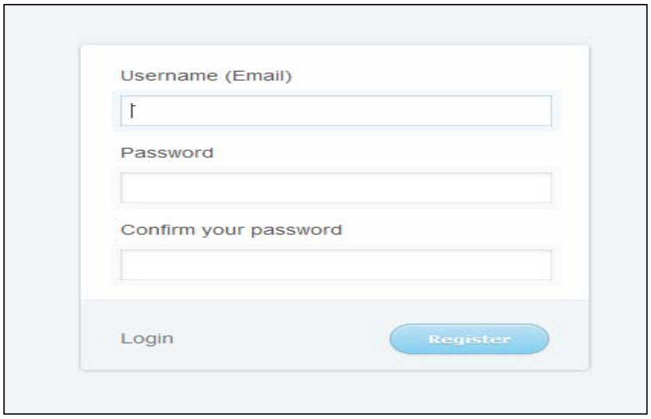
Current Conditions:
Fair, 27 C

Forecast:
Fri – Mostly Clear. High: 36 Low: 23
Sat – Sunny. High: 37 Low: 24

Full Forecast at Yahoo! Weather

(provided by The Weather Channel)

In the user registration page, the user can register in the application by entering his username and confirmed password, as shown in the following screenshot:



When the user clicks on the **Register** button, the registration client's JavaScript object ensures that the username and the passwords are entered correctly. The registration client uses the same rules of the login client in username and password validations. It also ensures that the confirmed password is the same as the entered password.

If the user's registration information is correct, the username and passwords are submitted to the server. The user information is registered in the system after performing server-side validations and checking that the user has not already registered in the application. If the user is already registered in the system then an error message appears to the user.

Exploring the application's HTML and JavaScript code

The following code snippet shows the HTML code of the login form in the `login.jsp` file. It is a simple form that has username and password fields with their labels, messages, a registration link, and a login button.

```
<form class="box login" action="/weatherApplication/LoginServlet"
method="post">
  <fieldset class="boxBody">
    <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"/>
  </fieldset>
  <div id="footer">
```

```
        <label><a href="register.jsp">Register</a></label>
        <input id="btnLogin" class="btnLogin" type="submit" value="Login"
         onclick="return validateLoginForm();"/>
      </div>
    </form>
```

When the **Login** button is clicked, the `validateLoginForm` JavaScript function is called. The following code snippet shows the `validateLoginForm` function in the `login.jsp` file:

```
    function validateLoginForm() {
      var loginClient = new weatherapp.LoginClient();

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

      return loginClient.validateLoginForm(loginForm);
    }
```

The `validateLoginForm` function calls the `LoginClient` JavaScript object that is responsible for validating the login form. It constructs a **JavaScript Object Notation** (**JSON**) object that includes the username, password, username message, and password message IDs, and then passes the constructed JSON object to the `validateLoginForm` function of the `LoginClient` object.

The weather application customizes a CSS3 based style from the blog CSS Junction:

`http://www.cssjunction.com/freebies/simple-login-from-html5css3-template-free/`

The following code snippet shows the `validateLoginForm` method of the `LoginClient` object in the `LoginClient.js` file. It validates that the username and the password fields are not empty and are compliant with the validation rules.

```
    if (typeof weatherapp == "undefined" || !weatherapp) {
      weatherapp = {};
    }

    weatherapp.LoginClient = function() {};

    weatherapp.LoginClient.prototype.validateLoginForm =
    function(loginForm) {

      if (this.validateEmptyFields(loginForm) &&
          this.validateUserName(loginForm) &&
            this.validatePassword(loginForm)) {

        return true;
      }

      return false;
    };
```

One of the recommended JavaScript's best practices is to use namespaces; the application defines a JavaScript namespace in order to avoid collisions with other JavaScript objects of similar names. The following code defines a `weatherapp` namespace if it is not already defined:

```
if (typeof weatherapp == "undefined" || !weatherapp) {
  weatherapp = {};
}
```

The following code snippet shows the `validateEmptyFields` method of the `LoginClient` object in the `LoginClient.js` file. It validates that the username and the password fields are not empty and if any of these fields are empty, an error message appears:

```
weatherapp.LoginClient.prototype.validateEmptyFields =
function(loginForm) {
  var passwordMessageID = loginForm.passwordMessage;
  var userNameMessageID = loginForm.userNameMessage;

  var passwordFieldID = loginForm.passwordField;
  var userNameFieldID = loginForm.userNameField;

  document.getElementById(passwordMessageID).innerHTML = "";
  document.getElementById(userNameMessageID).innerHTML = "";

  if (! document.getElementById(userNameFieldID).value) {
    document.getElementById(userNameMessageID).innerHTML = "(field is
    required)";

    return false;
  }
```

```
      if (! document.getElementById(passwordFieldID).value) {
        document.getElementById(passwordMessageID).innerHTML = "(field is
        required)";

      return false;
      }

      return true;
    };
```

The following code snippet shows the `validateUserName` method of the `LoginClient` object in the `LoginClient.js` file. It validates that the username is in the form of a valid e-mail:

```
    weatherapp.LoginClient.prototype.validateUserName =
    function(loginForm) {

      // the username must be an email...
      var userNameMessageID = loginForm.userNameMessage;
      var userNameFieldID = loginForm.userNameField;

    var userNameRegex = /^[_A-Za-z0-9-]+(\.[_A-Za-z0-9-]+)*@
    [A-Za-z0-9]+(\.[A-Za-z0-9]+)*(\.[A-Za-z]{2,})$/;
      var userName = document.getElementById(userNameFieldID).value;

    if(! userNameRegex.test(userName)) {
        document.getElementById(userNameMessageID).innerHTML = "(format is
        invalid)";

    return false;
    }

        return true;
    };
```

Using the regular expression `/^[_A-Za-z0-9-]+(\.[_A-Za-z0-9-]+)*@[A-Za-z0-9]+(\.[A-Za-z0-9]+)*(\.[A-Za-z]{2,})$/`, the username is validated against a valid e-mail form. If the username is not in a valid e-mail form then an error message appears in the username message span.

The following code snippet shows the `validatePassword` method of the `LoginClient` object in the `LoginClient.js` file. It validates if the password has at least one digit, one capital character, one small character, at least one special character, and also if it contains six characters or more:

```
    weatherapp.LoginClient.prototype.validatePassword =
function(loginForm) {

  // the password contains at least one digit, one capital and small
character
  // and at least one special character, and 6 characters or more...
  var passwordMessageID = loginForm.passwordMessage;
  var passwordFieldID = loginForm.passwordField;

  var passwordRegex = /((?=.*\d)(?=.*[a-z])(?=.*[A-Z])(?=.*[@#$%]).
{6,20})/;
  var password = document.getElementById(passwordFieldID).value;

  if (! (passwordRegex.test(password) && password.length >= 6)) {
    document.getElementById(passwordMessageID).innerHTML = "(format is
    invalid)";

    return false;
  }

  return true;
};
```

If the password is not compliant with the mentioned rules then an error message appears in the password message span.

If the username and the password fields pass the JavaScript validation rules, the login form submits its content to `LoginServlet`, which makes another server-side validation and then redirects the user to the weather checking page if the validation goes OK.

> It is very important not to rely on the JavaScript client-side validation only, because JavaScript can be disabled from the browser. So it is a must to always make a server-side validation besides the client-side validation.

The following code snippet shows the weather checking form of the weather application located in the `welcome.jsp` file. It contains a combobox filled with the Yahoo! Weather Where On Earth IDs (the WOEID is a unique reference identifier assigned by Yahoo! to identify any place on Earth) of different cities in the world.

```
<h1>Welcome to the weather application</h1>
<FORM method="post">
  <label class="label" for="postalCode">Select the Location: </label>
  <select id="w" class="selectField">
    <option value="1521894">Cairo, Egypt</option>
```

```
        <option value="906057">Stockholm, Sweden</option>
        <option value="551801">Vienna, Austria</option>
        <option value="766273">Madrid, Spain</option>
        <option value="615702">Paris, France</option>
        <option value="2459115">New York, USA</option>
        <option value="418440">Lima, Peru</option>
      </select>

      <input type="button" class="button" onclick="invokeWeatherClient();"
                 value="Get weather condition"/>
      <br/><br/>

      <div id="weatherInformation" class="weatherPanel">
      </div>
    </FORM>
```

When the **Get weather condition** button is clicked, the `invokeWeatherClient` function is called. The following code snippet shows the `invokeWeatherClient` function code in the `welcome.jsp` file:

```
    function invokeWeatherClient() {
      var weatherClient = new weatherapp.WeatherClient();
      var location = document.getElementById("w").value;

      weatherClient.getWeatherCondition({
        'location': location,
        'resultDivID': 'weatherInformation'
      },
    weatherClient.displayWeatherInformation,
    weatherClient.handleWeatherInfoError);
    }
```

The `invokeWeatherClient` function calls the `getWeatherCondition` method of the `WeatherClient` object. The first parameter of the `getWeatherCondition` method is the `weatherForm` object, which is a JSON object containing the location WOEID and the ID of the `DIV` element that receives the weather information HTML result of the Yahoo! Weather Representational State Transfer (REST) service. The second parameter represents the first callback, which is the `displayWeatherInformation` method that is called if the `getWeatherCondition` call succeeds. The last parameter represents the second callback, which is the `handleWeatherInfoError` method that is called if the `getWeatherCondition` call fails.

The following code snippet shows `getWeatherCondition` of the `WeatherClient` object in the `WeatherClient.js` file that sends an Ajax request to `WeatherProxyServlet` with the `w` parameter that represents the **WOEID**.

`WeatherProxyServlet` interacts with the Yahoo! Weather REST service in order to fetch the current weather information:

```
if (typeof weatherapp == "undefined" || !weatherapp) {
  weatherapp = {};
}

weatherapp.WeatherClient = function() {};
weatherapp.WeatherClient.xmlhttp;
weatherapp.WeatherClient.weatherForm;
weatherapp.WeatherClient.endpointURL = "";

weatherapp.WeatherClient.prototype.getWeatherCondition =
function(weatherForm, successCallBack, failureCallBack) {

  if (window.XMLHttpRequest) {
    this.xmlhttp = new XMLHttpRequest();
  } else {
    this.xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
  }

  var successCallBackLocal = successCallBack;
  var failureCallBackLocal = failureCallBack;
  var weatherClientLocal = this;

  this.xmlhttp.onreadystatechange = function() {
weatherClientLocal.weatherInformationReady(successCallBackLocal,
failureCallBackLocal);
  };

  this.weatherForm = weatherForm;

  if (typeof this.endpointURL == "undefined") {
    this.endpointURL = "";
  }

  this.xmlhttp.open("GET",
          this.endpointURL +
          "/weatherApplication/WeatherProxyServlet?w=" + weatherForm.
            location + "&preventCache=" + new Date().getTime(),
          true);

  this.xmlhttp.send();
};
```

```
weatherapp.WeatherClient.prototype.weatherInformationReady =
function(successCallBack, failureCallBack) {
  if (this.xmlhttp.readyState != 4) {
    return;
  }

  if (this.xmlhttp.status != 200)  {
    failureCallBack(this);

return;
      }

  if (this.xmlhttp.readyState == 4 && this.xmlhttp.status == 200) {
    successCallBack(this);
      }
};

weatherapp.WeatherClient.prototype.displayWeatherInformation =
function(weatherClient) {
  var resultDivID = weatherClient.weatherForm.resultDivID;

document.getElementById(resultDivID).innerHTML = weatherClient.
xmlhttp.responseText;
};

weatherapp.WeatherClient.prototype.handleWeatherInfoError =
function(weatherClient) {
  var resultDivID = weatherClient.weatherForm.resultDivID;

  alert ("Error: " + weatherClient.xmlhttp.responseText);
document.getElementById(resultDivID).innerHTML = "Error: " +
weatherClient.xmlhttp.responseText;
};
```

The `getWeatherCondition` method first creates an XML HTTP request object using `new XMLHttpRequest()` in case of IE7+, Firefox, Chrome, and Opera. In the case of IE5 and IE6, the XML HTTP request object is created using an ActiveX object `new ActiveXObject("Microsoft.XMLHTTP")`.

The `getWeatherCondition` method then registers both, the success callback (`successCallBack`) and the failure callback (`failureCallBack`) using the `weatherInformationReady` method that is called for every Ajax `readyState` change.

Finally, the getWeatherCondition method sends an asynchronous Ajax request to WeatherProxyServlet. When the Ajax response comes from the server and the operation is done successfully then the success callback is called, which is the displayWeatherInformation method. In the case of operation failure (which can happen, for example, if the passed WOEID is invalid or the Yahoo! Weather service is down), the failure callback is called, which is the handleWeatherInfoError method.

The displayWeatherInformation method displays the returned weather information HTML result from WeatherProxyServlet (which fetches the weather information from the Yahoo! Weather REST service) in the weatherInformation div element while the handleWeatherInfoError method displays the error message in the same div element and also displays an alert with the error message.

It is assumed that you are familiar with Ajax programming. If you are not familiar with Ajax programming, it is recommended to check the following introductory Ajax tutorial on w3schools:

`http://www.w3schools.com/ajax/default.asp`

In order to prevent IE from caching Ajax GET requests, a random parameter is appended using `new Date().getTime()`. In many JavaScript libraries, this can be handled through the framework APIs. For example, in Dojo the preventCache attribute of the dojo.xhrGet API can be used to prevent the IE Ajax GET caching.

The following code snippet shows the HTML code of the registration form in the register.jsp file. It consists of a username and two password fields with their corresponding labels, messages, login link, and a register button:

```
<form class="box register" method="post">
  <fieldset class="boxBody">

<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"/>

  </fieldset>
  <div id="footer">
    <label><a href="login.jsp">Login</a></label>
```

```
<input id="btnRegister" class="btnLogin" type="button"
value="Register" onclick="registerUser();" />
  </div>

</form>
```

When the **Register** button is clicked, the `registerUser` JavaScript function is called. The following code snippet shows the code of the `registerUser` function in the `register.jsp` file:

```
function registerUser() {
  var registrationClient = new weatherapp.RegistrationClient();

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

  if (registrationClient.validateRegistrationForm(registrationForm)) {

    registrationClient.registerUser(registrationForm,
                 registrationClient.displaySuccessMessage,
                     registrationClient.handleRegistrationError);
  }
}
```

The `registerUser` function is calling the `RegistrationClient` JavaScript object that is responsible for validating and submitting the registration form using Ajax to `RegistrationServlet`. `registerUser` constructs the `registrationForm` JSON object, which includes the username, password1, password2, username message, and password1 message IDs, and then passes the object to the `validateRegistrationForm` method of the `RegistrationClient` object.

If the validation passes, it calls the `registerUser` method of the `RegistrationClient` object. The first parameter of the `registerUser` method is the `registrationForm` JSON object. The second parameter is the success callback, which is the `displaySuccessMessage` method, while the last parameter is the failure callback, which is the `handleRegistrationError` method.

The following code snippet shows the code of the `validateRegistrationForm` method of the `RegistrationClient` object in the `RegistrationClient.js` file. It uses the validation methods of `LoginClient` in order to validate the empty username and password fields, and to validate if the username and the password fields conform to the validation rules. In addition to this, the `validateRegistrationForm` method validates if the two entered passwords are identical:

```
if (typeof weatherapp == "undefined" || !weatherapp) {
  weatherapp = {};
}

weatherapp.RegistrationClient = function() {};
weatherapp.RegistrationClient.xmlhttp;
weatherapp.RegistrationClient.endpointURL = "";

weatherapp.RegistrationClient.prototype.validateRegistrationForm =
function(registrationForm) {
  var userNameMessage = registrationForm.userNameMessage;
  var passwordMessage1 = registrationForm.passwordMessage1;

  var userNameField = registrationForm.userNameField;
  var passwordField1 = registrationForm.passwordField1;
  var passwordField2 = registrationForm.passwordField2;

  var password1 = document.getElementById(passwordField1).value;
  var password2 = document.getElementById(passwordField2).value;

  // Empty messages ...
  document.getElementById(userNameMessage).innerHTML = "";
  document.getElementById(passwordMessage1).innerHTML = "";

  // create the loginClient object in order to validate fields ...
  var loginClient = new weatherapp.LoginClient();

  var loginForm = {};

  loginForm.userNameField = userNameField;
  loginForm.userNameMessage = userNameMessage;
  loginForm.passwordField = passwordField1;
  loginForm.passwordMessage = passwordMessage1;

  // validate empty username and password fields.
  if (! loginClient.validateEmptyFields(loginForm)) {
    return false;
  }

  // validate that password fields have the same value...
```

```
    if (password1 != password2) {
  document.getElementById(passwordMessage1).innerHTML = "(Passwords must
  be identical)";

      return false;
    }

    // check if the username is correct...
    if (! loginClient.validateUserName(loginForm) ) {
  document.getElementById(userNameMessage).innerHTML = "(format is
  invalid)";

      return false;
    }

    // check if the password is correct...
    if (! loginClient.validatePassword(loginForm) ) {
  document.getElementById(passwordMessage1).innerHTML = "(format is
  invalid)";

      return false;
    }

    return true;
  };
```

The following code snippet shows the `registerUser` method code of the `RegistrationClient` object in the `RegistrationClient.js` file. It creates an Ajax POST request with the username and the passwords' (original password and confirmed password) data and sends them asynchronously to `RegistrationServlet`.

```
  weatherapp.RegistrationClient.prototype.registerUser =
  function(registrationForm, successCallBack, failureCallBack) {
    var userNameField = registrationForm.userNameField;
    var passwordField1 = registrationForm.passwordField1;
    var passwordField2 = registrationForm.passwordField2;

    var userName = document.getElementById(userNameField).value;
    var password1 = document.getElementById(passwordField1).value;
    var password2 = document.getElementById(passwordField2).value;

    if (window.XMLHttpRequest) {
      this.xmlhttp = new XMLHttpRequest();
    } else {
      this.xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
    }
    var successCallBackLocal = successCallBack;
    var failureCallBackLocal = failureCallBack;
    var registrationClientLocal = this;

    this.xmlhttp.onreadystatechange = function() {
  registrationClientLocal.registrationReady(successCallBackLocal,
  failureCallBackLocal);
    };

    if (typeof this.endpointURL == "undefined") {
      this.endpointURL = "";
    }

    this.xmlhttp.open("POST",
              this.endpointURL +
              "/weatherApplication/RegistrationServlet",
               true);

  this.xmlhttp.setRequestHeader("Content-type","application/x-www-form-
  urlencoded");

    this.xmlhttp.send(userNameField + "=" + userName + "&" +
              passwordField1 + "=" + password1 + "&" +
              passwordField2 + "=" + password2);
  };

  weatherapp.RegistrationClient.prototype.registrationReady =
  function(successCallBack, failureCallBack) {
    if (this.xmlhttp.readyState != 4) {
      return;
    }

    if (this.xmlhttp.status != 200)  {
      failureCallBack(this);
  return;
  }

    if (this.xmlhttp.readyState == 4 && this.xmlhttp.status == 200) {
      successCallBack(this);
        }
  };

  weatherapp.RegistrationClient.prototype.displaySuccessMessage =
  function(registrationClient) {
    alert("User registration went successfully ...");
  };

  weatherapp.RegistrationClient.prototype.handleRegistrationError =
  function(registrationClient) {
    alert(registrationClient.xmlhttp.responseText);
  };
```

`RegistrationServlet` validates the user data and ensures that the user did not already register in the application. When the Ajax response comes from the server, and the registration operation is completed successfully, the `displaySuccessMessage` method is called. If the registration operation failed (for example, if the user ID is already registered in the application), the `handleRegistrationError` method is called. Both the `displaySuccessMessage` and the `handleRegistrationError` methods display alerts to show the success and the failure registration messages.

Running the weather application

In order to run the weather application, you first need to download the `weatherApplication.war` file from the book's website (`www.packtpub.com`). Then you need to deploy the WAR file on Apache Tomcat 6. In order to install Apache Tomcat 6, you need to download it from `http://tomcat.apache.org/download-60.cgi`. Apache Tomcat 6.0 requires the Java 2 Standard Edition Runtime Environment (JRE) Version 5.0 or later.

In order to install JRE, you need to download and install the J2SE Runtime Environment as follows:

1. Download the JRE, release Version 5.0 or later, from `http://www.oracle.com/technetwork/java/javase/downloads/index.html`.
2. Install the JRE according to the instructions included with the release.
3. Set an environment variable named `JRE_HOME` to the pathname of the directory in which you installed the JRE, for example, `c:\jre5.0` or `/usr/local/java/jre5.0`.

After you download the binary distribution of Apache Tomcat 6, you need to unpack the distribution in a suitable location on the hard disk. After this, you need to define the `CATALINA_HOME` environment variable, which refers to the location of the Tomcat distribution.

Now, you can start Apache Tomcat 6 by executing the following command on Windows:

```
$CATALINA_HOME\bin\startup.bat
```

Start as while in Unix, you can execute the following command:

```
$CATALINA_HOME/bin/startup.sh
```

In order to make sure that the Apache Tomcat 6 starts correctly, you need to type the following URL in the browser:

```
http://localhost:8080/
```

After making sure that the Apache Tomcat 6 is running correctly, you can stop it by executing the following command on Windows:

```
$CATALINA_HOME\bin\shutdown.bat
```

Start as while in Unix, you can execute the following command:

```
$CATALINA_HOME/bin/shutdown.sh
```

Now, we come to the step of the weather application deployment where you need to get the `weatherApplication.war` file from the book resources. After getting the file, copy the WAR file to the `$CATALINA_HOME\webapps` folder, then start the Apache Tomcat 6 again.

In order to access the weather application, you can access it using the following URL:

```
http://localhost:8080/weatherApplication/login.jsp
```

For the sake of simplicity, there is a predefined username and password that can be used to access the weather application; the username is `admin@123.com` and the password is `Admin@123`. Another thing that has to be mentioned is that the registered users are not stored in a database; they are stored in the application scope, which means they will be available as long as the application is not restarted.

Summary

In this chapter, you learned what unit testing is, the requirements of a good unit test, and why we need unit testing. You got to know the difference between the Test-Driven Development and the traditional unit testing. In the JavaScript world, you understood the complexities of testing JavaScript code, and the requirements of good JavaScript unit testing tools. At the end of this chapter, I explored with you the weather web application use cases and its JavaScript code in detail, which we will unit test in the later chapters. In the next chapter, you will learn how to work with the Jasmine framework and how to use it for testing the weather application.