

# 4

## User Interface: Basic Elements

In this chapter, we will get to the actual hands-on work. We will develop samples based on DWR, which show how to dynamically change the common user interface elements such as tables and lists as well as field completion. We also make a dynamic user interface skeleton for our samples that will hold all the samples in this book.

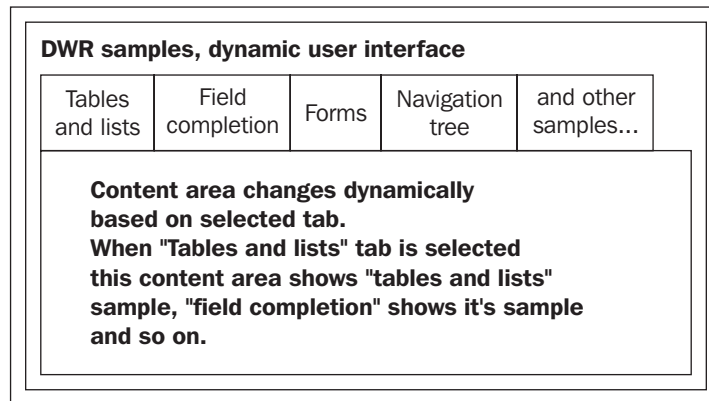
The section on dynamic user interfaces shows how to get started with a DWR application, and it presents a user interface skeleton that will be used to hold the tables and lists sample, and the field completion (aka. autosuggest/autocomplete) sample. Samples in the following chapter will use the same user interface skeleton, with the exception of the sample applications in Chapter 7.

The following are the sections in this chapter:

- Creating a Dynamic User Interface—starts with creating a web project and a basis for samples mentioned in this chapter
- Implementing Tables and Lists—shows us how to use DWR with them
- Implementing Field Completion—has a sample for typical field completion

### Creating a Dynamic User Interface

The idea behind a dynamic user interface is to have a common "framework" for all samples. We will create a new web application and then add new features to the application as we go on. The user interface will look something like the following figure:

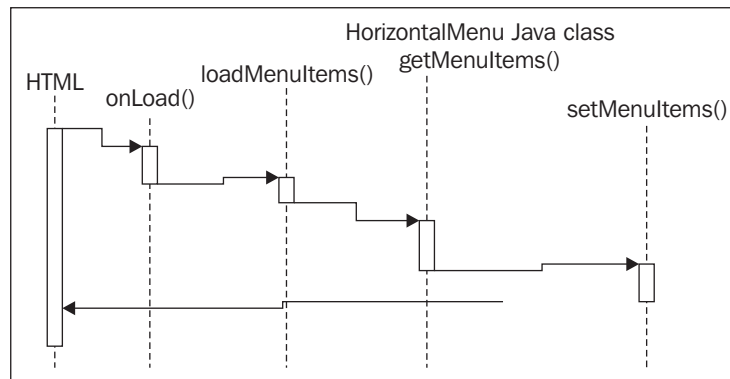


The user interface has three main areas: the title/logo that is static, the tabs that are dynamic, and the content area that shows the actual content.

The idea behind this implementation is to use DWR functionality to generate tabs and to get content for the tab pages. The tabbed user interface is created using a CSS template from the Dynamic Drive CSS Library (<http://dynamicdrive.com/style/csslibrary/item/css-tabs-menu>). Tabs are read from a properties file, so it is possible to dynamically add new tabs to the web page. The following screenshot shows the user interface.



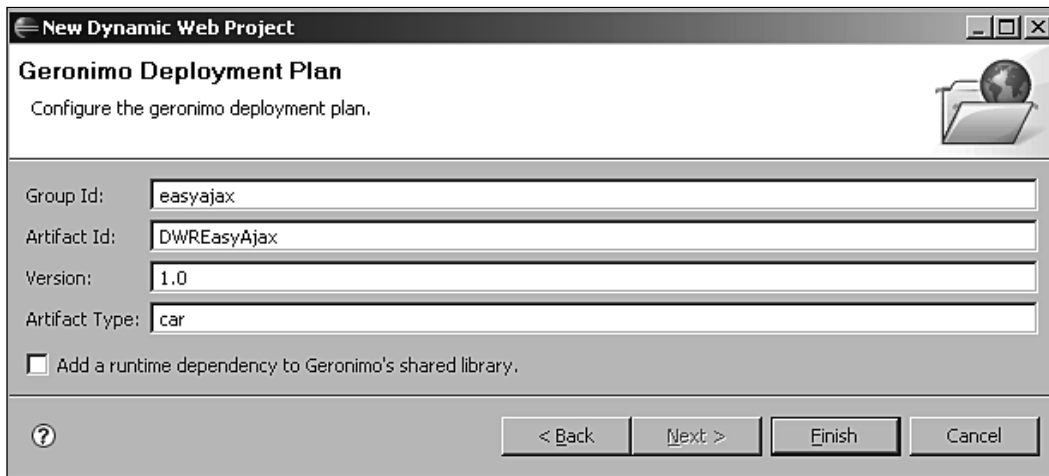
The following sequence diagram shows the application flow from the logical perspective. Because of the built-in DWR features we don't need to worry very much about how asynchronous AJAX "stuff" works. This is, of course, a Good Thing.



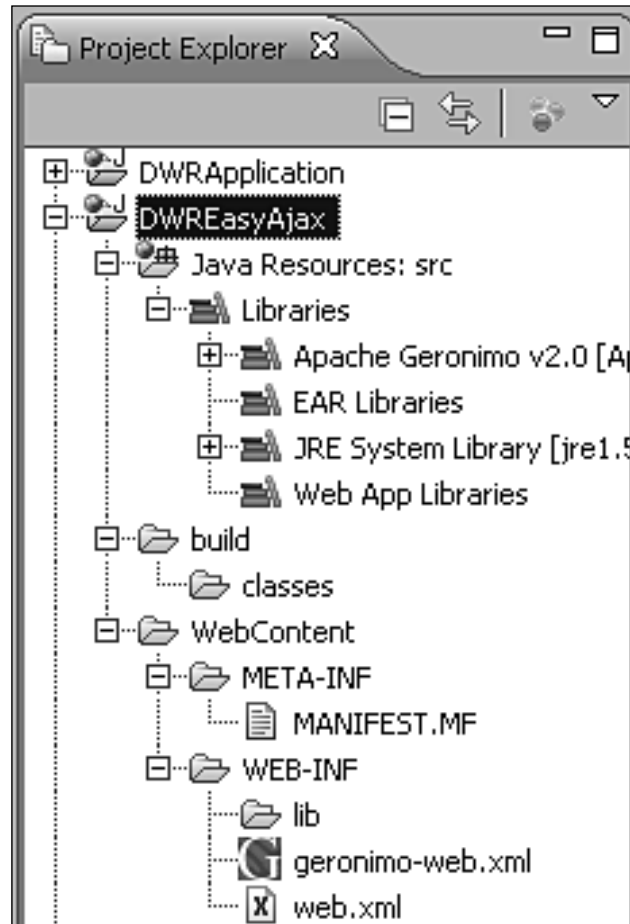
Now we will develop the application using the Eclipse IDE and the Geronimo test environment that we set up in the previous chapter.

## Creating a New Web Project

1. First, we will create a new web project. Using the Eclipse IDE we do the following: select the menu **File | New | Dynamic Web Project**.
2. This opens the **New Dynamic Web Project** dialog; enter the project name **DWREasyAjax** and click **Next**, and accept the defaults on all the pages till the last page, where **Geronimo Deployment Plan** is created as shown in the following screenshot:

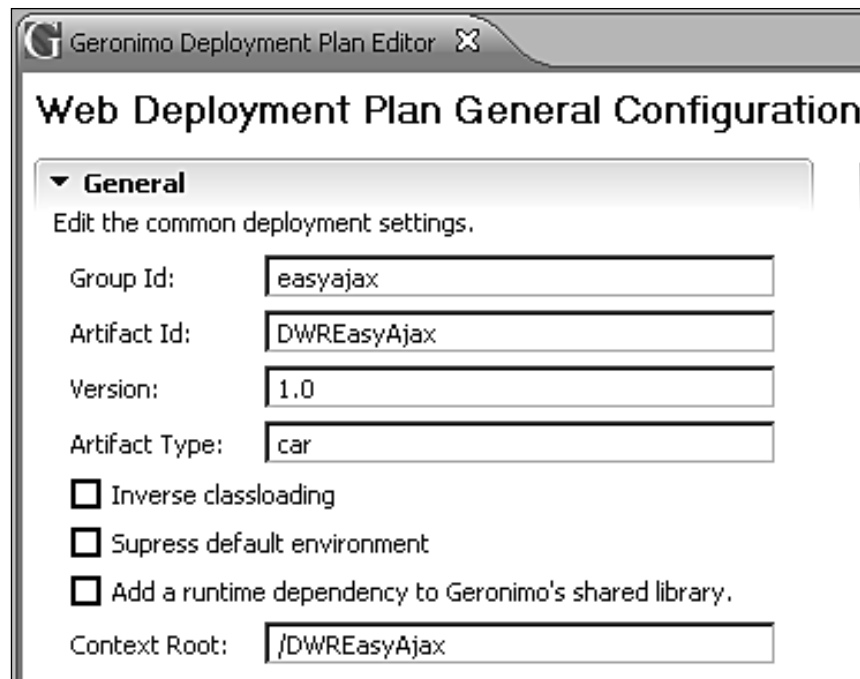


3. Enter **easyajax** as **Group Id** and **DWREasyAjax** as **Artifact Id**. On clicking **Finish**, Eclipse creates a new web project. The following screen shot shows the generated project and the directory hierarchy.



4. Before starting to do anything else, we need to copy DWR to our web application. All DWR functionality is present in the `dwr.jar` file, and we just copy that to the **WEB-INF | lib** directory.

A couple of files are noteworthy: `web.xml` and `geronimo-web.xml`. The latter is generated for the Geronimo application server, and we can leave it as it is. Eclipse has an editor to show the contents of `geronimo-web.xml` when we double-click the file.



## Configuring the Web Application

The context root is worth noting (visible in the screenshot above). We will need it when we test the application.

The other XML file, `web.xml`, is very important as we all know. This XML will hold the DWR servlet definition and other possible initialization parameters. The following code shows the full contents of the `web.xml` file that we will use:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.
sun.com/xml/ns/javaee/web-app_2_5.xsd"
  id="WebApp_ID" version="2.5">
  <display-name>DWREasyAjax</display-name>
  <servlet>
    <display-name>DWR Servlet</display-name>
    <servlet-name>dwr-invoker</servlet-name>
    <servlet-class>
      org.directwebremoting.servlet.DwrServlet
```

```
</servlet-class>
<init-param>
  <param-name>debug</param-name>
  <param-value>>true</param-value>
</init-param>
</servlet>

<servlet-mapping>
  <servlet-name>dwr-invoker</servlet-name>
  <url-pattern>/dwr/*</url-pattern>
</servlet-mapping>

<welcome-file-list>
  <welcome-file>index.html</welcome-file>
  <welcome-file>index.htm</welcome-file>
  <welcome-file>index.jsp</welcome-file>
  <welcome-file>default.html</welcome-file>
  <welcome-file>default.htm</welcome-file>
  <welcome-file>default.jsp</welcome-file>
</welcome-file-list>
</web-app>
```

We have already seen the `servlet` definition in Chapter 3, in the section on configuration. We use the same `debug-init` parameter here. Servlet mapping is the commonly used `/dwr/*`.

We remember that DWR cannot function without the `dwr.xml` configuration file. So we need to create the configuration file. We use Eclipse to create a new XML file in the **WEB-INF** directory. The following is required for the user interface skeleton. It already includes the `allow` element for our DWR based menu.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE dwr PUBLIC
  "-//GetAhead Limited//DTD Direct Web Remoting 2.0//EN"
  "http://getahead.org/dwr/dwr20.dtd">
<dwr>
  <allow>
    <create creator="new" javascript="HorizontalMenu">
      <param name="class" value="samples.HorizontalMenu" />
    </create>
  </allow>
</dwr>
```

In the `allow` element, there is a creator for the horizontal menu Java class that we are going to implement here. The creator that we use here is the `new` creator, which means that DWR will use an empty constructor to create Java objects for clients. The parameter named `class` holds the fully qualified class name.

## Developing the Web Application

Since we have already defined the name of the Java class that will be used for creating the menu, the next thing we do is implement it. The idea behind the `HorizontalMenu` class is that it is used to read a properties file that holds the menus that are going to be on the web page.

We add properties to a file named `dwrapplication.properties`, and we create it in the same `samples-package` as the `HorizontalMenu`-class. The properties file for the menu items is as follows:

```
menu.1=Tables and lists,TablesAndLists
menu.2=Field completion,FieldCompletion
```

The syntax for the menu property is that it contains two elements separated by a comma. The first element is the name of the menu item. This is visible to user. The second is the name of HTML template file that will hold the page content of the menu item.

The class contains just one method, which is used from JavaScript and via DWR to retrieve the menu items. The full class implementation is shown here:

```
package samples;

import java.io.IOException;
import java.io.InputStream;
import java.util.List;
import java.util.Properties;
import java.util.Vector;

public class HorizontalMenu {
    public HorizontalMenu() {

    }

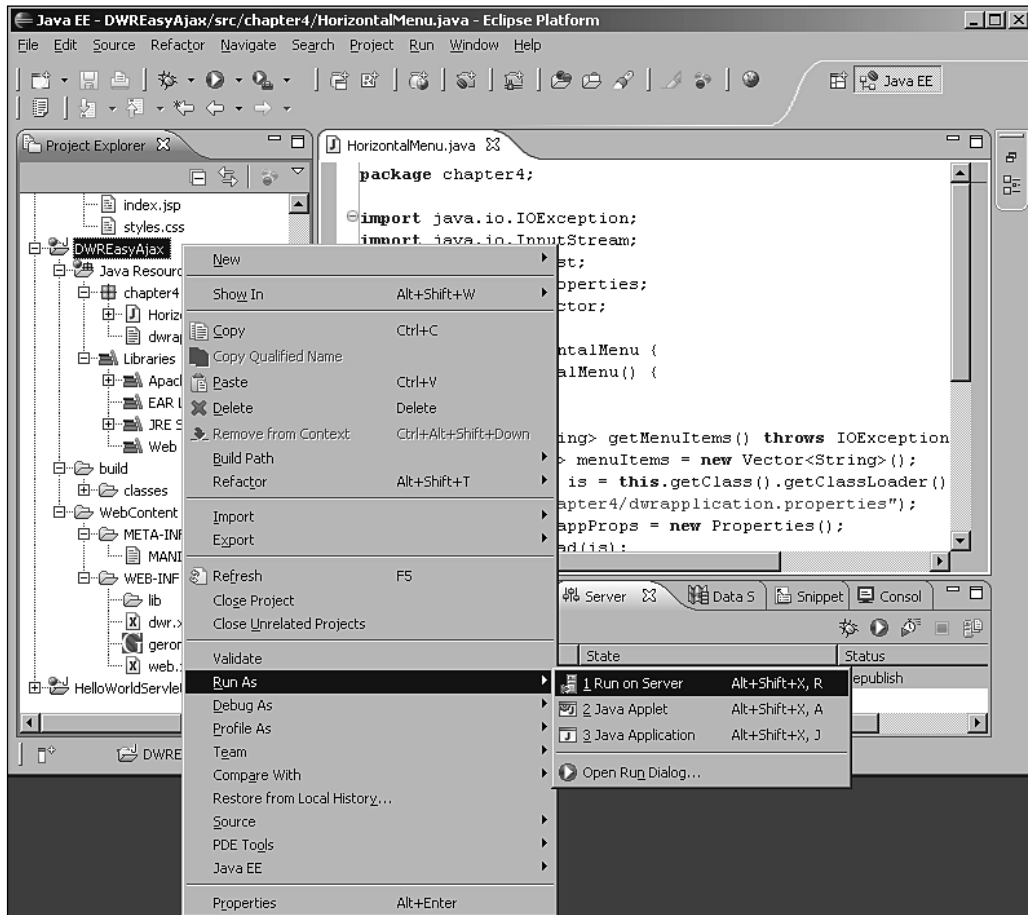
    public List<String> getMenuItems() throws IOException {
        List<String> menuItems = new Vector<String>();
        InputStream is = this.getClass().getClassLoader().
getResourceAsStream(
            "samples/dwrapplication.properties");
        Properties appProps = new Properties();
        appProps.load(is);
        is.close();
        for (int menuCount = 1; true; menuCount++) {
            String menuItem = appProps.getProperty("menu." + menuCount);
            if (menuItem == null) {
                break;
            }
            menuItems.add(menuItem);
        }
        return menuItems;
    }
}
```

The implementation is straightforward. The `getMenuItems()` method loads properties using the `ClassLoader.getResourceAsStream()` method, which searches the class path for the specified resource. Then, after loading properties, a `for` loop is used to loop through menu items and then a `List` of `String`-objects is returned to the client. The client is the JavaScript callback function that we will see later. DWR automatically converts the `List` of `String` objects to JavaScript arrays, so we don't have to worry about that.

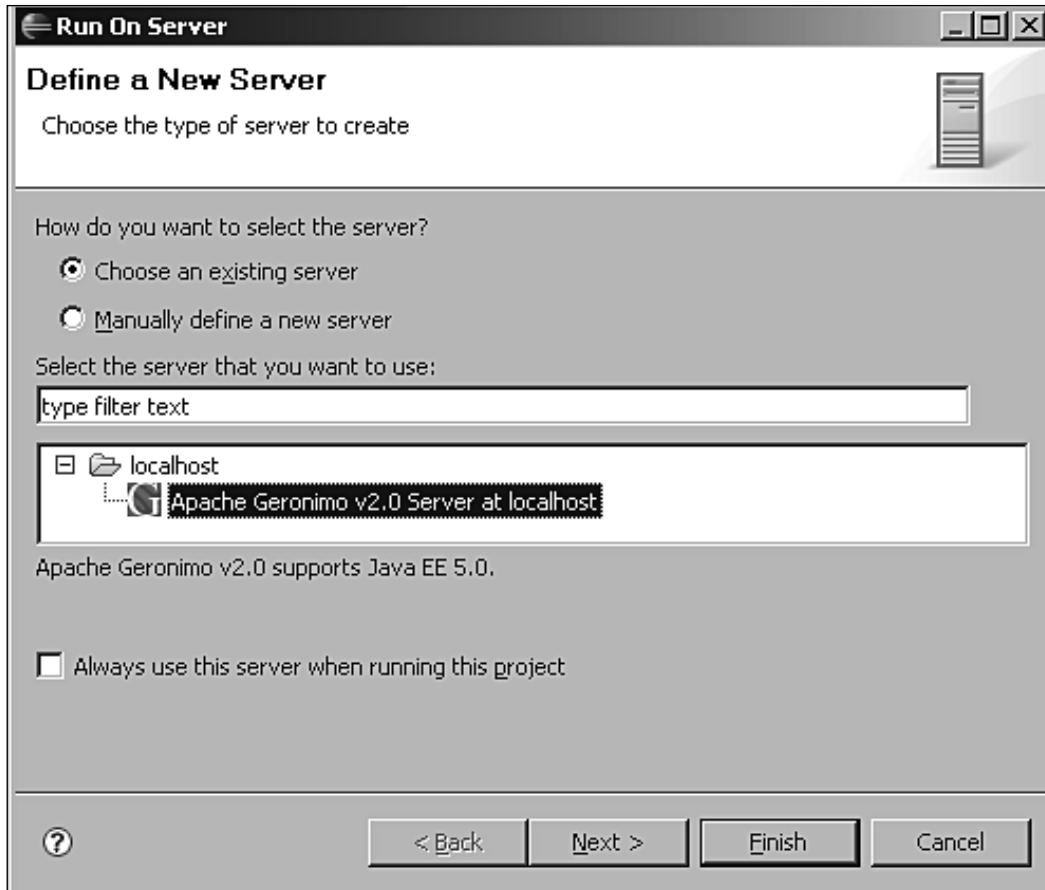
## Testing the Web Application

We haven't completed any client-side code now, but let's test the code anyway. Testing uses the Geronimo test environment.

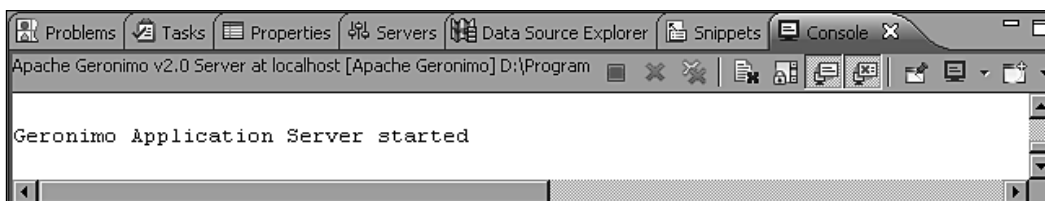
1. The **Project** context menu has the **Run As** menu that we use to test the application as shown in the following screenshot:



2. **Run on Server** opens a wizard to define a new server runtime. The following screenshot shows that the Geronimo test environment has already been set up, and we just click **Finish** to run the application. If the test environment is not set up, we can manually define a new one in this dialog:



3. After we click **Finish**, Eclipse starts the Geronimo test environment and our application with it. When the server starts, the **Console** tab in Eclipse informs us that it's been started.



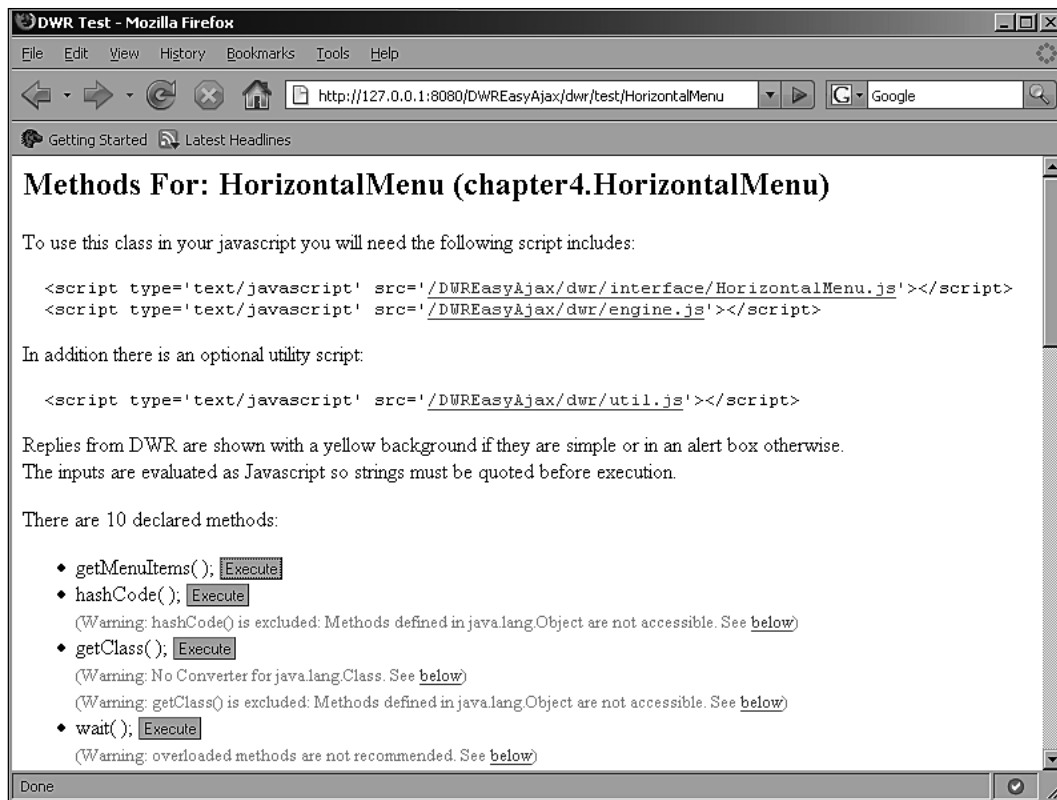
The **Servers** tab shows that the server is started and all the code has been synchronized, that is, the code is the most recent (Synchronization happens whenever we save changes on some deployed file.) The **Servers** tab also has a list of deployed applications under the server. Just the one application that we are testing here is visible in the **Servers** tab.



Now comes the interesting part – what are we going to test if we haven't really implemented anything? If we take a look at the `web.xml` file, we will find that we have defined one initialization parameter. The `Debug` parameter is true, which means that DWR generates test pages for our remoted Java classes. We just point the browser (Firefox in our case) to the URL `http://127.0.0.1:8080/DWREasyAjax/dwr` and the following page opens up:



This page will show a list of all the classes that we allow to be remoted. When we click the class name, a test page opens as in the following screenshot:



This is an interesting page. We see all the allowed methods, in this case, all public class methods since we didn't specifically include or exclude anything. The most important ones are the script elements, which we need to include in our HTML pages. DWR does not automatically know what we want in our web pages, so we must add the script includes in each page where we are using DWR and a remoted functionality.

Then there is the possibility of testing remoted methods. When we test our own method, `getMenuItems()`, we see a response in an alert box:



The array in the alert box in the screenshot is the JavaScript array that DWR returns from our method.

## Developing Web Pages

The next step is to add the web pages. Note that we can leave the test environment running. Whenever we change the application code, it is automatically published to test the environment, so we don't need to stop and start the server each time we make some changes and want to test the application.

The CSS style sheet is from the Dynamic Drive CSS Library. The file is named `styles.css`, and it is in the **WebContent** directory in Eclipse IDE. The CSS code is as shown:

```
/*URL: http://www.dynamicdrive.com/style/ */  
  
.basictab{  
padding: 3px 0;  
margin-left: 0;  
font: bold 12px Verdana;  
border-bottom: 1px solid gray;  
list-style-type: none;  
text-align: left; /*set to left, center, or right to align the menu as  
desired*/  
}  
  
.basictab li{  
display: inline;  
margin: 0;  
}
```

```
.basictab li a{
text-decoration: none;
padding: 3px 7px;
margin-right: 3px;
border: 1px solid gray;
border-bottom: none;
background-color: #f6ffd5;
color: #2d2b2b;
}

.basictab li a:visited{
color: #2d2b2b;
}

.basictab li a:hover{
background-color: #DBFF6C;
color: black;
}

.basictab li a:active{
color: black;
}

.basictab li.selected a{ /*selected tab effect*/
position: relative;
top: 1px;
padding-top: 4px;
background-color: #DBFF6C;
color: black;
}
```

This CSS is shown for the sake of completion, and we will not go into details of CSS style sheets. It is sufficient to say that CSS provides an excellent method to create websites with good presentation.

The next step is the actual web page. We create an `index.jsp` page, in the **WebContent** directory, which will have the menu and also the JavaScript functions for our samples. It should be noted that although all JavaScript code is added to a single JSP page here in this sample, in "real" projects it would probably be more useful to create a separate file for JavaScript functions and include the JavaScript file in the HTML/JSP page using a code snippet such as this:

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

We will add JavaScript functions later for each sample. The following is the JSP code that shows the menu using the remoted `HorizontalMenu` class.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-
1">
<link href="styles.css" rel="stylesheet" type="text/css"/>
<script type='text/javascript' src='/DWREasyAjax/dwr/engine.js'></
script>
<script type='text/javascript' src='/DWREasyAjax/dwr/util.js'></
script>
<script type='text/javascript' src='/DWREasyAjax/dwr/interface/
HorizontalMenu.js'></script>
<title>DWR samples</title>
<script type="text/javascript">

function loadMenuItems()
{
    HorizontalMenu.getMenuItems(setMenuItems);
}

function getContent(contentId)
{
    AppContent.getContent(contentId, setContent);
}

function menuItemFormatter(item)
{
    elements=item.split(',');
    return '<li><a href="#" onclick="getContent(\''+elements[1]+'\'
);return false;">'+elements[0]+'</a></li>';
}

function setMenuItems(menuItems)
{
    menu=dwr.util.byId("dwrMenu");
    menuItemsHtml='';
    for(var i=0;i<menuItems.length;i++)
    {
        menuItemsHtml=menuItemsHtml+menuItemFormatter(menuItems[i]);
    }
    menu.innerHTML=menuItemsHtml;
}

function setContent(htmlArray)
{
```

```
var contentFunctions='';
var scriptToBeEvaled='';
var contentHtml='';
for(var i=0;i<htmlArray.length;i++)
{
    var html=htmlArray[i];
    if(html.toLowerCase().indexOf('<script')>-1)
    {
        if(html.indexOf('TO BE EVALED')>-1)
        {
            scriptToBeEvaled=html.substring(html.indexOf('>')+1,
                                            html.indexOf('</'));
        }
        else
        {
            eval(html.substring(html.indexOf('>')+1,html.indexOf('</')));
            contentFunctions+=html;
        }
    }
    else
    {
        contentHtml+=html;
    }
}

contentScriptArea=dwr.util.byId("contentAreaFunctions");
contentScriptArea.innerHTML=contentFunctions;
contentArea=dwr.util.byId("contentArea");
contentArea.innerHTML=contentHtml;
if(scriptToBeEvaled!='')
{
    eval(scriptToBeEvaled);
}
}
</script>
</head>
<body onload="loadMenuItems()">
<h1>DWR Easy Java Ajax Applications</h1>
<ul class="basictab" id="dwrMenu">
</ul>
<div id="contentAreaFunctions">
</div>
<div id="contentArea">
</div>
</body>
</html>
```

This JSP is our user interface. The HTML is just normal HTML with a `head` element and a `body` element. The head includes reference to a style sheet and to DWR JavaScript files, `engine.js`, `util.js`, and our own `HorizontalMenu.js`. The `util.js` file is optional, but as it contains very useful functions, it could be included in all the web pages where we use the functions in `util.js`.

The `body` element has a `contentArea` place holder for the content pages just below the menu. It also contains the content area for JavaScript functions for a particular content. The `body` element `onload`-event executes the `loadMenuItems()` function when the page is loaded. The `loadMenuItems()` function calls the `remoted` method of the `HorizontalMenu` Java class. The parameter of the `HorizontalMenu.getItems()` JavaScript function is the callback function that is called by DWR when the Java method has been executed and it returns menu items.

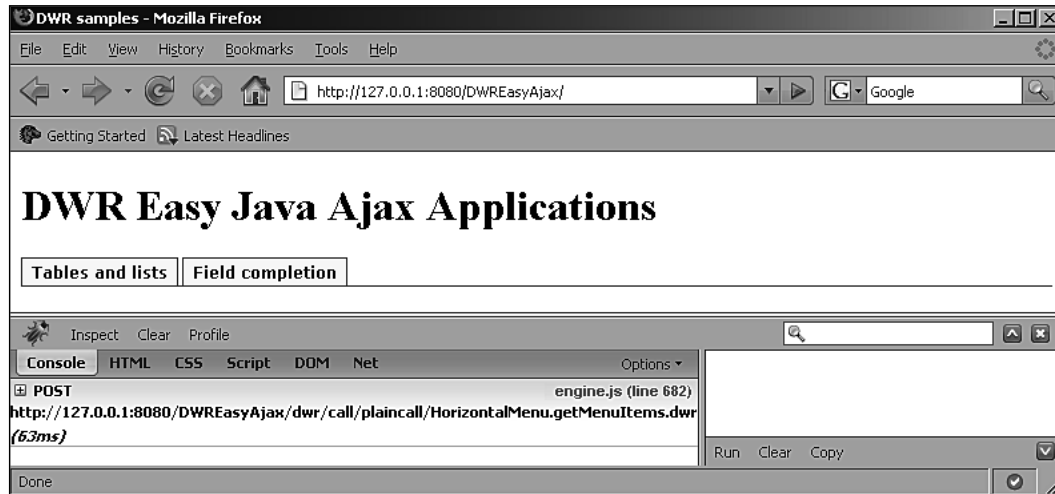
The `setMenuItems()` function is a callback function for the `loadMenuItems()` function mentioned in the previous paragraph. While loading menu items, the `Horizontal.getItems()` `remoted` method returns menu items as a `List` of `Strings` as a parameter to the `setMenuItems()` function. The menu items are formatted using the `menuItemFormatter()` helper function.

The `menuItemFormatter()` function creates `li` elements of menu texts. Menus are formatted as links, (`a href`) and they have an `onclick` event that has a function call to the `getContent`-function, which in turn calls the `AppContent.getContent()` function.

The `AppContent` is a `remoted` Java class, which we haven't implemented yet, and its purpose is to read the HTML from a file based on the menu item that the user clicked. Implementation of `AppContent` and the content pages are described in the next section.

The `setContent()` function sets the HTML content to the content area and also evaluates JavaScript options that are within the content to be inserted in the content area (this is not used very much, but it is there for those who need it).

Our dynamic user interface looks like this:



Note the Firebug window at the bottom of the browser screen. The Firebug console in the screenshot shows one POST request to our `HorizontalMenu.getItems()` method. Other Firebug features are extremely useful during development work, and we find it useful that Firebug has been enabled throughout the development work.

## Callback Functions

We saw our first callback function as a parameter in the `HorizontalMenu.getItems(setMenuItems)` function, and since callbacks are an important concept in DWR, it would be good to discuss a little more about them now that we have seen their first usage.

Callbacks are used to operate on the data that was returned from a remoted method. As DWR and AJAX are asynchronous, typical return values in **RPCs (Remote Procedure Calls)**, as in Java calls, do not work. DWR hides the details of calling the callback functions and handles everything internally from the moment we return a value from the remoted Java method to receiving the returned value to the callback function.

Two methods are recommended while using callback functions.

We have already seen the first method in the `HorizontalMenu.getItems (setMenuItems)` function call. Remember that there are no parameters in the `getMenuItems ()` Java method, but in the JavaScript call, we added the callback function name at the end of the parameter list. If the Java method has parameters, then the JavaScript call is similar to `CountryDB.getCountries (selectedLetters, setCountryRows)`, where `selectedLetters` is the input parameter for the Java method and `setCountryRows` is the name of the callback function (we see the implementation later on).

The second method to use callbacks is a meta-data object in the remote JavaScript call. An example (a full implementation is shown later in this chapter) is shown here:

```
CountryDB.saveCountryNotes(ccode, newNotes, {
    callback: function(newNotes)
    {
        //function body here
    }
});
```

Here, the function is anonymous and its implementation is included in the JavaScript call to the remoted Java method. One advantage here is that it is easy to read the code, and the code is executed immediately after we get the return value from the Java method. The other advantage is that we can add extra options to the call.

Extra options include timeout and error handler as shown in the following example:

```
CountryDB.saveCountryNotes(ccode, newNotes, {
    callback: function(newNotes)
    {
        //function body here
    },
    timeout: 10000,
    errorHandler: function(errorMessage) { alert(errorMessage); }
});
```

It is also possible to add a callback function to those Java methods that do not return a value. Adding a callback to methods with no return values would be useful in getting a notification when a remote call has been completed.

## Afterword

Our first sample is ready, and it is also the basis for the following samples. We also looked at how applications are tested in the Eclipse environment.

Using DWR, we can look at JavaScript code on the browser and Java code on the server as one. It may take a while to get used to it, but it will change the way we develop web applications. Logically, there is no longer a client and a server but just a single run time platform that happens to be physically separate. But in practice, of course, applications using DWR, JavaScript on the client and Java in the server, are using the typical client-server interaction. This should be remembered when writing applications in the logically single run-time platform.

## Implementing Tables and Lists

The first actual sample is very common in applications: tables and lists. In this sample, the table is populated using the DWR utility functions, and a remoted Java class. The sample code also shows how DWR is used to do inline table editing. When a table cell is double-clicked, an edit box opens, and it is used to save new cell data.

The sample will have country data in a CSV file: country **Name**, **Long Name**, two-letter **Code**, **Capital**, and user-defined **Notes**. The user interface for the table sample appears as shown in the following screenshot:

### DWR Easy Java Ajax Applications

Tables and lists
Field completion

#### Countries

Show countries starting with

Doubleclick "Notes"-cell to add notes to country.

Name	Long name	Code	Capital	Notes
Fiji	Republic of theFiji Islands	FJ	Suva	
Finland	Republic of Finland	FI	Helsinki	
France	French Republic	FR	Paris	

## Server Code for Tables and Lists

The first thing to do is to get the country data. Country data is in a CSV file (named `countries.csv` and located in the `samples` Java package). The following is an excerpt of the content of the CSV file (data is from `http://www.state.gov`).

```
Short-form name,Long-form name,FIPS Code,Capital

Afghanistan,Islamic Republic of Afghanistan,AF,Kabul

Albania,Republic of Albania,AL,Tirana

Algeria,People's Democratic Republic of Algeria,AG,Algiers

Andorra,Principality of Andorra,AN,Andorra la Vella

Angola,Republic of Angola,AO,Luanda

Antigua and Barbuda,(no long-form name),AC,Saint John's

Argentina,Argentine Republic,AR,Buenos Aires

Armenia,Republic of Armenia,AM,Yerevan
```

...

The CSV file is read each time a client requests country data. Although this is not very efficient, it is good enough here. Other alternatives include an in-memory cache or a real database such as Apache Derby or IBM DB2. As an example, we have created a `CountryDB` class that is used to read and write the country CSV. We also have another class, `DBUtils`, which has some helper methods. The `DBUtils` code is as follows:

```
package samples;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStream;
```

```
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.util.List;
import java.util.Vector;

public class DBUtils {

    private String fileName=null;
    public void initFileDB(String fileName)
    {
        this.fileName=fileName;
        // copy csv file to bin-directory, for easy
        // file access
        File countriesFile = new File(fileName);
        if (!countriesFile.exists()) {
            try {
                List<String> countries = getCSVStrings(null);
                PrintWriter pw;
                pw = new PrintWriter(new FileWriter(countriesFile));
                for (String country : countries) {
                    pw.println(country);
                }
                pw.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

    protected List<String> getCSVStrings(String letter) {
        List<String> csvData = new Vector<String>();
        try {
            File csvFile = new File(fileName);
            BufferedReader br = null;
            if(csvFile.exists())
            {
                br=new BufferedReader(new FileReader(csvFile));
            }
            else
            {
                InputStream is = this.getClass().getClassLoader()
                    .getResourceAsStream("samples/"+fileName);
                br=new BufferedReader(new InputStreamReader(is));
                br.readLine();
            }

            for (String line = br.readLine(); line != null; line =
br.readLine()) {
                if (letter == null
```

```
        || (letter != null && line.startsWith(letter))) {
            csvData.add(line);
        }
    }
    br.close();
} catch (IOException ioe) {
    ioe.printStackTrace();
}
return csvData;
}
}
```

The `DBUtils` class is a straightforward utility class that returns CSV content as a `List of Strings`. It also copies the original CSV file to the runtime directory of any application server we might be running. This may not be the best practice, but it makes it easier to manipulate the CSV file, and we always have the original CSV file untouched if and when we need to go back to the original version.

The code for `CountryDB` is given here:

```
package samples;

import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Arrays;
import java.util.List;
import java.util.Vector;

public class CountryDB {

    private DBUtils dbUtils = new DBUtils();
    private String fileName = "countries.csv";

    public CountryDB() {
        dbUtils.initFileDB(fileName);
    }

    public String[] getCountryData(String ccode) {
        List<String> countries = dbUtils.getCSVStrings(null);
        for (String country : countries) {
            if (country.indexOf(", " + ccode + ", ") > -1) {
                return country.split(", ");
            }
        }
        return new String[0];
    }
}
```

---

```
public List<List<String>> getCountries(String startLetter) {

    List<List<String>> allCountryData = new Vector<List<String>>();
    List<String> countryData = dbUtils.getCSVStrings(startLetter);
    for (String country : countryData) {
        String[] data = country.split(",");
        allCountryData.add(Arrays.asList(data));
    }
    return allCountryData;
}

public String[] saveCountryNotes(String ccode, String notes) {
    List<String> countries = dbUtils.getCSVStrings(null);
    try {
        PrintWriter pw = new PrintWriter(new FileWriter(fileName));
        for (String country : countries) {
            if (country.indexOf(", " + ccode + ",") > -1) {
                if (country.split(",").length == 4) {
                    // no existing notes
                    country = country + ", " + notes;
                } else {
                    if (notes.length() == 0) {
                        country = country.substring(0, country
                            .lastIndexOf(", "));
                    } else {
                        country = country.substring(0, country
                            .lastIndexOf(", "))
                            + ", " + notes;
                    }
                }
            }
        }
        pw.println(country);
    }
    pw.close();
} catch (IOException ioe) {
    ioe.printStackTrace();
}

String[] rv = new String[2];
rv[0] = ccode;
rv[1] = notes;
return rv;
}
}
```

The `CountryDB` class is a remoted class. The `getCountryData()` method returns country data as an array of strings based on the country code. The `getCountries()` method returns all the countries that start with the specified parameter, and `saveCountryNotes()` saves user added notes to the country specified by the country code.

In order to use `CountryDB`, the following script element must be added to the `index.jsp` file together with other JavaScript elements.

```
<script type='text/javascript' src='/DWREasyAjax/dwr/interface/
CountryDB.js'></script>
```

There is one other Java class that we need to create and remote. That is the `AppContent` class that was already present in the JavaScript functions of the home page. The `AppContent` class is responsible for reading the content of the HTML file and parses the possible JavaScript function out of it, so it can become usable by the existing JavaScript functions in `index.jsp` file.

```
package samples;

import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.util.List;
import java.util.Vector;
public class AppContent {

    public AppContent()
    {

    }

    public List<String> getContent(String contentId)
    {
        InputStream is = this.getClass().getClassLoader().
getResourceAsStream(
        "samples/"+contentId+".html");
        String content=streamToString(is);
        List<String> contentList=new Vector<String>();
        //Javascript within script tag will be extracted and sent
separately to client
        for(String script=getScript(content);!script.equals("");script=getScript(content))
        {
            contentList.add(script);
            content=removeScript(content);
        }
    }
}
```

```
    }
    //content list will have all the javascript
    //functions, last element is executed last
    //and all other before html content
    if(contentList.size()>1)
    {
        contentList.add(contentList.size()-1, content);
    }
    else
    {
        contentList.add(content);
    }
    return contentList;
}

public List<String> getLetters()
{
    List<String> letters=new Vector<String>();
    char[] l=new char[1];
    for(int i=65;i<91;i++)
    {
        l[0]=(char)i;
        letters.add(new String(l));
    }
    return letters;
}

public String removeScript(String html)
{
    //removes first script element
    int sIndex=html.toLowerCase().indexOf("<script ");
    if(sIndex!=-1)
    {
        return html;
    }
    int eIndex=html.toLowerCase().indexOf("</script>")+9;
    return html.substring(0, sIndex)+html.substring(eIndex);
}

public String getScript(String html)
{
    //returns first script element
    int sIndex=html.toLowerCase().indexOf("<script ");
    if(sIndex!=-1)
    {
        return "";
    }
}
```

```
        int eIndex=html.toLowerCase().indexOf("</script>")+9;
        return html.substring(sIndex, eIndex);
    }

    public String streamToString(InputStream is)
    {
        String content="";
        try
        {
            ByteArrayOutputStream baos=new ByteArrayOutputStream();
            for(int b=is.read();b!=-1;b=is.read())
            {
                baos.write(b);
            }
            content=baos.toString();
        }
        catch(IOException ioe)
        {
            content=ioe.toString();
        }
        return content;
    }
}
```

The `getContent()` method reads the HTML code from a file based on the `contentId`. `ContentId` was specified in the `dwrapplication.properties` file, and the HTML is just `contentId` plus the extension `.html` in the package directory. There is also a `getLetters()` method that simply lists letters from A to Z and returns a list of letters to the browser.

If we test the application now, we will get an error as shown in the following screenshot:



We know why the **AppContent is not defined** error occurs, so let's fix it by adding `AppContent` to the `allow` element in the `dwr.xml` file. We also add `CountryDB` to the `allow` element. The first thing we do is to add required elements to the `dwr.xml` file. We add the following creators within the `allow` element in the `dwr.xml` file.

```
<create creator="new" javascript="AppContent">
  <param name="class" value="samples.AppContent" />
  <include method="getContent" />
  <include method="getLetters" />
</create>
<create creator="new" javascript="CountryDB">
  <param name="class" value="samples.CountryDB" />
  <include method="getCountries" />
  <include method="saveCountryNotes" />
  <include method="getCountryData" />
</create>
```

We explicitly define the methods we are removing using the `include` elements. This is a good practice, as we don't accidentally allow access to any methods that are not meant to be removed.

## Client Code for Tables and Lists

We also need to add a JavaScript interface to the `index.jsp` page. Add the following with the rest of the scripts in the `index.jsp` file.

```
<script type='text/javascript' src='/DWREasyAjax/dwr/interface/
AppContent.js'></script>
```

Before testing, we need the sample HTML for the content area. The following HTML is in the `TablesAndLists.html` file under the **samples** directory:

```
<h3>Countries</h3>
<p>Show countries starting with
<select id="letters" onchange="selectLetter(this);return false;"> </
select><br/>
Doubleclick "Notes"-cell to add notes to country.
</p>
<table border="1">
  <thead>
    <tr>
      <th>Name</th>
      <th>Long name</th>
      <th>Code</th>
      <th>Capital</th>
      <th>Notes</th>
    </tr>
```

```
</thead>
<tbody id="countryData">
</tbody>
</table>

<script type='text/javascript'>
//TO BE EVALUED
AppContent.getLetters(addLetters);
</script>
```

The script element at the end is extracted by our Java class, and it is then evaluated by the browser when the client-side JavaScript receives the HTML. There is the select element, and its onchange event calls the selectLetter() JavaScript function. We will implement the selectLetter() function shortly.

JavaScript functions are added in the index.jsp file, and within the head element. Functions could be in separate JavaScript files, but the embedded script is just fine here.

```
function selectLetter(selectElement)
{
    var selectedIndex = selectElement.selectedIndex;
    var selectedLetter= selectElement.options[selectedIndex].value;
    CountryDB.getCountries(selectedLetter, setCountryRows);
}
function addLetters(letters)
{
    dwr.util.addOptions('letters', ['letter...']);
    dwr.util.addOptions('letters', letters);
}

function setCountryRows(countryData)
{
    var cellFuncs = [
        function(data) { return data[0]; },
        function(data) { return data[1]; },
        function(data) { return data[2]; },
        function(data) { return data[3]; },
        function(data) { return data[4]; }
    ];
    dwr.util.removeAllRows('countryData');
    dwr.util.addRows( 'countryData', countryData, cellFuncs, {
        cellCreator:function(options) {
            var td = document.createElement("td");
            if(options.cellNum==4)
            {
                var notes=options.rowData[4];
                if(notes==undefined)
```

```

        {
            notes='&nbsp;';// + options.rowData[2]+'notes';
        }
        var ccode=options.rowData[2];
        var divId=ccode+'_Notes';
        var tdId=divId+'Cell';
        td.setAttribute('id',tdId);
        var html=getNotesHtml(ccode,notes);
        td.innerHTML=html;
        options.data=html;
    }
    return td;
},
escapeHtml:false
});
}

function getNotesHtml(ccode,notes)
{
    var divId=ccode+'_Notes';
    return "<div onDbClick=
        \"editCountryNotes('"+divId+"','"+ccode+"');\" id=
        \""+divId+"\">"+notes+"</div>";
}

function editCountryNotes(id,ccode)
{
    var notesElement=dwr.util.byId(id);
    var tdId=id+'Cell';
    var notes=notesElement.innerHTML;
    if(notes=='&nbsp;')
    {
        notes='';
    }
    var editBox='<input id="'+ccode+'NotesEditBox" type=
        "text" value="'+notes+'"/><br/>';
    editBox+="<input type='button' id='"+ccode+'SaveNotesButton'
        value='Save' onclick='saveCountryNotes(\""+ccode+"\");'/>";
    editBox+="<input type='button' id='"+ccode+'CancelNotesButton'
        value='Cancel' onclick='cancelEditNotes
        (\""+ccode+"\");'/>";

    tdElement=dwr.util.byId(tdId);
    tdElement.innerHTML=editBox;
    dwr.util.byId(ccode+'NotesEditBox').focus();
}

```

```
function cancelEditNotes(ccode)
{
  var countryData=CountryDB.getCountryData(ccode, {
  callback:function(data)
  {
    var notes=data[4];
    if(notes==undefined)
    {
      notes='&nbsp;';
    }
    var html=getNotesHtml(ccode,notes);
    var tdId=ccode+'_NotesCell';
    var td=dwr.util.byId(tdId);
    td.innerHTML=html;
  }
  });
}

function saveCountryNotes(ccode)
{
  var editBox=dwr.util.byId(ccode+'NotesEditBox');
  var newNotes=editBox.value;
  CountryDB.saveCountryNotes(ccode,newNotes, {
  callback:function(newNotes)
  {
    var ccode=newNotes[0];
    var notes=newNotes[1];
    var notesHtml=getNotesHtml(ccode,notes);
    var td=dwr.util.byId(ccode+"_NotesCell");
    td.innerHTML=notesHtml;
  }
  });
}
```

There are lots of functions for table samples, and we go through each one of them.

---

The first is the `selectLetter()` function. This function gets the selected letter from the `select` element and calls the `CountryDB.getCountries()` remote Java method. The callback function is `setCountryRows`. This function receives the return value from the Java `getCountries()` method, that is `List<List<String>>`, a List of Lists of Strings.

The second function is `addLetters(letters)`, and it is a callback function for the `AppContent.getLetters()` method, which simply returns letters from A to Z. The `addLetters()` function uses the DWR utility functions to populate the letter list.

Then there is a callback function for the `CountryDB.getCountries()` method. The parameter for the function is an array of countries that begin with a specified letter. Each array element has a format: **Name**, **Long name**, (country code) **Code**, **Capital**, **Notes**. The purpose of this function is to populate the table with country data; and let's see how it is done. The variable, `cellFuncs`, holds functions for retrieving data for each cell in a column. The parameter named `data` is an array of country data that was returned from the Java class.

The table is populated using the DWR utility function, `addRows()`. The `cellFuncs` variable is used to get the correct data for the table cell. The `cellCreator` function is used to create custom HTML for the table cell. Default implementation generates just a `td` element, but our custom implementation generates the `td`-element with the `div` placeholder for user notes.

The `getNotesHtml()` function is used to generate the `div` element with the event listener for double-click.

The `editCountryNotes()` function is called when the table cell is double-clicked. The function creates input fields for editing notes with the **Save** and **Cancel** buttons.

The `cancelEditNotes()` and `saveCountryNotes()` functions cancel the editing of new notes, or saves them by calling the `CountryDB.saveCountryNotes()` Java method.

The following screenshot shows what the sample looks like with the populated table:



Now that we have added necessary functions to the web page we can test the application.

## Testing Tables and Lists

The application should be ready for testing if we have had the test environment running during development. Eclipse automatically deploys our new code to the server whenever something changes. So we can go right away to the test page `http://127.0.0.1:8080/DWREasyAjax`. On clicking **Tables and lists** we can see the page we have developed. By selecting some letter, for example "I" we get a list of all the countries that start with letter "I" (as shown in the previous screenshot).

Now we can add notes to countries. We can double-click any table cell under **Notes**. For example, if we want to enter notes to **Iceland**, we double-click the **Notes** cell in Iceland's table row, and we get the edit box for the notes as shown in the following screenshot:

The screenshot shows a Mozilla Firefox browser window displaying a web application. The browser title is "DWR samples - Mozilla Firefox". The address bar shows the URL "http://127.0.0.1". The page has two tabs: "Tables and lists" (selected) and "Field completion". The main content area is titled "Countries" and contains a dropdown menu with "I" selected. Below the dropdown is a table with columns: Name, Long name, Code, Capital, and Notes. The table lists Iceland, India, Indonesia, and Iran. The "Notes" cell for Iceland is double-clicked, showing an edit box with the text "been there" and "Save" and "Cancel" buttons. The browser's developer tools are open at the bottom, showing the Console with a POST request to "http://127.0.0.1:8080/DWREasyAjax/dwr/call/p".

Name	Long name	Code	Capital	Notes
Iceland	Republic of Iceland	IC	Reykjavik	been there
India	Republic of India	IN	New Delhi	
Indonesia	Republic of Indonesia	ID	Jakarta	
Iran	Islamic Republic of Iran	IR	Tehran	

The edit box is a simple text input field. We didn't use any forms. Saving and canceling editing is done using JavaScript and DWR. If we press **Cancel**, we get the original notes from the `CountryDB` Java class using DWR and saving also uses DWR to save data. `CountryDB.saveCountryNotes()` takes the country code and the notes that the user entered in the edit box and saves them to the CSV file. When notes are available, the application will show them in the country table together with other country information as shown in the following screenshot:



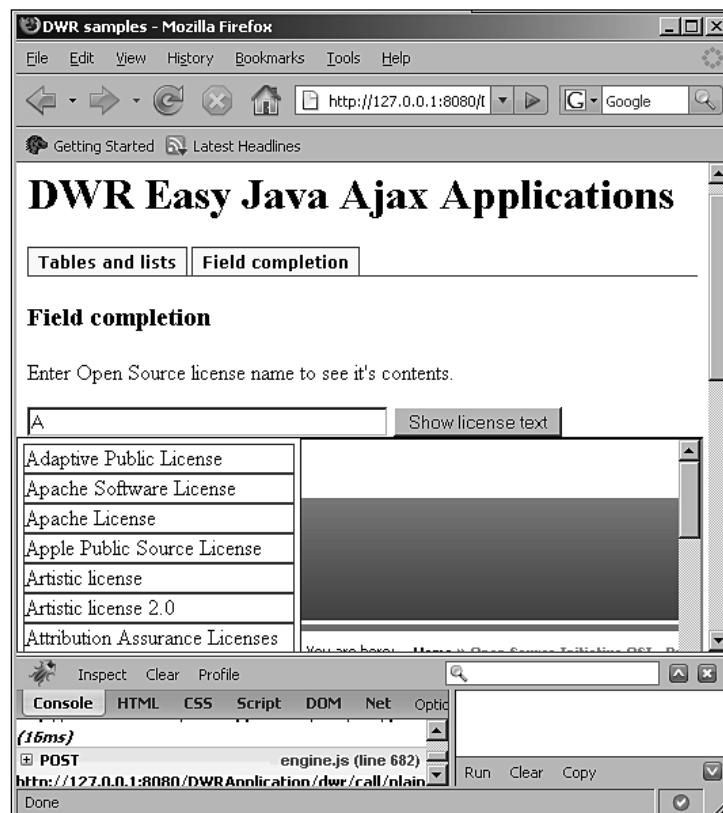
## Afterword

The sample in this section uses DWR features to get data for the table and list from the server. We developed the application so that most of the application logic is written in JavaScript and Java beans that are remoted. In principle, the application logic can be thought of as being fully browser based, with some extensions in the server.

## Implementing Field Completion

Nowadays, field completion is typical of many web pages. A typical use case is getting a stock quote, and field completion shows matching symbols as users type letters. Many Internet sites use this feature.

Our sample here is a simple license text finder. We enter the license name in the input text field, and we use DWR to show the license names that start with the typed text. A list of possible completions is shown below the input field. The following is a screenshot of the field completion in action:



Selected license content is shown in an `iframe` element from <http://www.opensource.org>.

## Server Code for Field Completion

We will re-use some of the classes we developed in the last section. `AppContent` is used to load the sample page, and the `DBUtils` class is used in the `LicenseDB` class. The `LicenseDB` class is shown here:

```
package samples;

import java.util.List;
import java.util.Vector;

public class LicenseDB{

    private DBUtils dbUtils=new DBUtils();

    public LicenseDB()
    {
        dbUtils.initFileDB("licenses.csv");
    }

    public List<String> getLicensesStartingWith(String startLetters)
    {
        List<String> list=new Vector<String>();
        List<String> licenses=dbUtils.getCSVStrings(startLetters);
        for(String license : licenses)
        {
            list.add(license.split(",")[0]);
        }
        return list;
    }

    public String getLicenseContentUrl(String licenseName)
    {
        List<String> licenses=dbUtils.getCSVStrings(licenseName);
        if(licenses.size(>0)
        {
            return licenses.get(0).split(",")[1];
        }
        return "";
    }
}
```

The `getLicenseStartingWith()` method goes through the license names and returns valid license names and their URLs. Similar to the data in the previous section, license data is in a CSV file named `licenses.csv` in the package directory. The following is an excerpt of the file content:

```
Academic Free License, http://opensource.org/licenses/afl-3.0.php
Adaptive Public License, http://opensource.org/licenses/apl1.0.php
Apache Software License, http://opensource.org/licenses/apache1-1.1.php
Apache License, http://opensource.org/licenses/apache2.0.php
Apple Public Source License, http://opensource.org/licenses/apsl-2.0.php
Artistic license, http://opensource.org/licenses/artistic-
license-1.0.php
...
```

There are quite a few open-source licenses. Some are more popular than others (like the Apache Software License) and some cannot be re-used (like the IBM Public License).

We want to remote the `LicenseDB` class, so we add the following to the `dwr.xml` file.

```
<create creator="new" javascript="LicenseDB">
  <param name="class" value="samples.LicenseDB"/>
  <include method="getLicensesStartingWith"/>
  <include method="getLicenseContentUrl"/>
</create>
```

## Client Code for Field Completion

The following script element will go in the `index.jsp` page.

```
<script type='text/javascript' src='/DWREasyAjax/dwr/interface/
LicenseDB.js'></script>
```

The HTML for the field completion is as follows:

```
<h3>Field completion</h3>
<p>Enter Open Source license name to see it's contents.
</p>
<input type="text" id="licenseNameEditBox" value="" onkeyup="showPopup
Menu()" size="40"/>
<input type="button" id="showLicenseTextButton" value="Show license
text" onclick="showLicenseText()"/>
<div id="completionMenuPopup"></div>
<div id="licenseContent"></div>
```

The `input` element, where we enter the license name, listens to the `onkeyup` event which calls the `showPopupMenu()` JavaScript function. Clicking the Input button calls the `showLicenseText()` function (the JavaScript functions are explained shortly). Finally, the two `div` elements are place holders for the pop-up menu and the `iframe` element that shows license content.

For the pop-up box functionality, we use existing code and modify it for our purpose (many thanks to <http://www.jtricks.com>). The following is the `popup.js` file, which is located under the **WebContent | js** directory.

```
//<script type="text/javascript"><!--
/* Original script by: www.jtricks.com
 * Version: 20070301
 * Latest version:
 * www.jtricks.com/javascript/window/box.html
 *
 * Modified by Sami Salkosuo.
 */
// Moves the box object to be directly beneath an object.
function move_box(an, box)
{
    var cleft = 0;
    var ctop = 0;
    var obj = an;

    while (obj.offsetParent)
    {
        cleft += obj.offsetLeft;
        ctop += obj.offsetTop;
        obj = obj.offsetParent;
    }

    box.style.left = cleft + 'px';

    ctop += an.offsetHeight + 8;

    // Handle Internet Explorer body margins,
    // which affect normal document, but not
    // absolute-positioned stuff.
    if (document.body.currentStyle &&
        document.body.currentStyle['marginTop'])
    {
        ctop += parseInt(
            document.body.currentStyle['marginTop']);
    }
}
```

---

```
    box.style.top = ctop + 'px';
}

var popupMenuInitialised=false;
// Shows a box if it wasn't shown yet or is hidden
// or hides it if it is currently shown
function show_box(html, width, height, borderStyle,id)
{
    // Create box object through DOM
    var boxdiv = document.getElementById(id);
    boxdiv.style.display='block';
    if(popupMenuInitialised==false)
    {
        //boxdiv = document.createElement('div');
        boxdiv.setAttribute('id', id);
        boxdiv.style.display = 'block';
        boxdiv.style.position = 'absolute';
        boxdiv.style.width = width + 'px';
        boxdiv.style.height = height + 'px';
        boxdiv.style.border = borderStyle;
        boxdiv.style.textAlign = 'right';
        boxdiv.style.padding = '4px';
        boxdiv.style.background = '#FFFFFF';
        boxdiv.style.zIndex='99';
        popupMenuInitialised=true;
        //document.body.appendChild(boxdiv);
    }

    var contentId=id+'Content';
    var contents = document.getElementById(contentId);
    if(contents==null)
    {
        contents = document.createElement('div');
        contents.setAttribute('id', id+'Content');
        contents.style.textAlign= 'left';
        boxdiv.contents = contents;
        boxdiv.appendChild(contents);
    }

    move_box(html, boxdiv);

    contents.innerHTML= html;

    return false;
}
```

```
function hide_box(id)
{
    document.getElementById(id).style.display='none';
    var boxdiv = document.getElementById(id+'Content');
    if(boxdiv!=null)
    {
        boxdiv.parentNode.removeChild(boxdiv);
    }
    return false;
}

//--></script>
```

Functions in the `popup.js` file are used as menu options directly below the edit box.

The `show_box()` function takes the following arguments: HTML code for the pop-up, position of the pop-up window, and the "parent" element (to which the pop-up box is related). The function then creates a pop-up window using DOM. The `move_box()` function is used to move the pop-up window to its correct place under the edit box and the `hide_box()` function hides the pop-up window by removing the pop-up window from the DOM tree.

In order to use functions in `popup.js`, we need to add the following script-element to the `index.jsp` file:

```
<script type='text/javascript' src='js/popup.js'></script>
```

Our own JavaScript code for the field completion is in the `index.jsp` file. The following are the JavaScript functions, and an explanation follows the code:

```
function showPopupMenu()
{
    var licenseNameEditBox=dwr.util.byId('licenseNameEditBox');
    var startLetters=licenseNameEditBox.value;
    LicenseDB.getLicensesStartingWith(startLetters, {
    callback:function(licenses)
    {
        var html="";
        if(licenses.length==0)
        {
            return;
        }
        if(licenses.length==1)
        {
            hidePopupMenu();
        }
    }
    });
}
```

```
        licenseNameEditBox.value=licenses[0];
    }
    else
    {
        for (index in licenses)
        {
            var licenseName=licenses[index]; //.split(",")[0];
            licenseName=licenseName.replace(/\\/g, "&quot;");
            html+="

"+licenseName+"</div>";
        }
        show_box(html, 200, 270, '1px solid', 'completionMenuPopup');
    }
    });
}

function hidePopupMenu()
{
    hide_box('completionMenuPopup');
}

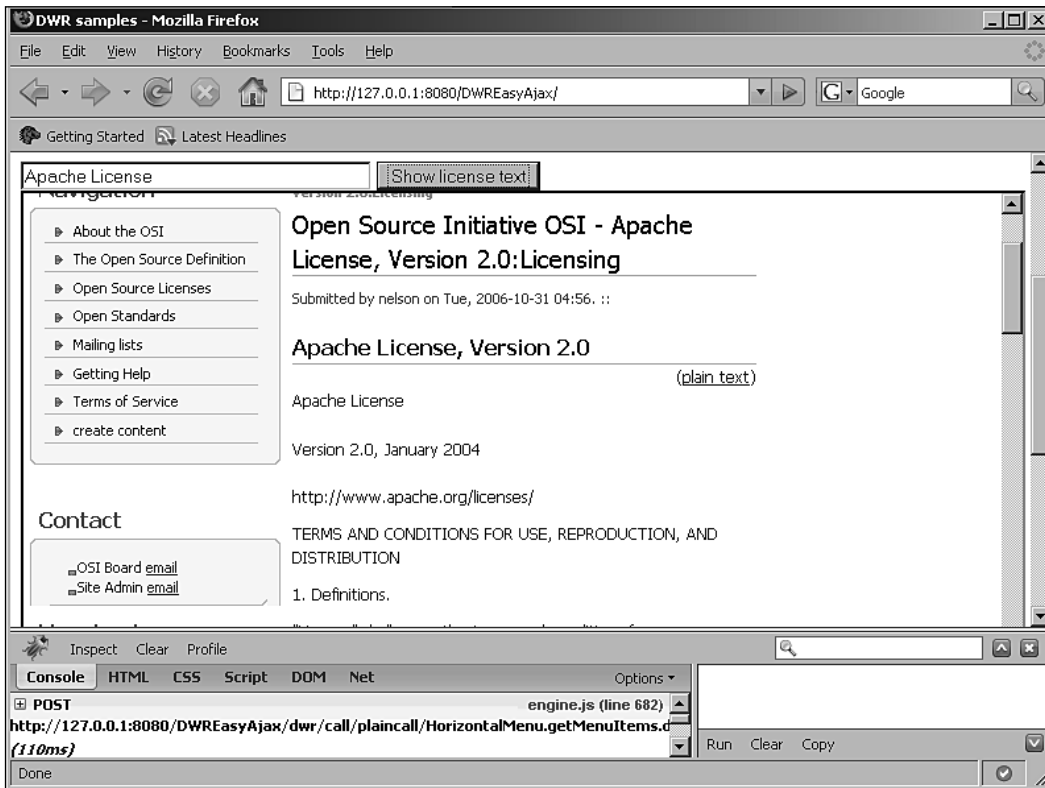
function completeEditBox(licenseName)
{
    var licenseNameEditBox=dwr.util.byId('licenseNameEditBox');
    licenseNameEditBox.value=licenseName;
    hidePopupMenu();
    dwr.util.byId('showLicenseTextButton').focus();
}

function showLicenseText()
{
    var licenseNameEditBox=dwr.util.byId('licenseNameEditBox');
    licenseName=licenseNameEditBox.value;
    LicenseDB.getLicenseContentUrl(licenseName, {
    callback:function(licenseUrl)
    {
        var html='<iframe src="'+licenseUrl+'" width="100%"
            height="600"></iframe>';
        var content=dwr.util.byId('licenseContent');
        content.style.zIndex="1";
        content.innerHTML=html;
    }
    });
}


```

The `showPopupMenu()` function is called each time a user enters a letter in the input box. The function gets the value of the input field and calls the `LicenseDB.getLicensesStartingWith()` method. The callback function is specified in the function parameters. The callback function gets all the licenses that match the parameter, and based on the length of the parameter (which is an array), it either shows a pop-up box with all the matching license names, or, if the array length is one, hides the pop-up box and inserts the full license name in the text field. In the pop up box, the license names are wrapped within the `div` element that has an `onclick` event listener that calls the `completeEditBox()` function.

The `hidePopupMenu()` function just closes the pop-up menu and the `completeEditBox()` function inserts the clicked license text in the input box and moves the focus to the button. The `showLicenseText()` function is called when we click the **Show license text** button. The function calls the `LicenseDB.getLicenseContentUrl()` method and the callback function creates an `iframe` element to show the license content directly from `http://www.opensource.org`, as shown in the following screenshot:



## **Afterword**

Field completion improves user experience in web pages and the sample code in this section showed one way of doing it using DWR.

It should be noted that the sample for field completion presented here is only for demonstration purposes.

## **Summary**

This chapter provided samples for a couple of common tasks that are used in web development: tables and lists, field completion, and even a generic frame, called a dynamic user interface, for our sample code. Both the tables and lists sample and the field completion sample had a very simple CSV-based "database" that holds the data for our purposes, and both had a remoted method that DWR uses to get the data from the server and show it in the client.

We also saw some good examples of HTML, CSS, and JavaScript. In fact, without knowledge of JavaScript it would be difficult to write web applications.

Many years ago, as some of you may remember, JavaScript was a dirty word in web development and no self-respecting developer would touch JavaScript. But change is a part of life, and in this case, change has been for the better. JavaScript and Java work very well together with the DWR in between.

The next chapter continues with the user interface part, and shows a couple more samples, including a map scrolling functionality, similar to what is found in the popular Google Maps website.