

Kick-start your Java apps, Part 2: Easy, lightweight Ajax development

Build an interactive Web application with the Kick-start your Java apps suite

Skill Level: Intermediate

[Sing Li \(westmakaha@yahoo.com\)](mailto:westmakaha@yahoo.com)

Author

Wrox Press

18 Apr 2006

Updated 05 Dec 2007

The combination of Eclipse, DB2® Express-C 9.5, and WebSphere® Application Server Community Edition 2.0 — all free to download, use, and deploy — is an excellent from-prototype-to-production suite for all of your Java™ and Java enterprise development needs. What might not be obvious is the relative ease with which you can use these proven tools to create, test, and deploy cutting-edge, lightweight applications as well. This tutorial guides you through the development of a small human-resources application, first using conventional JavaServer Pages (JSP) based technology, and then migrating it to a highly interactive solution using Ajax.

Section 1. Before you start

The needs of Java EE application and Web services developers are well served by the tight integration among the free tools that make up the Kick-start your Java apps suite: Eclipse, DB2 Express-C 9.5, and WebSphere Application Server Community Edition 2.0 (Application Server). The companion tutorial to this one, [Kick-start your Java apps, Part 1: Free software, fast development](#), covers how well these tools work together for Java EE software development.

Increasingly, Java developers are experimenting with highly interactive Web-based user interfaces, including alternatives that don't require the use of a conventional Java EE container. Such exploration often forces you to learn an unproven programming language, work with beta-quality tooling, and gamble on deployment technology that hasn't withstood the test of time.

It might surprise you to learn that the production-grade, robust set of free IBM-backed tools that you use for your Java EE development tasks can also be your best buddy in exploring the realm of alternative user-interface development. The great news is that all your code — lightweight or conventional — can take advantage of the rich, easy-to-use feature set of Eclipse, the maturity and robustness of DB2-Express C, and the world-class deployment support of Application Server.

This tutorial shows you how to move an application from a conventional design to one based on Asynchronous JavaScript and XML (Ajax) technology — all within the friendly and familiar environment of the Kick-start your Java apps suite.

The Kick-start combo

With the Kick-start your Java apps suite, IBM has put together a powerful combination of freely available software components for data access, application deployment, and development:

- DB2 Express-C 9.5 database server
- WebSphere Application Server Community Edition 2.0
- The Eclipse IDE

You're not locked in to using these packages in combination; each works with a variety of other open-standards based components. You can use Eclipse with other (even non-Java) application servers and other databases. Application Server doesn't require either DB2 Express-C 9.5 or Eclipse. And DB2 Express-C 9.5 can fit into development and deployment environments other than Eclipse and Application Server. But, in ways you'll understand fully from this tutorial, using the whole, tightly integrated suite can gain you a wealth of advantages.

About this tutorial

You'll start this tutorial by creating an employee-information panel using standard JSP and servlet technology. You'll do all development and testing using the Kick-start your Java apps suite. You'll design the application in Eclipse and then deploy and test it on Application Server. The application interactively fetches and displays employee information, including a photo, from a DB2 Express-C 9.5 database.

Then the tutorial discusses some of the limitations of the JSP application design and

shows how a lightweight Ajax-based solution can help to overcome them. It introduces basic Ajax concepts and a popular Java toolkit called Direct Web Remoting (DWR) (see [Resources](#)). You'll convert the application to a lightweight design. The redesigned application resides within an HTML page hosted on Application Server acting as a simple Web server, accessing DB2 Express C database information directly using Ajax and JavaScript Object Notation (JSON) serialization.

The tutorial guides you through:

- Developing an employee-information panel using conventional JSP and servlet technologies
- Handling the display of GIF photos from DB2 binary large object (BLOB) fields using a custom servlet
- Being introduced to Ajax and Direct Web Remoting
- Migrating the employee-information panel to a lightweight Ajax design
- Coding the Java-side support code for the new application
- JavaScript coding using the DWR utilities library
- Testing the new interactive employee-information panel

By the end of the tutorial, you'll appreciate some of the fundamental differences between conventional Java EE application design and the highly interactive Ajax-based approach. You'll also become comfortable with using the Kick-start your Java apps suite to explore present and future Java-based development frameworks.

Prerequisites

You should be familiar with Java development in general and server-side Java development specifically. This tutorial assumes you are familiar with basic Java EE deployment concepts, such as deployment descriptors and WAR archives. You should also be familiar with JSP programming and using tag libraries, such as the JavaServer Pages Standard Tag Library (JSTL). This tutorial assumes that you understand the general operations of a relational database and have programmed JDBC applications. It also assumes that you have completed the [Kick-start your Java apps: Free software, fast development](#) tutorial.

System requirements

To follow along and try out the code, you need working installations of:

- Sun's Java SE JDK 5 update 15 or the IBM SDK for Java Version 5 SR6 or later.
- The Kick-start your Java apps suite:
 - DB2 Express-C 9.5
 - Application Server version 2.0.0.1
 - Eclipse 3.3 and the WTP Server Adapter for Application Server V. 2.0For a proper configuration, you should follow the detailed download and installation instructions for these components in the relevant sections of [Kick-start your Java apps: Free software, fast development](#).
- The employee-listing application you built in [Kick-start your Java apps: Free software, fast development](#).
- The DB2 Express-C 9.5 sample database.
- DWR. Detailed instructions for experimenting with DWR are included in this tutorial.

The recommended system hardware configuration for trying out the tutorial:

- A system supporting the JDK/JRE listed above with at least 1GB of main memory (2GB recommended).
- At least 10MB of additional free disk space to install the software components and examples.

The instructions in the tutorial are based on a Windows® operating system. All of the tools and techniques covered in the tutorial also work on Linux® and Unix® systems.

Section 2. Overview

This tutorial builds on the conceptual foundation and system configuration from [Kick-start your Java apps: Free software, fast development](#). You'll use Eclipse to create two versions of a human-resource application that interactively accesses employee information stored in a DB2 Express-C 9.5 instance, and you'll use Application Server to deploy the applications.

In [Kick-start your Java apps: Free software, fast development](#), you:

- Downloaded and installed DB2 Express-C 9.5, Application Server,

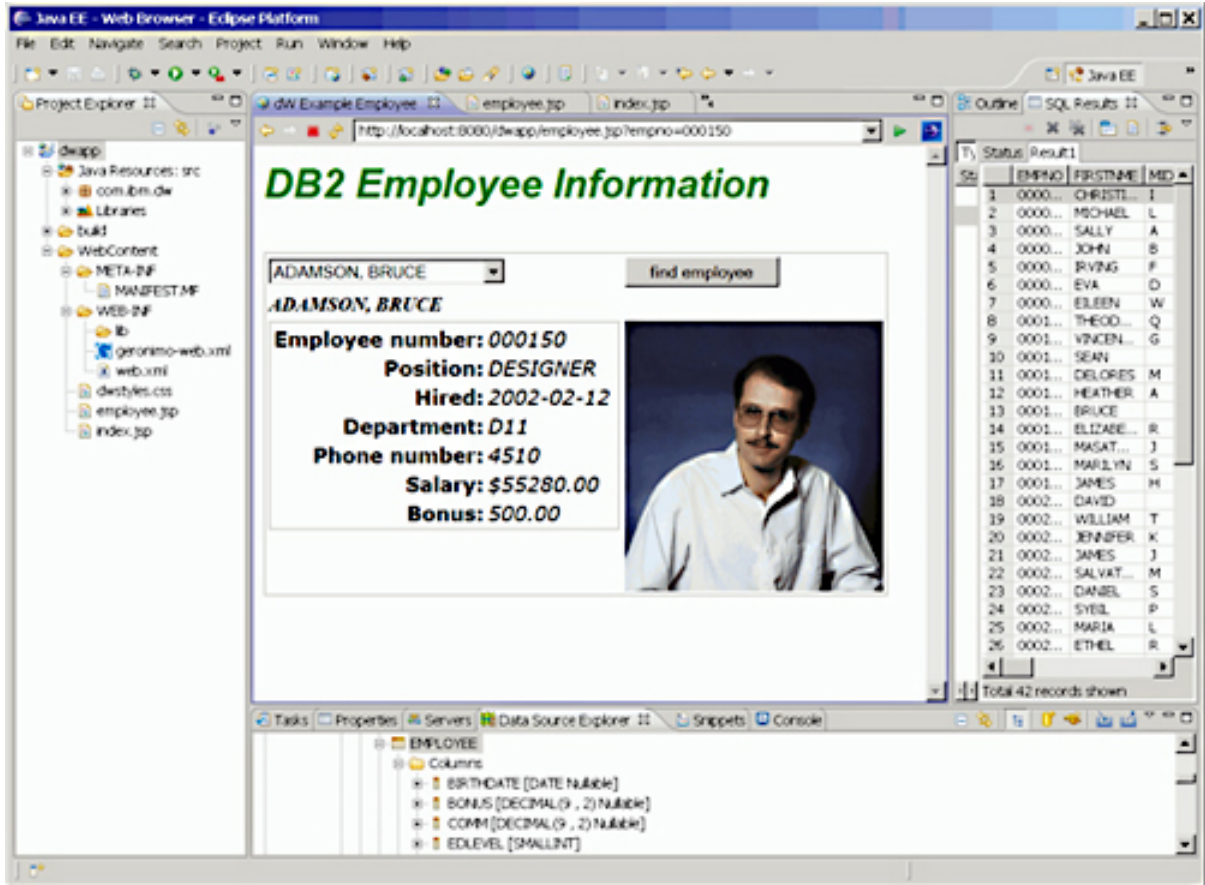
Eclipse, and the WTP Server Adapter for Application Server.

- Installed the sample database, provided by the DB2 Express-C 9.5 installation, containing employee information.
- Deployed a server-managed database pool in Application Server.
- Configured the integrated Data Source Explorer view in Eclipse to access the DB2 sample database's schemas and table contents.
- Coded a JSP application that accesses the DB2 sample database to list employee information, using Eclipse's Web Tools Platform (WTP) Web application development support.
- Deployed the JSP application to Application Server for testing, using the WTP Server Adapter for Application Server.
- Tested the JSP application by deploying it in an existing installation of Application Server.

Starting with the simple application you created in [Kick-start your Java apps: Free software, fast development](#), this tutorial adds a new JSP page that lets you interactively query employee information from the DB2 database. Unlike the original application, which lists information for all employees in a table, this JSP application displays the employee information one record at a time.

The displayed information includes a photograph of the employee, as shown in Figure 1:

Figure 1. The interactive employee-information-display application



To display the photograph, a specialized servlet queries the DB2 database for the binary image data stored in BLOB fields and then transfers the photograph to the browser as an image. The application-development and testing facilities of Eclipse's WTP help you create this servlet rapidly.

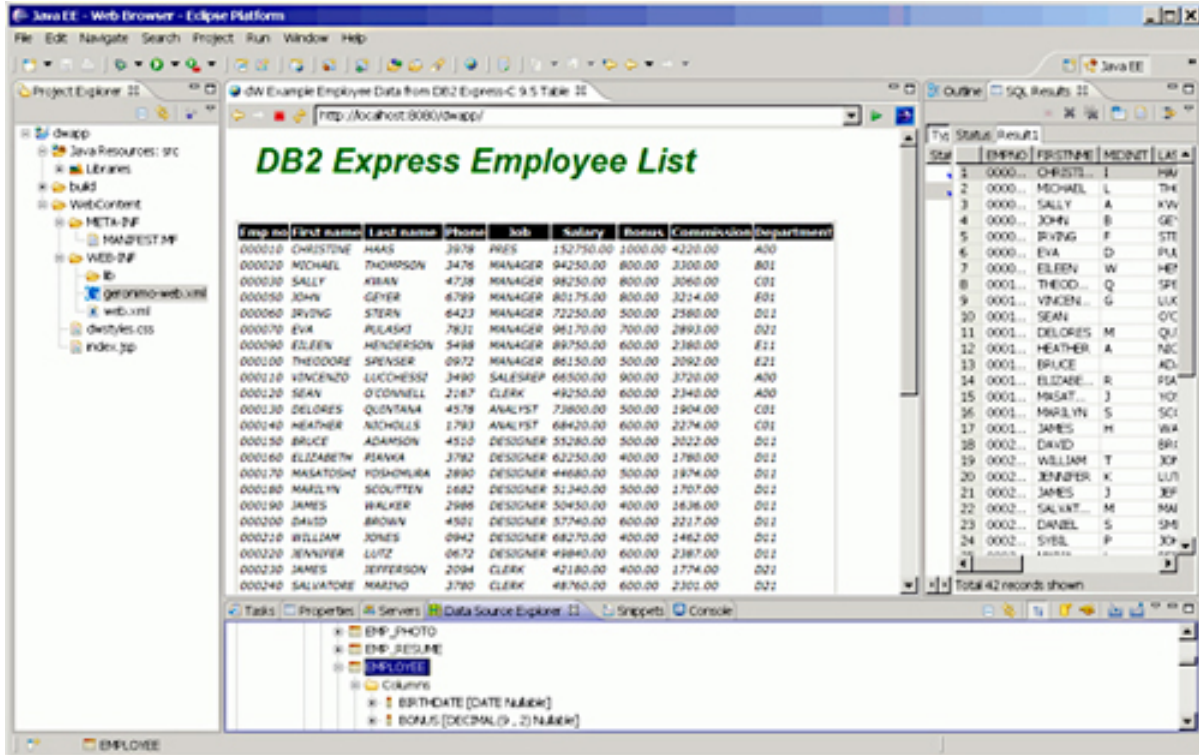
The interactivity of this design is limited, and it can't deal with a larger employee database, so I'll introduce an improved solution that addresses these shortcomings by applying modern lightweight Ajax techniques.

Section 3. Creating the JSP-based employee-information application

The first version of the application displays an employee's information, including his or her photo if available, as shown in [Figure 1](#). Users choose the employee information to be displayed from a drop-down list of employee names. In [Figure 1](#), the employee record for Bruce Adamson is selected and displayed.

The easiest way to create this application is to modify the project from [Kick-start your Java apps: Free software, fast development](#). Figure 2 shows the Eclipse IDE project from that tutorial:

Figure 2. The original dwapp Eclipse project showing an employee table



Starting from the dwapp project shown in Figure 2, you'll add or modify the Web application components in Table 1:

Table 1. The components of the employee information Web application

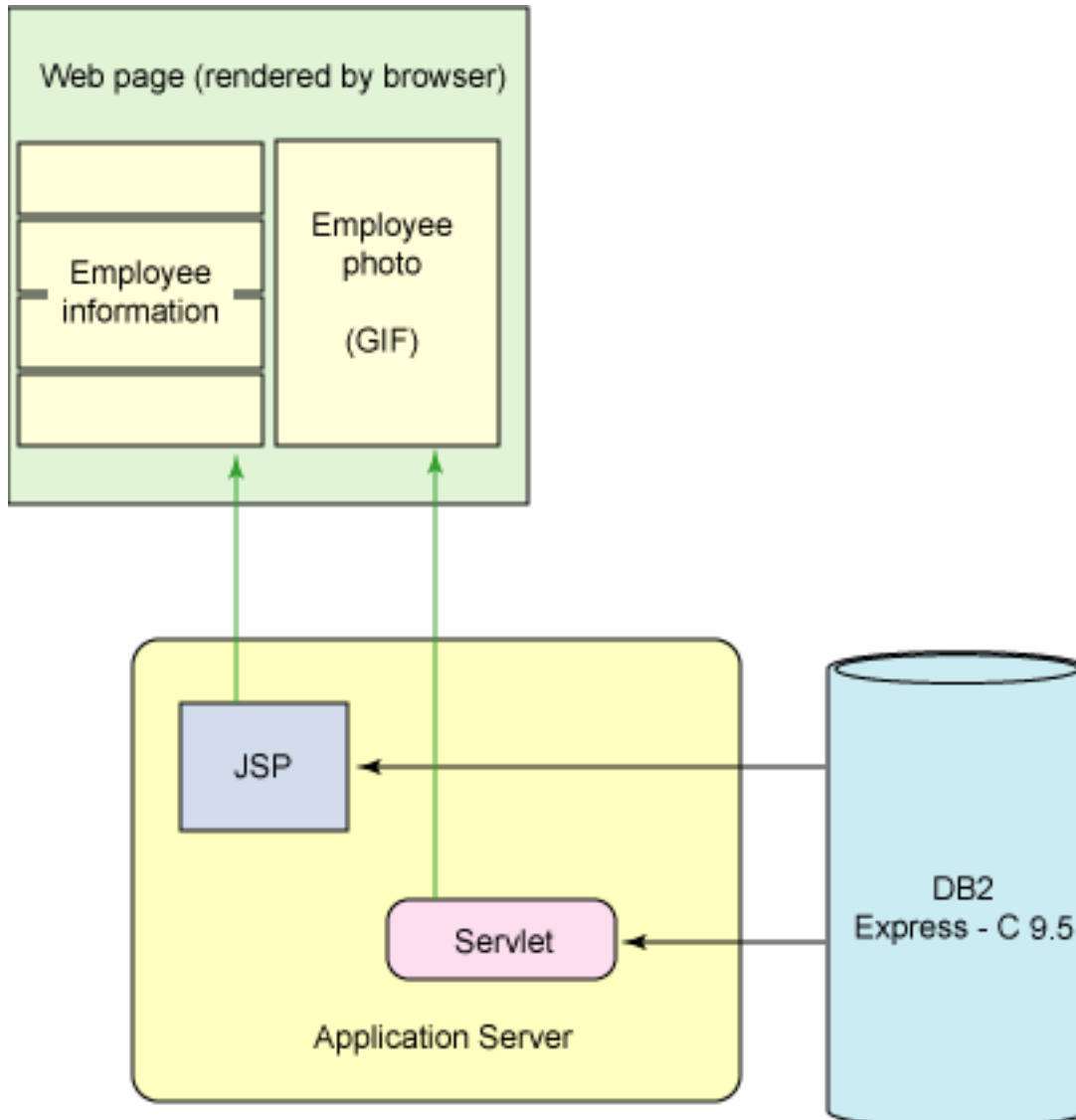
Component	Description
employee.jsp	A scriptless (free of embedded Java code) JSP page that handles user interaction and uses JSTL SQL tags to query the DB2 database for information. You'll add this component.
ShowPhoto.java servlet	A Java servlet that queries the DB2 database for employee photo information. Employee photos are stored in multiple formats in the database, within BLOB fields. Only the GIF-formatted photos are used. This servlet writes its output in a stream format that is compatible with the MIME type expected by a browser's display element. Database access is performed through a system data source obtained via the Java Naming and Directory Interface (JNDI), supported by the system-wide database pool configured on the Application Server from Kick-start your Java apps: Free software, fast

	development . You'll add this component.
<code>dwstyles.css</code>	You'll modify this stylesheet to handle additional formatting for the new <code>employee.jsp</code> page.

How the application components fit together

Figure 3 shows how the application components fit together:

Figure 3. Structure of the employee-information application



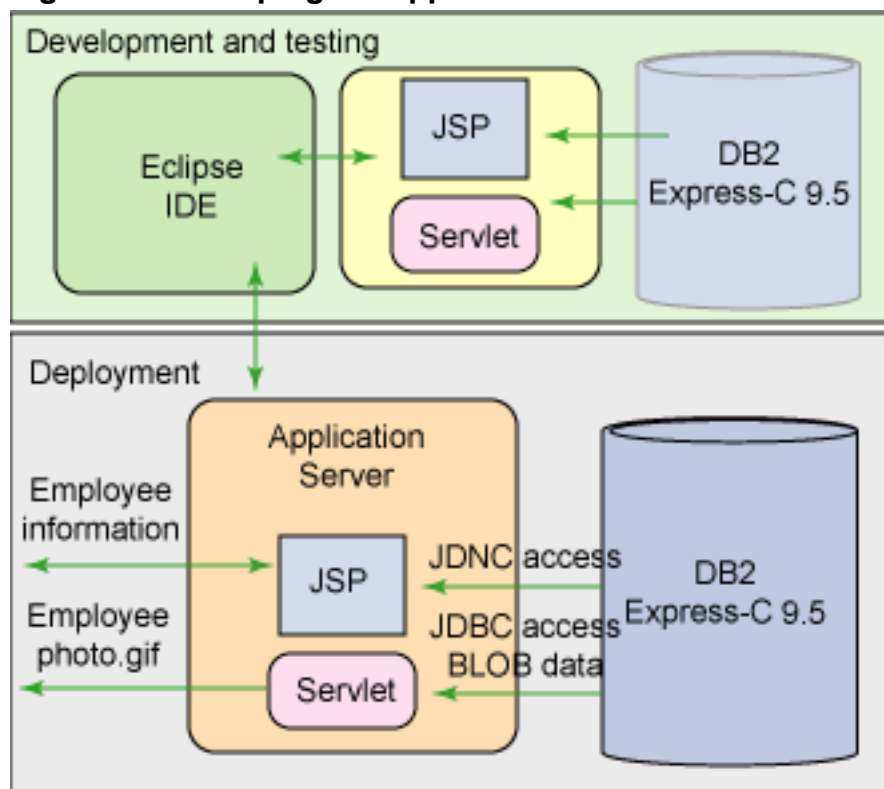
The `employee.jsp` component generates the Web page that contains the user interface. This generated Web page is rendered on the user's browser. Users interact with the rendered page to query for employee information by submitting an

HTML form. The JSP processes the submitted form data and fetches employee information from the database. The JSP then generates and returns a Web page with the information. The employee photo in GIF format is generated separately, by the ShowPhoto servlet. This servlet obtains the image data from the same DB2 Express-C 9.5 database.

Using the Kick-start your Java apps suite to code, test, and deploy applications

Figure 4 shows how the Kick-start suite helps in creating this application:

Figure 4. Developing the application with free tools



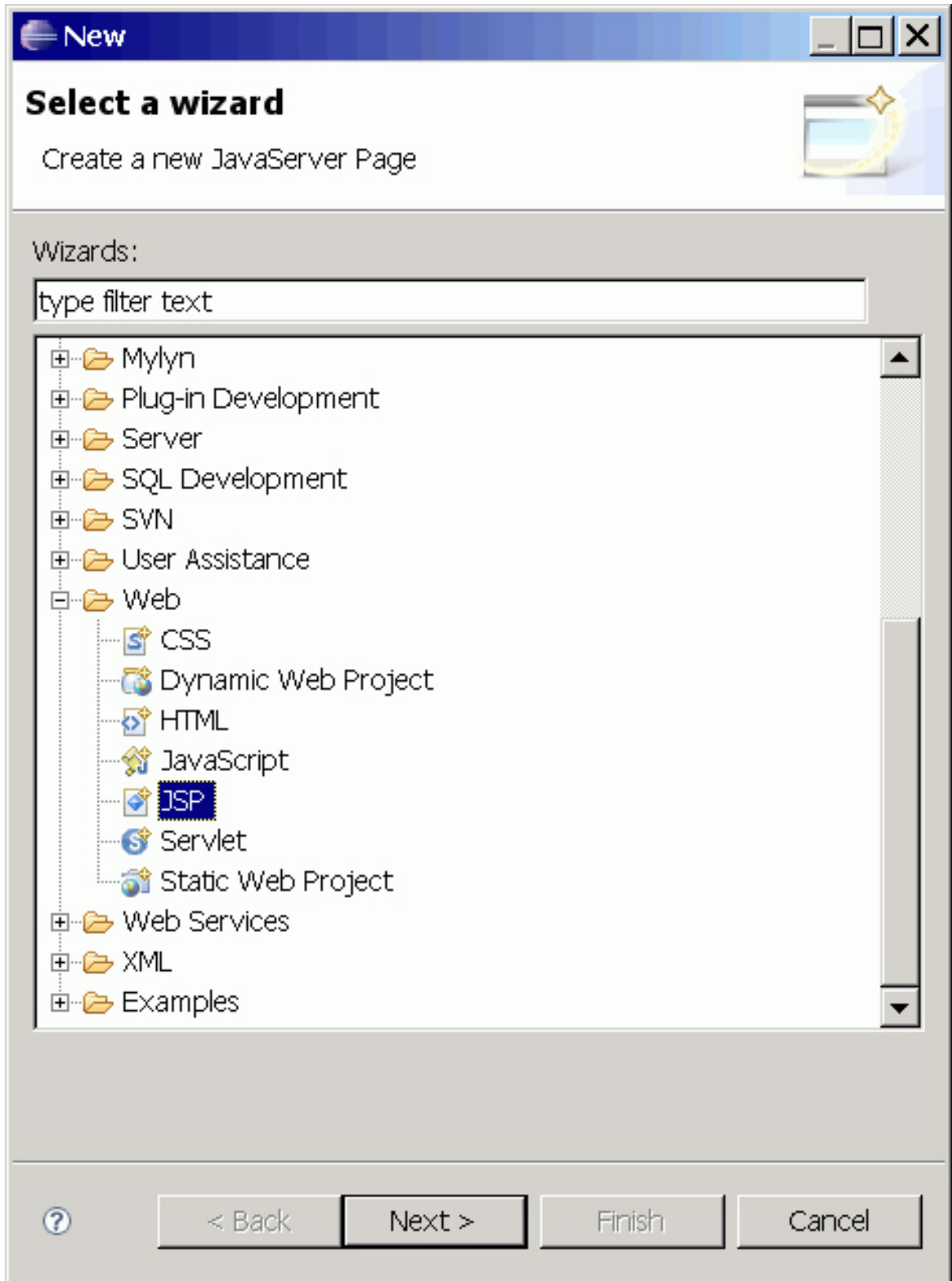
You'll code the application in Eclipse and use an instance of Application Server, controlled by Eclipse, to deploy and test the application.

The JSP component of the application accesses the DB2 Express-C 9.5 sample database through a JDBC pool managed by Application Server. The application's servlet component also uses the same managed pool to access BLOB fields from the database, transforming them into bitmap graphic information to be displayed as employee photographs in a browser.

The employee-selection JSP

Starting from the Eclipse dwapp project shown in [Figure 2](#), right-click the WebContent folder in the Navigator view and select **New > Other...** (or press **Ctrl+N**). Select **Web > JSP** from the New wizard, shown in Figure 5:

Figure 5. Creating a new JSP using the Eclipse New wizard



Click **Next** and then enter `employee.jsp` for the name of the new JSP. Click **Finish** to complete generating the initial JSP template.

Now, edit the generated JSP by adding the boldfaced code shown in Listing 1:

Listing 1. The `employee.jsp` component for handling user-interface interactions

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<link rel="stylesheet" type="text/css" href="dwstyles.css"/>
<title>dW Example Employee Information from DB2 Express C Table</title>
</head>
<body>
    <h1>DB2 Employee Information</h1>
    <br/>
    <c:url value="/employee.jsp" var="empURL"/>
    <form action="{empURL}" method="get">
    <table>
    <tr>

    <td colspan="2">
    <select name="empno">
    <sql:query var="employees" dataSource="jdbc/DataSource">
        SELECT * FROM EMPLOYEE ORDER BY LASTNAME
    </sql:query>

    <c:forEach var="employee" items="{employees.rows}">
        <option value="{employee.empno}">{employee.lastname},
    </c:forEach>
    </select>
    </td>
    <td>
    <input type="submit" value="find employee"/>
    </td>
    </tr>

    <c:if test="{!(empty param.empno)}">
    <tr>
    <sql:query var="selected" dataSource="jdbc/DataSource">
        SELECT * FROM EMPLOYEE WHERE EMPNO='{param.empno}'
    </sql:query>
    <c:forEach var="employee" items="{selected.rows}">
    <tr>
    <td colspan="3" class="name">
        {employee.lastname}, {employee.firstname}
    <br/>
    </td>
    </tr>
    <tr>
    <td colspan="2" valign="top">
    <table>
    <tr>
    <td class="label">Employee number:</td><td>{employee.empno}</td>
    </tr>
    <tr>
```

```

        <td class="label">Position:</td><td>${employee.job}</td>
    </tr>
    <tr>
        <td class="label">Hired:</td><td>${employee.hiredate}</td>
    </tr>
    <tr>
        <td class="label">Department:</td><td>${employee.workdept}</td>
    </tr>
    <tr>
        <td class="label">Phone number:</td><td>${employee.phoneno}</td>
    </tr>
    <tr>
        <td class="label">Salary:</td><td>$$${employee.salary}</td>
    </tr>
    <tr>
        <td class="label">Bonus:</td><td>${employee.bonus}</td>
    </tr>
</table>
</td>
<td>
    <c:url value="/ShowPhoto" var="photoURL">
    <c:param name="empno" value="${param.empno}" />
    </c:url>
    
</td>
</c:forEach>
</tr>
</c:if>
</table>
</form>
</body>
</html>

```

Following JSP best practices, `employee.jsp` contains no embedded Java code; all dynamic code generation is handled through JSTL and Expression Language (EL).

How `employee.jsp` operates

Figure 1 shows the layout of `employee.jsp`. A drop-down list — an HTML `<select>` element — holds the list of employee names.

The user selects any one of the employees from the list and clicks the **find employee** button. Detailed information on the selected employee is then displayed, together with a photo of the employee if available. (Not all employees have a photograph available in the DB2 sample database.)

In `employee.jsp`, the drop-down list is generated using the JSTL code in Listing 2:

Listing 2. Dynamic generation of employee names for selection

```

<select name="empno">
  <sql:query var="employees" dataSource="jdbc/DataSource">
    SELECT * FROM EMPLOYEE ORDER BY LASTNAME
  </sql:query>
  <c:forEach var="employee" items="${employees.rows}">

```

```

    <option value="{employee.empno}">${employee.lastname},
    ${employee.firstname}</option>
  </c:forEach>
</select>

```

You use the JSTL SQL `<sql:query>` tag to perform a query that selects all employee records from the DB2 database. This is similar to the query you used in the [Kick-start your Java apps: Free software, fast development](#) example. This time, the returned employee information is in alphabetical order by last name (thanks to the `ORDER BY LASTNAME` clause in the SQL query) to facilitate the selection of names from the drop-down list.

You perform the SQL query against the specified `jdbc/DataSource`. This is configured in the application's `geronimo-web.xml` deployment plan and points to the system-wide database pool called `dwDataSource`, which is configured on Application Server.

You then use the `<c:forEach>` JSTL iteration tag to iterate through the list of selected employees one row at a time. One HTML `<option>` is generated for each employee. The value of the `<option>` is the employee number, and the text shown is the employee's last and first names. For example, the `<option>` generated for Bruce Adamson is:

```
<option value="000150">ADAMSON, BRUCE</option>
```

The entire `<select>` element exists inside an enclosing HTML `<form>`, shown in boldface in Listing 3:

Listing 3. The `<select>` list inside the `<form>` element

```

    <c:url value="/employee.jsp" var="empURL"/>
<form action="{empURL}" method="get">
  <table>
  <tr>

  <td colspan="2">
    <select name="empno">
      ...
    </select>
  </td>
  <td>
    <input type="submit" value="find employee"/>
  </td>
  </tr>
  ...
</table>
</form>

```

The `<c:url>` URL link-generation tag creates a correctly formatted URL for you, pointing back to this same `employee.jsp` for form processing. This URL is placed in a

variable called `empURL` and is used in the `<c:form>` tag's `action` attribute.

When a user selects an employee — for example, Bruce Adamson (`empno=000150`) — and clicks the **find employee** button, the form information is submitted back to `employee.jsp` for processing with this URL:

```
http://localhost:8080/dwapp/employee.jsp?empno=000150
```

Processing HTML form submission

The `employee.jsp` page has two roles:

- To display the list of employees for selection
- To handle the form submission once an employee is selected

To determine if a form submission has been made, you need to check for the existence of an `empno` HTTP GET parameter. You can use the empty operator in JSTL for this check. The code that renders the employee information and photo is bracketed by a `<c:if>` tag that performs this test. This code fragment is shown in Listing 4:

Listing 4. Testing for the existence of the `empno` form parameter

```
        <c:if test="${!(empty param.empno)}">
<tr>
  <sql:query var="selected" dataSource="jdbc/DataSource">
    SELECT * FROM EMPLOYEE WHERE EMPNO='${param.empno}'
  </sql:query>
  <c:forEach var="employee" items="${selected.rows}">
    . . .
  </c:forEach>
</tr>
</c:if>
```

If the incoming HTTP request does not have the `empno` parameter, none of the employee-details information is generated.

To generate the employee-details information, another `<sql:query>` is performed against the `EMPLOYEE` table. This `SELECT` now pinpoints a single employee with the corresponding employee number. The JSTL implicit object, `param`, is used to render the value of the incoming `empno` parameter within the `SELECT` statement, shown in Listing 5:

Listing 5. Obtaining the record for a single employee using a dynamically generated SQL query

```
<c:if test="${!(empty param.empno)}">
  <tr>
    <sql:query var="selected" dataSource="jdbc/DataSource">
      SELECT * FROM EMPLOYEE WHERE EMPNO='${param.empno}'
    </sql:query>
    <c:forEach var="employee" items="${selected.rows}">
      ...
    </c:forEach>
  </tr>
</c:if>
```

Even though a `<c:forEach>` construct is used here, you are working with only a single row (because each employee has a unique employee number). This JSTL construct simplifies access to the row data in the returned result set. To generate the textual information, the employee variable set up by `<c:forEach>` can be used directly. For example, the following EL expression generates the employee's phone number within the `<c:forEach>` construct:

```
${employee.phoneno}
```

To generate the employee photograph, you use an HTML `` tag. The URL to display the image is generated using the JSTL `<c:url>` tag:

```
<c:url value="/ShowPhoto" var="photoURL">
  <c:param name="empno" value="${param.empno}" />
</c:url>

```

For example, the effective generated `` tag for Bruce Adamson (whose `empno` is 00150) is:

```

```

This means that the following URL must generate a stream of bits that represents the employee's photograph:

```
http://localhost:8080/dwapp/ShowPhoto?empno=00150
```

The `ShowPhoto.java` servlet generates the photograph bit stream.

Showing photographs from DB2 BLOB fields

The employee photographs are stored in the `EMP_PHOTO` table of the DB2 sample database. Table 2 describes this table's fields:

Table 2. The fields of the EMP_PHOTO table

Field name	Description
empno	The employee number of the corresponding employee.
photo_format	The format of the photograph. Photos are stored in multiple formats to facilitate display without complex conversion code. This application uses only the GIF-formatted photos.
picture	The binary bits of the picture, stored in a BLOB field. Maximum size of this field is 102,400 bytes.

To create the ShowPhoto.java servlet in the Eclipse IDE, first highlight **WebContent** and select **New > Other...** (or press **Ctrl+N**). Select **Web > Servlet** in the New wizard. Enter `com.ibm.dw` for Java Package and `ShowPhoto` for class name, as shown in Figure 6:

Figure 6. Creating a new servlet with Eclipse WTP's Create Servlet wizard

Create Servlet

Specify class file destination.

Project:

Folder:

Java package:

Class name:

Superclass:

Use existing Servlet class

Class name:

Click **Next**. On the next screen, you can set the URL mapping for the servlet. This URL mapping is automatically placed into the application's web.xml deployment descriptor as a `<servlet-mapping>` element. In this case, you can keep the default mapping to `/ShowPhoto`. Click **Finish**.

If you look at the Navigator view, you'll see the `com.ibm.dw.ShowPhoto.java` file under a `src` directory, outside of the `WebContent` folder. Eclipse WTP keeps the source code here for easy maintenance and organization but copies the compiled class files into the correct location under the `WEB-INF/classes` directory for Web application deployment automatically.

The steps you've just completed generate a skeletal servlet in `ShowPhoto.java`. Now edit the code by adding the boldfaced lines shown in Listing 6:

Listing 6. Additional code for the ShowPhoto.java servlet

```

package com.ibm.dw;

import java.io.IOException;
import java.io.InputStream;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.sql.DataSource;

/**
 * Servlet implementation class for Servlet: ShowPhoto
 *
 */
public class ShowPhoto extends javax.servlet.http.HttpServlet
    implements javax.servlet.Servlet {
    public ShowPhoto() {
        super();
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        String query = "select picture from emp_photo where
            photo_format='gif' and empno='"
            + request.getParameter("empno") + "'";

        DataSource dsource = null;
        Statement stmt= null;
        Connection conn =null;
        ResultSet rset = null;
        try {
            InitialContext context = new InitialContext();
            dsource =
                (DataSource)context.lookup("java:comp/env/jdbc/DataSource");
            conn = dsource.getConnection();
            stmt = conn.createStatement();
            rset = stmt.executeQuery(query);

            if (rset.next()) {
                byte[] buffer = new byte[32000];
                int size=0;
                InputStream istream;
                istream = rset.getBinaryStream(1);
                response.reset();
                response.setContentType("image/gif");
                while((size= istream.read(buffer))!= -1 ) {
                    response.getOutputStream().write(buffer,0,size);
                }

                response.flushBuffer();
                istream.close();
            }
        } catch(NamingException e) {

        } catch(SQLException e) {

        }
        finally {

```

```
        try {  
            if (rset != null)  
                rset.close();  
            if (stmt != null)  
                stmt.close();  
            if (conn != null)  
                conn.close();  
        } catch (SQLException ex ) {}  
    }  
}  
  
protected void doPost(HttpServletRequest request, HttpServletResponse response)  
    throws ServletException, IOException {  
    doGet(request, response);  
}  
}
```

Using Eclipse's automatic import organizer

You can cut and paste from this tutorial's source code (see [Download](#)) if you want. Or, if you are entering the code by hand, you don't need to enter the extensive import list. Eclipse can search all the project libraries available for your import symbols. Right-click within the code editor and select **Source > Organize Imports** (or press **Ctrl+Shift+O**). This scans your source code for all unresolved imports and adds all the `import` statements at the beginning of the source. If Eclipse comes across any ambiguity in resolving the imports, it prompts you to select the library package where the symbol should be imported from.

This feature can save you from a lot of tedious manual searching during daily coding activities.

Operation of the ShowPhoto servlet

The incoming HTTP request contains a parameter named `empno`. This parameter is used to fetch the correct photo from the `EMP_PHOTO` table. The query is set up in:

```
String query =  
"select picture from emp_photo where photo_format='gif' and empno='"  
+ request.getParameter("empno") + "'";
```

The query runs against the `jdbc/DataSource` that has been set up in the deployment plan, `geronimo-web.xml`. This data source links to the system-wide database pool called `dwDataSource` that is configured on Application Server. For servlets hosted in Application Server, you can perform the lookup using JNDI:

```
InitialContext context = new InitialContext();  
dsource = (DataSource)context.lookup("java:comp/env/jdbc/DataSource");
```

Once the data source has been obtained, you can then execute the query using standard JDBC coding:

```
conn = dsource.getConnection();
stmt = conn.createStatement();
rset = stmt.executeQuery(query);
```

To fetch the picture field as a binary input stream, use the `getBinaryStream()` method on field #1 — the `picture` field in the JDBC `ResultSet` from the `SELECT` query:

```
InputStream istream;
istream = rset.getBinaryStream(1);
```

You need to set the response of the servlet to the correct `contentType` to ensure that the browser can interpret the `image/gif` mime type property:

```
response.reset();
response.setContentType("image/gif");
```

All that is left to do is to read from the binary stream of the picture field and write it out to the response's output stream:

```
while((size= istream.read(buffer))!= -1 ) {
    response.getOutputStream().write(buffer,0,size);
}
```

This completes the `ShowPhoto.java` servlet. Next, you'll set up the application's initial welcome page.

Setting up the Web application's welcome page

By default, the Web application shows `index.jsp` as the welcome page. This is the page Application Server displays when the following URL is requested:

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

`index.jsp` currently contains the old employee-list application. You need to change the welcome page to point to `employee.jsp`. Open the `web.xml` file from the Navigator view and then look for the `<welcome-file-list>` element. This element contains the files that Application Server looks for when displaying the welcome page for the application. Make the following change (shown in boldface in

Listing 7) to the element:

Listing 7. Configuring employee.jsp as the application welcome page

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

Modifying the stylesheet

You also need to modify the original dwstyles.css stylesheet to include additional style elements, shown in boldface in Listing 8, used by employee.jsp:

Listing 8. New elements added to dwstyles.css stylesheet

```
@CHARSET "ISO-8859-1";
h1 {
  font-family: arial;
  font-size: 38;
  align: left;
  font-weight: bold;
  font-style: italic;
  color: green;
}
h2 {
  font-family: serif, times;
  font-size: 18;
  align: left;
  color: blue;
}
th {
  font-family: verdana, arial;
  font-size: 13;
  font-weight: bold;
  align: left;
  background-color: black;
  color: white;
}

td {
  font-family: verdana, arial;
  font-size: 12;
  font-style: italic;
  text-align: left;
}

table {
  border-style: solid;
  border-width: thin;
}

.label {
  font-size: 16;
  font-weight: bold;
  font-style: normal;
  text-align: right;
}
```

```
}  
  
.name {  
    font-size: 24;  
    font-weight: bold;  
    font-style: italic;  
    font-family: serif, times roman;  
}
```

Trying out the employee-information application

Before you can run the application, you need to build the project by compiling the Java code and creating the WAR file. Right-click `dwapp` in the Navigator view and select **Build Project**. Your Eclipse project might be set to build automatically, in which case you don't need to build manually each time. Check under the Project menu on Eclipse's menu bar to see if the project is set up for auto build.

To deploy your new application to Application Server and test it, right-click the `dwapp` project in the Navigator view and select **Run As > Run on Server**. Double check in the Servers view that the `dwapp` project has Synchronized status. If not, just right-click on the `dwapp` project in the Servers view and select **Publish**. Eclipse will then publish the latest WAR to the server and be in a synchronized state.

If Application Server isn't already running, Eclipse starts it. The application is bundled as a WAR file and deployed to Application Server.

Start an internal (or external) Web browser and access the application's URL: `http://localhost:8080/dwapp/`. (Look for the **Open Web Browser** button on the Eclipse toolbar for an internal browser.)

You can now try selecting different employees in the database to see their details. If a photo is available for an employee you select, it is also displayed.

Section 4. Improving the employee-information application

The version of the employee-information system you just finished is adequate for a very small company — one with up to 30 or so employees. To handle a company with a couple of hundred employees, using a single drop-down list for employee selection isn't practically feasible. Furthermore, shuttling the information of hundreds of employees from DB2 to the JSP each time the selection list is shown is both inefficient and slow.

In this section, you'll prepare to build a second cut of the information system that aims to improve this situation with an improved user-interface design.

Page-based user experience versus visual dynamic update

The JSP-based version handles employee searches by returning a new Web page. This is the standard action of JSP, increasingly being called *legacy page-based design*. The interaction sequence is:

1. The user makes a selection on a form (employee selection in this case).
2. The form data is submitted to a form processor back on the server (employee.jsp in this case).
3. The form processor returns a new HTML page to the user's browser.
4. The user's browser displays this new response page.

As a result, the user experience is rather clunky. The data-submission delay and the wait for response are highly perceptible. Changes are shown in an all-or-nothing fashion — even if only a small portion of the HTML page needs changing, the entire page is fetched from the server and re-rendered. In a nutshell, users' experience with typical Web applications is very unlike their experience with local applications such as word processors and calendars.

To improve the user experience over high-speed connections, a more dynamic approach can be taken that includes this series of interactions:

1. The user makes a selection on the HTML page.
2. Without submitting a form, JavaScript code on the browser sends the query entered by the user to the server; this happens in the background and is invisible to the user.
3. Without waiting for a reply, the browser continues to display the same page and allow user interaction.
4. After a while, the response to the query arrives back from the server, and JavaScript code on the browser receives the response. This is still invisible to the user, who can continue to interact with the user interface.
5. Using modern dynamic HTML (DHTML), which lets the Web page be modified on-the-fly using JavaScript code, the page is dynamically updated with the data from the response. Only the display elements that

need to be changed are modified.

The user experience is vastly improved. The wait for server response, and the page-by-page update/refresh, are gone. Instead, the user is greeted with a familiar interactive experience. The application feels substantially more like a local application.

Ajax for more responsive user interaction

The main properties of this more interactive application are:

- **Asynchronicity:** The query is sent by browser-side code asynchronously, without waiting for the query to return; because of this asynchronicity, no user-interface element needs to be frozen or blocked.
- **JavaScript:** The only universally supported browser-side programming language is JavaScript. All of the application's user-interface interactions are orchestrated using JavaScript code running on the browser.
- **XML:** This is not so visible, but the query needs to be shuttled from the browser to the server with bundled parameters. The response can also contain complex data structures (such as a list of employee records, with fields of different sizes and types) that must be parsed by the browser's JavaScript code before use. One proven and tested way of shuttling this complex data over an HTTP connection is through XML. XML processing libraries can perform much of this data-shuttling work.

Therefore, this highly interactive user experience comes to you courtesy of Asynchronous JavaScript and XML — known more simply in the development community as Ajax.

Coverage of Ajax topics in detail is beyond this tutorial's scope. Visit the developerWorks [Ajax resource center](#), and check out the *Ajax for Java developers* series by Philip McCarthy (see [Resources](#)), an excellent resource for Ajax education. This tutorial focuses on how you can apply Ajax to improve the user interface of the employee-information application and how to build Ajax-based solutions using Eclipse, DB2 Express-C 9.5, and Application Server.

Redesigning the application with Ajax

Figure 7 shows the improved user interface of the Ajax-based application you're about to build:

Figure 7. Improved user interface for the employee-information application



This version includes a data-entry field and a drop-down box. Users can:

1. Enter as many leading letters of the employee's last name as they wish in the data-entry field.
2. Tab out of the data-entry field to populate the drop-down list dynamically with last names of all employees matching the entered data.
3. Select an employee from the drop-down list. When the user tabs out of the drop-down list, the application immediately updates all the employee information, including the photograph.
4. Immediately repeat steps 1 and 2 to see information from another employee.

Note that at no time does the user need to deal with form submits or page-response delays. This design also has the following differences from the JSP-based version:

- The number of selections in the drop-down list is substantially reduced, letting the application deal with a larger population of employees.
- Fewer entries in the drop-down list means that less data is transferred

from the DB2 database to Application Server, thereby improving overall application performance.

- The interactivity of the user interface is greatly improved, thanks to Ajax.
- The displayed page is a plain vanilla HTML page, not a generated JSP page.
- The application has no direct dependency on a Java EE container. It does not directly rely on JSP, servlets, or EJBs for its operation.

Because of its containerless design and use of plain vanilla HTML, this style of solution is often called *lightweight construction*. In fact, you can simply write JavaScript code on an HTML page and then manipulate user-interface elements dynamically and query back-end databases with it. You can construct the back end completely from alternative technology, such as the Spring lightweight framework or even a non-Java technology such as Ruby on Rails (see [Resources](#)).

Section 5. Introducing Direct Web Remoting

Detailed study of low-level Ajax coding is not for the faint of heart. All Ajax code makes use of a built-in object of all modern browsers — including the latest versions of Firefox, Safari, and Internet Explorer — called `XMLHttpRequest`. JavaScript code can make calls to the server asynchronously through this object.

Unfortunately, different browsers implement the `XMLHttpRequest` object slightly differently. Further complicating the mix, different versions of same browser (Internet Explorer, for example) implement the object differently. Add to this the fact that JavaScript itself is often different for different browsers and versions, and you have a headache-in-waiting.

The complexity is compounded by lack of standardization on how data should be shuttled over XML requests and responses between the browser code and the server back end. Multiple competing alternatives exist, and some frustrated developers simply roll their own. To appreciate some of the potential complications, see Philip McCarthy's *Ajax for Java developers* series (see [Resources](#)).

Thankfully, as a Java developer, you can use a library stack that isolates most of the differences I've described and provides a workable and easy-to-code Ajax solution today. This library is called Direct Web Remoting (DWR) (see [Resources](#)).

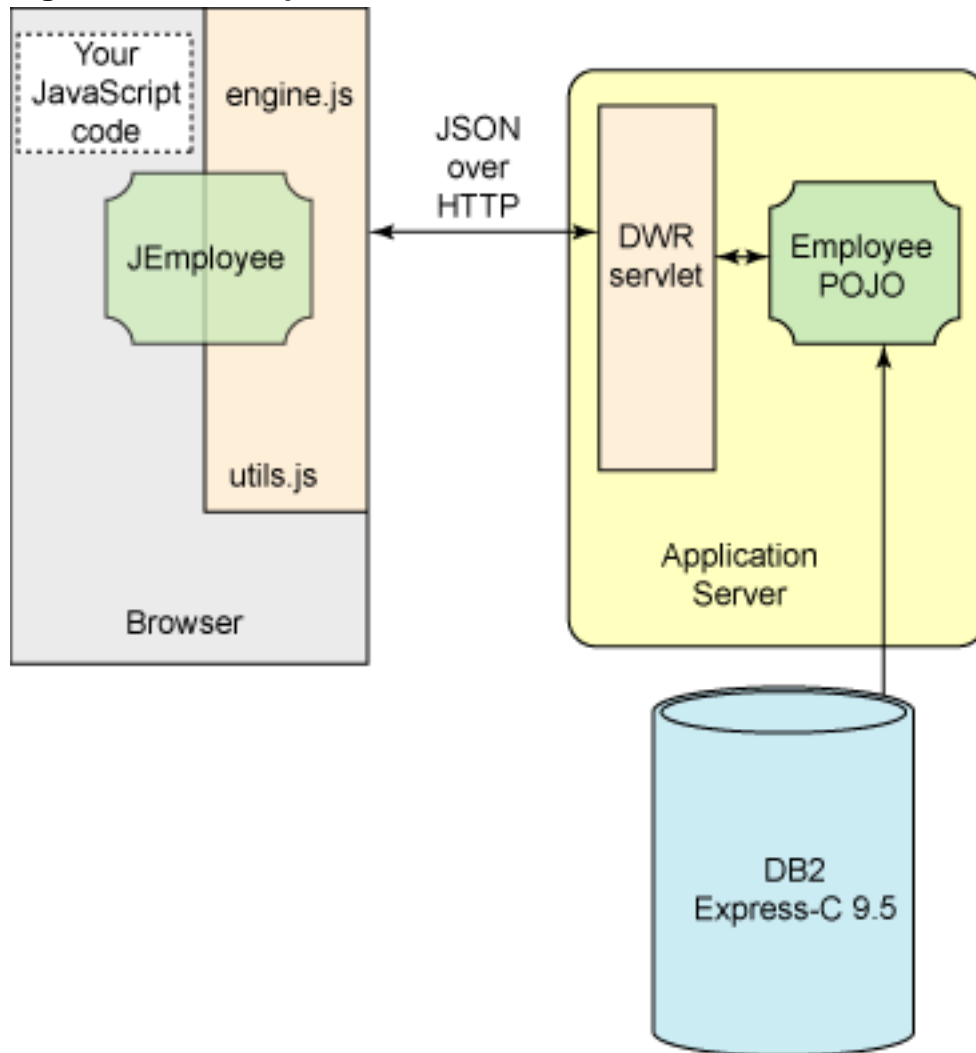
DWR is the brainchild of Joe Walker and Mark Goodwin. It is a JavaScript code library and Java servlet combination that lets you transparently make remote calls to

methods of server-side Java objects from a Web page, using Ajax. The evolutionary focus of this library has been ease of use for developers.

DWR is released under the Apache Software License 2.0, the same liberal license embraced by successful major open source projects such as Apache Tomcat, Apache Ant, and Apache Geronimo. You can create applications, commercial or otherwise, using the DWR library with no obligation to contribute back to the project. See the license terms for full details.

Figure 8 shows how DWR simplifies your Ajax programming life:

Figure 8. The components of DWR



How DWR works

DWR consists of two components:

- A JavaScript library that runs on the user's browser (utils.js and engine.js).
- A servlet that runs on Application Server.

The JavaScript code that you write can make use of DWR's JavaScript library. This library provides the following essential features:

- It insulates you from browser differences in handling Ajax calls.
- It facilitates certain DHTML operations when dynamically modifying Web page elements that would otherwise require highly complex JavaScript and DHTML coding.

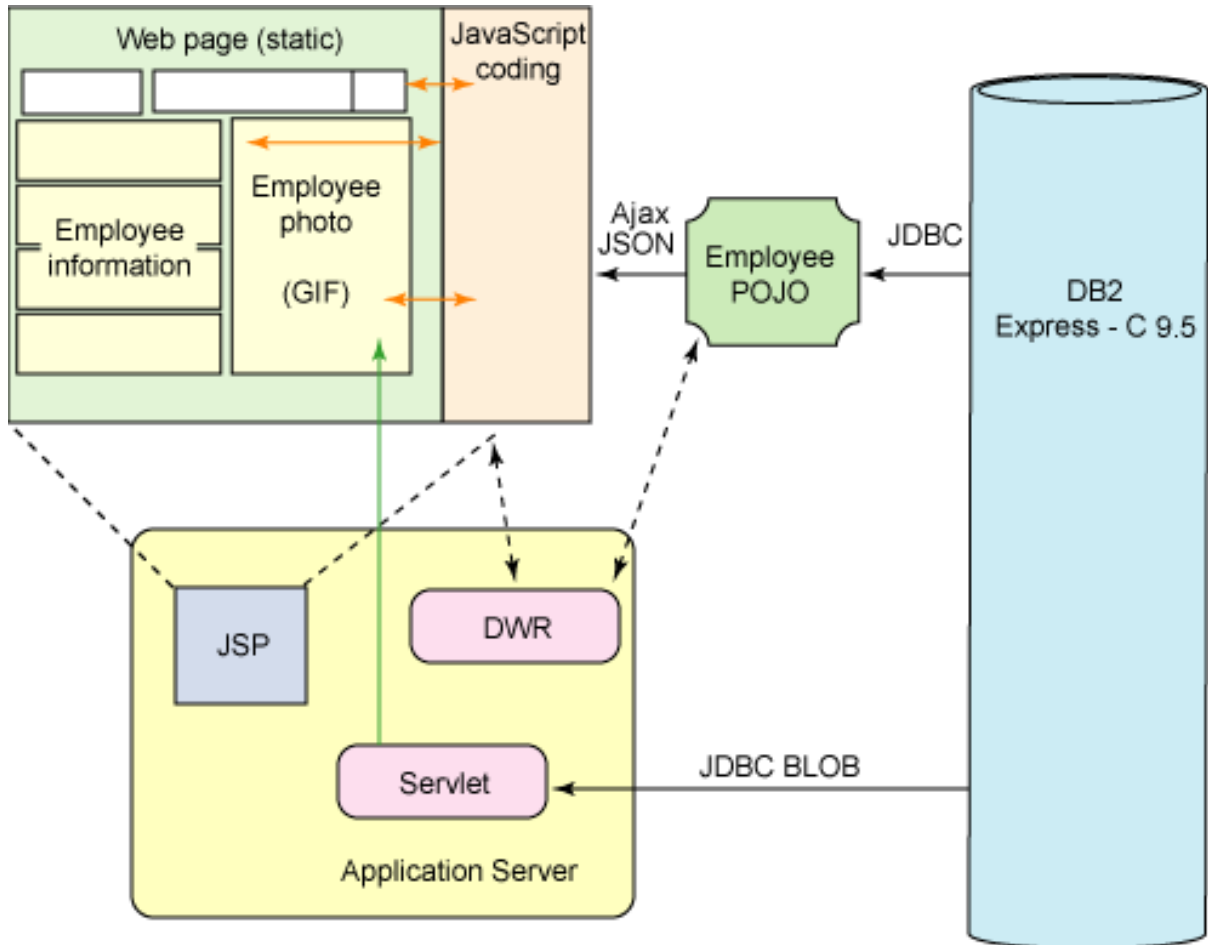
More important, DWR lets you *remote* certain Java objects. In Figure 8, the server-side Java `Employee` object (a Plain Old Java Object, or POJO) is remotod. DWR makes the object appear to be available to the JavaScript code running on the user's browser. The JavaScript code can access the server-side `Employee` objects as if they were local JavaScript objects. To take advantage of this remotod feature, the server-side `Employee` object gets its data through a direct JDBC connection to the DB2 Express-C 9.5 employee data. Because the object is remotod, this data is automatically made available to the JavaScript code on the client's browser.

Underneath the hood, DWR makes asynchronous calls through the `XMLHttpRequest` object provided by the browser to its own servlet running on Application Server. This servlet accesses the server-side `Employee` object and sends required data back through an HTTP response formatted in JavaScript Object Notation (JSON) format. JSON format is a JavaScript-friendly serialization format that lets the JavaScript interpreter in the Web browser create native JavaScript objects (mirroring the server-side `Employee` object) directly.

Toward a lightweight design

The new employee-information application, using DWR to implement Ajax, contains simpler and more direct code than its predecessor. Figure 9 shows how this new design works:

Figure 9. The new lightweight employee-information application



In Figure 9, the page displaying the employee information is now an HTML page, not a JSP. Application Server acts only as a Web server in this case, serving the static Web page to the user's browser. It does not act as a JSP container. The JavaScript code in the HTML page accesses the `Employee` objects, which are remoted using DWR. The server-side `Employee` objects need not be hosted inside a conventional Java EE container.

In this design, Application Server is used for only three purposes:

- To act as a static Web server and serve the static HTML page containing Ajax code
- To support the execution of DWR
- To support the ShowPhoto servlet for display of employee photos

Section 6. Experimenting with DWR

Application Server includes the stable version 1.1.4 of DWR by default. This example takes advantage of that fact and does not require an additional DWR download or installation to work. It is recommended that you use only the Application Server default library, version 1.1.4, with this example. Later versions may be available through the DWR download site for your own exploration.

To experiment now with DWR and better understand its operation, go to [Download DWR](#) and download only the `dwr.war` file for DWR version 1.1.4 (make sure you get the right version). The `dwr.war` file is a demonstration application that also contains some useful diagnostic and unit tests.

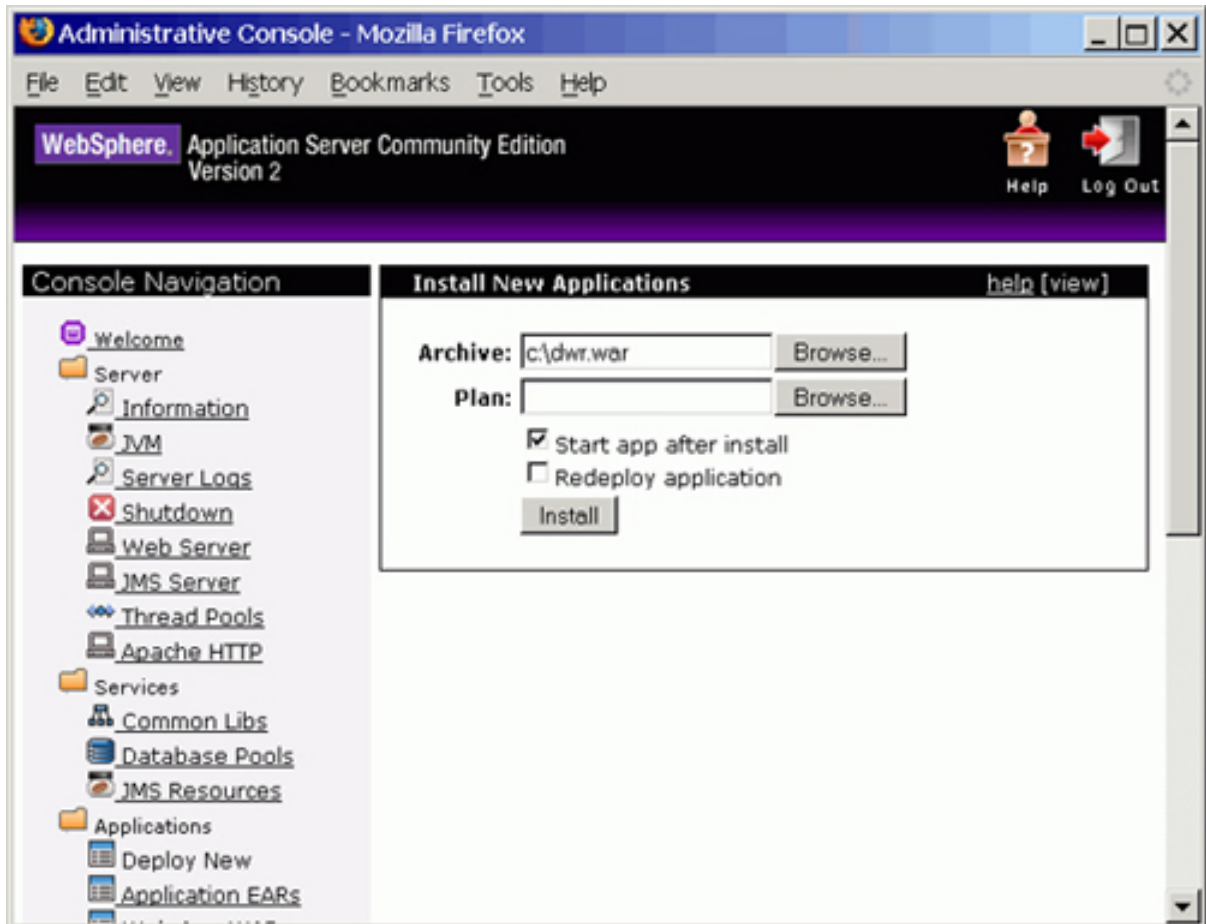
First encounter with Ajax on DWR

To see how quickly you can get started, first deploy the sample Web application in the `dwr.war` file. Make sure Eclipse is not running, and then start up Application Server.

Next, fire up the Application Server Administrative Console and deploy the `dwr.war` application:

1. With Application Server running, go to the URL `http://localhost:8080/console`.
2. Log on to the console using *system* for username and *manager* for password.
3. In the Console Navigation menu on the left, click on **Applications > Deploy New**.
4. Browse to your downloaded `dwr.war` file (using the **Browse** button) and click **Install**. Leave the Plan box empty. See Figure 10:

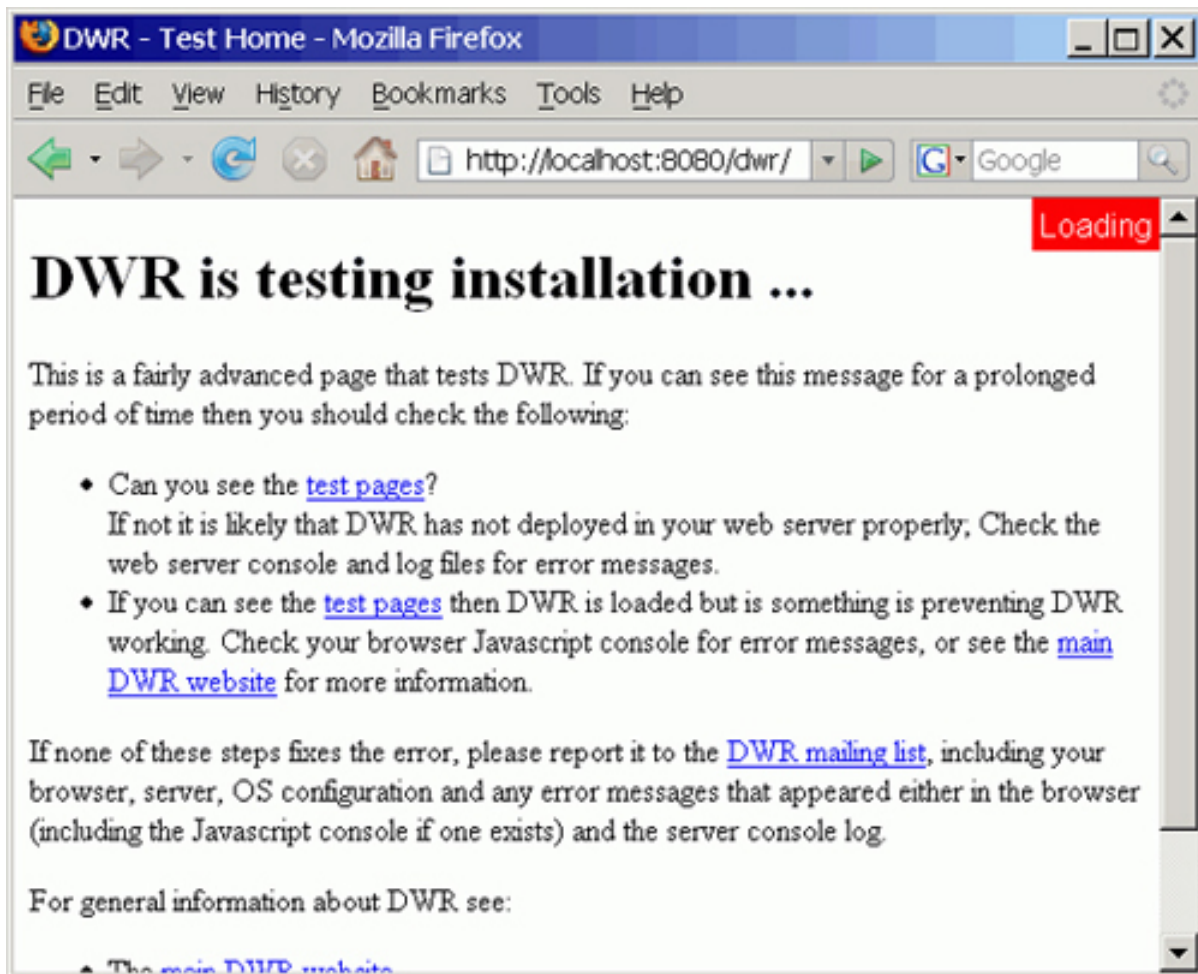
Figure 10. Deploying the DWR test Web application



Now, point your browser to the DWR test application's front page:
<http://localhost:8080/dwr/>.

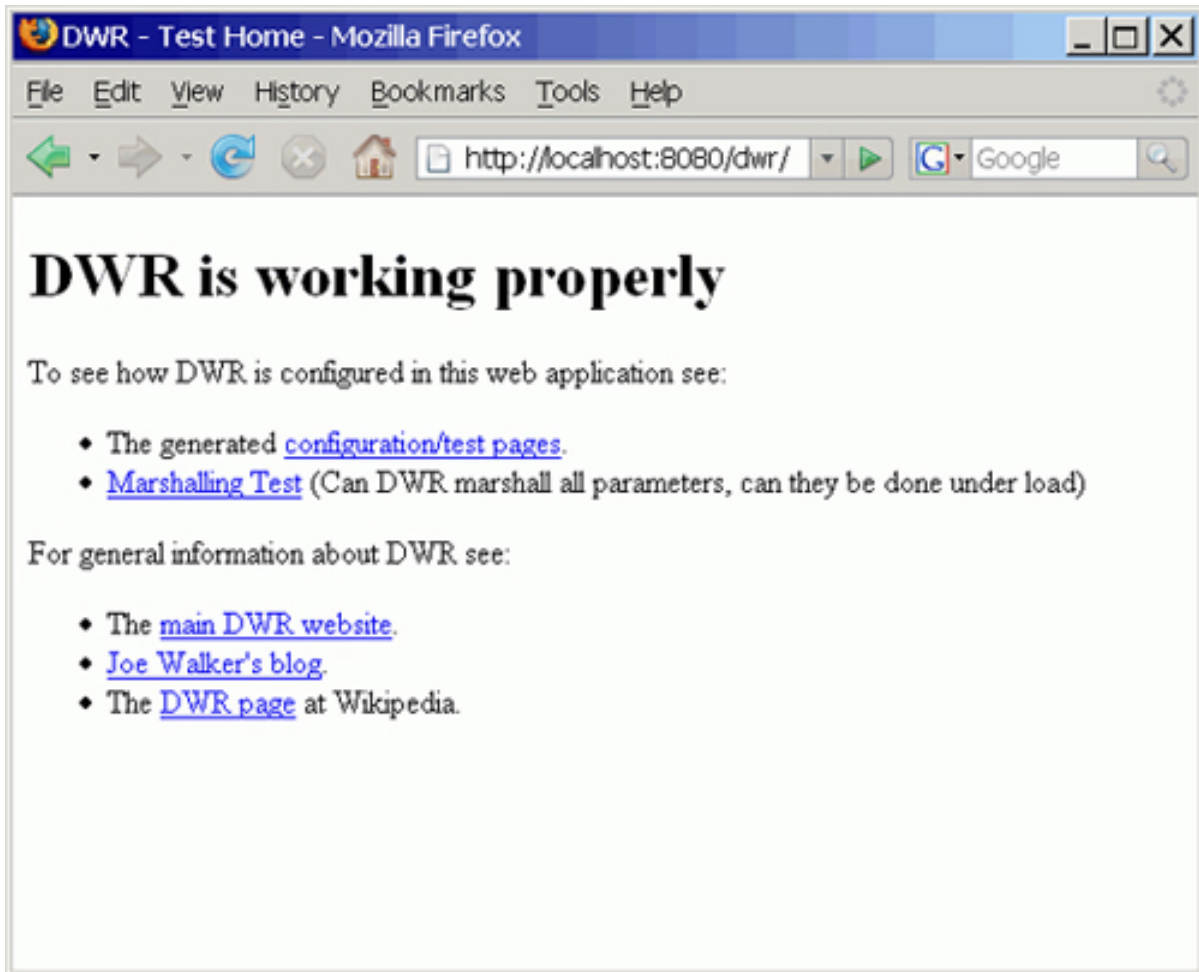
You'll see DWR and Ajax at work right away. The first page you displayed is the welcome page, shown in Figure 11:

Figure 11. Welcome page of the DWR application



If you blinked, you might have missed the welcome page shown in Figure 11 entirely (notice the red Loading indication on the top right corner). Using XMLHttpRequest, the embedded JavaScript code has asynchronously fetched the content of the second "DWR is working properly" page from the server and replaced the welcome page with it using DHTML. As a result, you see only the second page, shown in Figure 12:

Figure 12. Test page loaded asynchronously using DWR by application



You might want to browse through the unit tests this example application executes. This can help you to appreciate some of DWR's internal details. The next section shows how to add DWR to the employee-information application.

Section 7. Creating the new employee-information application

You're ready to upgrade the employee-information application to an Ajax version. This section takes you through the steps.

New employee-information application

The new employee-information application consists of the code components

described in Table 3:

Table 3. The components of the new employee-information application with Ajax

Component	Description
employee.html	An HTML page with the JavaScript code to implement the employee-information application.
Employee.java	A server-side Java class that accesses the DB2 Express-C 9.5 database using JDBC. This class is remoted to the client's browser using DWR.
ShowPhoto.java	The same ShowPhoto servlet as in the first (JSP-based) application. It is used to fetch employee photo information from DB2 Express-C 9.5 database and stream it to the user's browser.
dwstyles.css	The same stylesheet used in the first application.

Adding DWR to the application

To add DWR to the dwapp project, shut down Application Server and follow these steps in Eclipse:

1. In the web.xml file, add the boldfaced code shown in Listing 10:
Listing 10. Integrating the DWR support servlet into the application

```
<?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>dwapp</display-name>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>employee.jsp</welcome-file>
    <welcome-file>default.html</welcome-file>
    <welcome-file>default.htm</welcome-file>
    <welcome-file>default.jsp</welcome-file>
  </welcome-file-list>
  <resource-ref>
    <res-ref-name>jdbc/DataSource</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
```

```

    <res-auth>Container</res-auth>
    <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
<servlet>
  <description></description>
  <display-name>ShowPhoto</display-name>
  <servlet-name>ShowPhoto</servlet-name>
  <servlet-class>com.ibm.dw.ShowPhoto</servlet-class>
</servlet>
<servlet>
  <display-name>DWR Servlet</display-name>
  <servlet-name>dwr-invoker</servlet-name>
  <servlet-class>uk.ltd.getahead.dwr.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>

<servlet-mapping>
  <servlet-name>ShowPhoto</servlet-name>
  <url-pattern>/ShowPhoto</url-pattern>
</servlet-mapping>
  ...

```

- In the geronimo-web.xml file, add the dependency on the default DWR version 1.1.4 libraries already included with Application Server. Do this by adding the additional `<sys:dependency>` entry shown in Listing 11:

Listing 11. Exposing default DWR v. 1.1.4 libraries available to the application

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://geronimo.apache.org/xml/ns/j2ee/web-1.2"
  xmlns:nam="http://geronimo.apache.org/xml/ns/naming-1.2"
  xmlns:sec="http://geronimo.apache.org/xml/ns/security-1.1"
  xmlns:sys="http://geronimo.apache.org/xml/ns/deployment-1.2">
  <sys:environment>
    <sys:moduleId>
      <sys:groupId>default</sys:groupId>
      <sys:artifactId>dwapp</sys:artifactId>
      <sys:version>1.0</sys:version>
      <sys:type>car</sys:type>
    </sys:moduleId>

    <sys:dependencies>
      <sys:dependency>
        <sys:groupId>console.dbpool</sys:groupId>
        <sys:artifactId>dwDatasource</sys:artifactId>
      </sys:dependency>
      <sys:dependency>
        <sys:groupId>dwr</sys:groupId>
        <sys:artifactId>dwr</sys:artifactId>
        <sys:version>1.1.4</sys:version>
      </sys:dependency>
    </sys:dependencies>

```

```
</sys:environment>
    ...
```

3. Add a new DWR configuration file, `dwr.xml`, to the `WebContent\WEB-INF` directory. You'll find `web.xml` and `geronimo-web.xml` already there. The `dwr.xml` file should contain the code in Listing 12:

Listing 12. The `dwr.xml` file for DWR configuration

```
<!DOCTYPE dwr PUBLIC
    "-//GetAhead Limited//DTD Direct Web Remoting 1.0//EN"
    "http://www.getahead.ltd.uk/dwr/dwr10.dtd">

<dwr>
  <allow>
    <convert converter="bean" match="com.ibm.dw.Employee"/>

    <create creator="new" javascript="JEmployee">
      <param name="class" value="com.ibm.dw.Employee"/>
    </create>
  </allow>
</dwr>
```

The `dwr.xml` configuration file tells DWR which Java class to make remotely available. In this case, it says to expose the `com.ibm.dw.Employee` class as the `JEmployee` class in JavaScript code. The `<converter>` line specifies that DWR should allow transfer of objects of the type `com.ibm.dw.Employee` between the server and the client.

Creating the server-side Employee POJO

Add a new `Employee` class to the `com.ibm.dw` package, using the New wizard in Eclipse as you did with `ShowPhoto.java` (see the section [Showing photographs from DB2 BLOB fields](#)):

1. Highlight `WebContent`.
2. Press **Ctrl+N**.
3. Select **Class** in the New wizard.

Add the fields shown in boldface in Listing 13 to the `Employee` skeleton generated by Eclipse:

Listing 13. Adding fields to the Employee POJO

```
package com.ibm.dw;

import java.util.Date;

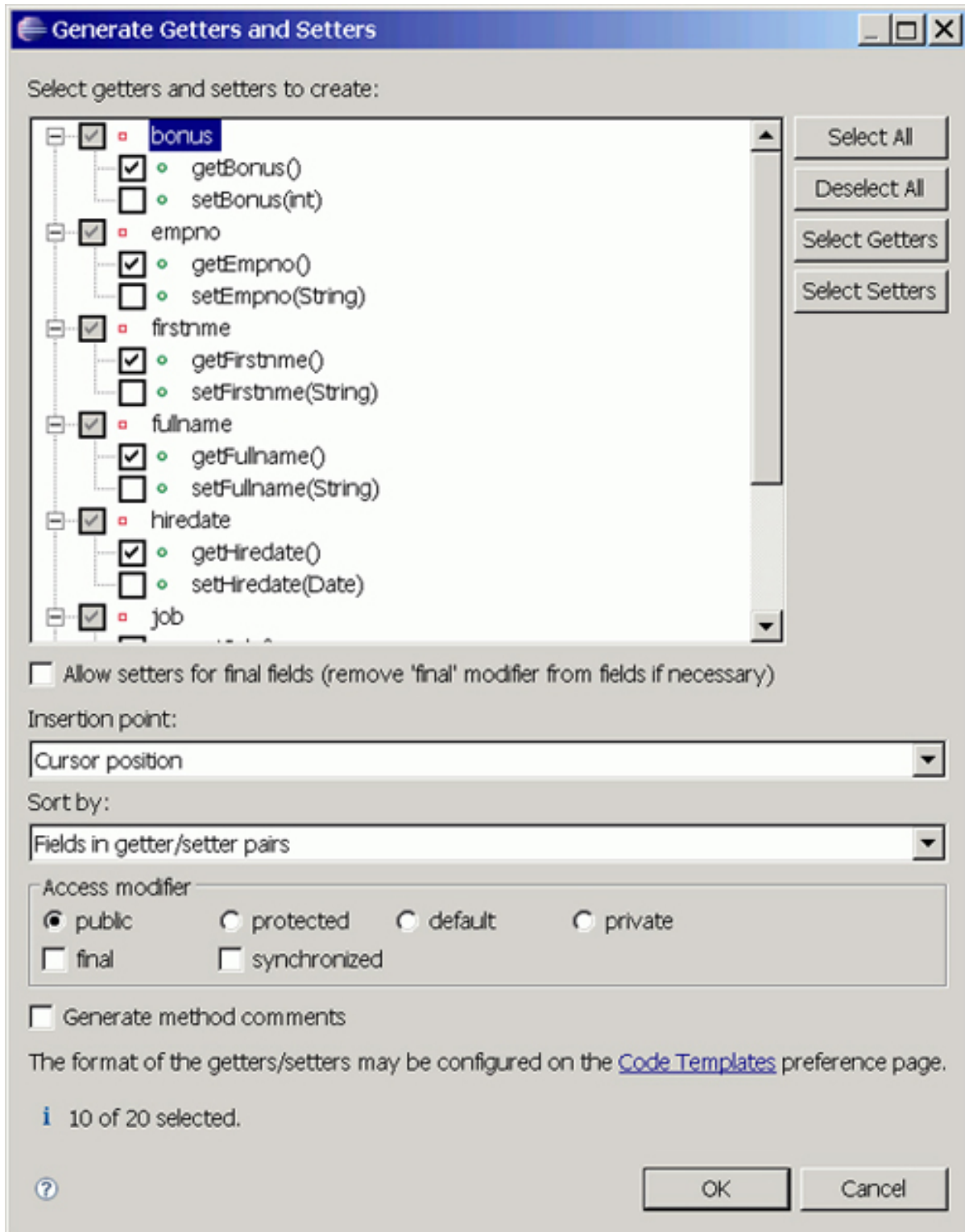
public class Employee {
    private String empno;
    private String firstnme;
    private String lastname;
    private String job;
    private Date hiredate;
    private String workdept;
    private String phoneno;
    private int salary;
    private int bonus;
    private String fullname;
    ...
}
```

Automatically generating getters using Eclipse

Coding getter methods for a Java object's fields is a frequent — and extremely tedious — development activity. Eclipse makes this task easy by providing an automatic code-generation facility.

In the Employee.java editor, right-click and select **Source > Generate Getters and Setters**. This starts the **Generate Getters and Setters** wizard, shown in Figure 13:

Figure 13. The Eclipse Generate Getter and Setters wizard



Click **Select Getters** and then click **OK**. This generates all the getter methods that you need for the `Employee` class automatically.

Fetching employee records from the database using JDBC

Next, add the private constructor shown in boldface in Listing 14 to the `Employee` class. Make sure you have the default public no-argument constructor (also shown in Listing 14) already defined:

Listing 14. The two constructors for `Employee` instances

```
public Employee() {  
}  
  
private Employee(String empno, String firstnme,  
                String lastname,  
                String job, Date hiredate, String workdept,  
                String phoneno, int salary, int bonus) {  
    this.empno = empno;  
    this.firstnme = firstnme;  
    this.lastname = lastname;  
    this.job = job;  
    this.hiredate = hiredate;  
    this.workdept = workdept;  
    this.phoneno = phoneno;  
    this.salary = salary;  
    this.bonus = bonus;  
  
    this.fullname = lastname + ", " + firstnme;  
}
```

The `fullname` field is used to display the names in the application's drop-down list.

Last but not least, add the `findStartsWith()` method, shown in Listing 15, to fetch employee records from the `jdbc/DataSource` database pool. This method returns an array of employees with names matching the specified `startText` string.

Listing 15. The `findStartsWith()` method

```
public static Employee [] findStartsWith(String startText) {  
    String query =  
        "select * from employee where lastname like '"  
        + startText + "%'";  
  
    DataSource dsource = null;  
    Statement stmt= null;  
    Connection conn =null;  
    ResultSet rset = null;  
  
    ArrayList result = new ArrayList();  
    try {  
        InitialContext context = new InitialContext();  
        dsource =  
            (DataSource)context.lookup("java:comp/env/jdbc/DataSource");  
        conn = dsource.getConnection();  
        stmt = conn.createStatement();  
        rset = stmt.executeQuery(query);  
    }  
}
```

```
while (rset.next()) {
    result.add(
        new Employee(
            rset.getString("empno"),
            rset.getString("firstnme"),
            rset.getString("lastname"),
            rset.getString("job"),
            rset.getDate("hiredate"),
            rset.getString("workdept"),
            rset.getString("phoneno"),
            rset.getInt("salary"),
            rset.getInt("bonus")
        )
    );
}
} catch(NamingException e) {
    // add handler or logging code here
} catch(SQLException e) {
    // add handler or logging code here
}
} finally {
    try {
        if (rset != null)
            rset.close();
        if (stmt != null)
            stmt.close();
        if (conn != null)
            conn.close();
    } catch (SQLException ex ) {}
}

int arraylen = result.size();
Employee [] retArray = new Employee[arraylen];
for (int i=0; i < arraylen; i++)
    retArray[i] = (Employee) result.get(i);

return retArray;
}
```

Before you save the `Employee.java` file, press **Ctrl+Shift+O** to add all the import statements required using the Eclipse organize-imports feature.

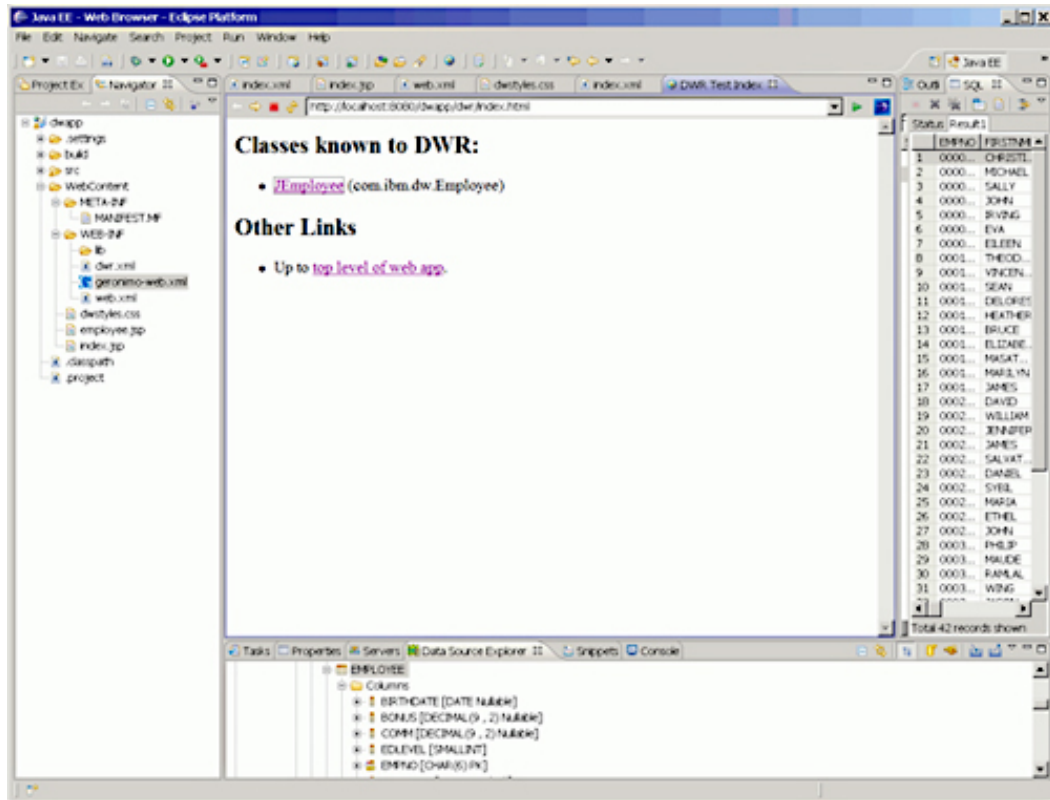
Note that although the `Employee.java` file is hand-coded in this example, it is not unreasonable in some projects to generate this server-side data-access code automatically using Java persistence technology such as JPA or an object-relational mapper such as Hibernate, Castor, or JDO (see [Resources](#)).

Verify remote exposure of the `Employee` class

To verify that the `Employee` class is remototed properly using DWR, perform the following steps:

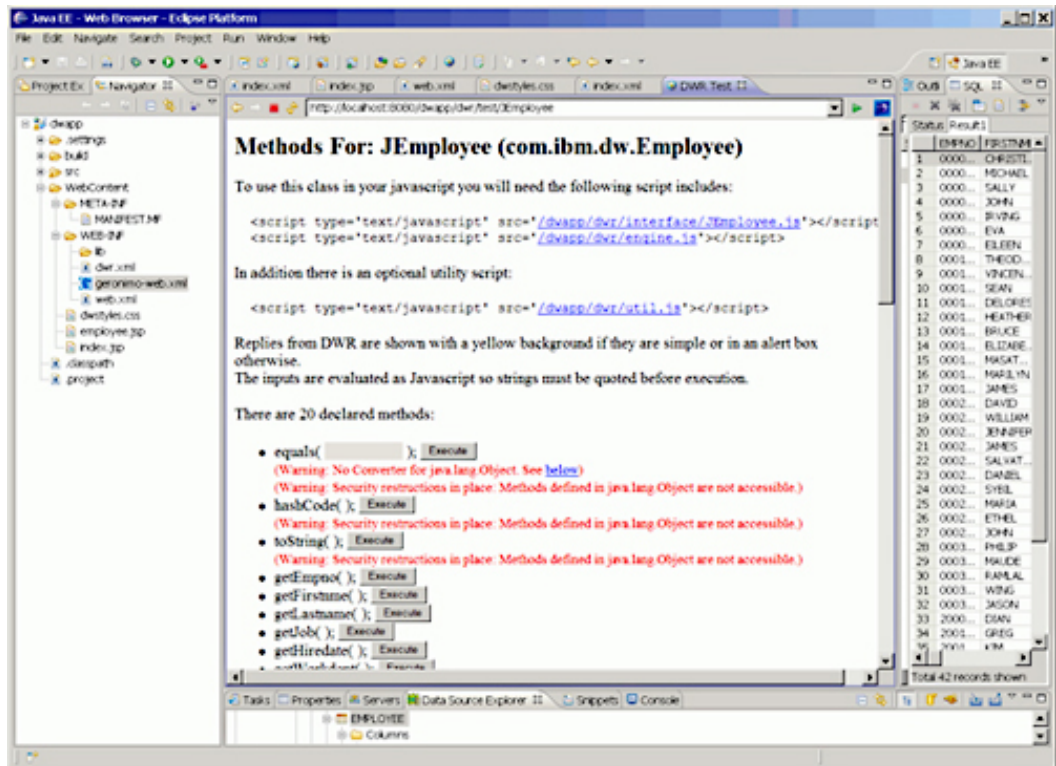
1. Build and deploy the project to the server by right-clicking dwapp and selecting **Run As > Run on Server**.
2. Point a browser session to `http://localhost:8080/dwapp/dwr/index.html`. You should see DWR reporting that `com.ibm.dw.Employee` has been exposed as `JEmployee`, as shown in Figure 14:

Figure 14. DWR confirming remoting of the Employee class



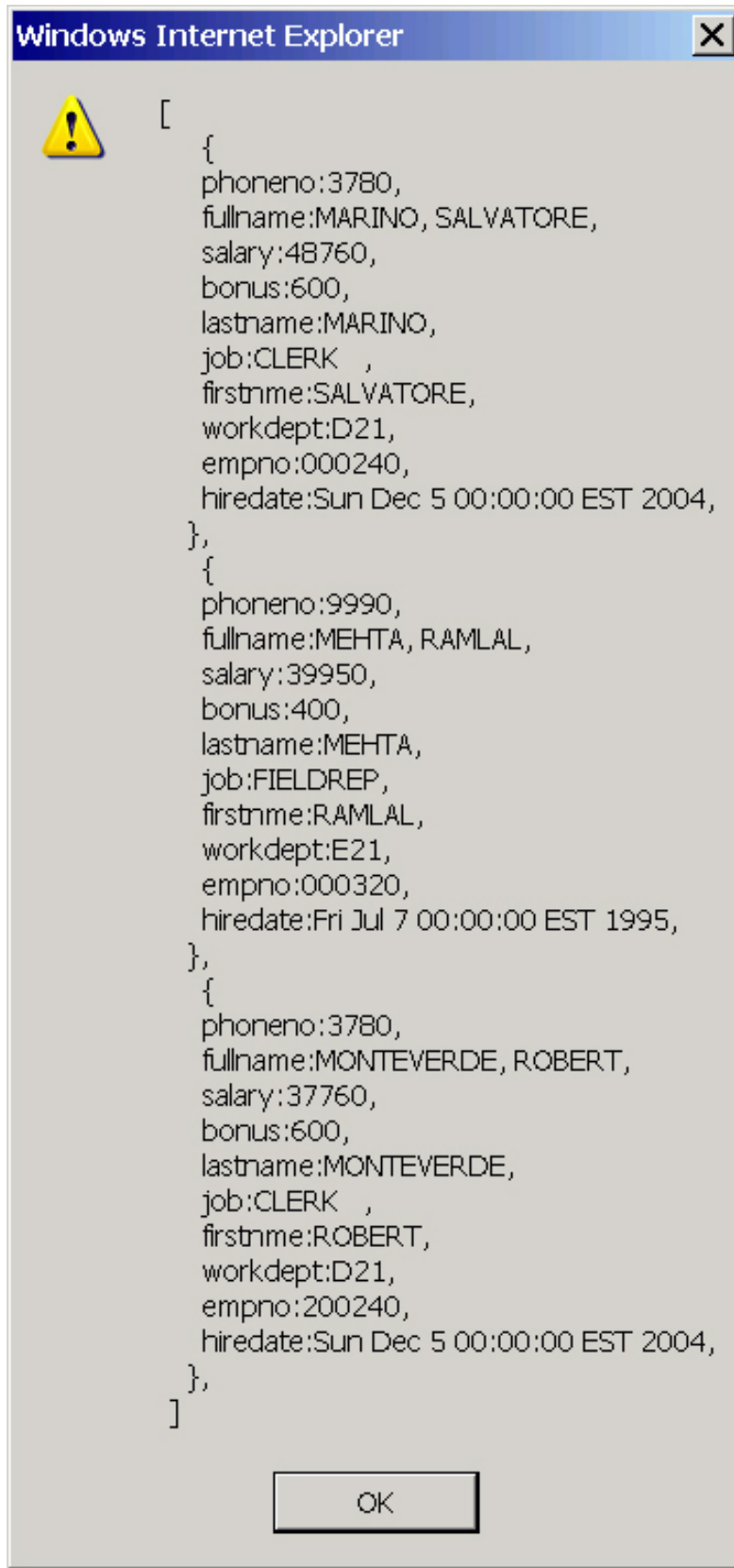
3. Click the `JEmployee` link. DWR now generates a page that lists all of the methods available with `JEmployee` (methods of `Employee` exposed remotely). Look for the `findStartsWith()` method, shown in Figure 15:

Figure 15. DWR-generated page showing all methods remotely exposed



- Enter the letter M inside the quotation marks and click **Execute**. This invokes the method on the JavaScript `JEmployee` object, which in turn calls the server-side `Employee` object queries the DB2 Express-C 9.5 database for the requested employee records and then DWR transfers the fetched data back to the browser asynchronously in JSON format. You'll see a JavaScript alert box with the fetched data, similar to Figure 16. In this case, three `JEmployee` instances are returned.

Figure 16. Employee records in the form of JEmployee JavaScript object instances, containing data fetched from the database



JavaScript coding in employee.html

The only code remaining for you to write is the employee.html Web page. The JavaScript code in this page handles all of the interaction with the user.

Use the Eclipse New wizard to generate the skeleton. Select **Web > HTML** and name it employee.html.

In the <head> section of the generated page, add the title and stylesheet reference, shown in boldface code in Listing 16:

Listing 16. Adding the title and stylesheet reference

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<link rel="stylesheet" type="text/css" href="dwstyles.css"/>
<title>dW Example Employee Data - AJAX version</title>
</head>
```

In the beginning of the <body> section, add the boldfaced JavaScript code from Listing 17:

Listing 17. Adding references to include the DWR JavaScript library code

```
<body>
<script type='text/javascript' src='/dwapp/dwr/interface/JEmployee.js'></script>
<script type='text/javascript' src='/dwapp/dwr/engine.js'></script>
<script type='text/javascript' src='/dwapp/dwr/util.js'></script>
<script type='text/javascript'>
var employees;
function update() {
    JEmployee.findStartsWith(DWRUtil.getValue
        ("searchblank").toUpperCase(),updatechoice);
}

function updatechoice(emps) {
    employees = emps;
    DWRUtil.removeAllOptions("empselect");
    DWRUtil.addOptions("empselect", emps, "empno", "fullname");
}

function setinfo() {
    selected = DWRUtil.getText("empselect");
    emplength = employees.length;
    for(count = 0; count < emplength; count++) {
        with (employees[count]) {
            if (fullname == selected) {
                DWRUtil.setValue("fullname", fullname);
                DWRUtil.setValue("empno", empno);
                DWRUtil.setValue("job", job);
                DWRUtil.setValue("workdept", workdept);
                DWRUtil.setValue("phoneno", phoneno);
                DWRUtil.setValue("salary", salary);
                DWRUtil.setValue("bonus", bonus);
            }
        }
    }
}
```



```
        <tr>
          <td class="label">Bonus:</td><td>${<span id="bonus" /></td>
        </tr>

      </table>
    </td>
    <td>
      <img name="photo" src="" />
    </td>
  </tr>
</table>
</body>
</html>
```

After the user enters a partial employee last name into the data-entry box and tabs out, the `onblur` event is triggered. The JavaScript `update()` function is called in this case:

```
<input name="searchblank" size="10" onblur="update()" />
```

The `update()` function calls the `JEmployee.findStartsWith()` method to fetch the `JEmployee` records remotely. This call is asynchronous. It returns immediately, and the code specifies that the function called `updatechoice()` should be called when the result arrives back from the server.

In the `updatechoice()` function, the returned array of `JEmployee` objects is assigned to a page-scoped variable called `employees`. Then the `<select>` list is emptied and then populated with the `fullname` field from the `JEmployee` objects. This is done using DHTML code. The `DWRUtil.removeAllOptions()` and `DWUtil.addOptions()` functions from the DWR JavaScript library make this easy and save you from writing browser-specific code.

This is how the `<select>` element is populated dynamically with employee name selection. When the user selects one of the employee names from the `<select>` element and then exits the field, the `<select>` element's `onblur` handler is called. The `<select>` element specifies this to be the `setinfo()` function:

```
<select name="empselect" onblur="setinfo()">
  <option value=""></option>
</select>
```

The `setinfo()` method determines the ID of the employee selected by the user and uses the ID to look through the page-scoped `employees` array for the selected `JEmployee` instance. This instance is then used to set the value of the information elements, using DHTML. Again, the `DWRUtil.setValue()` method from the DWR JavaScript library makes this simple and independent of browser idiosyncrasies.

Set up the Web application's welcome page

The default welcome page was set earlier to `employee.jsp`. You need to change it to `employee.html`. Make the change to `web.xml` highlighted in boldface in Listing 19:

Listing 19. Changing the default welcome page

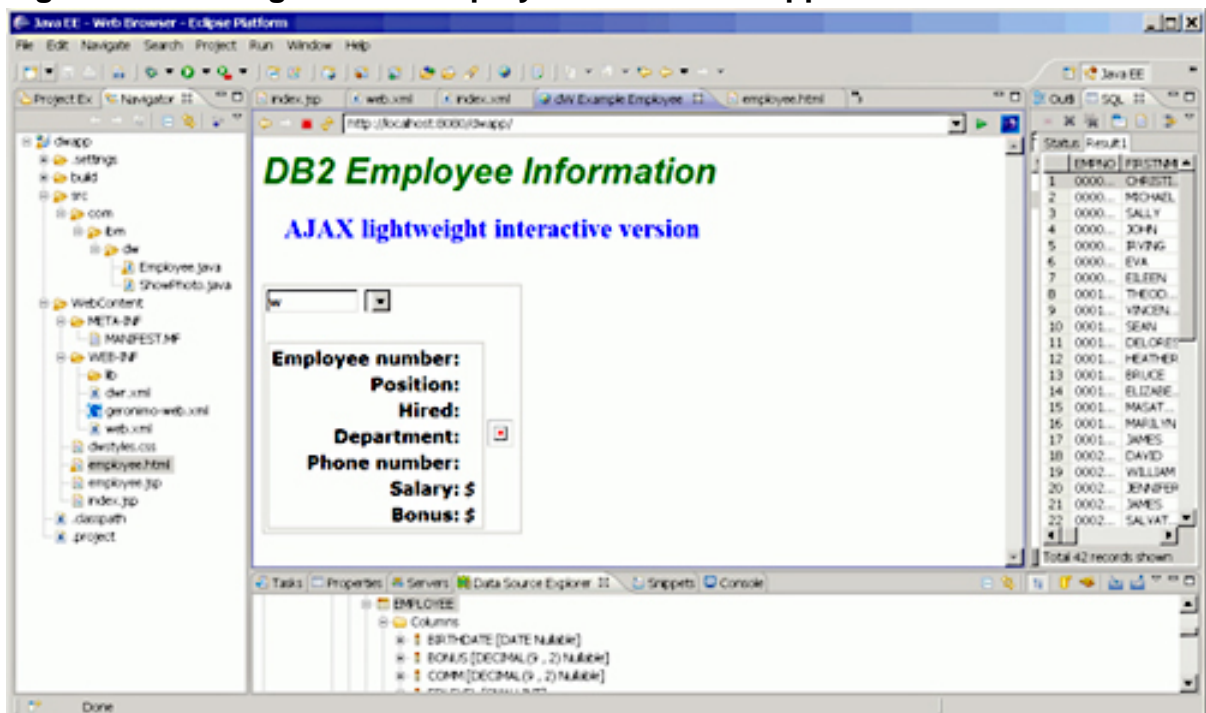
```
<welcome-file-list>
  <welcome-file>employee.html</welcome-file>
  <welcome-file>index.htm</welcome-file>
  <welcome-file>employee.jsp</welcome-file>
  <welcome-file>default.html</welcome-file>
  <welcome-file>default.htm</welcome-file>
  <welcome-file>default.jsp</welcome-file>
</welcome-file-list>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

Trying out the Ajax-based employee-information application

Right-click `dwapp` in the Navigator view and select **Run As > Run on Server**. This deploys the application to Application Server and starts it.

You should see the new application, with no selected employee data, as shown in Figure 17:

Figure 17. Starting the new employee-information application

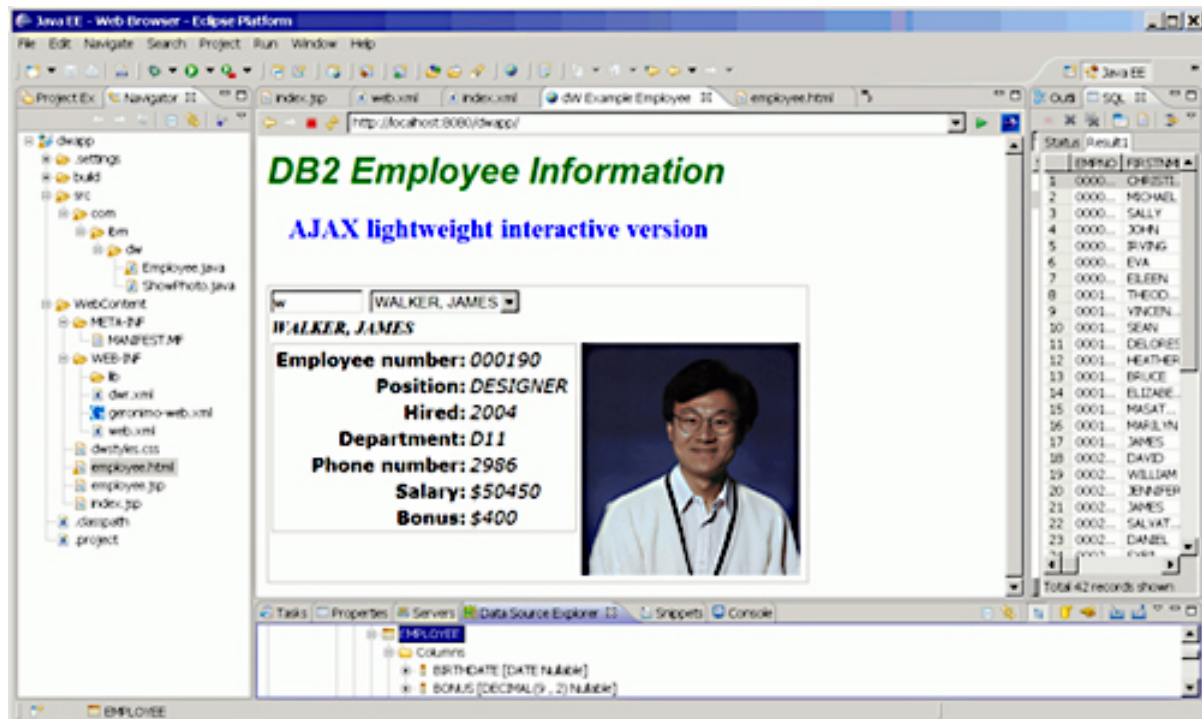


Enter `a` into the edit box and press **Tab**. Momentarily, you should see the drop-down list populated with *ADAMSON*, *BRUCE*. Tab out of the select list, and Bruce

Adamson's information is fetched dynamically.

Try entering `w` into the edit box and then `tab`. This time, *WALKER, JAMES* is shown in the drop-down list. `Tab` out of the list, and James Walker's information is fetched dynamically (see Figure 18). You may also want to try `l`, which should populate the list with more than one choice.

Figure 18. Ajax application with asynchronously fetched employee information



Notice how much more interactive and "natural" this Ajax interface feels when contrasted with the page-at-a-time user interface presented by the first application. This style of dynamically updated, highly responsive user interface is enabled by high-speed network connections and powerful back-end servers. This trend — often called Web 2.0 — is the wave of the future in Web applications.

Section 8. Summary

In this tutorial, you:

- Created an employee-information application using conventional JSP and servlet technology

- Coded the project in Eclipse and deployed it on Application Server for testing
- Created a servlet that serves employee photo bitmaps fetched from DB2 Express-C 9.5 BLOB data fields
- Downloaded and tested DWR for Ajax development within Eclipse
- Migrated the application to a lightweight Ajax design using DWR's JavaScript library
- Coded the `Employee` POJO for remoting through DWR
- Coded and tested the new Ajax application in Eclipse and deployed and tested it using Application Server

Eclipse, Application Server, and DB2 Express-C 9.5 don't restrict you in any way to Java EE-only development. In fact, the Kick-start your Java apps suite is an ideal platform for exploring lightweight and alternative interactive user-interface technology. You can test and debug any creations you develop during your explorations in the familiar environment of the Eclipse IDE, and you can deploy them for production on the robust and proven combination of DB2 Express-C 9.5 and Application Server.

Downloads

Description	Name	Size	Download method
Sample code	j-kickstart2code.zip	15KB	HTTP

[Information about download methods](#)

Resources

Learn

- ["Kick-start your Java apps: Free software, fast development"](#) (Sing Li, developerWorks, December 2007): Use IBM-backed open source and free software to kick-start your Java Web-based application development.
- ["DB2 Express-C, the developer-friendly alternative"](#) (Grant Hutchison, developerWorks, February 2006): Get started quickly using DB2 Express-C 9.5 for all of your applications.
- [Migrate to DB2 Express-C](#): Resources to help you get started migrating to DB2 Express-C today.
- [Kick-start your Java apps: Free software for rapid results](#): Get free software for rapid results.
- [Get started now with Eclipse](#): The developerWorks Eclipse resource page.
- ["Manage your Eclipse environment"](#) (Chris Aniszczyk and Phil Crosby, developerWorks, February 2006): Zen and the art of Eclipse maintenance.
- [Migrating to Eclipse: A developer's guide to evaluating Eclipse](#): Find out how Eclipse differs from Netbeans, IntelliJ IDEA, and Borland JBuilder.
- [WebSphere Application Server Community Edition](#): Documentation, FAQs, articles, and more resources.
- ["Migrate from JBoss to WebSphere Application Server Community Edition"](#) (Shyam Nagarajan, developerWorks, November 2005): An excellent guide for migrating your applications.
- [Direct Web Remoting](#): Visit the official DWR Web site, where you'll find a community of DWR users interacting with the creators.
- ["Introduction to Spring 2 and JPA"](#) (Sing Li, developerWorks, August 2006): Apply your Kick Start Java tools knowhow- use DB2 Express-C and Eclipse in learning the popular lightweight Spring 2 server framework. Combine this with the new Java Persistence API (JPA), supported by Application Server, to create POJO based web application system.
- [Ajax for Java developers](#) (Philip McCarthy, developerWorks, September - November 2005): This three-part series is an excellent introduction to Ajax and detailed coverage of the JSON remoting data format.
- [Ajax resource center](#): Visit this developerWorks site for an array of resources on creating interactive Web applications with Ajax technologies.
- [Introducing JSON](#): Read about JavaScript Object Notation and the available implementations in a variety of programming languages.

- [Ruby on Rails](#): Read about this popular lightweight rapid application creation framework.
- [Castor](#): You can use this object-to-relational mapping framework to generate your server-side POJOs.
- [Java Data Objects \(JDO\)](#): Find out how you can avoid direct JDBC coding with this popular persistence framework for Java objects.

Get products and technologies

- [WebSphere Application Server Community Edition V2.0.0.1](#): Download the application server.
- [DB2 Express-C 9.5](#): Download the database server.
- [Eclipse](#): Download the Eclipse SDK.
- [DWR](#): Download DWR.
- [PyDev plug-in](#): This plug-in enables rapid development of Web applications using the Python programming language.

Discuss

- [eclipse.org](#)
- [Eclipse newsgroups](#)
- [Eclipse developer mailing lists](#)
- [DB2 Express forum](#)
- [developerWorks blogs](#)

About the author

Sing Li



Sing Li is the author of *Professional Apache Tomcat*, *Early Adopter JXTA*, and *Professional Jini*, as well as numerous other books with Wrox Press. He is a regular contributor to technical magazines and is an active evangelist of the P2P evolution. Sing is a consultant and freelance writer and can be reached at westmakaha@yahoo.com.