Building Web-based applications with JDBC

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. About this tutorial

Should I take this tutorial?

If you're interested in the various approaches to building Web applications that access a database through JDBC, then this tutorial is for you. In this hands-on guide, you'll learn the fundamentals of this process through three separate techniques: a servlet approach, a JavaServer Pages (JSP) page approach, and combined JSP, JavaBeans, and servlet approach (also known as Model Two).

The example code in this tutorial has been developed to work with the DB2 Universal Database 7.2 (see JSP page and servlet resources on page 23 for more details on DB2), but modifying the code to work with other databases is trivial due to the use of a **`DataSource`** object.

This tutorial assumes you work comfortably with the Java programming language and does not delve into the particulars of servlets or JSP technology (see JSP page and servlet resources on page 23 for further reading on these subjects). The techniques presented, however, build on the skills developed in *Managing database connections with JDBC* and *Advanced database operations using JDBC*, and we recommend that you complete these tutorials prior to starting this one.

Sample application design

The example code snippets used throughout this tutorial are based on a fictitious message board application. It allows multiple authors to post messages and incorporates a searchable message digest that summarizes the status of the message board. If you plan on working through the examples in this tutorial, we recommend that you complete the *Advanced database operations with JDBC* tutorial first; doing so ensures your environment is correctly installed and configured.

The simplified version of the message board application discussed in this tutorial has three distinct entities:

* A message class
* An author class
* A digest class

Each of these entities has its own table in the database. The start page for the Web application displays the complete message digest. From the start page, a user is able to click on a message ID to display the associated message. Likewise, from a given message page, the user can click on a user ID to display the user profile (a picture in this case).

Tools for completing the tutorial

Although reading the examples helps you understand the concepts in this tutorial, most people learn better by actually trying out the examples for themselves. The following tools must be installed and working correctly before working through the examples:

* A text editor: Java source files are simply text, so to create and read them, only a text

editor is required. A Java IDE can also be used, but sometimes the IDE hides too many of the details.

* A Java development environment, such as the Java 2 SDK, which is available at *http://java.sun.com/j2se/1.4/*: The Java 2 SDK, Standard Edition version 1.4, includes the JDBC standard extensions as well as JNDI (Java Naming and Directory Interface), both of which are necessary for some of the examples in this tutorial.

* An SQL-compliant database: The examples in this tutorial have been tested with DB2 running on a Windows 2000 server, but because they are written using a **`DataSource`** object, they should easily convert to other databases, including Oracle and Microsoft SQL Server. JDBC and database resources on page 24 contains links to more information on both JDBC and databases.

* A JDBC driver: Because the JDBC API is predominantly composed of interfaces, an actual JDBC-driver implementation must be obtained to make the examples in this tutorial work. The examples use advanced JDBC functionality, and, therefore, they require an advanced JDBC driver. Most database and JDBC driver vendors supply an evaluation version of a particular JDBC driver, including IBM, which provides a good JDBC driver for DB2.

* A JSP page container: This type of container is required to build a JSP-based Web application. The reference implementation for both the servlet and JSP technology specification is the Apache Software Foundation's *Jakarta Tomcat* server, which is freely available, and is used throughout this tutorial.

* The message board tables from the tutorial "Advanced database operations with JDBC": These tables were created and populated for use in a fictitious message board application. This tutorial builds on that application and requires those tables and data to properly function. The source code from this tutorial can be downloaded from JSP page and servlet resources on page 23 .

About the author

Robert J. Brunner is an astrophysicist by day and a computer hacker by night. His principal employer is the California Institute of Technology, where he works on knowledge extraction

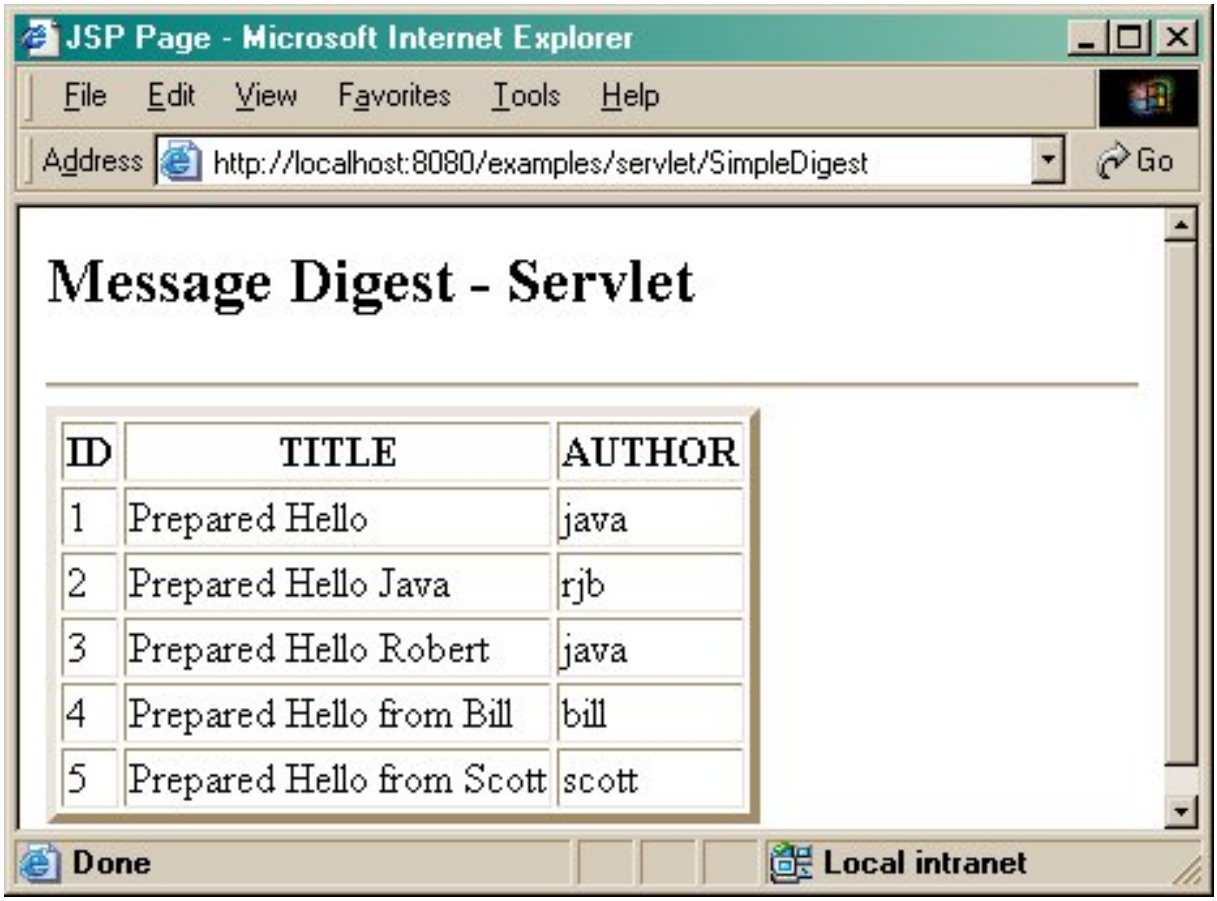
and data mining from large, highly distributed astronomical databases. He has provided consulting to scientific data centers around the world, provided Java and XML training to a variety of groups, and he is currently working on the book *Enterprise Java Database Programming*, which will be published by Addison-Wesley in 2002. Contact Robert at *rjbrunner@pacbell.net*.

Section 2. A servlet solution

A servlet-only approach

In a servlet-only approach, everything from processing the user request to querying the database to formulating the response must be handled.

To demonstrate this functionality, we'll be working with a single servlet. It's used to generate the digest view for our message board application, as shown in the figure below. All of the code in this tutorial is available in JSP page and servlet resources on page 23 . Simply unzip the code into your Apache Tomcat installation directory, and you'll be off and running.



A simple servlet

First, the servlet must be initialized. This includes importing all necessary classes, initializing all global parameters, and obtaining the **`DataSource`** used to connect to the messageboard database. In a real-world application, it's best to use a **`PooledDataSource`** to improve performance and eliminate any potential conflicts over using a single **`DataSource`** for all client requests. For simplicity, this tutorial does not use **`PooledDataSource`** objects.

Here's the code for initializing the servlet:

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.Hashtable ;
```

```
import javax.naming.* ;
import java.sql.* ;
import javax.sql.DataSource ;
public class SimpleDigest extends HttpServlet {
  private DataSource ds ;
  private String fsName = "jdbc/pjtutorial/db2" ;
  private String querySQL = "SELECT id, title, author FROM digest" ;
  public void init(ServletConfig config) throws ServletException {
    super.init(config);
    try{
      Hashtable env = new Hashtable() ;
      env.put(Context.INITIAL_CONTEXT_FACTORY,
      "com.sun.jndi.fscontext.RefFSContextFactory") ;
      Context ctx = new InitialContext(env) ;
      ds = (DataSource)ctx.lookup(fsName) ;
    }catch(NamingException e ) {
      System.err.println("Error: Unable to find DataSource") ;
    }
  }
```

Simple servlet: Method definitions

To complete the servlet, we next include the mandatory method definitions. Because of the simple approach, ignore the differences between **GET** and **POST** HTTP requests and process all requests in the same manner.

```
  public void destroy() {
  }
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, java.io.IOException {
    processRequest(request, response);
  }
  protected void doPost(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, java.io.IOException {
    processRequest(request, response);
  }
  public String getServletInfo() {
    return "Simple Digest Display Servlet";
  }
}
```

Simple servlet: Extract the data

Now we get to the meat of the code. The **processRequest** method gets the data out of the database and returns it to the client in the form of an HTML table. The first step is to perform the database query and initialize the HTML output.

```
protected void
  processRequest(HttpServletRequest request, HttpServletResponse response)
throws ServletException, java.io.IOException {
  Connection con = null ;
  try{
    con = ds.getConnection("java", "sun") ;
    java.sql.Statement stmt = con.createStatement() ;
    ResultSet rs = stmt.executeQuery(querySQL) ;
    ResultSetMetaData rsmd = rs.getMetaData() ;
    response.setContentType("text/html");
```

```
java.io.PrintWriter out = response.getWriter();
out.println("<html>");
out.println("<head>");
out.println("<title>JSP Page</title>");
out.println("</head>");
out.println("<body>");
out.println("<h2> Message Digest - Servlet </h2>");
out.println("<hr/>");
```

Simple servlet: Generating HTML

Now that the appropriate data has been extracted, it is time to generate the HTML that presents this data to the user. In this case, one big table is made. In reality, multiple pages are desirable to limit the result to a manageable number of messages per page.

```
  out.println("<table border=\"4\">");
  out.println("<tr>");
  for(int i = 1 ; i <= rsmd.getColumnCount() ; i++)
    out.println(\"<th>\" + rsmd.getColumnName(i) + "</th>");
  out.println("</tr>");
  while(rs.next()) {
    out.println("<tr>");
    out.println("<td>" + rs.getInt(1) + "</td>");
    out.println("<td>" + rs.getString(2) + "</td>");
    out.println("<td>" + rs.getString(3) + "</td>");
    out.println("</tr>");
  }
  out.println("</table>");
  out.println("</body>");
  out.println("</html>");
  out.close();
```

Simple servlet: Clean up

Finally, everything must be cleaned up. Because we are taking the simple approach of having a single **`Connection`** per request, everything must be explicitly freed. A better solution is to use **`PooledConnection`** objects, in which case only the **`PooledConnection`** object would be released.

```
    rs.close();
    stmt.close();
  }catch(SQLException ex){
    System.out.println("\nERROR:----- SQLException -----\n");
    while (ex != null) {
      System.out.println("Message:   " + ex.getMessage());
      System.out.println("SQLState:  " + ex.getSQLState());
      System.out.println("ErrorCode: " + ex.getErrorCode());
      ex = ex.getNextException();
    }
  }catch(Exception e ) {
    e.printStackTrace();
  }finally {
    try {
      if(con != null)
        con.close() ;
    }catch (SQLException ex)  {
      System.out.println("\nERROR:----- SQLException -----\n");
```

```
          System.out.println("Message:   " + ex.getMessage());
          System.out.println("SQLState:  " + ex.getSQLState());
          System.out.println("ErrorCode: " + ex.getErrorCode());
        }
      }
    }
    public String getServletInfo() {
      return "Simple Digest Display Servlet";
    }
  }
```

OK, it works, but ...

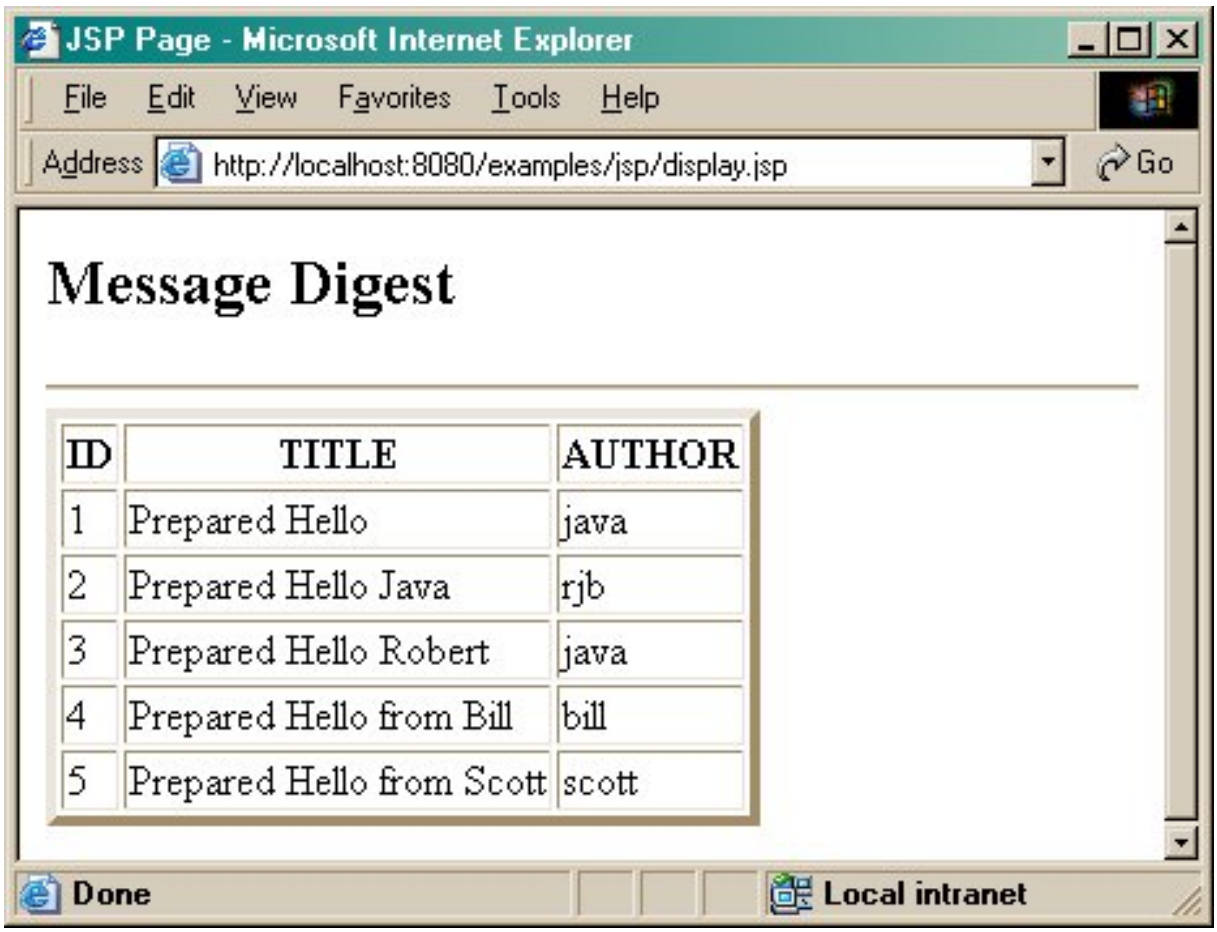
This simple servlet does its job, but it probably leaves you feeling unsatisfied (or maybe worse). Primarily, this is because of all of the ugly **`out.println()`** statements that generate the HTML code. If you think about it, this was a very simple Web page; what if you wanted something more complex, such as frames with special headers, footers, or advertisements? Or, what happens when your boss decides you need to change the appearance of the entire Web site? Suddenly you need to start modifying perfectly good code that is bug-free -- never a good idea. There has to be a better way!

Section 3. A JSP page solution

JSP page design

The servlet code was ugly and hard to maintain because of HTML in the Java code. The opposite approach, and one that is a bit more robust, is to add Java code to HTML. The JSP technology specification provides the ability to add Java code to an HTML file, which is then processed by the server to dynamically generate the response (note that this is different from JavaScript code, which executes on the client).

The first JSP page we will examine duplicates the work of the simple servlet, as shown in the figure below. Notice the different URL in the Web address bar.



display.jsp

Because this JSP page functions much the same as the simple servlet, it should look similar, albeit smaller: the bulky **`out.println()`** methods aren't needed and we don't have to explicitly worry about error handling.

First, everything must be initialized, which in a JSP page includes the **`page`** directive that handles the communication between the JSP page and the JSP page container. With this directive, you specify the content type of the HTTP response, which Java classes must be imported, and which JSP page will handle any programming errors (or exceptions). After this,

use a declaration element to set up the global parameters. The use (or abuse) of declaration elements is addressed later in this section.

```
<%@
  page
  language="java"
  errorPage="error.jsp"
  contentType="text/html"
  import="java.util.Hashtable,
    javax.naming.*,
    java.sql.*,
    javax.sql.DataSource"
%>
<%!
  String fsName = "jdbc/pjtutorial/db2" ;
  Connection con = null ;
  String querySQL = "SELECT id, title, author FROM digest" ;
%>
```

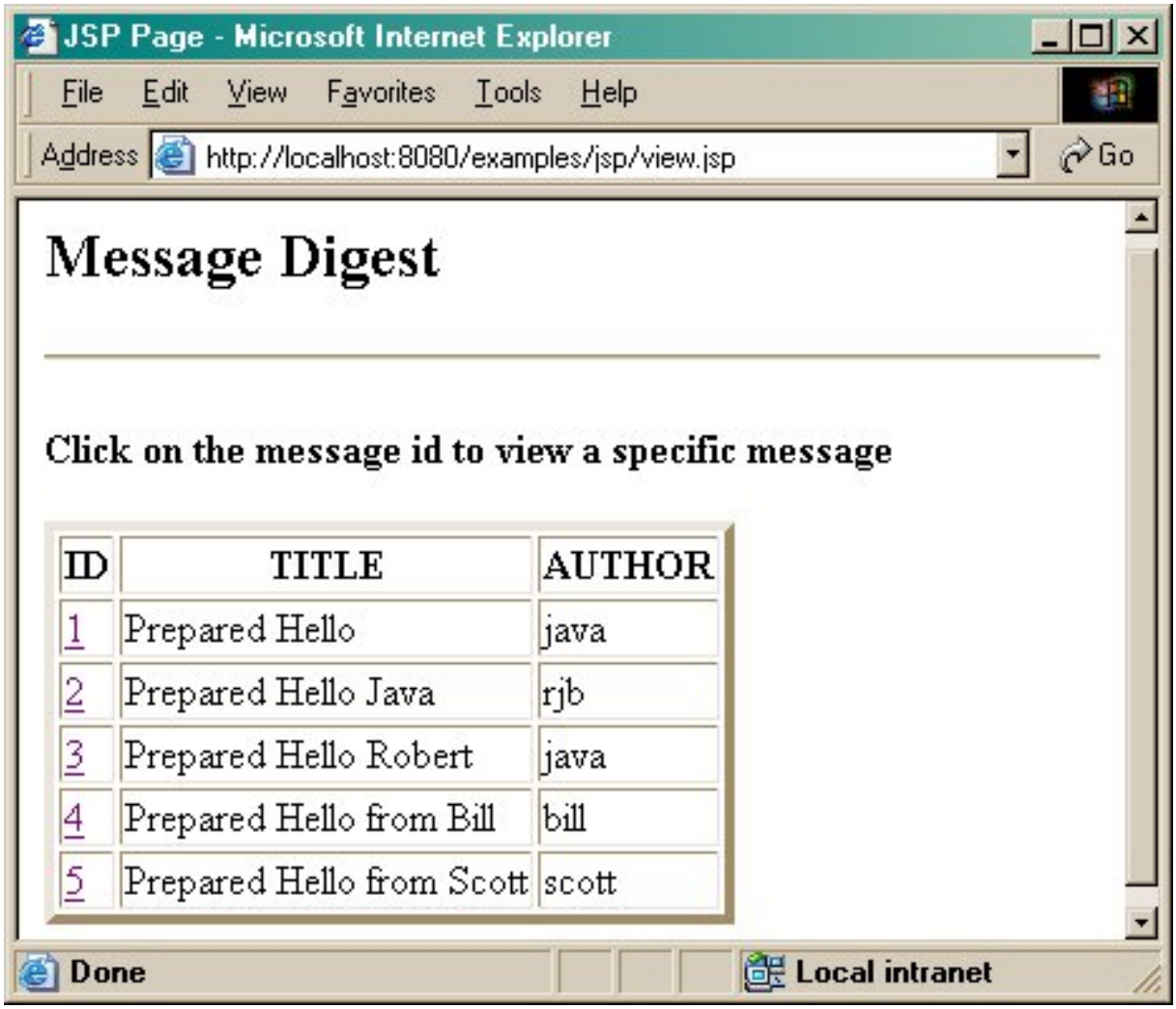
display.jsp: data and HTML generation

The rest of this JSP page extracts the relevant data from the database and generates the appropriate HTML.

```
<html>
<head><title>JSP Page</title></head>
<body>
<h2> Message Digest </h2>
<hr/>
<table border="4">
<tr>
<%
  Hashtable env = new Hashtable() ;
  env.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.sun.jndi.fscontext.RefFSContextFactory") ;
  Context ctx = new InitialContext(env) ;
  DataSource ds = (DataSource)ctx.lookup(fsName) ;
  con = ds.getConnection("java", "sun") ;
  java.sql.Statement stmt = con.createStatement() ;
  ResultSet rs = stmt.executeQuery(querySQL) ;
  ResultSetMetaData rsmd = rs.getMetaData() ;
  for(int i = 1 ; i <= rsmd.getColumnCount() ; i++) {
%>
    <th><%=rsmd.getColumnName(i)%></th>
<% } %>
</tr>
<%
  while(rs.next()) { %>
    <tr>
      <td> <%= rs.getInt(1)%> </td>
      <td> <%= rs.getString(2) %> </td>
      <td> <%= rs.getString(3) %> </td>
    </tr>
 <% }
  con.close();
%>
</table>
</body>
</html>
```

view.jsp

As you know, one of the objectives of this application/project is to allow the user to select a message from the digest to see its contents in more detail. With a few modifications to display.jsp, we end up with a new JSP page that displays the digest, but with clickable IDs as shown in the figure below.



Here is the view.jsp code. (Code blocks that haven't changed from display.jsp have been replaced with ellipses for clarity.)

```
...
  int id = 0 ;
%>
...
<h4> Click on the message ID to view a specific message </h4>
<table border="4">
...
<%
  while(rs.next()) {
    id = rs.getInt(1) ;
%>
    <tr>
      <td> <a href="message.jsp?id=<%= id %>"> <%= id %> </a> </td>
```

```
      <td> <%= rs.getString(2) %> </td>
      <td> <%= rs.getString(3) %> </td>
    </tr>
...
```

message.jsp

The previous example, view.jsp, displays the digest and allows the user to click on the message ID to view a particular message. To do so, it creates a hyperlink to message.jsp, with an ID attribute (in the form of an HTTP **GET** request) indicating which message should be selected. The message.jsp page retrieves the ID from the implicit request object and uses a JDBC **PreparedStatement** to retrieve the message, as shown in the figure below.

message.jsp (continued)

Next, we the need to convert the message text, which is stored as a clob, to character data. For this demonstration, we just convert the clob directly into a string, which is displayed to the user. Again, repetitive code has been replaced with ellipses; look at the included source

code for the complete files.

```
...
  String selectSQL = "SELECT author, title, message "
   + "FROM messages WHERE id = ?"  ;
  String author ;
  String title ;
  String message ;
%>
...
  PreparedStatement pstmt = con.prepareStatement(selectSQL) ;
  pstmt.setInt(1,
    Integer.parseInt(request.getParameter("id"))) ;
  ResultSet rs = pstmt.executeQuery() ;
  rs.next() ;
  author = rs.getString("author") ;
  title = rs.getString("title") ;
  Clob clob = rs.getClob("message") ;
  message = clob.getSubString(1, (int)clob.length()) ;
  rs.close();
  pstmt.close();
  con.close() ;
%>
<ul>
  <li> Author: <%= author %> </li>
  <li> Title:  <%= title %> </li>
  <li> Message:<p/> <%= message %> </li>
</ul>
<hr>
Return to <a href="view.jsp">digest</a> view.
</body>
</html>
```

error.jsp

Before finishing our JSP page discussion, let's take a look at the JSP page's error-handling mechanism. Recall that, as part of the **page** directive, the error.jsp page handles all error conditions. The figure below displays the result of calling error.jsp. Of course, you could change error.jsp to perform whatever functionality you feel appropriate. Error JSP pages have access to the **exception implicit** object, which allows you, the developer, to have information pertaining to the relevant problem. This information can be displayed to the user, dumped into an application log (as in the example below), or even e-mailed to the JSP page developer.

Here is the error.jsp code:

```
<%@page isErrorPage="true" contentType="text/html"
 %>
<html>
<head><title>Error</title></head>
<body bgcolor="RED">
<h1> Unfortunately, we are unable to fulfill your request.</h1>
<hr/>
<h2> A more detailed error message has been recorded to
assist in uncovering the problem condition. Please try back later.
</h2>
<%
application.log(exception.getMessage()) ;
 %>
<hr/>
</body>
</html>
```

OK, this also works, but ...

The JSP page solution works, too, but just like the servlet-only approach, it probably leaves you feeling a little unsettled. First, the JSP page solution has the opposite problem that the servlet approach did; that is, now Java code is mixed in with HTML. This is probably an acceptable solution for experienced Java developers, but Web designers won't want to have to mess with Java code, nor should they. For small projects, maybe only one person does everything; but for large projects (or Web sites) with hundreds of Web pages, responsibilities are generally divided into Web developers and Java programmers. What is needed is a

solution that takes advantage of this natural separation.

Other problems also exist, such as the use of JSP page declaration elements. Because JSP pages are translated into servlets, the declaration element is essentially translated into class variables. Thus they have global scope, which is probably not a good thing. Another problem is that database-specific code is in the JSP pages. What happens if the database table is changed to add a column? Someone will now have to go in and modify the JSP pages appropriately. There has to be a better way.

Section 4. A Model Two approach

JSP, servlets, and JavaBeans

In contrast to the two previous approaches, a better solution is to delegate responsibility to different components. This approach builds on the Model-View-Controller design pattern. First, you have a servlet, which acts as the Controller. All user requests are funneled through a servlet, which delegates the actions out to the appropriate worker servlets. This design allows for a single sign-on approach where all authentication can be encapsulated and carried along throughout the Web application.

The model is encapsulated by JavaBeans' beans, which maintain the data throughout the application. In this approach, the worker servlets are responsible for creating and managing the beans. Finally, the view functionality is covered by JSP pages. The JSP pages extract the model from the beans and present the data to the user. This approach naturally allows separation of responsibilities: The servlet has the bulk of the heavy lifting, including database interactions, while the JSP page handles all the presentation logic. Note that while it isn't done in this tutorial, you can actually strive for no Java code in a JSP page through the use of custom JSP page actions (which are created using JSP tag libraries).

The Controller servlet

The following servlet performs the Controller function. Essentially, all it needs to do is extract the action parameter from the HTTP request and, depending on what the action is, call the appropriate worker servlet. If you want to add in authentication, modify this servlet appropriately.

```
import javax.servlet.*;
import javax.servlet.http.*;
public class Controller extends HttpServlet {
  public void init(ServletConfig config) throws ServletException {
    super.init(config);
  }
  public void destroy() {
  }
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, java.io.IOException {
    processRequest(request, response);
  }
  protected void doPost(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, java.io.IOException {
    processRequest(request, response);
  }
  protected void
    processRequest(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, java.io.IOException {
    String action = request.getParameter("action") ;
    String dispatch ="/servlet/" ;
    if(action == null)
      dispatch += "Digest" ;
    else if(action.equals("message"))
      dispatch += "Message" ;
    else if(action.equals("author"))
      dispatch += "Author" ;
    RequestDispatcher dispatcher = request.getRequestDispatcher(dispatch) ;
    dispatcher.forward(request, response) ;
```

```
  }
  public String getServletInfo() {
    return "Controlling Servlet for Model2 Demonstration";
  }
}
```

The Digest servlet

The first worker servlet shown is the Digest worker. This servlet extracts the digest data from the database and creates a **Vector** of **DigestMessageBean** objects. At the conclusion, this servlet calls the appropriate JSP page to display the data.

```
...
import com.persistentjava.DigestMessageBean ;
public class Digest extends HttpServlet {
...
  protected void
    processRequest(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, java.io.IOException {
    Connection con = null ;
    try{
      con = ds.getConnection("java", "sun") ;
      java.sql.Statement stmt = con.createStatement() ;
      ResultSet rs = stmt.executeQuery(querySQL) ;
      ResultSetMetaData rsmd = rs.getMetaData() ;
      response.setContentType("text/html");
      java.io.PrintWriter out = response.getWriter();
      Vector digest = new Vector() ;
      while(rs.next()) {
        DigestMessageBean dmb = new DigestMessageBean() ;
        dmb.setId(rs.getInt(1)) ;
        dmb.setTitle(rs.getString(2)) ;
        dmb.setAuthor(rs.getString(3)) ;
        digest.addElement(dmb) ;
      }
      rs.close();
      stmt.close();
      request.setAttribute("digest", digest) ;
      String dispatch ="/jsp/viewbean.jsp" ;
      RequestDispatcher dispatcher = request.getRequestDispatcher(dispatch) ;
      dispatcher.forward(request, response) ;
    }catch(SQLException ex){
...
```

viewbean.jsp

Now the digest information can be displayed to the user. In contrast to the original **view.jsp** page, you need to extract only the relevant data from the **Vector** of **DigestMessageBean** objects. First, tell the JSP page to use a JavaBeans bean, then extract the beans and generate the HTML table. Here is the viewbean.jsp code:

```
<%@
  page
  contentType="text/html"
  import="java.util.*"
%>
<html>
```

```
<head><title>Servlet-JSP Page</title></head>
<body>
<h2> Message Digest </h2>
<hr/>
<h4> Click on the message id to view a specific message </h4>
<table border="4">
<tr>
  <th>ID</th>
  <th>Title</th>
  <th>Author</th>
</tr>
<jsp:useBean
  id="dmb"
  scope="request"
  class="com.persistentjava.DigestMessageBean"/>
<%
  Vector digest = (Vector)request.getAttribute("digest") ;
  for (Enumeration e = digest.elements() ; e.hasMoreElements() ;) {
    dmb =(com.persistentjava.DigestMessageBean)e.nextElement();
%>
    <tr>
      <td> <a href="Controller?action=message&id=<%= dmb.getId() %>">
           <%= dmb.getId() %>
      </td>
      <td> <%= dmb.getTitle() %> </td>
      <td> <%= dmb.getAuthor() %> </td>
    </tr>
<%
     }
%>
</table>
</body>
</html>
```

DigestMessageBean

What about the **`View`**? This requires us to write a **`DigestMessageBean`** class in order to encapsulate the data appropriately.

```
package com.persistentjava ;
import java.beans.*;
import java.io.Serializable ;
public class DigestMessageBean implements Serializable {
  private int id ;
  private String title ;
  private String author;
  public DigestMessageBean() {
  }
  public int getId() {
    return id ;
  }
  public String getTitle() {
    return title ;
  }
  public String getAuthor() {
    return author ;
  }
  public void setId(int value) {
    id = value ;
  }
  public void setTitle(String value) {
```

```
    title = value ;
  }
  public void setAuthor(String value) {
    author = value ;
  }
}
```

The message worker

Now you can easily add more worker servlets to the Web application. Look carefully at the **viewbean.jsp** page -- the hyperlink calls the Controller servlet with the action parameter set to **message**, and an additional **id** attribute set to the appropriate message ID (just like the JSP page example).

```
...
import com.persistentjava.MessageBean ;
public class Message extends HttpServlet {
...
  private String querySQL = "SELECT title, author, message "
    + "FROM messages WHERE id = ?" ;
 ...
  protected void
    processRequest(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, java.io.IOException {
    Connection con = null ;
    try{
      con = ds.getConnection("java", "sun") ;
      PreparedStatement pstmt = con.prepareStatement(querySQL) ;
      pstmt.setInt(1, Integer.parseInt(request.getParameter("id"))) ;
      ResultSet rs = pstmt.executeQuery() ;
      ResultSetMetaData rsmd = rs.getMetaData() ;
      rs.next() ;
      Clob clob = rs.getClob("message") ;
      MessageBean mb = new MessageBean() ;
      mb.setTitle(rs.getString("title")) ;
      mb.setAuthor(rs.getString("author")) ;
      mb.setMessage(clob.getSubString(1, (int)clob.length())) ;
      rs.close();
      pstmt.close();
      request.setAttribute("message", mb) ;
      String dispatch ="/jsp/messagebean.jsp" ;
      RequestDispatcher dispatcher = request.getRequestDispatcher(dispatch) ;
      dispatcher.forward(request, response) ;
    }catch(SQLException ex){
 ...
```

messagebean.jsp

This JSP page is considerably simpler because it uses one bean, the **MessageBean**, which in turn generates the relevant HTML.

```
<%@
  page
  language="java"
  errorPage="error.jsp"
  contentType="text/html"
%>
```

```
<html>
<head><title>JSP Message Display</title></head>
<body>
<h2> Message Display </h2>
<hr/>
<jsp:useBean
  id="mb"
  scope="request"
  class="com.persistentjava.MessageBean"/>
<%
  mb = (com.persistentjava.MessageBean)request.getAttribute("message") ;
%>
<ul>
  <li> Author:
       <a href="Controller?action=author&author=<%= mb.getAuthor() %>">
       <%= mb.getAuthor() %>
       </a>
  </li>
  <li> Title:  <%= mb.getTitle() %> </li>
  <li> Message:<p/> <%= mb.getMessage() %> </li>
</ul>
<hr>
Return to <a href="Controller">digest</a> view.
</body>
</html>
```

MessageBean

Again, we must create the appropriate **MessageBean** class in order to encapsulate the message data.

```
package com.persistentjava ;
import java.beans.*;
import java.io.Serializable ;
public class MessageBean implements Serializable {
  private String message ;
  private String title ;
  private String author;
  public MessageBean() {
  }
  public String getTitle() {
    return title ;
  }
  public String getAuthor() {
    return author ;
  }
  public String getMessage() {
    return message ;
  }
  public void setTitle(String value) {
    title = value ;
  }
  public void setAuthor(String value) {
    author = value ;
  }
  public void setMessage(String value) {
    message = value ;
  }
}
```

The author worker

One final worker servlet example is the author servlet, which is interesting because it extracts an image from the database and sends it to the client. First, the content type must be set to *image/jpeg*. Next, extract the blob from the database, extract the data as an array of bytes, and finally, write the array of bytes directly to the **`ServerOutputStream`** (which is optimized to handle binary data). We can do this because the JPEG image is stored directly in the database, eliminating the need for any special JPEG encoders. Recall that an image was inserted for the user "rjb" (which is me); when the author rjb is selected, the Web page below is displayed.

The author worker: Relevant code

```
...
public class Author extends HttpServlet {
...
  private String selectSQL = "SELECT photo FROM authors WHERE author = ?"  ;
...
  protected void
    processRequest(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, java.io.IOException {
    Connection con = null ;
    try{
      con = ds.getConnection("java", "sun") ;
      PreparedStatement pstmt = con.prepareStatement(selectSQL) ;
      pstmt.setString(1, request.getParameter("author")) ;
      ResultSet rs = pstmt.executeQuery() ;
```

```
        rs.next() ;
        Blob blob = rs.getBlob("photo") ;
        byte[] data = blob.getBytes(1, (int)blob.length()) ;
        response.setContentType("image/jpeg") ;
        ServletOutputStream sos = response.getOutputStream() ;
        sos.write(data) ;
        sos.flush() ;
        sos.close() ;
        rs.close();
        pstmt.close();
 ...
```

Review

As you probably gleaned from this section, the Model Two -- or MVC approach -- provides significant advantages over the other two approaches. First, by encapsulating the different responsibilities, an application has been made that is easier to maintain and can still be extended easily. Second, this approach makes it easy to divide responsibilities for different components of the Web application among those who can best do them. For example, you can allow a Web designer to control the JSP pages, while a Java programmer can control the programming logic. If necessary, the database code could be encapsulated to provide even finer control of the application and those who build and maintain it.

We found a better way!

Section 5. Summary

Overview

This tutorial built on the database and tables created in the tutorial "Advanced database operations with JDBC" to build a Web application. After finishing this tutorial, you should understand how to:

* Write a servlet that communicates with a database
* Write a JSP page that communicates with a database
* Write a Controller servlet
* Write a Model Two Web application

JSP page and servlet resources

Source code

* Download the source code (examples.zip) used in this tutorial.

* Download the source code from the tutorial "Advanced database operations with JDBC" (brunner2-code.zip).

Servlet information

* Because servlets provide a component-based, platform-independent method for building Web-based applications, without the performance limitations of CGI programs, they are the technology of choice for extending and enhancing Web servers. For more information, visit the Java official servlets *home page*.

* Jeanne Murray's tutorial "*Building Java HTTP Servlets*" (developerWorks, September 2000) teaches basic concepts about servlets: what, why, and how. She also provides some practical experience writing simple HTTP servlets.

* Jeanne's follow-up servlet tutorial (developerWorks, December 2000) provides details on incorporating *session tracking* in your servlets.

JSP technology

* For more information on JavaServer Pages technology, visit the official *home page* from Sun Microsystems.

* In "*Filtering tricks for your Tomcat*" (developerWorks, June 2001) Sing Li offers a look at how servlet filtering has changed in Tomcat 4 (Servlet 2.3).

* Noel Bergman's tutorial "*Introduction to JavaServer Pages technology*" (developerWorks, October 2001) introduces the concepts, syntax, and semantics of JSP pages complete with topical examples that illustrate each element and illuminate important issues.

* The *JSP Insider* Web site is a great resource for JSP page developers. In addition to technology news and articles, the are also numerous code snippets and examples available for download.

Model Two

* Dan Malks and Sameer Tyagi offer a comprehensive look at the fundamentals of *JSP architecture* (developerWorks, February 2001). The authors discuss how the JSP architecuture allows for a distinction between the roles of Web designer and software developer. The article includes an overview of the Model Two architecture.

* *Struts* is an open source package from the Apache Software Foundation that simplifies Model Two Web design.

* "*Struts, an open-source MVC implementation* " (developerWorks, February 2001) by Malcolm Davis introduces Struts, a Model-View-Controller (MVC) implementation that uses servlets and JavaServer Pages (JSP) technology and demonstrates how Struts can help developers control change in Web projects.

JDBC and database resources

JDBC information

* Visit the official *JDBC home page* for the JDBC 2.0 and 3.0 specifications and other information.

* *JDBC API Tutorial and Reference, Second Edition* (Addison-Wesley, 1999) by White, Fisher, Cattell, Hamilton, and Hapner is *the* reference for JDBC developers.

* "*Managing database connections with JDBC*" and "*Advanced database operations with JDBC*" (developerWorks, November 2001), both by Robert Brunner, develop and sharpen your JDBC skills.

* Lennart Jorelid shows you how to use JDBC for *industrial-strength performance* .

* "*What's new in JDBC 3.0*" (develperWorks, July 2001) by Josh Heidebrecht provides an overview of the new features and enhancements in the new spec.

* "*An easy JDBC wrapper*" (developerWorks, August 2001) by Greg Travis describes a simple wrapper library that makes basic database usage a snap.

* The *Java application development with DB2* Web site provides an important collection of useful DB2 and Java links.

* The *Java Naming and Directory Interface* (JNDI) Web site provides a wealth of information regarding naming services, including an excellent JNDI tutorial. To use the JNDI examples, you will need to download the filesystem context provider, which is available from the site.

JDBC driver

* To find a JDBC driver for a particular database, visit Sun's *searchable database of JDBC drivers*.
* *Merant* is a third-party vendor that provides JDBC drivers for a range of databases.
* *i-net software* is another third-party vendor for JDBC drivers.
* *SourceForge* offers an open source JDBC driver for the MySQL database.

Database

* Much of the example code in this tutorial was developed using *DB2 Universal Database*. If you're using this platform or wish to learn more about it from a technical perspective, visit the *DB2 Developer Domain*, a centralized technical resource for the DB2 developer community.
* For more reading on SQL, see *SQL for Web Nerds*, which contains detailed discussions on data modeling, construction queries, triggers, views, trees, and data warehousing.

Feedback

We welcome your feedback on this tutorial and look forward to hearing from you. Additionally, you are welcome to contact the author, Robert Brunner, directly at *rjbrunner@pacbell.net*.

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial Building tutorials with the Toot-O-Matic demonstrates how to use the Toot-O-Matic to create your own tutorials. developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.