UNIT V: Integrating Servlets and JSP, Accessing database with JDBC

Understanding the need for Model View Controller, MVC Framework, Architecture of approach, Implementing MVC with *RequestDispathcher,,* Summarizing MVC code, Using JDBC in General, Basic JDBC Examples, Simplifying Database Access with JDBC Utilities, Using Prepared Statements

UNDERSTANDING THE NEED FOR MODEL VIEW CONTROLLER

Servlets are great when your application requires a lot of real programming to perform its task. Servlets can use cookies, track sessions, save information between requests, compress pages, access databases and perform many other tasks flexibly and efficiently.

But, generating HTML with servlets can be tedious and can produce a result that is hard to modify.

JSP lets you separate much of the presentation from the dynamic content. That way, you can write the HTML in the normal manner, even using HTML-specific tools and putting your Web content developers to work on your JSP documents.

JSP expressions, scriptlets, and declarations let you insert simple Java code into the servlet that results from the JSP page, and directives let you control the overall layout of the page.

If we are able to JSP & Servlets then we can utilizes the full strength of both the technologies.

If we integrate or combine these two technologies for creating dynamic pages then it is called MFC architecture or model2 architecture.

Use the MVC architecture.

In MVC architecture the original request is handled by a servlet. The servlet invokes the business-logic and data-access code and creates beans to represent the results (that's the model).

Then, the servlet decides which JSP page is appropriate to present those particular results and forwards the request there (the JSP page is the view).

The servlet decides what business logic code applies and which JSP page should present the results (the servlet is the controller).

MVC FRAMEWORKS:

MVC Frameworks is used to separate the code that creates and manipulates the data from the code that presents the data.

Some basic tools are needed to implement this presentation-layer & data manipulation layer separation. In very complex applications, MVC framework is sometimes beneficial.

The most popular of these frameworks is **Apache Struts**;

Struts is useful and widely used, you should not feel that you must use Struts in order to apply the MVC approach.

For simple and moderately complex applications, implementing MVC using RequestDispatcher is straightforward and flexible.

ARCHITECTURE OR APPROACH?

The term "architecture" means "overall system design." Although many systems are really designed with MVC, but it is not necessary to redesign your overall system just to make use of the MVC approach. Not at all.

It is quite common for applications to handle some requests with servlets, other requests with JSP pages, and still others with servlets and JSP acting in conjunction.

You don't need to redesign your entire system architecture just to use the MVC approach: You can simply apply MVC for the parts of your application where it fits best.

IMPLEMENTING MVC WITH RequestDispatcher

The most important point about MVC is the idea of separating the business logic and data access layers from the presentation layer. Following is the summary of the required steps for implementing MVC.

RequestDispatcher is a predefined class used to implements MVC.

1. Define beans to represent the data. Beans are just Java objects that follow a few simple conventions. Your first step is defining beans to represent the results that will be presented to the user.

Servlet or other Java routine will be creating the beans; the requirement for creating an bean is default constructor, keeping variables private and using accessor methods that follow the get/set naming convention.

Since the JSP page will only access the beans, not create or modify them, a common practice is to define value objects: objects that represent results but have little or no additional functionality.

2. Use a servlet to handle requests. In most cases, the servlet reads request parameters

Once the bean classes are defined, the next task is to write a servlet to read the request information.

Although the servlets use the normal techniques to read the request information and generate the data, they do not use the normal techniques to output the results.

In fact, with the MVC approach the servlets do not create any output; the output is completely handled by the JSP pages. So, the servlets do not call response.set- ContentType, response.getWriter, or out.println.

3. Populate the beans. The servlet invokes business logic (application specific code) or data-access code to obtain the results. The results are placed in the beans that were defined in step 1.

After you read the form parameters, you use them to determine the results of the request. These results are determined in a completely application-specific manner. You might call some business logic code, invoke an Enterprise JavaBeans component, or query a database. No matter how you come up with the data, you need to use that data to fill in the value object beans that you defined in the first step.

4. Store the bean in the request, session, or servlet context. The servlet calls setAttribute on the request, session, or servlet context objects to store a reference to the beans that represent the results of the request.

A servlet can store data for JSP pages in three main places: in the HttpServlet- Request, in the HttpSession, and in the ServletContext.

• Storing data that the JSP page will use only in this request. First, the servlet would create and store data as follows:

```
ValueObject value = new ValueObject(...);

request.setAttribute("key", value);
```

Next, the servlet would forward the request to a JSP page that uses the following to retrieve the data.

<jsp:useBean id="key" type="somePackage.ValueObject" scope="request" />

Note that request attributes have nothing to do with request parameters or request headers. The request attributes are independent of the information coming from the client; they are just application-specific entries in a hash table that is attached to the request object. This table simply stores data in a place that can be accessed by both the current servlet and JSP page, but not by any other resource or request.

• Storing data that the JSP page will use in this request and in later requests from the same client. First, the servlet would create and store data as follows:

ValueObject value = new ValueObject(...);

HttpSession session =

request.getSession();

session.setAttribute("key", value);

Next, the servlet would forward to a JSP page that uses the following to retrieve the data:

<jsp:useBean id="key" type="somePackage.ValueObject" scope="session" />

• Storing data that the JSP page will use in this request and in later requests from any client. First, the servlet would create and store data as follows:

ValueObject value = new ValueObject(...);

getServletContext().setAttribute("key", value);

Next, the servlet would forward to a JSP page that uses the following to retrieve the data:

<jsp:useBean id="key" type="somePackage.ValueObject" scope="application" />

5. Forward the request to a JSP page. The servlet determines which JSP page is appropriate to the situation and uses the forward method of RequestDispatcher to transfer control to that page.

You forward requests with the forward method of RequestDispatcher. You obtain a RequestDispatcher by calling the getRequestDispatcher method of ServletRequest, supplying a relative address.

You are permitted to specify addresses in the WEB-INF directory; clients are not allowed to directly access files in WEB-INF, but the server is allowed to transfer control there.

Using locations in WEB-INF prevents clients from inadvertently accessing JSP pages directly, without first going through the servlets that create the JSP data.

Request Forwarding Example

```
public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
{
String operation = request.getParameter("operation");
if (operation == null)
{
operation = "unknown";
}
String address;
if (operation.equals("order"))
{
address = "/WEB-INF/Order.jsp";
} else if (operation.equals("cancel"))
{
address = "/WEB-INF/Cancel.jsp";
}
else
{
address = "/WEB-INF/UnknownOperation.jsp";
}
RequestDispatcher dispatcher = request.getRequestDispatcher(address);
dispatcher.forward(request, response);
}
```

6. **Extract the data from the beans.** The JSP page accesses beans with jsp:useBean and a scope matching the location of step 4. The page then uses jsp:getProperty to output the bean properties.

The JSP page does not create or modify the bean; it merely extracts and displays data that the servlet created.

Once the request arrives at the JSP page, the JSP page uses jsp:useBean and jsp:getProperty to extract the data.

There are two differences however:

• The JSP page never creates the objects. The servlet, not the JSP page, should create all the data objects. So, to guarantee that the JSP page will not create objects, you should use

<jsp:useBean ... type="package.Class" />

instead of

<jsp:useBean ... class="package.Class" />.

• The JSP page should not modify the objects. So, you should use jsp:getProperty but not jsp:setProperty.

The scope you specify should match the storage location used by the servlet. For example, the following three forms would be used for request-, session-, and application- based sharing, respectively.

<jsp:useBean id="key" type="somePackage.SomeBeanClass" scope="request" />

<jsp:useBean id="key" type="somePackage.SomeBeanClass" scope="session" />

<jsp:useBean id="key" type="somePackage.SomeBeanClass" scope="application" />

SUMMARIZING MVC CODE:

Summarizing the MVC code that would be used for understanding the data sharing methods, data sharing methods mainly categories into: request-based, session based, and application-based MVC approaches.

Request-Based Data Sharing

With request-based sharing, the servlet stores the beans in the HttpServlet- Request, where they are accessible only to the destination JSP page.

Servlet

ValueObject value = new ValueObject(...);

request.setAttribute("key", value);

RequestDispatcher dispatcher =request.getRequestDispatcher("/WEB-INF/SomePage.jsp");

dispatcher.forward(request, response);

JSP Page

<jsp:useBean id="key" type="somePackage.ValueObject" scope="request" />

<jsp:getProperty name="key" property="someProperty" />

Session-Based Data Sharing

With session-based sharing, the servlet stores the beans in the HttpSession, where they are accessible to the same client in the destination JSP page or in other pages.

Servlet

ValueObject value = new ValueObject(...);

HttpSession session =

request.getSession();

session.setAttribute("key", value);

RequestDispatcher dispatcher = request.getRequestDispatcher("/WEB

INF/SomePage.jsp"); dispatcher.forward(request, response);

JSP Page

<jsp:useBean id="key" type="somePackage.ValueObject" scope="session" />

<jsp:getProperty name="key" property="someProperty" />

Application-Based Data Sharing

With application-based sharing, the servlet stores the beans in the Servlet- Context, where they are accessible to any servlet or JSP page in the Web application.

To guarantee that the JSP page extracts the same data that the servlet inserted, you should synchronize your code as below.

Servlet

synchronized(this) {

ValueObject value = new ValueObject(...);

getServletContext().setAttribute("key", value);

```
RequestDispatcher dispatcher = request.getRequestDispatcher("/WEB-INF/SomePage.jsp");
```

dispatcher.forward(request, response);

}

JSP Page

<jsp:useBean id="key" type="somePackage.ValueObject" scope="application" />

<jsp:getProperty name="key" property="someProperty" />

USING JDBC IN GENERAL

JDBC (Java Database Connectivity) provides a standard library for accessing relational databases. By using the JDBC API, you can access a wide variety of SQL databases with exactly the same Java syntax.

There are seven standard steps used for accessing databases.

1. Load the JDBC driver. To load a driver, you specify the classname of the database driver in the **Class.forName** method. By doing so, you automatically create a driver instance and register it with the JDBC driver manager.

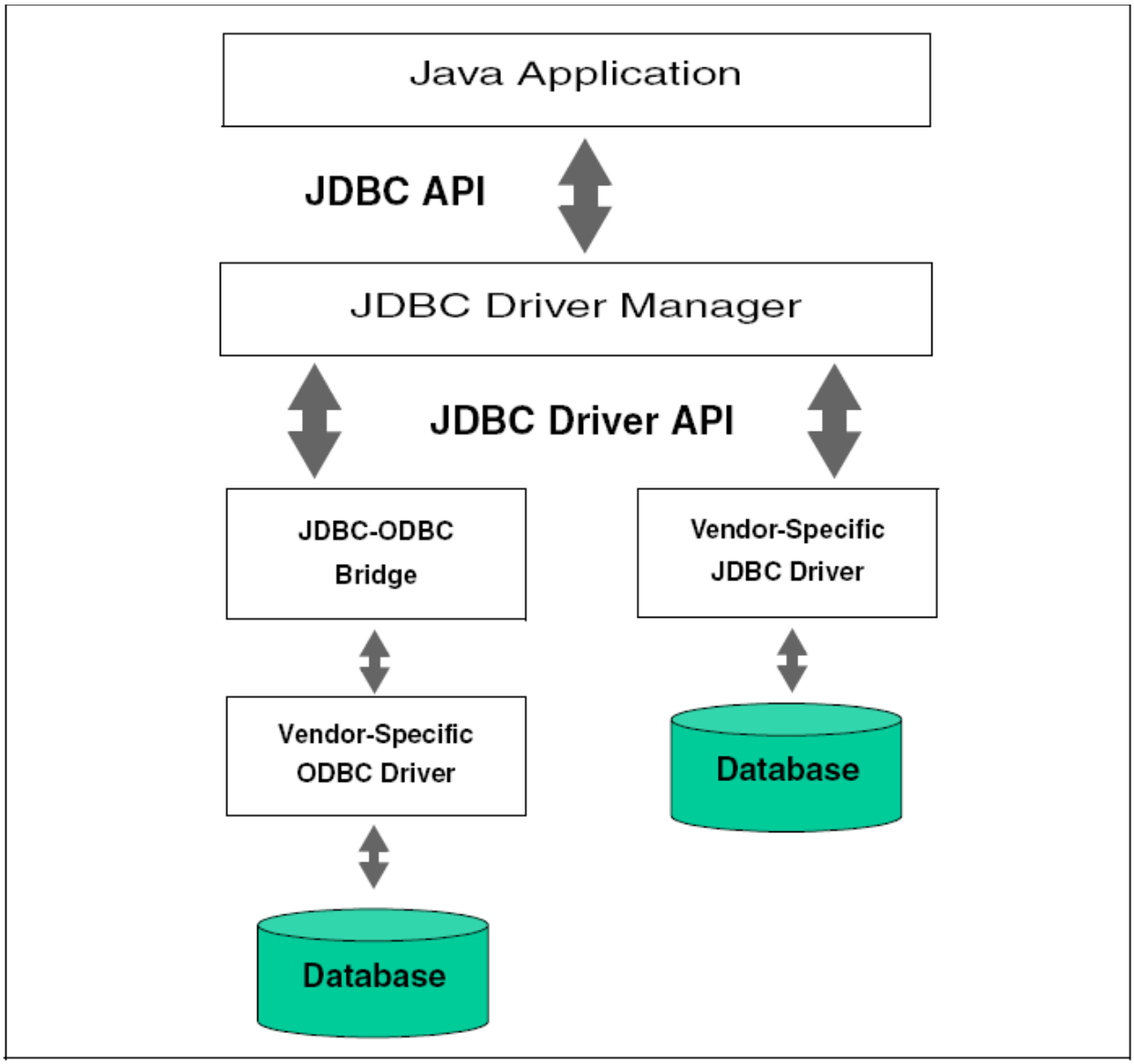
The driver is the piece of software that knows how to talk to the actual database server.

Class.forName is method used to load the driver for accessing database, the Syntax for loading the driver is as follows:

This method takes a string representing a fully qualified classname or driver name and loads the corresponding class. This call could throw Exception, so it should be inside a try/catch block as shown below.

```
try
{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
}
catch(Exception e)
{
System.out.println("Error loading driver: " + e.getMessage());
}
```

Following Figure shows implementation data access in JAVA.



2. Define the connection URL. In JDBC, a connection URL specifies the server host, port, and database name with which to establish a connection.

Once you have loaded the JDBC driver, you must specify the location of the database server.

For accessing the MS-Access database using ODBC the URL will be:

String msAccessURL = "jdbc:odbc:cocsit";

Here **cocsit** is the ODBC name specified in control panel.

3. **Establish the connection**. With the connection URL, username, and password, a network connection to the database can be established. Once the connection is established, database queries can be performed until the connection is *closed.*

To make the actual network connection, pass the URL, database username, and database password to the getConnection method of the DriverManager class, as illustrated in the following example. Note that getConnection throws an SQLException, so you need to use a try/catch block.

For accessing MS-Access database using ODBC

Connection con =DriverManager.getConnection("jdbc:odbc:cocsit");

For accessing Oracle database using ODBC:

Connection con =DriverManager.getConnection("jdbc:odbc:cocsit", "scott", "tiger");

The Connection class includes other useful methods, which we briefly describe below.

- prepareStatement: Creates precompiled queries for submission to the database.
- prepareCall: Accesses stored procedures in the database.
- rollback/commit: Controls transaction management.
- close: Terminates the open connection.
- isClosed: Determines whether the connection timed out or was explicitly closed.

4. Create a Statement object. Creating a Statement object enables you to send queries and commands to the database.

A Statement object is used to send queries and commands to the database. It is created from the Connection using createStatement as follows.

Statement statement = connection.createStatement();

Most, but not all, database drivers permit multiple concurrent Statement objects to be open on the same connection.

5. Execute a query or update. Given a Statement object, you can send SQL statements to the database by using the execute, executeQuery, executeUpdate, or executeBatch methods.

Once you have a Statement object, you can use it to send SQL queries by using the executeQuery method, which returns an object of type ResultSet. Here is an example.

String query = "SELECT col1, col2, col3 FROM sometable";

ResultSet resultSet = statement.executeQuery(query);

The following list summarizes commonly used methods in the Statement class.

• executeQuery. Executes an SQL query and returns the data in a

ResultSet. The ResultSet may be empty, but never null.

• executeUpdate. Used for UPDATE, INSERT, or DELETE commands.

Returns the number of rows affected, which could be zero. Also provides support for Data Definition Language (DDL) commands, for example, CREATE TABLE, DROP TABLE, and ALTER TABLE.

• executeBatch. Executes a group of commands as a unit, returning an array with the update counts for each command. Use addBatch to add a command to the batch group. Note that vendors are not required to implement this method in their driver to be JDBC compliant.

6. Process the results. When a database query is executed, a ResultSet is returned. The ResultSet represents a set of rows and columns that you can process by calls to next and various getXxx methods.

The simplest way to handle the results is to use the next method of ResultSet to move through the table a row at a time. Within a row, ResultSet provides various getXxx methods that take a column name or column index as an argument and return the result in a variety of different Java types. For instance, use getInt if the value should be an integer, getString for a String, and so on for most other data types. If you just want to display the results, you can use getString for most of the column types. However, if you use the version of getXxx that takes a column index (rather than a column name), note that columns are indexed starting at 1 (following the SQL convention), not at 0 as with arrays, vectors, and most other data structures in the Java programming language.

Here is an example that prints the values of the first two columns and the first name and last name, for all rows of a ResultSet.

```
while(resultSet.next())

{ System.out.println(resultSet.getString(1) + " " +

resultSet.getString(2) + " " +

resultSet.getString("firstname") + " "
```

resultSet.getString("lastname"));

}

7. Close the connection. When you are finished performing queries and processing results, you should close the connection, releasing resources to the database.

The following list summarizes useful ResultSet methods.

• next / previous: Moves the cursor to the next (any JDBC version) or previous (JDBC version 2.0 or later) row in the ResultSet, respectively.

• relative / absolute: The relative method moves the cursor a relative number of rows, either positive (up) or negative (down). The absolute method moves the cursor to the given row number. If the absolute value is negative, the cursor is positioned relative to the end of the ResultSet (JDBC 2.0).

• getXxx : Returns the value from the column specified by the column name or column index as an Xxx Java type (see java.sql.Types). Can return 0 or null if the value is an SQL NULL.

• wasNull : Checks whether the last getXxx read was an SQL NULL. This check is important if the column type is a primitive (int, float, etc.) and the value in the database is 0. A zero value would be indistinguishable from a database value of NULL, which is also returned as a 0. If the column type is an object (String, Date, etc.), you can simply compare the return value to null.

• findColumn: Returns the index in the ResultSet corresponding to the specified column name.

• getRow: Returns the current row number, with the first row starting at 1 (JDBC 2.0).

• getMetaData. Returns a ResultSetMetaData object describing the ResultSet. ResultSetMetaData gives the number of columns and the column names.

BASIC JDBC EXAMPLES, SIMPLIFYING DATABASE ACCESS WITH JDBC UTILITIES, USING PREPARED STATEMENTS

[Study various example on database handling for insert, update, delete, select and using prepareStatment that are taken in class]

Simple Example to Insert the record into Ms Access Database.

```
<html>
<head>
<title>new user</title>
</head>
<body bgcolor="#FFFFFF" text="#000000">
<h1>Please enter your details</h1>
<form action="validate_new_user.jsp" method="post">
   <table cellspacing="5" cellpadding="5" border="1">
      <tr>
         <td align="right">Name:</td>
         <td><input type="text" name="NewName"></td>
      </tr>
      <tr>
         <td align="right">Email Address:</td>
         <td><input type="text" name="EmailAddress"></td>
      </tr>
      <tr>
         <td align="right">Nationality:</td>
         <td><input type="text" name="Nationality"></td>
      </tr>
      <tr>
         <td align="right">User Name:</td>
         <td><input type="text" name="UserName"></td>
      </tr>
      <tr>
         <td align="right">Password:</td>
         <td><input type="password" name="Password"></td>
      </tr>

   </table>
   <input type="submit" value="Submit">
</form>
</body>
</html>
```

validate_new_user.jsp

```
<html>
<head>
<title>Validate New User</title>
</head>
<body bgcolor="#FFFFFF" text="#000000">
<%@ page import="javax.servlet.*" %>
<%@ page import="javax.servlet.http.*" %>
<%@ page import="java.sql.*" %>
<%
      Connection con = null;

      String names = request.getParameter("NewName"); String eaddress
      = request.getParameter("EmailAddress"); String nation =
      request.getParameter("Nationality"); String userid =
      request.getParameter("UserName"); String pw =
      request.getParameter("Password");

      String queryText = "insert into user_details (\"fullName\", \"emailAddress\",

\"nationality\", \"username\", \"password\") values('"+names+"','"+eaddress+"','"+nation+"','"+

userid+"','"+pw+"')";

      try {
          Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
         con=DriverManager.getConnection("jdbc:odbc:cocsit","",""); Statement stat =
         con.createStatement();
         int r= stat.executeUpdate(queryText);
         stat.close();
         con.close();

      }
         catch (Exception e) { }
      response.sendRedirect("success.htm");
%>

</body>
</html>
```