|  |  |
| --- | --- |
|  | **CE212 Lab 6: Forward and Include, Using Databases**  **Introduction**  In this lab we shall work through the forward and include examples from part 5 of the lecture slides, then introduce the use of relational databases in the context of web applications.  **Using <jsp:forward>**  We shall first develop the forward example from part 5. Open the IntelliJ project you used for lab 5, and in the **webapps** folder create a new file called **forwarder.jsp** and give it the following contents:  <%@ page contentType="text/html;charset=UTF-8" language="java" %>  <jsp:useBean id="myBean" class="test.TestBean" scope = "request"/>  <jsp:setProperty name="myBean" property="x" value="12"/>  <html>  <head><title>Simple jsp page with a Forward</title></head>  <body>Place your content here  <jsp:forward page="UseBean.jsp" />  </body>  </html>  Note that it refers to two other files. **TestBean** is a Java class to be defined in a package called **test** and **UseBean.jsp** is the page that will use the value of the bean, extracting some data from it to display (in this case the value of the property x).  The code for these is given below:  package test;  public class TestBean {  int x = 99;  public String a;  public TestBean() { }  public TestBean(int x) {  this.x = x;  }  public int getX() {  return x;  }  public void setX(int x) {  this.x = x;  }  public String toString() {  return "Bean value = " + x + " : " + a;  }  }  <%@ page contentType="text/html;charset=UTF-8" language="java" %>  <jsp:useBean id="myBean" class="test.TestBean" scope="request"/>  <html>  <head><title>Simple Bean Usage</title></head>  <body>  <h2>Bean test: <%= myBean %></h2>  </body>  </html>  Create these files in the appropriate folders (see lab 5 if necessary) and run the web page.  **Page Inclusion**  In this simple example, we'll create some data on the requested page, and then use the <jsp:include> mechanism to include another page.  After the page has been included, control is returned to the caller.  The other files (**UseBean.jsp** and **TestBean.java**) are the same as for the forwarding example.  The JSP to include another page is given below.  <%@ page contentType="text/html;charset=UTF-8" language="java" %>  <jsp:useBean id="myBean" class="test.TestBean" scope = "request"/>  <jsp:setProperty name="myBean" property="x" value="11"/>  <html>  <head><title>Simple jsp page with an Include</title></head>  <body>Bean Value in Caller: <%= myBean %>  <%  // set a field that's not a property  myBean.a = "Set a non-property";  %>  <h2> Included file is below </h2>  <jsp:include page="UseBean.jsp" />  </body>  </html>  **Experiment:**try modifying the scope of the bean to **page** (in both or either of the caller and callee) and observe the effects.  What do you notice?  Try to predict the output of the program before running it, then see if your predictions were correct.  Note that as it stands, the program deletes each object after its details are printed.  **HSQLDB Introduction**  We shall use HSQLDB, a Pure java relational database. We first present code to save and retrieve data, before looking at their use in web applications  HSQLDB ([**http://hsqldb.org**](http://hsqldb.org/)) is a pure Java relational database.  Being pure Java makes it easy to install (simply place the **.jar** file on the classpath).  The user interface to HSQLDB is not as friendly as using MS Access, but it's a good choice for Java applications that require relational database connectivity.  Copy the file [**hsqldb.jar**](https://moodle.essex.ac.uk/pluginfile.php/886670/mod_resource/content/8/lab6/hsqldb.jar) to your **tomcat\lib** folder.  Create a new IntelliJ project and inside it create a package called shop. IntelliJ must be informed of the location of the jar file. Go to **File - Project Structure**, then click on **Libraries** and then on the **+** icon at the top of the frame. (If a brief drop-down menu appears select **Java**). Navigate to and select the HSQLDB jar file then click **OK**.  To check that you can run HSQLDB, create a new package called **shop** within your IntelliJ project and add the file [**HSQLTest.java**](https://moodle.essex.ac.uk/pluginfile.php/886670/mod_resource/content/8/lab6/HSQLTest.java) (taken from the HSQLDB user guide) to this package.  Spend some time studying the code, and then run it.  Running the first time should produce the following output:  0 Ford 100  1 Toyota 200  Running the second time should produce this:  0 Ford 100  1 Toyota 200  4 Ford 100  5 Toyota 200  Copy the file [**HsqlShopDB.java**](https://moodle.essex.ac.uk/pluginfile.php/886670/mod_resource/content/8/lab6/HsqlShopDB.java) into your **shop** package. The code includes methods to add products to the database, given Java objects of type **Product**.  You should also copy the files [**Product.java**](https://moodle.essex.ac.uk/pluginfile.php/886670/mod_resource/content/8/lab6/Product.java) and [**Basket.java**](https://moodle.essex.ac.uk/pluginfile.php/886670/mod_resource/content/8/lab6/Basket.java) into your **shop** package.  Run the HSQLShopDB program.  We now wish to examine it; the main parts are explained below.  **Connecting to HSQLDB**  The following code is the constructor - it loads the class files for the driver (if not already loaded) and then sets up a connection for use by other methods later::  public HSQLShopDB() {  try {  Class.forName("org.hsqldb.jdbc.JDBCDriver");  con = DriverManager.getConnection("jdbc:hsqldb:file:"  + dbFile, // filenames  "sa", // username  ""); // password  System.out.println("created con");  } catch (Exception e) {  System.out.println("Exception: " + e);  }  }  In addition to that, note the convenience method **update()**, which takes a string and executes the specified update on the database (opening and closing a statement to achieve this). It is declared as synchronized to prevent any attempts at simultaneous updates.  // use for SQL commands CREATE, DROP, INSERT and UPDATEE  // from TestDB in HSQLDB Guide.pdf  public synchronized void update(String expression) {  try {  Statement st = con.createStatement(); // statements  int i = st.executeUpdate(expression); // run the query  if (i == -1) {  System.out.println("db error : " + expression);  }  st.close();  }  catch (SQLException e) {  e.printStackTrace();  }  }  **Table Creation**  Tables can be created by calling the **update** method (see above) given an SQL 'Create Table' string.  The code for creating two of the shop tables is shown below::  public void createTables() {  update(  "CREATE TABLE Product (" +  " PID VARCHAR(256)," +  " name VARCHAR(256)," +  " price FLOAT,"  ) "  );  update(  "CREATE TABLE CustOrder (" +  " PID VARCHAR(256)," +  " OrderID VARCHAR(256)," +  " email VARCHAR(256)," +  " quanitity FLOAT," +  " price FLOAT" +  ") "  );  }  **Insertion**  Insertion can be done by calling the **update** method with an appropriate SQL insert statement.  The following methods are used to add some sample products into the database.  Note the use of the **String.format()** method to ease the process of creating the string.  Also note that in the SQL string, number types are not enclosed in quotes!  public void addTestProducts() {  addProduct(new Product("art1", "Stax", 49.99);  addProduct(new Product("art2", "Linez", 99.99);  addProduct(new Product("art3", "Stax", 200.0);  }  public void addProduct(Product p) {  String add = String.format("INSERT INTO Product VALUES " +  "('%s', '%s', '%s')",  p.PID, p.name, p.price); System.out.println(add);  update(add);  }  **Retrieval**  Care must be taken with retrieval: a **ResultSet** is only valid while a statement is open; this makes it harder to use a general-purpose **query()** method to act as a convenient analogue to our **update()** method.  The sample code below shows how a set of **Product** objects can be retrieved from the database.  public Collection<Product> getProductCollection(String query) {  LinkedList<Product> list = new LinkedList<Product>();  try {  Statement s = con.createStatement();  ResultSet rs = s.executeQuery(query);  while (rs.next()) {  Product product = new Product(  rs.getString("PID"),  rs.getString("name"),  rs.getDouble("price")  );  list.add(product);  }  return list;  }  catch (Exception e) {  System.out.println("Exception in getProducts(): " + e);  return null;  }  }  **Update**  Exercise: write a Java method to update the product table, increasing the price of all products by 10%.  **Deletion**  All the rows in a table can be deleted by executing the following kind of update:  db.update("DELETE FROM Product");  Use a WHERE clause to only delete selected rows..  A table can be deleted using the 'Drop TABLE' statement e.g.  db.update("Drop TABLE Product");  **Shutting Down**  This should be called on any open database connections before the Java virtual machine exits (e.g. before Tomcat is shutdown) :  public void shutdown() throws SQLException {  Statement st = con.createStatement();  // db writes out to files and performs clean shut down  // otherwise there will be an unclean shutdown  // when program ends  st.execute("SHUTDOWN");  con.close(); // if there are no other open connection  }  **An HSQLDB Web App**  So far we've only run the save and retrieval programs in stand-alone mode.  We will now create a JSP page **ListProducts.jsp**, which simply prints the name and price of each product in the database within a very simple web page.  The JSP is shown here as an image; you can base your code on this. (Note that this image is from a screen-shot of a ListProducts.jsp file that used an object-oriented database; your code should use shop instead of oodb. as the package name and RDB instead of OODB as a class name. This solution uses a **DummyDB**; we will simply exchange this for the real DB when ready (see below). The last import statement should initially be omitted - we have not yet written the **RDB**  ListProducts.jsp  You will need to create files DBInterface.java and DummyDB.java in the **shop** package. For our current purposes the interface need contain only a single method **Collection<Product> getAllProducts()**. The file DummyDB.java should simulate in Java the results of retrieving the contents of a database table; it should create a list of products (e.g. a linked list or an array list) directly in the Java code, and simply return that collection when the **getAllProducts()** method is called. You will need to create dummy data for each product - for example:  public class DummyDB implements DBInterface  { Collection<Product> prods;  public DummyDB()  { prods = new ...;  prods.add(new Product(  "Filter Coffee Machine", 17.99, "img/filter.jpg"))  // repeat this for several other products  }  public Collection<Product> getAllProducts() {  return prods;  }  }  This should produce output a bit like this, depending on what data you put in:  To make the JSP page work with live data, you need to provide an implementation of the **DBInterface** interface in a file called **RDB.java**.  Add to the JSP file the line to import this that was previously omitted.  Note that the existing methods of **QueryProducts.java** assume that the retrieved objects will be accessed while a database connection is open.  When working within a real web application, we need to take care about when connections are opened and closed. For now, we shall simply open the connection in the constructor.  Create the file **RDB.java** within the **shop** package; it contents should be as shown below  shop/RDB.java  package rdb;  import java.sql.\*;  import java.util.Collection;  import java.util.LinkedList;  public class RBB implements DBInterface {  private Connection con;  public OODB() {  try {  Class.forName("org.hsqldb.jdbc.JDBCDriver");  con = DriverManager.getConnection("jdbc:hsqldb:file:shopDB", "sa", "");  System.out.println("Created con");  } catch (Excedption e)  { System.out.println("Exception: " + e);  }  }  public Collection<Product> getAllProducts() {  // need something like the getAllProducts from assignment 2  }  } |