

The NetBeans E-commerce Tutorial - Connecting the Application to the Database

Tutorial Contents

1. [Introduction](#)
2. [Designing the Application](#)
3. [Setting up the Development Environment](#)
4. [Designing the Data Model](#)
5. [Preparing the Page Views and Controller Services](#)
6. **Connecting the Application to the Database**
 - [Adding Sample Data to the Database](#)
 - [Creating a Connection Pool and Data Source](#)
 - [Testing the Connection Pool and Data Source](#)
 - [Setting Context Parameters](#)
 - [Working with JSTL](#)
 - [Troubleshooting](#)
 - [See Also](#)
7. [Adding Entity Classes and Session Beans](#)
8. [Managing Sessions](#)
9. [Integrating Transactional Business Logic](#)
10. [Adding Language Support](#)
11. [Securing the Application](#)
12. [Testing and Profiling](#)
13. [Conclusion](#)

This tutorial unit focuses on communication between the database and the application. You begin by adding sample data to the database and explore some of the features provided by the IDE's SQL editor. You set up a data source and connection pool on the GlassFish server, and proceed by creating a JSP page that tests the data source by performing a simple query on the database.

This unit also addresses how the application retrieves and displays images necessary for web presentation, and how to set context parameters and retrieve their values from web pages. Once you are certain the data source is working correctly, you apply JSTL's `core` and `sql` tag libraries to retrieve and display category and product images for the [index](#) and [category](#) pages.



You can view a live demo of the application that you build in this tutorial: [NetBeans E-commerce Tutorial Demo Application](#).



Software or Resource	Version Required
NetBeans IDE	Java bundle, 6.8 or 6.9
Java Development Kit (JDK)	version 6
GlassFish server	v3 or Open Source Edition 3.0.1
MySQL database server	version 5.1
AffableBean project	snapshot 2
website images	n/a

Notes:

- The NetBeans IDE requires the Java Development Kit (JDK) to run properly. If you do not have any of the resources listed above, the JDK should be the first item that you download and install.
- The NetBeans IDE Java Bundle includes Java Web and EE technologies, which are required for the application you build in this tutorial.
- The NetBeans IDE Java Bundle also includes the GlassFish server, which you require for this tutorial. You could [download the GlassFish server independently](#), but the version provided with the NetBeans download has the added benefit of being automatically registered with the IDE.
- You can follow this tutorial unit without having completed previous units. To do so, perform the following three steps:


- **Set up your MySQL database server.** Follow the steps outlined in: [Communicating with the Database Server](#).
- **Create the `affablebean` schema on the database server.**
 - Click on `affablebean_schema_creation.sql` and copy (Ctrl-C; ⌘-C on Mac) the entire contents of the file.
 - Open the IDE's SQL editor. In the Services window (Ctrl-5; ⌘-5 on Mac), right-click the `affablebean` database connection () node and choose Execute Command. The IDE's SQL editor opens.
 - Paste (Ctrl-V; ⌘-V on Mac) the entire contents of the `affablebean.sql` file into the editor.
 - Click the Run SQL () button in the editor's toolbar. The script runs on your MySQL server. Tables are generated for the `affablebean` database.
- Open the [project snapshot](#) in the IDE. In the IDE, press Ctrl-Shift-O (⌘-Shift-O on Mac) and navigate to the location on your computer where you unzipped the downloaded file.

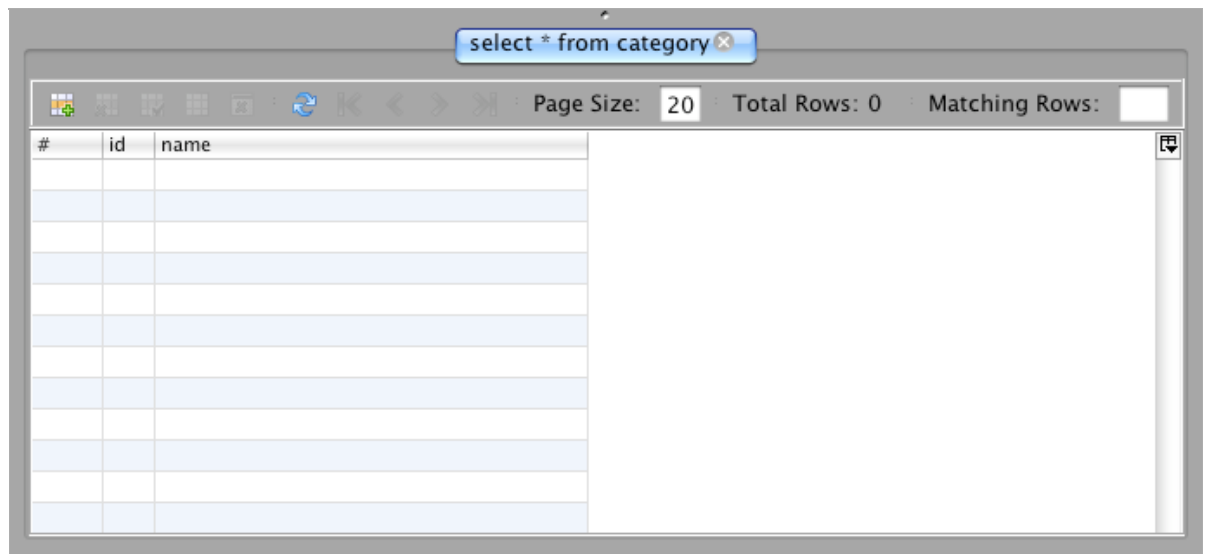
Adding Sample Data to the Database

Begin by adding sample data to the `category` and `product` tables. You can do this using the IDE's SQL editor, which allows you to interact directly with the database using native SQL. The IDE's SQL support also includes a GUI editor that enables you to add, remove, modify and delete table records.

- [category table](#)
- [product table](#)

category table

1. In the Services window (Ctrl-5; ⌘-5 on Mac), right-click the `category` table () node and choose View Data. The SQL editor opens and displays with a GUI representation of the `category` table in the lower region. Note that the table is empty, as no data has yet been added.




Also, note that the native SQL query used to generate the GUI representation is displayed in the upper region of the editor: 'select * from category'.

2. Delete 'select * from category' and enter the following SQL statement:

```
INSERT INTO `category` (`name`) VALUES ('dairy'),('meats'),('bakery'),('fruit & veg');
```


This statement inserts four new records, each with a unique entry for the 'name' column. Because the `id` column was specified as `AUTO_INCREMENT` when you created the schema, you do not need to worry about supplying a value.

3. Click the Run SQL () button in the editor's toolbar. The SQL statement is executed.

4. To confirm that the data has been added, run the 'select * from category' query again. To do so, you can use the SQL History window. Click the SQL History () button in the editor's toolbar and double-click the 'select * from category' entry. The SQL History window lists all SQL statements that you recently executed in the IDE.

Watch the screencast below to see how you can follow the above steps. When typing in the editor, be sure to take advantage of the IDE's code completion and suggestion facilities.

product table

1. Right-click the product table () node and choose Execute Command. Choosing the Execute Command menu option in the Services window opens the SQL editor in the IDE.
2. Copy and paste the following INSERT statements into the editor.

```
--  
-- Sample data for table `product`  
--  
  
INSERT INTO `product` (`name`, price, description, category_id) VALUES ('milk', 1.70,  
'semi skimmed (1L)', 1);  
INSERT INTO `product` (`name`, price, description, category_id) VALUES ('cheese',  
2.39, 'mild cheddar (330g)', 1);  
INSERT INTO `product` (`name`, price, description, category_id) VALUES ('butter',  
1.09, 'unsalted (250g)', 1);  
INSERT INTO `product` (`name`, price, description, category_id) VALUES ('free range  
eggs', 1.76, 'medium-sized (6 eggs)', 1);  
  
INSERT INTO `product` (`name`, price, description, category_id) VALUES ('organic meat  
patties', 2.29, 'rolled in fresh herbs<br>2 patties (250g)', 2);  
INSERT INTO `product` (`name`, price, description, category_id) VALUES ('parma ham',  
3.49, 'matured, organic (70g)', 2);  
INSERT INTO `product` (`name`, price, description, category_id) VALUES ('chicken  
leg', 2.59, 'free range (250g)', 2);  
INSERT INTO `product` (`name`, price, description, category_id) VALUES ('sausages',  
3.55, 'reduced fat, pork<br>3 sausages (350g)', 2);  
  
INSERT INTO `product` (`name`, price, description, category_id) VALUES ('sunflower  
seed loaf', 1.89, '600g', 3);
```

```

INSERT INTO `product` (`name`, price, description, category_id) VALUES ('sesame seed
bagel', 1.19, '4 bagels', 3);
INSERT INTO `product` (`name`, price, description, category_id) VALUES ('pumpkin seed
bun', 1.15, '4 buns', 3);
INSERT INTO `product` (`name`, price, description, category_id) VALUES ('chocolate
cookies', 2.39, 'contain peanuts<br>(3 cookies)', 3);

INSERT INTO `product` (`name`, price, description, category_id) VALUES ('corn on the
cob', 1.59, '2 pieces', 4);
INSERT INTO `product` (`name`, price, description, category_id) VALUES ('red
currants', 2.49, '150g', 4);
INSERT INTO `product` (`name`, price, description, category_id) VALUES ('broccoli',
1.29, '500g', 4);
INSERT INTO `product` (`name`, price, description, category_id) VALUES ('seedless
watermelon', 1.49, '250g', 4);

```

Examine the above code and note the following points:

- By examining the affablebean [schema generation script](#), you'll note that the `product` table contains a non-nullable, automatically incremental primary key. Whenever you insert a new record into the table (and don't explicitly set the value of the primary key), the SQL engine sets it for you. Also, note that the `product` table's `last_update` column applies `CURRENT_TIMESTAMP` as its default value. The SQL engine will therefore provide the current date and time for this field when a record is created.

Looking at this another way, if you were to create an `INSERT` statement that didn't indicate which columns would be affected by the insertion action, you would need to account for all columns. In this case, you could enter a `NULL` value to enable the SQL engine to automatically handle fields that have default values specified. For example, the following statement elicits the same result as the first line of the above code:

```
INSERT INTO `product` VALUES (NULL, 'milk', 1.70, 'semi skimmed (1L)', NULL, 1);
```

After running the statement, you'll see that the record contains an automatically incremented primary key, and the `last_update` column lists the current date and time.

- The value for the final column, '`category_id`', must correspond to a value contained in the `category` table's `id` column. Because you have already added four records to the `category` table, the `product` records you are inserting reference one of these four records. If you try to insert a `product` record that references a `category_id` that doesn't exist, a foreign key constraint fails.

3. Click the Run SQL () button in the editor's toolbar.

Note: View the Output window (Ctrl-4; ⌘-4 on Mac) to see a log file containing results of the execution.

4. Right-click the `product` table () node and choose View Data. You can see 16 new records listed in the table.

SQL Command 1

Connection: jdbc:mysql://localhost:3306/affablebean...


```
1 select * from product;
```


select * from product

Page Size: 20 Total Rows: 16 Page: 1 of 1





#	id	name	price	description	last_update	category_id
1	1	milk	1.70	semi skimmed (1L)	2010-06-21 19:25:53.0	1
2	2	cheese	2.39	mild cheddar (330g)	2010-06-21 19:25:53.0	1
3	3	butter	1.09	unsalted (250g)	2010-06-21 19:25:53.0	1
4	4	free range eggs	1.76	medium-sized (6 eggs)	2010-06-21 19:25:54.0	1
5	5	organic meat patties	2.29	rolled in fresh herbs 2 patties (250g)	2010-06-21 19:25:54.0	2
6	6	parma ham	3.49	matured, organic (70g)	2010-06-21 19:25:54.0	2
7	7	chicken leg	2.59	free range (250g)	2010-06-21 19:25:54.0	2
8	8	sausages	3.55	reduced fat, pork 3 sausages (350g)	2010-06-21 19:25:54.0	2
9	9	sunflower seed loaf	1.89	600g	2010-06-21 19:25:54.0	3
10	10	sesame seed bagel	1.19	4 bagels	2010-06-21 19:25:54.0	3
11	11	pumpkin seed bun	1.15	4 buns	2010-06-21 19:25:54.0	3
12	12	chocolate cookies	2.39	contain peanuts (3 cookies)	2010-06-21 19:25:54.0	3
13	13	corn on the cob	1.59	2 pieces	2010-06-21 19:25:54.0	4
14	14	red currants	2.49	150g	2010-06-21 19:25:54.0	4
15	15	broccoli	1.29	500g	2010-06-21 19:25:54.0	4
16	16	seedless watermelon	1.49	250g	2010-06-21 19:25:54.0	4


NetBeans GUI Support for Database Tables

In the Services window, when you right-click a table () node and choose View Data, the IDE displays a visual representation of the table and the data it contains (as depicted in the image above). You can also use this GUI support to add, modify, and delete table data.

- **Add new records:** To add new records, click the Insert Record () button. An Insert Records dialog window displays, enabling you to enter new records. When you click OK, the new data is committed to the database, and the GUI representation of the table is automatically updated.

Click the Show SQL button within the dialog window to view the SQL statement(s) that will be applied upon initiating the action.

- **Modify records:** You can make edits to existing records by double-clicking directly in table cells and modifying field entries. Modified entries display as **green text**. When you are finished editing data, click the Commit Record () button to commit changes to the actual database. (Similarly, click the Cancel Edits () button to cancel any edits you have made.
- **Delete individual records:** Click a row in the table, then click the Delete Selected Record () button. You can also delete multiple rows simultaneously by holding Ctrl (⌘ on Mac) while clicking to select rows.
- **Delete all records:** Deleting all records within a table is referred to as 'truncating' the table. Click the Truncate Table () button to delete all records contained in the displayed table.

If the displayed data needs to be resynchronized with the actual database, you can click the Refresh Records () button. Note that much of the above-described functionality can also be accessed from the right-click menu within the GUI editor.

Creating a Connection Pool and Data Source

From this point onward, you establish connectivity between the MySQL database and the `affablebean` application through the GlassFish server which it is deployed to. This communication is made possible with the Java Database Connectivity (JDBC) API. The JDBC API is an integration library contained in the JDK (refer back to the component diagram displayed in the tutorial [Introduction](#)).

Although this tutorial does not work directly with JDBC programming, the application that we are building does utilize the JDBC API whenever communication is required between the SQL and Java languages. For example, you start by creating a *connection pool* on the GlassFish server. In order for the server to communicate directly with the MySQL database, it requires the [Connector/J](#) JDBC driver which converts JDBC calls directly into a MySQL-specific protocol. Later in this tutorial unit, when you apply JSTL's `<sql:query>` tags to query the `affablebean` database, the tags are translated into JDBC Statements.


A connection pool contains a group of reusable connections for a particular database. Because creating each new physical connection is time-consuming, the server maintains a pool of available connections to increase performance. When an application requests a connection, it obtains one from the pool. When an application closes a connection, the connection is returned to the pool. Connection pools use a JDBC driver to create physical database connections.

A data source (a.k.a. a JDBC resource) provides applications with the means of connecting to a database. Applications get a database connection from a connection pool by looking up a data source using the Java Naming and Directory Interface ([JNDI](#)) and then requesting a connection. The connection pool associated with the data source provides the connection for the application.

In order to enable the application access to the `affablebean` database, you need to create a connection pool and a data source that uses the connection pool. Use the NetBeans GlassFish JDBC Resource wizard to accomplish this.

Note: You can also create connection pools and data sources directly on the GlassFish server using the GlassFish Administration Console. However, creating these resources in this manner requires that you manually enter database connection details (i.e., username, password and URL). The benefit of using the NetBeans wizard is that it extracts any connection details directly from an existing database connection, thus eliminating potential connectivity problems.

To access the console from the IDE, in the Services window right-click the Servers > GlassFish node and choose View Admin Console. The default username / password is: `admin / adminadmin`. If you'd like to set up the connection pool and data source using the GlassFish Administration console, follow steps 3-15 of the [NetBeans E-commerce Tutorial Setup Instructions](#). The setup instructions are provided for later tutorial units.

1. Click the New File () button in the IDE's toolbar. (Alternatively, press Ctrl-N; ⌘-N on Mac.)
2. Select the **GlassFish** category, then select **JDBC Resource** and click Next.
3. In Step 2 of the JDBC Resource wizard, select the **Create New JDBC Connection Pool** option. When you do so, three new steps are added to the wizard, enabling you to specify connection pool settings.
4. Enter details to set up the data source:
 - **JNDI Name:** `jdbc/affablebean`
By convention, the JNDI name for a JDBC resource begins with the `'jdbc/'` string.
 - **Object Type:** `user`
 - **Enabled:** `true`

Steps

1. Choose ...
2. **General Attributes - JDBC Resource**
3. Properties
4. Choose Database Connection
5. Add Connection Pool Properties
6. Add Connection Pool Optional Properties

General Attributes

Provide configuration information for the JDBC Resource.
Either choose an existing JDBC Connection Pool, or create a new JDBC Connection Pool.
Fields with an * mark are required.

☐ Use Existing JDBC Connection Pool

< Select from the list >

☒ Create New JDBC Connection Pool

JNDI Name:* jdbc/affablebean

Object Type: user

Enabled: true

Description:

Help < Back Next > Finish Cancel

5. Click Next. In Step 3, Additional Properties, you do not need to specify any additional configuration information for the data source.
6. Click Next. In Step 4, Choose Database Connection, type in AffableBeanPool as the JDBC connection pool name. Also, ensure that the Extract from Existing Connection option is selected, and that the jdbc:mysql://localhost:3306/affablebean connection is listed.
7. Click Next. In Step 5, Add Connection Pool Properties, specify the following details:
 - **Datasource Classname:** com.mysql.jdbc.jdbc2.optional.MysqlDataSource
 - **Resource Type:** javax.sql.ConnectionPoolDataSource
 - **Description:** (Optional) Connects to the affablebean database

Also, note that the wizard extracts and displays properties from the existing connection.

Steps

1. Choose ...
2. General Attributes – JDBC Resource
3. Properties
4. Choose Database Connection
5. **Add Connection Pool Properties**
6. Add Connection Pool Optional Properties

Add Connection Pool Properties

Enter the Datasource Classname, URL, and User to continue.
Hit the Enter key to save values in the Properties table.

Datasource Classname:

Resource Type:

Description:

Properties:

Name	Value
URL	jdbc:mysql://localhost:3306/affablebean
User	root
Password	nbusser

Add Remove

Help < Back Next > Finish Cancel

8. Click Finish. The wizard generates a `sun-resources.xml` file for the project that contains all information required to set up the connection pool and data source on GlassFish. The `sun-resources.xml` file is a deployment descriptor specific to the GlassFish application server. When the project next gets deployed, the server will read in any configuration data contained in `sun-resources.xml`, and set up the connection pool and data source accordingly. Note that once the connection pool and data source exist on the server, your project no longer requires the `sun-resources.xml` file.
9. In the Projects window (Ctrl-1; ⌘-1 on Mac), expand the Server Resources node and double-click the `sun-resources.xml` file to open it in the editor. Here you see the XML configuration required to set up the connection pool and data source. (Code below is formatted for readability.)

```
<resources>
  <jdbc-resource enabled="true"
    jndi-name="jdbc/affablebean"
    object-type="user"
    pool-name="AffableBeanPool">

    <jdbc-connection-pool allow-non-component-callers="false"
      associate-with-thread="false"
      connection-creation-retry-attempts="0"
      connection-creation-retry-interval-in-seconds="10"
      connection-leak-reclaim="false"
      connection-leak-timeout-in-seconds="0"
      connection-validation-method="auto-commit"
      datasource-
classname="com.mysql.jdbc.jdbc2.optional.MysqlDataSource"
      fail-all-connections="false"
      idle-timeout-in-seconds="300"
      is-connection-validation-required="false"
      is-isolation-level-guaranteed="true"
      lazy-connection-association="false"
      lazy-connection-enlistment="false"
      match-connections="false"
      max-connection-usage-count="0"
      max-pool-size="32"
```



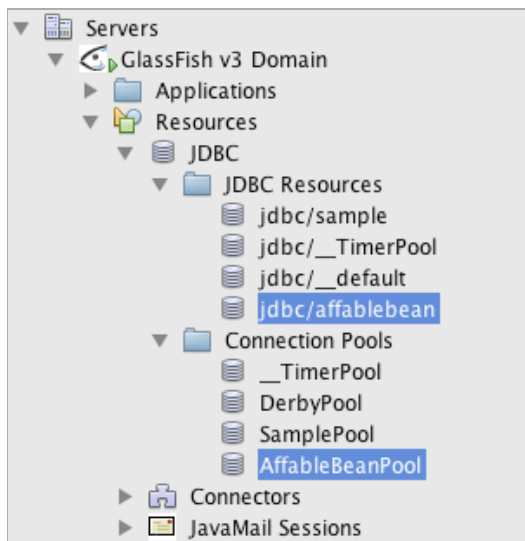
```

        max-wait-time-in-millis="60000"
        name="AffableBeanPool"
        non-transactional-connections="false"
        pool-resize-quantity="2"
        res-type="javax.sql.ConnectionPoolDataSource"
        statement-timeout-in-seconds="-1"
        steady-pool-size="8"
        validate-atmost-once-period-in-seconds="0"
        wrap-jdbc-objects="false">

        <description>Connects to the affablebean database</description>
        <property name="URL" value="jdbc:mysql://localhost:3306/affablebean"/>
        <property name="User" value="root"/>
        <property name="Password" value="nuser"/>
    </jdbc-connection-pool>
</resources>

```

10. In the Projects window (Ctrl-1; ⌘-1 on Mac), right-click the AffableBean project node and choose Deploy. The GlassFish server reads configuration data from the `sun-resources.xml` file and creates the `AffableBeanPool` connection pool, and `jdbc/affablebean` data source.
11. In the Services window, expand the Servers > GlassFish > Resources > JDBC node. Here you can locate the `jdbc/affablebean` data source listed under JDBC Resources, and the `AffableBeanPool` connection pool listed under Connection Pools.



Right-click data source and connection pool nodes to view and make changes to their properties. You can associate a data source with any connection pool registered on the server. You can edit property values for connection pools, and unregister both data sources and connection pools from the server.

Testing the Connection Pool and Data Source

Start by making sure the GlassFish server can successfully connect to the MySQL database. You can do this by pinging the `AffableBeanPool` connection pool in the GlassFish Administration Console.


Then proceed by adding a reference in your project to the data source you created on the server. To do so, you create a `<resource-ref>` entry in the application's `web.xml` deployment descriptor.

Finally, use the IDE's editor support for the `JSTL sql` tag library, and create a JSP page that queries the database and outputs data in a table on a web page.

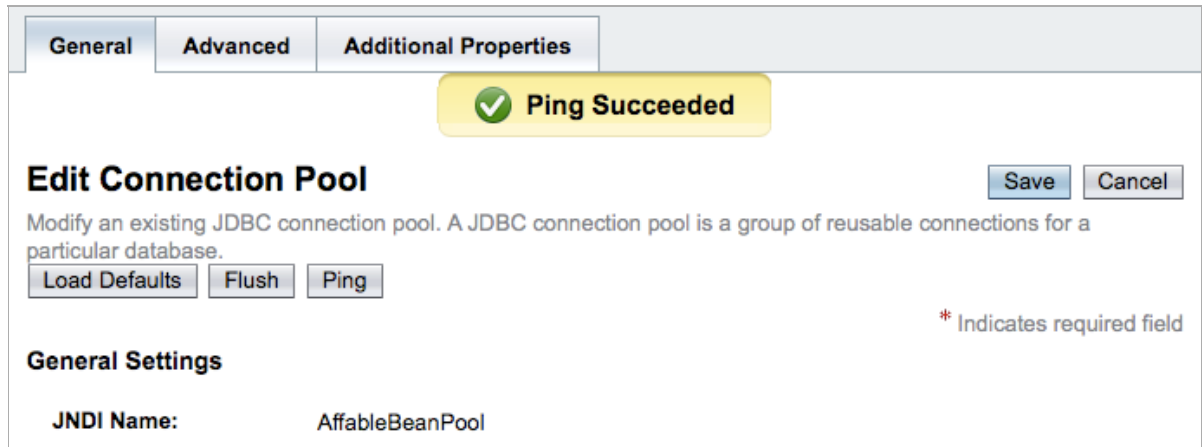
- [Pinging the Connection Pool](#)
- [Creating a Resource Reference to the Data Source](#)

- [Querying the Database from a JSP Page](#)

Pinging the Connection Pool

1. Ensure that the GlassFish server is already running. In the Services window (Ctrl-5; ⌘-5 on Mac), expand the Servers node. Note the small green arrow next to the GlassFish icon ().

(If the server is not running, right-click the server node and choose Start.)
2. Right-click the server node and choose View Admin Console. The GlassFish Administration Console opens in a browser.
3. Log into the administration console. The default username / password is: admin / adminadmin.
4. In the console's tree on the left, expand the Resources > JDBC > Connection Pools nodes, then click AffableBeanPool. In the main window, the Edit Connection Pool interface displays for the selected connection pool.
5. Click the Ping button. If the ping succeeds, the GlassFish server has a working connection to the affablebean database on the MySQL server.

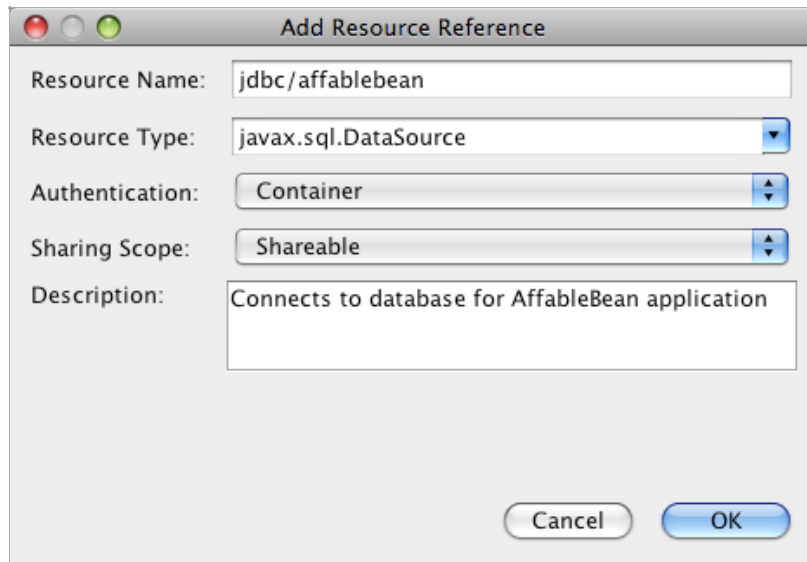


The screenshot shows the 'Edit Connection Pool' window in the GlassFish Administration Console. The 'General' tab is active. A yellow notification banner at the top indicates 'Ping Succeeded'. The window title is 'Edit Connection Pool' with 'Save' and 'Cancel' buttons. Below the title, a description reads: 'Modify an existing JDBC connection pool. A JDBC connection pool is a group of reusable connections for a particular database.' There are three buttons: 'Load Defaults', 'Flush', and 'Ping'. A red asterisk note on the right says '* Indicates required field'. Under the 'General Settings' section, the 'JNDI Name' is set to 'AffableBeanPool'.

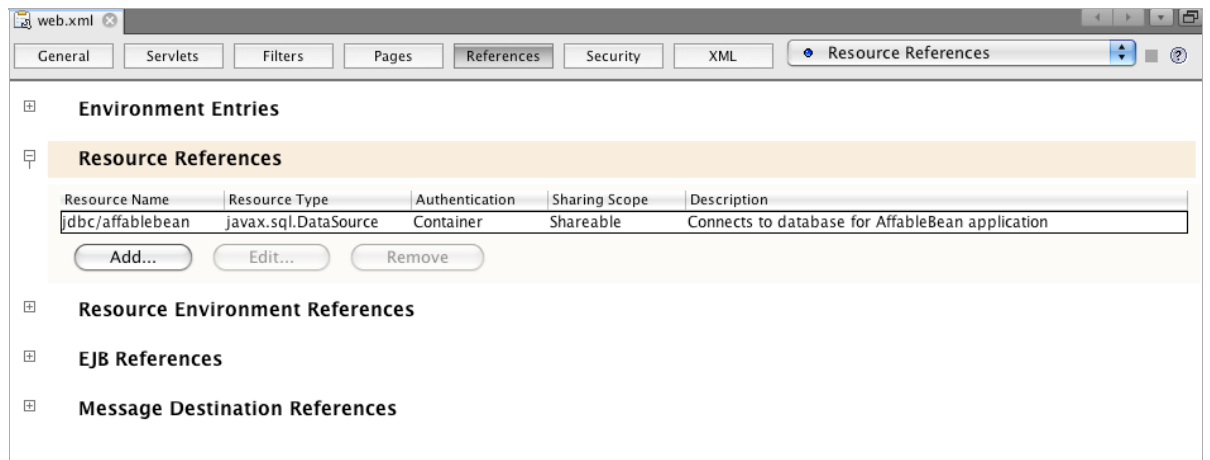
(If the ping fails, see suggestions in the [Troubleshooting](#) section below.)

Creating a Resource Reference to the Data Source

1. In the Projects window, expand the Configuration Files folder and double-click web.xml. A graphical interface for the file displays in the IDE's main window.
2. Click the References tab located along the top of the editor. Expand the Resource References heading, then click Add. The Add Resource Reference dialog opens.
3. Enter the following details into the dialog:
 - **Resource Name:** jdbc/affablebean
 - **Resource Type:** javax.sql.ConnectionPoolDataSource
 - **Authentication:** Container
 - **Sharing Scope:** Shareable
 - **Description:** (Optional) Connects to database for AffableBean application




- Click OK. The new resource is added under the Resource References heading.

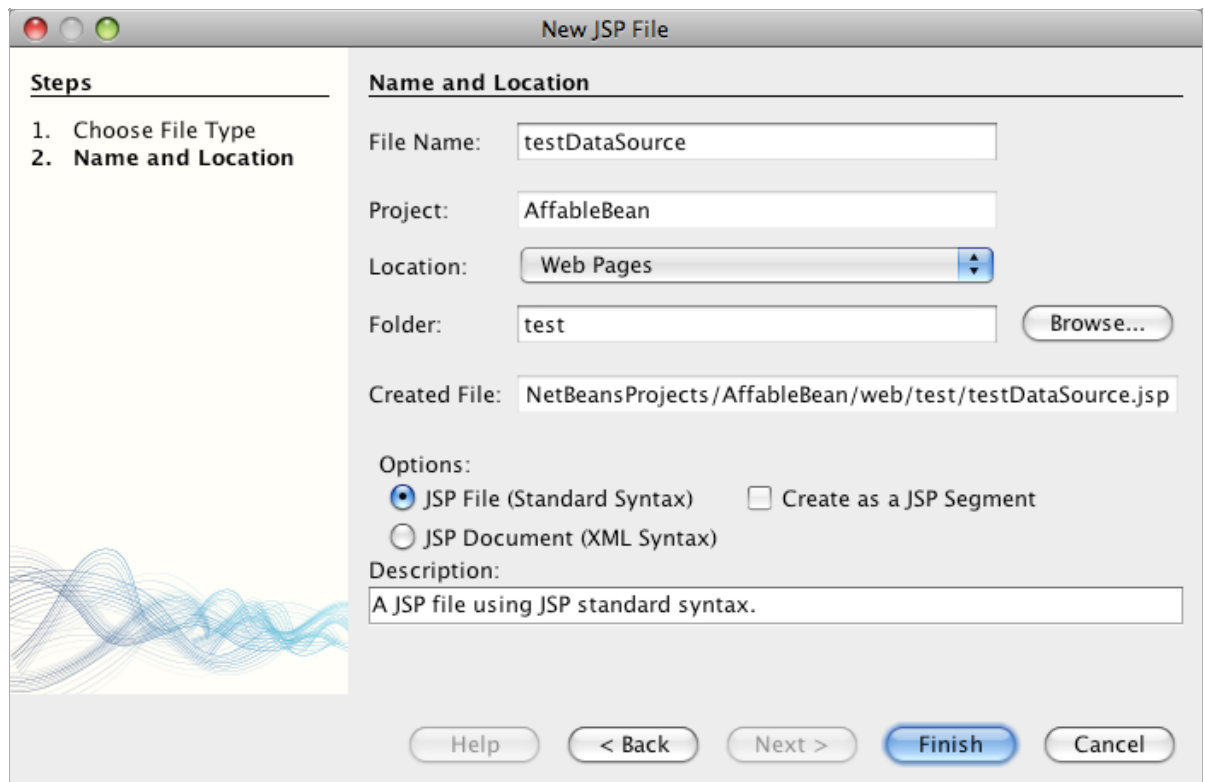


To verify that the resource is now added to the web.xml file, click the XML tab located along the top of the editor. Notice that the following `<resource-ref>` tags are now included:

```
<resource-ref>
  <description>Connects to database for AffableBean application</description>
  <res-ref-name>jdbc/affablebean</res-ref-name>
  <res-type>javax.sql.ConnectionPoolDataSource</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
```

Querying the Database from a JSP Page

- Create a new JSP page to test the data source. Click the New File () button. (Alternatively, press Ctrl-N; ⌘-N on Mac.)
- Select the Web category, then select the JSP file type and click Next.
- Enter 'testDataSource' as the file name. In the Folder field, type in 'test'.



New JSP File

Steps

1. Choose File Type
2. Name and Location

Name and Location

File Name:

Project:

Location:

Folder:

Created File:

Options:

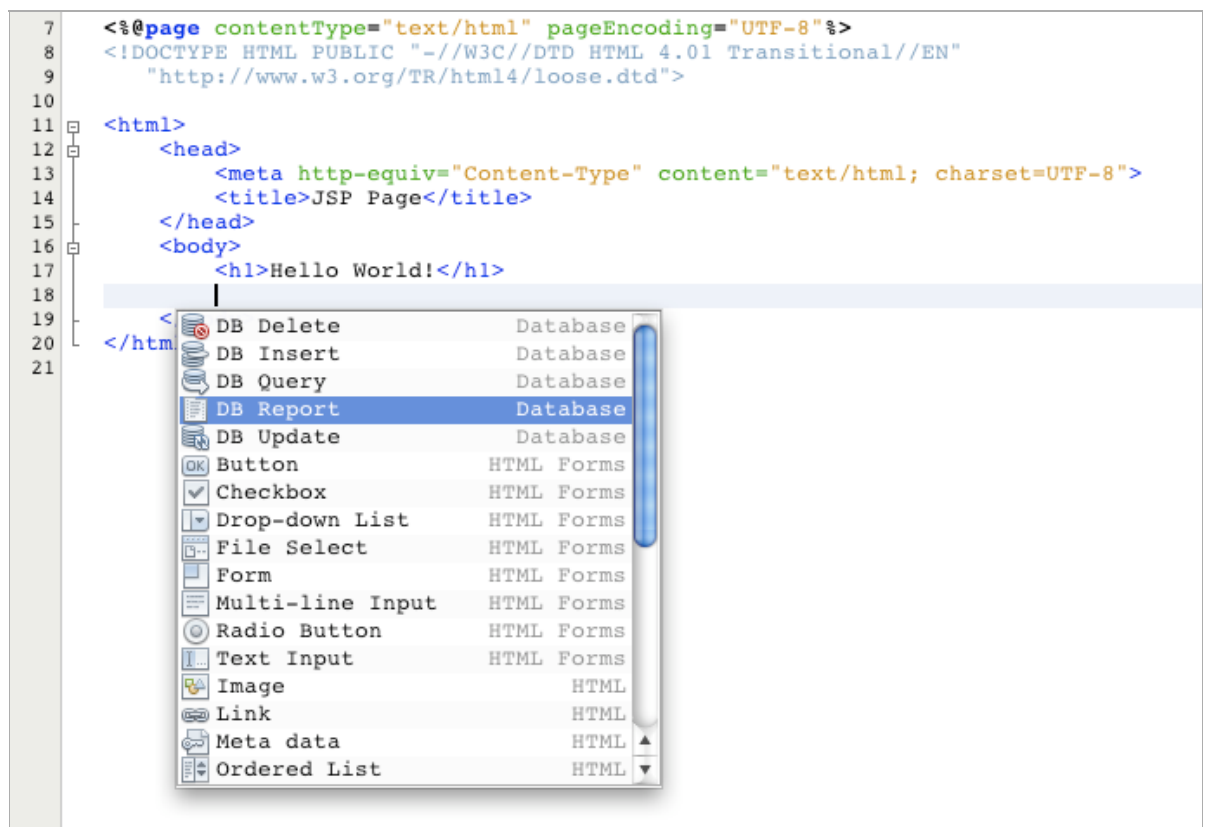
☒ JSP File (Standard Syntax) ☐ Create as a JSP Segment

☐ JSP Document (XML Syntax)

Description:

The project does not yet have a folder named 'test' within the Web Pages location (i.e., within the web folder). By entering 'test' into the Folder field, you have the IDE create the folder upon completing the wizard.

4. Click finish. The IDE generates a new `testDataSource.jsp` file, and places it into the new `test` folder within the project.
5. In the new `testDataSource.jsp` file, in the editor, place your cursor at the end of the line containing the `<h1>` tags (line 17). Press Return, then press Ctrl-Space to invoke code suggestions. Choose DB Report from the list of options.



```

7  <%@page contentType="text/html" pageEncoding="UTF-8"%>
8  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
9    "http://www.w3.org/TR/html4/loose.dtd">
10
11  <html>
12  <head>
13    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
14    <title>JSP Page</title>
15  </head>
16  <body>
17    <h1>Hello World!</h1>
18
19  <
20  </html>
21

```

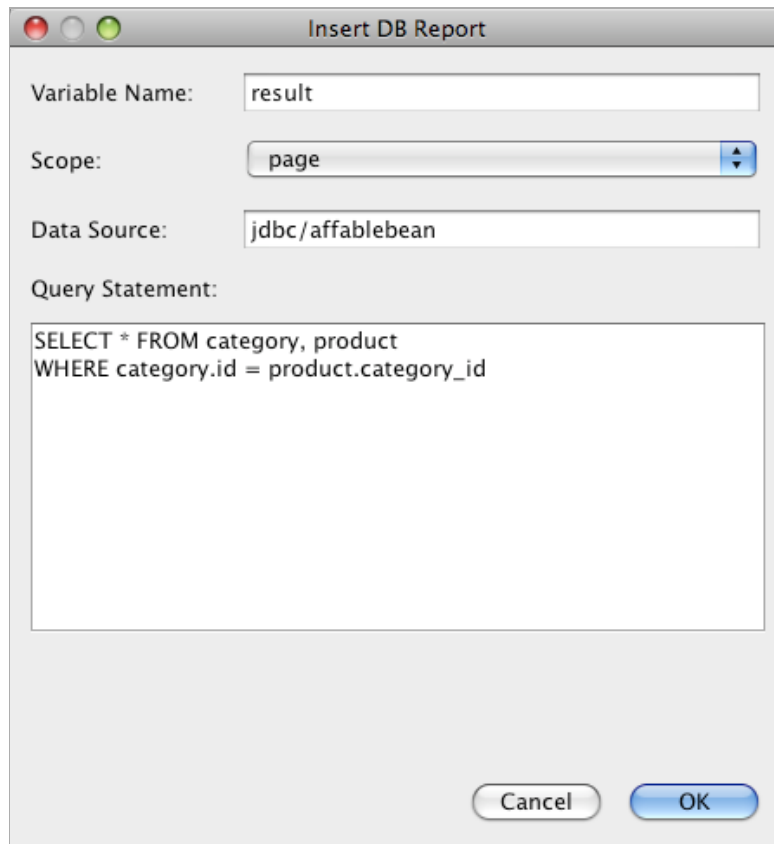
Code completion menu:

DB Delete	Database
DB Insert	Database
DB Query	Database
DB Report	Database
DB Update	Database
OK Button	HTML Forms
Checkbox	HTML Forms
Drop-down List	HTML Forms
File Select	HTML Forms
Form	HTML Forms
Multi-line Input	HTML Forms
Radio Button	HTML Forms
Text Input	HTML Forms
Image	HTML
Link	HTML
Meta data	HTML
Ordered List	HTML

If line numbers do not display, right-click in the left margin of the editor and choose Show Line Numbers.

6. In the Insert DB Report dialog, specify the data source and modify the SQL query to be executed:

- **Data Source:** jdbc/affablebean
- **Query Statement:** `SELECT * FROM category, product WHERE category.id = product.category_id`



7. Click OK. The dialog adds the taglib directives for the JSTL core and sql libraries to the top of the file:

```
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql"%>
```

The dialog also generates template code to display the query results in an HTML table:

```
<sql:query var="result" dataSource="jdbc/affablebean">
    SELECT * FROM category, product
    WHERE category.id = product.category_id
</sql:query>

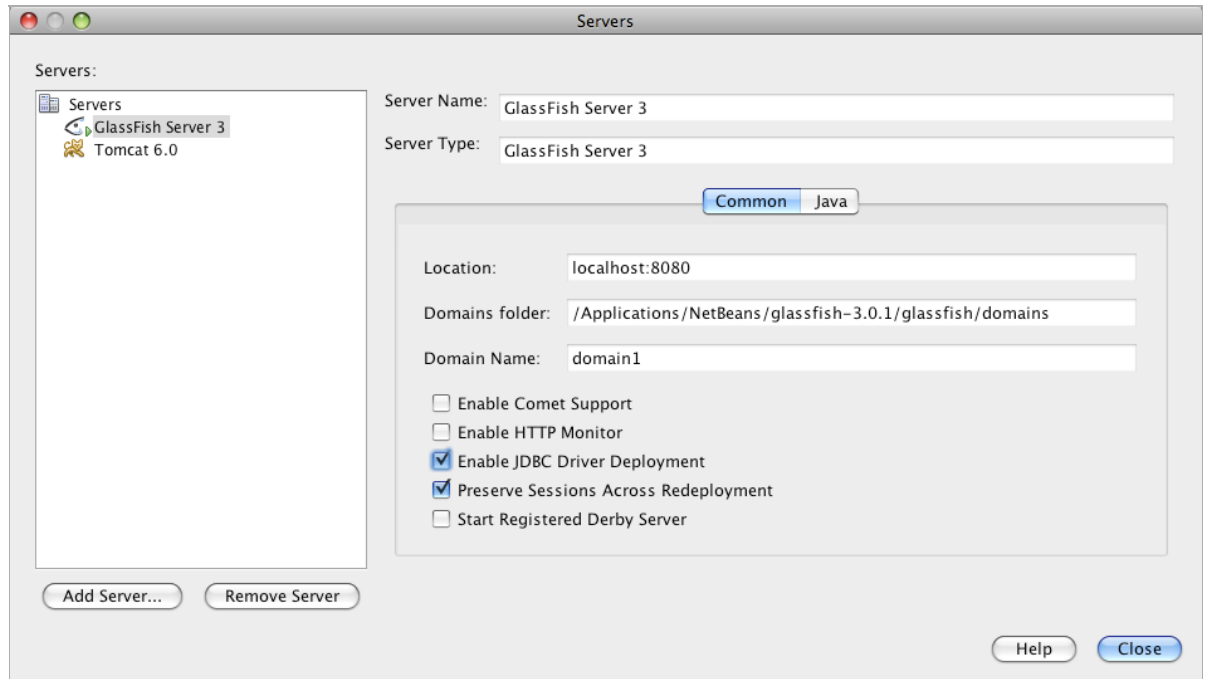
<table border="1">
    <!-- column headers -->
    <tr>
        <c:forEach var="columnName" items="${result.columnNames}">
            <th><c:out value="${columnName}" /></th>
        </c:forEach>
    </tr>
    <!-- column data -->
    <c:forEach var="row" items="${result.rowsByIndex}">
        <tr>
            <c:forEach var="column" items="${row}">
                <td><c:out value="${column}" /></td>
            </c:forEach>
        </tr>
    </c:forEach>
</table>
```

```

        </c:forEach>
    </tr>
</c:forEach>
</table>

```

- Before running the file in a browser, make sure you have enabled the JDBC driver deployment option in NetBeans' GlassFish support. Choose Tools > Servers to open the Servers window. In the left column, select the GlassFish server you are deploying to. In the main column, ensure that the 'Enable JDBC Driver Deployment' option is selected, then click Close.



For Java applications that connect to a database, the server requires a JDBC driver to be able to create a communication bridge between the SQL and Java languages. In the case of MySQL, you use the [Connector/J](#) JDBC driver. Ordinarily you would need to manually place the driver JAR file into the server's `lib` directory. With the 'Enable JDBC Driver Deployment' option selected, the server performs a check to see whether a driver is needed, and if so, the IDE deploys the driver to the server.

- Right-click in the editor and choose Run File (Shift-F6; fn-Shift-F6 on Mac). The `testDataSource.jsp` file is compiled into a servlet, deployed to the server, then runs in a browser.
- Open the Output window (Ctrl-4; ⌘-4 on Mac) and click the 'AffableBean (run)' tab. The output indicates that the driver JAR file (`mysql-connector-java-5.1.6-bin.jar`) is deployed.



- Examine `testDataSource.jsp` in the browser. You see an HTML table listing data contained in the `category` and `product` tables.

The screenshot shows a web browser window with the title 'JSP Page'. The address bar displays 'http://localhost:8080/AffableBean/test/testDataSource.jsp'. The page content features a large 'Hello World!' heading followed by a table of food items. The table has 8 columns: 'id', 'name', 'id', 'name', 'price', 'description', 'last_update', and 'category_id'. It contains 16 rows of data, categorized by food type (dairy, meats, bakery, fruit & veg).

id	name	id	name	price	description	last_update	category_id
1	dairy	1	milk	1.70	semi skimmed (1L)	2010-06-21 19:25:53.0	1
1	dairy	2	cheese	2.39	mild cheddar (330g)	2010-06-21 19:25:53.0	1
1	dairy	3	butter	1.09	unsalted (250g)	2010-06-21 19:25:53.0	1
1	dairy	4	free range eggs	1.76	medium-sized (6 eggs)	2010-06-21 19:25:54.0	1
2	meats	5	organic meat patties	2.29	rolled in fresh herbs 2 patties (250g)	2010-06-21 19:25:54.0	2
2	meats	6	parma ham	3.49	matured, organic (70g)	2010-06-21 19:25:54.0	2
2	meats	7	chicken leg	2.59	free range (250g)	2010-06-21 19:25:54.0	2
2	meats	8	sausages	3.55	reduced fat, pork 3 sausages (350g)	2010-06-21 19:25:54.0	2
3	bakery	9	sunflower seed loaf	1.89	600g	2010-06-21 19:25:54.0	3
3	bakery	10	sesame seed bagel	1.19	4 bagels	2010-06-21 19:25:54.0	3
3	bakery	11	pumpkin seed bun	1.15	4 buns	2010-06-21 19:25:54.0	3
3	bakery	12	chocolate cookies	2.39	contain peanuts (3 cookies)	2010-06-21 19:25:54.0	3
4	fruit & veg	13	corn on the cob	1.59	2 pieces	2010-06-21 19:25:54.0	4
4	fruit & veg	14	red currants	2.49	150g	2010-06-21 19:25:54.0	4
4	fruit & veg	15	broccoli	1.29	500g	2010-06-21 19:25:54.0	4
4	fruit & veg	16	seedless watermelon	1.49	250g	2010-06-21 19:25:54.0	4

(If you receive a server error, see suggestions in the [Troubleshooting](#) section below.)

At this stage, we have set up a working data source and connection pool on the server, and demonstrated that the application can access data contained in the `affablebean` database.

Setting Context Parameters

This section demonstrates how to configure context parameters for the application, and how to access parameter values from JSP pages. The owner of an application may want to be able to change certain settings without the need to make intrusive changes to source code. Context parameters enable you application-wide access to parameter values, and provide a convenient way to change parameter values from a single location, should the need arise.

Setting up context parameters can be accomplished in two steps:

1. Listing parameter names and values in the web deployment descriptor
2. Calling the parameters in JSP pages using the `initParam` object

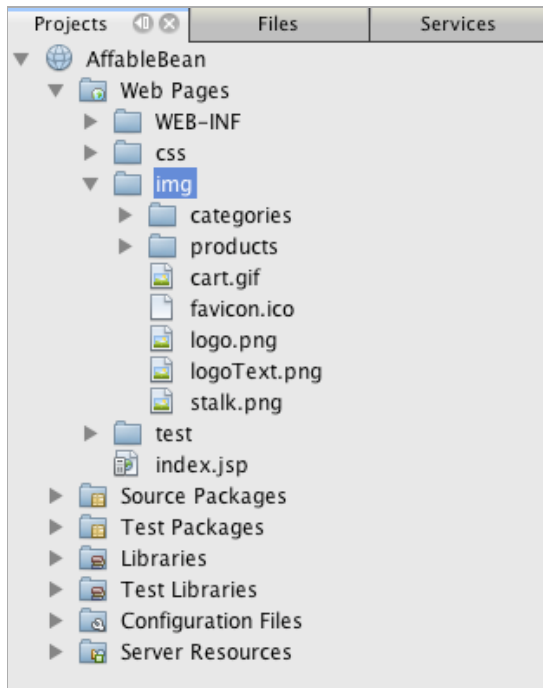
The JSP Expression Language (EL) defines *implicit objects*, which `initParam` is an example of. When working in JSP pages, you can utilize implicit objects using dot notation and placing expressions within EL delimiters (`{ . . }`). For example, if you have an initialization parameter named `myParam`, you can access it from a JSP page with the expression `{initParam.myParam}`.

For more information on the JSP Expression Language and implicit objects, see the following chapter in the Java EE 6 Tutorial:
[Chapter 6 - Expression Language](#).

By way of demonstration, you create context parameters for the image paths to category and product images used in the `AffableBean` project. Begin by adding the provided image resources to the project, then perform the two steps outlined above.

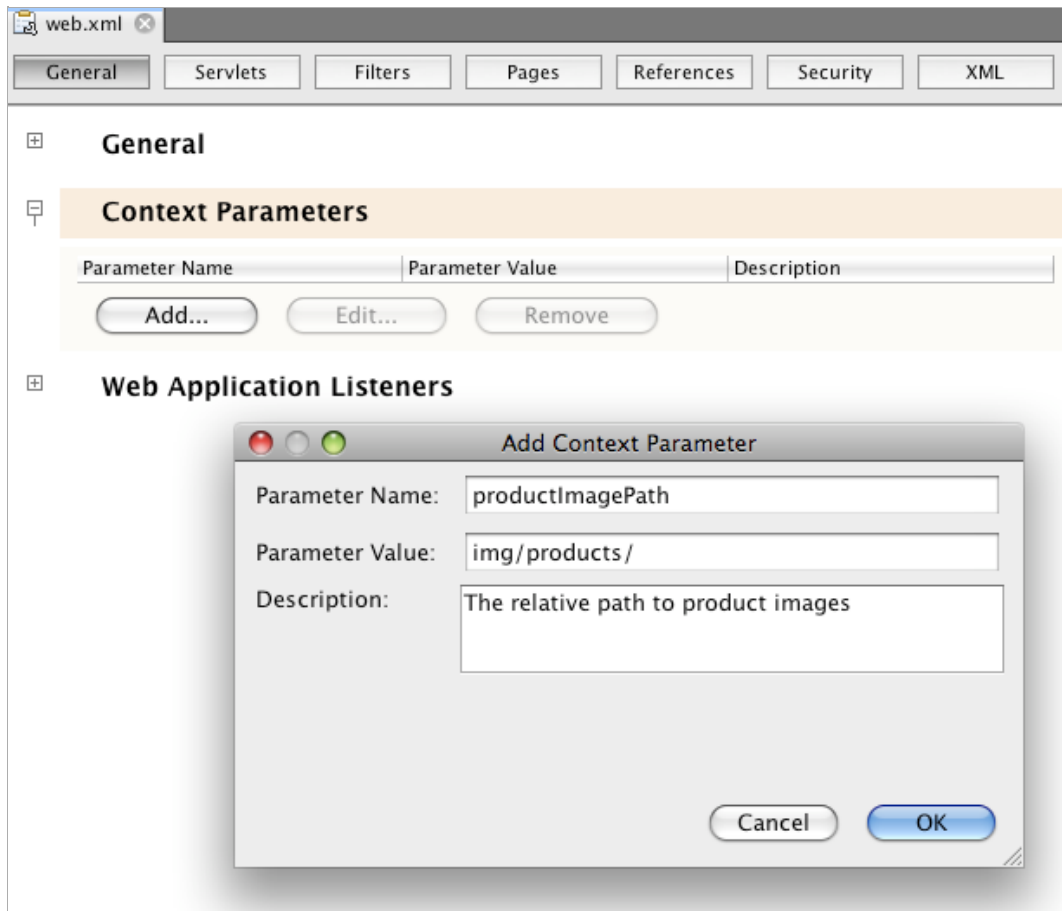
1. Download the [website sample images](#), and unzip the file to a location on your computer. The unzipped file is an `img` folder that contains all of the image resources required for the `AffableBean` application.
2. Import the `img` folder into the `AffableBean` project. Copy (Ctrl-C; ⌘-C on Mac) the `img` folder, then in the IDE's Projects

window, paste (Ctrl-V; ⌘-V on Mac) the folder into the project's Web Pages node.

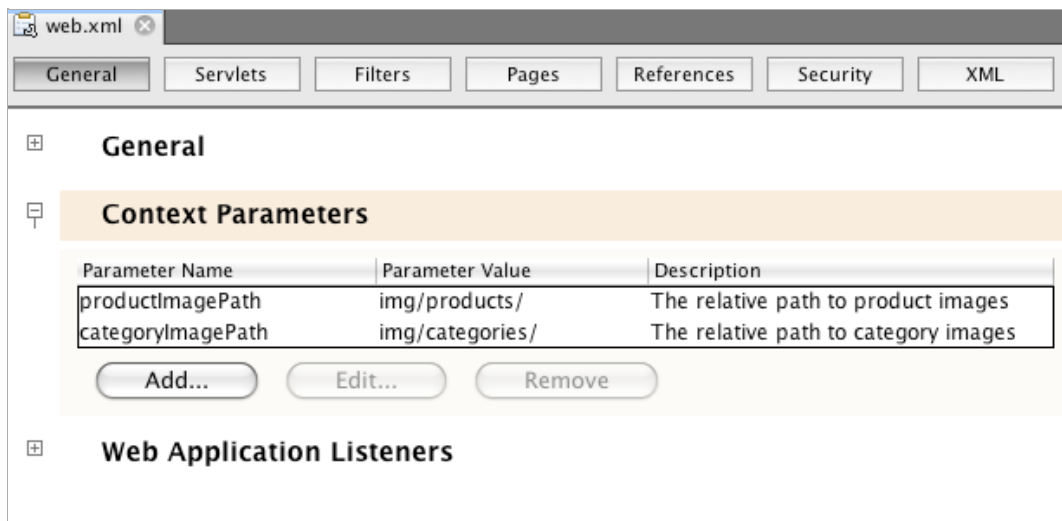


The `categories` and `products` folders contain the images that will be displayed in the `index` and `category` pages, respectively.

3. Open the project's web deployment descriptor. In the Projects window, expand the Configuration Files node and double-click `web.xml`.
4. Click the General tab, then expand Context Parameters and click the Add button.
5. In the Add Context Parameter dialog, enter the following details:
 - **Parameter Name:** `productImagePath`
 - **Parameter Value:** `img/products/`
 - **Description:** *(Optional)* The relative path to product images



6. Click OK.
7. Click the Add button again and enter the following details:
 - **Parameter Name:** categoryImagePath
 - **Parameter Value:** img/categories/
 - **Description:** (Optional) The relative path to category images
8. Click OK. The two context parameters are now listed:



9. Click the XML tab to view the XML content that has been added to the deployment descriptor. The following `<context-param>` entries have been added:

```

<context-param>
  <description>The relative path to product images</description>
  <param-name>productImagePath</param-name>
  <param-value>img/products/</param-value>
</context-param>
<context-param>
  <description>The relative path to category images</description>
  <param-name>categoryImagePath</param-name>
  <param-value>img/categories/</param-value>
</context-param>

```


10. To test whether the values for the context parameters are accessible to web pages, open any of the project's web pages in the editor and enter EL expressions using the `initParam` implicit object. For example, open `index.jsp` and enter the following (New code in **bold**):

```

<div id="indexLeftColumn">
  <div id="welcomeText">
    <p>[ welcome text ]</p>

    <!-- test to access context parameters -->
    categoryImagePath: ${initParam.categoryImagePath}
    productImagePath: ${initParam.productImagePath}
  </div>
</div>

```

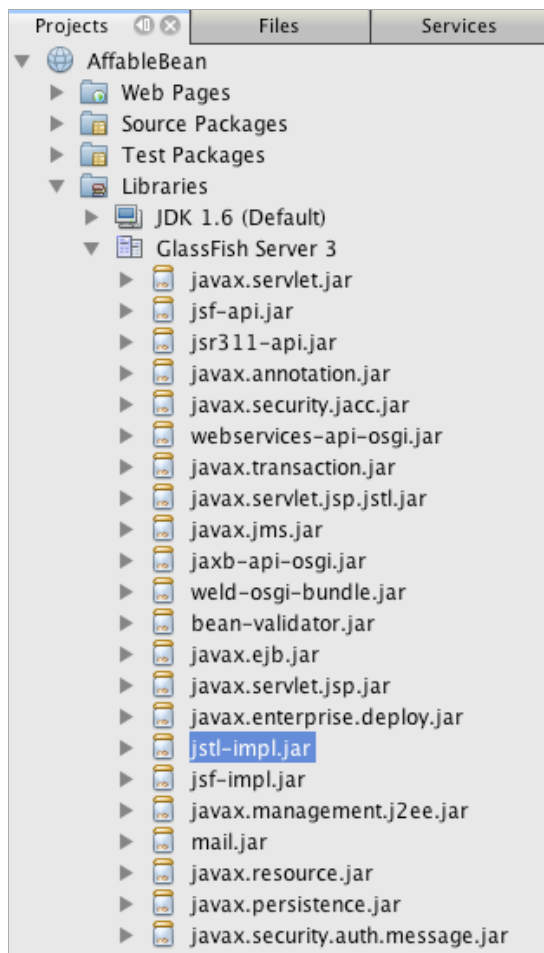
11. Run the project. Click the Run Project () button. The project's index page opens in the browser, and you see the values for the `categoryImagePath` and `productImagePath` context parameters displayed in the page.



Working with JSTL

So far in this tutorial unit, you've established how to access data from the `affablebean` database, add image resources to the project, and have set up several context parameters. In this final section, you combine these achievements to plug the product and category images into the application. In order to do so effectively, you need to begin taking advantage of the JavaServer Pages Standard Tag Library (JSTL).

Note that you do not have to worry about adding the JSTL JAR file (`jstl-impl.jar`) to your project's classpath because it already exists. When you created the `AffableBean` project and selected GlassFish as your development server, the libraries contained in the server were automatically added to your project's classpath. You can verify this in the Projects window by expanding the `AffableBean` project's Libraries > GlassFish Server 3 node to view all of the libraries provided by the server.



The `jstl-impl.jar` file is GlassFish' implementation of JSTL, version 1.2.

You can also download the GlassFish JSTL JAR file separately from: <http://jstl.dev.java.net/download.html>

Before embarking upon an exercise involving JSTL, one implementation detail needs to first be clarified. Examine the files contained in the `categories` and `products` folders and note that the names of the provided image files match the names of the category and product entries found in the database. This enables us to leverage the database data to dynamically call image files within the page. So for example, if the web page needs to access the image for the broccoli product entry, you can make this happen using the following statement.

```
${initParam.productImagePath}broccoli.png
```

After implementing a JSTL `forEach` loop, you'll be able to replace the hard-coded name of the product with an EL expression that dynamically extracts the name of the product from the database, and inserts it into the page.

```
${initParam.productImagePath}${product.name}.png
```

Begin by integrating the category images into the index page, then work within the category page so that data pertaining to the selected category is dynamically handled.

- [index page](#)
- [category page](#)

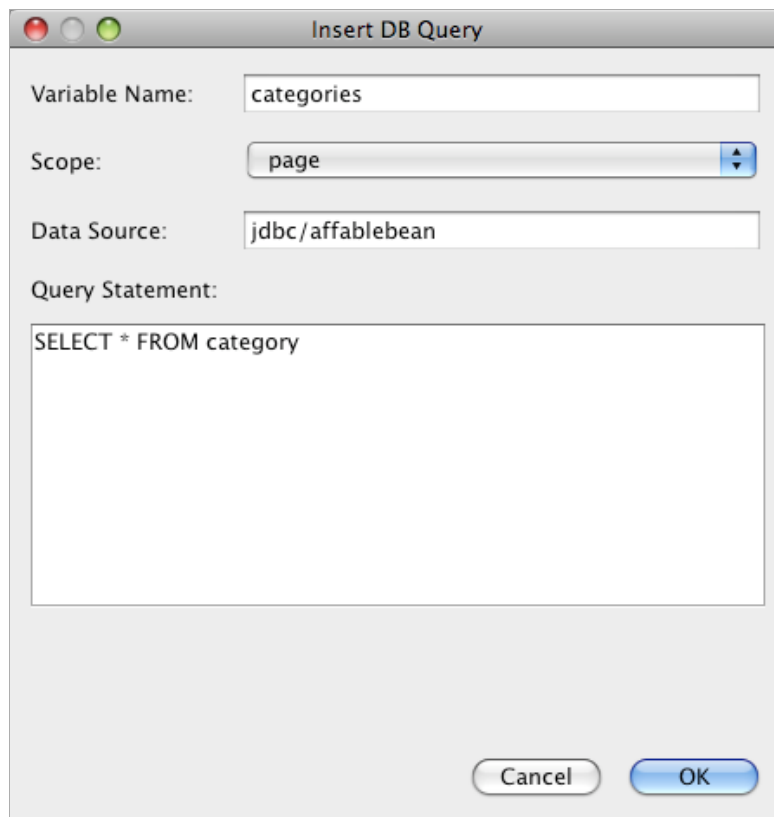
[index page](#)

1. In the Projects window, double-click the `index.jsp` node to open it in the editor. (If already opened, press Ctrl-Tab to select it in the editor.)
2. At the top of the file, before the first `<div>` tag, place your cursor on a blank line, then type 'db' and press Ctrl-Space. In the code-completion pop-up window that displays, choose DB Query.



The screenshot shows a code editor with a JSP file. A code completion pop-up is visible for the text 'db'. The pop-up lists several options: 'DB Delete Database', 'DB Insert Database', 'DB Query Database' (which is highlighted), 'DB Report Database', and 'DB Update Database'. The background code includes a JSP header with document information and a `<div>` tag containing a `<p>` tag with the text 'welcome text'.

3. In the Insert DB Query dialog, enter the following details:
- **Variable Name:** categories
 - **Scope:** page
 - **Data Source:** jdbc/affablebean
 - **Query Statement:** SELECT * FROM category



The screenshot shows the 'Insert DB Query' dialog box. It has four main fields: 'Variable Name' with the value 'categories', 'Scope' with a dropdown menu set to 'page', 'Data Source' with the value 'jdbc/affablebean', and 'Query Statement' with the text 'SELECT * FROM category'. At the bottom, there are 'Cancel' and 'OK' buttons.

4. Click OK. The dialog generates an SQL query using JSTL `<sql:query>` tags. Also, note that the required reference to the `sql taglib` directive has been automatically inserted at the top of the page. (Changes displayed in **bold**.)

```
<%@taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql"%>
<%--
```

```

Document      : index
Created on    : Sep 5, 2009, 4:32:42 PM
Author       : nbuser
--%>

<sql:query var="categories" dataSource="jdbc/affablebean">
    SELECT * FROM category
</sql:query>

```

```

<div id="indexLeftColumn">
    <div id="welcomeText">
        <p>[ welcome text ]</p>

```

The SQL query creates a result set which is stored in the `categories` variable. You can then access the result set using EL syntax, e.g., `${categories}` (demonstrated below).

- Place your cursor at the end of '`<div id="indexRightColumn">`' (line 22), hit return, type 'jstl' then press Ctrl-Space and choose JSTL For Each.

```

8  <sql:query var="categories" dataSource="jdbc/affablebean">
9      SELECT * FROM category
10 </sql:query>
11
12     <div id="indexLeftColumn">
13         <div id="welcomeText">
14             <p>[ welcome text ]</p>
15
16             <!-- test to access context parameters -->
17             categoryImagePath: ${initParam.categoryImagePath}
18             productImagePath: ${initParam.productImagePath}
19
20         </div>
21     </div>
22     <div id="indexRightColumn">
23         jstl
24         <div id="categoryBox">
25             <div id="categoryLabelText">dairy</span>
26             <div class="categoryBox">
27                 <a href="#">
28
29
30

```

- In the Insert JSTL For Each dialog, enter the following details:

- Collection:** `${categories.rows}`
- Current Item of the Iteration:** `category`

Insert JSTL For Each

Collection:

Current Item of the Iteration:

☐ Fixed Number of Iterations

Begin:

End:

Step:

7. Click OK. The dialog sets up syntax for a JSTL `forEach` loop using `<c:forEach>` tags. Also, note that the required reference to the `core` taglib directive has been automatically inserted at the top of the page. (Changes displayed in **bold**.)

```
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql"%>
```

...

```
<div id="indexRightColumn">
    <c:forEach var="category" items="categories.rows">
    </c:forEach>
    <div class="categoryBox">
```

If you are wondering what 'rows' refers to in the generated code, recall that the `categories` variable represents a result set. More specifically, `categories` refers to an object that implements the `javax.servlet.jsp.jstl.sql.Result` interface. This object provides properties for accessing the rows, column names, and size of the query's result set. When using dot notation as in the above example, '`categories.rows`' is translated in Java to '`categories.getRows()`'.

8. Integrate the `<c:forEach>` tags into the page. You can nest the `<div class="categoryBox">` tags within the `forEach` loop so that HTML markup is generated for each of the four categories. Use EL syntax to extract the category table's `id` and `name` column values for each of the four records. Make sure to delete the other `<div class="categoryBox">` tags which exist outside the `forEach` loop. When you finish, the complete `index.jsp` file will look as follows. (`<c:forEach>` tags and contents are displayed in **bold**.)

```
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql"%>
<!--
    Document      : index
    Created on    : Sep 5, 2009, 4:32:42 PM
    Author       : nbuser
--%>

<sql:query var="categories" dataSource="jdbc/affablebean">
    SELECT * FROM category
</sql:query>

    <div id="indexLeftColumn">
        <div id="welcomeText">
            <p>[ welcome text ]</p>


            <!-- test to access context parameters -->
            categoryImagePath: ${initParam.categoryImagePath}
            productImagePath: ${initParam.productImagePath}
        </div>
    </div>

    <div id="indexRightColumn">
        <c:forEach var="category" items="${categories.rows}">
            <div class="categoryBox">
                <a href="category?${category.id}">

                    <span class="categoryLabelText">${category.name}</span>

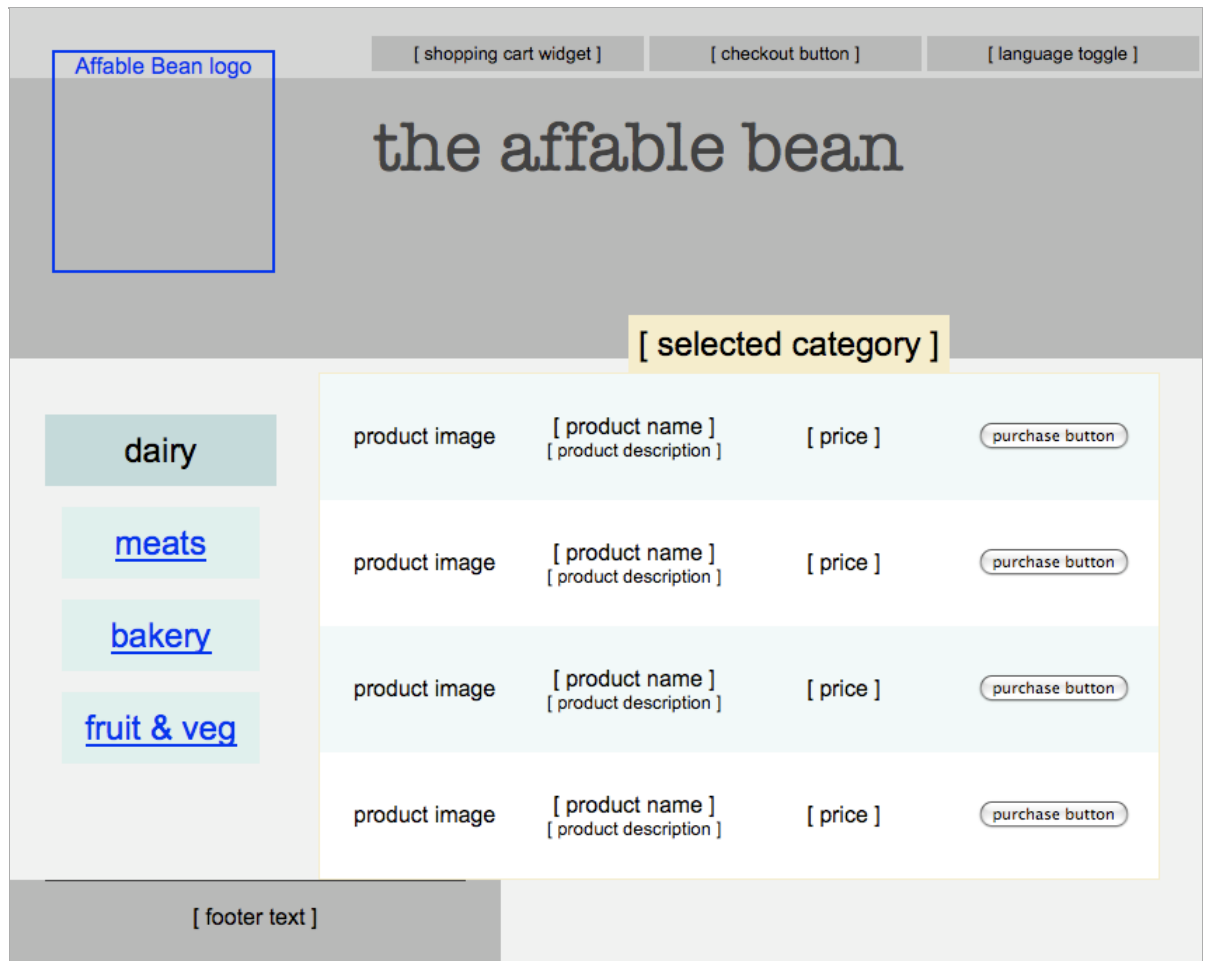
                    
    </a>
</div>
</c:forEach>
</div>

```

- Click the Run Project () button. The project's index page opens in the browser, and you see the names and images of the four categories.



- Click any of the four images in the browser. The category page displays.



To understand how linking takes place between the index and category pages, reexamine the HTML anchor tags within the `forEach` loop:

```
<a href="category?${category.id}">
```

When a user clicks the image link in the browser, a request for 'category' is sent to the application's context root on the server. In your development environment, the URL is as follows:

```
http://localhost:8080/AffableBean/category
```

This URL can be explained in the following manner:

- `http://localhost:8080`: The default location of the GlassFish server on your computer
- `/AffableBean`: The context root of your deployed application
- `/category`: The path to the request

Recall that in [Preparing the Page Views and Controller Servlet](#), you mapped a request for '/category' to the `ControllerServlet`. Currently, the `ControllerServlet` internally forwards the request to `/WEB-INF/view/category.jsp`, which is why the category page displays upon clicking an image link.

You can verify the application's context root by expanding the Configuration Files node in the Projects window, and opening the `sun-web.xml` file. The `sun-web.xml` file is a deployment descriptor specific to GlassFish.

Also, note that a question mark (?) and category ID are appended to the requested URL.

```
<a href="category?${category.id}">
```

This forms the *query string*. As is demonstrated in the next section, you can apply

`(pageContext.request.queryString)` to extract the value of the query string from the request. You can then use the category ID from the query string to determine which category details need to be included in the response.

category page

Three aspects of the category page need to be handled dynamically. The left column must indicate which category is selected, the table heading must display the name of the selected category, and the table must list product details pertaining to the selected category. In order to implement these aspects using JSTL, you can follow a simple, 2-step pattern:

1. Retrieve data from the database using the JSTL `sql` tag library.
2. Display the data using the JSTL `core` library and EL syntax.

Tackle each of the three tasks individually.

Display selected category in left column

1. In the Projects window, double-click the `category.jsp` node to open it in the editor. (If already opened, press Ctrl-Tab to select it in the editor.)
2. Add the following SQL query to the top of the file.

```
<sql:query var="categories" dataSource="jdbc/affablebean">
    SELECT * FROM category
</sql:query>
```

Either use the Insert DB Query dialog as [described above](#), or use the editor's code suggestion and completion facilities by pressing Ctrl-Space while typing.

3. Between the `<div id="categoryLeftColumn">` tags, replace the existing static placeholder content with the following `<c:forEach>` loop.

```
<div id="categoryLeftColumn">

    <c:forEach var="category" items="${categories.rows}">

        <c:choose>
            <c:when test="${category.id == pageContext.request.queryString}">
                <div class="categoryButton" id="selectedCategory">
                    <span class="categoryText">
                        ${category.name}
                    </span>
                </div>
            </c:when>
            <c:otherwise>
                <a href="category?${category.id}" class="categoryButton">
                    <div class="categoryText">
                        ${category.name}
                    </div>
                </a>
            </c:otherwise>
        </c:choose>

    </c:forEach>

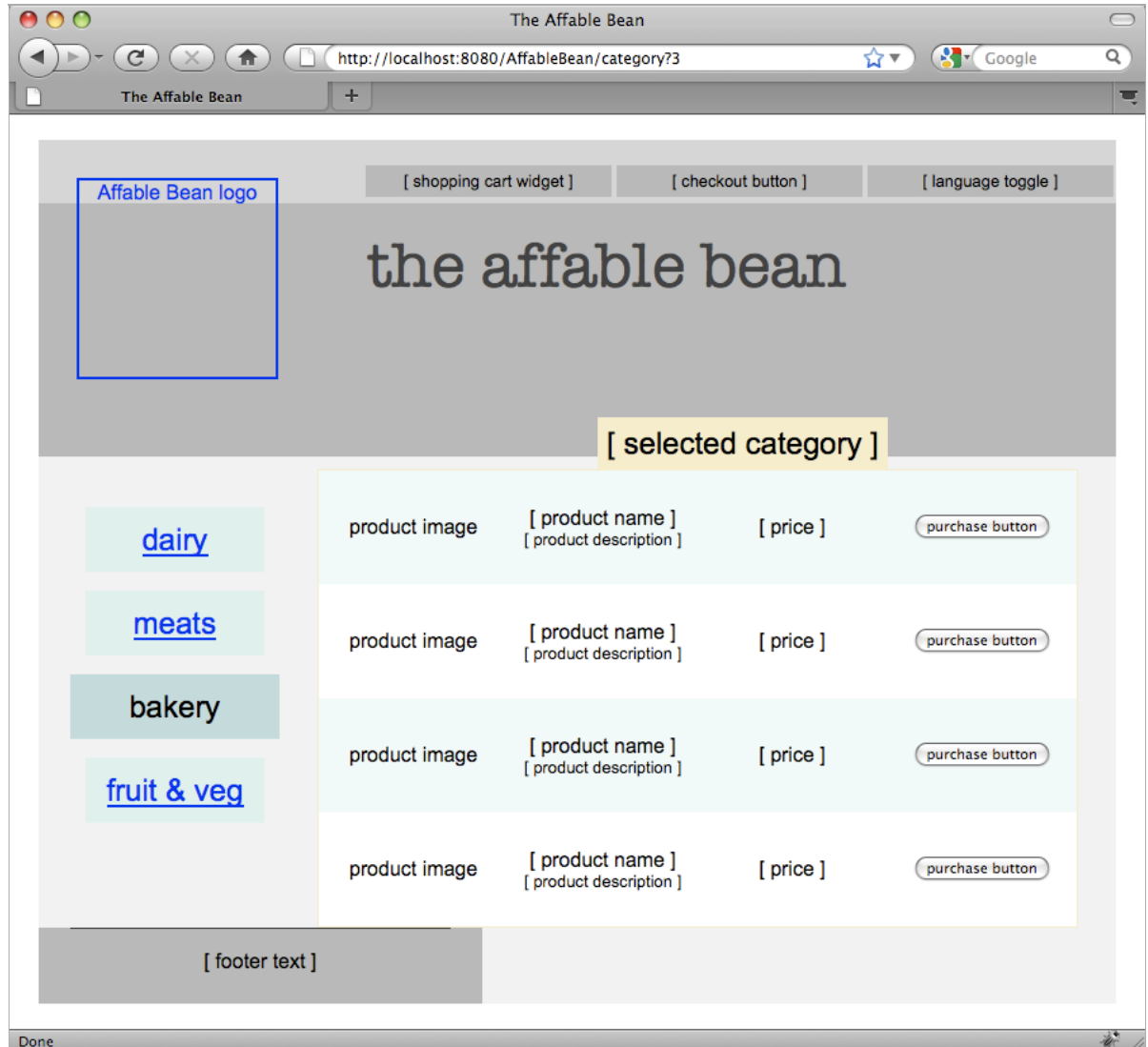
</div>
```

In the above snippet, you access the request's query string using `'pageContext.request.queryString'`. `pageContext` is another [implicit object](#) defined by the JSP Expression Language. The EL expression uses the [PageContext](#) to access the current request (an [HttpServletRequest](#) object). From [HttpServletRequest](#), the `getQueryString()` method is called to obtain the value of the request's query string.

4. Make sure to add the JSTL `core` and `sql` taglib directives to the top of the page. (This is done automatically when using the editor's code suggestion and completion facilities.)

```
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql"%>
```

5. Run the project. In the browser, navigate to the category page and click the category buttons in the left column. Each time you click, the page refreshes highlighting the selected category.



Also, note that the ID of the selected category is displayed in the page's URL. (In the above image, the bakery category is selected, and '3' is appended to the URL in the browser's navigation toolbar.)

Your servlet container (i.e., GlassFish) converts JSP pages into servlets before running them as part of a project. You can view the generated servlet for a JSP page by right-clicking the page node in the Projects window and choosing View Servlet. Of course, you first need to run the project so that the servlet is generated. Taking the `index.jsp` file as an example, when you choose View Servlet, the IDE displays a read-only copy of the generated servlet, `index_jsp.java`, in the editor. The servlet exists on the server at: `<gf-install-dir>/glassfish/domains/domain1/generated/jsp/AffableBean/org/apache/jsp/index_jsp.java`.


Examining Implicit Object Values using the IDE's Debugger




You can use the IDE's Java debugger to examine values for implicit objects. To do so, set a breakpoint on a line containing JSP or JSTL syntax in a JSP page, then run the debugger. When the debugger suspends on the breakpoint, you can open the Variables window (Window > Debugging > Variables) to inspect values currently held by the application.

Taking your current implementation of `category.jsp` as an example, perform the following steps:

1. Set a breakpoint on the line containing:

```
<c:when test="${category.id == pageContext.request.queryString}">
```

(To set a breakpoint, click in the left margin of the line. A breakpoint () icon displays.)

2. In the IDE's main toolbar, click the Debug Project () button. A debugging session is activated for the project, and the application's index page opens in the browser.
3. Click the bakery category in the index page. (You know that the ID for the bakery category is '3').
4. Return to the IDE, and note that the debugger is suspended on the line containing the breakpoint. When suspended, the margin shows a green arrow on the breakpoint (), and the line displays with green background.
5. Open the Variables window (Ctrl-Shift-1) and expand the Implicit Objects > pageContext > request > queryString node. Inspect the variable value and note that the value is '3', corresponding to the category ID from your selection.
6. Press the Finish Debugger Session () button to terminate the debugger session.

Display title heading above product table

1. Add the following SQL query to the top of the file, underneath the query you just implemented. (New query is shown in **bold**.)

```
<sql:query var="categories" dataSource="jdbc/affablebean">
    SELECT * FROM category
</sql:query>

<sql:query var="selectedCategory" dataSource="jdbc/affablebean">
    SELECT name FROM category WHERE id = ?
    <sql:param value="${pageContext.request.queryString}" />
</sql:query>
```

2. Use JSP EL syntax to extract the category name from the query and display it in the page. Make the following change to the `<p id="categoryTitle">` element. (Displayed in **bold**.)

```
<p id="categoryTitle">${selectedCategory.rows[0].name}</p>
```

Since the result from the `selectedCategory` query contains only one item (i.e., user can select only one category), you can retrieve the first row of the result set using `'selectedCategory.rows[0]'`. If a user selects the 'meats' category for example, the returned expression would be `'{name=meats}'`. You could then access the category name with `'${selectedCategory.rows[0].name}'`.

3. Save (Ctrl-S; ⌘-S on Mac) changes made to the file.
4. Return to the browser and refresh the category page. The name of the selected category now displays above the product table.



Note: As demonstrated in this and the previous step, you do not need to explicitly recompile, deploy, and run the project with each change to your code base. The IDE provides a Deploy on on Save feature, which is enabled for Java web projects by default. To verify that the feature is activated, right-click your project node in the Projects window and choose Properties. In the Project Properties window, click the Run category and examine the 'Deploy on Save' option.

Display product details within the table

1. Add the following SQL query to the top of the file, underneath the previous queries you implemented. (New query is shown in **bold**.)

```
<sql:query var="categories" dataSource="jdbc/affablebean">
    SELECT * FROM category
</sql:query>

<sql:query var="selectedCategory" dataSource="jdbc/affablebean">
    SELECT name FROM category WHERE id = ?
    <sql:param value="${pageContext.request.queryString}"/>
</sql:query>

<sql:query var="categoryProducts" dataSource="jdbc/affablebean">
    SELECT * FROM product WHERE category_id = ?
    <sql:param value="${pageContext.request.queryString}"/>
</sql:query>
```

2. Between the `<table id="productTable">` tags, replace the existing static table row placeholders (`<tr>` tags) with the following `<c:forEach>` loop. (Changes are displayed in **bold**.)

```

<table id="productTable">

    <c:forEach var="product" items="${categoryProducts.rows}" varStatus="iter">

        <tr class="${((iter.index % 2) == 0) ? 'lightBlue' : 'white'}">
            <td>
                
            </td>
            <td>
                ${product.name}
                <br>
                <span class="smallText">${product.description}</span>
            </td>
            <td>
                &euro; ${product.price} / unit
            </td>
            <td>
                <form action="addToCart" method="post">
                    <input type="hidden"
                        name="productId"
                        value="${product.id}">
                    <input type="submit"
                        value="add to cart">
                </form>
            </td>
        </tr>

    </c:forEach>

</table>

```

Note that in the above snippet an EL expression is used to determine the background color for table rows:

```
class="${((iter.index % 2) == 0) ? 'lightBlue' : 'white'}"
```

The API documentation for the `<c:forEach>` tag indicates that the `varStatus` attribute represents an object that implements the `LoopTagStatus` interface. Therefore, `iter.index` retrieves the index of the current round of the iteration. Continuing with the expression, `(iter.index % 2) == 0` evaluates the remainder when `iter.index` is divided by 2, and returns a boolean value based on the outcome. Finally, an EL conditional operator (`? :`) is used to set the returned value to 'lightBlue' if true, 'white' otherwise.

For a description of JSP Expression Language operators, see the Java EE 5 Tutorial: [JavaServer Pages Technology > Unified Expression Language > Operators](#).

3. Save (Ctrl-S; ⌘-S on Mac) changes made to the file.
4. Return to the browser and refresh the category page. Product details now display within the table for the selected category.



You have now completed this tutorial unit. In it, you explored how to connect your application to the database by setting up a connection pool and data source on the server, then referenced the data source from the application. You also created several context parameters, and learned how to access them from JSP pages. Finally, you implemented JSTL tags into the application's web pages in order to dynamically retrieve and display database data.

You can download and examine [snapshot 3](#) if you'd like to compare your work with the solution project. The solution project contains enhancements to the HTML markup and stylesheet in order to properly display all provided images. It also provides welcome page text, and a basic implementation for the page footer.

[Send Us Your Feedback](#)

Troubleshooting

If you are having problems, see the troubleshooting tips below. If you continue to have difficulty, or would like to provide constructive feedback, use the Send us Your Feedback link.

- You receive the following exception:

```
org.apache.jasper.JasperException: PWC6188: The absolute uri:
http://java.sun.com/jsp/jstl/core cannot be resolved in either web.xml or the jar files
deployed with this application
```

This is a [known issue](#) for NetBeans IDE 6.9. Try to deploy the project, then access the file by typing its URL in the browser. For example, if you are trying to view `testDataSource.jsp` in a browser, enter `'http://localhost:8080/AffableBean/test/testDataSource.jsp'` in the browser's URL field directly. Otherwise, add the IDE's JSTL 1.1 library to the project. In the Projects window, right-click the Libraries node and choose Add Library. Select JSTL 1.1. For more information, see: <http://forums.netbeans.org/topic28571.html>.

- You receive the following exception:

```
javax.servlet.ServletException: javax.servlet.jsp.JspException: Unable to get
connection, DataSource invalid: "java.sql.SQLException: Error in allocating a
connection. Cause: Class name is wrong or classpath is not set for :
com.mysql.jdbc.jdbc2.optional.MysqlDataSource"
```

This can occur when the MySQL driver has not been added to the domain lib folder. (Note that after adding, it is necessary to restart the server if it is already running.)

- You receive the following exception:

```
javax.servlet.ServletException: javax.servlet.jsp.JspException: Unable to get
connection, DataSource invalid: "java.sql.SQLException: No suitable driver found for
jdbc/affablebean"
```

This can occur when the jdbc/affablebean resource reference hasn't been added to the web.xml deployment descriptor.

- You receive the following exception:

```
javax.servlet.ServletException: javax.servlet.jsp.JspException: Unable to get
connection, DataSource invalid: "java.sql.SQLException: Error in allocating a
connection. Cause: Connection could not be allocated because: Access denied for user
'root'@'localhost' (using password: YES) "
```

This can occur when you are using an incorrect username/password combination. Make sure the username and password you use to connect to the MySQL server are correctly set for your connection pool in the sun-resources.xml file. Also, check that the username and password are correctly set for the connection pool in the GlassFish Administration Console.

See Also

NetBeans Resources

- [Connecting to a MySQL Database](#)
- [Introduction to Developing Web Applications](#)
- [Creating a Simple Web Application Using a MySQL Database](#)
- [Screencast: Database Support in NetBeans IDE](#)

MySQL Resources

- [The MySQL Community Librarian](#)
- [MySQL 5.1 Reference Manual](#)
- [MySQL and Java](#)
- [MySQL Forums](#)

JSP & EL Resources

- **Product Page:** [JavaServer Pages Technology](#)
- **Specification Download:** [JSR 245: JSP and EL 2.2 Maintenance Release](#)
- **API Documentation:** [JavaServer Pages 2.1 API Documentation](#)
- **Supporting Documentation:** [Java EE 5 Tutorial - Chapter 5: JavaServer Pages Technology](#)
- **Syntax Reference:** [JavaServer Pages 2.0 Syntax Reference](#)
- **Official Forum:** [Web Tier APIs - JavaServer Pages \(JSP\) and JSTL](#)

JSTL Resources

- **Product Page:** [JavaServer Pages Standard Tag Library](#)
- **Specification Download:** [JSR 52: JSTL 1.2 Maintenance Release](#)
- **Implementation Download:** [GlassFish JSTL Project Download](#)
- **Tag Library Documentation:** [JSTL 1.1 Tag Reference](#)
- **API Documentation:** [JSTL 1.1 API Reference](#)

Technical Articles & Reference Cards

- [Developing Web Applications With JavaServer Pages 2.0](#)
- [Web Tier to Go With Java EE 5: Summary of New Features in JSP 2.1 Technology](#)
- [Unified Expression Language](#)
- [Practical JSTL, Part 1](#)
- [A JSTL primer, Part 4: Accessing SQL and XML content](#)
- [JavaServer Pages v2.0 Syntax Card](#)
- [Essential JSP Expression Language Reference Card](#)
- [The Java Tutorials: JDBC Database Access](#)
- [Database Programming with JDBC and Java, Second Edition](#)
- [Essential JSP Expression Language Reference Card](#)
- [The JNDI Tutorial](#)