Getting Started with Relational Persistence

HELP.BCJAVA_DEV_PERS

Release 646

Copyright

Icons in Body Text

Icon	Meaning
	Caution
	Example
	Note
	Recommendation
	Syntax

Additional icons are used in SAP Library documentation to help you identify different types of information at a glance. For more information, see *Help on Help → General Information Classes and Information Classes for Business Information Warehouse* on the first page of any version of *SAP Library*.

Typographic Conventions

Type Style	Description
Example text	Words or characters quoted from the screen. These include field names, screen titles, pushbuttons labels, menu names, menu paths, and menu options. Cross-references to other documentation.
Example text	Emphasized words or phrases in body text, graphic titles, and table titles.
EXAMPLE TEXT	Technical names of system objects. These include report names, program names, transaction codes, table names, and key concepts of a programming language when they are surrounded by body text, for example, SELECT and INCLUDE.
`Example text`	Output on the screen. This includes file and directory names and their paths, messages, names of variables and parameters, source text, and names of installation, upgrade and database tools.
Example text	Exact user entry. These are words or characters that you enter in the system exactly as they appear in the documentation.
<Example text>	Variable user entry. Angle brackets indicate that you replace these words and characters with appropriate entries to make entries in the system.
EXAMPLE TEXT	Keys on the keyboard, for example, F2 or ENTER.

Getting Started with Relational Persistence

In this section you will learn how to work with Open SQL/JDBC and Open SQL/SQLJ in the SAP NetWeaver environment. The simple example demonstrates how you can store data in database tables and use a simple Web interface to insert data into the tables and to retrieve data using queries. The Web application accesses two tables in which data about the employees and the departments in a company is stored.

Procedure

The development of the example includes the following steps:

Creating the Database Tables

The data in this example is stored in two database tables:

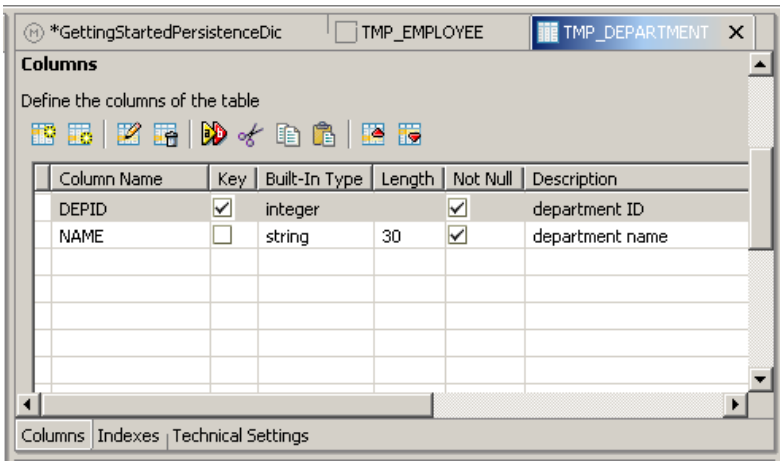
- TMP_DEPARTMENT contains data describing the properties of the departments
- TMP_EMPLOYEE describes the individual employees.

> The tables are the same as the ones used in Getting Started with JDO [Extern]. If you have already tried the JDO example, you can re-use the Java Dictionary project in this example.

Procedure

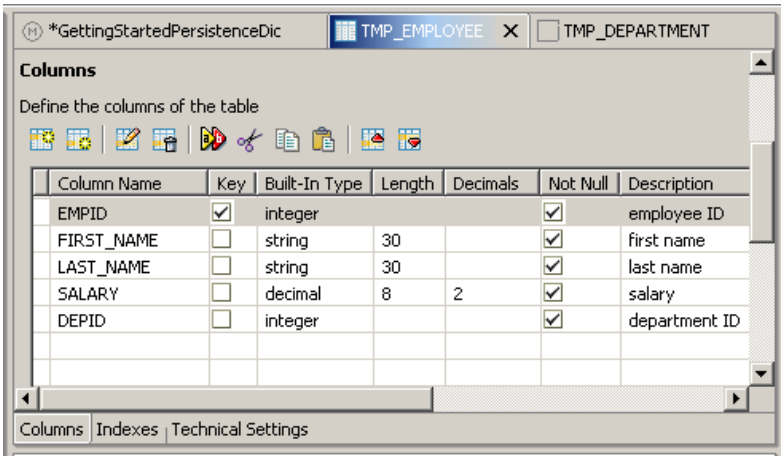
1. Open the Dictionary perspective and create a new Dictionary project – for example, *GettingStartedPersistenceDic*. Confirm the default project language setting (A*merican English*).
2. Create a table called TMP_EMPLOYEE for the employee data, and a table called TMP_DEPARTMENT for the department data. For more information, see Creating Tables [Extern].
3. In the TMP_DEPARTMENT table add the following columns:
   - DEPID
   - NAME

   Modify the parameters of the columns as follows:



4. In the TMP_EMPLOYEE table add the following columns:
   - EMPID
   - FIRST_NAME
   - LAST_NAME
   - SALARY
   - DEPID

   Modify the parameters of the columns as follows:



5. Save your data.

Result

You have created the database tables TMP_DEPARTMENT and TMP_EMPLOYEE in the offline Java Dictionary. The tables do not exist in the database yet. Now you can use the database tables in the offline SQLJ checker.

Go on with creating the Web project [Seite 6].

Creating the Web Project

The presentation layer of our application is a simple Web application with a Servlet and a static HTML page. You are going to access the database directly from the Web application.

The Web Project that you create here is the framework for developing the application.

Prerequisites

You must have created the database tables [Seite 5] in the *GettingStartedPersistenceDic* project.

Procedure

1. Go to *File* → *New* → *Project*… Choose *J2EE* in the left-hand tab, and *Web Module Project* in the right-hand tab.
2. Choose *Next*.
3. Enter a name for the project – for example, *GettingStartedOpenSQLWeb*.
4. Choose *Finish*.
5. In the *J2EE Development* perspective open the context menu of the *GettingStartedOpenSQLWeb* project, and choose *Properties*.
6. In the left-hand pane choose *Java Build Path*.
7. Choose *Projects* and select *GettingStartedPersistenceDic* from the list.
8. Choose *Libraries*. Choose *Add Variable*. Select ECLIPSE_HOME and choose *Extend*. Browse to the following file: *ECLIPSE_HOME/plugins/com.sap.sqlj/lib/sqljapi.jar.*
9. Choose *OK*.

Result

You have created the Web project to hold all Java and HTML sources. Now you can develop the EmployeeData class [Seite 7].

Developing the EmployeeData Class

The instances of the EmployeeData class are data transfer objects (DTO), which wrap the employee-related data exchanged between the persistence and the presentation layer.

Prerequisites

You must have created the GettingStartedOpenSQLWeb project [Seite 6].

Procedure

1. In the *Java* perspective, choose *GettingStartedOpenSQLWeb* and open the context menu. Choose *New* → *Package*. Enter **`temp.persistence.gettingstarted.dao`** as the package name.
2. Choose *Finish*.
3. From the context menu of *GettingStartedOpenSQLWeb* choose *New* → *Class*. To choose a package for the class, use *Browse…* next to the *Package* field, and select `temp.persistence.gettingstarted.dao` from the list.
4. Enter **`EmployeeData`** in the *Name* field and choose *Finish*.
5. The Java file is created and opens automatically. You can enter the code for the EmployeeData class.

```
package temp.persistence.gettingstarted.dao;

import java.math.BigDecimal;

public class EmployeeData {

   int _empId;
   String _firstName;
   String _lastName;
   BigDecimal _salary;
   int _depId;

   // the constructor
   public EmployeeData(int empId, String firstName, String
lastName, BigDecimal salary, int depId) {
      _empId = empId;
      _firstName = firstName;
      _lastName = lastName;
      _salary = salary;
      _depId = depId;
   }

    // the getter methods
   public int getDepId() {
      return _depId;
   }

   public int getEmpId() {
      return _empId;
   }

   public String getFirstName() {
      return _firstName;
   }

   public String getLastName() {
      return _lastName;
   }

   public BigDecimal getSalary() {
      return _salary;
   }
}
```

6. Save and close the file.

Result

You have created the EmployeeData class. Go on with developing the data access interface [Seite 9].

Developing the Data Access Interface

In this example you are going to implement data access using either SQLJ or JDBC. Therefore, you need to develop the DAO interface for database access, which enables you to choose between SQLJ and JDBC at runtime.

Prerequisites

You must have developed the EmployeeData class [Seite 7].

Procedure

1. In the *Java* perspective, choose *GettingStartedOpenSQLWeb* project and open its context menu.
2. Choose *New* → *Interface*. To choose a package, use *Browse*… next to the *Package* field, and select `temp.persistence.gettingstarted.dao` from the list.
3. Enter DAO in the *Name* field and choose *Finish*.
4. The Java file is created and opens automatically. Enter the code of the interface.

```
package temp.persistence.gettingstarted.dao;

import java.sql.SQLException;

public interface DAO {

   public void createDepartment(int depId, String depName)
throws SQLException;

   public void createEmployee(EmployeeData employee) throws
SQLException;

   public EmployeeData[] getEmployeesFromDepartment(int
depId) throws SQLException;

   public void commit() throws SQLException;

   public void rollback() throws SQLException;

   public void close() throws SQLException;

}
```

5. Save and close the file.

Result

You have created a data access interface. First, you are going to implement the interface using SQLJ. Go on with creating an SQLJ connection context class [Seite 10].

Creating the SQLJ Connection Context

To access the database using SQLJ, you need to create a connection context class `Ctx` that is associated with a JDBC DataSource. At runtime, an instance of this class represents a database connection.

Prerequisites

You must have created the GettingStartedOpenSQLWeb project [Seite 6].

Procedure

1. In the *Java* perspective, select *GettingStartedOpenSQLWeb* project and open its context menu.
2. Choose *New* → *Other*. Choose *Persistence* in the left-hand pane and *SQLJ Source* in the right-hand pane.
3. Choose *Next*.
4. Enter **`Ctx`** in the *Name* field. To choose a package, use *Browse*… next to the *Package* field, and choose `temp.persistence.gettingstarted.dao` from the list.
5. Choose *Finish*.
6. The SQLJ file opens automatically once it has been created. Enter the following code:

```
package temp.persistence.gettingstarted.dao;

#sql public context Ctx with (dataSource =
"java:comp/env/TMP_PERSISTENCE_EXAMPLE");
```

7. Save and close the file.

Result

You have created the SQLJ connection context for the example. Go on with creating the ResultSet iterator [Seite 10] for the employee data.

Creating an SQLJ ResultSet Iterator

To process the data from the table TMP_EMPLOYEE, you need to create an SQLJ result set iterator. This named iterator enables the access to the result set columns using named getter methods.

Prerequisites

You must have created the GettingStartedOpenSQLWeb project [Seite 6].

Procedure

1. In the *Java* perspective, select *GettingStartedOpenSQLWeb* project and open its context menu.
2. Choose *New* → *Other*… Choose *Persistence* in the left-hand pane and *SQLJ Source* in the right-hand pane.
3. Choose *Next*.

4. Enter **`EmployeeIter`** in the *Name* field. To choose a package, use *Browse*… next to the *Package* field, and choose `temp.persistence.gettingstarted.dao` from the list.
5. Choose *Finish*.
6. The SQLJ file opens automatically after it has been created. Enter the following code:

```
package temp.persistence.gettingstarted.dao;

#sql iterator EmployeeIter(int EMPID, String FIRST_NAME,
String LAST_NAME, java.math.BigDecimal SALARY);
```

7. Save and close the file.

Result

You can now implement the data access interface with SQLJ [Seite 11].

Implementing the Data Access Interface with SQLJ

To access the database using SQLJ, you need an appropriate implementation of the data access interface `temp.persistence.gettingstarted.dao.DAO`. This procedure demonstrates how you implement the interface with SQLJ.

The example uses local transactions in the communication with the database. This model can be used in simple applications only. Therefore, when writing complex J2EE applications, you must use JTA transactions [Extern].

Prerequisites

You must have created the:

- GettingStartedOpenSQLWeb project [Seite 6]
- DAO interface [Seite 9]
- and the connection context Ctx [Seite 10] and the result set iterator EmployeeIter [Seite 10].

Procedure

1. In the *Java Perspective*, select *GettingStartedOpenSQLWeb* and open its context menu.
2. Choose *New* → *Other*… Select *Persistence* in the left-hand pane, and *SQLJ Source* in the right-hand pane.
3. Choose *Next*.
4. Enter **`SqljDAO`** as the class name. To choose a package, use *Browse*… next to the *Package* field, and then select `temp.persistence.gettingstarted.dao` from the list. To choose the interface that the class implements, use *Add*… next to the Interfaces field. Type `DAO` in the *Choose Interfaces* field. Select the DAO interface from the `temp.persistence.gettingstarted.dao` package and confirm the selection by choosing *OK*.
5. To create the file, choose *Finish*.

6. The SQLJ file opens automatically. Modify the generated code as follows:

   a. After the package declaration, add an import declaration for `java.util.ArrayList`, which you will use to order the employee data obtained from the database by a select query:

```
import java.util.ArrayList;
```

   b. Add a constructor for the class and declare the connection context `ctx` variable:

```
public SqljDAO() throws SQLException {
      ctx = new Ctx();
      if (ctx == null) {
         throw new SQLException("ctx == null");
      }
      ctx.getConnection().setAutoCommit(false);
   }

   Ctx ctx;
```

   c. Implement the `createDepartment()` method of the DAO interface. It should insert data records about the created departments in the relevant columns of the TMP_DEPARTMENT table:

```
public void createDepartment(int depId, String depName)
      throws SQLException {
        #sql [ctx] { insert into TMP_DEPARTMENT (DEPID,
NAME)
                   values (:depId, :depName) };
}
```

   d. Implement the `createEmployee()` method of the DAO interface. It should insert data records in the relevant columns of the TMP_EMPLOYEE table:

```
public void createEmployee(EmployeeData employee) throws
SQLException {
      #sql [ctx] { insert into TMP_EMPLOYEE (EMPID,
FIRST_NAME, LAST_NAME, SALARY, DEPID)
                   values (:(employee.getEmpId()),
                           :(employee.getFirstName()),
                           :(employee.getLastName()),
                           :(employee.getSalary()),
                           :(employee.getDepId())) };
}
```

   e. Implement the `getEmployeesFromDepartment()` method. As its name implies, the method should select from the database tables the records for the employees in a particular department. The records are ordered in an ArrayList, which is then cast to an array of EmployeeData objects.

```
public EmployeeData[] getEmployeesFromDepartment(int depId)
throws SQLException {
      EmployeeIter iter = null;
      ArrayList list = new ArrayList();

      #sql [ctx] iter = { select EMPID, FIRST_NAME,
LAST_NAME, SALARY
                          from TMP_EMPLOYEE
                          where DEPID = :depId };
      while (iter.next()) {
         EmployeeData data = new EmployeeData(iter.EMPID(),

iter.FIRST_NAME(),

iter.LAST_NAME(),

iter.SALARY(),
                                                depId);
         list.add(data);
      }
      return (EmployeeData[])list.toArray(new
EmployeeData[] {});
}
```

f. Implement the methods for finishing the work and closing the connection:

```
// commits the work
public void commit() throws SQLException {
      #sql [ctx] { commit work };
}

// rolls back the work
public void rollback() throws SQLException {
      #sql [ctx] { rollback work };
}

// closes the connection
public void close() throws SQLException {
      if (ctx != null)
          ctx.close();
}
```

7. Save and close the file.

Result

Now you have an SQLJ-based implementation of the data access interface. Go on with implementing the interface using JDBC [Seite 13].

Implementing the Data Access Interface with

JDBC

Once you have implemented the `temp.persistence.gettingstarted.dao.DAO` interface using SQLJ, you also need to provide a JDBC-based implementation to enable your application to access the database using JDBC.

The JDBC-based implementation uses local transactions in the communication with the database. This model can be used in simple applications only. Therefore, when writing complex J2EE applications, you must use JTA transactions [Extern].

Prerequisites

You must have created the GettingStartedOpenSQLWeb project [Seite 6].

You must have developed the DAO interface [Seite 9].

Procedure

1. Select *GettingStartedOpenSQLWeb* project and open its context menu. Choose *Properties*.
2. Choose *Java Build Path*. In the *Source* tab choose *Add Folder…*.
3. Expand *GettingStartedOpenSQLWeb* and select *gen_sqlj*. Confirm your choice using *OK*.
4. In the *Java* perspective, select *GettingStartedOpenSQLWeb* and open its context menu.
5. Choose *New* → *Class*. Enter **`JdbcDAO`** as the class name. To choose a package, use *Browse…* next to the *Package* field, and then select `temp.persistence.gettingstarted.dao` from the list. To choose the interface that the class implements, use *Add…* next to the *Interfaces* field. Type DAO in the *Choose Interfaces* field. Select the DAO interface from the `temp.persistence.gettingstarted.dao` package and confirm the selection by choosing *OK*.
6. To create the file, choose *Finish*.
7. The Java file opens automatically after being created. Modify the code of the class as follows.

   a. Add a constructor of the class in which you must obtain a connection to the database using a DataSource. The connection is an instance of the `java.sql.Connection` interface. Include a declaration for the connection object as well.

```
public JdbcDAO() throws SQLException {

      // obtain a database connection using a DataSource
      try {
         InitialContext ctx = new InitialContext();
         DataSource dataSource =
            (DataSource)
ctx.lookup("java:comp/env/TMP_PERSISTENCE_EXAMPLE");
         conn = dataSource.getConnection();
         if (conn == null) {
            throw new SQLException("conn == null");
         }
         conn.setAutoCommit(false);
      } catch (NamingException ex) {
```

```
            throw new SQLException("NamingException: " +
ex.getMessage());
         }
      }

Connection conn;
```

b. Implement the `createDepartment()` method of the DAO interface. Using a PreparedStatement, it should insert the department data records into the relevant columns of the TMP_DEPARTMENT table.

```
public void createDepartment(int depId, String depName)
   throws SQLException {
   PreparedStatement stmt =
      conn.prepareStatement(
         "insert into TMP_DEPARTMENT (DEPID, NAME) values
(?, ?)");
   try {
      stmt.setInt(1, depId);
      stmt.setString(2, depName);
      stmt.executeUpdate();
   } finally {
      stmt.close();
   }
}
```

c. Implement the `createEmployee()` method of the DAO interface. It should insert the employee data records in the relevant columns of the TMP_EMPLOYEE table.

```
public void createEmployee(EmployeeData employee) throws
SQLException {
   PreparedStatement stmt =
      conn.prepareStatement(
         "insert into TMP_EMPLOYEE (EMPID, FIRST_NAME,
LAST_NAME, SALARY, DEPID) "
            + "values (?, ?, ?, ?, ?)");
   try {
      stmt.setInt(1, employee.getEmpId());
      stmt.setString(2, employee.getFirstName());
      stmt.setString(3, employee.getLastName());
      stmt.setBigDecimal(4, employee.getSalary());
      stmt.setInt(5, employee.getDepId());
      stmt.executeUpdate();
   } finally {
      stmt.close();
   }
}
```

d. Implement the `getEmployeesFromDepartment()` method of the DAO interface. Using a ResultSet, it should select the employees in a particular department. The records are added in an ArrayList, which is then cast to an array of EmployeeData objects.

```
public EmployeeData[] getEmployeesFromDepartment(int depId)
   throws SQLException {
      ArrayList list = new ArrayList();
      PreparedStatement stmt = conn.prepareStatement(
         "select EMPID, FIRST_NAME, LAST_NAME, SALARY "
            + "from TMP_EMPLOYEE "
            + "where DEPID = ?");
      try {
         stmt.setInt(1, depId);
         ResultSet rs = stmt.executeQuery();
         try {
            while (rs.next()) {
               EmployeeData data =
                  new EmployeeData(
                     rs.getInt(1),
                     rs.getString(2),
                     rs.getString(3),
                     rs.getBigDecimal(4),
                     depId);
               list.add(data);
            }
         } finally {
            rs.close();
         }
         return (EmployeeData[]) list.toArray(new
EmployeeData[] {
         });
      } finally {
      stmt.close();
   }
}
```

e. Implement the methods for finishing the work and closing the connection.

```
// commits the work
public void commit() throws SQLException {
   conn.commit();
}

// rolls back the work
public void rollback() throws SQLException {
   conn.rollback();
}

// closes the connection
public void close() throws SQLException {
   if (conn != null)
      conn.close();
}
```

f. To add the required imports, position the cursor anywhere in the Java editor and open the context menu. Choose *Source* → *Organize Imports*. Select `javax.sql.DataSource` and then choose *Next*.

g. Select `java.sql.Connection` and confirm by choosing *Finish*. The following import declarations are added to the existing import of `java.sql.SQLException`:

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.ArrayList;

import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.sql.DataSource;
```

8. Save and close the file.

Result

Using the implementations of the DAO interface, you can now access the database via both SQLJ and JDBC. The next step is to create the presentation layer [Seite 17] for the application.

Developing the Web Front End

To complete the application, you have to develop its presentation layer. It is Web-based and consists of a static HTML and a servlet that communicates with the implementation of the data access interface to send the user input, and to return the results of the database operations.

Prerequisites

You must have created the GettingStartedOpenSQLWeb project [Seite 6].

You must have implemented the DAO interface using both SQLJ [Seite 11] and JDBC [Seite 13].

Procedure

Creating the HTML Page

1. In the *J2EE Development* perspective, select *GettingStartedOpenSQLWeb* and open its context menu.
2. Choose *New* → *HTML*. Enter **`index`** as the page name.
3. Choose *Finish*.
4. The HTML file opens automatically. In the *Source* tab enter the source of the page. It should contain forms for the following functions:
   - Creating new department
   - Creating new employee
   - Selecting the employees from a department with a given ID
   - Switching between SQLJ and JDBC

Here is the source of the HTML page:

```
<html>
<head>
      <meta http-equiv= "Content-Type" content= "text/html"
>
      <title> Getting Started With Java Persistence
</title>
      <style type="text/css">
            h2 { text-align:left; }
            h4 { text-align:left; }
            h5 { text-align:left; margin-left:0; margin-
right:0}
            .framed { border:solid 1px; }
            .narrow { margin-top:1px; margin-bottom:1px }
      </style>
</head>
<body>
      <h4 class="narrow"> SAP WEB APPLICATION SERVER </h4>
      <hr>
      <h2>Getting Started With Relational Persistence</h2>
      <h4>Select an option and enter the relevant data. To
confirm, use <i>Submit</i>. The department ID is
<u>required</u> for all options.</h4>
      <h5 class="narrow">Recommended sequence:</h5>
      <blockquote>
      <h5 class="narrow">1. Create a department</h5>
      <h5 class="narrow">2. Create employees within the
department</h5>
      <h5 class="narrow" style= "margin-bottom: 10px">3.
List employees </h5>
      </blockquote>
      <form method= "POST" action= "ProcessInput" >
      <table class="framed" style= "background-
color:lightblue" >
            <tr>
               <td class="framed">Department ID: <input
type= "text" name= "DEPID" size= "10" value="1"></td>
            </tr>
            <tr>
                <td class="framed"><input type= "radio"
value= "NEW_DEP" name= "ACTION" > New Department:</td>
             <td class="framed">
               <table>
                     <tr>
                         <td> Name:</td>
                         <td><input type= "text" name=
"NAME" size= "30"></td>
                     </tr>
               </table>
                </td>
            </tr>
            <tr>
                <td class="framed"><input type= "radio"
name= "ACTION" value= "NEW_EMP" > New Employee:</td>
                <td class="framed">
                  <table>
                        <tr>
                            <td>Employee ID:</td>
```

```
                            <td><input type= "text" name=
"EMPID" size= "10"</td>
                        </tr>
                        <tr>
                            <td>First Name: </td>
                            <td><input type= "text" name=
"FIRST_NAME" size= "30" ></td>
                        </tr>
                        <tr>
                            <td>Last Name: </td>
                            <td><input type= "text" name=
"LAST_NAME" size= "30" ></td>
                        </tr>
                        <tr>
                            <td>Salary: </td>
                            <td><input type= "text" name=
"SALARY" size= "10"></td>
                        </tr>
                    </table>
                </td>
            </tr>
            <tr>
                <td class="framed"><input type= "radio"
name= "ACTION" value= "LIST" checked>List Employees</td>
            </tr>
      </table>
      <table>
            <tr>
                <td><input type="radio" value="SQLJ"
checked name="DAO">SQLJ</td>
                <td><input type="radio" value="JDBC"
name="DAO">JDBC</td>
            </tr>
      </table>
      <p><input type="submit" value="Submit" name="B3"></p>
     </form>
</body>
</html>
```

5. Save and close the file.

Creating the Servlet

1. In the J2EE Development perspective, select *GettingStartedOpenSQLWeb* and open its context menu.
2. Choose *New* → *Package*. Enter **`temp.persistence.gettingstarted.web`** as the package name and choose *Finish*.
3. In the context menu of *GettingStartedOpenSQLWeb* choose *New* → *Servlet*. Enter **`ProcessInput`** as the servlet name. Choose *HTTP Servlet* for Servlet Type. To choose a package, use *Browse…* next to the *Servlet Package* field, and choose `temp.persistence.gettingstarted.web` from the list.
4. Choose *Finish*.
5. The file opens for editing automatically once it has been created. Edit its code as follows:
    a. Implement the `doGet()` method, which receives the input from the HTML page and invokes the relevant methods of the SqljDAO and JdbcDAO classes:

```
protected void doGet(
HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
   response.setContentType("text/html");
   DAO dao = null;
   PrintWriter out = response.getWriter();
   try {
      out.println("<html><body>");
      String action = request.getParameter("ACTION");
      int depId =
Integer.parseInt(request.getParameter("DEPID"));
      String daoType = request.getParameter("DAO");
      if (daoType.equals("SQLJ")) {
         dao =
            (DAO) Class

   .forName("temp.persistence.gettingstarted.dao.SqljDAO")
               .newInstance();
         out.println("<h4>Using SQLJ</h4>");
      } else {
         dao =
            (DAO) Class

   .forName("temp.persistence.gettingstarted.dao.JdbcDAO")
               .newInstance();
         out.println("<h4>Using JDBC</h4>");
      }
      try {
         if (action.equals("NEW_DEP")) {
            String depName = request.getParameter("NAME");
            dao.createDepartment(depId, depName);
            dao.commit();
            out.println("Department \"" + depName + "\"
created.");
         } else if (action.equals("NEW_EMP")) {
            int empId =
Integer.parseInt(request.getParameter("EMPID"));
            String firstName =
request.getParameter("FIRST_NAME");
            String lastName =
request.getParameter("LAST_NAME");
            BigDecimal salary =
               new
BigDecimal(request.getParameter("SALARY"));
            EmployeeData data =
               new EmployeeData(
                  empId,
                  firstName,
                  lastName,
                  salary,
                  depId);
            dao.createEmployee(data);
            out.println(
               "Employee \""
                  + firstName
                  + " "
                  + lastName
                  + "\" created.");
            dao.commit();
```

```
         } else if (action.equals("LIST")) {
            EmployeeData[] emps =
dao.getEmployeesFromDepartment(depId);
            if (emps.length == 0) {
               out.println("<br>no data");
            } else {
               out.println("<table><tr>");
               out.println("<td>Employee ID</td>");
               out.println("<td>First Name</td>");
               out.println("<td>Last Name</td>");
               out.println("<td>Salary</td>");
               out.println("<td>Department ID</td></tr>");
               for (int i = 0; i < emps.length; i++) {
                  out.println("<tr>");
                  out.println("<td>" + emps[i].getEmpId() +
"</td>");
                  out.println(
                     "<td>" + emps[i].getFirstName() +
"</td>");
                  out.println(
                     "<td>" + emps[i].getLastName() +
"</td>");
                  out.println("<td>" + emps[i].getSalary()
+ "</td>");
                  out.println("<td>" + emps[i].getDepId() +
"</td>");
                  out.println("</tr>");
               }
               out.println("</table>");
            }
         } else {
            out.println("Illegal action: " + action);
         }
      } finally {
         dao.close();
      }
   } catch (Throwable ex) {
      out.println("Exception caught");
      out.println("<code>");
      ex.printStackTrace(out);
      out.println("</code>");
   } finally {
      out.println("<p><a href=index.html>Home</a></p>");
      out.println("</body></html>");
   }
}
```

b. Implement the `doPost()` method, which invokes `doGet()`:

```
protected void doPost(
   HttpServletRequest request,
   HttpServletResponse response)
   throws ServletException, IOException {
   doGet(request, response);
}
```

c. To add the required imports, position the cursor anywhere in the Java editor and open the context menu. Choose *Source* → *Organize Imports.* The following import declarations are added to the existing ones:

```
import java.io.PrintWriter;
import java.math.BigDecimal;

import temp.persistence.gettingstarted.dao.DAO;
import temp.persistence.gettingstarted.dao.EmployeeData;
```

6. Save and close the file.

Result

You have developed all components of the example application. The next step is its assembly [Seite 22].

Assembling the Application

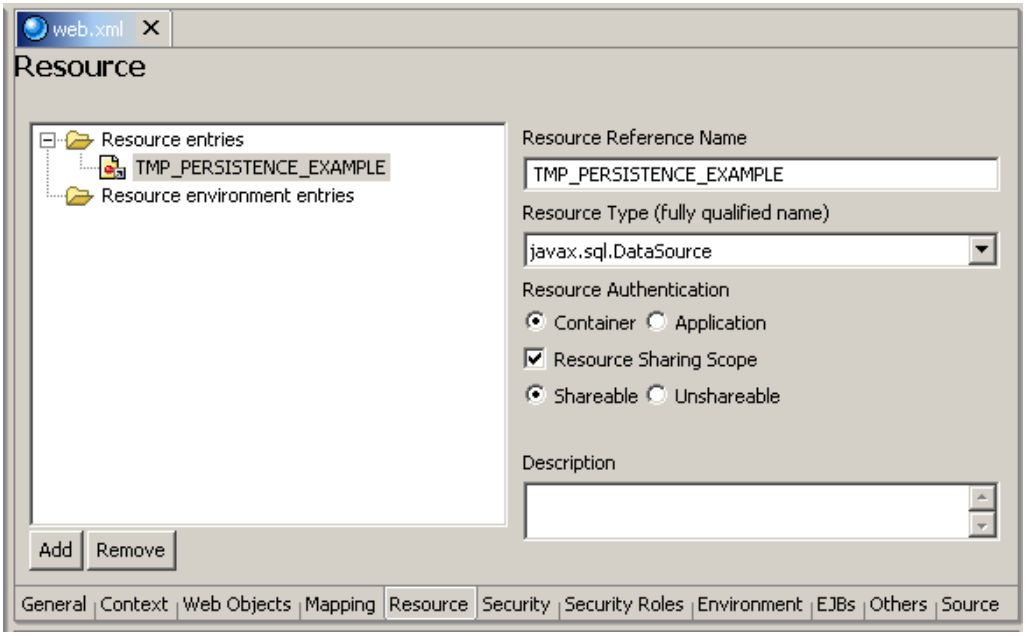
To prepare your application for deployment, you have to:

- Build the WAR
- Create an Enterprise Application Project
- Create a DataSource Alias
- Assemble the EAR file

Procedure

Building the WAR

1. In the *J2EE Development* perspective, extend the *GettingStartedOpenSQLWeb* node and open *web.xml*.
2. Go to the *Mapping* tab, select *Servlet Mappings* and choose *Add*.
3. Add the ProcessInput servlet and enter **`ProcessInput`** in the *URLPattern* field.
4. You must also add a resource reference for the DataSource that your application uses. To do this, go to the *Resource* tab, select *Resource entries* and choose *Add*. Modify the entry as follows:



For more information about DataSource resource references, see Accessing DataSource [Extern].

5. Save and close the XML file.
6. From the context menu of the project choose *Build WAR File*.

Creating the Enterprise Application Project

1. Choose *File* → *New* → *Project*. Select *J2EE* in the left-hand pane and *Enterprise Application Project* in the right-hand pane. Choose *Next*.
2. Enter a name for the project – for example, *GettingStartedOpenSQLEar*. Choose *Next*.
3. In the *Referenced Projects* field choose *GettingStartedOpenSQLWeb*. Choose *Finish*.
4. Extend the *GettingStartedOpenSQLEar* node and open *application.xml*.
5. On the *General* tab, enter *GettingStartedOpenSQL* as a display name. On the *Modules* tab enter *gettingstarted-sql* as a context root.
6. Save and close the XML file.

Creating the DataSource Alias

1. Select the *GettingStartedOpenSQLEar* project and open its context menu. Choose *New* → *META-INF/data-source-aliases.xml*.
2. Enter *TMP_PERSISTENCE_EXAMPLE* as the name of the DataSource alias.

   For more information about DataSource aliases, see Managing Aliases [Extern].
3. Choose *Finish*.

Assembling the EAR

Open the context menu of the *GettingStartedOpenSQLEar* project and choose *Build EAR File*. The system informs you if the process has finished successfully.

Result

Now you can deploy and run the application [Seite 24].

Deploying and Running the Application

The last step in the development of the Persistence Example application is its deployment and testing. In this procedure you have to:

- Deploy the database tables
- Deploy the application EAR
- Run the application

Prerequisites

- You must have entered the SAP J2EE Engine settings.

  To set up the SAP J2EE Engine, go to *Window* → *Preferences* → *SAP J2EE Engine*. You can set either remote or local installation. The default message server port is 3601.
- You must have launched the SAP J2EE Engine. For more information, see Starting and Stopping the SAP System [Extern].

Procedure

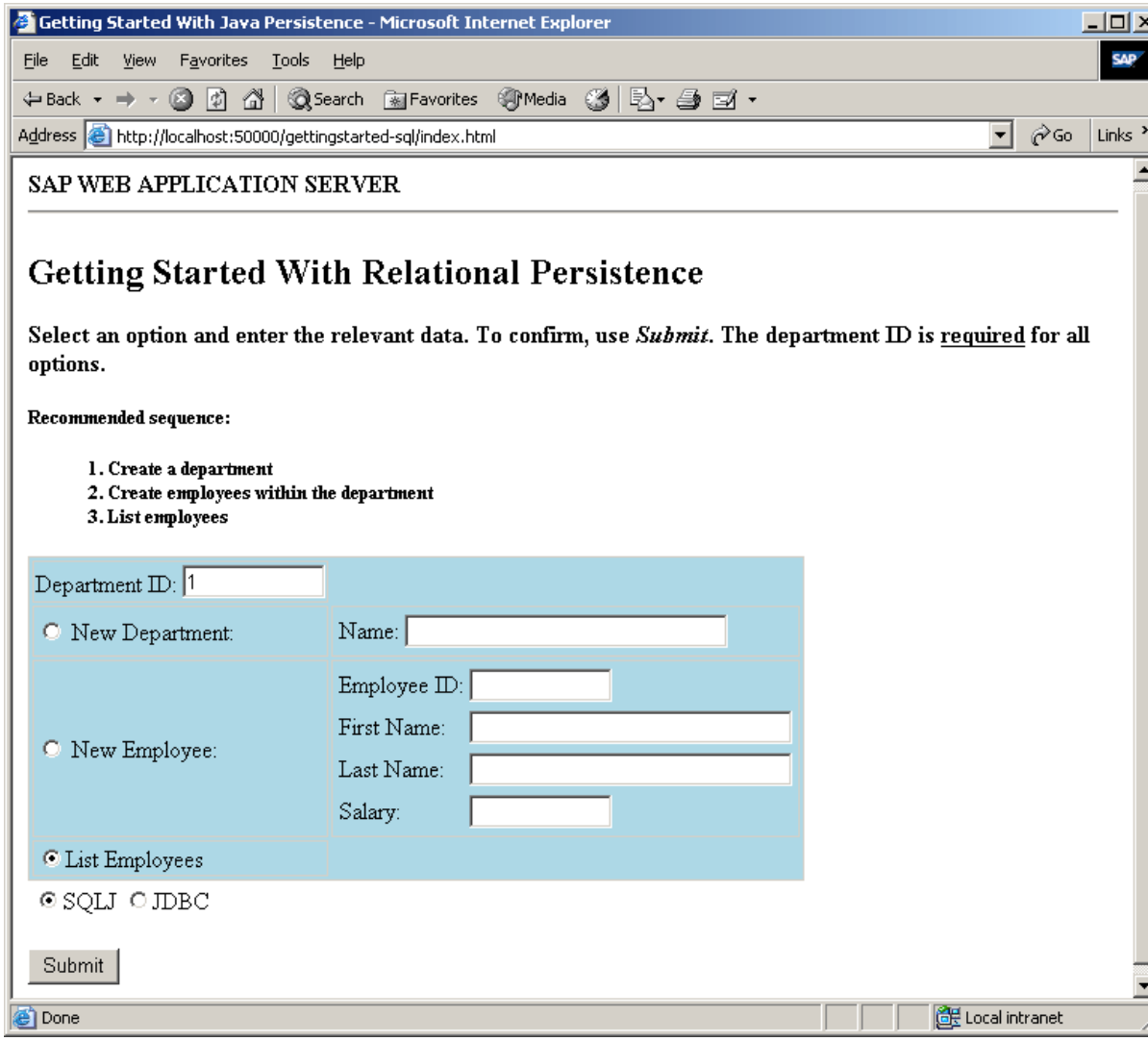
1. Deploy the database tables
   a. In the *Dictionary* perspective, open the context menu of the *GettingStartedPersistenceDic* project.
   b. Choose *Create Archive*.
   c. In the context menu of the project choose *Deploy*.

   The tables are the same as the ones used in Getting Started with JDO [Extern]. If you have already tried the JDO example, and have the tables deployed in the database, you can skip this step.
2. Deploy the application EAR:
   a. In the *J2EE Development* perspective, extend the *GettingStartedOpenSQLEar* project node.
   b. Open the context menu of *GettingStartedOpenSQLEar.ear* and choose *Deploy to J2EE engine*.
   c. The application starts automatically after being deployed.
3. Test the application:
   a. Invoke the application in your browser:

   ```
   http://<yourhost>:<HTTP port>/<Context Root>/
   ```

http://localhost:50000/gettingstarted-sql/



b. Create a department:
   i. Enter a *Department ID* – for example, 1.
   ii. Choose *New Department* and enter a name.
   iii. Choose either *SQLJ* or *JDBC*.
   iv. Choose *Submit*.

c. Create an employee:
   i. Enter the ID of the department you have created.
   ii. Choose *New Employee* and enter the required data.
   iii. Choose either *SQLJ* or *JDBC*.
   iv. Choose *Submit*.

d. List all employees in a department.
   i. Enter the ID of an existing department.
   ii. Choose *List Employees*.
   iii. Choose either *SQLJ* or *JDBC*.

iv. Choose *Submit*.