Getting Started with JDO

HELP.BCJAVA_DEV_PERS

Release 646

Copyright

© Copyright 2004 SAP AG. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or for any purpose without the express permission of SAP AG. The information contained herein may be changed without prior notice.
Some software products marketed by SAP AG and its distributors contain proprietary software components of other software vendors.

Microsoft®, WINDOWS®, NT®, EXCEL®, Word®, PowerPoint® and SQL Server® are registered trademarks of Microsoft Corporation.

IBM®, DB2®, DB2 Universal Database, OS/2®, Parallel Sysplex®, MVS/ESA, AIX®, S/390®, AS/400®, OS/390®, OS/400®, iSeries, pSeries, xSeries, zSeries, z/OS, AFP, Intelligent Miner, WebSphere®, Netfinity®, Tivoli®, Informix and Informix® Dynamic ServerTM are trademarks of IBM Corporation in USA and/or other countries.

ORACLE® is a registered trademark of ORACLE Corporation.

UNIX®, X/Open®, OSF/1®, and Motif® are registered trademarks of the Open Group.

Citrix®, the Citrix logo, ICA®, Program Neighborhood®, MetaFrame®, WinFrame®, VideoFrame®, MultiWin® and other Citrix product names referenced herein are trademarks of Citrix Systems, Inc.
HTML, DHTML, XML, XHTML are trademarks or registered trademarks of W3C®, World Wide Web Consortium, Massachusetts Institute of Technology.

JAVA® is a registered trademark of Sun Microsystems, Inc.

JAVASCRIPT® is a registered trademark of Sun Microsystems, Inc., used under license for technology invented and implemented by Netscape.

MarketSet and Enterprise Buyer are jointly owned trademarks of SAP AG and Commerce One.
SAP, SAP Logo, R/2, R/3, mySAP, mySAP.com and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP AG in Germany and in several other countries all over the world. All other product and service names mentioned are trademarks of their respective companies.

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 JDO 5
Creating the Database Tables 5
Creating the Web Project 7
Defining the Persistence Capable Classes 7
Defining the Object Identity Classes 11
Defining the JDO Metadata 14
Defining the O/R Mapping 15
Running the JDO Enhancer and Checker Tools 17
Implementing the Business Logic 20
Developing the Web Front End 24
Assembling the Application 29
Deploying and Running the Application 30

Getting Started with JDO

In this section you will learn how to work with Java Data Objects in the SAP NetWeaver environment. The simple example included in the guide introduces the basic features of JDO, such as creating and reading persistent objects. A Web interface is used to pass the parameters to the underlying business logic and to visualize the result for the user. The persistent layer is represented by a relational database. The Web application accesses two tables in which data about the employees and the departments in a company is stored.

The JDO tools are still not integrated in the SAP NetWeaver Developer Studio; therefore, you will have to create some of the XML files for the example manually. You will find detailed instructions about how to do that in the guide.

Procedure

To get started with JDO, you have to learn how to:

- Define the database tables [Seite 5]
- Create a Web project and configure the build path [Seite 7]
- Create the persistence capable classes [Seite 7]
- Create the object identity classes [Seite 11]
- Define the persistent properties of the JDO [Seite 14]
- Define the object/relational mapping for the persistent capable classes [Seite 15]
- Enhance the JDO classes and check the consistency of the tool [Seite 17]
- Implement the business logic [Seite 20]
- Develop the Web front end [Seite 24]
- Assemble the application [Seite 29]
- Deploy and run the application [Seite 30]

Creating the Database Tables

The SAP implementation of the JDO standard uses a relational database as a store for the persistent data. The data in the database is organized in tables. Each persistent capable class is mapped to a table that contains a column for each persistent field of the class.

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

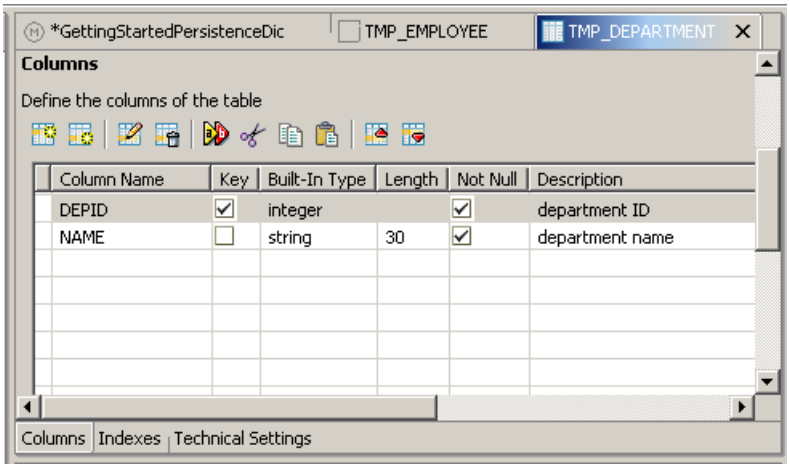
- TMP_DEPARTMENT contains data describing the properties of the departments
- TMP_EMPLOYEE describes the individual employees.

The tables are the same as the ones used in Getting Started with Relational Persistence [Extern]. If you have already tried the relational persistence example, and have the tables deployed in the database, you can skip this step.

Procedure

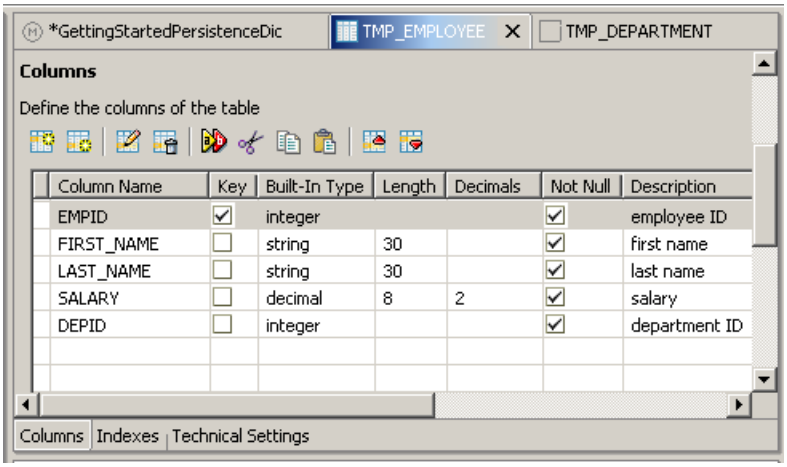
1. Open the Dictionary perspective and create a new Dictionary project, such as *GettingStartedPersistenceDic*. Confirm the default project language setting (*American English*).
2. Create a table called `TMP_EMPLOYEE` for the employee data, and a table called `TMP_DEPARTMENT` for the department data. 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.

Now you have to create the Web project [Seite 7] for the application.

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 work with the persistent objects using a helper class that implements the business logic.

The Web project that you create here is the framework for developing the application. It will contain the JDO classes, the Web front end module, and the helper class.

Prerequisites

You are in the SAP NetWeaver Developer Studio [Extern].

Procedure

1. Choose *File* → *New* → *Project…*. Choose *J2EE* in the left-hand tab and *Web Module Project* in the right-hand tab. Choose *Next*.
2. Enter a name for the project, such as *GettingStartedJDOWeb*. Choose *Finish.*
3. In the *J2EE Development* perspective open the context menu of the *GettingStartedJDOWeb* project. Choose *Properties*.
4. In the left-hand pane choose *Java Build Path*.
5. Choose *Libraries* and then choose *Add Variable*.
6. Select ECLIPSE_HOME and choose *Extend*.
7. Browse to the following file:
   ECLIPSE_HOME/plugins/com.sap.ide.eclipse.ext.libs.jdo/lib/jdo.jar.
8. Repeat the procedure to add the following file:
   ECLIPSE_HOME/plugins/com.sap.jdo.api/lib/sapjdoapi.jar.
9. Choose *OK*.

Result

You have created the project and you have added all libraries that are necessary for compiling the JDO. Now you can define the persistent capable classes [Seite 7].

Defining the Persistence Capable Classes

Persistence capable classes define objects that can be stored in and retrieved from the persistent data store.

We are now going to create the following persistence capable classes:

- Employee – represents the employees of a company
- Department – represents the departments of the company.

Prerequisites

You must have created a Web Project and set its build path [Seite 7].

Procedure

The persistent fields in the Employee and the Department classes are declared as private variables. To read and write data in the fields, use the `getXXX()` and `setXXX()` method model in the same way as they are used in the JavaBeans model.

The relationship between the Employee and the Department instances is many-to-one – that is, each employee can work in a single department, although a department may comprise an arbitrary number of employees. This relationship is implemented by the field `Department` in the Employee class, and the field `Employees` in the Department class.

Both the Employee and Department instances are identified by a unique numeric identifier. It is set in the constructor of the class, and can only be read using a `get()` method, but you cannot change it in a `set()` method.

Furthermore, the JDO standard requires that each persistence capable class has a no-args constructor.

Creating the Employee Class

1. In the *J2EE Development* perspective, open the context menu of your Web project.
2. Choose *New* → Package. Enter **`temp.persistence.gettingstarted.jdo`** as the package name and choose *Finish*.
3. Choose *New* → *Java Class*. Enter **`Employee`** as the class name. To choose a package, choose *Browse…* and select `temp.persistence.gettingstarted.jdo`. To create the class, choose *Finish*.
4. The new Java file opens automatically. You can enter the code for the `Employee` class.

   Each employee is defined by the following attributes:

   - First name
   - Last name
   - Salary
   - Numeric identifier (employee ID)

   You must include an import declaration for the class `java.math.BigDecimal`, because the variable `salary` is of type BigDecimal.

   The Employee class then looks as follows:

```
package temp.persistence.gettingstarted.jdo;

import java.math.BigDecimal;

public class Employee {

   // Class attributes: the persistent fields of an
employee.
   // Also defined inside the file Employee.jdo
   private int empId;
```

```
    private String firstName;
    private String lastName;
    private BigDecimal salary;
    private Department department;

    // Required: a no-args constructor
    public Employee() {
       this.empId = -1;
       this.firstName = "INITIAL";
       this.lastName = "INITIAL";
       salary = new BigDecimal(0.0);
       department = null;
    }

    // Constructor where the ID is set
    public Employee(int empId) {
       this.empId = empId;
       this.firstName = "INITIAL";
       this.lastName = "INITIAL";
       salary = new BigDecimal(0.0);
       department = null;
    }

    // Implement the getter methods of the class
    public Department getDepartment() {
       return department;
    }

    public int getEmpId() {
       return empId;
    }

    public String getFirstName() {
       return firstName;
    }

    public String getLastName() {
       return lastName;
    }

    public BigDecimal getSalary() {
       return salary;
    }

    // Implement the setter methods of the class
    public void setDepartment(Department dept) {
    this.department = dept;
    }

    public void setFirstName(String fname) {
       firstName = fname;
    }

    public void setLastName(String lname) {
       lastName = lname;
    }

    public void setSalary(BigDecimal sal) {
       salary = sal;
    }
```

```
}
```

Creating the Department Class

1. Use the procedure described above to create a new class. At step 2, enter **`Department`** as the class name.
2. Enter the code for the `Department` class.

   The department instances are defined by the following attributes:

   - Name
   - Numeric identifier (Department ID)

   You have to import the `java.util.*` package because the type of the employees field is `java.util.Set`, and the `getEmployees()` method returns a `java.util.Collection`. In addition, you have to implement logic for adding and removing employees in the department, and you can use `java.util.HashSet` to organize the addition of employees.

   The Department class then looks as follows:

```
package temp.persistence.gettingstarted.jdo;

import java.util.*;

public class Department {

   // Class attributes: the persistent fields of a
department.
   // Also defined inside the file Department.jdo
   private int depId;
   private String name;
   private Set employees;

   // Required: a no-args constructor
   public Department() {
      this.depId = -1;
      name = "INITIAL";
      employees = new HashSet();
   }

   // Constructor where the ID is set
   public Department(int depId) {
      this.depId = depId;
      name = "INITIAL";
      employees = new HashSet();
   }

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

   public Collection getEmployees() {
      return employees;
   }

   public String getName() {
```

```
        return name;
    }

    // Implement the setter methods
    public void setName(String newName) {
        name = newName;
    }

    public void setEmployees(Set empSet) {
        this.employees = empSet;
    }

    // Assign employees to the department
    public void addEmployee(Employee emp) {
        if (employees == null)
        employees = new HashSet();
        employees.add(emp);
    }

    // Remove employees from the department
    public void removeEmployee(Employee emp) {
        if (employees != null)
        employees.remove(emp);
    }
}
```

Result

You have implemented the persistence capable classes. Continue by adding the object identity classes [Seite 11].

Defining the Object Identity Classes

The JDO identity guarantees that there is a single JDO instance associated to a persistence manager that represents a particular data store object. The SAP JDO implementation supports application identity only. This means that the identity of a JDO instance is determined by the values of a subset of the persistent fields (the primary key fields) in the persistence capable class. JDO requires the identity of a persistence capable class to be represented by instances of a special object identity class.

For the Employee and the Department persistence capable classes, you have to create object identity classes for the `empId` and `depId` fields respectively. These are the primary key fields of the classes that define the identity of their instances.

Prerequisites

You must have defined the persistence capable classes [Seite 7] Employee and Department.

Procedure

To create object identity classes for Employee and Department classes, you have to follow the JDO requirements and recommendations:

- For a given persistence capable class, the corresponding object identity class is most conveniently defined as a public static inner class `Id` of the persistence capable classes.
- For each primary key field of the persistence capable class, the object identity class has a corresponding public instance field with the same name and type.
- The object identity class has a no-arguments (no-args) constructor similarly to the persistence capable classes.
- The object identity class also has a string constructor. It returns an instance that is equal to an instance returned as a string by the `toString()` method.
- The object identity class has to define `hashCode` and `equals` appropriately.

For the Employee class, extend the source code and add the inner object identity class, which is implemented as follows:

```
static public class Id {

// public field corresponding to the primary key of the PC
class
   public int empId;

   static { // establish the relation: Employee$Id class
            // is the identity class for the PC class
Employee.
      SAPJDOHelper.registerPCClass(Employee.class);
   }

   public Id() { // required: a no-args constructor
   }

   public Id(int empId) {
      this.empId = empId;
   }

   public Id(String string) { // required: a string
constructor
                     // defined as the counterpart of
toString()
      empId = Integer.parseInt(string);
   }

   public int hashCode() { // required: implement
hashCode()
      return empId;
   }

   public String toString() { // required: toString()
defined
                  // as the counterpart of the string
constructor
      return Integer.toString(empId);
   }

   public boolean equals(Object that) { // required: define
equals()
      if (that == null || !(that instanceof Id))
         return false;
      else
```

```
            return empId == ((Id) that).empId;
      }
}
```

For the Department class, the object identity class is coded as follows:

```
static public class Id {

// public field corresponding to the primary key of the PC
class
   public int depId;

   static { // establish the relation: Department$Id class
         // is the identity class for the PC class
Department.
      SAPJDOHelper.registerPCClass(Department.class);
   }

   public Id() { // required: a no-args constructor
   }

   public Id(String string) { // required: a string
constructor
                     // defined as the counterpart of
toString()
      depId = Integer.parseInt(string);
   }

   public Id(int depId) {
      this.depId = depId;
   }

   public int hashCode() { // required: implement
hashCode()
      return depId;
   }

   public String toString() { // required: toString()
defined
                  // as the counterpart of the string
constructor
      return Integer.toString(depId);
   }

   public boolean equals(Object that) { // required: define
equals()
      if (that == null || !(that instanceof Id))
         return false;
      else
         return depId == ((Id) that).depId;
   }
}
```

In order to compile the Employee and the Department classes once you have added the inner ID classes, you must add the following import declaration:

```
import com.sap.jdo.SAPJDOHelper;
```

Result

You can now define the XML metadata [Seite 14] for the JDO.

Defining the JDO Metadata

For each persistence capable class you have to create an XML document – the JDO metadata that declares the persistence properties of the persistence capable class. This document defines the class as persistence capable and declares its persistent and primary key fields. The JDO metadata file must have the extension ".jdo" and must be located in the same directory as the persistence capable class.

Prerequisites

You have created the persistence capable classes [Seite 7] Employee and Department.

Procedure

1. Open the context menu of your project and choose *New* → *File*.
2. Choose *<project_name>/source/temp/persistence/gettingstarted/jdo* as a location where the file is to be saved.
3. Enter **`Employee.jdo`** as the file name.
4. Choose *Finish*. The file is opened automatically for editing.
5. Enter the following in the file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
  <package name="temp.persistence.gettingstarted.jdo">
    <class name="Employee" identity-type="application" objectid-
class="Employee$Id">
      <field name="empId" persistence-modifier="persistent"
primary-key="true"/>
      <field name="firstName"  persistence-
modifier="persistent"/>
      <field name="lastName"   persistence-
modifier="persistent"/>
      <field name="salary"     persistence-
modifier="persistent"/>
      <field name="department" persistence-
modifier="persistent"/>
    </class>
  </package>
</jdo>
```

6. Use the same procedure to create **`Department.jdo`** in folder *<project_name>/source/temp/persistence/gettingstarted/jdo.* Enter the following in the file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
  <package name="temp.persistence.gettingstarted.jdo">
    <class name="Department" identity-type="application"
objectid-class="Department$Id">
      <field name="depId"     persistence-modifier="persistent"
primary-key="true"/>
      <field name="name"      persistence-
modifier="persistent"/>
      <field name="employees" persistence-modifier="persistent">
        <collection element-
type="temp.persistence.gettingstarted.jdo.Employee"/>
      </field>
    </class>
  </package>
</jdo>
```

7. Save and close the files.

Result

Now you have to define the O/R mapping [Seite 15].

Defining the O/R Mapping

For each persistent capable class, you have to create an XML document that defines how the persistent fields of the persistent capable class are mapped to the corresponding database tables. The object/relational (O/R) mapping file must have the extension ".map" and must be located in the same folder as the persistent capable class. The format of the class is defined by the JDO mapping metadata DTD [Extern] (in the Reference Manual). For more information about O/R mapping, see Mapping Persistent Classes to Database Tables [Extern].

Procedure

1. Open the context menu of your project and choose *New* → *File*.
2. Browse to the folder where the Employee class is saved – that is, *<project_name>/source/temp/persistence/gettingstarted/jdo*.
3. Enter **`Employee.map`** as the file name and choose *Finish*.
4. The file opens automatically. Enter the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE map SYSTEM "map.dtd">
<map version="1.0">
  <package name="temp.persistence.gettingstarted.jdo">
    <class name="Employee">
      <field name="empId">
```

```
        <column name="EMPID" table="TMP_EMPLOYEE"/>
      </field>
      <field name="firstName">
        <column name="FIRST_NAME" table="TMP_EMPLOYEE"/>
      </field>
      <field name="lastName">
        <column name="LAST_NAME" table="TMP_EMPLOYEE"/>
      </field>
      <field name="salary">
        <column name="SALARY" table="TMP_EMPLOYEE"/>
      </field>
      <relationship-field name="department"
multiplicity="one">
          <foreign-key name="EMPLOYEE_TO_DEPARTMENT"
               foreign-key-table="TMP_EMPLOYEE"
               primary-key-table="TMP_DEPARTMENT">
            <column-pair foreign-key-column="DEPID"
primary-key-column="DEPID"/>
          </foreign-key>
      </relationship-field>
    </class>
  </package>
</map>
```

5. Repeat the procedure described above to create a file named **`Department.map`** in folder *<project_name>/source/temp/ persistence/gettingstarted/jdo*. Enter the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE map SYSTEM "map.dtd">
<map version="1.0">
  <package name="temp.persistence.gettingstarted.jdo">
    <class name="Department">
      <field name="depId">
        <column name="DEPID" table="TMP_DEPARTMENT"/>
      </field>
      <field name="name">
        <column name="NAME" table="TMP_DEPARTMENT"/>
      </field>
     <relationship-field name="employees"
multiplicity="many" update="false">
         <foreign-key name="EMPLOYEE_TO_DEPARTMENT"
            foreign-key-table="TMP_EMPLOYEE"
            primary-key-table="TMP_DEPARTMENT">
            <column-pair foreign-key-column="DEPID"
primary-key-column="DEPID"/>
         </foreign-key>
       </relationship-field>
    </class>
  </package>
</map>
```

Result

You have created the files defining the O/R mapping for the persistence capable classes.

You can now check the overall model for consistency and enhance the classes [Seite 17].

Running the JDO Enhancer and Checker Tools

The JDO Enhancer is a bytecode modifier of Java class files that turns plain Java classes into persistence capable ones. Although you have defined the Employee and Department classes as persistence capable by including the persistent fields and creating the JDO metadata, the classes still do not comply thoroughly with the requirements because they do not implement the `javax.jdo.spi.PersistenceCapable` interface. Therefore, you have to enhance them using the JDO Enhancer, which adds the necessary methods to the classes based on the information in the metadata XML file.

The process of enhancing the classes also involves performing consistency checks. They ensure that there are no discrepancies between the Java model (described in Defining the Persistence Capable Classes [Seite 7] and Defining the Object Identity Classes [Seite 11]), the JDO model (in Defining the JDO Metadata [Seite 14]), the JDBC types (in Creating the Database Tables [Seite 5]), and the mapping model (in Defining the O/R Mapping [Seite 15]). The JDO Checker is the tool that checks the consistency of the overall model.

Integration

The JDO Enhancer and Checker tools are currently not integrated into the build process of the SAP NetWeaver Developer Studio. Therefore, you have to run the tools manually using a proper build file.

Prerequisites

You must have:

- Created the database tables.
- Created the Employee and Department classes.
- Defined the JDO metadata and the O/R mapping.

Activities

The most convenient way to run the JDO Enhancer is using the ANT build tool that is integrated into the SAP NetWeaver Developer Studio as a plugin.

1. Use the context menu of the *GettingStartedJDOWeb* project to create a new file called *build.xml*. Choose to save the file in the *GettingStartedJDOWeb* main directory.
2. The file opens automatically. Enter the following contents:

```
<project name="GettingStartedWithJDO" default="enhance"
basedir="..">

  <property name="sourceproject.dir"
value="GettingStartedJDOWeb"/>
  <property name="dictproject.dir"
value="GettingStartedPersistenceDic"/>
  <property name="src.dir"
value="${sourceproject.dir}/source"/>
  <property name="bin.dir"
value="${sourceproject.dir}/bin"/>
  <property name="catalog.dir"
value="${dictproject.dir}/gen_ddic/dbtables/"/>
```

```
  <property name="enhancer"
value="com.sap.jdo.enhancer.Main"/>

  <property name="utility"
value="com.sap.jdo.sql.util.JDO"/>
  <property name="tssap" value="C:/Program
Files/SAP/JDT/eclipse/plugins"/>
  <property name="jdo"
value="${tssap}/com.sap.ide.eclipse.ext.libs.jdo/lib/jdo.ja
r"/>
  <property name="xml"
value="${tssap}/com.tssap.sap.libs.xmltoolkit/lib/sapxmltoo
lkit.jar"/>
  <property name="jdoutil"
value="${tssap}/com.sap.jdo.utils/lib/sapjdoutil.jar"/>
  <property name="dictionary"
value="${tssap}/com.sap.dictionary.database/lib/jddi.jar"/>
  <property name="logging"
value="${tssap}/com.tssap.sap.libs.logging/lib/logging.jar"
/>
  <property name="catalogreader"
value="${tssap}/com.sap.opensql/lib/opensqlapi.jar"/>
  <property name="classpath"
value="${jdo};${jdoutil};${xml}"/>
  <property name="classpath.check"
value="${classpath};${dictionary};${logging};${catalogreade
r};${bin.dir}"/>

  <target name="enhance">
    <antcall target="enhance.Employee"/>
    <antcall target="enhance.Department"/>
  </target>

  <target name="check">
    <antcall target="check.Employee"/>
    <antcall target="check.Department"/>
  </target>

  <target name="enhance.Employee">
    <java
      fork="yes"
      failonerror="yes"
      classname="${enhancer}"
      classpath="${classpath}">
      <arg line= "-f -d" />
      <arg value= "${bin.dir}" />
      <arg value=
"${src.dir}/temp/persistence/gettingstarted/jdo/Employee.jd
o" />
      <arg value=
"${bin.dir}/temp/persistence/gettingstarted/jdo/Employee.cl
ass" />
    </java>
  </target>

  <target name="enhance.Department">
    <java
      fork="yes"
      failonerror="yes"
      classname="${enhancer}"
```

```
      classpath="${classpath}">
      <arg line= "-f -d" />
      <arg value= "${bin.dir}" />
      <arg value=
"${src.dir}/temp/persistence/gettingstarted/jdo/Department.
jdo" />
      <arg value=
"${bin.dir}/temp/persistence/gettingstarted/jdo/Department.
class" />
    </java>
  </target>

  <target name="check.Employee">
    <java
      fork="yes"
      failonerror="yes"
      classname="${utility}"
      classpath="${classpath.check}">
      <arg line= "-v -p" />
      <arg value= "${sourceproject.dir}/checker.properties"
/>
       <arg value= "-c" />
      <arg value= "${catalog.dir}" />
      <arg value= "check" />
      <arg value=
"temp/persistence/gettingstarted/jdo/Employee.class" />
    </java>
  </target>

  <target name="check.Department">
    <java
      fork="yes"
      failonerror="yes"
      classname="${utility}"
      classpath="${classpath.check}">
      <arg line= "-v -p" />
      <arg value= "${sourceproject.dir}/checker.properties"
/>
      <arg value= "-c" />
      <arg value= "${catalog.dir}" />
      <arg value= "check" />
      <arg value=
"temp/persistence/gettingstarted/jdo/Department.class" />
    </java>
  </target>

</project>
```

The **tssap** property must point to the installation directory of the SAP NetWeaver Developer Studio.

3. Use the context menu of the *GettingStartedJDOWeb* project to create a new file called *checker.properties*. Choose to save the file in the *GettingStartedJDOWeb* main directory. The file opens automatically. Enter the following contents:

```
com.sap.jdo.sql.mapping.useCatalog=true
```

```
com.sap.jdo.sql.mapping.checkConsistency=true
com.sap.jdo.sql.mapping.checkConsistencyDeep=true
```

4. In the *Java* perspective, open the context menu of the *build.xml* file and choose *Run Ant…*.
5. In the *Targets* tab, choose the option *check*, in addition to the default target *enhance*.
6. Choose *Apply* and then *Run*. The output of the process is printed in the console of the Developer Studio.

Result

The classes now implement the `javax.jdo.spi.PersistenceCapable` interface. The next step is implementing the business logic [Seite 20].

Implementing the Business Logic

According to the J2EE standard, the Web front end should not directly access the persistence layer. Typically, a J2EE application should use a façade session bean for the communication between the Web components and the JDO instances. In this example, the business logic is implemented in a plain Java class for simplicity. The class includes methods that modify the data in the data store using JDO.

Prerequisites

You must have created:

- The GettingStartedJDOWeb project
- The `temp.persistence.gettingstarted.jdo` package
- The persistence capable classes Employee and Department.

Procedure

1. In the *J2EE Development* perspective, choose *GettingStartedJDOWeb* project and open its context menu.
2. Choose *New* → *Java Class*. To choose a package for the class, use *Browse…* next to the *Package* field, and select `temp.persistence.gettingstarted.jdo` from the list.
3. Enter **`BusinessLogic`** as the name of the class. Choose *Finish*.
4. The new Java file is created and opens automatically. Edit its contents as follows:

   a. The PersistenceManagerFactory (PMF) acts as a factory for PersistenceManagers (PMs) that establish the actual connection to the underlying data store. Therefore, the first thing a JDO application typically does is to acquire a PMF using a JNDI lookup operation.

   In the beginning of the class, define a global variable for the PMF reference. Then in the constructor of the class, look up the PMF from the local environment `java:comp/env` using the JNDI name of the instance `jdo/defaultPMF`. Cast

the object to `javax.jdo.PersistenceManagerFactory` and assign the value to the variable that you have declared.

```
public class BusinessLogic {

   private PersistenceManagerFactory pmf; // Persistence
Manager Factory (PMF)

   public BusinessLogic()
   throws NamingException {

   // Get the Persistence Manager Factory via JNDI using a
reference.
   // (You have to declare this reference within the
deployment descriptor.)
      Context ctx = new InitialContext();
      pmf = (PersistenceManagerFactory)
ctx.lookup("java:comp/env/jdo/defaultPMF");
   }
...
```

b. Implement a method that creates a new department by name and ID. The method must first create a new instance of the Department class, which is transient – that is, it is not stored in the database. To make it persistent, you must get a PersistenceManager instance and invoke its `makePersistent()` method with the new JDO instance as a parameter. Note that the example uses the `javax.jdo.Transaction` interface for transaction demarcation. Therefore, you must first get a reference to a PM, then start a transaction, and then invoke the `makePersistent()` method. In the end of the method, you must complete the transaction and close the PM.

```
public void createDepartment(
   int depId,
   String name)
   throws JDOException {

   PersistenceManager pm = null;
   try {
      Department dep = new Department(depId);
      dep.setName(name);

      // Get the Persistence Manager (PM)
      // PM manages transactions, the life cycle of
objects, etc.
      pm = pmf.getPersistenceManager();

      // Start a local transaction.
      Transaction tx = pm.currentTransaction();
      tx.begin();

      // Create a new persistent department object
      pm.makePersistent(dep);
      // Insert the new department to the database
      tx.commit();
   }
```

```
   finally {
      // Close the Persistence Manager
      if (pm != null && !pm.isClosed()) {
      if (pm.currentTransaction().isActive())
         pm.currentTransaction().rollback();
         pm.close();
      }
   }
}
```

c. Implement a method that creates a new employee in a given department. The logic is the same as in the `createDepartment()` method:

```
public void createEmployee(
   int empId,
   String fName,
   String lName,
   BigDecimal salary,
   int depId)
   throws JDOException {
   PersistenceManager pm = null;
   try {
      Employee emp = new Employee(empId);
      emp.setFirstName(fName);
      emp.setLastName(lName);
      emp.setSalary(salary);

// Obtain the related department object. We assume the
department exists.
      pm = pmf.getPersistenceManager();
      Department department =
(Department)pm.getObjectById(new Department.Id(depId),
false);

      // Start a local transaction
      Transaction tx = pm.currentTransaction();
      tx.begin();

      // Establish a relationship between the employee and
the department
      emp.setDepartment(department);
      department.addEmployee(emp);

      // Create a new persistent employee object
      pm.makePersistent(emp);

      // Write the new employee to the database. Update the
department object.
      tx.commit();
   }
   finally {
      // Close the Persistence Manager
      if (pm != null && !pm.isClosed()) {
         if (pm.currentTransaction().isActive())
            pm.currentTransaction().rollback();
         pm.close();
      }
   }
```

```
}
```

d. Implement a method that retrieves the data about all employees in a department. The method uses a query with a parameter, which defines the ID of the department. The query returns a collection, which is then transformed into an array of employee records. Since the method does not involve data modification, you do not need to execute the query in a transaction:

```
public Employee[] getEmployeesFromDepartment(
   int departmentId)
   throws JDOException {

   PersistenceManager pm = null;
   Employee[] result = new Employee[] {};
   try {
      // Prepare the query.
      pm = pmf.getPersistenceManager();
      Query query = pm.newQuery(Employee.class,
"department.depId == id");
      query.declareParameters( "int id" );

      // Cast the return value of query.execute() to
java.util.Collection
      Collection col = (Collection)query.execute(new
Integer(departmentId));

      // convert the collection to an array. Return the
array.
      // The method returns Employee PC objects for
simplicity
      // It is recommended to create additional plain Java
      // objects (Data Transfer Objects DTO) as the return
      //objects
      pm.retrieveAll(col);
      pm.makeTransientAll(col);
      result = (Employee[]) col.toArray(new Employee[] {});

      //free the resources
      query.close(col);
   }
   finally {
      // Close the Persistence Manager
      if (pm != null && !pm.isClosed())
            pm.close();
   }
   return (result);
}
```

e. To add the required imports, position the cursor anywhere in the Java editor and open the context menu. Choose *Source* → *Organize Imports*.

f. Select `javax.naming.Context` and confirm by choosing *Finish*. The following import statements are added after the package declaration:

```
import java.math.BigDecimal;
```

```
import java.util.Collection;

import javax.jdo.JDOException;
import javax.jdo.PersistenceManager;
import javax.jdo.PersistenceManagerFactory;
import javax.jdo.Query;
import javax.jdo.Transaction;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
```

g. Save and close the file.

Result

You have implemented the business logic of the example. The next step is to implement the Web front end [Seite 24].

Developing the Web Front End

Use

The presentation logic in this example is implemented by a simple HTML page and a servlet, which invokes the methods of the BusinessLogic [Seite 20] class and shows the result to the user.

Prerequisites

You must have created the GettingStartedJDOWeb project [Seite 7].

You must have implemented the persistence [Seite 7] and the business [Seite 20] tiers of the application.

Procedure

Creating the HTML Page

1. In the J2EE Development perspective, open the context menu of the *GettingStartedJDOWeb* project.
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:
   a. Creating a new department
   b. Creating a new employee
   c. Selecting the employees from a department with a given ID

   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 Java Data Objects (JDO)</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>
      <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, choose *GettingStartedJDOWeb* project and open its context menu. Choose *New* → *Package*.
2. Enter **`temp.persistence.gettingstarted.web`** as the name of the package. Choose *Finish*.
3. From the context menu of *GettingStartedJDOWeb* project choose *New* → *Servlet*.
4. Enter **`ProcessInput`** as the servlet name. Choose *HTTPServlet* for the servlet type. To choose a package, use *Browse…* next to the *Package* field.
5. Select `temp.persistence.gettingstarted.web` from the list. Confirm the settings by choosing *Finish*.
6. The servlet class is created and opens automatically. Edit the contents of the file as follows:
   a. Implement the `doGet()` method. It must parse the input from the HTML page and invoke the relevant method of the `BusinessLogic` class:

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

   response.setContentType("text/html");
   PrintWriter out = response.getWriter();
   try {
      // the plain Java class BusinessLogic.java
      // encapsulates the business logic:
      BusinessLogic businessLogic = new BusinessLogic();
      out.println("<html><body>");
      out.println("<h4>Using Java Data Objects
```

```
(JDO)</h4>");

      // read the user input
      // invoke the appropriate business logic
functionality
      // generate the response (html) page
      String action = request.getParameter("ACTION");
      if (action.equals("NEW_DEP")) {
         int depId =
Integer.parseInt(request.getParameter("DEPID"));
         String depName = request.getParameter("NAME");
         businessLogic.createDepartment(depId, depName);
         out.println("Department \"" + depName + "\"
created.");

      } else if (action.equals("NEW_EMP")) {
         int depId =
Integer.parseInt(request.getParameter("DEPID"));
         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"));

         businessLogic.createEmployee(empId, firstName,
lastName, salary, depId);

         out.println("Employee \"" + firstName + " " +
lastName + "\" created.");

      } else if (action.equals("LIST")) {

         int depId =
Integer.parseInt(request.getParameter("DEPID"));

         Employee[] emps =
businessLogic.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>" + depId + "</td>");
            out.println("</tr>");
         }
            out.println("</table>");
         }

      } else {
         out.println("Illegal action: " + action);
      }

   } catch (Throwable ex) { // catches any exception thrown

      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. It must invoke `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.jdo.BusinessLogic;
import temp.persistence.gettingstarted.jdo.Employee;
```

7. Save and close the file.

Result

You have developed all components of the application. Now go on with its assembly [Seite 29].

Assembling the Application

Use

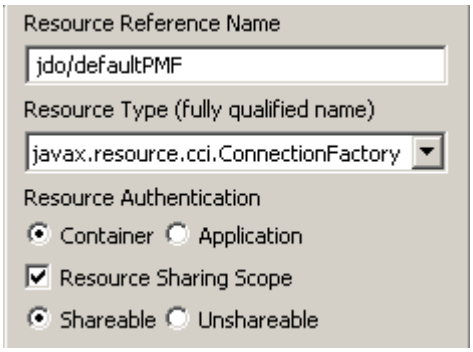
To prepare your application for deployment, you have to:

- Build the WAR file
- Create an Enterprise Application Project and assemble the EAR file

Procedure

Building the WAR

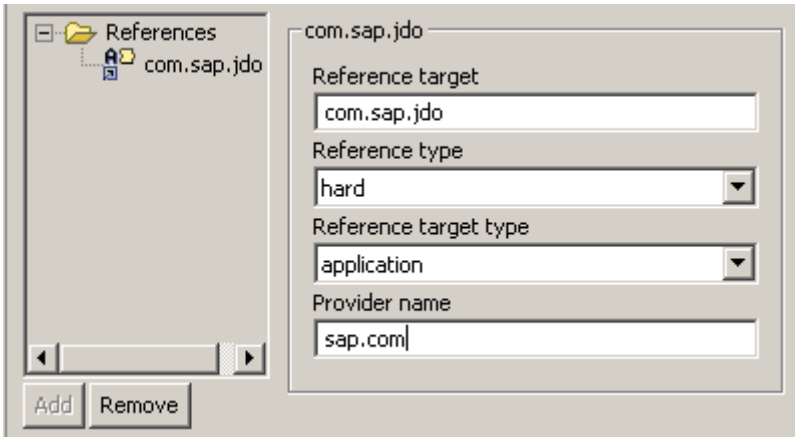
1. In the *J2EE Development* perspective, extend the *GettingStartedJDOWeb* node and open *web.xml*.
2. Go to the *Mapping* tab and choose *Add* under the *Servlet mappings* field. Add the `ProcessInput` servlet and enter **`ProcessInput`** in the *URL Pattern* field.
3. Go to the *Resource* tab. Choose *Resource entries* and then *Add*. Enter the following values:



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

Assembling the EAR

1. Choose *File* → *New* → *Project*. Select *J2EE* in the left-hand pane, and *Enterprise Application Project* in the right-hand pane.
2. Choose *Next*.
3. Enter a name for the project – for example, *GettingStartedJDOEar*. Choose *Next*.
4. In the *Referenced Projects* field, choose *GettingStartedJDOWeb* and then choose *Finish*.
5. Extend the *GettingStartedJDOEar* node and open *application.xml*.
6. On the *General* tab, enter *GettingStartedJDO* as a display name. On the *Modules* tab enter *gettingstarted-jdo* as a context root.
7. Save and close the XML file.
8. Open *application-j2ee-engine.xml*.
9. On the *General* tab, select *References* and choose *Add* → *Create new*.
10. Enter the following values:



11. Save and close the XML file.
12. From the context menu of the *GettingStartedJDOEar* project, choose *Build EAR File*. The system informs you if the process finished successfully.

Result

Now you can deploy and run the application [Seite 30].

Deploying and Running the Application

The last step in the development of the 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 Relational Persistence [Extern]. If you have already tried the relational persistence example, and have deployed the tables, you can skip this step.
2. Deploy the application EAR:

a. In the *J2EE Development* perspective, extend the *GettingStartedJDOEar* project node.

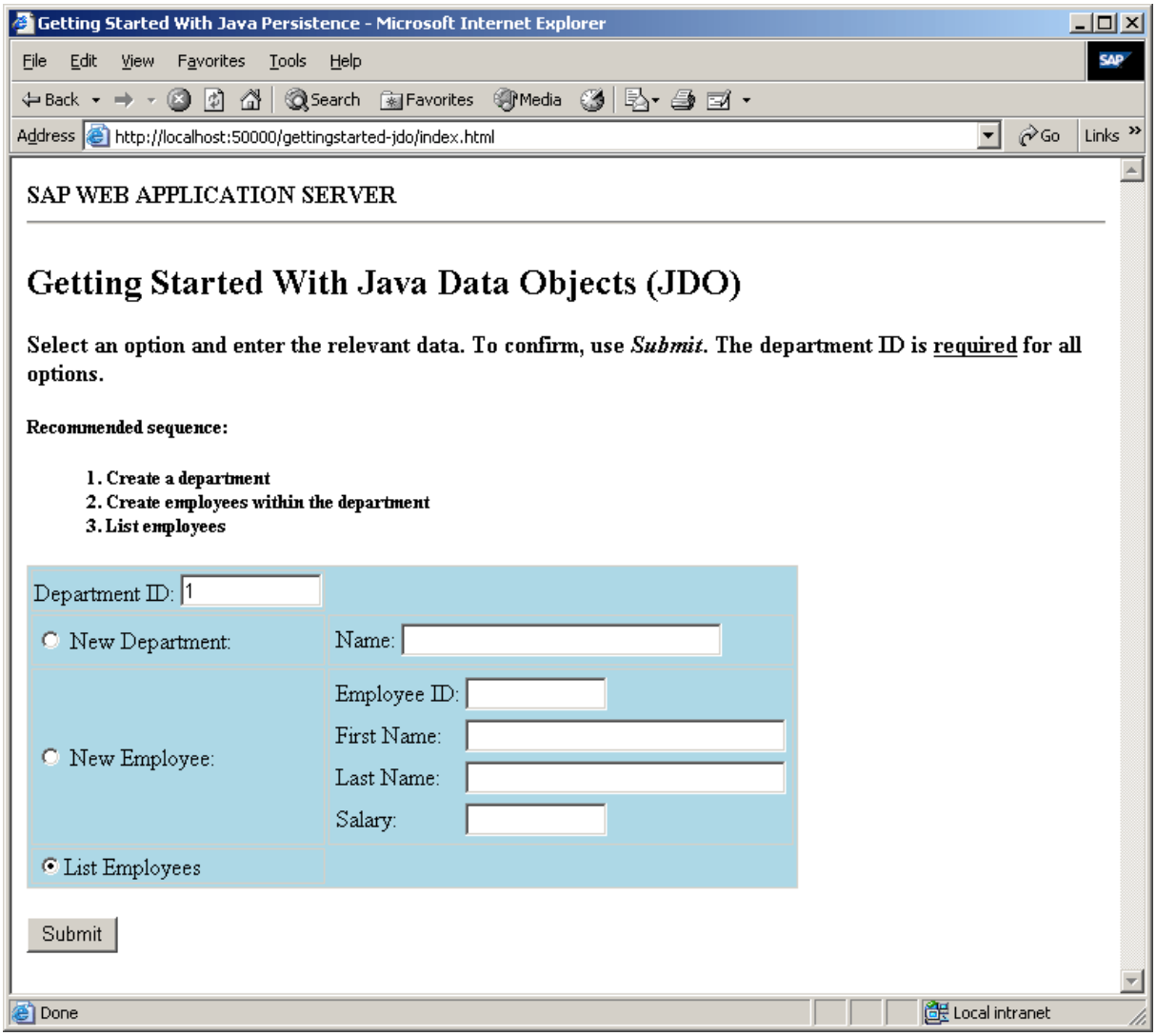
b. Open the context menu of *GettingStartedJDOEar.ear* and choose *Deploy to J2EE engine*.

c. The application starts automatically after it is deployed.

3. Test the application:

a. Invoke the application in your browser:
`http://<yourhost>:<HTTP port>/<Context Root>/`

`http://localhost:50000/gettingstarted-jdo/`



b. Create a department:

i. Enter a *Department ID*, such as 1.

ii. Choose *New Department* and enter a name.

iii. 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 *Submit*.

d. List all employees in a department.

i. Enter the ID of an existing department.

ii. Choose *List Employees*.

iii. Choose *Submit*.