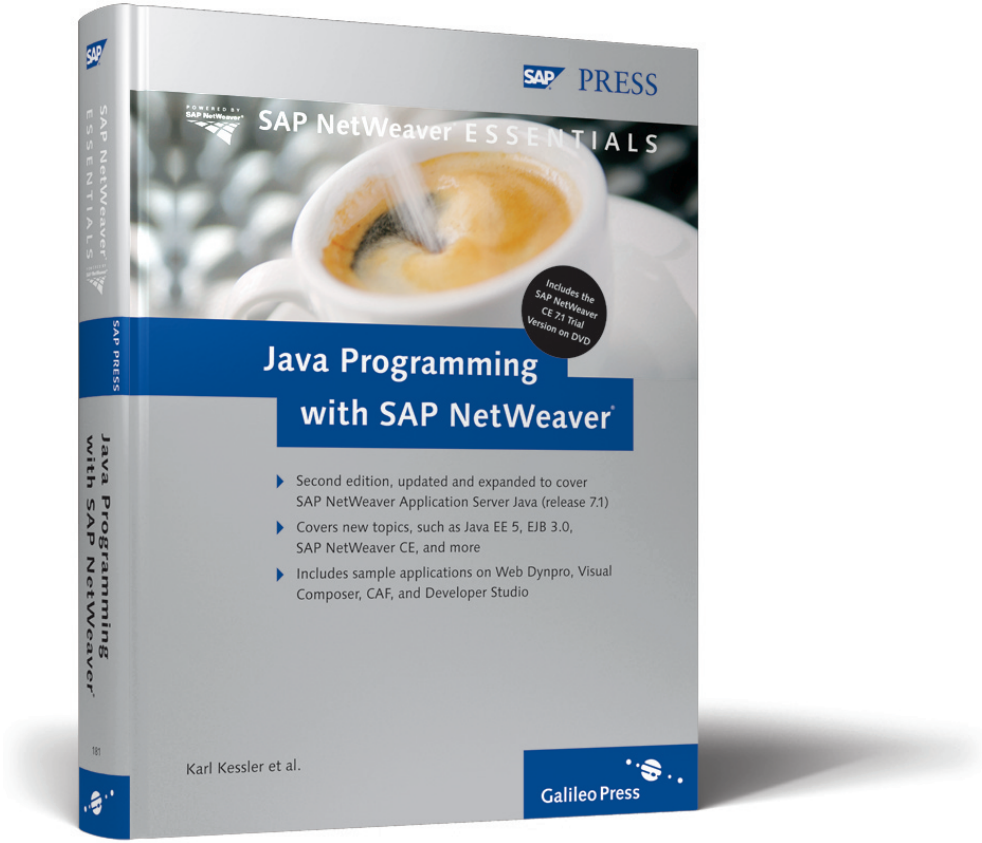
Alfred Barzewski, Carsten Bönnen, Bertram Ganz,
Wolf Hengevoss, Karl Kessler, Markus Küfer,
Anne Lanfermann, Miroslav Petrov, Susanne Rothaug,
Oliver Stiefbold, Volker Stiehl

Java Programming with SAP NetWeaver®



Galileo Press

Bonn • Boston

Contents at a Glance

Contents

Preface to the Second Edition

Content

Two years have passed since the first edition of *Java Programming with the SAP Web Application Server* was published. While the first edition described SAP NetWeaver 2004 and SAP NetWeaver 7.0 (equivalent to SAP NetWeaver 2004s) a complete revision became necessary due to the market introduction of SAP NetWeaver Composition Environment 7.1:

- On the one hand, many programming techniques such as Web Dynpro Java or the SAP NetWeaver Developer Studio have undergone major changes. SAP NetWeaver 7.1 was the first enterprise platform to support the Java EE 5 standard that demonstrates the high speed of innovation of SAP NetWeaver. All aspects are covered thoroughly in this new edition.
- On the other hand, the positioning of SAP NetWeaver as a technology platform has evolved based on its strong market adoption. You can derive this from how SAP NetWeaver is used today: as a foundation for SAP's solutions such as SAP ERP and the SAP Business Suite on one side, on the other side as integration und composition platform for the Enterprise Service-Oriented Architecture (enterprise SOA) of SAP.

Because of this, new chapters have been added that introduce the composition technologies. Of great importance in this context is the interoperability between the different releases of SAP NetWeaver (7.0 und 7.1), including their varying speeds in terms of innovation. We still involved experienced authors of the different topic areas for the second edition as well.

Structure

The presentation starts in Chapter 1, *SAP NetWeaver*, with the positioning of SAP NetWeaver as platform for enterprise SOA as well as an introduction of the major SAP NetWeaver capabilities.

In Chapter 2, *Overview of the SAP NetWeaver Developer Studio*, the focus is on Developer Studio. The Java EE 5 programming model, the development, and the deployment of a sample application is shown in Chap-

ter 3, *SAP NetWeaver Developer Studio — Step-by-Step to a Sample Application*. The focus here is not so much on a complete discussion of the Java EE 5 programming model (there are plenty of publications out there), but how the Java EE 5 model is supported by the many perspectives of the Developer Studio.

In Chapter 4, *Java Persistence*, the different approaches to Java persistence supported by SAP that are based on Enterprise JavaBeans 3.0 and the Java Persistence API are introduced. Chapter 5, *Web Services and Enterprise Services in the SAP NetWeaver Composition Environment*, leads into the world of enterprise SOA, based on standard Web service technology, and describes how to develop applications that consume Enterprise Services.

Chapter 6, *Developing Business Applications with Web Dynpro*, is dedicated to Web Dynpro because of the importance of the user interface. The Portal integration of Web Dynpro applications is described in Chapter 7, *Running Web Dynpro Applications in SAP NetWeaver Portal*. The Visual Composer as a tool for model-driven UI development is presented in Chapter 8, *SAP NetWeaver Visual Composer*. Further techniques for the creation of Composite Applications are discussed in Chapter 9, *Developing Composite Applications*.

The Java development process and the development infrastructure offered by SAP comprise three chapters. In Chapter 10, *SAP NetWeaver Development Infrastructure and* the *Component Model — Concepts*, the fundamental component model and the basic elements of the infrastructure are presented. Chapter 11, *SAP NetWeaver Development Infrastructure — Configuration and Administration*, explains the setup and administration of the Java Development Infrastructure. The Java EE 5 sample from Chapter 3 is revisited in Chapter 12, *SAP NetWeaver Development Infrastructure — Developing an Example Application Step-by-Step*, in order to demonstrate the development infrastructure.

In Chapter 13, *SAP NetWeaver Application Server Java — Architecture*, the architecture, scalability, and robustness of SAP NetWeaver Application Server 7.1, based on SAP's Java Virtual Machine, are discussed. The presentation concludes with Chapter 14, *Supportability of the SAP NetWeaver*

Composition Environment, which presents supportability aspects that are critical for the successful operation of applications.

DVD content

On the trial DVD, you will find a test and evaluation version of SAP NetWeaver Composition Environment 7.1, including SAP NetWeaver Developer Studio. The samples that are discussed in the various chapters are stored on the DVD as well. You will find details about installation and configuration on the start page of the DVD that is displayed automatically when you insert the DVD into the drive.

Acknowledgments

At this point, I would like to thank the authors. Without their passion the second edition would not have been possible: Alfred Barzewski for introduction of SAP NetWeaver Developer Studio (Chapter 2) and the basic Java EE 5 sample application (Chapter 3); Markus Küfer for the presentation of Java Persistence (Chapter 4); Susanne Rothaug und Anne Lanfermann for introduction and consumption of Enterprise Services (Chapter 5); Bertram Ganz for presentation of Web Dynpro Java (Chapter 6); Oliver Stiefbold for the creation of the DVD trial version of SAP NetWeaver Composition Environment as well as for the chapter on Portal integration (Chapter 7); Carsten Bönnen for his contribution on Visual Composer (Chapter 8); Volker Stiehl for the introduction to the development of Composite Applications (Chapter 9); Wolf Hengevoss for the presentation of SAP NetWeaver Development Infrastructure (Chapters 10, 11, and 12); and finally Miroslav Petrov for the overview of supportability (Chapter 14). You will find the bios of the authors at the end of the book. The chapters on the positioning of SAP NetWeaver (Chapter 1) and the presentation of the server architecture (Chapter 13) fall under my responsibility. Special thanks go to the translation team at SAP AG who created the English version: Paul Smith, Neil Matheson, Susan Want, Michèle Coghlan, and Abigail Haley. Last but not least, I would like to thank Stefan Proksch from SAP PRESS for his ongoing support and advice during the project.

Karl Kessler
Vice President, Product Management SAP NetWeaver

Using a concrete example, this chapter will introduce you to the practical side of working with the SAP NetWeaver Developer Studio. On this guided tour, you will set up — step-by-step — a simple employee application using the Java EE 5 standard. The ultimate aim is to then deploy and execute the application on the SAP NetWeaver Application Server. You will have the opportunity of getting to know the close interaction between different tools of the development environment.

3 SAP NetWeaver Developer Studio — Step-by-Step to a Sample Application

Prerequisites

You will get optimum use out of this chapter if you are very familiar with the Java programming language and, in addition, already have experience with using the Java EE 5 programming model. To be able to reconstruct the steps in a practical way, you need the SAP NetWeaver Developer Studio and access to the SAP NetWeaver Application Server Java. The SAP NetWeaver Composition Environment 7.1, on the DVD of this book, is suitable for this purpose. It is best if you install this version before you start with the hands-on exercises.

Goals

The tutorial application, which you will develop step-by-step, is focused more on didactic aspects than on any endeavor to implement a realistic application scenario. Therefore, you need neither a bank application nor a complex warehouse scenario. Rather, it is our intention to introduce to you, with the help of a straightforward example, the options that the Developer Studio provides as a development environment for enterprise applications on the basis of established Java standards. In the foreground, therefore, you have the interaction between different toolsets, and the linking up of services that efficiently support the development process and the daily work of the developer.

You can view this chapter as an introduction to working with the Developer Studio. After processing all the steps of this chapter, you will be able to organize the basic processes and development steps (UI and EJB development, layout of the data model, etc.) within the framework of the Java EE standard development using suitable tools. You will also be able to map the tasks to the appropriate project types and corresponding development objects.

Local development process

All the steps are described solely from the viewpoint of a local development process. The project resources are created and managed exclusively on the local hard drive. The SAP component model is not used in the tutorial application. The projects concerned are not development components, unlike the scenario based on the use of the SAP NetWeaver Development Infrastructure. However, in Chapter 9, *Developing Composite Applications*, you will learn how to migrate this tutorial application in the NWDI context onto the SAP component model and also migrate it using the corresponding services.

3.1 Employee Tutorial Application

The tutorial application uses a simplified employee data model and should enable the user to create new employee data records and to print data on existing employees. In the application architecture, we make a distinction between clearly defined layers — for example, the presentation layer, the business logic layer, the data retrieval layer, and the persistence layer. Actually, this would not be as absolutely necessary for such a simple case as the one here. Nonetheless, you should familiarize yourself from the beginning with the typical architecture of business applications. In particular, you will get a first impression of how this architecture is mirrored in the development process and how the developer is supported with the organization of his projects through the Developer Studio.

Architecture of the Tutorial Application

While developing the user interface, you access the UI technology called JavaServer Pages (JSP), which has established itself within the standard

Web applications. With the help of a simple example, you will see how you can set up a simple interface and also access the server components underneath it.

The business logic is based on Enterprise JavaBeans 3.0 and is limited to one single, stateless session bean. With the session bean, we can formally distinguish between the business interface and session bean implementation. All the business methods of the session bean are linked to a corresponding business interface so that JavaServer Pages can access the session bean with the help of this interface. In addition, the session bean encapsulates the respective accesses to the persistence layer API.

You model the business data using a single entity that is used both in the business logic and the presentation layer. Because entities are regular Java objects, they can also be used for the data transport to the presentation layer. Corresponding data transfer objects are thus not required. In this connection, the entity is detached from the current transaction context. This is clearly shown in Figure 3.1 by the dotted border.

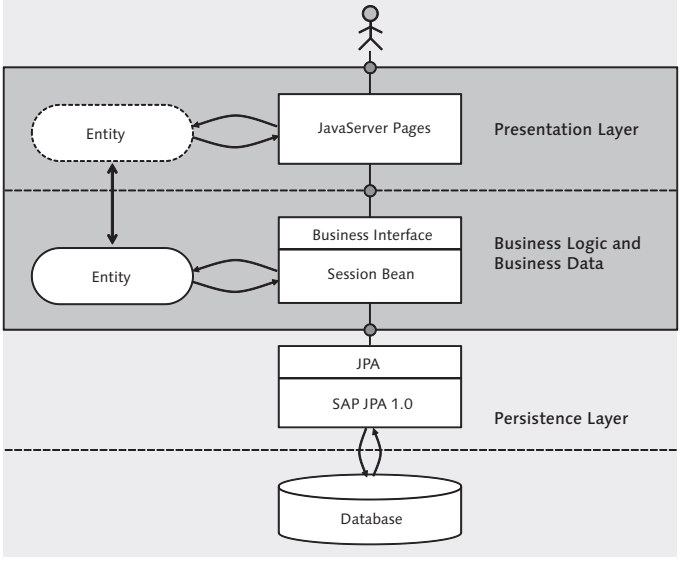


Figure 3.1 Typical Architecture of a Java EE 5 Business Application

Business applications generally cannot do without keeping data persistent in a database. With Java EE 5, a new object-relational persistent

framework — the Java Persistence API (JPA) — has been introduced as part of the Java EE standard. This type of framework has, essentially, the following tasks: ensuring mapping of Java objects onto the relational database; translating various queries as well as changes to Java objects into suitable SQL statements; and, finally, taking care of the entire communication with the database.

As shown in Figure 3.1, the current SAP NetWeaver Application Server contains the actual JPA implementation with the name SAP JPA 1.0. The JPA, however, does not supply the required database tables or table definitions onto which the respective entities are mapped. Instead, it assumes that these tables already exist. You will provide the required tables with the help of the Java Dictionary. Using Open JDBC, you can create the actual database objects in the assigned database schema using the table definitions.

Project View of Tutorial Application

You will begin the development of the tutorial application by first creating the basic data model. In this process, you create a database-independent table definition using the Java Dictionary. Starting from a Dictionary project, you create an SDA archive (Software Delivery Archive) and deploy it on the application server. After this step, the table is physically available on the database.

For access to data records, use JPA entities. The implementation of the business logic for the application (creating new employees, displaying employee data) is taken over by an EJB 3.0 Stateless Session Bean. In this case, the EJB module project in the Developer Studio serves as a container for all enterprise JavaBeans, including the entity, as well as for all further resources, such as the corresponding configuration files and deployment descriptors.

For the implementation of the Web client, a simple interface is provided with the help of JavaServer pages. This should also be able to pass the data to the session bean. All Web resources are managed in a separate project — the Web module project — together with the appropriate configuration files.

In an enterprise application project, you then bring all the resources together to a type of Java EE 5 overall application. You need to deploy the resulting archive (EAR) first before you can call the employee application for the first time. Figure 3.2 groups the basic activities together and depicts the organization of the most important development projects in the respective project types of the Developer Studio.

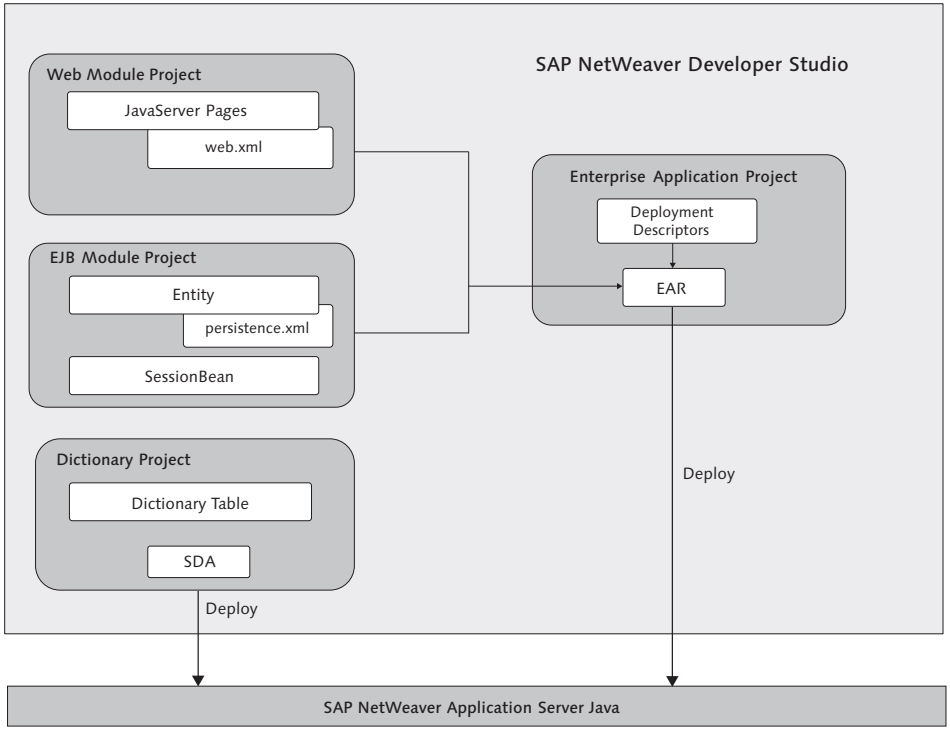


Figure 3.2 Organization of the Development Objects of the Employee Application in the Developer Studio

3.2 First Steps

Start parameters

To start the Developer Studio, the activated platform runtime requires, in addition to access to a Java Virtual Machine (VM), a path specification for storing all the metadata for project information and user-specific settings. A standard Java VM is normally assigned during installation of the Developer Studios and entered as the start parameter.

When you start[1] the Developer Studio for the first time after installation has been completed, you must generally specify the default workspace. The start process will then be interrupted and the system displays a dialog box for selecting the workspace directory. You will then either accept the default value or choose a different directory for the default workspace in order to continue the startup process. When you start the Developer Studio again, the assigned workspace will be used. The start process will then be performed without interruption.

When called up for the first time, the development environment displays a greeting page that looks similar to the one in Figure 3.3. You can consider this page as the starting point for your development activities that will supply you with tutorials, example and reference applications, and selected links to documentation and other information material.

Figure 3.3 SAP NetWeaver Developer Studio After First Call — Greeting Page

1 In general, you start the Developer Studio using the desktop shortcut or from the Microsoft® Windows® Start menu. One alternative and very flexible option is if you use batch files. Even several batch files can be used as configuration files to start the Developer Studio, depending on requirements, using different parameters.

Settings under Windows Preferences

At this point we recommend that you familiarize yourself with the standard settings of the Developer Studio and that you add more entries, where required. You can reach the preferences page through the menu path **Windows • Preferences**. When you are working through the steps in this chapter, you will need the link to the Java application server. Therefore, you should have a corresponding entry set under **SAP AS Java**. We will look at other settings that you require for being able to use the Java development infrastructure in Chapter 11, *SAP NetWeaver Development Infrastructure — Configuration and Administration*.

3.3 Defining the Data Model

Before you develop the employee application, you must first define a suitable data model that will serve as the basis for this application. For didactic reasons, however, no great emphasis is placed on a sophisticated data model with a large number of complex tables and relationships to one another. Instead, the data model should be kept relatively simple so that you can manage with a single table that takes on the management of persistent employee data.

In this first practical step, you will create a new table in the Java Dictionary and add the required columns in the corresponding editor. Afterward, you will create an appropriate archive for this table definition. From the Developer Studio, you are then in the position to deploy this archive on the application server. This way you ensure that the table definition, which is initially available only on a local basis, is converted into physical representation on the database instance.

3.3.1 Creating a Dictionary Project

To create tables, you first need a suitable project in the Developer Studio. Dictionary projects are intended precisely for this purpose. These are projects that serve, at design time, as containers both for Dictionary data types and structures as well as for tables or views in tables. You can create an initial project framework for the new Dictionary project using a wizard.

New project wizard

1. You start the creation wizard through the menu path **File** • **New** • **Project**. In the Wizard window you now see, select the category **Dictionary** and then the entry **Dictionary Project** (Figure 3.4). To get to the next dialog step, choose **Next**.

Figure 3.4 Selection of Dictionary Project in New Project Wizard

2. In the displayed wizard window, you will be prompted to assign general project properties. For this purpose, enter the name "EmployeeDic" for the Dictionary project in the corresponding input field, but leave the standard settings for **Project contents** and **Project language** unchanged (Figure 3.5).

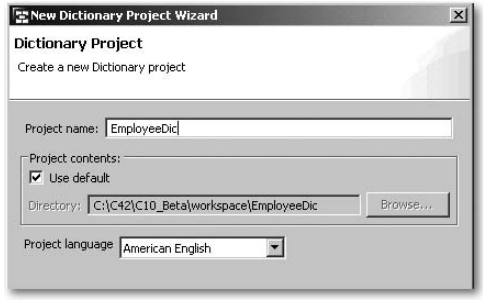


Figure 3.5 General Specifications for Dictionary Project

3. Now you only need to choose **Finish** and leave the rest of the work to the creation wizard. This generates a standard structure for the new

Dictionary project and creates the project folder with the name *EmployeeDic* in the assigned workspace directory. If you now open the Dictionary perspective, a project node with the same name can be seen in the Dictionary Explorer.

In the same manner, it is possible to create, in the Developer Studio, other project types such as Web Dynpro projects, for example, or the different Java EE project types using a suitable wizard.

3.3.2 Defining an Employee Table

In the next step, you create a table for the employee table as part of the project you have just created and then enter the required table fields as columns.

1. To create a table, it is best if you display the project **EmployeeDic** in the Dictionary Explorer. There you can expand the project structure and open the context menu for the node **Database Tables**.
2. To start the creation wizard, simply choose the menu path **Create Table** from the context menu (Figure 3.6). In the displayed dialog box, you will be prompted to assign a name for the table.

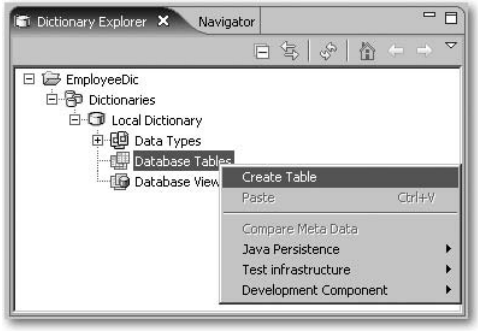


Figure 3.6 Creating a Table in the Dictionary Project

Name conventions for database objects

Keep in mind that, as a rule, a standard prefix is already provided for the table name in the input field. As you can see, this prefix is derived from the default setting that is entered for the Dictionary objects under **Windows • Preferences • Dictionary • Name Server Prefix**. This name prefix is based on the naming convention for database tables

and enables you to uniquely separate development objects that are created at customer sites, partner sites, and at SAP — with the aim of avoiding name conflicts.[2] The two namespaces `TMP_*` and `TEST_*` are of special importance here. These can be used for test objects and prototypes.

3. In this current example, therefore, it will suffice if you use the name prefix "TMP". For the suffix itself, enter the name "EMPLOYEES" and choose **Finish**.
4. As a result, there is a corresponding entry for the new table in the project structure under the node **Database Tables**. By double-clicking the table name, you start the table editor and can now add the individual table fields.

Table fields

5. The first field should have the name "ID". Enter it under **Column Name** in the first line of the table matrix. Because this table field is the primary key of the table, check the field **Key**. Under **Built-In Type**, choose the data type `long` and enter a short description "Employees ID" under **Description**. In the standard version, the property **Not Null** is set for each new field and you use the option of defining initial values for each field of the database table.
6. The second table field contains the name "LAST_NAME". In addition, a **String** of length **30** is assigned as data type[3] to this field as well as the short text "Employees last name".
7. Additional table fields include `FIRST_NAME` and `DEPARTMENT` and `VERSION`. You can see how these are defined in Figure 3.7. Finally, save the current status of the table definition using the appropriate icon in the toolbar.

Now the basic properties of the employee table are set. However, we would like to point out an important general aspect here: You will learn how to set up an index for a table column and how you can activate the table buffering in the table editor. It is a good idea to follow the basic principle: Make as many decisions as possible already at design time!

2 Under *http://service.sap.com/namespaces*, customers and partners of SAP can reserve a name prefix for database objects.

3 From the specifications for the **Built-in Type** and **Length**, you get the assignment to the JDBC Type. This is automatically converted by the wizard.

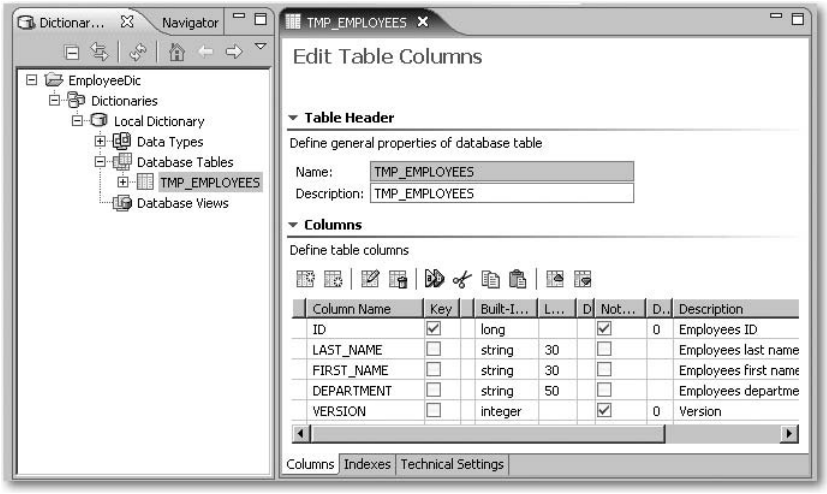


Figure 3.7 Definition of Columns for Table TMP_EMPLOYEES in the Table Editor

Secondary index

Generally speaking, there is a distinction between the primary index and the secondary index for tables — and you will use a secondary index. The primary index is sorted by the key fields of the table and automatically created together with the physical table on the database. Normally, data records are sorted by the value in the primary key. However, if you expect to have frequent access in the application to another field in data records, we recommend that you set up a secondary index for this field.

1. To create, for example, an index to the field `LASTNAME`, simply click the tab **Indexes** in the table editor and then choose the plus character icon on the left in the toolbar.
2. In the displayed wizard, enter "EMPLOYEES_I1" as a suffix for the index name[4] and complete this step with **Finish**. Afterward, expand the tree structure you have just created for the new index and choose the option **Add/Edit Index Fields** from the context menu of the **Fields** node.
3. You now get a list of the table fields and you can choose the field you require (Figure 3.8).

4 Similar to the table name, the standard prefix flows into the index name. Just like table names, index names are limited to 18 characters.

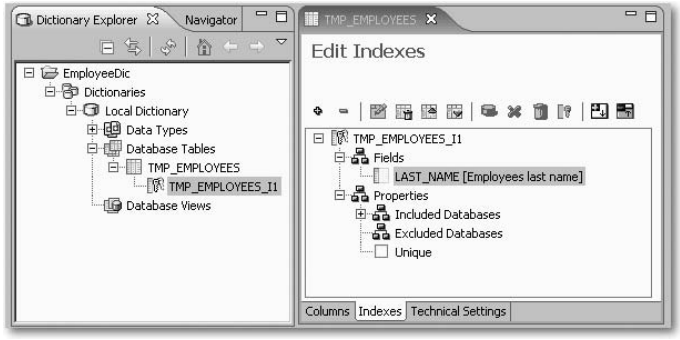


Figure 3.8 Definition of an Index in the Table Editor

Technical settings

4. To activate a table buffer, too, you only require a couple of mouse clicks. Simply choose the tab **Technical Settings** in the table editor, select the respective checkbox, and assign the buffer granularity,[5] as displayed in Figure 3.9.

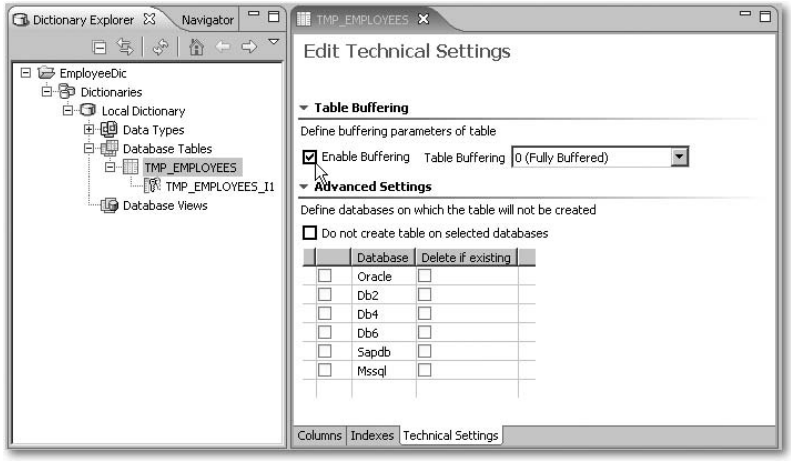


Figure 3.9 Activating the Table Buffer in the Table Editor

5. In the course of the previous procedure, certain table definition data was generated for this project. To save the entire result of your efforts so far, choose the appropriate icon in the toolbar.

5 With the granularity function, you can define whether the table is to be loaded with all data records (fully buffered) or only partially loaded into the buffer as soon as the first data record is accessed.

In this way, the table is completely defined and exists as a local project resource in the form of an XML file. A further result is that the table is now part of the Java Dictionary and has a database-independent definition.

3.4 Implementing Access to Table Data

Java persistence API

At this point, you need to decide how you wish to perform access to table data records in a Java application. Generally speaking, there are several options for data persistence within the framework of Java development and all have their special aspects and strengths. Because the SAP NetWeaver Application Server Java already supports the newer version Java EE 5-Standard, you will use the Java persistence API (JPA) in the current tutorial application. A discussion of the various persistence records in the AS Java context is provided in Chapter 4, *Java Persistence*. This chapter is concerned solely with the topic of persistence.

The JPA is the new object-relational persistence API for Java EE 5 and implemented as an integral part of the Java EE standard. With this technical solution that is extremely easy for the programmer to use, the "lightweight" Java objects, also called entities, are mapped onto relational database tables. Entities are based on regular Java objects, often called POJOs (Plain Old Java Object), and do not have to implement special interfaces or enhance special classes. In addition to the typical class implementation, however, you will also have to provide mapping to suitable database tables as well as mapping of persistent attributes to the respective table fields.[6] For specifying this type of metadata, the JPA provides the comfortable use of annotations that can be added — either manually or using OR mapping tools — to the source code of the entity class.

6 Within the framework of the pending delivery of the SAP NetWeaver Composition Environment, you can find very easy-to-use solutions based on close integration between the individual tools and frameworks. Accordingly, it should be possible, using the table definitions from the Dictionary project, to generate entities for the EJB module, and vice versa.

3.4.1 EJB-Creating a Module Project

To create entities, you first need a new EJB module project.

1. For this purpose, once again start the **New Project Wizard** through the menu path **File • New • Project**. As seen in Figure 3.10, select the category **EJB • EJB 3.0** and then **EJB Project 3.0** in the displayed wizard window.

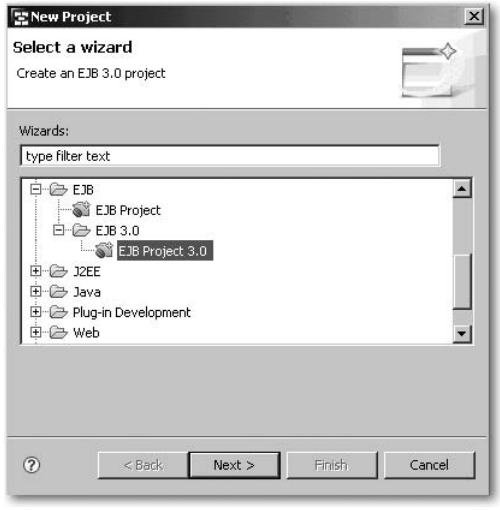


Figure 3.10 Selection of EJB Module Project in the New Project Wizard

2. By clicking **Next**, you proceed to the next wizard window. There you enter "EmployeeEjb" as the name for the new project. In addition, you accept the default settings and complete this procedure with **Finish**.
3. The creation wizard generates an initial project framework for the new EJB project and creates a project folder in the directory.
4. Now start the J2EE perspective, if you have not done so already, and display the project structure in the **Project Explorer**. This view will now serve as your central starting point for all future activities concerning the EJB 3.0 development.

In the next step, add an entity named `Employee` to this project.

3.4.2 Defining an Employee Entity

As already mentioned, the data model should be kept as simple as possible in this introductory example. Therefore, you should define only one single entity named `Employee`, to correspond with the already existing table called `TMP_EMPLOYEES`.

General Properties of the Entity Class

In the next step you create a new, serializable Java class. This will be a class that, for the most part, declares the appropriate attributes and provides the corresponding set and get methods.

New class wizard

1. To create such a class for the EJB module project, open the context menu for the project node and choose the option **New • Class**.
2. Enter "Employee" as the name for the new class and assign the package `com.sap.demo.entity`. In addition, activate the option **Constructors from Superclass** and add the interface **Serializable** to your selection.
3. Then accept the standard default settings and create the class by pressing **Finish**.
4. When you have completed the creation procedure, start the Java editor and add some field definitions to the actual class body[7]:

   ```
   private long employeeId;
   private String lastName;
   private String firstName;
   private String department;
   private int version;
   ```

   In this way, you equip the entity class with the exact fields that you created in the corresponding employee table as table fields.

7 In accordance with the specification, we recommend creating a version field (`version`) for the entity. This field is used by the JPA container at runtime in order to implement optimum verification and thus ensure that no competing accesses are implemented for one and the same data source. As soon as the container registers accesses of this type, an exception is thrown for the transaction. The most recent data state is then retained and a rollback is set for the current transaction. With these simple means, you help to maintain data consistency.

5. Then, in the editor, select all the rows with the fields you have just created and choose **Source • Generate Getters and Setters...** from the context menu. In the displayed window, click the key **Select All**. In this way, the corresponding getter and setter methods are generated for all fields, in accordance with Listing 3.1.

```
public class Employee implements Serializable {
   private static final long serialVersionUID = 111L;
   private long employeeId;
   private String lastName;
   private String firstName;
   private String department;
   private int version;
   // non-arg constructor
   public Employee() {
   }
   public String getDepartment() {
   return department;
   }
   public void setDepartment(String department) {
   this.department = department;
   }
   public long getEmployeeId() {
   return employeeId;
   }
   public void setEmployeeId(long employeeId) {
   this.employeeId = employeeId;
   }
   [...]
}
```

Listing 3.1 Implementation of a Regular Java Class Named Employee

6. Finally, save the current editor content using the appropriate icon in the toolbar.

The implementation of the class `Employee` has thus far shown no anomalies. It defines five fields: `employeeId`, `lastName`, `firstName`, `department`, and `version`, and places the getter and setter methods at your disposal in accordance with the name convention for JavaBeans. It should be mentioned, however, that the JPA demands a parameter-free constructor for an entity. But further constructors can be added.

Because this is not an abstract class that also avails of a `public` constructor, you have thus far been dealing with a POJO that can already be instantiated. Moreover, the class implements the interface `java.io.Serializable` so that entity objects can be serialized through remote calls or in Web service calls, respectively.

In this connection, follow the general recommendation and explicitly declare a version number[8] named `serialVersionUID` for the serializable class. For this reason, a same-name field that is `static`, `final`, and of the type `long` was added subsequently in the declaration part.

Object-Relational Mapping

Strictly speaking, we do not yet have an entity here, but only a simple JavaBean object. What is missing is a type of meta information[9] that describes the mapping of the Java object onto the relational database. Using the JPA, this is easily achieved, simply by adding the annotations to the source code of the Java class.

With the simple addition `@Entity` to the class definition, you identify the class `Employee` as an EJB 3.0 Entity. With this step, you set the command that the entity is suitably mapped to a database table. In addition, the persistence framework requires information as to how the entity is mapped to the relational database table.

It should be remembered that the JPA provides the application developer with a very comfortable path to realize this kind of object-relational mapping, based on a record of plausible default rules. If no explicit specifications are made — for example, for the name of the table or the individual table fields — the JPA assumes certain plausible assumptions.

8 The version number `serialVersionUID` is required by the serialization runtime for each serializable class for verification purposes. If a serializable class does not explicitly declare a `serialVersionUID`, a default value is calculated by the runtime for this version number. This default value can, however, depend on the compiler implementation. To guarantee a consistent version number for all compilers, we recommend that you explicitly declare a `serialVersionUID` for the class.

9 With the JPA, meta information can be stored for the entity class in the form of a separate XML file, as before, using deployment descriptors. The use of annotations, however, is to be preferred — in particular, because this is normal practice in the standard Java SE 5.0.

In this example, an entity with the name `Employee` would be mapped onto a database table with the name `EMPLOYEE` in accordance with these rules. However, because the names in this case are to be different, you have the option of overwriting them using the annotation `@Table`. You proceed in a similar fashion when mapping the persistent fields of the entity to the corresponding table fields. If a persistent field deviates from the name of the table field onto which it is to be mapped, the annotation `@Column` is added with the specification of the corresponding field name. This situation applies, for example, to the field `EmployeeId`, which is mapped onto the table field with the name `ID`. The situation is different, however, with the persistent field `department`, which is mapped onto a table field with the same name.[10] Here you do not have to make any explicit specification. To identify the version field as such, it is necessary for you to add the corresponding annotation `@Version` to the field `version`.

1. Now add the required annotations to the Java source code of the class `Employee`, as displayed in Listing 3.2.

```
@Entity
@Table(name="TMP_EMPLOYEES")
public class Employee implements Serializable {
      @Column(name="ID")
      private long employeeId;
      @Column(name="LAST_NAME")
      private String lastName;
      @Column(name="FIRST_NAME")
      private String firstName;
      private String department;
      @Version
      private int version;
      [...]
}
```

Listing 3.2 Annotations in the Source Code of the Employee Class

10 The JPA specification supplies no guidelines on adherence to uppercase or lowercase lettering for tables and field names. For the implementation of SAP JPA 1.0, the following rule therefore applies: If the name of the table or table field is listed explicitly using the annotation, uppercase and lowercase lettering is taken into consideration. If, on the other hand, the table name or table field is generated in accordance with the default rule, uppercase lettering is used.

2. If you have not already done so, finally create the missing imports for the employee class. For this purpose, click on an arbitrary position in the Java editor and choose **Source • Organize Imports** from the context menu.
3. The missing import lines are then added, as shown in Listing 3.3. Now, no more errors should be displayed in the source code of the Bean class.

```
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Table;
import javax.persistence.Version;
```

Listing 3.3 Supplementing Certain Import Lines for the Class Employee

Generating the Primary Key

So that each instance of an entity can be uniquely identified, the entity class must have an identifier that can simultaneously serve in the assigned table as a primary key. For this reason, a field of the name `EmployeeId` is already created and you will use it as an identifier for the employee entity. The specification of the identifier field is done easily with the annotation `@Id`, which you place in front of the field. In this case, this field with the primary key of the corresponding database field is identified through the mapping onto the table field `ID`.

Now you are faced with the question as to which generation method is to be used to generate the primary key. Admittedly, there are many solutions and strategies, but it would go beyond the scope of these explanations. But this much should be said: Generally speaking, key fields can be provided using the database or using the server container, or even through the application itself. The JPA specification is again of help to the developer and provides various strategies for automatic ID generation. The developer does not need to implement any ID generation logic, but instead can initiate automatic primary key generation using the annotation `@GeneratedValue` and can also use the various generation strategies.

Table for ID generation

In the following section, you will see how the table strategy is implemented. Here a special table for the generation of the ID value is used.

First, however, you must create a corresponding table because it is not automatically provided by the framework as a type of system table. This work step is very simple.

1. Again start the Dictionary perspective and add a further table definition named `TMP_ID_GEN` to the already existing project **EmployeeDic**.
2. The new table should be defined exactly as shown in Figure 3.11 and contain the two fields `GEN_KEY` and `GEN_VALUE`.[11] The field `GEN_KEY` defines the table key and will contain the fully qualified class name at runtime. The field `GEN_VALUE` is provided for storing the last generated ID value.

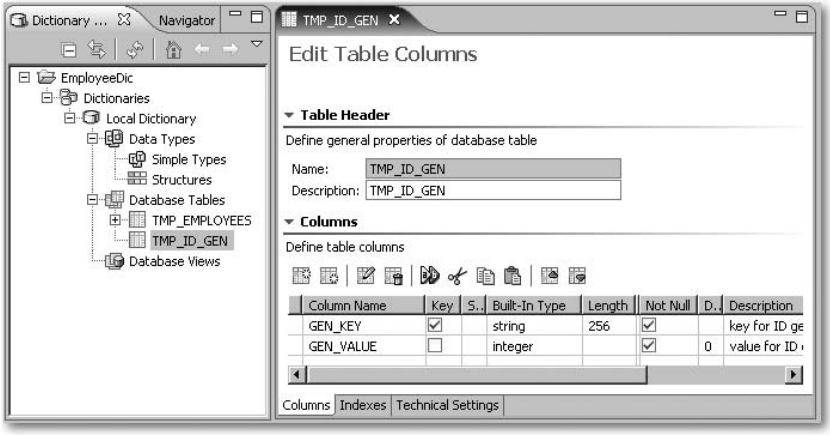


Figure 3.11 Creating Another Table Definition for ID Generation in the Dictionary Project EmployeeDic

Annotation for the ID generator

3. As displayed in Listing 3.4, it is possible to generate a suitable ID generator with the help of this new table. In the source code of the employee class, therefore, add the appropriate annotation `@TableGenerator` by putting the class name in the front. The element `table` references the table you have just created for ID generation while the element `name` is used to identify the generator. The name of the generator, in turn, is specified through the element `generator` using

11 The two table columns `GEN_KEY` and `GEN_VALUE` identify the standard names of the SAP JPA implementation for tables for ID generation. Alternatively, you can define other column names for the ID table, but in this case you must ensure corresponding mapping in the annotation for the table generator (`@TableGenerator`).

the annotation `@GeneratedValue`. As shown in Listing 3.4, add this annotation to the class attribute `employeeId`. Through the other element `strategy`, you instruct the container to use the generation method with the strategy of the type `TABLE` at runtime.

```
@TableGenerator(name="idGenerator", table="TMP_ID_GEN",
pkColumnName="GEN_KEY", valueColumnName="GEN_VALUE")
public class Employee implements Serializable {
@Id
@GeneratedValue (strategy=GenerationType.TABLE,
                 generator="idGenerator")
@Column(name="ID")
private long employeeId;
[...]
```

Listing 3.4 Definition of a Primary Key for the Employee Entity

Formulating the Query Using an EJB QL Statement

Search queries in finder methods

Search queries are often used during access to database data. The specification for EJB 3.0 provides multiple options on how you can implement queries. The named parameters are an important element here; they are used both in static and dynamic queries.

You may remember how for the earlier EJB versions' static queries were defined in the EJB deployment descriptor and then, in an additional step, how the behavior of the finder methods were to be specified using the EJB QL statements. EJB version 3.0 continues this approach and provides for this purpose a simplified execution method. It allows the programmer to add static queries using the annotation `@NamedQuery` within the Java source code. This is a predefined query that is identified by its name. As is standard with finder methods, the method behavior is not specified using Java source code but through EJB QL statements. You can use these to formulate suitable search queries.

This is precisely what you will do at this point by formulating the required EJB QL statement for a query named `Employee.findAll`.

1. Create the annotation `@NamedQuery` in the Java Editor, before the definition of the class `Employee`.

2. The element `name` serves to identify the query using a string value. The element `query` adopts the EJB QL statement `SELECT e FROM Employee e` (Listing 3.5).

```
@NamedQuery(name="Employee.findAll",
            query="SELECT e FROM Employee e")
@Entity
@Table(name="TMP_EMPLOYEES")
public class Employee implements Serializable {

[...]
```

Listing 3.5 Definition of a Query for Displaying All the Employee Objects

3.4.3 Configuring the Application for Database Accesses

So that the EJB container can handle the database accesses in the first place, it must know certain global settings for persistence, such as the name of the data source, which is required for the link to the database. As a rule, configuration tasks are taken on by the server, but there are some exceptions. Therefore, we will describe how these few configuration tasks are to be executed once by the developer for the EJB project. This manual configuration of the Java persistence takes place in a special persistence descriptor with the name *persistence.xml*.

Defining the Persistence Unit

In a typical EJB application, the data model consists mostly of several entities that reference each other and are to be mapped onto one and the same database schema. You must now ensure that all entity classes that belong together also build a logical unit for the EJB container at runtime, are managed by the entity manager as such, and fall back on one and the same data source. This kind of logical unit is described as a persistence unit.

Persistence unit combines entities

The persistence unit is comprised of entities of an application that are addressed at runtime through the Entity Manager. Remember that a persistence unit must be set explicitly — even when, as in this case, only a single entity is involved.

All entities that belong together and form a persistence unit can be listed explicitly in the configuration file. This, however, is not absolutely necessary because the persistence framework otherwise searches through the application for entities and automatically finds them. In this tutorial application, only the name and a short description should be specified for the persistence unit. The basic principle is to perform configuration only in an exceptional case. In addition, two further specifications are required — one for the JTA data source and one for the version generator.

Configuration in persistence.xml

Because this kind of configuration file is not yet contained in the current EJB project, create the file *persistence.xml* using the appropriate XML schema.

1. To do this, select the EJB project in the **Project Explorer** and navigate to the folder **META-INF**.
2. From the context menu for this folder, choose the menu path **New • Other • XML • XML** and navigate with **Next** to the next step.
3. On the displayed wizard page, select the option **Create XML file from an XML schema file** and again press the **Next button**.
4. On the following wizard page, enter "persistence.xml" as the file name and choose **Next**.
5. As shown in Figure 3.12, now decide on the option **Select XML Catalog entry** and then **persistence_1_0.xsd** from the displayed XML catalog.

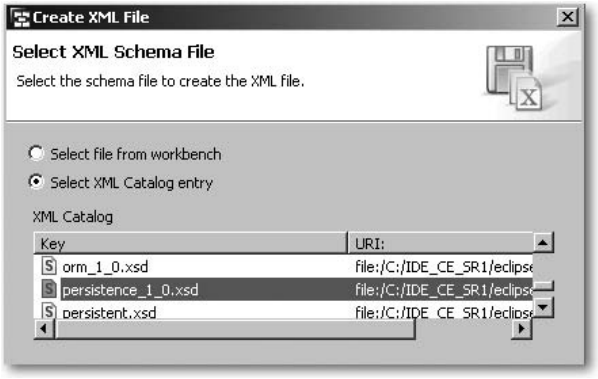


Figure 3.12 Selection of XML Schemas when Creating persistence.xml for the EJB Project

6. On the next wizard page, confirm your selection by pressing **Finish**.
7. To complete the content for the persistence descriptor, use the design view (Figure 3.13) and, using the context menu for the last entry, create a few additional tags. You will find the required configuration specifications in Table 3.1.

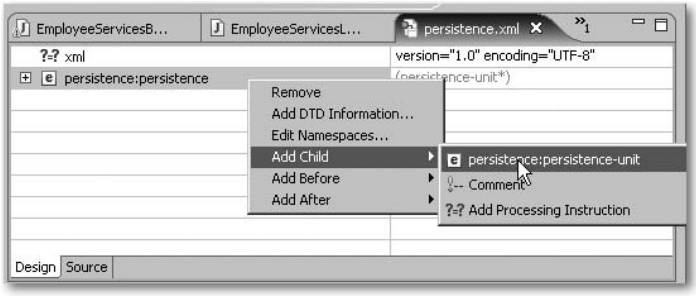


Figure 3.13 Creating More Tags in the Design View of persistence.xml

XML Tag	Assigned Value
`persistence-unit | name`	`EmployeePU`
`persistence: description`	Sample Application Persistence Unit
`persistence: jta-data-source`	`TMP_EMPLOYEES_DATA`
`persistence: properties`	
`property | name`	`com.sap.engine.services.orpersistence.generator.versiontablename`
`property | value`	`TMP_ID_GEN`

Table 3.1 Specifications for Persistence Unit in persistence.xml

8. The name of the persistence unit is generally optional and is added to the XML source within the element `<persistence-unit>`. We will refer to the persistence unit again when the instance of the unit is to be accessed in the session bean using the Entity Manager.
9. Within the element `<persistence-unit>`, use the tag `<jta-data-source>` to enter the data source alias. We will also deal with this in more detail at a later point.

10. Finally, enter a `<property>` to enable versioning of the data source. This specification is necessary for the following reason:

    So that versioning of the data source is at all useful, the JPA specification requires that the data source uses the isolation level `READ_COMMITTED`. However, to cater to the difference between this requirement and the actual isolation level `READ_UNCOMMITTED`, you will need a suitable version generator. You address this kind of generator in the *persistence.xml* through a certain property of the persistence unit. You specify this property using the name element from Table 3.1.

    In addition, a version generator requires a suitable database table. Any arbitrary generator table that has the field names `GEN_KEY` and `GEN_VALUE` is suitable for this. By all means, the table `TMP_ID_GEN` previously created is suitable for this purpose. Therefore, assign it as the property value.

11. The generated XML source then corresponds to the lines in Listing 3.6:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns=
 "http://java.sun.com/xml/ns/persistence" [...]>
<persistence-unit name="EmployeePU">
  <description>
    Sample Application Persistence Unit
  </description>
  <jta-data-source>TMP_EMPLOYEES_DATA</jta-data-source>
  <properties>
    <property name = "com.sap.engine.services.
                      orpersistence.generator.
                      versiontablename"
                      value= "TMP_ID_GEN">
    </property>
  </properties>
</persistence-unit>
</persistence>
```

Listing 3.6 Resulting XML Source of persistence.xml

3.5 Defining the Business Logic

After you have completed the data accesses in Section 3.4, *Implementing Access to Table Data*, using a JPA entity, now turn to the business logic. Because you are using EJBs, session beans[12] are usually the best way of encapsulating the business logic.

Stateless session bean

For this purpose, you will now create a stateless session bean named `EmployeeServices`, and then add and implement the required business methods. Using the business methods, arbitrary clients should be in a position to adopt the employee registration data entered by the user and finally pass them for storage to the entity `Employee`. Also, it should be possible for all existing data records on all existing employees to be passed to clients for display purposes.

3.5.1 Creating a Session Bean

To create a session bean named `Employee Services`, start the appropriate creation wizard.

1. Start the context menu on the project node **EmployeeEjb** in the Project Explorer, and choose the menu option **New • EJB • EJB 3.0 • EJB Session Bean 3.0**.
2. In the displayed dialog box, assign certain elementary properties in accordance with the list in Table 3.2.
3. Because no further options are required for the session bean you want to create, choose **Finish**. By doing this, you start the generation procedure.

12 In distributed applications, session beans implement the application-relevant processes and tasks, take care of the transaction management, and arrange access to low-level components, such as entities or other data access components as well as auxiliary classes. This use of session beans matches the session façade design pattern and serves to define a clear separation between the different levels (data accesses and business logic) with the aim of increasing performance at runtime.

Field Name	Assigned Value
EJB Class Name	`EmployeeServices`
EJB Project	`EmployeeEjb`
Default EJB Package	`com.sap.demo.session`
Session Type	Stateless
Transaction Type	Container
Create Business Interface	Checkbox **Local** activated

Table 3.2 General Properties when Creating the Session Bean EmployeeServices

4. As shown in Figure 3.14, the wizard creates a bean class `EmployeeServicesBean` and the respective business interface `EmployeeServicesLocal`.

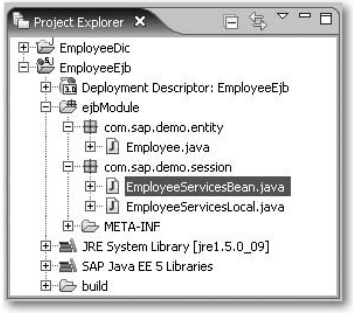


Figure 3.14 Session Bean EmployeeServices in the Project Explorer

When you created the session bean, you assigned the type `Stateless`. Therefore, this meta information is stored in the generated bean class by placing the respective annotation `@Stateless` in front of the class name. In contrast to stateful session beans, this session bean is not able to store its state in its instance variable. The application does not provide for storing user-specific information. As a result, the methods of the session bean will behave as stateless.

The declaration of the business interface as a local interface in turn means that the annotation `@Local` is generated for the interface name. As already mentioned, business methods of the session bean are exposed at

this business interface so that a client can access the session bean with the help of this precise interface. In the case of a local interface, one can assume that the EJB and the client are on the same server.

3.5.2 Implementing the Session Bean Class

Business methods

So far, the procedure has been mostly declarative in nature. Now the implementation of the specific service functions for the tutorial application will follow. Using the business methods, you will implement the functions that are also available to the client application. In the case of business methods, you are dealing with special methods of the session bean that implement their specific service functions that are available on an external basis.

Generally speaking, business methods are declared as regular Java methods in local, remote, or, if necessary, in both business interfaces, and are implemented in the respective session bean class. Depending on the business interface that provides the appropriate business method, this method is available either for local or remote clients, or for both.

In the following explanations, you will see that you only need to supply local clients with data and implement certain business methods that already show, in the examples, how some of the basic operations on persistent objects are to be performed. In this way, you will learn — with the help of the method `createEmployee()` — how a new data record is created and how it is stored permanently in the database. You will also learn how to implement the search for a data record with the primary key using the method `getEmployeeById()`. A further read access is connected with a query execution and defines the third business method `getAllEmployees()` for the tutorial application.

Keeping the Instance of the Entity Manager

Entity Manager as interface to database

Business methods should be able to create new data records, manipulate existing ones, and finally synchronize the changes with the database. For this reason, you require a kind of local interface for interaction with the database. The JPA provides the application developer with such an interface to the Entity Manager. The purpose of the Entity Manger is to

control the lifecycle of entity objects and to change their status. Using the Entity Manger, you can perform all database operations on entities and thus create, change, read, search for, or even delete objects on the database. Listing 3.7 shows you how you access the persistence unit, starting from the session bean, and how you define the Entity Manager for the persistence unit.

```
@Stateless
public class EmployeeServicesBean implements
EmployeeServicesLocal {
      @PersistenceContext (unitName = "EmployeePU")
      private EntityManager eManager;
[...]
}
```

Listing 3.7 Access to the Entity Manager Within the Session Bean

As you can see from Listing 3.7, the session bean declares a variable of the type `EntityManager`, without having a certain value assigned to it. Two aspects are of interest here: First, the source code is part of the session bean and is executed on the application server. On the other hand, the variable is provided with the annotation `@PersistenceContext`. In addition, this annotation contains the element `unitName`. Using this parameter, you enter the name of the persistence unit on which the Entity Manager operates. You will surely remember that the entered value corresponds exactly to the name you have already entered in the configuration file *persistence.xml*.

This is of interest here because the session bean uses a technique called resource injection. Due to the annotation, it is left up to the server to supply the variable (here: `eManager`) with an `EntityManager` instance. In this way, you ensure that this variable is always correctly initialized when a business method is called for the first time.

Creating a New Employee Data Record on the Database

After you have seen how you can access the Entity Manager in the bean class, you will now see — on the basis of the business method `createEmployee()` — how easily a new data record can be created on the database. First, an instance of the entity `Employee` is created using a construc-

tor. The data required for specifying an employee is passed in the form of a method parameter. Access to the persistent fields is performed using the setter methods. The fully specified object `employee` is finally passed using the method `persist()` to the Entity Manager that triggers permanent storage on the database. As a result, the business method returns an ID of the type `long`.

The complete implementation of this method can be seen in Listing 3.8.

```
public long createEmployee(String lastName, String
firstName, String department) {
      long result = 0;
      Employee employee = new Employee();
      employee.setFirstName(firstName);
      employee.setLastName(lastName);
      employee.setDepartment(department);
      eManager.persist(employee);
      result = employee.getEmployeeId();
      return result;
}
```

Listing 3.8 Source Code for the Business Method createEmployee()

Searching for a Data Record Using an ID

Frequently, an application must be in the position to first identify a certain data record on the database before it can perform a new operation on this data. The search for a certain employee data record using the ID (that is, the primary key) is shown as an example in Listing 3.9.

```
public Employee getEmployeeById(long empId) {
       Employee employee =
       eManager.find(Employee.class, Long.valueOf(empId));
       return employee;
}
```

Listing 3.9 Source Code for the Business Method getEmployeeById()

In this case, the call takes place using the `find()` method of the Entity Manager. This method contains two arguments: The first argument is the entity class of the object to be searched for, while the second argument is the object representation of the entity identifier, that is, the key

field. The `find()` method returns the found entity instance or `null` if no such entity was found in the database. Because the `find()` method was implemented generically, a casting of the resulting value is not required in this case. In other words, the `find()` method is parameterized in such a way that the type of the returned result matches the type of the first argument of the method call. Whatever the case, an instance of the type `Employee` is returned.

Executing a Query

Another business method `getAllEmployees()` will now solely be used to demonstrate how a query can be used to read a resulting set. You will remember how you formulated a named query in Section 3.4.2, *Defining an Employee Entity*, using a select clause. In that case, a search query was stored with the symbolic name `Employee.findAll` in the source code of the `Employee` entity. Now you should use these queries to read the database records. It should be possible to return a list of all existing employees for a particular client.[13]

As you can see from the implementation of this finder method in Listing 3.10, query objects can be created through the Entity Manager. This is done by calling the method `createNamedQuery()`. As a parameter, a place holder that contains only the name of the defining query is passed. The execution of the query in the database and the reading of the resulting set is performed using the query method `getResultList()`.

```
@SuppressWarnings("unchecked")
public List<Employee> getAllEmployees() {
      Query query =
        eManager.createNamedQuery("Employee.findAll");
      List<Employee> result =
       (List<Employee>) query.getResultList();
      return result;
}
```

Listing 3.10 Source Code of the Business Method getAllEmployees()

13 The clients can be Java Server Faces, JSPs, servlets, Java classes, a different EJB, or a Web service client. Clients, particularly in large applications, have the advantage that they themselves do not have to define any search queries. Changes and adjustments to the query form can be done centrally, without any effect on the client.

To minimize the number of warnings at compiler time, add the appropriate annotation to the source code. `@SuppressWarnings` is used solely to suppress certain compiler warnings in connection with this method.

Defining the Transaction Behavior of the Business Methods

So far you have not made specifications at any time regarding the transaction behavior of the business methods. Only when you created the session bean did you determine the transaction type with the attribute **Container**. This means that, in such a case, the EJB container takes over control of the transaction. In relation to the Entity Manager, you have already seen that the EJB container takes over important standard tasks from the programmer.

With this transaction type, you do not have to set the commit or rollback methods. You can leave this task entirely up to the EJB container. As a rule, there is the option with EJBs of implementing the transaction logic on a program-controlled basis in the bean class itself. However, the decision in favor of transaction behavior based on the container-supported approach when a session bean is created has already been made.

Transaction attributes

You determine the desired transaction behavior for the individual business methods of the application using transaction attributes. As a result, all operations that go beyond read access to data records must take place within a transaction. Because transaction attributes are metadata, we use, as usual, predefined annotations. The required additions in the source code are shown in Listing 3.11.

```
@Stateless
@TransactionManagement
  (value=TransactionManagementType.CONTAINER)
public class EmployeeServicesBean
             implements EmployeeServicesLocal {
[...]
   public long createEmployee(String lastName, String
   firstName, String department) {
   [...]
   }
   @TransactionAttribute(TransactionAttributeType.SUPPORTS)
   public Employee getEmployeeById(long empId) {
```

```
    [...]
    }
    @TransactionAttribute(TransactionAttributeType.SUPPORTS)
    public List<Employee> getAllEmployees() {
    [...]
    }
[...]
}
```

Listing 3.11 Transaction Attributes of the Business Methods of the Session Bean EmployeeServices

First, the annotation `@TransactionManagement` defines that transaction control for the entire session bean is delegated to the container. Another annotation, `@Transaction Attribute`, enables you to adapt the transaction context individually to each single method. Remember that no annotation was added to the method `createEmployee()`. The reason for this is solely that the default behavior is to be applied to the basic operation. The corresponding transaction attribute is called `REQUIRED` and requires that a new transaction is always started whenever this is necessary. If, for example, at the time of the method call no transaction is active, the application server automatically starts a new transaction, executes the business method, and sets a transaction commit immediately thereafter. On the other hand, if a transaction is already available, this one is used. This way you can see that creating a new employee data record definitely requires a transaction, albeit one that is not necessarily exclusive. For this purpose, the transaction attribute `REQUIRED` is ideal.

The methods `getEmployeeById()` and `getAllEmployees()` are quite different. Both methods implement solely reading accesses. Because no changes to data records result, a transaction is actually not required. If, however, a transaction is active at the time of the method call, this one is used. A new transaction, on the other hand, is not started. On the basis of this tolerance toward the transactions, the default behavior can be overwritten with the transaction attribute `SUPPORTS`. In this way, the source code of the bean class is complete. If you have not done so already, add the missing imports to the bean class. Then click on an arbitrary position in the Java editor and select **Source • Organize Imports** from the context menu. The missing import lines are then added. Now no further errors should be displayed in the source code of the bean class. Finally, adapt

the formatting of the new lines by choosing **Source • Format** from the editor context menu and then save the editor content using the corresponding icon in the toolbar.

3.5.3 Adding Business Methods to the Business Interface

Because the business methods already defined in the bean class are not automatically added to the appropriate business interface, you must perform this step manually.

1. To propagate individual business methods from the bean class to the business interface, select the bean class in the project explorer by double-clicking it. If no outline view is displayed in the current view, open this one first.
2. As shown in Figure 3.15, select all the business methods within the outline view, open the context menu, and then choose the option **EJB Methods • Add to Local Interfaces**.

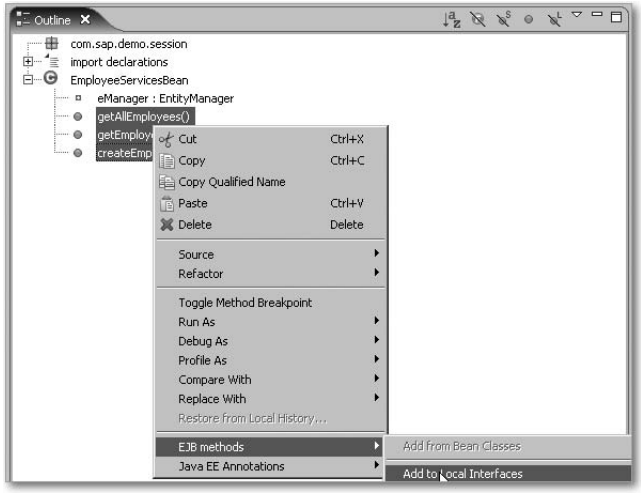


Figure 3.15 Propagating the Business Methods to the Business Interface, Starting from the Outline View

3. Listing 3.12 shows the resulting source code of the business interface.

```
import javax.ejb.Local;
import java.util.List;
```

```
import com.sap.demo.entity.Employee;

@Local
public interface EmployeeServicesLocal {
      public List<Employee> getAllEmployees ();
      public Employee getEmployeeById (long empId);
      public long createEmployee (String lastName, String
                  firstName, String department);
}
```

Listing 3.12 Business Methods in the Business Interface EmployeeServicesLocal

3.6 Creating a JSP-Based Web Application

The Developer Studio provides a special project structure for managing Web resources such as JavaServer pages, JavaServer faces, servlets, static HTML pages, and custom-tag libraries, as well as screen and graphic files. To prepare the initial project frame, you will create a corresponding project — that is, a Web module project — at the very outset.

To keep the Web application as simple as possible, add a JSP to the project as the only resource and implement with it the user interface of the Web client. In addition to the actual presentation editing, the accesses to the business methods of the session bean `EmployeeServices` should be implemented. As an example, use some information on the configuration of the Web application in the corresponding deployment descriptor.

3.6.1 Creating a Web Module Project

To create a Web module project, perform the following steps:

Container for Web resources

1. Start the **New Project Wizard** through the menu option **File • New Project**.
2. In the displayed wizard window, select the category **Web • Web 2.5** and then **Dynamic Web Project 2.5**.
3. With **Next**, you proceed to the next wizard window. There you enter "EmployeeWeb" as the project name. Otherwise, take the default settings and close the procedure by pressing **Finish**.

4. The best way of looking at the project frame is in the Project Explorer. In the JSP, you want to access resources from the Ejb module project. Therefore, you must also take this project dependency into account. For this purpose, click the project name **EmployeeWeb** and select the menu option **Properties** from the context menu.
5. In accordance with the specifications in Figure 3.16, select the property **Java Build Path**, click the tab **Projects**, and assign the desired project.

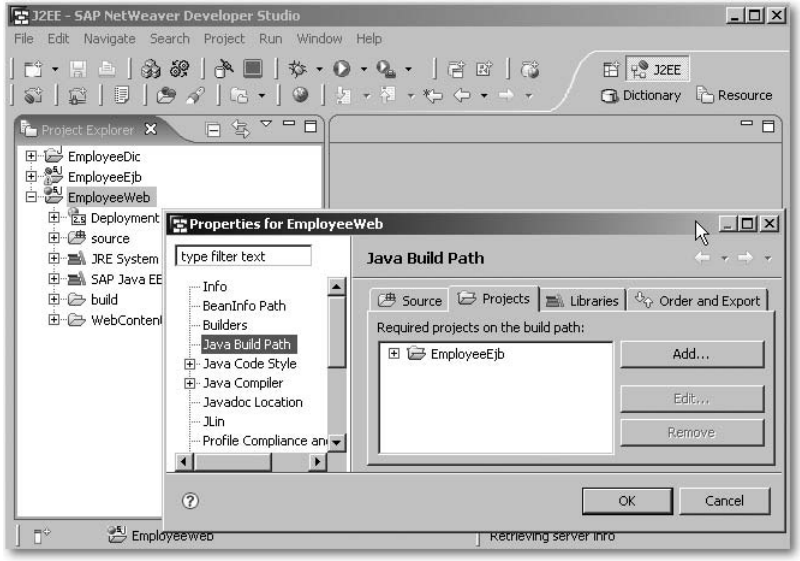


Figure 3.16 Assigning Java Build Path to the Project EmployeeWeb

3.6.2 Implementing the User Interface with JSP

Now you can begin with the implementation of the user interface in the JSP editor.

1. To add a JSP to the new project, click on the entry **WebContent** in the Project Explorer and choose the menu option **New • JSP...**.
2. You now see a wizard in which you can enter the name "index.jsp".
3. Further specifications are not required. Close the procedure by pressing **Finish**.

Generally speaking, there are two sections in the source code of the JSP:

- A HTML basic structure that, for the most part, defines a static input form for the Web application
- A dynamic Java-based section with which you can implement accesses to the business logic

HTML Basic Structure

The form for registering new employees could hardly be easier. It contains, in addition to the two input fields for the names, a selection list with the corresponding departments and a pushbutton with which the user can trigger registration. All these interface elements are listed within a HTML table. The complete structure is displayed in Listing 3.13.

```
<%@ page language="java" [...] %>
<!DOCTYPE html PUBLIC  [...] >
<html>
[...]
<!-- Import statements   -->

<!-- Reference to Session Bean  -->
<body style= "font-family:Arial;"  bgcolor="D2D8E1">
<h2>
Register New Employee
</h2>
<form method="GET">
<table border=0 align=center>
<tr>
<td width="150" >First name: <td>
<input type="text" name="firstname" value = "" size="20">
<tr>
<td width="150" >Last name: <td>
<input type="text" name="lastname" value = "" size="20">
<tr>
<td width="150" >Department: <td>
   <select name="department" >
    <option value="DEVELOPMENT">Development</option>
    <option value="TRAINING"> Training</option>
    <option value="MANAGEMENT"> Management</option>
    <option value="ARCHITECTURE"> Architecture</option>
```

```
  </select>
<tr>
<td><td><br>
<input type="submit" value="Create" name="create">
</table>
<br>
</form>
<!- Invoke business method -->
</body>
</html>
```

Listing 3.13 HTML Basic Structure for the JSP

Access to the Session Bean

In accordance with Listing 3.14, first supplement the JSP source code with some page directives whose import attributes contain the required package for JNDI Lookup as well as the business interface.

So that the reference to the session beam is retained, a JNDI Lookup is performed on the context variable `context`. Remember that with `EmployeeRegister` in Listing 3.14 you use a symbolic name for the session bean. This name must also be specified as the reference name in the deployment descriptor of the Web application. The result of the lookup is assigned to the local employee object `employeeService` after the business method casting is completed.

```
<%
<!-- Import statements   -->
<%@ page import="javax.naming.*" %>
<%@ page import=
         "com.sap.demo.session.EmployeeServicesLocal" %>

<!-- Reference to Session Bean  -->
<%!
private EmployeeServicesLocal employeeService;
private void lookup() {
   try {
      InitialContext context = new InitialContext();
      employeeService = (EmployeeServicesLocal)
      context.lookup("java:comp/env/EmployeeRegister");
   } catch (Exception ex) {
```

```
      System.out.println("Couldn't find bean"+
                         ex.getMessage());
   }
}

public void jspDestroy() {
   employeeService = null;
}
%>
```

Listing 3.14 Dynamic Section of the JSP — Access to the Session Bean

Calling the Business Method

In the following section, not all business methods will be called within this simple Web application. We merely wish to show an example of how, starting from a JSP, the business method `createEmployee()` can be called in order to create a new data record on the database. This, too, is an intended simplification because, normally, a JSP-based Web application consists of a combination of JSPs and servlets, and possibly also further JavaBeans as auxiliary classes in order to achieve strict separation between the actual presentation layer and the controller layer. While JSPs are preferred for presentation editing and thus also preferred as a view component, servlets or JavaBeans usually act as a controller and implement the application logic.

To save some of the typing work, the call for the method `createEmployee()` is embedded in the JSP source. However, it could be transferred easily to a servlet or to a JavaBean in the role of a controller. As you can see from Listing 3.15, a JNDI lookup precedes the actual call of the session bean method. This lookup was previously encapsulated in a separate method `lookup()`. Therefore, the local session bean object `employeeService` calls the business method `createEmployee()`. The local variables `lName`, `fName` and `eDepartment` are passed as parameters. They were defined in the first lines of the source code excerpt and adopt the currently entered user values. Whenever the business method is successfully executed in the EJB container, it returns a valid EmployeeID. Otherwise an exception is thrown and a corresponding text is displayed on the user interface.

```
<%
  lookup();
  if(this.employeeService == null) {
    throw new IllegalStateException("Bean not available!");
  }

  String fName = request.getParameter("firstname");
  String lName = request.getParameter("lastname");

  String eDepartment = request.getParameter("department");
  if(lName == null || fName == null
    || lName.length() == 0 || fName.length() == 0) return;
  long empID = employeeService.createEmployee(lName, fName,
               eDepartment);
  if(empID == 0)
      out.println("<H3> Failed!  </H3>");
      else
      out.println("<H3> Success! </H3>");
%>
```

Listing 3.15 Dynamic Section of the JSP — Business Method Call

3.6.3 Descriptions in the Deployment Descriptor web.xml

You can get information on the configuration of the Web application from the deployment descriptor *web.xml*. The entries contained there are evaluated at deployment time by the Web container. On one hand, the Web container receives all information as to how the individual resources of the project fit with each other. On the other hand, the assignment of security roles is contained in *web.xml*. These are the security roles through which the access authorization for the Web application can be controlled at runtime.

However, only certain mapping information is stored in the descriptor. In the following step, you will define, as an example, a reference to the required session bean using a symbolic name, but you will not make any further specifications on the configuration of the Web application.

Symbolic Name for the Session Bean

Reference to session bean

For access to the session bean, we have used a symbolic name, not the real bean name, in the JSP source code (see Listing 3.14). So that the

Web container can assign such a name at runtime as well, a corresponding mapping regulation must be stored in the deployment descriptor.

1. To define a symbolic name for a reference to the session bean, open the deployment descriptor *web.xml*. Click on the tab **Design** and add a further tag for the EJB reference.
2. Similar to the specifications in Figure 3.17, set the EJB name "EmployeeRegister" so that it matches the entry in the JNDI lookup in the JSP source code from Listing 3.14.

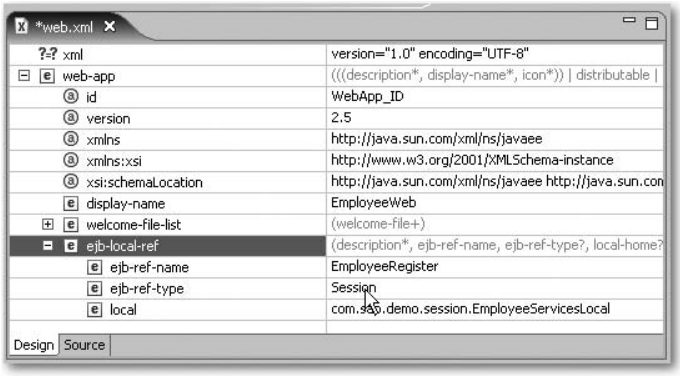


Figure 3.17 Setting the EJB Reference to the Session Bean in web.xml

3. The new entries are automatically added to the XML source at a suitable position. You can ensure this is the case by clicking on the tab **Source** and then navigating in the displayed XML source to the element `<ejb-local-ref>` (Listing 3.16).

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_ID" version="2.5" [...]">
  <display-name> EmployeeWeb</display-name>
  <welcome-file-list>
     [...]
  </welcome-file-list>
  <ejb-local-ref>
    <ejb-ref-name>EmployeeRegister</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <local>
      com.sap.demo.session.EmployeeServicesLocal
    </local>
```

```
  </ejb-local-ref>
</web-app>
```

Listing 3.16 Generated XML Source for the Reference to the Session Bean

4. With this entry, you have now defined a mapping between a freely selectable reference name (symbolic name) and the real bean name. Thus, the reference name assigned in the source code of the JSP for the session bean can be used, and it remains unchanged there, even if the bean name changes.

3.7 Defining and Deploying the Java EE Overall Application

While the business functions are provided with the EJB module, the suitable Web components have now been added with the Web module. At this point, only the components for a Java EE overall application need to be combined. The Developer Studio provides a special project type for this purpose. It is referred to as an enterprise application project.

To create the employee overall application, first create an enterprise application project named "EmployeeEar". Here you also set up a configuration file for the data source alias before you generate the appropriate EAR for the overall application and finally deploy this on the Java application server.

3.7.1 Creating the Enterprise Application Project

To create the project, proceed as follows:

1. Start the **New Project Wizard** through the menu option **File • New Project**.
2. In the displayed wizard window, select the category **J2EE • Java EE** and finally **Enterprise Application Project 5**.
3. By pressing **Next**, you proceed to the next wizard window. There you enter "EmployeeEar" as the project name. Accept the default settings and proceed to the next wizard window by pressing **Next**.

4. In accordance with the specifications in Figure 3.18, assign the EJB module *EmployeeEjb.jar* and the Web module *EmployeeWeb.war* to the EAR project before you complete the procedure by pressing **Finish**.

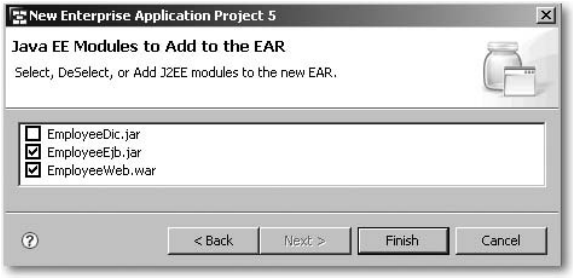


Figure 3.18 Creating the Modules Ejb and Web for the EAR Project

3.7.2 Creating the Data Source Alias

You will remember that you entered a name for the data source alias in the deployment descriptor *persistence.xml*. Such a data source alias has not been created anywhere so far. You will now perform this step. However, before continuing further, we would like to illustrate briefly the importance of the data source alias.

Data source alias as link between application and database

Excursion into Database Accesses and the Data Source Alias

The data source alias is required in order to enable communication to the database for table accesses from the application. A data source alias is a logical name for server-side access to a database resource (in this case, the table). The connection pool on the application server has knowledge of the path to the database table — that is, the actual data source that must already exist on the server.

The default data source (also called system data source) plays a special role here. This data source is automatically created during the installation of the AS Java and is not associated with a specific application. The default data source is intended as a default connection pool for use by several applications.

If you now create the data source alias in the Enterprise Application Project, you will associate it with the default data source. The use of an alias at this point has several advantages:

- The developer does not have to specify the physical path name for the database resource. Only the database system requires this information for managing its own resources.

- In addition, the alias is assigned in the Developer Studio and assigned to a specific project. Thus, the administrative task that the developer would be required to take on in a separate administration tool is dispensed with.
- Last, the use of a data source alias enables you to keep the entire application portable.

Because a corresponding configuration file for the data source alias is not yet contained in the current project, you will now create this first — starting from the appropriate XML schema.

1. In the **Project Explorer**, select the project and navigate to the folder **META-INF**.
2. From the context menu of this folder, select the menu path **New • Other • XML • XML** and press the button **Next**. On the displayed wizard page, choose the option **Create XML file from an XML schema file** and press **Next**.
3. On the following wizard page, enter "data-source-aliases.xml" as the file name and again choose **Next**.
4. As you can see in Figure 3.19, you decide on the option **Select XML Catalog entry** and then select **data-source-aliases.xsd** from the displayed XML catalog.

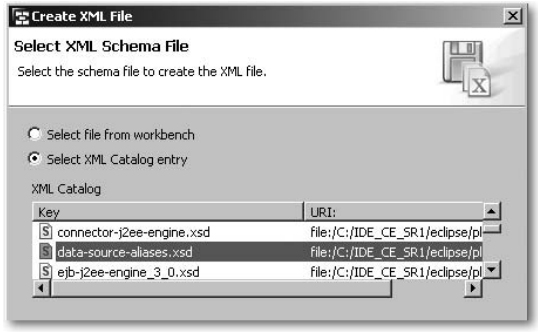


Figure 3.19 Creating a Data Source Alias for the EAR Project

5. On the following wizard page, select **data-source-aliases** for the **Root element** and confirm this by pressing **Finish**. With this step, the XML file named *data-source-aliases.xml* is created. It is then visible in the project structure.

6. Now open the XML editor and enter the system data source "${com.sap.datasource.default}" and the name "TMP_EMPLOYEES_DATA" for the alias.

The alias name is assigned to the system data source at deployment. You can easily follow this by looking at the generated XML source. It matches the lines shown in Listing 3.17.

```
<?xml version="1.0" encoding="UTF-8"?>
<data-source-aliases [...]>
  <aliases>
    <data-source-name>
      ${com.sap.datasource.default}
    </data-source-name>
    <alias>TMP_EMPLOYEES_DATA</alias>
  </aliases>
</data-source-aliases>
```

Listing 3.17 com.sap.datasource.default as Representation of the System Data Source on the AS Java

3.7.3 Deployment of the Employee Application

Preparations for deployment

Before the deployment of the application starts, you should check that the server process was started and that the database is online. The prerequisite for this, however, is that the AS Java has been registered in the Developer Studio.

As shown in Figure 3.20, two different options are provided on the **Preferences** page. Depending on whether the assigned AS Java was installed on the local host or under an arbitrary address in the LAN, you must distinguish between the option for the remote installation and the option for local installation. The required entries are contained in the system information, which you will find on the server welcome page.

Deploying the Dictionary Table Definitions

To be able to transfer the table definitions from the Dictionary project to a database instance, you need an archive file. This kind of Dictionary archive represents a transportable unit of the Dictionary project and combines all the Dictionary definitions of the project from the gener-

ated metadata. Only when the created archive is deployed on the application server is the physical representation of the corresponding table generated on the database instance on the assigned database using CREATE TABLE.

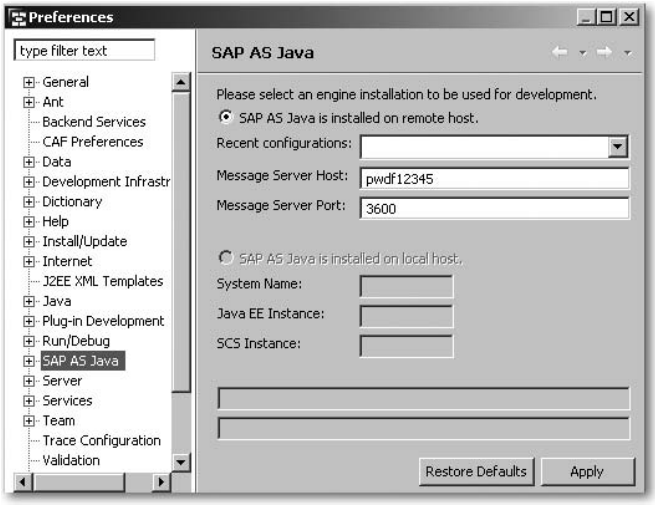


Figure 3.20 Registering the Application Server Java Under Preferences

1. To create the archive, choose the project node in the Dictionary Explorer, open the context menu, and select the option **Create Archive**.
2. Afterward, choose **Deploy** from the context menu of the project node (Figure 3.21).

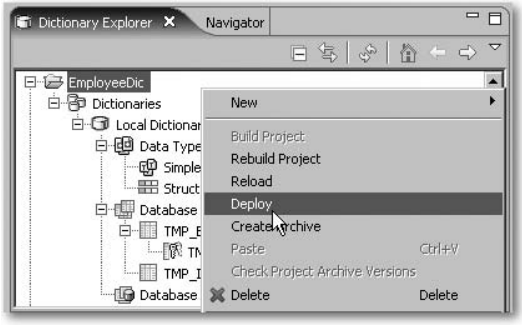


Figure 3.21 Deploying the Table from the Dictionary Project

3. From the **Deploy View Console** (Figure 3.22), you immediately get a report as to whether or not the deployment activity was successful. From this view, you can also look at the corresponding log file.

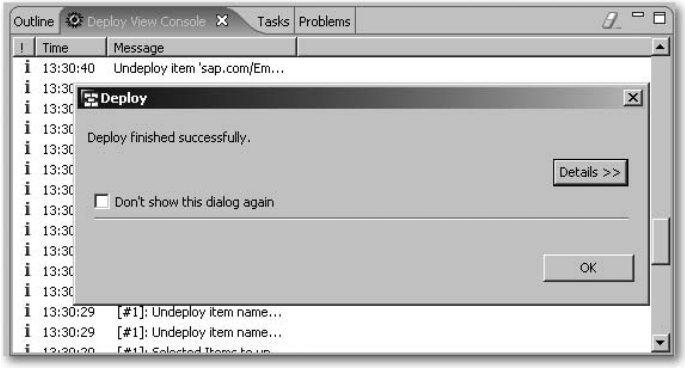


Figure 3.22 Displaying the Deploy Output View After Successful Deployment

4. If you are implementing the MaxDB as a database system and have installed the SQL studio, you can now easily check that both tables have been correctly created on the database instance. For this purpose, you need only to log on to the database server through the SQL Studio and to search for `TMP_EMPLOYEES` and `TMP_ID_GEN` in the list of all currently deployed tables on the server. If you have knowledge of SQL, you can also create some data records for the new employee table in the SQL Studio.

SQL Studio

Creating and Deploying the Enterprise Application Archives

The Enterprise Application Archive groups the JAR and WAR archives into one single archive with information from the corresponding deployment descriptors. The EAR contains, in addition to the business logic components, the presentation components. It can be easily created and deployed in one step from the Developer Studio using the Servers view.

1. Open the **Servers** view. The entry **SAP Server** should already be displayed here.
2. From the context menu, choose the option **Add and Remove Projects**. As shown in Figure 3.23, select the EAR project and confirm with **Fin-**

ish. Thus, in one single step you have added the deployable projects to the Servers view, generated the respective archives WAR, JAR, and EAR (and added them to the project view in the Project Explorer), and immediately triggered the deployment.

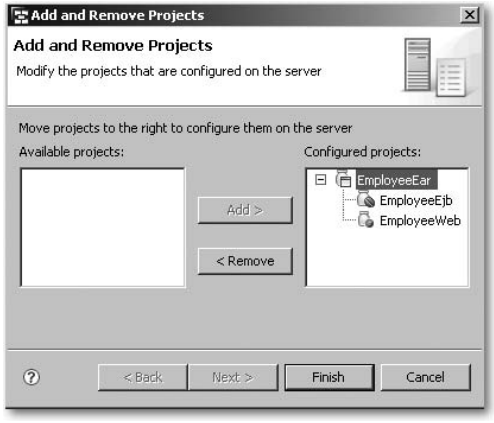


Figure 3.23 EAR Deployment Using the Servers View

3. If your deployment activity has been successful, you will receive the message "Deployment finished successfully".

3.7.4 Starting the Employee Application

Provided the server is called **localhost** and can be reached under the port **50100**, you can start the employee application with the URL *http://localhost:50100/EmployeeWeb/index.jsp* (Figure 3.24).

Figure 3.24 Starting the Employee Application in the Browser

Index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X