Creating a J2EE-Based Car Rental Application

SAP NetWeaver 04
Icons in Body Text

<table>
<thead>
<tr>
<th>Icon</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>⚠️</td>
<td>Caution</td>
</tr>
<tr>
<td>📢</td>
<td>Example</td>
</tr>
<tr>
<td>✍️</td>
<td>Note</td>
</tr>
<tr>
<td>💡</td>
<td>Recommendation</td>
</tr>
<tr>
<td>_syntax</td>
<td>Syntax</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Type Style</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Example text</em></td>
<td>Words or characters quoted from the screen. These include field names, screen titles, pushbuttons labels, menu names, menu paths, and menu options.</td>
</tr>
<tr>
<td></td>
<td>Cross-references to other documentation.</td>
</tr>
<tr>
<td><strong>Example text</strong></td>
<td>Emphasized words or phrases in body text, graphic titles, and table titles.</td>
</tr>
<tr>
<td>EXAMPLE TEXT</td>
<td>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.</td>
</tr>
<tr>
<td><em>Example text</em></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Example text</strong></td>
<td>Exact user entry. These are words or characters that you enter in the system exactly as they appear in the documentation.</td>
</tr>
<tr>
<td><code>&lt;Example text&gt;</code></td>
<td>Variable user entry. Angle brackets indicate that you replace these words and characters with appropriate entries to make entries in the system.</td>
</tr>
<tr>
<td>EXAMPLE TEXT</td>
<td>Keys on the keyboard, for example, F2 or ENTER.</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Creating a J2EE-Based Car Rental Application</td>
<td>6</td>
</tr>
<tr>
<td>Defining a Data Model</td>
<td>8</td>
</tr>
<tr>
<td>Creating a Dictionary Project</td>
<td>9</td>
</tr>
<tr>
<td>Creating a Table</td>
<td>10</td>
</tr>
<tr>
<td>Adding Table Columns</td>
<td>11</td>
</tr>
<tr>
<td>Creating an SDA</td>
<td>13</td>
</tr>
<tr>
<td>Deploying an SDA</td>
<td>14</td>
</tr>
<tr>
<td>Implementing Data Access</td>
<td>15</td>
</tr>
<tr>
<td>Creating an EJB Module Project</td>
<td>15</td>
</tr>
<tr>
<td>Creating the QuickBookingBean Entity Bean</td>
<td>17</td>
</tr>
<tr>
<td>Adding the Implementation of the ejbCreate Method</td>
<td>23</td>
</tr>
<tr>
<td>Implementing the Business Logic</td>
<td>25</td>
</tr>
<tr>
<td>Creating and Implementing Auxiliary Classes</td>
<td>25</td>
</tr>
<tr>
<td>Creating a Java Project as a Source Container</td>
<td>26</td>
</tr>
<tr>
<td>The Constants Class</td>
<td>26</td>
</tr>
<tr>
<td>The QuickBookingModel JavaBean</td>
<td>28</td>
</tr>
<tr>
<td>The Exception Class QuickCarRentalException</td>
<td>31</td>
</tr>
<tr>
<td>Exporting the JAR File</td>
<td>32</td>
</tr>
<tr>
<td>Creating the Session Bean QuickOrderProcessorBean</td>
<td>33</td>
</tr>
<tr>
<td>Implementing the Session Bean Class</td>
<td>37</td>
</tr>
<tr>
<td>Throwing the Exception QuickCarRentalException</td>
<td>40</td>
</tr>
<tr>
<td>Implementing the saveBooking() Method</td>
<td>44</td>
</tr>
<tr>
<td>Implementing the cancelBooking() Method</td>
<td>45</td>
</tr>
<tr>
<td>Implementing the viewActiveBookings() Method</td>
<td>46</td>
</tr>
<tr>
<td>Creating a JAR</td>
<td>47</td>
</tr>
<tr>
<td>Editing Deployment Descriptors</td>
<td>48</td>
</tr>
<tr>
<td>Adding Descriptions to ejb-jar.xml</td>
<td>48</td>
</tr>
<tr>
<td>Adding Descriptions to persistent.xml</td>
<td>53</td>
</tr>
<tr>
<td>Adding Descriptions to ejb-j2ee-engine.xml</td>
<td>56</td>
</tr>
<tr>
<td>Creating a Jar File</td>
<td>56</td>
</tr>
<tr>
<td>Implementing a Web Application</td>
<td>57</td>
</tr>
<tr>
<td>Creating a Web Module Project</td>
<td>57</td>
</tr>
<tr>
<td>Creating the JSP quickCarRentalView</td>
<td>59</td>
</tr>
<tr>
<td>Adding Source Code</td>
<td>60</td>
</tr>
<tr>
<td>Creating the QuickReservationServlet Servlet</td>
<td>64</td>
</tr>
<tr>
<td>Adding Source Code to the Servlet</td>
<td>65</td>
</tr>
<tr>
<td>Creating a Web Archive</td>
<td>69</td>
</tr>
<tr>
<td>Editing Deployment Descriptors</td>
<td>69</td>
</tr>
<tr>
<td>Adding Descriptions to web.xml</td>
<td>69</td>
</tr>
</tbody>
</table>
Creating a J2EE-Based Car Rental Application

The Task
Using this tutorial, a Web application for renting cars will be implemented by means of the J2EE standard model.

Users should be able to choose the type of car, pickup and drop off points, and the start and end of the rental period using a Web browser. Finally, they should be able to make a booking. In addition, they should be able to display an overview of all active bookings. They should also be able to cancel existing bookings.

These functions will also be made available to external consumers as a Web service.

About the Implementation Steps
You will begin developing the car rental application by creating the data model on which it is based. To do this, you only need a single table that can record the reservation data. For this purpose you will first create a local table definition using a Dictionary project in the Developer Studio, and then build the associate archive (SDA), before deploying this to the J2EE Engine and creating a corresponding table in the database.

For accesses to table data, you will implement a container-managed persistence entity bean in an EJB module project. The implementation of the application’s business logic (creating a booking, displaying all available bookings, canceling a booking) will be included in a stateless session bean.

Along with the entity and session beans, you will also create and implement the necessary additional auxiliary classes. You will store these auxiliary classes externally in a separate Java project and package them in a JAR file.

To implement the Web client, you will create a JSP and a servlet in a Web module project. The JSP is responsible for displaying the view in the Web client, while the servlet is needed to pass the data to the JSP in a suitable form and to address the session bean with the requests from the JSP.

You will then generate a Web archive (WAR), which contains the Web resources and the associated deployment descriptors.

The two archives, JAR and WAR, are then combined in an Enterprise Application project, to form a kind of complete application. You must first deploy the resulting archive file (EAR) before you can start and test the car rental application.

In the last step, you will use the complete car rental application as a basis for defining a Web service, and thus make its functions available to other, external applications.
Objectives
By the end of this tutorial, you will be able to:

- Comprehend the general process flow when developing J2EE applications using the J2EE toolset in the SAP NetWeaver Developer Studio.
- Assign components of a J2EE application to the appropriate project types.
- Create database-independent definitions of tables as part of a project in the Developer Studio, create the associated archive, deploy the archive, and create the table definition in the database.
- Implement access to table records using entity beans of the type container-managed persistence.
- Create the business logic components and implement simple business methods.
- Implement simple user interfaces as J2EE Web clients using JSPs and servlets.
Generate the corresponding archives from the available EJB and Web resources.

Maintain frequently used entries for different deployment descriptors.

Bundle, deploy, and call all resources and archives in one complete J2EE application.

Using a session bean implementation as a starting point, define a Web service and make it available for external Web service clients.

Prerequisites

Systems, Installations, and Authorizations

☐ The SAP NetWeaver Developer Studio is installed on your computer.

☐ You can access the SAP J2EE Engine and you are connected to a database instance of the SAP DB.

☐ We also recommend that you install the SQL Studio (although this is optional), so that you can look at the tables you create in the SAP NetWeaver Developer Studio and if necessary edit or delete them.

Knowledge

☐ Java knowledge and basic knowledge of the J2EE programming model would be advantageous.

☐ You have acquired some basic experience with the J2EE toolset in the Developer Studio, for example by working through the Tutorial Writing Your First J2EE Application.

Next step:

Defining a Data Model [page 8]

Defining a Data Model

Before you start developing your car rental application, you must first create a suitable data model that will serve as a basis for this application. For didactic reasons, this data model will be kept quite simple. For storing the model data, you will therefore only need one table, which will manage persistent car rental data.

In this tutorial, you will learn how to create a table in the Java Dictionary and fully specify it in the appropriate editor. You will create a corresponding archive from this table definition, which is initially only local. Using the Software Deployment Manager (SDM), you will then be able to deploy this archive on the SAP J2EE Engine and create a physical table on the database instance of the SAP DB.

Next step:

Creating a Dictionary Project [page 9]
Creating a Dictionary Project

To be able to create tables, you first need an appropriate project in the Developer Studio. A Dictionary project allows you to create and manage the table definition locally. You create a suitable project framework for the new Dictionary project using the appropriate wizard.

Prerequisites

- You have launched the SAP NetWeaver Developer Studio.

Procedure

1. Choose File → New → Project to start the New Project wizard.
2. In the wizard, select the Dictionary category (in the left pane), followed by Dictionary Project (in the right pane). Choose Next.
3. Give your project the name QuickCarRentalDictionary and leave the default settings for Project contents and Project language unchanged.
4. Choose Finish to create the project.

Result

The wizard generates a standard structure for your new Dictionary project and creates the project folder QuickCarRentalDictionary in the default workspace.

It automatically opens the Dictionary perspective and displays the project structure in the Dictionary Explorer or in the Navigator view.
Next step:
Creating a Table [page 10]

Creating a Table

Prerequisites

The structure of your project QuickCarRentalDictionary is currently displayed in the Dictionary Explorer.

Procedure

1. Expand the project node QuickCarRentalDictionary → Dictionaries → Local Dictionary and open the context menu for Database Tables.
2. Choose Create Table to start the appropriate wizard.
3. Enter the name QCR_RESERVATION for the table.

<table>
<thead>
<tr>
<th>New Database Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enter the name of the new Database Table</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table Name</th>
<th>QCR_RESERVATION</th>
</tr>
</thead>
</table>


Result

The wizard creates a table definition in your workspace folder and inserts an additional node with the name of the table into the project structure.

Next step:
Adding Table Columns [page 11]
Adding Table Columns

Prerequisites

- The structure of your project QuickCarRentalDictionary is currently displayed in the Dictionary Explorer.
- You have created the table QCR_RESERVATION.

Procedure

1. If not already done so, open the table editor by double-clicking the name of the newly created table QCR_RESERVATION in the Dictionary Explorer.
2. Enter the table description Table contains all reservation data for QuickCarRental example in the Description field.
3. Choose the icon Add Column to create the first table column.
   A wizard is started in which you can fully specify the table column.
4. On the Built-in Type tab, assign the following values to the selected properties:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Built-in Type</th>
<th>Length</th>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>string</td>
<td>10</td>
<td>activated</td>
<td>unique identifier for reservation</td>
</tr>
</tbody>
</table>

5. Confirm by choosing Finish
In creating this first column with the name **ID**, you have created the key element of the table.

6. Repeat the last steps for the other columns. Define the columns as follows:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Built-in Type</th>
<th>Length</th>
<th>Not Null</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEHICLETYPEID</td>
<td>string</td>
<td>20</td>
<td>activated</td>
<td>type of the car</td>
</tr>
<tr>
<td>RESERVATIONDATE</td>
<td>timestamp</td>
<td></td>
<td>activated</td>
<td>creation date for reservation data</td>
</tr>
<tr>
<td>DATEFROM</td>
<td>timestamp</td>
<td></td>
<td>activated</td>
<td>date, when reservation period is started</td>
</tr>
<tr>
<td>DATETO</td>
<td>timestamp</td>
<td></td>
<td>activated</td>
<td>date, when reservation period ends</td>
</tr>
<tr>
<td>PICKUPLOCATION</td>
<td>string</td>
<td>30</td>
<td>activated</td>
<td>pickup location</td>
</tr>
<tr>
<td>DROPOFFLOCATION</td>
<td>string</td>
<td>30</td>
<td>activated</td>
<td>dropoff location</td>
</tr>
<tr>
<td>STATUS</td>
<td>string</td>
<td>20</td>
<td>activated</td>
<td>status of reservation</td>
</tr>
</tbody>
</table>

Note that the assignment of the **JDBC Type** property results from the specifications for **Built-in Type** and **Length** and is automatically set by the wizard.

7. Save the new metadata by choosing the icon (Save All Metadata) from the toolbar.

**Result**

You have created the table **QCR_RESERVATION** locally in your Dictionary project and fully specified it with table columns. As a component of the Java Dictionary, the table has a database-independent definition.
Creating an SDA

You require an SDA file to allow you to transfer the table definitions from the current Dictionary project to a database instance of the SAP DB. An SDA (Software Delivery Archive) is an archive file that represents a deployable unit of the Dictionary project.

Prerequisites

- The table `QCR_RESERVATION` is fully defined.
- The structure of your project `QuickCarRentalDictionary` is currently displayed in the Dictionary Explorer.

Procedure

1. If you have not already done so, save the metadata for your project in its current state.
2. Select the project node `QuickCarRentalDictionary` in the Dictionary Explorer, and choose Rebuild Project.
3. Choose Create Archive from the context menu of the project node.

Checking the Result

4. To check that the new archive was created for the project, switch to the Navigator view (tab adjacent to the Dictionary Explorer).
5. Expand the project node `QuickCarRentalDictionary`.
   Here, you should see a file called `QuickCarRentalDictionary.sda`.

Next step: Deploying an SDA [page 14]
Deploying an SDA

To create a physical representation of the table from the local table definition on a database instance, you must deploy the newly created archive file (SDA). However, before you do this, make sure all the following prerequisites are met:

**Prerequisites**

- You have made sure that that archive file (SDA) was created successfully.
- You have made sure that the SAP J2EE Engine has been launched and that you are connected to a database instance of the SAP DB.
  
  To do this, refer to [Starting and Stopping the SAP J2EE Engine](extern)
- If you want to ensure that your tables were correctly created on the database instance of the SAP DB, you also require the SQL Studio. This must be already installed. However, this prerequisite is optional.

**Procedure**

1. Select the project node (`QuickCarRentalDictionary`) in the `Dictionary Explorer`.
2. Open the context menu and choose `Deploy`.

If the deployment, which is carried out by the Software Deployment Manager, is successful, the appropriate message is displayed in the `Deploy Output View`.

**Result**

The SDA file is deployed and the table is created in the database instance. In so doing, the table definition from the Java Dictionary is converted to a definition of the relevant database – in this case, the SAP DB.

**Checking the Result**

You do not have to carry out the following check. It is only really necessary if you have installed the SQL Studio.

To make sure that the table was correctly created on the database instance, you can log on to the SQL Studio and view the list of all tables currently deployed on the J2EE Engine. Proceed as follows:

1. Call the SQL Studio, for example by choosing `Start → Programs → SAP DB → SQL Studio` or directly using the start file `<drive>\sapdb\SQLStudio\SQLSto.exe`.
2. Choose `Connect`.
   
   A logon window is displayed.
3. Enter the logon data (Server, Database, User and Password) and choose `OK` to confirm your entries.
4. In the navigation tree on the left, open the `Tables` folder and expand the user node (for example `SAPC11DB`).
   
   The system now displays all tables that belong to the user.
5. Find the table `QCR_RESERVATION`. You may have to choose `Continue` to display all tables.
6. Open the table by double-clicking **QCR_RESERVATION**.

   In the right frame, the SQL Studio displays the columns of the table.

   ![Table](image)

   **Table Name**: QCR_RESERVATION
   **Based on**: QCR_RESERVATION (Table)

   ![Columns](image)

   **Next step:**
   Implementing Data Access [page 15]

---

**Implementing Data Access**

In this unit, you will start by learning how to create an EJB Module project named **QuickCarRentalEjb**.

You will then create an entity bean **QuickBookingBean** of the type *container-managed persistence (CMP)*. You will define this bean in such a way that it will be associated with the table **QCR_RESERVATION**, which you created previously in the database. You will create a corresponding CMP field for each column in this table. For each bean instance, these fields will contain the data for the associated data record. You will then be able to create, edit, and delete data records in the car rental application using the appropriate access methods for these CMP fields.

You will take into account the data needed to create a new table record when implementing the **ejbCreate** method.

So that the CMP fields can be accessed from the business logic, the last step in this unit will entail propagating the access methods from the local interface.

**Next step:**
Creating an EJB Module Project [page 15]

---

**Creating an EJB Module Project**

To create and work with entity beans, you first need an appropriate project in the Developer Studio. For this reason, you will use the wizard to generate a suitable project framework for the new EJB project. Once you have set up this framework, you can create and implement all
the sources of the EJB, especially the entity beans. The EJB Module project can contain both session beans and entity beans.

EJB Module project also contains the deployment descriptors that will be included in the JAR file.

**Prerequisites**

- You have launched the SAP NetWeaver Developer Studio.

**Procedure**

1. Choose *File* → *New* → *Project* to start the *New Project* wizard.
2. In the wizard, select the *J2EE* category (in the left pane), followed by *EJB Module Project* (in the right pane). Choose *Next*.
3. Give your EJB project the name *QuickCarRentalEjb* and leave the default settings for *Project contents* unchanged.
4. Choose *Finish*.

**Result**

The wizard generates a standard project structure for your new J2EE project and creates the project folder *QuickCarRentalEjb* in your default workspace.

The new project node appears in the J2EE Explorer. It contains the deployment descriptor files *ejb-j2ee-engine.xml, ejb-jar.xml*.

**Next step:**

Creating the QuickBookingBean Entity Bean [page 17]
Creating the QuickBookingBean Entity Bean

You will now create a new entity bean named the QuickBookingBean. This bean will manage the data in the table QCR_RESERVATION. In particular, it will enable you to create new records and change or delete existing data.

Prerequisites

☐ The structure of your project QuickCarRentalEjb is currently displayed in the J2EE Explorer.

Procedure

Specifying the general properties of the EJB

1. In the J2EE Explorer, to open the relevant wizard, choose New → EJB… from the context menu of the project node QuickCarRentalEjb.
2. In the New EJB wizard that appears, assign the following values to the properties of the new EJB:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EJB Name</td>
<td>QuickBookingBean</td>
</tr>
<tr>
<td>EJB Project</td>
<td>QuickCarRentalEjb</td>
</tr>
<tr>
<td>Bean Type</td>
<td>Entity Bean</td>
</tr>
<tr>
<td>Default EJBPackage</td>
<td>com.sap.examples.quickcarrental</td>
</tr>
<tr>
<td>Generate default interfaces</td>
<td>activated</td>
</tr>
</tbody>
</table>

3. Choose Next.

The wizard displays the screen containing names of the remote and the local interfaces.

4. Accept the suggested values and choose Next.

In the wizard screen that appears, you will be able to specify the persistence properties.

5. Choose Container Managed Persistence as the Persistence type.

Specifying the primary key and other CMP fields

Every instance of an entity bean is identified by means of a unique key. For this purpose, you will first create a field named bookingId and define it as the primary key. Each instance of the entity bean QuickBookingBean will be uniquely identified with this key field. You will then add other CMP fields, which store the associated records for the table QCR_RESERVATION.

You will map the table columns to the CMP fields of the entity bean as follows:

<table>
<thead>
<tr>
<th>Column in QCR_RESERVATION</th>
<th>Entity bean field (CMP field)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>bookingId</td>
</tr>
<tr>
<td>VEHICLETYPEID</td>
<td>vehicleTypeId</td>
</tr>
<tr>
<td>RESERVATIONDATE</td>
<td>reservationDate</td>
</tr>
<tr>
<td>DATEFROM</td>
<td>dateFrom</td>
</tr>
</tbody>
</table>
6. To add a field, in the left pane in the Wizard, select the Persistent Fields node and choose the Add button.

   To the right input fields appear, where you can specify the field.

7. Enter the name `bookingId` and accept the suggested value `java.lang.String` for the type.

8. To define `bookingId` as the primary key field, in the left pane, choose the node `PrimaryKey`.

9. Choose `Simple` and assign `bookingId` as the primary key field.

10. To add the next field, in the left pane in the Wizard, select the Persistent Fields node again and choose the Add button.
11. Enter the name `vehicleTypeId` and accept the suggested value `java.lang.String` for the type.

12. To add the next field, in the left pane in the Wizard, select the `Persistent Fields` node and choose `Add` again.

13. Enter the name `reservationDate` and assign it the type `java.util.Date`.

   If you choose the `Full Qualified Name` for the date using the `Choose` button, make sure that you select the correct package for this class when you search for the date class (by entering `Date`) – in this case, `java.util.Date`.

14. Add the other fields in a similar way, assigning them the following names and types:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type (Full Qualified Name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dateFrom</td>
<td>java.util.Date</td>
</tr>
<tr>
<td>dateTo</td>
<td>java.util.Date</td>
</tr>
<tr>
<td>pickupLocation</td>
<td>java.lang.String</td>
</tr>
</tbody>
</table>
15. Choose Next.

In the wizard dialog that appears, you can define the methods of the entity bean.

**Editing the **ejbCreate** method**

You need to extend the standard method **ejbCreate**, so that the fields you have created – bookingId, vehicleTypeId, reservationDate, dateFrom, dateTo, pickupLocation, dropoffLocation and status – are passed to this method as parameters.

16. To add new parameters to the bean method **ejbCreate**, select the node of the same name under Create Methods in the tree structure.

Input fields appear to the right of the tree. You can change the method definition there.

17. Under Parameters, choose Add.

18. Enter the name bookingId and the type **java.lang.String** for the first parameter.

19. To add the other parameters, repeat the last two steps as appropriate. Use the same name and type assignment as specified above:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type (Full Qualified Name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>vehicleTypeId</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>reservationDate</td>
<td>java.util.Date</td>
</tr>
<tr>
<td>dateFrom</td>
<td>java.util.Date</td>
</tr>
<tr>
<td>dateTo</td>
<td>java.util.Date</td>
</tr>
<tr>
<td>pickupLocation</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>dropoffLocation</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>status</td>
<td>java.lang.String</td>
</tr>
</tbody>
</table>

Always create parameters in the correct order, since this is the order used when **ejbCreate** is called.
Creating the finder method \texttt{ejbFindByStatus}

In the steps that follow you create an additional finder method named \texttt{ejbFindByStatus}. This method returns the bean instances of all the database entries that match a status value that has been passed as a parameter.

20. To create the new finder method \texttt{ejbFindByStatus}, select the \textit{Finder Methods} node in the tree display (left) and choose the \textit{Add} button. Input fields appear to the right of the tree. You can define the new finder method there.

21. Enter the name \texttt{ejbFindByStatus} and assign it the return type \texttt{java.util.Collection}.

22. Under Parameters, choose \textit{Add}, to add the parameter with the name \texttt{status} and the type \texttt{java.lang.String}.
23. Choose Finish to create the entity bean.

Result

The wizard generates the following: the bean class QuickBookingBean; the local interface QuickBookingLocal; remote interface QuickBooking; local home interface QuickBookingLocalHome and home interface QuickBookingHome, with the appropriate set and get, finder, and EJB default methods.

In the J2EE Explorer, a new substructure named QuickBookingBean is inserted automatically under QuickCarRental → ejb-jar.xml.

Additionally a new deployment descriptor persistent.xml has been created in the EJB Module project. You will need it later to assign the data source to the entity bean.
Next step: Adding the Implementation of the ejbCreate Method [page 23]

Adding the Implementation of the ejbCreate Method

When implementing the method `ejbCreate`, the only thing you need to take into account is the data needed to create a new table record. In this case, all the parameters passed when the `ejbCreate` method is called must be passed to the associated CMP fields of the entity bean. You will implement this using the corresponding set methods. The EJB container then takes over the task of saving the record in the table.

Prerequisites

- You have created the entity bean `QuickBookingBean`.
- The structure of your project `QuickCarRentalEjb` is currently displayed in the J2EE Explorer.

Procedure

1. In the project structure, expand the `QuickCarRentalEjb → ejb-jar.xml` node and double-click the name `QuickBookingBean`.
   
   An editor appears, in which you can navigate to any of the bean's components.

2. Choose Navigate to Bean Class.
The Java Source Editor opens and displays the generated source code of the bean class.

3. Navigate to the implementation of the method `ejbCreate`.

   The Outline view allows you to navigate quickly and easily. To navigate to the implementation of the method `ejbCreate`, simply click the method name in the Outline view.

   ![Outline view](image)

4. Add the following code after `// TODO : Implement`:

   ```java
   setBookingId(bookingId);
   setReservationDate(reservationDate);
   setDateFrom(dateFrom);
   setDateTo(dateTo);
   setStatus(status);
   setVehicleTypeId(vehicleTypeId);
   setPickupLocation(pickupLocation);
   setDropoffLocation(dropoffLocation);
   ```

5. If necessary, correct the formatting of these lines by choosing Source → Format from the context menu.

6. Save the contents of the editor by choosing the appropriate icon from the toolbar.

**Result**

You have added calls of the set methods to the method `ejbCreate` of the bean class. The passed values can now be assigned to the associated CMP fields of the bean.
Implementing the Business Logic

Now that you have implemented data access using an entity bean in the previous teaching unit, you will continue in this unit by implementing the business logic of the car rental application.

For this purpose, you will create a stateless session bean named QuickOrderProcessorBean in your EJB project, and then add and implement the necessary business methods. QuickOrderProcessorBean should be able to store the booking data entered by the user, check it, if necessary convert it to the appropriate objects, and then pass it to the entity bean to be saved. In addition, the user should be able to delete, change, and format records for display in the client.

Creating and Implementing Auxiliary Classes

Before you start to construct the business logic for your car rental application and create a session bean, you will first create three auxiliary classes. These classes contain implementations that will be used at different points in the application – including from within different project types (such as a EJB Module project or Web Module project) – or even from a Web service.

You will create these auxiliary classes in a separate Java project that only acts as a source container.

<table>
<thead>
<tr>
<th>Class</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constants</td>
<td>This class contains the definition of constants that are needed at different points in the application.</td>
</tr>
<tr>
<td>QuickBookingModel</td>
<td>You need this JavaBean class to store the data from the entity bean QuickBookingBean in a form that can be displayed. QuickBookingModel should be passed directly to a servlet or Web service so that the booking data can be processed.</td>
</tr>
<tr>
<td>QuickCarRentalException</td>
<td>You need this exception class to handle the specific exceptions thrown by the car rental application.</td>
</tr>
</tbody>
</table>

Creating a Java project as a source container [page 26]
Creating a Java Project as a Source Container

Because it is the best to store all auxiliary classes externally, you will need a Java project as a separate source container. This enables a clear separation between EJBs and the required auxiliary classes.

Procedure

1. Choose the menu File → New → Project… to start the New Project Wizard.
2. Select Java (in the left pane), followed by Java Project (in the right pane). Choose Next.
3. In the Project name field, type the name Helperclasses for your new Java project and leave the default settings for Project contents unchanged.
   A dialog box appears asking you whether you want to switch to Java Perspective.
5. Choose Yes.

Result

The wizard generates a new Java project that will serve as a source container for all your auxiliary classes.

Next step:

The Constants Class [page 26]

The Constants Class

In the Constants class, you define constants that the application needs. These constants will be used by the beans in the EJB project or by the servlet and the JSP in the Web project.

You will declare the following Constants in the constants class:

<table>
<thead>
<tr>
<th>Constants</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATUS_ACTIVE,</td>
<td>These two constants indicate the status of a booking. They are used in the business logic of a session bean to inform the entity bean of the</td>
</tr>
<tr>
<td>STATUS_CANCELLLED</td>
<td>status of the current booking.</td>
</tr>
<tr>
<td>ACTION_SAVE,</td>
<td>These constants specify the action that the user at the client end has just carried out. The JSP and the servlet need them, so that they</td>
</tr>
<tr>
<td>ACTION_CANCEL</td>
<td>know what button the user has just clicked.</td>
</tr>
<tr>
<td>RESERVATIONS,</td>
<td>These constants are used to store, in the session bean, the context data for the booking and for messages displayed for the user.</td>
</tr>
<tr>
<td>CLIENT_MESSAGE</td>
<td>Both constants are initialized in the session bean.</td>
</tr>
</tbody>
</table>
LOCATION, VEHICLE_TYPE

These constants contain an array with data for the reservation location and the car type.

**Prerequisites**

- The structure of the Java project **Helperclasses** is currently displayed in the Package Explorer.

**Procedure**

**Creating the Java class Constants in the Java project**

1. In the Package Explorer, select the project node **Helperclasses**.
2. From the context menu, choose **New → Class**.
3. In the **New Java Class** wizard that appears, assign the following values:

<table>
<thead>
<tr>
<th>Source Folder</th>
<th>Helperclasses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>com.sap.engine.examples.util</td>
</tr>
<tr>
<td>Name</td>
<td>Constants</td>
</tr>
<tr>
<td>Modifiers</td>
<td>public</td>
</tr>
<tr>
<td>Superclass</td>
<td>java.lang.Object</td>
</tr>
<tr>
<td>Inherited abstract methods</td>
<td>activated</td>
</tr>
</tbody>
</table>

Which method stubs would you like to create?

- [ ] public static void main(String[] args)
- [ ] Constructors from superclass
- [x] Inherited abstract methods
4. Leave all the other suggested values unchanged and choose Finish.

The wizard creates the package com.sap.engine.examples.util, locates the new class Constants there and opens the Java Editor.

**Defining Constants**

5. In the Java Editor, add the following constant definitions to the body of the class:

```java
public static final java.text.SimpleDateFormat FORMATTER = new java.text.SimpleDateFormat("dd.MM.yyyy");

public static final java.lang.String STATUS_ACTIVE = "ACTIVE";
public static final java.lang.String STATUS_CANCELLED = "CANCELLED";

public static final int ACTION_SAVE = 0;
public static final int ACTION_CANCEL = 1;

public static final String RESERVATIONS = "rent.car.customer.reservations";
public static final String CLIENT_MESSAGE = "rent.car.customer.message";

public static final String[] LOCATION = 
{ "Heidelberg","Frankfurt","Kassel","Berlin","Mannheim","Leipzig","Hamburg " };
public static final String[] VEHICLE_TYPE = 
{ "Economy","Compact","Intermediate","Full Size","Premium","Luxury","Convertible","Mini Van" };
```

6. If necessary, correct the formatting of these lines by choosing Source → Format from the context menu.

7. Save the contents of the editor by choosing the appropriate icon in the toolbar.

**Result**

You have created and completely implemented the auxiliary class Constants.

**Next step:**

The QuickBookingModel JavaBean [page 28]

---

The QuickBookingModel JavaBean
The JavaBean class `QuickBookingModel` should be able to store all the data that are needed for the client to display a booking. `QuickBookingModel` will be initialized in the session bean and passes the booking data either to the client or to the Web service.

For this purpose, the `QuickBookingModel` must contain a field for each relevant value. The fields will be named `bookingId`, `vehicleTypeId`, `dateFrom`, `dateTo`, `pickupLocation`, and `dropoffLocation`, similar to their definition in the entity bean. However, unlike the definition in the entity bean, the fields `dateFrom` and `dateTo` are defined as Strings in this class. This ensures that the data in the client can be displayed at once. In addition, the field `price` is defined here. This field contains the total price of the booking and is calculated by the business logic in the session bean. In contrast, the values from the fields `reservationDate` and `status` from the entity bean are not needed here and are not included in the JavaBean class.

Moreover, `QuickBookingModel` must be serializable, so that it can be passed to a Web service. That is, the bean class must implement the `Serializable` interface.

**Prerequisites**

- The structure of the Java project `Helperclasses` is currently displayed in the `Package Explorer`.

**Procedure**

**Creating the JavaBean `QuickBookingModel` in the Java project.**

1. In the Package Explorer, select the project node `Helperclasses`.
2. From the context menu, choose `New → Other...`
3. Select `Java` (in the left pane), followed by `Class` (in the right pane). Choose `Next`.
4. In the `New Java Class` wizard that appears, assign the following values:

<table>
<thead>
<tr>
<th>Source Folder</th>
<th>Helperclasses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>com.sap.engine.examples.util</td>
</tr>
<tr>
<td>Name</td>
<td>QuickBookingModel</td>
</tr>
<tr>
<td>Modifiers</td>
<td>public</td>
</tr>
<tr>
<td>Superclass</td>
<td>java.lang.Object</td>
</tr>
<tr>
<td>Interfaces</td>
<td>java.io.Serializable</td>
</tr>
<tr>
<td>Inherited abstract methods</td>
<td>activated</td>
</tr>
</tbody>
</table>

To add the `java.io.Serializable` interface, choose `Add` and enter `Serializable` in the dialog box. Then choose `java.io` as the `Qualifier`.

Creating a J2EE-Based Car Rental Application 29
5. Leave all the other suggested values unchanged and choose Finish.

The wizard creates the new JavaBean class QuickBookingModel in the Java Editor.

**Implementing the JavaBean QuickBookingModel**

At this point, you will define the necessary fields and generate the associated get/set methods.

6. In the Java Editor, add the following field definitions to the body of the class:

```java
private java.lang.String bookingId;
private java.lang.String vehicleType;
private java.lang.String dateFrom;
private java.lang.String dateTo;
private java.lang.String pickupLocation;
private java.lang.String dropoffLocation;
private java.lang.String price;
```

7. In the Java Editor, select all the lines with the field definitions you have just added and click the right mouse button.

8. Choose Source → Generate Getter and Setter from the context menu.

9. In the selection dialog that appears, choose Select All and confirm your choice by choosing OK.

The appropriate get and set methods are generated automatically for all the fields.
10. If you need to correct the formatting, choose the *Format* function from the context menu of the Java Editor.

11. Save the contents of the editor by choosing the appropriate icon in the toolbar.

**Result**

You have created and completely implemented the JavaBean class *QuickBookingModel*.

**Next step:**

The Exception Class *QuickCarRentalException* [page 31]

---

**The Exception Class *QuickCarRentalException***

The purpose of the exception class *QuickCarRentalException* is to supply the car rental application with all the application-specific exceptions. Each exception should be wrapped as a *QuickCarRentalException*, so that an exception that is appropriate for the car rental application is always delivered to the client.

*QuickCarRentalException* extends *java.lang.Exception*.

**Prerequisites**

- The structure of the Java project *Helperclasses* is currently displayed in the *Package Explorer*.

**Procedure**

**Creating *QuickCarRentalException* in the Java project.**

1. In the *Package Explorer*, select the project node *Helperclasses*.
2. Form the context menu, choose *New* → *Other*...
3. Select *Java* (in the left pane), followed by *Class* (in the right pane). Choose *Next*.
4. In the *New Java Class* wizard that appears, assign the following values:

<table>
<thead>
<tr>
<th>Source Folder</th>
<th>Helperclasses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>com.sap.engine.examples.util</td>
</tr>
<tr>
<td>Name</td>
<td>QuickCarRentalException</td>
</tr>
<tr>
<td>Modifiers</td>
<td>public</td>
</tr>
<tr>
<td>Superclass</td>
<td>java.lang.Exception</td>
</tr>
<tr>
<td>Inherited abstract methods</td>
<td>activated</td>
</tr>
</tbody>
</table>

5. Leave all the other suggested values unchanged and choose *Finish*.

The wizard creates the new file *QuickCarRentalException.java* and opens the Java Editor.
Adding the superclass constructors

6. In the Java Editor position the cursor anywhere in the source code and click the right mouse button.

7. Choose Source → Add Constructors from Superclass from the context menu.

The following code is generated:

```java
/**
 * @param message
 */
public QuickCarRentalException(String message) {
    super(message);
    // TODO Auto-generated constructor stub
}

/**
 * @param cause
 */
public QuickCarRentalException(Throwable cause) {
    super(cause);
    // TODO Auto-generated constructor stub
}

/**
 * @param message
 * @param cause
 */
public QuickCarRentalException(String message, Throwable cause) {
    super(message, cause);
    // TODO Auto-generated constructor stub
}
```

8. Delete the generated // TODO lines.

9. Save the contents of the editor by choosing the appropriate icon from the toolbar.

Result

You have created and completely implemented the auxiliary class QuickCarRentalException.

Next step:

Exporting the JAR file [page 32]

Exporting the JAR File
To add your auxiliary classes to the EAR file for the later deployment step, you first need to pack all the resources into a JAR file. In this step, you will utilize the export wizard that allows you to create a JAR file and export it to any file system path.

**Prerequisites**

- □ The structure of the Java project *Helperclasses* is currently displayed in the *Package Explorer*.

**Procedure**

1. In the *Package Explorer*, select the project node *Helperclasses*.
2. From the context menu, choose *Export*...
3. Select *JAR file* and choose *Next*.
4. In the JAR export wizard that appears, select all sources of the project *Helperclasses* to be exported.
5. Choose the *Browse*… button to specify the export destination. To do this, select the project folder *Helperclasses* from within your workspace and enter *helper.jar* as the name of the new JAR file.
6. Leave the other settings unchanged and choose *Finish*.

**Result**

The wizard creates a new JAR file *helper.jar* and adds an appropriate node to the project structure. You will add this JAR file to the Enterprise Application project later when creating the EAR file.

![Package Explorer](image)

Next step:

- *Creating the Session Bean QuickOrderProcessorBean* [page 33]

**Creating the Session Bean**

**QuickOrderProcessorBean**

You will now create a stateless session bean named *QuickOrderProcessorBean* in your EJB Module project *QuickCarRentalEJB*. This session bean will implement the business logic of the car rental application. In particular, you will add the business methods to the bean that enable the application to save bookings, cancel bookings, and display all active bookings at the client end.
**Prerequisites**

- You have created the EJB project `QuickCarRentalEjb`. The structure of this project is currently displayed in the *J2EE Explorer*.
- You have created the necessary auxiliary classes in the Java project `Helperclasses`.

**Procedure**

**Specifying the project dependencies of the EJB project.**

Since the auxiliary classes from the Java project are to be referenced in the EJBs, you will need to add the Java project `Helperclasses` as a required project to the EJB project.

1. In the J2EE Explorer, select the project node `QuickCarRentalEjb` and open the context menu.
2. Choose *Properties*.
3. Choose *Java Build Path* in the left frame. In the right frame, choose the *Projects* tab.
4. Select `Helperclasses` and confirm with *OK*.

![Properties for QuickCarRentalEjb](image)

**Specifying the general properties of the EJB**

1. In the J2EE Explorer, to open the relevant wizard, choose *New → EJB…* from the context menu of the project node `QuickCarRentalEjb`.
2. In the *New EJB* wizard that appears, assign the following values to the properties of the new EJB:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EJB Name</td>
<td><code>QuickOrderProcessorBean</code></td>
</tr>
<tr>
<td>EJB Project</td>
<td><code>QuickCarRentalEjb</code></td>
</tr>
<tr>
<td>Beantype</td>
<td><code>Stateless Session Bean</code></td>
</tr>
<tr>
<td>Default Package</td>
<td><code>com.sap.examples.quickcarrental</code></td>
</tr>
<tr>
<td>Generate default interfaces</td>
<td><strong>activated</strong></td>
</tr>
</tbody>
</table>

3. Choose *Next*.
4. Accept the settings in the next screen and choose *Next*.
   
   In the wizard dialog that appears, you can add the necessary business methods.

**Creating business methods**

To make the specific features of the business logic available, you will need the following business methods:
### Business Method

<table>
<thead>
<tr>
<th>Method</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>saveBooking</td>
<td>Saves the booking data.</td>
</tr>
<tr>
<td>cancelBooking</td>
<td>Implements the cancellation of a booking and sets its status to CANCELLED.</td>
</tr>
<tr>
<td>viewActiveBookings</td>
<td>Provides data for all active bookings to the client or to a Web service.</td>
</tr>
</tbody>
</table>

---

**Method saveBooking**

5. To add the first method, `saveBooking`, select the *Business Methods* folder in the left pane of the wizard and choose *Add*.

   To the right, input fields appear, where you can specify the method.

6. Enter the name `saveBooking` and as the return type assign the auxiliary class `com.sap.engine.examples.util.QuickBookingModel`.

7. Add the following parameters to the method:

<table>
<thead>
<tr>
<th>Name of the parameter</th>
<th>Type (Full Qualified Name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>vehicleTypeId</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>dateFromString</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>dateToString</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>pickupLocation</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>dropoffLocation</td>
<td>java.lang.String</td>
</tr>
</tbody>
</table>

---

![Diagram of the method saveBooking and parameters](image)
Method cancelBooking

8. To add the method cancelBooking, select the Business Methods folder again in the left pane of the wizard and choose Add.

9. Enter the name cancelBooking and assign the return type java.lang.String.

10. Add the parameter bookingId of type java.lang.String to this method.

Method viewActiveBookings

11. Add the third method, viewActiveBookings. Enter the following values:

<table>
<thead>
<tr>
<th>Name</th>
<th>viewActiveBookings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return Type</td>
<td>com.sap.engine.examples.util.QuickBookingModel</td>
</tr>
<tr>
<td>Array</td>
<td>activated</td>
</tr>
<tr>
<td>Array/Input field</td>
<td>1  (for a one-dimensional array)</td>
</tr>
<tr>
<td>Parameters</td>
<td>None</td>
</tr>
</tbody>
</table>

12. Choose Finish to create the session bean.

Result

The wizard generates: the bean class QuickOrderProcessorBean; the local interface QuickOrderProcessorLocal; remote interface QuickOrderProcessor; local home interface QuickOrderProcessorLocalHome and home interface QuickOrderProcessorBeanHome, with the appropriate EJB default methods and the three business methods.

In the J2EE Explorer, a new substructure named QuickOrderProcessorBean is inserted automatically under QuickCarRentalEjb → ejb-jar.xml.
Implementing the Session Bean Class

Before you actually implement the business methods, you need to carry out some preparation.

First, you will add a variable named bookingHome, which will be instantiated with the entity bean QuickBookingBean. Whenever an instance of the QuickOrderProcessorBean is created, this variable will also be instantiated with the local home interface of the entity bean. For this reason, you will initialize the variable bookingHome in the method ejbCreate().

In addition, you will add some private auxiliary methods to the standard implementation of the bean class:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>generateId()</td>
<td>You use this method to generate a booking ID.</td>
</tr>
<tr>
<td>getBookingModel()</td>
<td>You need this method whenever the JavaBean object QuickBookingModel is to be filled with the data from the entity bean QuickBookingBean.</td>
</tr>
<tr>
<td>getDate()</td>
<td>You need this method to get the corresponding date object from a string entered at the client end.</td>
</tr>
</tbody>
</table>

Prerequisites

- You have created the session bean QuickOrderProcessorBean.
- The structure of your project QuickCarRentalEjb is currently displayed in the J2EE Explorer.

Procedure

1. In the project structure, expand the QuickCarRentalEjb \(\rightarrow\) ejb-jar.xml node and double-click the name QuickOrderProcessorBean, if you have not already done so.
2. In the overview that appears, choose **Navigate to Bean Class**. The Java Editor opens and displays the generated source code of the bean class `QuickOrderProcessorBean`.

3. Add the private variable `bookingHome` to the start of the body of the class:

   ```java
   public class QuickOrderProcessorBean implements SessionBean {

   private QuickBookingLocalHome bookingHome;
   }
   ```

4. Then, at the end of the class body (after `ejbCreate()`), add the following implementations for the methods `generateId()`, `getBookingModel()`, and `getDate()`:

   ```java
   /**
    * Create Method.
    */
   public void ejbCreate() throws CreateException {
      // TODO : Implement
   }

   private String generateId() {
      Date temp = new Date();
      String stemp = String.valueOf(temp.hashCode());
      if (stemp.length() < 10)
         return stemp.substring(1, stemp.length());
      else
         return stemp.substring(1, 9);
   }

   private QuickBookingModel getBookingModel(QuickBookingLocal booking) {
      QuickBookingModel data = new QuickBookingModel();
      data.setBookingId(booking.getBookingId());
      data.setPickupLocation(booking.getPickupLocation());
      data.setVehicleType(booking.getVehicleTypeId());
      data.setDateFrom(Constants.FORMATTER.format(booking.getDateFrom()));
      data.setDateTo(Constants.FORMATTER.format(booking.getDateTo()));
      data.setDropoffLocation(booking.getDropoffLocation());
      data.setPrice(Float.toString(40*days) + " EUR");
   }
   ```
private Date getDate(java.lang.String dateString) throws QuickCarRentalException{

try{
    return Constants.FORMATTER.parse(dateString);
}

catch(java.text.ParseException pe){
    pe.printStackTrace();
    throw new QuickCarRentalException("The date has to be in format dd.MM.yyyy, for example 21.04.2004");
}
}

Make sure that the method used here, generateId(), is not used to implement an ID in a live system. In this case, we are simply simulating generating an ID. In a live system, always use the correct ID generation process.

5. If necessary, correct the formatting of these lines by choosing Source → Format from the context menu.

6. To add the import statements, position the cursor anywhere in the Java Editor and choose Source → Organize Imports from the context menu.
   Since the class Date is available twice, you will now be asked which package you want to use for Date.

7. Choose the package java.util and confirm by choosing Finish.

8. If you are asked to choose between different Constants classes, select com.sap.engine.examples.util.Constants

   The following import statements are added to the source code:

   ```java
   package com.sap.engine.examples.ejb.quickcarrental;
   import java.util.Date;
   import javax.ejb.CreateException;
   import javax.ejb.SessionBean;
   import javax.ejb.SessionContext;
   import com.sap.engine.examples.util.Constants;
   import com.sap.engine.examples.util.QuickBookingModel;
   import com.sap.engine.examples.util.QuickCarRentalException;
   ```
The Java Editor no longer displays any errors.

9. Navigate to the implementation of the method `ejbCreate()` and replace // TODO : Implement with the following code:

```java
try {
    Context ctx = new InitialContext();
    bookingHome =
        (QuickBookingLocalHome) ctx.lookup("java:comp/env/QuickCarRental/QuickBookingBean");
}
catch (NamingException e) {
    throw new CreateException(e.getMessage());
}
```

10. If necessary, correct the formatting of these new lines of code.

11. To add the appropriate import statements, choose Source → Organize Imports from the context menu.

Since the class `Context` occurs twice, you will now be asked which package you want to use for `Context`.


The appropriate import statements are added to the source code.

13. Save the contents of the editor by choosing the appropriate icon from the toolbar.

**Result**

The Developer Studio updates and compiles your project sources. (Note: Compilation only occurs if you are using the Workbench standard settings.) Your Tasks view should no longer show any other errors.

**Next step:**

Throwing the Exception QuickCarRentalException [page 40]

---

**Throwing the Exception QuickCarRentalException**

All three business methods, `saveBooking`, `cancelBooking`, and `viewActiveBookings`, can throw the application-specific exception `QuickCarRentalException`. For this reason, we must take account of this exception in the local and remote interfaces.
**Prerequisites**

- You have created the exception class `QuickCarRentalException`.
- The structure of your project `QuickCarRentalEjb` is currently displayed in the J2EE Explorer.

**Procedure**

**Throwing QuickCarRentalException in the local interface**

1. In the project structure, expand the `QuickCarRentalEjb → ejb-jar.xml` node and double-click the name of the session bean `QuickOrderProcessorBean`, if you have not already done so.
2. In the overview that appears, choose *Navigate to Local Interface*.
   
The Java Editor opens and displays the source code of the local interface `QuickOrderProcessorLocal`.
3. Add `throws QuickCarRentalException` to the end of each method (before the semi-colon).
4. To add the import statements, choose *Source → Organize Imports* from the context menu.

The source code now looks like this:

```java
package com.sap.engine.examples.ejb.quickcarrental;
import javax.ejb.EJBLocalObject;
import com.sap.engine.examples.util.QuickBookingModel;
import com.sap.engine.examples.util.QuickCarRentalException;
public interface QuickOrderProcessorLocal extends EJBLocalObject {

/**
 * Business Method.
 */
public QuickBookingModel saveBooking (String vehicleTypeId,
 String dateFromString,
 String dateToString,
 String pickupLocation,
 String dropoffLocation) throws QuickCarRentalException;

/**
 * Business Method.
 */
public String cancelBooking(String bookingId) throws QuickCarRentalException;

/**
 */
```

Creating a J2EE-Based Car Rental Application 41
5. Save the contents of the editor by choosing the appropriate icon in the toolbar.

**Throwing `QuickCarRentalException` in the remote interface**

6. Choose *Navigate to Remote Interface* and repeat steps 3-5 as appropriate for the remote interface `QuickOrderProcessorRemote`. Add `QuickCarRentalException` as a second exception (to `throws` `RemoteException`). Separate the two exceptions with a comma.

The source code of the remote interface now looks like this:

```java
package com.sap.engine.examples.ejb.quickcarrental;
import java.rmi.RemoteException;
import javax.ejb.EJBObject;
import com.sap.engine.examples.util.QuickBookingModel;
import com.sap.engine.examples.util.QuickCarRentalException;

public interface QuickOrderProcessor extends EJBObject {

    /**
     * Business Method.
     */
    public QuickBookingModel[] viewActiveBookings() throws 
    QuickCarRentalException;

    /**
     * Business Method.
     */
    public QuickBookingModel saveBooking(
        String vehicleTypeId,
        String dateFromstring,
        String dateToString,
        String pickupLocation,
        String dropoffLocation)
        throws RemoteException, QuickCarRentalException;

    /**
     * Business Method.
     */
    public String cancelBooking(String bookingId) throws RemoteException,
    QuickCarRentalException;

    /**
     * Business Method.
     */
    public QuickBookingModel[] viewActiveBookings() throws RemoteException,
    QuickCarRentalException;
```
Throwing `QuickCarRentalException` in the bean class

7. Choose Navigate to Bean Class and add `throws QuickCarRentalException` to the end of the method header of the methods `saveBooking`, `cancelBooking`, and `viewActiveBookings` (before the curly bracket `{`), in the JavaBean class `QuickOrderProcessorBean`.

Then add the necessary import statements by choosing Source -> Organize Imports from the context menu. Finally, save the contents of the editor.

The source code now looks like this:

```java
/**
   * Business Method.
   */
public QuickBookingModel saveBooking(
    String vehicleTypeId,
    String dateFromString,
    String dateToString,
    String pickupLocation,
    String dropoffLocation) throws QuickCarRentalException {
    // TODO : Implement
    return null;
}
/**
   * Business Method.
   */
public String cancelBooking(String bookingId) throws QuickCarRentalException {
    // TODO : Implement
    return null;
}
/**
   * Business Method.
   */
public QuickBookingModel[] viewActiveBookings() throws QuickCarRentalException {
    // TODO : Implement
    return null;
}
```
Result

The three business methods `saveBooking`, `cancelBooking`, and `viewActiveBookings` can now throw the application-specific exception `QuickCarRentalException`.

Next step:
Implementing the `saveBooking()` Method [page 44]

Implementing the `saveBooking()` Method

The `saveBooking` method passes the booking data that it receives as parameters to the entity bean `QuickBookingBean`. The entity bean then writes the record to the database instance of the QCR_RESERVATION table. If an error occurs, the method throws a `QuickCarRentalException`.

Prerequisites

- You have opened the bean class `QuickOrderProcessorBean` in the Java Editor.

Procedure

1. Add the following code in the `saveBooking` method:

```java
Date dateFrom=getDate(dateFromString);
Date dateTo=getDate(dateToString);
QuickBookingLocal booking = null;
java.util.Date reservationDate = new java.util.Date();

java.lang.String bookingId = generateId();
try {
    booking =
        bookingHome.create(
            bookingId,
            vehicleTypeId,
            reservationDate,
            dateFrom,
            dateTo,
            pickupLocation,
            dropoffLocation,
```

Creating a J2EE-Based Car Rental Application
2. Replace the default return value `return null;` with:

   ```java
   return getBookingModel(booking);
   ```

3. To add the import statements, choose `Source → Organize Imports` from the context menu.

4. If necessary, correct the formatting of these lines by choosing `Format` from the context menu.

5. Save the contents of the editor by choosing the appropriate icon from the toolbar.

**Result**

You have completely implemented the business method `saveBooking`.

**Next step:**

Implementing the `cancelBooking()` Method [page 45]

---

**Implementing the `cancelBooking()` Method**

The `cancelBooking` method sets the status of a booking to `CANCELLED`. If an error occurs, the method throws a `QuickCarRentalException`.

**Prerequisites**

- You have opened the bean class `QuickOrderProcessorBean` in the Java Editor.

**Procedure**

1. Add the following code in the `cancelBooking()` method:

   ```java
   try {
       QuickBookingLocal booking = bookingHome.findByPrimaryKey(bookingId);
       booking.setStatus(Constants.STATUS_CANCELLED);
   } catch (FinderException e) {
       e.printStackTrace();
   }
   ```
2. Replace the default return value `return null;` with:

```java
return bookingId + " cancelled.";
```

3. To add the import statements, choose `Source → Organize Imports` from the context menu.
4. If necessary, correct the formatting of these new lines of code.
5. Save the contents of the Editor.

**Result**

You have completely implemented the business method `cancelBooking`.

**Next step:**
Implementing the `viewActiveBookings()` Method [page 46]

### Implementing the `viewActiveBookings()` Method

The `viewActiveBookings` method returns all the bookings that have the status `ACTIVE` and fills the instances of the JavaBean `QuickBookingModel` with this booking data. The client can then use this data to display the bookings in the user interface. The instances of the `QuickBookingModel` are passed as an array to the client or Web service. If an error occurs, the method throws a `QuickCarRentalException`.

**Prerequisites**

- You have opened the bean class `QuickOrderProcessorBean` in the Java Editor.

**Procedure**

1. Add the following code in the `viewActiveBookings` method:

```java
ArrayList bookings = new ArrayList();
try {
    Collection active = bookingHome.findByStatus(Constants.STATUS_ACTIVE);
    for (Iterator iterator = active.iterator(); iterator.hasNext();)
```
bookings.add(
    getBookingModel((QuickBookingLocal) iterator.next()));
}

} catch (FinderException e) {
    e.printStackTrace();
    throw new QuickCarRentalException(e.getMessage());
}

QuickBookingModel[] result = new QuickBookingModel[bookings.size()];
bookings.toArray(result);

2. Replace the default return statement return null; with:

   return result;

3. If necessary, correct the formatting of these new lines of code.

4. To add the import statements, choose Source → Organize Imports from the context menu.

5. In the selection dialog that appears, choose java.util.Iterator as the Iterator class.

6. Save the contents of the Editor.

Result
You have completely implemented the business method viewActiveBookings.

Next step:
Creating a JAR [page 47]

Creating a JAR

In this unit you will learn about all the steps you need to take in order to pack the EJB classes and interfaces from the EJB Module project QuickCarRentalEjb, with the associated deployment descriptors, in a Jar file.

As part of this procedure, you will add several descriptions in the standard J2EE deployment descriptors ejb-jar.xml and persistent.xml.

Next step:
Editing Deployment Descriptors [page 48]
Editing Deployment Descriptors

The descriptions in the deployment descriptors specify the properties of the JAR and its associated beans.

<table>
<thead>
<tr>
<th>Deployment Descriptor</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>ejb-jar.xml</td>
<td>Describes the standard J2EE properties of the Enterprise JavaBeans.</td>
</tr>
<tr>
<td>persistent.xml</td>
<td>Describes the mapping of entity beans and their CMP fields to the corresponding database tables and table fields.</td>
</tr>
<tr>
<td>ejb-j2ee-engine.xml</td>
<td>Contains entries specific to the J2EE Engine.</td>
</tr>
</tbody>
</table>

Next step:
Adding Descriptions to ejb-jar.xml [page 48]

Adding Descriptions to ejb-jar.xml

In addition to the predefined entries, you will now enter specific deployment information about the two beans QuickBookingBean and QuickOrderProcessorBean in the J2EE standard descriptor ejb-jar.xml.

<table>
<thead>
<tr>
<th>Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>General Descriptions</strong></td>
<td></td>
</tr>
<tr>
<td>Display name</td>
<td>Returns a label for the Jar file to be created.</td>
</tr>
<tr>
<td>Description</td>
<td>Returns a description for the deployment descriptor as a whole.</td>
</tr>
<tr>
<td><strong>Deployment Information for the Entity Bean</strong></td>
<td></td>
</tr>
<tr>
<td>Abstract schema name</td>
<td>Declares an identifier for the entity bean QuickBookingBean in the Jar file. This name can then be referenced as a place-holder for the bean in statements in the EJB query language.</td>
</tr>
<tr>
<td>EJB QL</td>
<td>Contains a SELECT statement for the query method findByStatus of the entity bean.</td>
</tr>
<tr>
<td>Description</td>
<td>Contains a query description.</td>
</tr>
</tbody>
</table>

Deployment Information for the Entity Bean
Returns a local reference for the session bean `QuickOrderProcessorBean` to the entity bean `QuickBookingBean`. The entry defines the mapping from a reference name `<ReferenceName/BeanName>` that can be selected to the bean name. In this way, the reference name for the entity bean can be used in the source code of the session bean. It remains unchanged there, even if the name of the bean itself changes.

**Prerequisites**

- The structure of your EJB Module project `QuickCarRentalEjb` is currently displayed in the J2EE Explorer.

**Procedure**

**Specifying the general properties of the beans**

1. Double-click the `ejb-jar.xml` node of the `QuickCarRentalEjb` project structure.

   The Developer Studio opens a multipage editor, which allows you to enter bean-specific descriptions in the standard deployment descriptor `ejb-jar.xml`.

2. On the General tab, enter in the Display Name field the name `QuickCarRental`, along with a short text in Description – such as Quick Start for Car Rental example.

These entries are automatically added to the XML source at the appropriate point. You can check this out by choosing the Source tab and then navigating to this point in the XML source.

```xml
<ejb-jar>
  <description>Quick Start for Car Rental example.</description>
  <display-name>QuickCarRental</display-name>
  <enterprise-beans>
    ...
  </enterprise-beans>
</ejb-jar>
```
Entering entries for the entity bean

3. Choose the Enterprise Beans tab.

4. In the tree structure that appears, expand the entity beans node and choose QuickBookingBean.
   Input fields appear to the right of the tree structure.

5. Enter in the field Abstract schema name the name QuickBooking.

6. Expand the tree structure entity beans → QuickBookingBean → query.

7. In this structure, choose the node for the finder method findByStatus.
   To the right, input fields appear, where you can enter the EJB QL statements.

8. In the EJB QL Statement field, enter the following SELECT statement: select object(b) from QuickBooking b where b.status like ?1
   In the Description field, add the text Find method for getting all bookings that have the given status.
The following entries are added to the XML source:

```xml
<entity>
  <ejb-name>QuickBookingBean</ejb-name>
  ...
  <abstract-schema-name>QuickBooking</abstract-schema-name>
  ...
  <query>
    <description>Find method for getting all bookings that have the given status</description>
    <query-method>
      <method-name>findByStatus</method-name>
      <method-params>
        <method-param>java.lang.String</method-param>
      </method-params>
    </query-method>
    <ejb-ql>select object(b) from QuickBooking b where b.status like ?1</ejb-ql>
  </query>
</entity>
```

**Entering entries for the session bean**

9. In the tree structure, expand the node session beans → QuickOrderProcessorBean.
10. In this structure, select the `ejb-local-ref` node and choose Add.

A selection dialog appears with all the EJB projects created in the workspace.

![Choose EJBs](image1)

11. Choose the entity bean `QuickBookingBean` and confirm your choice with OK.

Input fields appear to the right of the tree. You can define a local reference `QuickOrderProcessorBean` to `QuickBookingBean` there.

12. In `EJB Reference Name`, enter `QuickCarRental/QuickBookingBean`.

![EJB jar.xml](image2)

The following entries are added to the XML source:

```xml
<enterprise-beans>
  <session>
    <ejb-name>ABQuickOrderProcessorBean</ejb-name>
    ...
    <ejb-local-ref>
      <description></description>
    </ejb-local-ref>
    <ejb-ref-name>QuickCarRental/QuickBookingBean</ejb-ref-name>
  </session>
</enterprise-beans>
```
13. Save your changes to the deployment descriptors by choosing the appropriate icon in the toolbar.

**Result**

You have entered all the bean-specific descriptions that your application needs in the standard J2EE deployment descriptor.

**Next step:**  
Adding Descriptions to persistent.xml [page 53]

**Adding Descriptions to persistent.xml**

In the deployment descriptor `persistent.xml`, you will enter the data source that points to the entity bean. You will also map the entity bean and the associated CMP fields to the table and its columns.

**Prerequisites**

- The structure of your project `QuickCarRentalEjb` is currently displayed in the J2EE Explorer.

**Procedure**

**Specifying the general properties**

1. Double-click the `persistent.xml` node of the `QuickCarRentalEjb` project structure.
   
   The Developer Studio opens a multipage editor, which allows you to enter persistent properties in the standard deployment descriptor `persistent.xml`.

2. In the General tab, in the Datasource name field, enter the name `CAR_RENTAL_POOL`.

3. Tick the Specify database vendor checkbox; in Database vendor, choose `SAPDB`.

4. Leave the other default settings on this tab unchanged.
Assigning a table to the entity bean

5. Choose the Entity Beans tab.

6. In the left pane, click the QuickBookingBean node.
   In the right pane, input fields appear where you can assign the table.

7. Click the button.
   You can assign the table in the wizard that appears.

8. Choose QuickCarRentalDictionary → QCR_RESERVATION and confirm your choice with OK.
Mapping CMP fields to table fields

9. Expand the QuickBookingBean node.
10. In this structure, click the node cmp-field \( \rightarrow \) bookingId.
11. Choose ID as the Column name.

The assignment is confirmed with an output line *The column is verified by the data dictionary.*

12. Repeat the above procedure as appropriate for the other CMP fields.

Map the fields and table columns as follows:

<table>
<thead>
<tr>
<th>CMP field</th>
<th>Table column</th>
</tr>
</thead>
<tbody>
<tr>
<td>vehicleTypeId</td>
<td>VEHICLETYPEID</td>
</tr>
</tbody>
</table>
13. Save your changes to the deployment descriptors by choosing the appropriate Save icon in the toolbar.

### Result

You have entered all the bean-specific descriptions that your application needs in the J2EE deployment descriptor `persistent.xml`.

**Next step:**

*Adding Descriptions to ejb-j2ee-engine.xml [page 56]*

---

**Adding Descriptions to ejb-j2ee-engine.xml**

The deployment descriptor `ejb-j2ee-engine.xml` generally defines other properties specific to the SAP J2EE Engine. For example, you can specify the JNDI name of the beans in this file. However, these special entries are not required for this example application.

**Next step:**

*Creating a Jar File [page 56]*

---

**Creating a Jar File**

You have maintained the descriptions needed for your car rental application in the deployment descriptors and can now create the corresponding archive file.

**Prerequisites**

- The structure of your project `QuickCarRentalEjb` is currently displayed in the J2EE
Procedure

1. Choose the project node `QuickCarRentalEjb` in the J2EE Explorer.
2. Open the context menu and choose `Build EJB Archive`.
3. Confirm the message that files were generated successfully with `OK`.

Result

The archive file `QuickCarRentalEjb.jar` has been created. It contains all the classes in the referenced EJB Module project and the three deployment descriptors.

Next step:

Implementing a Web Application [page 57]

Implementing a Web Application

In this unit, you will learn about all the steps you need to take to develop the Web resources for the car rental Web client in the Developer Studio.

To start with, you will create a special project known as a Web Module project. Using this generated standard project as a starting point, you will create and implement a JSP and a servlet. You will then define the user interface for the Web client using this JSP. You will implement the servlet such that it can handle requests from the JSP, call the session bean business methods, and format the data for renewed display in the JSP.

Next step:

Creating a Web Module Project [page 57]

Creating a Web Module Project

To add and edit Web resources for your car rental application, you will first create a Web Module project called `QuickCarRentalWeb`. Since the bean classes and auxiliary classes
from the EJB Module project are to be referenced in the Web resources, you will need both, to enter the EJB Module project **QuickCarRentalEjb** and the Java project **Helperclasses** in the project dependencies.

**Prerequisites**

- You have created the entity bean **QuickBookingBean** and the stateless session bean **QuickOrderProcessorBean** in the project **QuickCarRentalEjb**.
- You have already created the Java project **Helperclasses**.

**Procedure**

**Creating the project structure**

1. Choose **File → New → Project** to start the **New Project** wizard.
2. In the wizard, select the **J2EE** category (in the left pane), followed by **Web Module Project** (in the right pane). Choose **Next**.

3. Give your Web Module project the name **QuickCarRentalWeb** and leave the default settings for **Project contents** unchanged.
4. Choose **Finish**.

The wizard generates a initial project structure for the new Web Module project **QuickCarRentalWeb**. It already contains the two deployment descriptors **web.xml** and **web-j2ee-engine.xml**.

**Specifying the project dependencies**

For the build path, both the Java project **Helperclasses** and the EJB Module project **QuickCarRentalEjb** are to be referenced in the new project.

1. In the J2EE Explorer, select the project node **QuickCarRentalWeb** and open the context menu.
2. Choose **Properties**.
3. Choose **Java Build Path** in the left frame. In the right frame, choose the **Projects** tab.
4. Select the projects Helperclasses and QuickCarRentalEjb.

5. Confirm with OK.

Next step:
Creating the JSP quickCarRentalView [page 59]

Creating the JSP quickCarRentalView

To develop the user interface for the car rental application, you will create and implement a JSP named QuickCarRentalView. When doing so, you should define only two areas in this user interface, each of which will contain a corresponding HTML table.

The first HTML table implements an input form for the new booking data for the rental of the car. There should also be a button on this input form and a linked action for sending a new data record to the database.

The second HTML table displays the current bookings. Each booking can be cancelled by means of an appropriate action.

Prerequisites

☐ The structure of your project QuickCarRentalWeb is currently displayed in the J2EE Explorer.

Procedure

1. In the J2EE Explorer, to open the relevant wizard, choose New → JSP… from the context menu of the project node QuickCarRentalWeb.

2. In JSP Name, enter quickCarRentalView.
3. Choose Finish.

**Result**

The wizard creates the file `quickCarRentalView.jsp` in the project folder `webContent`. The associated JSP Editor opens automatically and displays a default message in the Preview.

Next step:  
*Adding Source Code [page 60]*

---

**Adding Source Code**

**Prerequisites**

- You have created the JSP `quickCarRentalView`.

**Procedure**

1. Choose the *Source* tab to insert the source code.
2. Replace the pregenerated source code completely with the following:

```jsp
<%@ page import="com.sap.engine.examples.util.Constants" language="java" %>
<html>
<head><title>QuickCarRental</title></head>
<body style="font-family:Arial;">
    <p style="font-weight:bold; font-size:14pt">QuickCarRental</p>
    <br>
    <table border="0" bgcolor="D2D8E1" cellspacing="2" cellpadding="2" width="720">
        <tr><td><font color="red"><%=(session.getAttribute(Constants.CLIENT_MESSAGE)!=null)%>=(session.getAttribute(Constants.CLIENT_MESSAGE))></font></td></tr>
    </table>
</body>
</html>
```

---

Creating a J2EE-Based Car Rental Application 60
<tr align="center" style="font-size:9pt; font-weight:bold">
<td width="120">Pick-up Location</td>
<td width="120">Vehicle Type</td>
<td width="120">Drop-off Location</td>
<td width="120">Pickup Date</td>
<td width="120">Return Date</td>
<td width="120"></td>
</tr>
<tr align="center" bgcolor="ffffff">
<td align="center">
<select name="pickupLocation">
   <% for (int j = 0;j < Constants.LOCATION.length;j++) {
      String value = Constants.LOCATION[j];
   %>
   <option value="<%=value%>"<%=value%></option>
   <% } %>
</select>
</td>
<td align="center">
<select name="vehicleTypeId">
   <% for (int j = 0;j < Constants.VEHICLE_TYPE.length;j++) {
      String value = Constants.VEHICLE_TYPE[j];
   %>
   <option value="<%=value%>"<%=value%></option>
   <% } %>
</select>
</td>
<td align="center">
<select name="dropoffLocation">
   <% for (int j = 0;j < Constants.LOCATION.length;j++) {
      String value = Constants.LOCATION[j];
   %>
   <option value="<%=value%>"<%=value%></option>
   <% } %>
</select>
</td>
<td align="center">
<input name="pickupDate" type="text" size="13" maxlength="10"/>
</td>
<td>
<Cre
</td>
</tr>
<tr align="center" bgcolor="ffffff">
<td align="center">
<select name="vehicleTypeId">
   <% for (int j = 0;j < Constants.VEHICLE_TYPE.length;j++) {
      String value = Constants.VEHICLE_TYPE[j];
   %>
   <option value="<%=value%>"<%=value%></option>
   <% } %>
</select>
</td>
<td align="center">
<select name="dropoffLocation">
   <% for (int j = 0;j < Constants.LOCATION.length;j++) {
      String value = Constants.LOCATION[j];
   %>
   <option value="<%=value%>"<%=value%></option>
   <% } %>
</select>
</td>
<td align="center">
<input name="pickupDate" type="text" size="13" maxlength="10"/>
</td>
<td>
<Cre
</td>
</tr>
<table border="0" bgcolor="#D2D8E1" cellspacing="2" cellpadding="2" width="720">
<form name="reservations" method="POST" action="/QuickCarRental">
<tr align="center" style="font-size:9pt; font-weight:bold">
<th width="20">&nbsp;</th>
<th width="80">ID</th>
<th width="120">Vehicle Type</th>
<th width="120">Pick-up Location</th>
<th width="120">Drop-off Location</th>
<th width="80">Begin Date</th>
<th width="80">End Date</th>
<th width="100">Price</th>
</tr>
<% for (int i = 0; i < reservations.size(); i++) { %>
<tr align="center" bgcolor="#ffffff" style="font-size:9pt;">
<td>
<input type="checkbox" name="check" value="<%=reservation[0]%>"/>
</td>
<% for (int j = 0; j < reservation.length; j++) { %>
<td><%=reservation[j]%></td>
<% } %>
</tr>
<% } %>
<tr align="center" bgcolor="#DEE3E9">
</form>
</table>

Creating a J2EE-Based Car Rental Application
3. If necessary, correct the formatting of these lines by choosing *Source → Format* from the context menu.

4. Save the contents of the editor by choosing the appropriate icon from the toolbar.

**Result**

The Developer Studio refreshes the JSP source code. If you now choose the *Preview* tab in the JSP Editor, the following preview layout is displayed:

**QuickCarRental**

<table>
<thead>
<tr>
<th>Pick-up Location</th>
<th>Vehicle Type</th>
<th>Drop-off Location</th>
<th>Pickup Date</th>
<th>Return Date</th>
<th>Add Reservation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>Vehicle Type</th>
<th>Pick-up Location</th>
<th>Drop-off Location</th>
<th>Begin Date</th>
<th>End Date</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**No Reservations**

**Next step:**

*Creating the servlet QuickReservationServlet [page 64]*
Creating the QuickReservationServlet Servlet

You will now add another Web resource, a servlet named QuickReservationServlet, to the Web project.

The servlet is need to handle requests sent from the JSP; to process them; to call the appropriate business methods from the session bean QuickOrderProcessorBeans; and finally to format the data records for display in the JSP.

Prerequisites

The structure of the Web project QuickCarRentalWeb is currently displayed in the J2EE Explorer.

Procedure

1. In the J2EE Explorer, to open the relevant wizard, choose New → Servlet… from the context menu of the project node QuickCarRentalWeb.
2. In Servlet Name, enter QuickReservationServlet.
3. Choose the servlet type HTTP Servlet.
4. Assign the package com.sap.examples.quickcarrental.servlet as the Servlet Package.
5. Activate the checkboxes for the two servlet methods doGet() and doPost().


Result

The wizard creates the servlet file QuickReservationServlet along with the two methods doGet() and doPost().
Next step:
Adding Source Code to the Servlet [page 65]

Adding Source Code to the Servlet

Prerequisites
☐ You have created the servlet QuickReservationServlet.

Procedure
1. Open the Java editor for the new created servlet QuickReservationServlet.
2. Add the following source code after the doPost() method:

```java
public void doWork(
    HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException {
QuickOrderProcessorLocal order = initializeController();
handleRequest(request, response, order);
viewAllBookings(request, order);
HttpSession session = request.getSession(true);
RequestDispatcher dispatcher =
    request.getRequestDispatcher( "/view");
try {
dispatcher.forward(request, response);
} catch (IOException e) {
    e.printStackTrace();
    throw new ServletException(e.getMessage());
}
}

private QuickOrderProcessorLocal initializeController() {
    try {
        Context ctx = new InitialContext();
        QuickOrderProcessorLocalHome orderHome =
            (QuickOrderProcessorLocalHome) ctx.lookup(
                "java:comp/env/ejb/QuickOrderProcessorBean");
        return orderHome.create();
    }
```
```java
public void handleRequest(
    HttpServletRequest request, 
    HttpServletResponse response, 
    QuickOrderProcessorLocal order) 
    throws ServletException {
    HttpSession session = request.getSession(true);
    session.setAttribute(Constants.CLIENT_MESSAGE, null);
    String action = request.getParameter("appAction");
    if (action != null && action.length() > 0) {
        switch (Integer.parseInt(action)) {
            case Constants.ACTION_SAVE :
                saveAction(request, order);
                break;
            case Constants.ACTION_CANCEL :
                cancelAction(request, order);
                break;
        }
    }
}

private void viewAllBookings{
    HttpServletRequest request, 
    QuickOrderProcessorLocal order) {
    HttpSession session = request.getSession(true);
    QuickBookingModel[] bookings;
    try {
        bookings = order.viewActiveBookings();
        session.setAttribute(
            Constants.RESERVATIONS,
```
formatBookings(bookings));
}
}

private void saveAction(
    HttpServletRequest request,
    QuickOrderProcessorLocal order) {
    HttpSession session = request.getSession(true);
    try {
        java.lang.String dateFrom = request.getParameter("pickupDate");
        java.lang.String dateTo = request.getParameter("dropoffDate");
        String vehicleTypeId = request.getParameter("vehicleTypeId");

        String pickupLocation = request.getParameter("pickupLocation");
        String dropoffLocation = request.getParameter("dropoffLocation");
        order.saveBooking(vehicleTypeId, dateFrom, dateTo, pickupLocation, dropoffLocation);
    } catch (QuickCarRentalException e) {
        session.setAttribute(Constants.CLIENT_MESSAGE, e.getMessage());
    }
}

private void cancelAction(
    HttpServletRequest request,
    QuickOrderProcessorLocal order) {
    HttpSession session = request.getSession(true);
    String[] selectedBookings = request.getParameterValues("check");
    for (int i = 0; i < selectedBookings.length; i++) {
        try {
            order.cancelBooking((String) selectedBookings[i]);
        } catch (QuickCarRentalException e) {
            session.setAttribute(Constants.CLIENT_MESSAGE, e.getMessage());
        }
    }
}

private ArrayList formatBookings(QuickBookingModel[] bookings){
ArrayList bookings_list = new ArrayList();
DateFormat dateFormat = DateFormat.getDateInstance(DateFormat.MEDIUM);
for (int i=0;i<bookings.length;i++){
    String[] booking = new String[7];
    booking[0] = bookings[i].getBookingId();
    booking[1] = bookings[i].getVehicleType();
    booking[2] = bookings[i].getPickupLocation();
    booking[3] = bookings[i].getDropoffLocation();
    booking[4] = bookings[i].getDateFrom();
    booking[5] = bookings[i].getDateTo();
    booking[6] = bookings[i].getPrice();
    bookings_list.add(booking);
}

return bookings_list;
}

3. Add the statement calling doWork() to the two servlet methods doGet() and doPost(). To do so, replace the comment line //TODO : Implement in the two methods with:

```java
doWork(request, response);
```

4. If necessary, correct the formatting of these lines by choosing Source → Format from the context menu.

5. To add the import statements, position the cursor anywhere in the Java Editor and choose Source → Organize Imports from the context menu.

6. Choose the context javax.naming.Context and if necessary com.sap.engine.examples.util.Constants, and confirm by choosing OK.

   The appropriate import statements are added to the source code.

7. Save the contents of the editor by choosing the appropriate icon from the toolbar.

**Result**

The Developer Studio refreshes the servlet source code. Your Tasks view should no longer show any other errors.

**Next step:**

Creating a Web Archive [page 69]
Creating a Web Archive

In this unit you will learn about all the steps you need to take in order to pack all the Web resources from the Web Module project QuickCarRentalWeb, with the associated deployment descriptors, in a Web archive.

You will enter some descriptions in the J2EE standard deployment descriptor web.xml, before you generate the corresponding Web archive (WAR file).

Next step:
Editing Deployment Descriptors [page 69]

Editing Deployment Descriptors

The descriptions in the deployment descriptors specify the properties of the Web archive (WAR) and its associated Web resources.

<table>
<thead>
<tr>
<th>Deployment Descriptor</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>web.xml</td>
<td>Describes the standard J2EE properties of the Web resources (JSPs, servlets, and so on). These properties include mapping information, security entries (access restrictions and security roles), and entries concerning EJB reference names.</td>
</tr>
<tr>
<td>web-j2ee-engine.xml</td>
<td>Contains entries concerning Web resources specific to the SAP J2EE Engine.</td>
</tr>
</tbody>
</table>

Next step:
Adding Descriptions to web.xml [page 69]

Adding Descriptions to web.xml

In addition to the predefined entries, you will now enter specific deployment information about the two Web resources in the J2EE standard descriptor web.xml.

<table>
<thead>
<tr>
<th>Description</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Descriptions</td>
<td></td>
</tr>
<tr>
<td>Display Name</td>
<td>Returns a label for the War file to be created.</td>
</tr>
<tr>
<td>Description</td>
<td>Returns a description for the deployment descriptor as a whole.</td>
</tr>
<tr>
<td>Mapping entries</td>
<td></td>
</tr>
</tbody>
</table>
URL pattern
Defines symbolic names for the JSP file and for the servlet.
This approach has an advantage over directly calling the file names, in
that the name of the components can be changed without your having to
adapt the source code accordingly.

Reference entries

EJB local references
Contains a local reference name for the session bean
QuickOrderProcessorBean. This reference name must match the
call in the servlet.

Prerequisites
☐ The structure of your Web Module project **QuickCarRentalWeb** is currently displayed in the **J2EE Explorer**.

Procedure

Specifying the general properties

1. Double-click the **web.xml** node of the **QuickCarRentalWeb** project structure.
   The Developer Studio opens a multipage editor, which allows you to enter descriptions
   in the standard deployment descriptor **web.xml**.

2. On the **General** tab, enter in the **Display Name** field the name **Rent-A-Car**, along with
   a short text in **Description** – such as **Web resources for Car Rental example**.

   These entries are automatically added to the XML source at the appropriate point.

```
<web-app>
  <display-name>Rent-A-Car</display-name>
  <description>Web resources for car rental example.</description>
  ...
</web-app>
```
Defining a URL pattern

3. Choose the Mapping tab.

4. Select Servlet Mappings and choose Add.

5. In the list that appears, select the two entries QuickReservationServlet and quickCarRentalView.jsp and confirm by choosing OK.

6. Assign the URL Pattern to each Web resource as follows:

<table>
<thead>
<tr>
<th>Servlet Name</th>
<th>URL Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>QuickReservationServlet</td>
<td>/</td>
</tr>
<tr>
<td>quickCarRentalView.jsp</td>
<td>/view</td>
</tr>
</tbody>
</table>

The following entries are automatically added to the XML source:

```xml
<web-app>
  ...
  <servlet-mapping>
    <servlet-name>QuickReservationServlet</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>quickCarRentalView.jsp</servlet-name>
    <url-pattern>/view</url-pattern>
  </servlet-mapping>
</web-app>
```

Note:
In the JSP file quickCarRentalView.jsp, the servlet is called using / . Conversely in the servlet source code, the JSP file is called using /view (see the example below):

```java
... public void doWork(
  ...
  RequestDispatcher dispatcher =
      request.getRequestDispatcher("/view");
  ...
```

Entering a local reference to an EJB

7. Choose the EJBs tab.

8. In the EJB Local References frame, choose Add.

9. From the selection list that appears, choose QuickOrderProcessorBean and confirm using OK.
The following lines are automatically added to the XML source:

```xml
<web-app>
  ...
  <ejb-local-ref>
    <description/>
    <ejb-ref-name>ejb/QuickOrderProcessorBean</ejb-ref-name>
  ...
  <ejb-link>QuickCarRentalEjb.jar#QuickOrderProcessorBean</ejb-link>
  </ejb-local-ref>
</web-app>
```

**Note:**
In this case, the EJB name and the reference name match. If, however, the name of the session bean changes later, the corresponding bean call in the servlet source code can remain unchanged, provided you have used the reference name there.

You can find the reference name for the session bean in the corresponding line in the source code of `QuickReservationServlet`:

```java
private QuickOrderProcessorLocal initializeController() throws ServletException {
  try {
    Context ctx = new InitialContext();
    QuickOrderProcessorLocalHome orderHome =
      (QuickOrderProcessorLocalHome) ctx.lookup(
        "java:comp/env/ejb/QuickOrderProcessorBean");
  ...
```

10. Save your changes to the deployment descriptor entries by choosing the appropriate icon in the toolbar.

**Result**
You have entered all the necessary descriptions for this example application in the standard J2EE deployment descriptor `web.xml`.

In this case also, no J2EE Engine-specific entries are needed in the deployment descriptor `web-j2ee-engine.xml`. Thus, you have completely specified your Web Application project and can now continue by creating a corresponding archive file for the Web resources.
Next step:
Creating a War File [page 73]

Creating a War File

Prerequisites

☐ The structure of your Web Module project QuickCarRentalWeb is currently displayed in the J2EE Explorer.

Procedure

11. Choose the project node QuickCarRentalWeb in the J2EE Explorer.
12. Open the context menu and choose Build Web Archive.
13. Confirm the message that files were generated successfully with OK.

Result

The archive file QuickCarRentalWeb.war has been created. It contains all the classes in the referenced Web project and the two deployment descriptors from the Web Application project. The archive file is added to the project structure as a separate node.

Next step:
Creating and Deploying the Complete J2EE Application [page 73]

Creating and Deploying the Complete J2EE Application

In this unit you will learn carry out the necessary steps in order to pack all the resources and archives you have created, with the associated deployment descriptors, in an EAR file, to create a complete car rental application. This archive file will contain all the component parts...
of the application and will reference the two archives you created previously, 
QuickCarRentalEjb.jar and QuickCarRentalWeb.war.

To create your complete car rental application, you will first create an Enterprise Application 
project called QuickCarRentalEar. You will also enter some descriptions in the J2EE 
standard deployment descriptor application.xml, before you generate the corresponding 
EAR file as an archive for the complete application and finally deploy this to the J2EE Engine.

Next step:
Creating an Enterprise Application Project [page 74]

Creating an Enterprise Application Project

The Enterprise application project contains a reference to the two archive projects 
QuickCarRentalJar and QuickCarRentalWar as well as the two deployment 
descriptors application.xml and application-j2ee-engine.xml.

Prerequisites
☐ You have created the projects QuickCarRentalEjb and QuickCarRentalWeb.

Procedure
1. Choose File → New → Project to start the New Project wizard.
2. In the wizard, select the J2EE category (in the left pane), followed by Enterprise 
   Application Project (in the right pane). Choose Next.

3. Give your Enterprise application project the name QuickCarRentalEar and leave 
   the default settings for Project contents unchanged.
4. Choose Next.
   In the next wizard dialog, you can choose the referenced archive projects.
5. Choose the EJB Module Project QuickCarRentalEjb and the Web Module project 
   QuickCarRentalWeb.
Result

The wizard generates a project structure for the new Enterprise Application project QuickCarRentalEar and creates a project folder of the same name in the default workspace.

The project already contains the two deployment descriptors application.xml and application-j2ee-engine.xml.

Next step:
Editing Deployment Descriptors [page 75]

Editing Deployment Descriptors

The descriptions in the deployment descriptors specify the properties of the complete application (or EAR file).

<table>
<thead>
<tr>
<th>Deployment Descriptor</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>application.xml</td>
<td>Describes the standard J2EE properties of the complete application and the referenced modules. In particular, you must specify the URL where the J2EE application is stored on the J2EE Engine.</td>
</tr>
<tr>
<td>application-j2ee-engine.xml</td>
<td>Contains additional entries specific to the SAP J2EE Engine. You do not need to make any entries for the car rental application in this deployment descriptor.</td>
</tr>
</tbody>
</table>

Next step:
Adding Descriptions to application.xml [page 75]
In addition to the predefined entries, you will now record specific deployment information in the J2EE standard descriptor application.xml.

<table>
<thead>
<tr>
<th>Description</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>General Descriptions</strong></td>
<td></td>
</tr>
<tr>
<td>Display Name</td>
<td>Returns a label for the Ear file to be created.</td>
</tr>
<tr>
<td>Description</td>
<td>Returns a description for the deployment descriptor as a whole.</td>
</tr>
<tr>
<td><strong>Special entries for modules</strong></td>
<td></td>
</tr>
<tr>
<td>Context root</td>
<td>The context root defines the root directory for the URL from which the application is called. The appropriate URL (for localhost) is: <a href="http://localhost:50000/">http://localhost:50000/</a>&lt;Context-Root&gt;</td>
</tr>
</tbody>
</table>

**Prerequisites**

- The structure of your QuickCarRentalEar project is currently displayed in the J2EE Explorer.

**Procedure**

**Specifying the general properties**

1. Double-click the application.xml node of the QuickCarRentalEar project structure. The Developer Studio opens a multipage editor, which allows you to record descriptions in the standard deployment descriptor application.xml.

2. On the General tab, enter the name QuickCarRentalApplication in the Display Name field, along with a short text in Description – such as Modules for car rental example.

**Specifying the context root**

3. Choose the Modules tab.

4. Choose the project node QuickCarRentalWeb.war in the left pane.

5. Change the name of the predefined value in the Context Root field to QuickCarRental.
The following entries are automatically added to the XML source:

```xml
<application>
  <display-name>QuickCarRentalApplication</display-name>
  <description>Modules for car rental example.</description>
  <module>
    <ejb>QuickCarRentalEjb.jar</ejb>
  </module>
  <module>
    <web>
      <web-uri>QuickCarRentalWeb.war</web-uri>
      <context-root>QuickCarRental</context-root>
    </web>
  </module>
</application>
```

**Note:**

Thus the URL for your car rental application (for localhost) is:
http://localhost:50000/QuickCarRental

6. Save your changes to the deployment descriptor entries by choosing the appropriate icon in the toolbar.

**Result**

You have entered all the descriptions that your application needs in the standard J2EE deployment descriptor `application.xml`.

Since no J2EE Engine-specific entries are needed in the deployment descriptor `application-j2ee-engine.xml` for this application, the Enterprise Application project is now completely
specified. You can now continue by creating a appropriate archive file for the complete car rental application.

Next step:
Creating DataSource Alias [page 78]

Creating a DataSource Alias

In order to enable communication with the database and to access the table from within the application, you must create a DataSource alias. This allows you to refer to the default DataSource from the J2EE Engine. The default DataSource is automatically created when you install the SAP J2EE Engine and it is not associated with a particular application which means that multiple applications can use a single DataSource.

The application only needs to know the name of the DataSource alias; the actual path to the database is stored in the DataSource definition in the server. The application can thus find all tables for the DataSource, including QCR_RESERVATION.

Prerequisites

☐ Your application makes use of the default DataSource from the J2EE Engine.
☐ You have entered CAR_RENTAL_POOL in the DataSource name field in the deployment descriptor persistent.xml of the QuickCarRentalEjb project (this step is described in Adding Descriptions to persistent.xml [page 53]).
☐ The structure of your project QuickCarRentalEar is currently displayed in the J2EE Explorer.

Procedure

1. Choose the project node QuickCarRentalEar.
2. Open the context menu and choose New → META-INF/data-source-aliases.xml.
3. In the wizard screen that appears, enter CAR_RENTAL_POOL in the Alias Name field.

The wizard adds a new XML file data-source-aliases.xml to the project structure.

Result

You have added a DataSource alias with the name CAR_RENTAL_POOL to the project. The new alias is automatically assigned to the default DataSource. You now are able to check it by double-clicking the node META-INF/data-source-aliases.xml in the J2EE Explorer. The XML editor then display the following code:
<?xml version="1.0"?>
<data-source-aliases>
  <aliases>
    <data-source-name>${com.sap.datasource.default}</data-source-name>
    <alias>CAR_RENTAL_POOL</alias>
  </aliases>
</data-source-aliases>

Note:
In the generated XML, the name ${com.sap.datasource.default} represents the default DataSource stored in the server.

Next step:
Creating an Ear File [page 79]

Creating an Ear File

Prerequisites

☐ The structure of your Enterprise Application project QuickCarRentalEar is currently displayed in the J2EE Explorer.

Procedure

5. If necessary, choose the menu path Project → Rebuild All to rebuild all projects.
6. Select project node QuickCarRentalEar again and open the context menu.
7. Choose Build Application Archive.
8. Choose OK to confirm the message saying that files were generated successfully.

Result

The archive file QuickCarRentalEar.ear has been created. It contains all the Web resources, the EJBs, and the auxiliary classes, along with the DataSource alias and the two deployment descriptors from the Enterprise Application project. The archive file is added to the project structure.
Deploying an EAR

Before you start your car rental application for the first time, you must deploy the EAR file you have created to the J2EE Engine.

Prerequisites

- You have created an EAR file.
- You have made sure that the SAP J2EE Engine has been launched.
  
  To do this, refer to Starting and Stopping the SAP J2EE Engine [extern]

Procedure

1. Expand the project node QuickCarRentalEar in the J2EE Explorer and choose the EAR file QuickCarRentalEar.ear.
2. Open the context menu for this node and choose Deploy to J2EE engine.

Result

The EAR file is deployed to the J2EE Engine. By completing this step, you have made all the preparations necessary to start and test your car rental application in a Web browser.

Executing the Car Rental Application

Prerequisites

- You have deployed the SDA file with the table definition QCR_RESERVATION.
- You have created the application archive (QuickCarRentalEar.ear).
- You have started the SAP J2EE Engine.
  
  To do this, refer to Starting and Stopping the SAP System [extern].

Procedure
1. Call the car rental application by entering its URL in the Web browser.

<table>
<thead>
<tr>
<th>URL for the application:</th>
<th>http://&lt;host&gt;:&lt;port&gt;/QuickCarRental</th>
</tr>
</thead>
<tbody>
<tr>
<td>URL for the application with JSP mapping:</td>
<td>http://&lt;host&gt;:&lt;port&gt;/QuickCarRental/view</td>
</tr>
</tbody>
</table>

The Web browser displays the initial input screen for the car rental client.

![QuickCarRental](image)

The appropriate records are displayed as a table.

![QuickCarRental](image)

Next step: Making the Car Rental Application Available as a Web Service [page 82]
Making the Car Rental Application Available as a Web Service

To also enable external applications to use the car rental application, it should now be made available as a Web service. In this unit you will become familiar with a procedure for creating a new Web service, named **QuickCarRentalService**, using the specific implementation of the car rental application as a starting point. In this procedure, you will create both the virtual interface and the associated Web service definition, along with a configuration, in one step. Afterwards, you need only deploy the new Web service.

**Next step:**
[Creating a Web Service [page 82]]

Creating a Web Service

**Prerequisites**
- You have created and completely implemented the session bean **QuickOrderProcessorBean** in the EJB Module project.
- You have created the Enterprise Application project **QuickCarRentalEar**.

**Procedure**

1. In the J2EE Explorer, expand the project structure **QuickCarRentalEjb** → **ejb-jar.xml**.
2. Select the session bean **QuickOrderProcessorBean** and choose **New → Web Service** from the context menu.
   You can create a new Web service in the wizard that appears.
3. In **Web Service Name**, enter **QuickCarRentalService**.
4. Choose **Simple SOAP** as the **Default Configuration Type**.

   ![Web Service Creation Wizard]

   - **Endpoint:** **QuickOrderProcessorBean**
   - **Web Service Name:** **QuickCarRentalService**
   - **Default Configuration Type:** **Simple SOAP**
   - **Configuration Name:** **Config1**
   - **Access URL:** `/QuickCarRentalService/Config1`

5. Leave all the other predefined values unchanged and choose **Next**.
6. Accept the selection of all three business methods and choose *Next*.

   ![Web Services](image)

   **Method Selection**
   Select methods to be contained in the virtual interface (web service)

   Methods to be contained:
   - cancelBooking(java.lang.String) : java.lang.String
   - viewActiveBookings : com.sap.engine.examples.util.QuickBookingModel[]

7. Accept the default values for the virtual interface and for the Web service definition.

8. Assign the project *QuickCarRentalEar* as the *EAR Project*.

   ![Project Configuration](image)

   **Source Folder:** QuickCarRentalEJB
   **Package:** com.sap.engine.examples.ejb.q
   **Endpoint:** com.sap.engine.examples.ejb.quickcarren
   **Virtual Interface:** QuickCarRentalServiceVi
   **Web Service Definition:** QuickCarRentalServiceWsd
   **EAR Project:** QuickCarRentalEar

9. Choose *Finish*.

**Result**

The wizard generates: A virtual interface named *QuickCarRentalServiceVi*; the associated Web service definition *QuickCarRentalServiceWsd*; a configuration named *Config1* and

It also inserts the corresponding nodes in the structure of the EJB Module project and automatically starts the *WS Deployment Descriptor Editor*. 
Next step:
Deploying the Web Service QuickCarRentalService [page 84]

Deploying the Web Service
QuickCarRentalService

You still need to deploy the new Web service QuickCarRentalService to the J2EE Engine as follows: Proceed as follows:

Process Flow

1. First update the Jar file in the project QuickCarRentalEjb, as described in Creating a Jar File [page 56].
2. Also update the EAR file in the project QuickCarRentalEar, as described in Creating an EAR File [page 79].
3. Redeploy the EAR file, as described in Deploying an EAR File [page 80].

Result

You have added the Web service QuickCarRentalService to the complete car rental application and deployed it to the J2EE Engine. This Web service is now available to external users.