Get Started Developing, Debugging, and Deploying Custom J2EE Applications Quickly and Easily with SAP NetWeaver Developer Studio

Karl Kessler

Karl Kessler joined SAP in 1992, and in 1994 became a member of the product management group for the ABAP Workbench. In 1996, Karl became product manager for business programming languages and frameworks. In 2003, he took over product management of the SAP NetWeaver platform, including SAP Web Application Server, SAP NetWeaver Developer Studio, SAP Enterprise Portal, Web services, J2EE, and ABAP.

(complete bio appears on page 40)

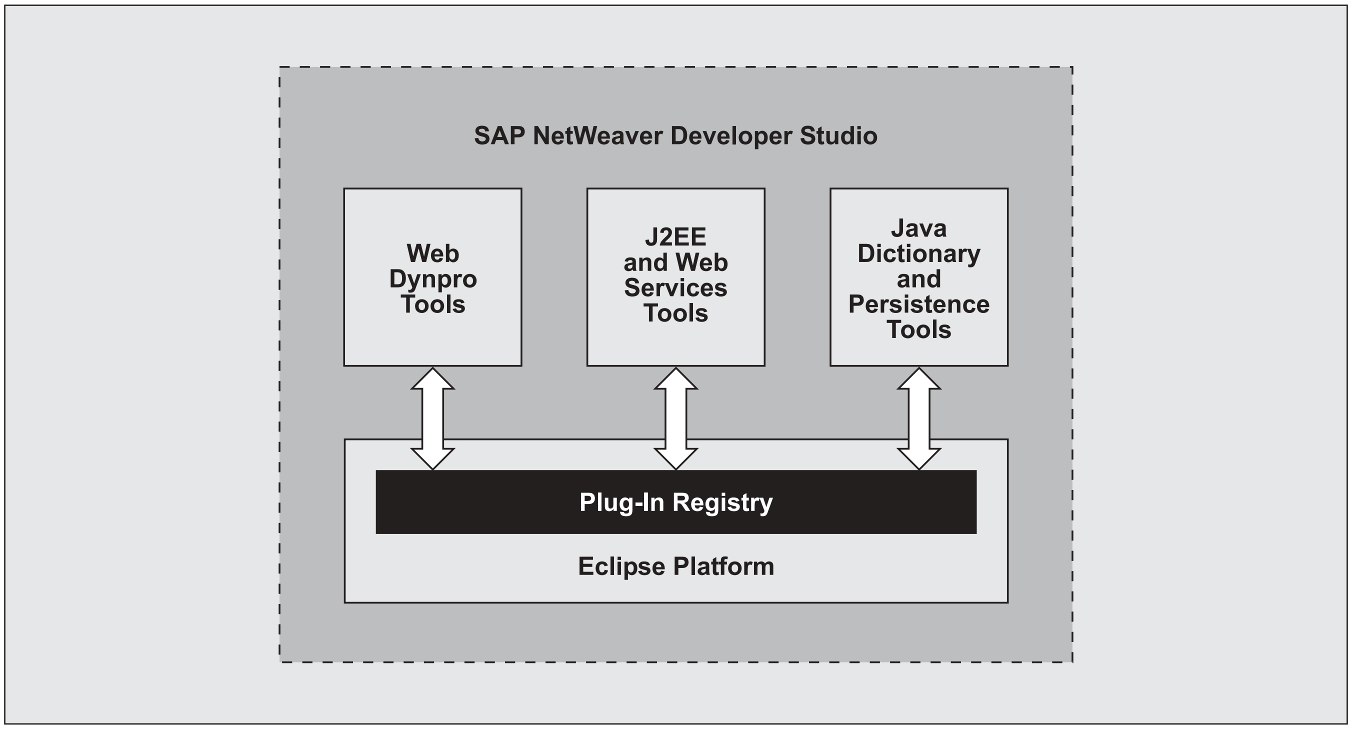
SAP Web Application Server (SAP Web AS), at version 6.40 as of this writing, is SAP's strategic, open application platform for running and managing applications written in either ABAP or Java. SAP Web AS underlies the entire line of SAP NetWeaver components (e.g., SAP Business Information Warehouse 3.5+ and SAP Enterprise Portal 6.0 SP3). SAP Web AS began its open support for Java applications with version 6.20, enabling a single, integrated platform for running both Java and ABAP applications. The J2EE Engine inside SAP Web AS 6.40 is also J2EE 1.3-compliant, enabling your existing 1.3-compliant Java applications to run alongside your newly developed applications, saving you countless hours of redevelopment work.

To help customers more easily develop Java applications, SAP Web AS 6.40 includes a PC-based integrated development environment (IDE) called SAP NetWeaver Developer Studio.[1] Like most IDEs, SAP NetWeaver Developer Studio contains a comprehensive set of tools for developing, debugging, and deploying Java applications for SAP Web AS. Rather than create this tool from scratch, SAP built SAP NetWeaver Developer Studio as a set of plug-ins to a generic, extensible, open source IDE called Eclipse[2] (see **Figure 1** on the next page). To SAP this means faster releases, fewer stability issues, and the flexibility to easily add components like Web Dynpro, Enterprise Portal development tools, Web services, and so on. To you this means a single, stable tool you only need to learn once to leverage any number of technologies.

[1] SAP NetWeaver Developer Studio is for developing Java applications only. ABAP applications are still developed with the ABAP Workbench.

[2] To learn more about Eclipse, go to **www.eclipse.org**.

Figure 1 SAP NetWeaver Developer Studio Is Built As a Set of "Plug-Ins" to the Eclipse Platform


SAP NetWeaver Developer Studio
Web Dynpro Tools
J2EE and Web Services Tools
Java Dictionary and Persistence Tools
Plug-In Registry
Eclipse Platform

This article shows developers how to create custom J2EE applications for SAP Web AS 6.40 and higher using SAP NetWeaver Developer Studio, which can be downloaded from **http://sdn.sap.com** as part of SAP NetWeaver Developer Workplace.

In the first part of this article, you'll learn how to use SAP NetWeaver Developer Studio to define and maintain SQL database tables, and to develop, debug, and deploy applications based on JavaServer Pages (JSPs), JavaBeans, and Java Database Connectivity (JDBC). The second part of the article then introduces you to Enterprise JavaBeans (EJBs) and shows you how to incorporate them into your applications to create a more robust, reusable design.

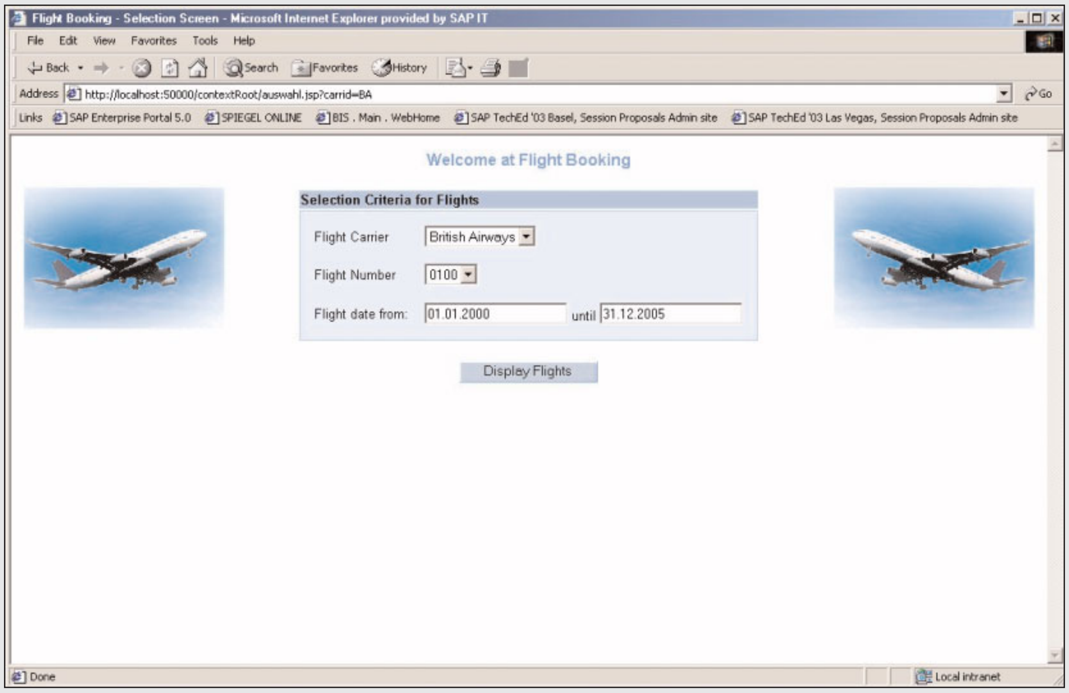
Developing a Simple Web Application

Suppose we want to develop a simple Web application that enables users to search for flights in a small flight database[3] and book a reservation for a particular flight. The following three screenshots demonstrate the desired functionality:

- **Figure 2** shows a selection screen for finding flights. When a user chooses an airline from the *Flight Carrier* dropdown list, valid flight numbers (connection IDs) for this airline are loaded into the *Flight Number* dropdown list. The user then specifies a date range and clicks on the *Display Flights* button to continue.
- **Figure 3** shows the result if matching flights are found, which includes additional flight details like price and capacity. The user can then book a flight by clicking on its corresponding *Book* button.

[3] We'll create this database in the next section — it can be hosted locally or on your SAP Web AS server. The data retrieval code we'll use will work for SQL databases accessible via JDBC. If your application needs access to SAP R/3 data, you'll need to use the SAP Java Connector (JCo) instead (see the note on page 15 for more on this).

Figure 2 *Flight Selection Screen*


Flight Booking - Selection Screen - Microsoft Internet Explorer provided by SAP IT
File Edit View Favorites Tools Help
Back Search Favorites History
Go
Links SAP Enterprise Portal 5.0 SPIEGEL ONLINE BIS . Main . WebHome SAP TechEd '03 Basel, Session Proposals Admin site SAP TechEd '03 Las Vegas, Session Proposals Admin site
Display Flights
Done
Local intranet

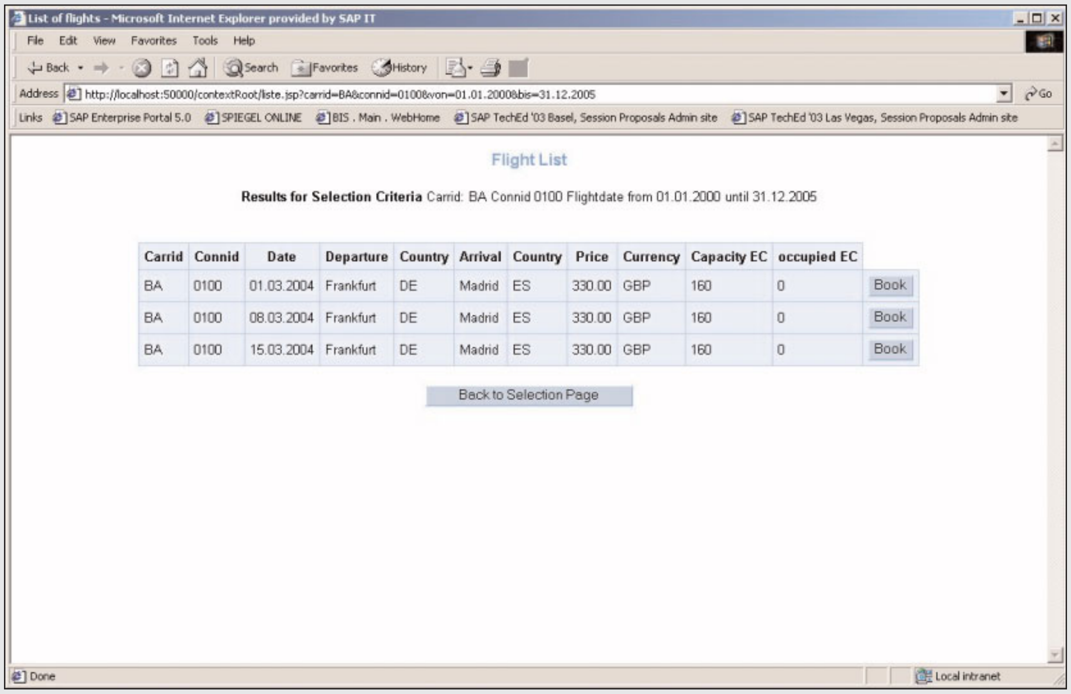
Address http://localhost:50000/contextRoot/auswahl.jsp?carrid=BA

Welcome at Flight Booking

Selection Criteria for Flights

Flight Carrier	British Airways
Flight Number	0100
Flight date from:	01.01.2000 until 31.12.2005

Figure 3 *Flight Results Retrieved by the Search*


List of flights - Microsoft Internet Explorer provided by SAP IT
File Edit View Favorites Tools Help
Back Search Favorites History
Go
Links SAP Enterprise Portal 5.0 SPIEGEL ONLINE BIS . Main . WebHome SAP TechEd '03 Basel, Session Proposals Admin site SAP TechEd '03 Las Vegas, Session Proposals Admin site
Back to Selection Page
Done
Local intranet

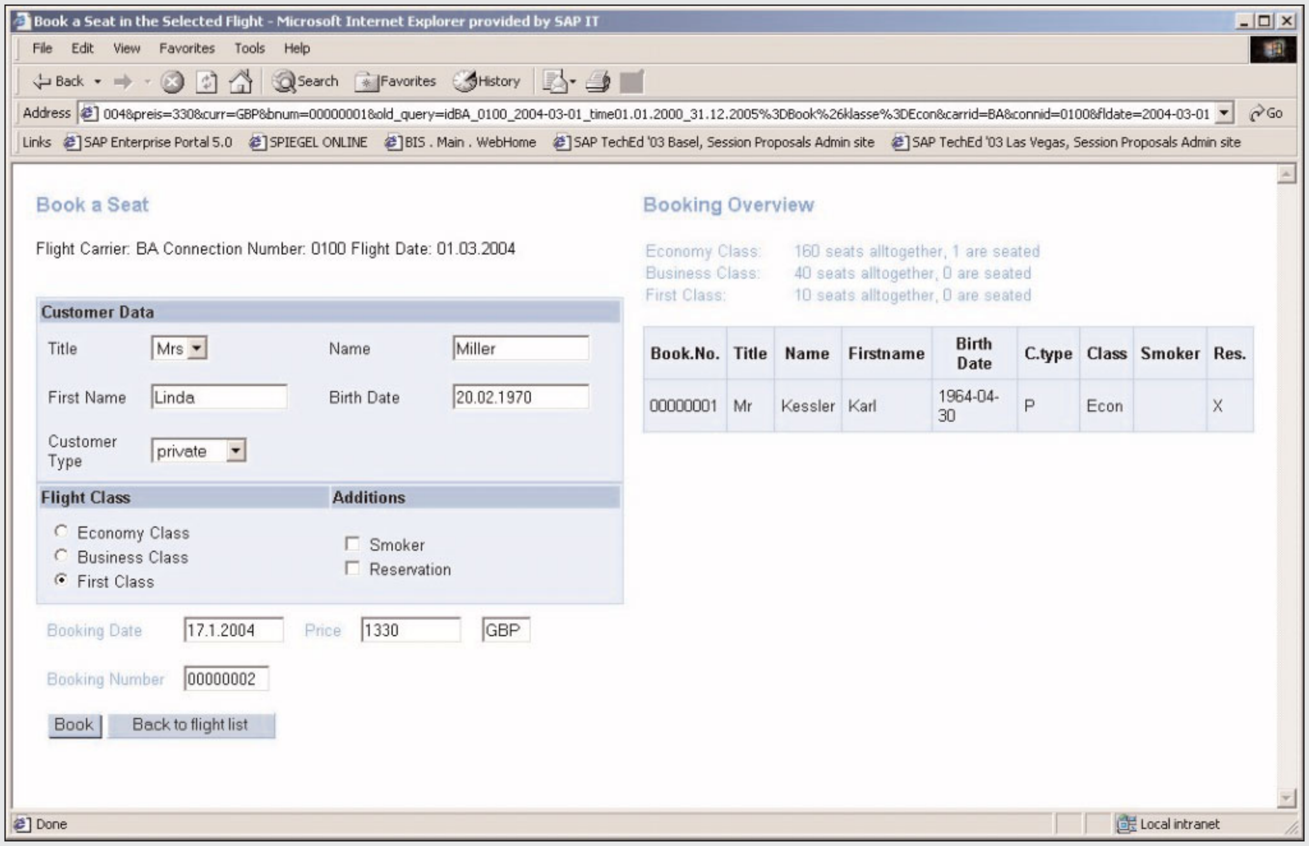
Address http://localhost:50000/contextRoot/liste.jsp?carrid=BA&connid=0100&von=01.01.2000&bis=31.12.2005

Flight List

Results for Selection Criteria Carrid: BA Connid 0100 Flightdate from 01.01.2000 until 31.12.2005

Carrid	Connid	Date	Departure	Country	Arrival	Country	Price	Currency	Capacity EC	occupied EC	
BA	0100	01.03.2004	Frankfurt	DE	Madrid	ES	330.00	GBP	160	0	Book
BA	0100	08.03.2004	Frankfurt	DE	Madrid	ES	330.00	GBP	160	0	Book
BA	0100	15.03.2004	Frankfurt	DE	Madrid	ES	330.00	GBP	160	0	Book

Figure 4 *Flight Booking Page*


Book a Seat in the Selected Flight - Microsoft Internet Explorer provided by SAP IT

File Edit View Favorites Tools Help
Back Search Favorites History

Address 004&preis=330&curr=GBP&bnum=00000001&old_query=idBA_0100_2004-03-01_time01.01.2000_31.12.2005%3DBook%26klasse%3DEcon&carrid=BA&connid=0100&fldate=2004-03-01

Go
Links SAP Enterprise Portal 5.0 SPIEGEL ONLINE BIS . Main . WebHome SAP TechEd '03 Basel, Session Proposals Admin site SAP TechEd '03 Las Vegas, Session Proposals Admin site

Book a Seat

Flight Carrier: BA Connection Number: 0100 Flight Date: 01.03.2004

Customer Data

Title	Mrs	Name	Miller
First Name	Linda	Birth Date	20.02.1970
Customer Type	private		

Flight Class

- Economy Class
- Business Class
- [✓ First Class]

Additions

- ☐ Smoker
- ☐ Reservation

Booking Date 17.1.2004 Price 1330 GBP

Booking Number 00000002

Booking Overview

Economy Class:	160 seats alltogether, 1 are seated
Business Class:	40 seats alltogether, 0 are seated
First Class:	10 seats alltogether, 0 are seated

Book.No.	Title	Name	Firstname	Birth Date	C.type	Class	Smoker	Res.
00000001	Mr	Kessler	Karl	1964-04-30	P	Econ		X

- **Figure 4** shows the booking page. On the left-hand side, the user enters details like the desired flight class and clicks on the *Book* button. If the flight is booked successfully, the table on the right-hand side is updated immediately with a confirmation of the user's booking.

To gain some practice using SAP NetWeaver Developer Studio and Java, let's first implement the selection screen of this application (Figure 2) in a straightforward way using JavaServer Pages (JSPs) and Java Database Connectivity (JDBC). We'll then add the booking screen (Figure 4) in a more extensible, robust way using Enterprise JavaBeans (EJBs).

But before we start creating the application, we need to create the backend database that serves as the data source.

✓ ***Note!***

You can download the full source code for the JSP/JDBC example and the EJB example, including the flight results page and a configuration page, at ***www.SAPpro.com****.*

✓ *Demo Prerequisites*

To build, debug, and run the demo application locally, the following elements must be in place:

- *SAP NetWeaver Developer Workplace 6.40 must be installed on your Windows 2000 or XP system. An evaluation version is available at* **http://sdn.sap.com**. *SAP NetWeaver Developer Workplace 6.40 includes SAP NetWeaver Developer Studio 6.40 and supporting components (e.g., a SQL database installation).*
- *A Microsoft Internet Explorer 5.5 or higher browser must be installed.*

Defining the Backend Database

The first step in developing the demo is to create a small database that stores flight information and user bookings.

While we have several choices for storing the data (in an SAP system database, in a Microsoft Access database, or in a third-party SQL server, for example), for simplicity we'll store the example data in the central SQL database installed with SAP NetWeaver Developer Workplace. This choice also allows us to explore a very powerful feature of SAP NetWeaver Developer Studio called the Dictionary perspective,[4] which enables you to create and manage tables in the central database of your SAP Web AS installation.[5]

[4] The Eclipse user interface organizes information into views and perspectives. A view is a small window that contains specific information like lists of files in your project. A perspective is a collection of views related to a given task like debugging or coding. To open a specific view, go to *Window → Show View → Other*. To see a list of available perspectives, go to *Window → Open Perspective → Other*.

[5] As of SAP NetWeaver Developer Studio 6.40, the Dictionary perspective can only be used to access and maintain tables in the SAP Web AS central database or a local database installed with SAP NetWeaver Developer Workplace — i.e., you can't use it to view or modify other SQL databases.

✓ *Note!*

Each SAP Web AS installation is built around one or more "central" SQL databases. During installation, the administrator must choose whether to store the platform's ABAP and Java data in two separate databases or in a single database with separate schemas. In either case, ABAP and Java data are kept entirely separate, and transactions can never span both the ABAP and Java sides. This separation provides stability and the flexibility to run just the ABAP or Java environment.

To simplify development and testing, SAP NetWeaver Developer Workplace installs a small version of the SAP Web AS system's central database locally on your PC for the Java environment. SAP NetWeaver Developer Studio is configured to work with this database by default. You can also opt to work directly with the SAP Web AS system's database from within the SAP NetWeaver Developer Studio preferences dialog (Window → Preferences).

Figure 5 *Table Definitions*

SCARR_BA

```
SCARR_BA (
CARRID string 3 key,
CARRNAME string 25 not null)
```

SBOOK_BA

```
SBOOK_BA (
CARRID string 3 key,
CONNID string 4 key,
FLDATE date key,
BOOKID string 8 key,
CUSTTYPE string 1 not null,
FCLASS string 5 not null,
SMOKER string 1,
LOCCURAM decimal 15,2,
LOCCURKEY string 5 not null,
ORDER_DATE date,
CANCELLED string 1,
RESERVED string 1,
PASSNAME string 25 not null,
FORENAME string 15 not null,
PASSFORM string 15 not null,
PASSBIRTH date not null)
```

SFLIGHT_BA

```
SFLIGHT_BA (
CARRID string 3 key,
CONNID string 4 key,
FLDATE date key,
DEPTIME time not null,
ARRTIME time not null,
COUNTRYFR string 3 not null,
CITYFR string 20 not null,
COUNTRYTO string 3 not null,
CITYTO string 20 not null,
PRICE decimals 15,2,
CURRENCY string 5,
SEATSMAX integer,
SEATSOCC integer,
SEATSMAX_B integer,
SEATSOCC_B integer,
SEATSMAX_F integer,
SEATSOCC_F integer)
```

Three tables (defined in **Figure 5**) will be sufficient to store the flight data for the example:

- **SCARR_BA** to provide data for the flight selection screen (Figure 2)
- **SFLIGHT_BA** to provide data for the flight results screen (Figure 3)
- **SBOOK_BA** to provide data for the booking screen (Figure 4)

Creating these tables is easy. Here's what to do:

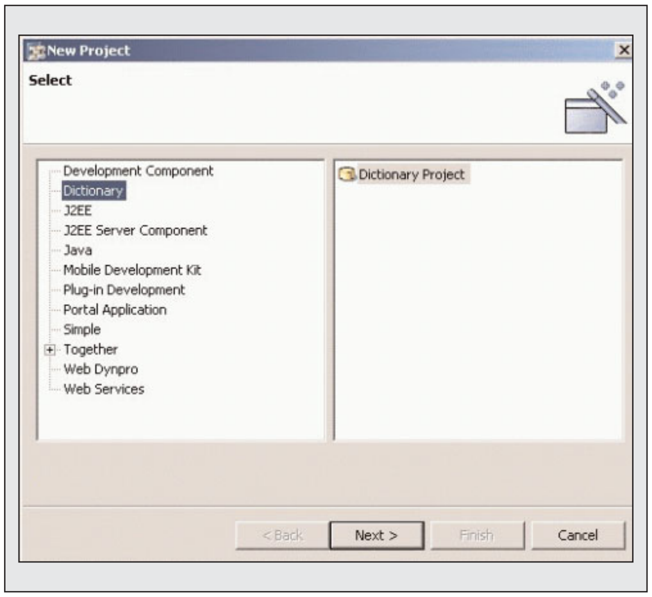
1. Double-click on your SAP Microsoft Management Console (SAPMMC) shortcut and check the status of the J2EE Engine. If all lights are green, the engine is up and running. If not, right-click on the instance name[6] and choose *Start* from the context menu.

> ✓ ***Note!***
>
> *If you don't have access to the SAPMMC, either assume the J2EE Engine is running or confirm it with your administrator.*

2. Start SAP NetWeaver Developer Studio from your desktop and choose *File* → *New* → *Project*.

[6] The instance name that appears in your console will depend on your installation options. The default name is *C11*.

Figure 6 Create a New Project



3. A pop-up appears asking for the project type (see **Figure 6**). Select *Dictionary* in the left pane, highlight *Dictionary Project* in the right pane, and press *Next* to launch the Dictionary project wizard.
4. Enter *FlightDB* for the project name and press *Finish*, which will enable the Dictionary perspective.
5. Expand the *FlightDB* project structure completely in the Dictionary Explorer view in the upper left pane (see **Figure 7**).
6. Right-click on the item *Database Tables* and select *Create Table* from the context menu.
7. In the pop-up dialog, enter table name *SCARR_BA*, which will open the table editor in the upper right pane (also shown in Figure 7).

Figure 7 SAP NetWeaver Developer Studio Dictionary Perspective

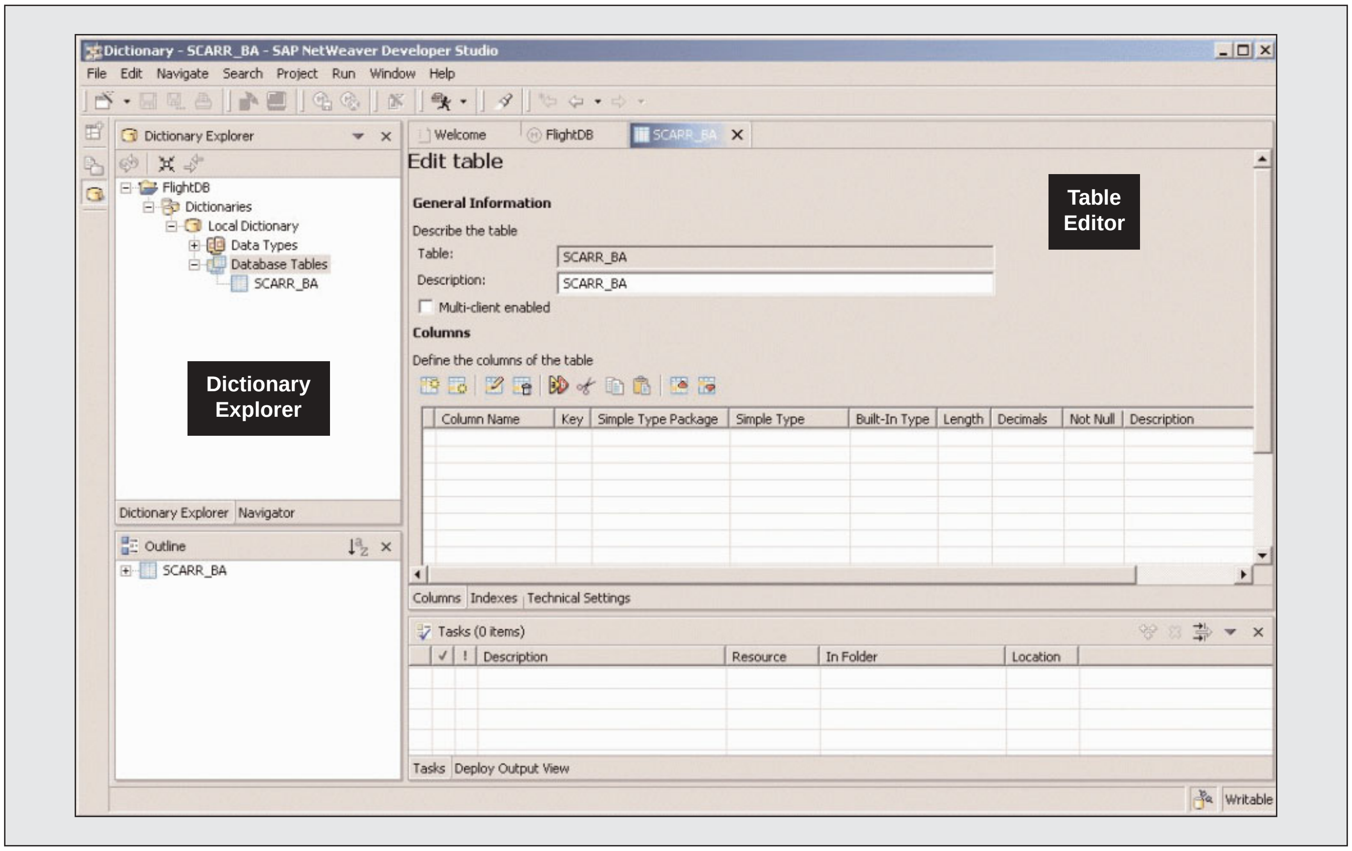
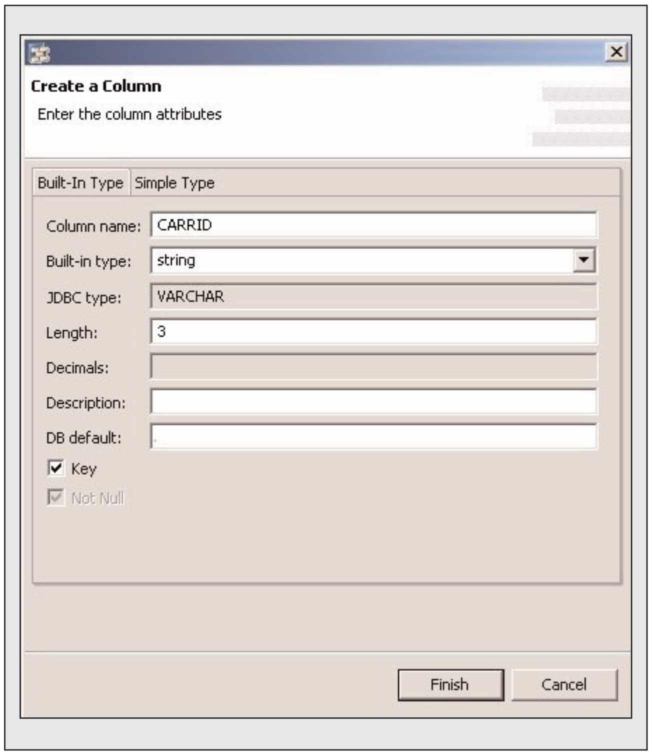


Table: SCARR_BA
Description: SCARR_BA

Column Name	Key	Simple Type Package	Simple Type	Built-In Type	Length	Decimals	Not Null	Description

✓	!	Description	Resource	In Folder	Location

Figure 8 Add Columns to the Database Table



Create a Column

Enter the column attributes

Column name: CARRID
Built-in type: string
JDBC type: VARCHAR
Length: 3
Decimals:
Description:
DB default:
[✓ Key]
[✓ Not Null]

8. Click on the *Add Column* button (**)** in the table editor. In the pop-up, specify the column name, type, length, and key property (see **Figure 8**), and press *Finish*.

9. Repeat step 8 for all of the table columns (refer back to Figure 5 for the values).

10. Select the *Save All Metadata* button (**)** from the SAP NetWeaver Developer Studio standard toolbar, which will save the table definition to the local file system.

11. Repeat steps 6-10 for tables SFLIGHT_BA and SBOOK_BA (again, refer back to Figure 5 for the values).

12. Finally, right-click on the *FlightDB* project name in the Dictionary Explorer view and select *Rebuild Project* → *Create Archive* → *Deploy* from the context menu. This will create a DDL script that is executed immediately on the database.

A pop-up that indicates whether the deployment was successful will appear. If the deployment was not successful, it will identify potential sources of error.

Using SQL Studio to Add Data to the Database Tables

So far, we have created three new database tables in our underlying database — SCARR_BA, SFLIGHT_BA, and SBOOK_BA. Now we need to add data to the tables. SAP NetWeaver Developer Studio does not offer a built-in data browser or table maintenance function, so you must use a utility of the underlying database software to examine and modify the contents of the tables. SQL Studio, the tool for MaxDB,[7] is installed with SAP NetWeaver Developer Workplace, and is what I will use to demonstrate the steps here.

✓ ***Tip***

If you are adept at SQL, you can instead use the SQL Dialog tool to perform the table inserts directly. Choose the exclamation mark from the toolbar to submit your query.

Here's what to do:

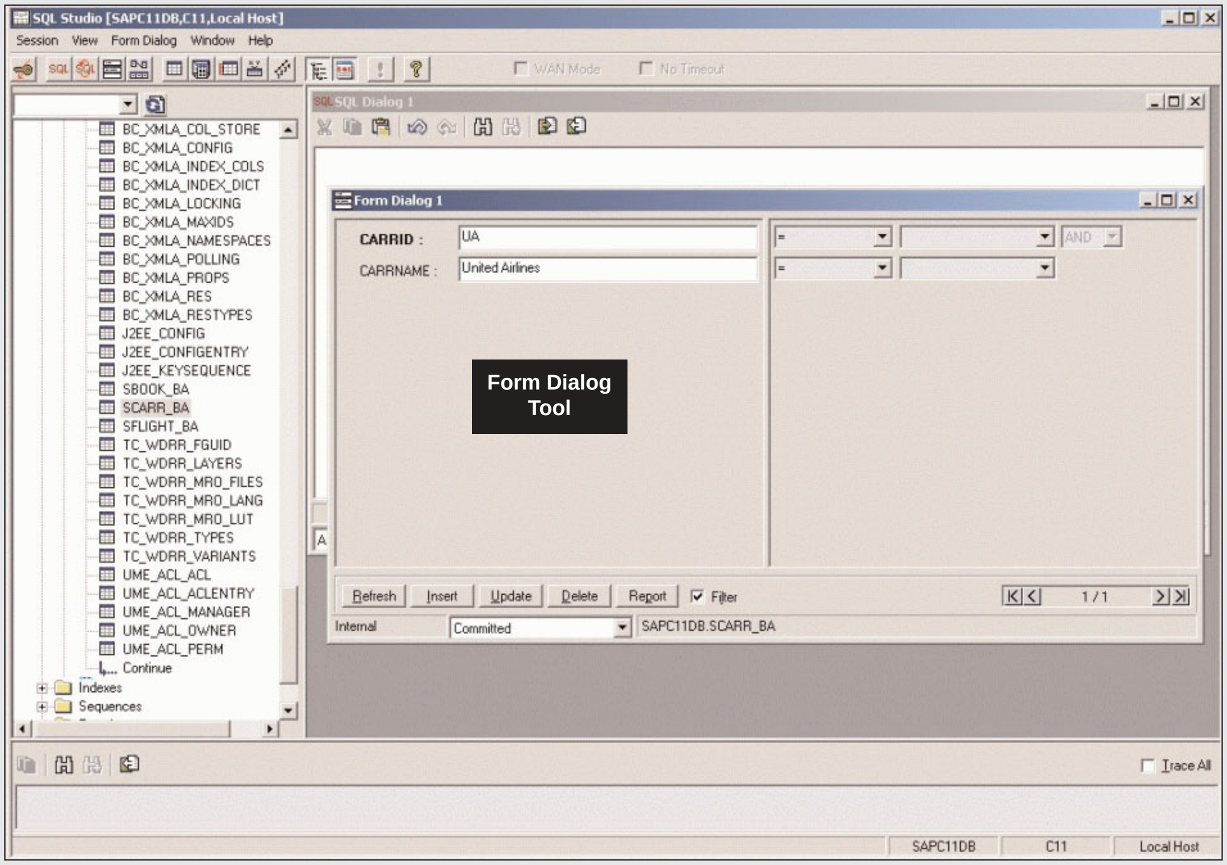
1. Launch SQL Studio from the desktop.

2. Log on as a database administrator.

3. Select the Form Dialog tool (**)** from the SQL Studio standard toolbar, which opens the screen shown in the right pane of **Figure 9**.

[7] SAP Web AS runs on all database platforms supported by SAP (Oracle, DB2, etc.), including MaxDB, which is a rebranded and enhanced version of SAP DB, SAP's open source, enterprise-level database. You can learn more at **www.mysql.com/maxdb**.

Figure 9 *Load the New Tables with Test Data*


SQL Studio [SAPC11DB,C11,Local Host]
Session View Form Dialog Window Help
SQL Dialog 1
Form Dialog 1
Form Dialog Tool
Refresh Insert Update Delete Report Filter
1 / 1
Trace All
SAPC11DB C11 Local Host

CARRID : UA
CARRNAME : United Airlines
Internal Committed SAPC11DB.SCARR_BA

BC_XMLA_COL_STORE, BC_XMLA_CONFIG, BC_XMLA_INDEX_COLS, BC_XMLA_INDEX_DICT, BC_XMLA_LOCKING, BC_XMLA_MAXIDS, BC_XMLA_NAMESPACES, BC_XMLA_POLLING, BC_XMLA_PROPS, BC_XMLA_RES, BC_XMLA_RESTYPES, J2EE_CONFIG, J2EE_CONFIGENTRY, J2EE_KEYSEQUENCE, SBOOK_BA, SCARR_BA, SFLIGHT_BA, TC_WDRR_FGUID, TC_WDRR_LAYERS, TC_WDRR_MRO_FILES, TC_WDRR_MRO_LANG, TC_WDRR_MRO_LUT, TC_WDRR_TYPES, TC_WDRR_VARIANTS, UME_ACL_ACL, UME_ACL_ACLENTRY, UME_ACL_MANAGER, UME_ACL_OWNER, UME_ACL_PERM, Continue, Indexes, Sequences

4. In the pane on the left, scroll down to the tables added in the previous section and drag table *SCARR_BA* to the Form Dialog pane.

5. Enter a carrier ID and name (*UA* and *United Airlines* in the example) and click on the *Insert* button.

6. Add 3-5 carriers to the table.

7. Finally, add 5-10 flight records to the SFLIGHT_BA table in a similar fashion.

Remember the values you've entered here — you'll need them to test your application.

Now that we've got our backend database set up, we're ready to start creating the application itself. In the next section, we'll create the static portion of the application user interface using JSPs.

Implementing the Static Portion of the User Interface with JSPs

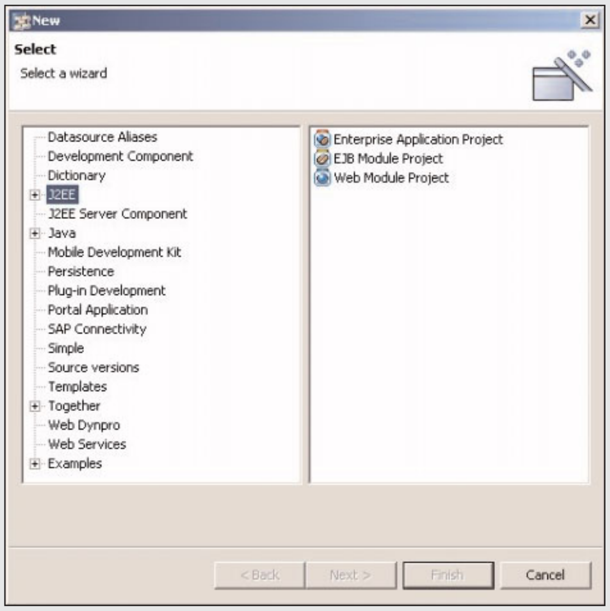
While you'll probably favor developing most of your Java user interfaces with Web Dynpro[8] due to its simplicity, visual consistency, and maintainability, sometimes you might just want to code a simple Java

[8] Web Dynpro, available with SAP Web AS 6.30 and higher, is a new visual programming model for building Web-based business applications. Web Dynpro for Java is new as of SAP Web AS 6.40.

application using JSPs. JSPs let you dynamically generate HTML code for all or part of a Web page at runtime by placing Java code in the page alongside traditional HTML. When the page is requested by a browser, the Web server executes the code and folds any resulting HTML into the final, static HTML page. Since all of this is done on the Web server, the page received by the browser looks just like any other static HTML page, so there are no special frontend requirements for using JSPs. SAP NetWeaver Developer Studio accelerates JSP development with features like JSP/HTML automatic tag completion and an integrated debugger that lets you step through the Java code within your JSP pages. To demonstrate SAP NetWeaver Developer Studio's powerful JSP features, this is the approach we'll use to start building the example application.

We'll start by creating the static portion of the example application's selection screen.

Figure 10   The Project Creation Wizard



Step 1: Create a Web Project

We first need to create a Web project for the JSP application:

1. Open the J2EE Development perspective within SAP NetWeaver Developer Studio via menu path *Windows → Open Perspective → Other*. Select *J2EE Development* and press *OK*.

2. Click on the *New* button ( ), which launches the project creation wizard (**Figure 10**).

3. Select *J2EE* in the left pane. The right pane lists the three supported J2EE project types: *Enterprise Application Project*, *EJB Module Project*, and *Web Module Project*.

4. Select *Web Module Project* and press *Next*.

5. Specify a name for the Web project (*FlightWeb* in the example).

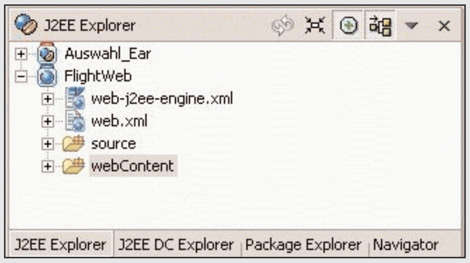
6. Press *Finish*.

✓ *Note!*

The Enterprise Application Project type is for bundling and deploying J2EE components. The EJB Module Project type is for Enterprise JavaBeans development. The Web Module Project type is for developing JSP/servlet applications. We will use Web Module Project here and the other two project types later in the article.

An outline of the Web module project (see **Figure 11**) appears in the J2EE Explorer view of SAP NetWeaver Developer Studio (where the Dictionary Explorer view appeared in Figure 7, when the Dictionary perspective was enabled). Notice the two standard "Web descriptors" that were automatically added to the project (*web.xml* and *web-j2ee-engine.xml*). The *web.xml* file controls standard properties of the Web user interface, like context parameters and URL mappings; the *web-j2ee-engine.xml* file contains parameters that are specific to

Figure 11 Outline of the Web Module Project in the J2EE Explorer View



the J2EE Engine, such as login configurations and the alerts to raise when the engine is about to shut down. The system uses these files to control the servlet container's behavior when executing the Web application. This kind of deployment information is separated from program code to increase portability among different J2EE servers.[9]

You'll also notice two folders within your project in the J2EE Explorer view, *source* and *webContent*. The *source* folder will house all the Java packages used in the Web tier of the application, such as *com.sap.container* and *com.yourname.yourpackage*. The *webContent* folder will house objects like JSP pages, graphics files, JavaScript libraries, and cascading style sheets (CSS).

Step 2: Create the Selection Page

To create a JSP page for the selection screen, follow these steps:

1. Right-click on the *webContent* folder in the J2EE Explorer view and select *New* → *JSP* from the context menu.
2. Specify *selection.jsp* as the name for the first JSP page.
3. Press *Finish*.
4. Access the JSP editor by selecting the *Source* tab in the upper right pane of the SAP NetWeaver Developer Studio screen (where the table editor appeared in Figure 7, when the Dictionary perspective was enabled). The JSP editor will contain some standard starter code for creating the JSP page.

[9] For more on portability, see the article "Gain a Real-World Understanding of How Your Applications Will Operate on a New Platform — Porting a J2EE Application to SAP Web Application Server" on page 69 of this issue.

✓ ***Tip***

While the JSP editor is more or less a basic source editor, it has some "smarts" that will accelerate coding, such as auto-completion for JSP and HTML tags. If you type "<" in the JSP editor, for example, a dropdown list of available JSP and HTML tags appears. As you type further, the list scrolls to the first matching item. If you select a tag from the list, such as <jsp:useBean>, the editor offers the attributes of the selected tag.

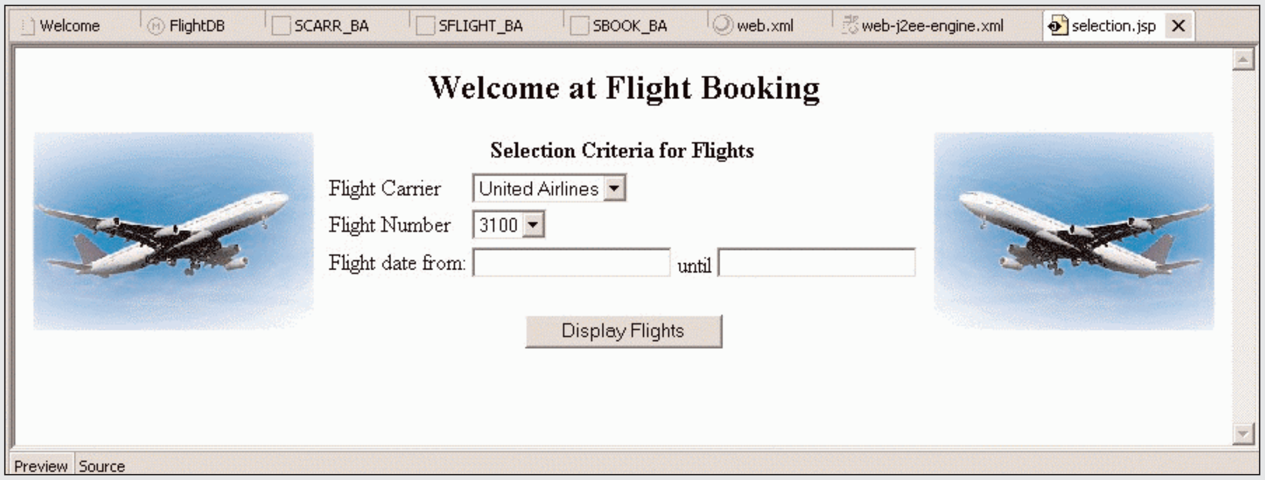
Step 3: Add Static HTML to the Selection Page

The next step is to add the static HTML parts to the code for the *selection.jsp* page. The complete source code is listed in Figure A in the appendix (see page 41). Note that I've used a static HTML form to provide selections on airline, connection ID (flight number), and flight date for a user-defined interval of departure dates.

✓ ***Tip***

The HTML code in Figure A in the appendix references two graphics files that are contained in the ZIP file available at ***www.SAPpro.com****. To import the files into your Web project, right-click on the webContent folder in the J2EE Explorer view, select Import File, and locate the files one at a time using the file selection box. You should then see them listed in the webContent folder.*

Figure 12 *Preview the HTML Layout in the JSP Editor*



Welcome at Flight Booking

Selection Criteria for Flights

Flight Carrier: United Airlines
Flight Number: 3100
Flight date from: until

Display Flights

To verify the code, click on the *Preview* tab in the JSP editor to see the HTML layout. Your output should resemble **Figure 12**.

Step 4: Add a Cascading Style Sheet (CSS)

Next, let's quickly improve the application's appearance using a cascading style sheet (CSS) — a collection of style rules that specify font sizes and other formatting that the browser should apply when it encounters a specific HTML tag or symbolic name. You can create style sheets as text files inside your Web project. Simply right-click on the *webContent* folder in the J2EE Explorer view, choose *File* → *New* from the context menu, and enter a name for the file (e.g., *style.css*). A text editor will appear. Enter the code listed in Figure B in the appendix (see page 42), and then add the following line to the header of the selection screen page:

```
<link rel="stylesheet"
     type="text/css"
     href="style.css">
```

This tells the browser to download and read the formatting rules defined in *style.css*.

Refresh the preview pane by selecting the *Preview* tab again (see **Figure 13**). Looks much better, right?

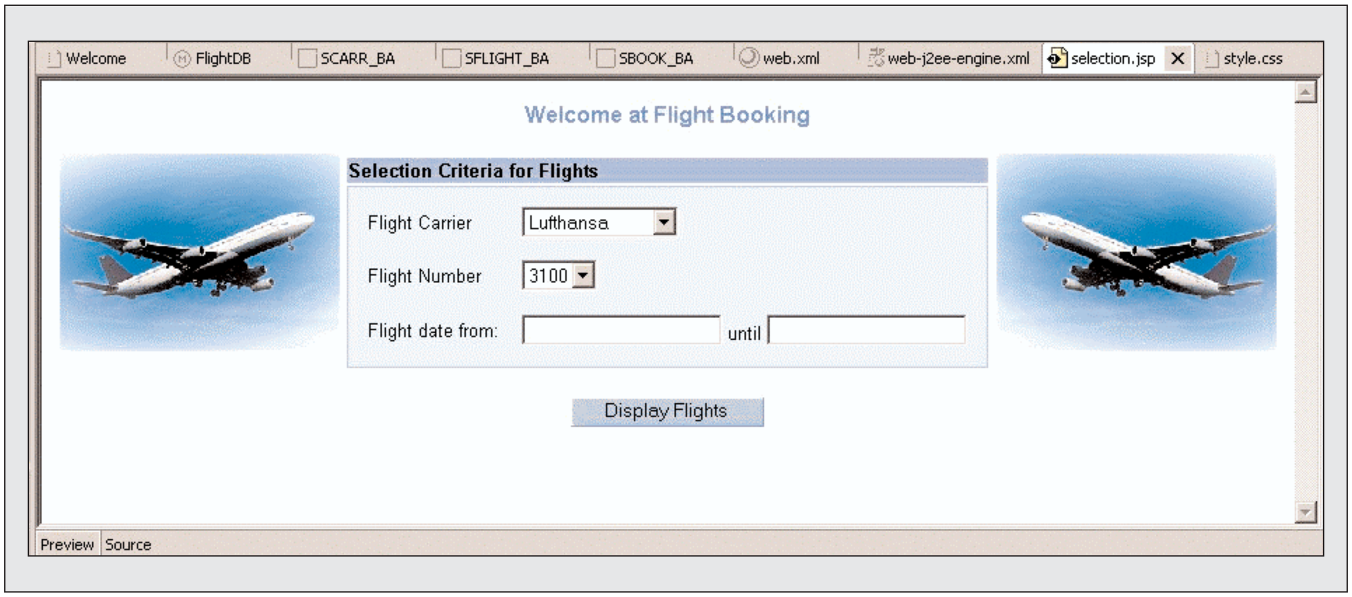
✓ ***Note!***

If you are unfamiliar with CSS, don't worry — the concept is easy to understand with a bit of background. Each entry (rule) in a style sheet maps a set of formatting settings either to all HTML tags of a specific type, or to a symbolic name defined and specifically assigned to an HTML element via an attribute called "class." An example of the former are rules 1, 2, and 3 in Figure B in the appendix, which instruct the browser to render all headings, paragraphs, lists, tables, etc., in Arial font. Rules 4, 5, and 6 are examples of the latter.

Learning and using cascading style sheets is time well spent, because you can instantly alter font, size, positioning, and other types of formatting across the site by modifying a single file, and without cluttering pages with formatting code.

So far, we've created a Web project for the JSP application, added static HTML to the selection screen page, and greatly improved its appearance by applying a cascading style sheet. Now comes the fun part: implementing the dynamic portion of the page with Java.

Figure 13 *The Selection Page After Applying the Cascading Style Sheet*



Selection Criteria for Flights

Flight Carrier: Lufthansa

Flight Number: 3100

Flight date from: until

Dynamically Displaying Flight Information from the Database with JDBC and JavaBeans

To make our sample application more "real-world," let's fetch the entries in the *Flight Carrier* and *Flight Number* dropdown lists directly from the database rather than hard-coding the values in the JSP page. The best (and easiest) way to do this is to create a JavaBean that reads data from the database via JDBC,[10] and embed some server-side script in the page to generate HTML from the data stored in public attributes of the JavaBean. While we could technically embed the database access code directly in the JSP page itself, encapsulating it in a reusable JavaBean streamlines the JSP page, improves its robustness and maintainability, and dramatically reduces overall testing time, since each component can be tested independently.

In the next sections, we'll add the dynamic data display to the example application's selection screen.

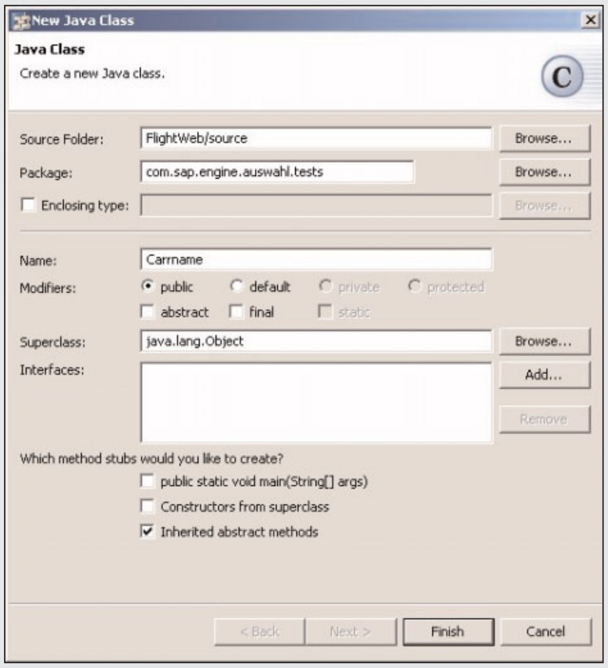
✓ ***Note!***

Unlike with SQL databases, if you want to extract and use SAP data in your Java applications, you'll need to call one or more function modules or BAPIs on the SAP system (vs. accessing the system's database tables directly with JDBC). SAP provides the middleware you need to connect to and make these calls in a package called the SAP Java Connector (JCo), which is conveniently integrated into SAP NetWeaver Developer Studio as a library. The process basically involves using JCo's visual tool (File → New → Other → SAP connectivity → Enterprise Connector) to generate proxies (Java classes) for the function module you want to call, along with its import, export, and table parameters. While generating the proxies with the tool is easy, using them takes a bit more experience.[11]

[10] Java Database Connectivity (JDBC) is the standard Java protocol from Sun Microsystems for interacting with SQL databases, and is analogous to the role of Open Database Connectivity (ODBC) in the Microsoft technology stack.

[11] For more details on using JCo, see the following *SAP Professional Journal* articles by Thomas G. Schuessler: "Repositories in the SAP Java Connector (JCo)" (March/April 2003); "Server Programming with the SAP Java Connector (JCo)" (September/October 2003); and "Tips and Tricks for SAP Java Connector (JCo) Client Programming" (January/February 2004).

Figure 14 The Java Class Wizard


New Java Class

Java Class

Create a new Java class.

Source Folder: FlightWeb/source

Package: com.sap.engine.auswahl.tests

☐ Enclosing type:

Name: Carrname

Modifiers: ● public ○ default ○ private ○ protected

☐ abstract ☐ final ☐ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

☐ public static void main(String[] args)

☐ Constructors from superclass

☑ Inherited abstract methods

Step 1: Create a JavaBean That Will Access the Database

Defining the JavaBean is easy:

1. In the J2EE Explorer view, right-click on the *source* folder to open its context menu.

2. Select *New* → *Java Class*, which launches the Java class wizard (**Figure 14**).

> ✓ ***Note!***
>
> *You may have noticed that we're using a typical Java class wizard here, not a JavaBean wizard. What makes a JavaBean a JavaBean is adherence to a set of standards when coding the class methods. In all other respects, it's just a regular Java class.*

3. Specify the package name *com.sap.engine.auswahl.tests* and the class name *Carrname*.

4. Press *Finish*.

Step 2: Add Code to Access the Database via JDBC

Next, we need to define methods within the JavaBean that fetch records from the database and store them in attributes declared as array variables (similar to ABAP internal tables). You'll find the required code in **Figure 15** and available for download at **www.SAPpro.com**. The source editor should appear immediately after the Java class wizard is done. If it doesn't, just double-click on the class name in the *source* folder in the J2EE Explorer view.

Methods *getCarrArr* and *getConnids* encapsulate the database access based on JDBC. *getCarrArr* joins tables SCARR_BA and SFLIGHT_BA and stores all records found in an array variable called *area*. Method *getConnids* likewise fetches all connection IDs that match the user-specified carrier, *carrid*. To prevent unnecessary database accesses, the methods *getLength* and *getConnLength* first check whether the arrays were fetched earlier and, if not, trigger the fetch. *getCarrid*, *getCarrname*, and *getConnid* fetch the next airline's respective connections in a cursor-oriented fashion. This is useful in order to "loop over the internal tables" *area* and *area2*.

You may have noticed that the code uses a connection object variable called *conn*, but that this variable has not been declared nor the connection created. To keep the code clean, I encapsulated both of these items in a superclass[12] named *Datenzugriff*, which we'll now create. Simply relaunch the Java class wizard, add a new class called *Datenzugriff* to the

[12] In Figure 15, class *Carrname* extends *Datenzugriff*, so *Carrname* is a subclass of *Datenzugriff*, and *Datenzugriff* is a superclass of *Carrname*.

Figure 15 *Methods for Retrieving Records from the Database*

```
package com.sap.engine.auswahl.tests;

/**
 * @author d040404
 *
 * Java Bean with methods to retrieve flight data from
 * a SQL database
 */

import java.sql.*;

public class Carrname extends Datenzugriff {

    private String last;
    private String[][] area;
    private String[] area2;
    private int i = 0;
    private int j = 0;
    private int k = 0;
    private int l = 0;

    public String[][] getCarrArr() throws Exception {
    String carrname, carrid;
    Statement stmt = conn.createStatement();
    try {ResultSet rs = stmt.executeQuery("Select scarr_ba.carrid,
      scarr_ba.carrname from SFLIGHT_BA inner join SCARR_BA ON sflight_ba.carrid =
      scarr_ba.carrid Group By scarr_ba.carrid, scarr_ba.carrname");
    area = new String[2][20];
     while (rs.next()) {
      carrname = rs.getString(2);
      carrid = rs.getString(1);
      area[0][i] = carrid;
      area[1][i] = carrname;
      i = i + 1;
     }
     return area;}
     finally {stmt.close();}

    }

    public String[] getConnids(String carrid)throws Exception{
    String connid;
    PreparedStatement ps = conn.prepareStatement("Select carrid, connid from
      SFLIGHT_BA where carrid = ? GROUP BY carrid,connid");
    if (carrid == null)
    carrid = area[0][0];
    ps.setString(1,carrid);
    try{
```

(continued on next page)

Figure 15 (continued)

```
    ResultSet rs = ps.executeQuery();
    area2 = new String[20];
     while (rs.next()) {
       connid = rs.getString(2);
       area2[k] = connid;
       k = k + 1;
     }
     return area2;
     }
     finally {
       ps.close();}

    }

    public int getConnLength(String carrid) throws Exception {
     if (k == 0) {
           getConnids(carrid);
     }
     return k;
    }

    public String getConnid() throws Exception {
    String connid;
    connid = area2[l];
    l = l+1;
    return connid;
    }

    public int getLength() throws Exception {
    if (i == 0) {
    getCarrArr();}
    return i;

    }

    public String getCarrid() throws Exception {
    String carrid;
    carrid = area[0][j];
    return carrid;
    }

    public String getCarrname() throws Exception {
    String carrname;
    carrname = area[1][j];
    j = j+1;
    return carrname;
    }
}
```

Figure 16 *Establishing a Connection to the Central Database*

```
package
com.sap.engine.auswahl.tests;

import java.sql.*;
import javax.sql.*;
import javax.naming.*;

public class Datenzugriff {

   Connection conn;

public void createConn() throws SQLException {

     if (conn == null){

           try {
                 InitialContext ctx = new InitialContext();
                 DataSource dataSource = (DataSource) ctx.lookup("jdbc/FLUGDATEN");
                 conn = dataSource.getConnection();

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

com.sap.engine.auswahl.tests package, and enter the code in **Figure 16**.

The class contains the code to obtain a connection object from the J2EE server. The one and only method, *createConn*, first looks up connection parameters for the data source — which I named *jdbc/FLUGDATEN* — from the configuration files using the standard JNDI[13] method *lookup.* (We'll create and maintain a resource entry with this name in the *web.xml* deployment descriptor later when we add the booking functionality to the application using EJBs. At that time, you must specify the name *jdbc/FLUGDATEN* exactly as in the JNDI lookup call in Figure 16.) The code then obtains the connection from the J2EE Engine's connection pool and stores the resulting connection object in the *conn* field. Since *Carrname* is a subclass of *Datenzugriff*, field *conn* is globally accessible by its methods.

As you'll see, encapsulating the database access in a JavaBean in this way greatly simplifies creation, testing, and maintenance of JSP pages, and is thus a critical skill to learn when developing dynamic JSP applications.

Step 3: Adjust the JSP Page to Retrieve Data via the JavaBean

To review, the process flow of the selection page is as follows: Initially, the available airline IDs and names are fetched and presented in the *Flight Carrier* drop-down list. The first of the airline IDs is selected by

[13] The Java Naming and Directory Interface (JNDI) provides standard functionality for extracting system or database connection parameters from Java platform configuration files. This frees you from having to hard-code volatile connection information and recompile your classes each time it changes.

default, and the corresponding connection IDs (flight numbers) are displayed in the *Flight Number* drop-down list. Whenever a new airline is selected, the flight numbers need to be updated.

With your JavaBean in hand, all that's left is to add code to the JSP page to:

1. Instruct the JavaBean to retrieve the airlines and flights during page execution.
2. Generate HTML that renders the dropdown lists with the airline and connection IDs.
3. Trigger a page refresh when a new airline ID is selected.

Modify the code of the *selection.jsp* page (Figure A in the appendix) to match **Figure 17**. The important sections are highlighted. Section ❶ executes on the server, and the *<jsp:useBean …>* tag tells it to

Figure 17 Dynamic Code for Loading Result Data from the JavaBean

```
<%@ page language = "java" %>

<jsp:useBean id = "carr" class = "com.sap.engine.auswahl.tests.Carrname" scope =
"page">                                                                        ❶

<HTML>

<HEAD>

<TITLE> Flight Booking - Selection Screen </TITLE>

<link rel="stylesheet" type="text/css" href="style.css">

<script language="JavaScript">                                                 ❷

      function waehle_connid() {
      for(i=0; i < document.auswahl.carrid.length; i++){
      if (document.auswahl.carrid.options[i].selected==true)
      var carr = document.auswahl.carrid.options[i].value;}
      var mylink = "http://localhost:50000/FlightApp/selection.jsp?carrid="+carr;
      window.location.href = mylink;
      }
</script>



</HEAD>

<BODY BGCOLOR="#FFFFFF">

<% carr.createConn(); %>
<h2 align="center"> Welcome at Flight Booking</H2>
<% String selcarr = request.getParameter("carrid"); %>                         ❸
<% String selconn = request.getParameter("connid"); %>

<img src="Flugzeug.gif" align="left" width="200">
<img src="Flugzeug_r.gif" align="right" width="200">
```

(continued on next page)

Figure 17 (continued)

```
<form name="auswahl" action="liste.jsp" onSubmit="return checkForm()">
<table align="center">
<tr>
<th class="highlight">Selection Criteria for Flights</TH>
</tr>
<TR>
<td class="light">
   <TABLE>
    <tr><td>Flight Carrier</TD>                                                 ❹
        <td><select name="carrid" size="1" onChange="waehle_connid()">
        <% int i;
        for (i = 1; i <= carr.getLength(); i++) { %>
        <% String actcarr = carr.getCarrid(); %>
        <option value="<%= actcarr %>" <% if (actcarr.equals(selcarr)) { %> selected
<% } %>>
        <jsp:getProperty name="carr" property="carrname"/></option>
        <% } %>
            </select>
        </TD>
    </tr>
    <tr><td>Flight Number</TD>
        <td><select name="connid" size="1">
            <% int j;
            for (j = 1; j <= carr.getConnLength(selcarr); j++) {
            String actconn = carr.getConnid(); %>
            <option <% if(actconn.equals(selconn)){%> selected  <%}%>><%= actconn
%></option>
            <% } %>
            </select>
        </TD>
    </tr>
    <tr>
    <td><label for="von">Flight date from:</label></td>
    <td><input type="text" id="von" name="von" >
        <label for="bis">until</LABEL>
        <input type="text" id="bis" name="bis" >
    </td>
    </tr>
    </table>


</td>
</tr>
</table>
<br>
  <center><input type="submit" class="button" value=" Display Flights "></center>
<br>
</form>
</BODY>

</HTML>
                                                                                ❺
</jsp:useBean>
```

instantiate the JavaBean so the code in the page can access its methods and attributes. The corresponding tag in section ❺ removes the bean from scope and gives the server license to discard the object from memory.

Section ❷ declares a JavaScript function called *waehle_connid* that adds the currently selected airline as a URL parameter to the current page's URL, and instructs the browser to "refresh" the page immediately by setting the window's *location.href* property to *http://localhost:50000/FlightApp/selection.jsp?carrid=BA*, for example, if the value of *carrid* is *BA*. This code executes solely on the client side.[14] If you look closely at the attributes of the *select* element in section ❹, you'll see that this function is called immediately whenever the user selects a new airline from the drop-down list.

Next, sections ❸ and ❹ execute on the server. Section ❸ reads the incoming URL parameters (if specified) and sets the *selcarr* variable accordingly. Section ❹ renders the airline and flight number drop-down lists with data from the database retrieved via a method call on the JavaBean. If no airline is chosen (e.g., during the initial page load), the first available airline is selected by default.

✓ ***Note!***

In general, JSP code that is executed when serving the page is surrounded by <% … %> delimiters, but developers can also use the more verbose <script language=Java runat=server> … </script> tag pair. One notable exception is tags prefaced by "jsp:" (e.g., <jsp:useBean> in Figure 17), which also run on the server side.

Now that we've coded the database retrieval and static and dynamic portions of the JSP page, we are ready to debug the application.

[14] While we haven't done it here, it is possible to run Java code on the server side.

Test-Driving the Sample JSP Application Locally

To simplify development and debugging, SAP NetWeaver Developer Studio comes with a built-in Java runtime environment and debugger. This lets you test and perfect your code locally without affecting other developers sharing your company's actual J2EE central instance. Local debugging also helps you distinguish problems with your application from connectivity, configuration, or other issues that may occur on the central SAP Web AS instance. But before you can run or debug the application, you must bundle and deploy it to your local J2EE runtime or to an SAP Web AS server.

✓ ***Note!***

Remember, if you prefer to test and deploy your applications on your central SAP Web AS server, just modify the J2EE Engine settings in the SAP NetWeaver Developer Studio preferences dialog (Window → Preferences).

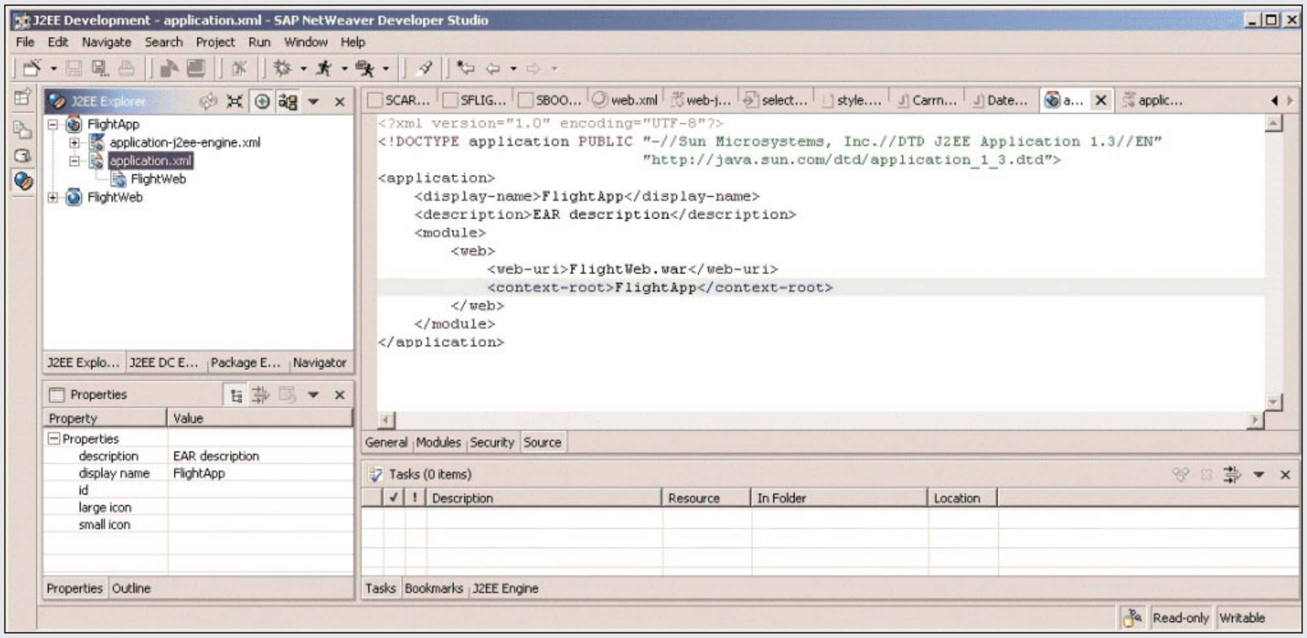
Bundling and Deploying Your Application

You may have heard that Java is a "slick" technology — its approach to application deployment is no exception. Rather than clutter the server with hundreds of individual class files and other resources, Java offers a way to bundle them all together into a neat, compressed enterprise application archive (EAR) file, which functions like a ZIP file. You then deploy[15] this EAR file to the J2EE server and "register" your new application through a series of entries added to a central XML file.

Fortunately, SAP NetWeaver Developer Studio

[15] Deployment is a relatively complex procedure and takes some time to complete. A lot of semantic checks are carried out, and the server environment is updated. It is much more than a copy operation.

Figure 18 The Enterprise Application Project Added to the J2EE Explorer View


J2EE Development - application.xml - SAP NetWeaver Developer Studio
File Edit Navigate Search Project Run Window Help
J2EE Explorer
FlightApp
application-j2ee-engine.xml
application.xml
FlightWeb
FlightWeb
J2EE Explo... J2EE DC E... Package E... Navigator
General Modules Security Source
Tasks (0 items)
Description Resource In Folder Location
Tasks Bookmarks J2EE Engine
Properties Outline
Read-only Writable

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE Application 1.3//EN"
                             "http://java.sun.com/dtd/application_1_3.dtd">
<application>
    <display-name>FlightApp</display-name>
    <description>EAR description</description>
    <module>
        <web>
            <web-uri>FlightWeb.war</web-uri>
            <context-root>FlightApp</context-root>
        </web>
    </module>
</application>
```

Property	Value
Properties	
description	EAR description
display name	FlightApp
id	
large icon	
small icon	

does most of this work for you. All you need to do is create an enterprise application project that includes the module projects you want to deploy, and maintain the application's deployment descriptor.

Let's first create the required enterprise application project:

1. Select *File* → *Project* → *New* from the SAP NetWeaver Developer Studio menu, which starts the project creation wizard (see Figure 10).

2. Choose *J2EE* in the left pane and *Enterprise Application Project* in the right pane, and press *Next*.

3. Enter *FlightApp* as the project name and press *Next*.

4. The system then prompts you to specify the Web or EJB module projects you want to include (deploy) as part of the project. This approach lets you develop and test application components independently, then bundle them together during deployment. Select the *FlightWeb* module project created earlier.

5. Press *Finish*.

Your enterprise application project and associated files will be added to the J2EE Explorer view. Your screen should look similar to **Figure 18**.

The enterprise application project will contain, by default, two descriptor files: a standard descriptor file[16] called *application.xml* and an SAP-specific file called *application-j2ee-engine.xml*, as shown in Figure 18. The *application.xml* file includes entries that control general properties such as the context root and security settings. The *application-j2ee-engine.xml* file contains settings like failover semantics and additional classpath settings that control the behavior of the application when a J2EE cluster node goes down.

[16] The standard descriptor file is common to all J2EE applications and doesn't contain any SAP-specific information.

At this point we need to make a few minor adjustments:

1. Pull up the XML source code for the *application.xml* file by double-clicking on it and selecting the *Source* tab in the pane at the upper right.

2. Search for the string *contextRoot* and replace it with *FlightApp*. This creates a relative path name for your application that you'll use in the URL to access the application (e.g., *http://localhost:50000/FlightApp/selection.jsp*). You can also use this relative path name within your Web pages — for example, to refer to images in an images directory like *FlightApp/images/myimage.jpg*.

3. Next, define any data source aliases used by the application. Remember that in class *Datenzugriff* (Figure 16), we referred to the database as *FLUGDATEN*. Right-click on the name of the enterprise application project (*FlightApp*) in the left pane and select *New* → *META_INF/data-source-aliases.xml*.

4. In the pop-up, name the data source *FLUGDATEN*.

✓ *Note!*

The data source alias refers to the underlying database, so we don't need to specify a host name. The alias is stored in an XML file like a descriptor.

5. Press *Finish*.

6. Next, right-click on the application project name again and select *Build EAR file* from the context menu. After some processing, an EAR file should appear in the J2EE Explorer view (*FlightApp.ear* in the example).

7. Right-click on the EAR file and select *Deploy to J2EE engine* from its context menu. SAP NetWeaver Developer Studio deploys your project files from the development directory to the J2EE Engine on your PC (or your central SAP Web AS server, depending on your settings), and makes them ready in the server to be executed. If all goes well, you should see the message *FlightApp.ear successfully deployed* in the deploy output view, which is opened automatically.

8. Finally, to run the application, pull up a browser window on your PC and type *http://localhost:50000/FlightApp/selection.jsp* to launch the application.

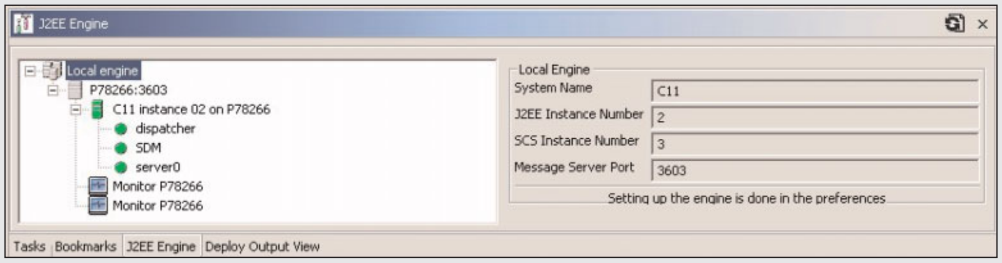
✓ *Tip*

By default, your local J2EE instance is set up to listen and respond on port 50000. If you're testing on an actual SAP Web AS server, replace the localhost and port with the address and port on which your instance is configured to listen.

The resulting page should look similar to the selection screen shown in Figure 2. Test the application by selecting some of the values you defined in the database tables earlier. If you're testing locally and you get a *page not found* message in your browser, check that the context root is set to *FlightApp* in the *application.xml* file — the server might not be recognizing *FlightApp* in the URL. Also, check that the local J2EE Engine is running as described in the next section. If you are testing on an actual SAP Web AS system, your SAP Web AS probably either isn't receiving the HTTP request or isn't configured properly.[17] In both environments, dumps or stack traces generated by the J2EE Engine indicate that there is a problem with the application and that you should debug. We'll look at how to do this next.

[17] Troubleshooting with your SAP Web AS administrator, at least when developing your first few applications, will be the fastest approach.

Figure 19 *The J2EE Engine Status Display*



Local engine
- P78266:3603
  - C11 instance 02 on P78266
    - dispatcher
    - SDM
    - server0
  - Monitor P78266
  - Monitor P78266

Local Engine

System Name	C11
J2EE Instance Number	2
SCS Instance Number	3
Message Server Port	3603

Setting up the engine is done in the preferences

> ✓ ***Note!***
>
> *JavaServer Pages get compiled by the J2EE server the first time they are requested. For this reason, SAP NetWeaver Developer Studio is unable to report compilation errors during deployment. You'll see them in your browser when you try to access the page.*

Debugging the JSP Application

To debug the application, you must first set a break-point and prepare the local J2EE Engine. You can do all of this directly inside SAP NetWeaver Developer Studio. Here's how:

1. Open the JavaBean or JSP page in the source editor. Add a breakpoint by right-clicking in the first column of the desired line and choosing *Add Breakpoint*.

2. Select *Window* → *Show View* → *Other* from the menu.

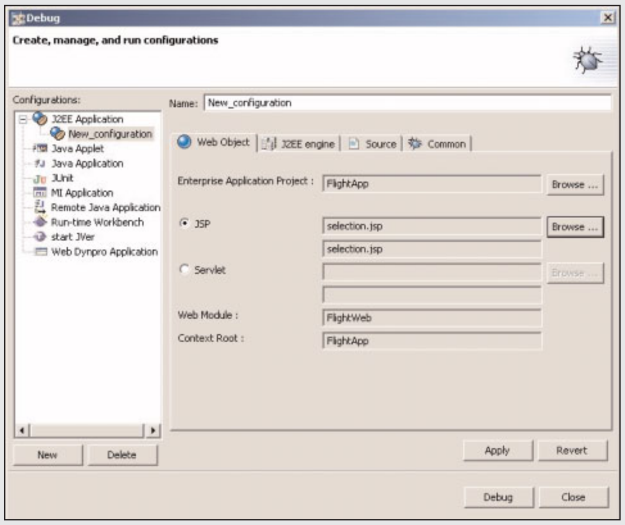
3. In the pop-up, choose *J2EE* → *J2EE engine* and click on *OK*. The lower-right pane of the SAP NetWeaver Developer Studio screen will display the status of the J2EE Engine (see **Figure 19**).

4. Right-click on the *server0* node and select *Enable Debugging of process* from the context menu. This will shut down the local engine and restart it in debug mode.

5. Select the *Debug…* button (**)** from the SAP NetWeaver Developer Studio standard toolbar, which launches the screen shown in **Figure 20**.

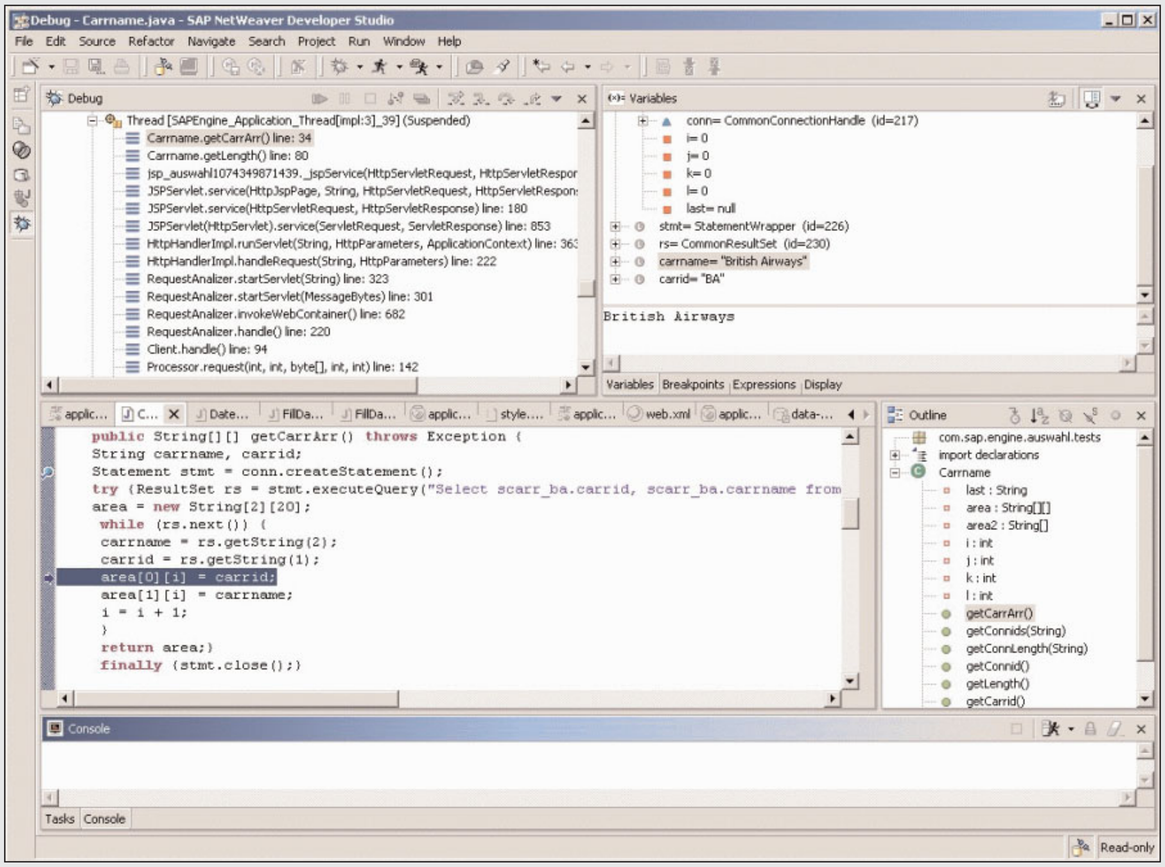
6. Select *J2EE Application* in the left pane and click on the *New* button.

Figure 20 *Debugger Settings Screen*



Name:	New_configuration
Enterprise Application Project :	FlightApp
(●) JSP	selection.jsp
	selection.jsp
( ) Servlet	
Web Module :	FlightWeb
Context Root :	FlightApp

Figure 21 *Debug Display Screen*


Debug - Carrname.java - SAP NetWeaver Developer Studio
File Edit Source Refactor Navigate Search Project Run Window Help
Debug
Variables
Variables Breakpoints Expressions Display
applic... C... Date... FillDa... FillDa... applic... style.... applic... web.xml applic... data-...
Outline
Console
Tasks Console
Read-only

```
Thread [SAPEngine_Application_Thread[impl:3]_39] (Suspended)
    Carrname.getCarrArr() line: 34
    Carrname.getLength() line: 80
    jsp_auswahl1074349871439._jspService(HttpServletRequest, HttpServletRespor
    JSPServlet.service(HttpJspPage, String, HttpServletRequest, HttpServletRespon:
    JSPServlet.service(HttpServletRequest, HttpServletResponse) line: 180
    JSPServlet(HttpServlet).service(ServletRequest, ServletResponse) line: 853
    HttpHandlerImpl.runServlet(String, HttpParameters, ApplicationContext) line: 363
    HttpHandlerImpl.handleRequest(String, HttpParameters) line: 222
    RequestAnalizer.startServlet(String) line: 323
    RequestAnalizer.startServlet(MessageBytes) line: 301
    RequestAnalizer.invokeWebContainer() line: 682
    RequestAnalizer.handle() line: 220
    Client.handle() line: 94
    Processor.request(int, int, byte[], int, int) line: 142
```

```
    conn= CommonConnectionHandle  (id=217)
    i= 0
    j= 0
    k= 0
    l= 0
    last= null
stmt= StatementWrapper  (id=226)
rs= CommonResultSet  (id=230)
carrname= "British Airways"
carrid= "BA"

British Airways
```

```
public String[][] getCarrArr() throws Exception {
String carrname, carrid;
Statement stmt = conn.createStatement();
try {ResultSet rs = stmt.executeQuery("Select scarr_ba.carrid, scarr_ba.carrname from
area = new String[2][20];
 while (rs.next()) {
 carrname = rs.getString(2);
 carrid = rs.getString(1);
 area[0][i] = carrid;
 area[1][i] = carrname;
 i = i + 1;
 }
 return area;}
 finally {stmt.close();}
```

```
com.sap.engine.auswahl.tests
import declarations
Carrname
    last : String
    area : String[][]
    area2 : String[]
    i : int
    j : int
    k : int
    l : int
    getCarrArr()
    getConnids(String)
    getConnLength(String)
    getConnid()
    getLength()
    getCarrid()
```

7. Specify *selection.jsp* as the page to debug, and press *Debug*.

8. The debugger pops up as shown in **Figure 21**. The active thread and call stack are displayed in the upper left, local variables are displayed in the upper right, the code being debugged is displayed in the lower left, and the class's outline is displayed in the lower right. The console across the bottom shows any standard output generated by the code.

The debugger, like the Java source editor, is a standard, built-in Eclipse component. As with most debuggers, you can single-step, step over, step out, and execute entire code sections using the corresponding buttons in the toolbar. Debugging Web-based Java applications is nearly identical to debugging Business Server Pages. While an in-depth discussion about debugging is beyond the scope of this article, here are some tips to help:

- ✓ The debugger lets you debug all types of server-side Java components, like JSP pages, servlets,[18] JavaBeans, and Enterprise JavaBeans, and lets you seamlessly follow the flow of control across different types of objects.

- ✓ The debugger cannot help you debug static HTML pages or client-side script code like JavaScript or VBScript — you must instead use

[18] A servlet is a special Java class that accepts HTTP request information and returns HTML via an HTTP response. Interestingly, during deployment JSP pages are "secretly" converted and compiled into Java classes that behave like servlets.

the Microsoft Script Debugger (available from **www.microsoft.com**).

- ✓ To test code that is "several pages deep" — i.e., code that gets triggered after several request-response cycles[19] — just launch and navigate through the application as usual with your browser. The debugger will pop up when it reaches a breakpoint in the code.

- ✓ Once the debugger is activated, it remains active until you manually turn it off. You should deactivate the debugger when you're done using it to return your applications to "standard speed" and free up server resources.

[19] A request-response cycle is a handoff of control between the browser and the server. A single cycle consists of a browser's request and a server's HTML response. In stateless applications, no information is retained on the server after the response is generated, so the browser must always submit sufficient data to fulfill a given request.

Improving Reusability and Robustness with Enterprise JavaBeans

While JSPs, JavaBeans, and JDBC offer an easy, modular way to develop Web applications, an even better approach became available with J2EE version 1.3[20] with the introduction of Enterprise JavaBeans (EJBs), which come in two main types: *session* beans and *entity* beans (see the sidebar below). This approach involves using JSPs and JavaBeans for the presentation layer, implementing the business logic with a number of session beans, and modeling the database layer with entity beans. Given the enhanced scalability and flexibility of EJBs over JavaBeans, this approach yields a more robust, reusable design structured more around business

[20] As you may recall, SAP NetWeaver Developer Studio supports J2EE 1.3.

What Are Enterprise JavaBeans (EJBs)?

Two types of Enterprise JavaBeans (EJBs) are used in applications that require synchronous data flow, like the example application: entity beans and session beans. (A third type of bean, message-driven beans, are used for asynchronous data flow; this type is not discussed here.)

Entity Beans

Entity beans represent business objects stored in an underlying database system (e.g., an order or a reservation). The state of an entity bean gets saved to the database when the application finishes or a transaction on that entity commits.

Entity beans come in two flavors: with bean-managed persistence (BMP) and with container-managed persistence (CMP). With BMP, you must manually implement the database storage and retrieval logic with JDBC. This gives you the most control over how you retrieve the data, perform transformations, etc. With CMP, the database storage and retrieval is handled automatically by the EJB container (i.e., the EJB's runtime environment). All you need to do is tell the container how to map the entity bean's fields to database table fields at design time using a visual tool. In the example, we will use CMP to demonstrate the design-time capabilities of SAP NetWeaver Developer Studio and the power of EJB functionality within the J2EE Engine.

Session Beans

Session beans encapsulate the transient state of a business application, such as processing an order. Session beans can be stateful or stateless. Stateful session beans get created when the first page is requested and remain live during the entire session. Stateless session beans do not preserve their state. When the browser sends a new page request, the session bean is created anew. Information from a previous page is no longer available. In the demo application, processing associated with a flight reservation is a good candidate for being modeled as a stateless session bean, because once the flight is booked, the session information is no longer needed, and you can continue with the next flight booking.

(continued on next page)

objects, functions, and rules than the nuances of the particular business application.

To gain some experience in working with EJBs, let's add booking functionality to the sample application by creating a session bean and an entity bean in a separate project.

Step 1: Create the SBOOK_CMP Table

To better separate the effects of the JDBC- and EJB-based demos, let's create a copy of the SBOOK_BA table we created earlier to use for the EJB demo:

1. Open the Dictionary perspective, as described previously.
2. Select the *SBOOK_BA* table in the Dictionary Explorer view.
3. Right-click on the table and select *Copy* from the context menu.
4. Select the *Database Tables* folder and choose *Paste* from the context menu.
5. In the pop-up, name the table *SBOOK_CMP*.
6. To simplify the EJB handling, move the *bookid* field to the first line and make it the only key field.
7. Save the meta-model.
8. Choose *Deploy Table* from the table's context menu.

The next step is to create an entity bean that represents a flight booking stored in the SBOOK_CMP table. All of the database access code will be encapsulated in this entity bean to keep the application logic in the session bean "clean."

Step 2: Create a New EJB Project to House the Entity and Session Beans

Before we can create the entity bean, we must first create an EJB project:

1. Select *File → New → Project* to start the project creation wizard (see Figure 10).

(continued from previous page)

Use stateless session beans whenever possible, since they save memory and offer better scalability. When you need to preserve data between page requests, use stateful session beans and explicitly release them from memory as soon as possible.

What's the Difference Between EJBs and JavaBeans?

Technically, EJBs and JavaBeans are just regular Java classes that comply with the structural and naming conventions defined in their respective standards. Both JavaBeans and EJBs provide a way to encapsulate and reuse code in an independently deployable object, much like classes or function groups in the ABAP world.

EJBs go far beyond JavaBeans in their level of scalability, reusability, and support for componentization, although this flexibility does come with some performance overhead. The main operational difference between EJBs and regular JavaBeans are that EJBs run in an EJB container set up and managed by the J2EE server, while JavaBeans run like JSPs and servlets. The EJB container provides a range of sophisticated, out-of-the-box services to both EJBs and their clients, from remote method invocation, to state management, to thread control and pooling, and much more.

The good news is that developing EJBs is very similar to developing regular JavaBeans, with a few differences in the standard methods the bean must provide. Most of the major differences will only become apparent if you delve into more advanced development techniques.

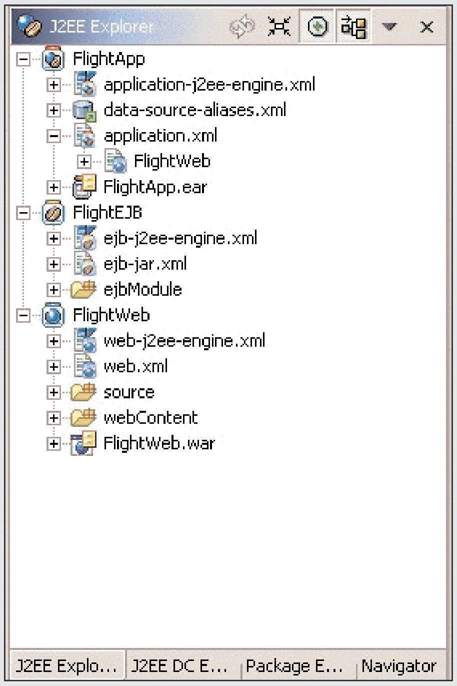
2. Select *J2EE* in the left pane and *EJB Module Project* in the right pane.

3. Specify the project's name — *FlightEJB* in the example.

4. Press *Finish*.

As you can see in **Figure 22**, in addition to the *FlightWeb* Web project and the *FlightApp* enterprise application project created earlier, there is now also a locally created *FlightEJB* project opened in the J2EE Explorer view. The *FlightEJB* project contains a folder called *ejbModule* for the EJBs and two deployment descriptors: a generic one called *ejb-jar.xml* and one for SAP-specific settings called *ejb-j2ee-engine.xml*.

Figure 22 The J2EE Explorer View with the FlightEJB Project Added



Step 3: Add a New Entity Bean to the Project

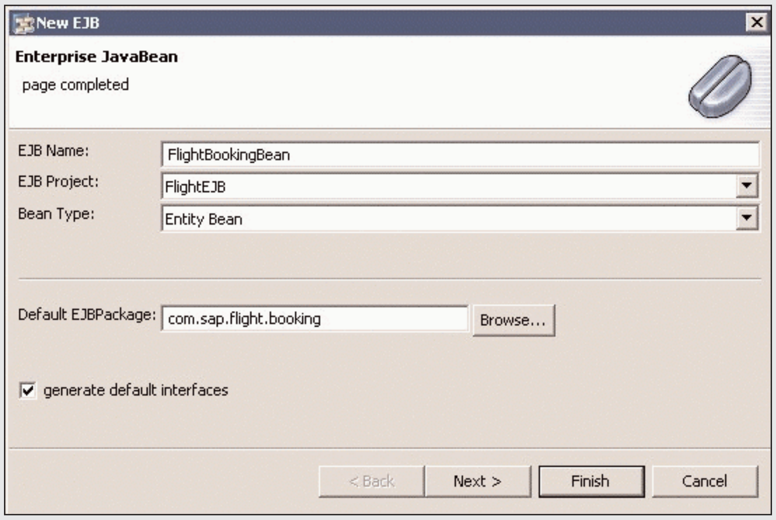
To add a new entity bean to the project:

1. Select the EJB project (*FlightEJB*) in the J2EE Explorer view.

2. Right-click and choose *New* → *EJB...* from the context menu, which will launch the EJB creation wizard (see **Figure 23**).

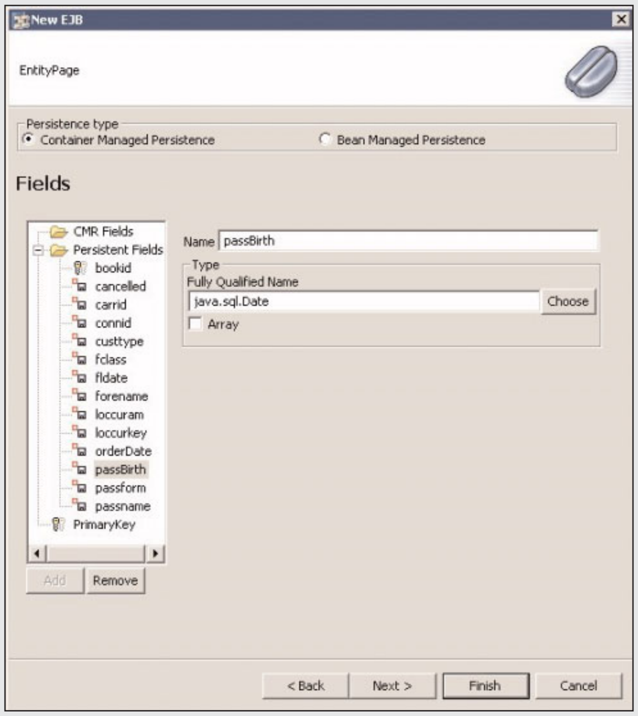
3. Enter *FlightBookingBean* as the EJB name, *com.sap.flight.booking* as the package name, and *Entity Bean* as the bean type.

Figure 23 The EJB Creation Wizard



EJB Name: FlightBookingBean
EJB Project: FlightEJB
Bean Type: Entity Bean
Default EJBPackage: com.sap.flight.booking
☑ generate default interfaces

Figure 24 *Specify Persistence Settings*



4. Press *Next.*

5. A pop-up will appear (see **Figure 24**) for specifying persistence settings. Choose container-managed persistence (CMP), because it is the easier of the two types (the EJB container will handle nearly all the storage details for you).[21]

6. Highlight the *Persistent Fields* folder in the left pane and press *Add*.

7. Enter the new field's name and Java type. Start with *carrid* and type *java.lang.String*.

8. Repeat steps 6 and 7 for all of the fields in

✓ *Note!*

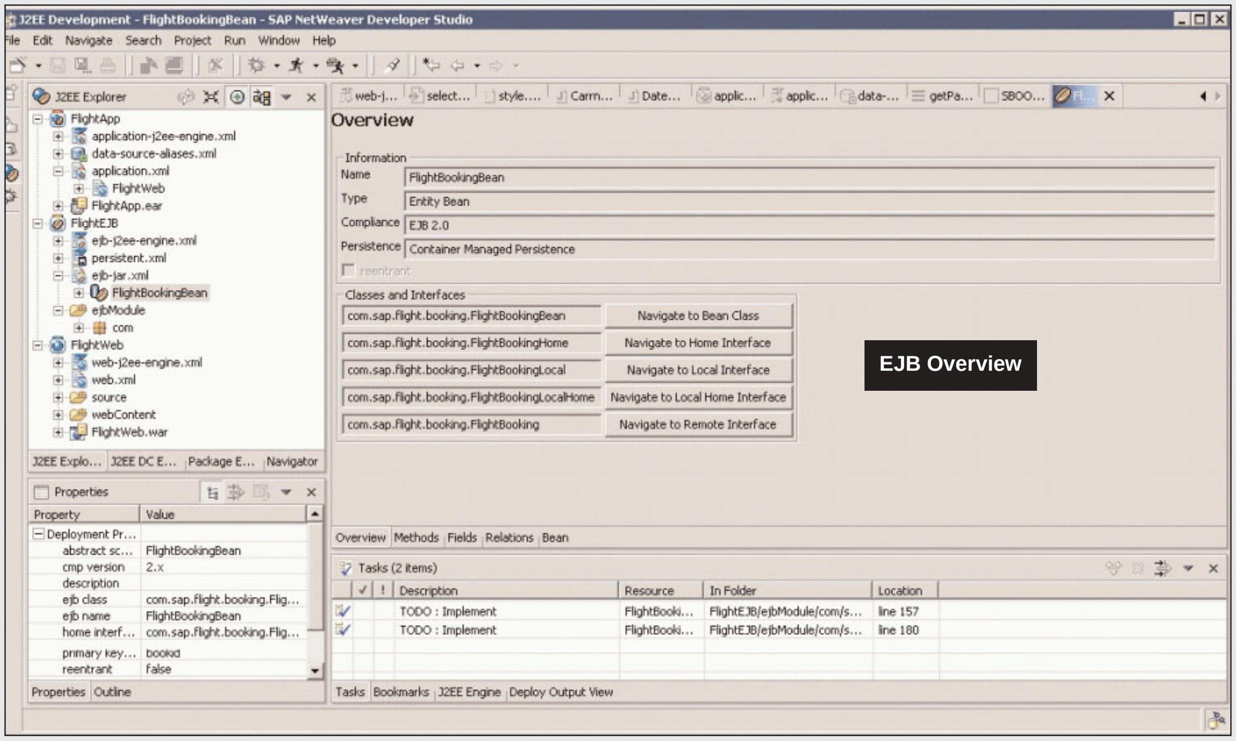
When choosing CMP, the screen requires you to specify all of the fields you want to persist in the underlying database when it is saved by the container. For demonstration purposes, specify all of the fields in the booking table, SBOOK_CMP (refer back to the fields listed for table SBOOK_BA in Figure 5).

✓ *Note!*

Choose the Java type java.lang.String for VARCHAR columns and java.sql.Date for DATE columns. Use java.math.BigDecimal for numeric fields. The persistent field names need not be the same as the database columns. Choose similar names, but use syntactical conventions typical for Java. Consider using the following names: carrid, connid, fldate, bookid, custtype, fclass, smoker, loccuram, loccurkey, orderDate, cancelled, reserved, passname, forename, passform, passBirth.

[21] For more on Java persistence, see the article "A Guided Tour of the SAP Java Persistence Framework — Achieving Scalable Persistence for Your Java Applications" on page 111 of this issue.

Figure 25 *Detail Screen of the FlightBookingBean Entity Bean*


J2EE Development - FlightBookingBean - SAP NetWeaver Developer Studio
File Edit Navigate Search Project Run Window Help
J2EE Explorer
FlightApp
application-j2ee-engine.xml
data-source-aliases.xml
application.xml
FlightWeb
FlightApp.ear
FlightEJB
ejb-j2ee-engine.xml
persistent.xml
ejb-jar.xml
FlightBookingBean
ejbModule
com
FlightWeb
web-j2ee-engine.xml
web.xml
source
webContent
FlightWeb.war
J2EE Explo... J2EE DC E... Package E... Navigator
web-j... select... style.... Carrn... Date... applic... applic... data-... getPa... SBOO... Fl...
Overview
Navigate to Bean Class
Navigate to Home Interface
Navigate to Local Interface
Navigate to Local Home Interface
Navigate to Remote Interface
EJB Overview
Overview Methods Fields Relations Bean
Properties Outline
Tasks Bookmarks J2EE Engine Deploy Output View

Information

Name	FlightBookingBean
Type	Entity Bean
Compliance	EJB 2.0
Persistence	Container Managed Persistence

☐ reentrant

Classes and Interfaces

com.sap.flight.booking.FlightBookingBean
com.sap.flight.booking.FlightBookingHome
com.sap.flight.booking.FlightBookingLocal
com.sap.flight.booking.FlightBookingLocalHome
com.sap.flight.booking.FlightBooking

Properties

Property	Value
Deployment Pr...	
abstract sc...	FlightBookingBean
cmp version	2.x
description	
ejb class	com.sap.flight.booking.Flig...
ejb name	FlightBookingBean
home interf...	com.sap.flight.booking.Flig...
primary key...	bookid
reentrant	false

Tasks (2 items)

✓	!	Description	Resource	In Folder	Location
		TODO : Implement	FlightBooki...	FlightEJB/ejbModule/com/s...	line 157
		TODO : Implement	FlightBooki...	FlightEJB/ejbModule/com/s...	line 180

SBOOK_CMP. Your screen should look similar to Figure 24 when you're done.

9. Highlight the *PrimaryKey* item (the bottom-most item in Figure 24) and select field *bookid* from the dropdown list that will appear in the right pane.

10. Press *Next*.

11. A pop-up appears that specifies method names and their parameters. The screen is similar to the persistence settings screen.

12. Select the method *ejbCreate*.

13. Choose *Add* in the right pane to add new parameters. You must add all persistent fields and their types as parameters for this method.

14. Press *Finish*.

The EJB creation wizard should add an item called *FlightBookingBean* to your project, and a file called *persistent.xml*, which we'll soon use to bind the bean to table SBOOK_CMP in our database.

Step 4: Add Code to Method ejbCreate to Save Its Data to the Database

Before the entity bean will work properly, we need to add a small amount of code to the bean's constructor. Follow these steps:

1. Go to the J2EE Explorer view and expand the *ejb-jar.xml* folder (see **Figure 25**).

2. Double-click on *FlightBookingBean*, which will open an overview of the EJB in the upper right pane.

3. Click on the *Navigate to Bean Class* button to navigate to the bean's source code.

4. Scroll down to the implementation of the *ejbCreate* method.

5. Place the following source code within method *ejbCreate*:

```
setBookid(bookid);
setCarrid(carrid);
setConnid(connid);
setFldate(fldate);
setCusttype(custtype);
setFclass(fclass);
setLoccuram(loccuram);
setLoccurkey(loccurkey);
setOrderDate(orderDate);
setCancelled(cancelled);
setReserved(reserved);
setSmoker(smoker);
setPassname(passname);
setForename(forename);
setPassform(passform);
setPassBirth(passBirth);
```

6. Save the project.

This code passes the parameters to the EJB container, which creates a corresponding database record *automatically* when the method completes in the container.

✓ ***Note!***

If we had chosen bean-managed persistence, this automatic function would not be available — we would have to code the database insert ourselves.

Also notice how the code uses predefined *set* methods to modify attribute values rather than setting them directly with an assignment operator (equals sign). This illustrates one of two important rules of entity beans — entity attributes can only be changed using the bean's *set* methods (not with an assignment operator), and attributes can only be read using the bean's *get* methods (not with dot notation). This rule helps ensure data integrity, since important conversions and validations are often coded into these *set* and *get* methods.

Step 5: Map the Entity Bean to Its Data Source

There's one more thing to do to get the entity bean to work — the EJB container needs to know the datastore to use, and the rules for mapping classes to tables and fields to columns. We accomplish this via a simple mapping procedure:

1. Double-click on the descriptor *persistent.xml* in the J2EE Explorer view.

2. Select the data source *FLUGDATEN*.

3. Navigate to the *Entity Beans* tab in the upper right pane.

4. Select the table *SBOOK_CMP* in the *Table name* field (see **Figure 26**).

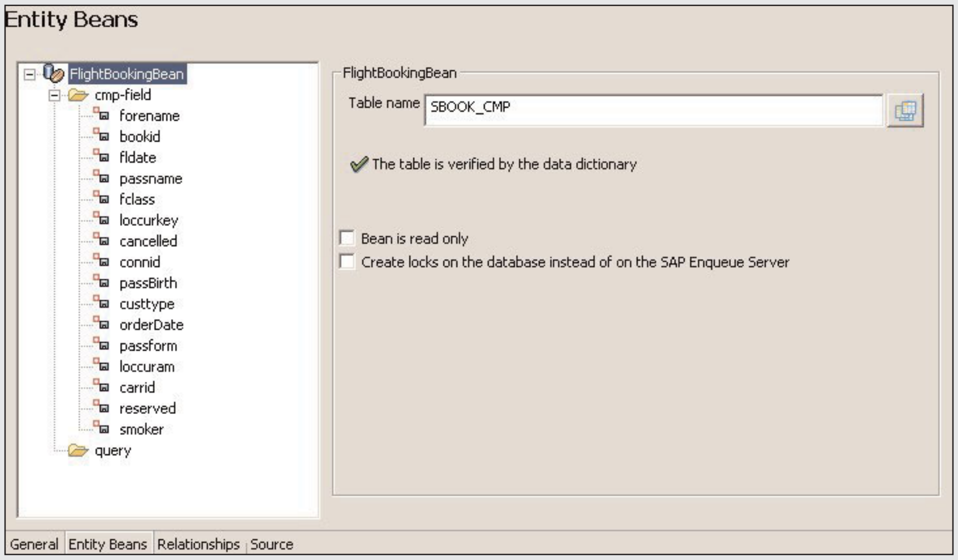
5. The system automatically maps the CMP fields to the underlying database columns based on their order. A success or error message will appear on the right-hand side (a success message appears next to the check mark in Figure 26).

This completes the implementation of the entity bean. The mapping causes the EJB container to handle the database access automatically. When the application tries to create a new booking, it will be inserted in the SBOOK_CMP table. If the application tries to find a booking, it will select a record from the table.

Step 6: Create a Session Bean for the Flight Booking Business Logic

Creating a session bean will be easy now that you're familiar with the EJB concept. You use the same

Figure 26 *Map the Entity Bean's Attributes to Table Fields*


Entity Beans

FlightBookingBean
- cmp-field
  - forename
  - bookid
  - fldate
  - passname
  - fclass
  - loccurkey
  - cancelled
  - connid
  - passBirth
  - custtype
  - orderDate
  - passform
  - loccuram
  - carrid
  - reserved
  - smoker
- query

FlightBookingBean

Table name: SBOOK_CMP

✓ The table is verified by the data dictionary

☐ Bean is read only
☐ Create locks on the database instead of on the SAP Enqueue Server

wizard used for creating entity beans. The only difference is the type of bean you choose on the first screen:

1. Highlight the EJB project name (*FlightEJB*) in the J2EE Explorer view.

2. Right-click and select *New* → *EJB* from the context menu.

3. Enter *FlightBookingProcessorBean* as the name of the bean.

4. Select *Stateless Session Bean* as the bean type.

> ✓ **Note!**
>
> *Since our application can perform all of its processing — i.e., create the reservation — in one request-response cycle, we don't need to retain the session bean in memory (see the sidebar on page 27 for more on stateless vs. stateful session beans).*

5. Enter *com.sap.flight.bookingprocessor* as the package name. Using a different package name will let us see how to import classes from other packages later.

6. Press *Finish*.

The bean's class and interfaces are generated as before with the entity bean.

Step 7: Add a Business Method to the Session Bean

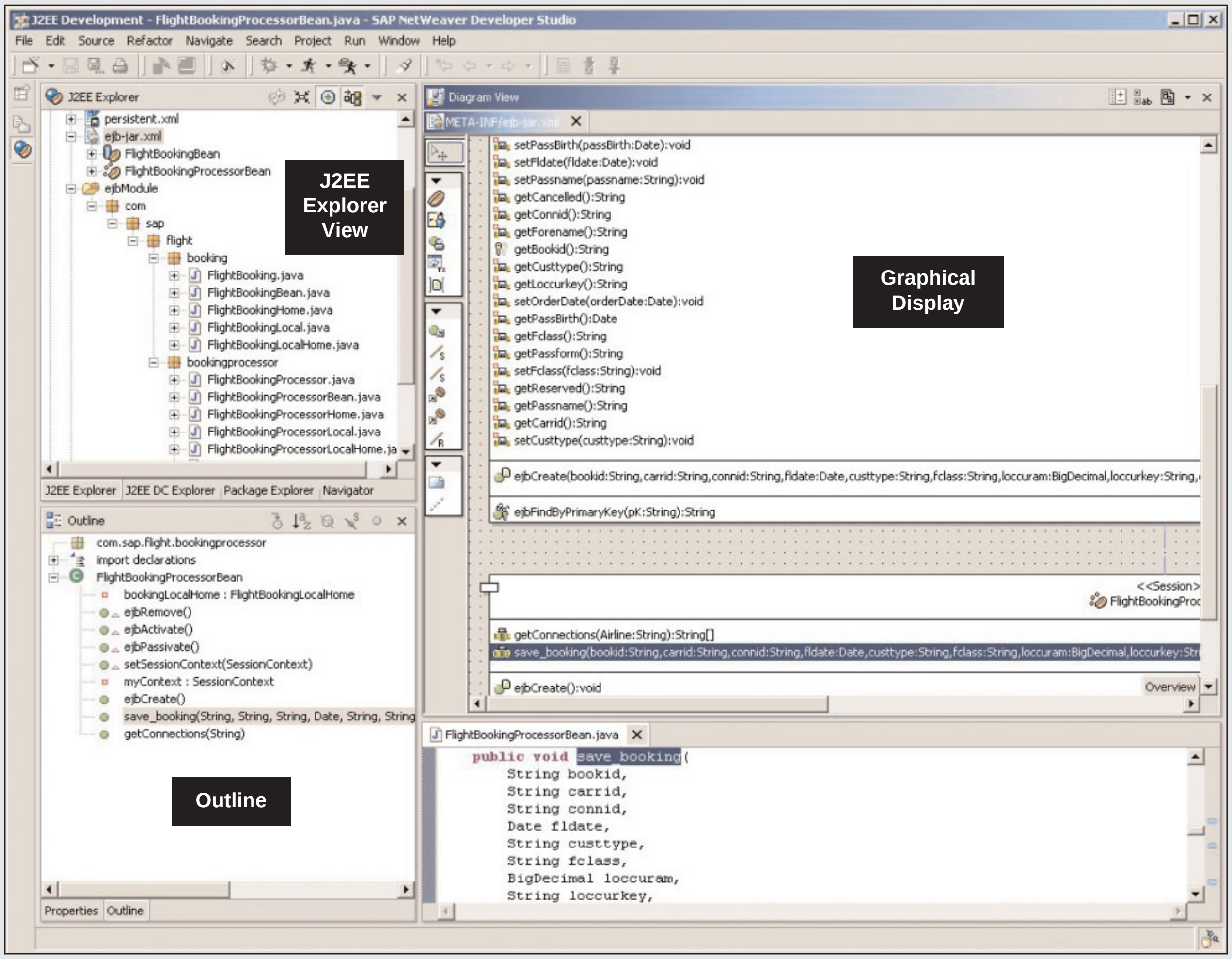
A special perspective in SAP NetWeaver Developer Studio called the J2EE Developer perspective contains a tool that lets you graphically create and modify J2EE artifacts.[22] You now can add business methods and attributes freely from within the diagram. Let's use this tool to add a method called *save_booking* to the session bean:

1. Highlight your EJB project again in the J2EE Explorer view.

2. Right-click and select *Open EJB module diagram* from the context menu.

[22] Artifacts are JavaBeans, JSPs, servlets, session beans, entity beans, etc.

Figure 27 *Graphical Display of the EJB Attributes and Methods*


J2EE Development - FlightBookingProcessorBean.java - SAP NetWeaver Developer Studio
File Edit Source Refactor Navigate Search Project Run Window Help
J2EE Explorer
J2EE Explorer View
Graphical Display
Outline
Diagram View
META-INF/ejb-jar.xml
J2EE Explorer
J2EE DC Explorer
Package Explorer
Navigator
Properties
Outline
Overview
FlightBookingProcessorBean.java

```
persistent.xml
ejb-jar.xml
  FlightBookingBean
  FlightBookingProcessorBean
ejbModule
  com
    sap
      flight
        booking
          FlightBooking.java
          FlightBookingBean.java
          FlightBookingHome.java
          FlightBookingLocal.java
          FlightBookingLocalHome.java
        bookingprocessor
          FlightBookingProcessor.java
          FlightBookingProcessorBean.java
          FlightBookingProcessorHome.java
          FlightBookingProcessorLocal.java
          FlightBookingProcessorLocalHome.ja
```

```
com.sap.flight.bookingprocessor
import declarations
FlightBookingProcessorBean
  bookingLocalHome : FlightBookingLocalHome
  ejbRemove()
  ejbActivate()
  ejbPassivate()
  setSessionContext(SessionContext)
  myContext : SessionContext
  ejbCreate()
  save_booking(String, String, String, Date, String, String
  getConnections(String)
```

```
setPassBirth(passBirth:Date):void
setFldate(fldate:Date):void
setPassname(passname:String):void
getCancelled():String
getConnid():String
getForename():String
getBookid():String
getCusttype():String
getLoccurkey():String
setOrderDate(orderDate:Date):void
getPassBirth():Date
getFclass():String
getPassform():String
setFclass(fclass:String):void
getReserved():String
getPassname():String
getCarrid():String
setCusttype(custtype:String):void
ejbCreate(bookid:String,carrid:String,connid:String,fldate:Date,custtype:String,fclass:String,loccuram:BigDecimal,loccurkey:String,
ejbFindByPrimaryKey(pK:String):String

<<Session>>
FlightBookingProc
getConnections(Airline:String):String[]
save_booking(bookid:String,carrid:String,connid:String,fldate:Date,custtype:String,fclass:String,loccuram:BigDecimal,loccurkey:Str
ejbCreate():void
```

```
public void save_booking(
    String bookid,
    String carrid,
    String connid,
    Date fldate,
    String custtype,
    String fclass,
    BigDecimal loccuram,
    String loccurkey,
```

3. **Figure 27** shows the graphical display.

4. Highlight the session bean (*FlightBookingProcessorBean*) in the J2EE Explorer view.

5. Right-click and select *New* → *Business Method* from the context menu.

6. Create a new business method called *save_booking*. Your new method will also appear in the bean's class outline (lower left pane in Figure 27) as well as in the J2EE Explorer view (upper left pane in Figure 27).

Before we can add code to the method, we need to define parameters we'll require a calling program to pass in. We did this for the entity bean earlier by editing the bean's class. For session beans, it's the same procedure:

1. Bring up the visual parameter maintenance tool for the method by double-clicking on the business method's name under the *ejb-jar.xml* file in the J2EE Explorer view (select *ejb-jar.xml* → *FlightBookingProcessorBean* → *save_booking*).

2. Add the following parameters as before with the

Figure 28 Initialize the Entity Bean's Interface in the Session Bean's ejbCreate Method

```
try {
          Context ctx = new InitialContext();
          bookingLocalHome =
                (FlightBookingLocalHome) ctx.lookup(
                   "java:comp/env/ejb/FlightBookingBean");

   } catch (NamingException e) {
                throw new CreateException(e.getMessage());
```

entity bean: *bookid*, *carrid*, *connid*, *fldate*, *custtype*, *fclass*, *loccuram*, *loccurkey*, *orderDate*, *cancelled*, *reserved*, *smoker*, *passname*, *forename*, *passform*, and *passBirth*. Remember to specify Java type *java.lang.String* for VARCHAR columns, *java.sql.Date* for DATE columns, and *java.math.BigDecimal* for numeric fields.

Our session bean now has one business method, which we'll call later from the booking JSP page.

Step 8: Add Code to Instantiate the Entity Bean

The first thing we need to do is create a reference to the *FlightBooking* entity bean's class for use within the session bean. To do this, return to the source code for the class by double-clicking on the session bean in the J2EE Explorer view and selecting the *bean* tab. Declare an instance variable in the session bean's class called *bookingLocalHome*, of type *FlightBookingLocalHome*, as shown below:

```
private FlightBookingLocalHome
  bookingLocalHome;
```

Notice how EJBs, as mentioned earlier, are always referenced using one of their interfaces instead of directly using their class name. In this case, we're using the object's "local home" interface, since it will run in the same container as the session bean.

Next, since we defined the entity bean as belonging to a different package than the session bean,[23] Java requires an *import* statement at the top of the session bean class, so it can resolve the reference to *FlightBookingLocalHome*. The good news is you can have SAP NetWeaver Developer Studio maintain all of the import declarations for you by right-clicking anywhere in the session bean's source editor and selecting *Source* → *Organize Imports* from the context menu. This is a standard Eclipse feature.

Now we're ready to add source code, in two parts. First we need to initialize the entity bean's interface in the session bean's standard *ejbCreate* method.[24] Navigate to this method within the class's source code and add the code shown in **Figure 28**.

The code in Figure 28 uses JNDI to obtain an interface reference to the entity bean's local home interface, which gets stored in *bookingLocalHome*. It does not instantiate an actual entity bean. If the EJB class isn't found — e.g., because it wasn't deployed correctly — an exception is thrown.

Before continuing, select the *Organize Imports* feature again, which automatically scans the code for references to classes outside the current package, and inserts the needed import statements into your code. This time it displays a list to choose from. Select the *javax.naming.Context* interface, which is needed to use the *Context* object in the code in Figure 28.

[23] Our entity bean class belongs to the *com.sap.flight.booking* package, while our session bean's package is *com.sap.flight.bookingprocessor*.

[24] This is a standard method added automatically to all EJBs by SAP NetWeaver Developer Studio, according to the J2EE standard.

Figure 29 Create a New Entity Bean Instance Using Its Local Home Interface

```
try {
     bookingLocalHome.create(
                       bookid,
                       carrid,
                       connid,
                       fldate,
                       custtype,
                       fclass,
                       loccuram,
                       loccurkey,
                       orderDate,
                       cancelled,
                       reserved,
                       smoker,
                       passname,
                       forename,
                       passform,
                   passBirth);
     } catch (CreateException e) { }
```

Finally, locate and add the code in **Figure 29** to the *save_booking* method.

✓ ***Note!***

The database commit is implicit, occurring automatically when the ejbCreate method completes. You can also span transaction boundaries over several method invocations if you specify transaction attributes accordingly. Rollbacks are performed automatically when an exception occurs, and can also be executed programmatically. For specifics on transaction handling in J2EE applications, visit Sun's developer site at ***http://java.sun.com/j2ee****.*

The code in Figure 29 uses the entity bean's local home interface to create a new entity bean instance, which is the standard way to instantiate an EJB. The J2EE container calls the entity bean's *ejbCreate* method. Recall that the entity bean's *ejbCreate* method then passes this data to its container, which saves a record immediately to the database.

Step 9: Declare the Entity Bean's Symbolic Name for JNDI

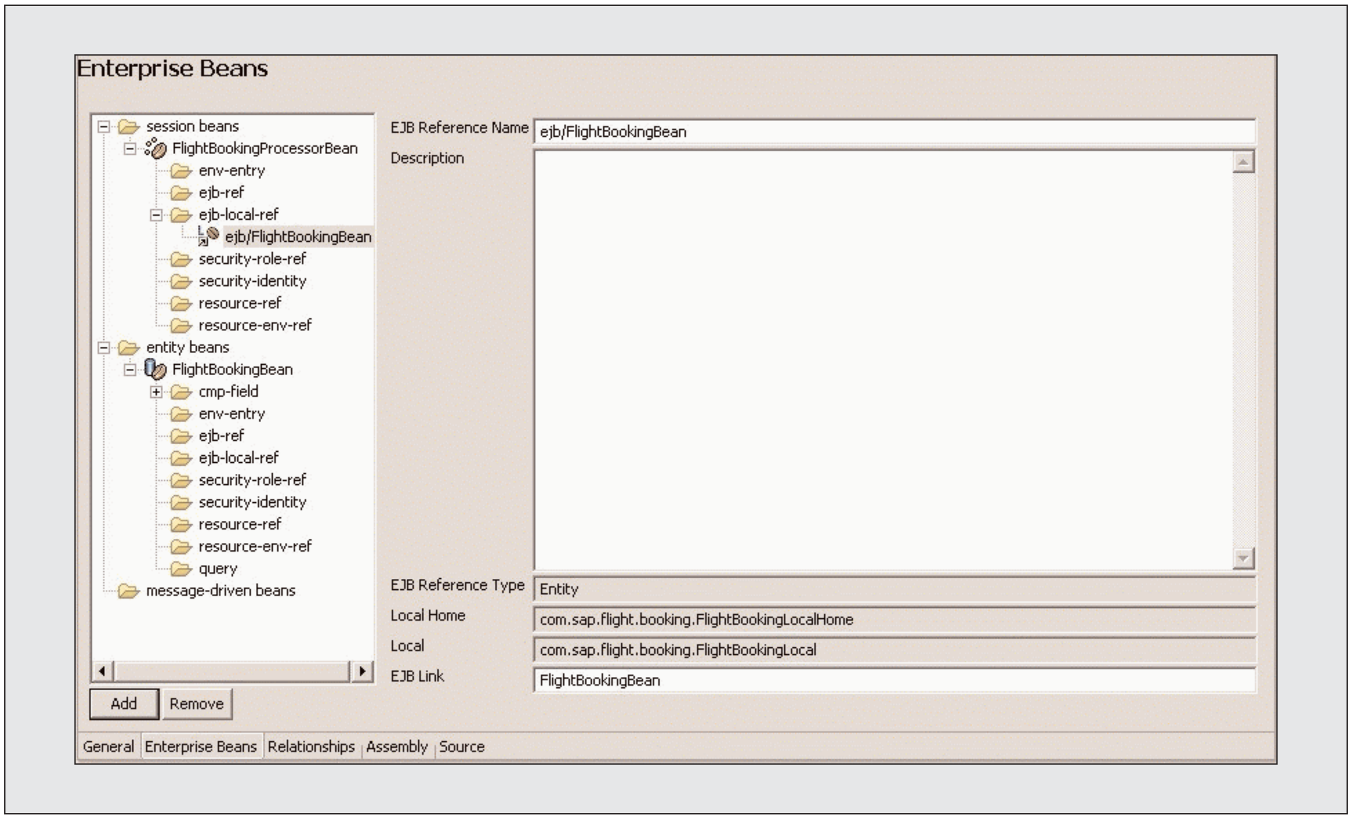
Finally, we need to tell the system how to map the symbolic name *ejb/FlightBookingBean* used in the session bean's *ejbCreate* method in Figure 28 to the actual *FlightBookingBean* class. In SAP NetWeaver Developer Studio parlance, this is called "creating a local reference." If you skip this step, you'll receive errors like "unresolved symbol" when trying to compile and run the application.

Here's how to create the local reference:

1. In the J2EE Explorer view, double-click on the *ejb-jar.xml* descriptor file.
2. Select the *Enterprise Beans* tab (**Figure 30**).
3. Expand the *session beans* folder.
4. Expand the *FlightBookingProcessorBean* session bean.
5. Mark the *ejb-local-ref* subfolder.
6. Press *Add*.
7. In the pop-up, choose the *FlightBookingBean* entity bean.
8. Save your work.

That's it! The entity and session beans are complete. Unfortunately, as with regular JavaBeans, there's no way to test the beans in isolation, since they're not independently executable — this requires special test software. Instead, we'll test the beans as part of the integrated, executable JSP application.

Figure 30 *Create a Local Reference*



EJB Reference Name: ejb/FlightBookingBean
Description:
EJB Reference Type: Entity
Local Home: com.sap.flight.booking.FlightBookingLocalHome
Local: com.sap.flight.booking.FlightBookingLocal
EJB Link: FlightBookingBean

Leveraging the EJBs from the JSP Pages

Just as we defined a local reference in the session bean code to refer to the entity bean, we need to define a local reference in the JSP pages to refer to the session bean. Follow these steps:

1. Expand the *FlightWeb* Web project in the J2EE Explorer view.

2. Right-click and select *properties* from the context menu.

3. A dialog will appear asking you which project contains the objects you want to reference. Choose *FlightEJB*, since it contains the session bean.

   At this point, we've told SAP NetWeaver Developer Studio that we want to interlink the two projects. If you forget this step and make changes to the EJB project, the Web project will not be recompiled automatically, which causes inconsistencies.

Next, we need to create the actual reference to the session bean:

1. Double-click on the *web.xml* descriptor file. This is analogous to the *ejb-jar.xml* file in the EJB project.

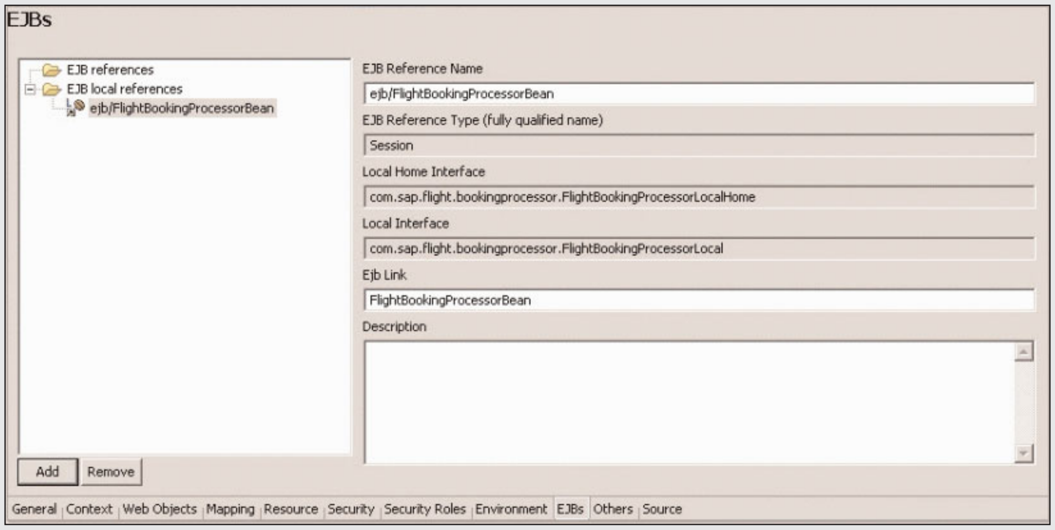
2. Select the *EJBs* tab.

3. Mark the *EJB local references* folder.

4. Press *Add*.

5. Select the *FlightBookingProcessorBean* session bean from the dialog that appears.

Figure 31 Information About the Established EJB Reference to the Session Bean


EJBs
EJB references
EJB local references
ejb/FlightBookingProcessorBean
Add
Remove
General | Context | Web Objects | Mapping | Resource | Security | Security Roles | Environment | EJBs | Others | Source

EJB Reference Name: ejb/FlightBookingProcessorBean
EJB Reference Type (fully qualified name): Session
Local Home Interface: com.sap.flight.bookingprocessor.FlightBookingProcessorLocalHome
Local Interface: com.sap.flight.bookingprocessor.FlightBookingProcessorLocal
Ejb Link: FlightBookingProcessorBean
Description:

6. The upper right pane of the SAP NetWeaver Developer Studio screen will look like **Figure 31**.

With the references in place, we're finally ready to add the booking JSP page to the project. Better yet, let's jump ahead and add all of the remaining files needed to make the demo application fully functional (e.g., the list results page, a confirmation page, etc.). You'll find the following files in the ZIP file available at **www.SAPpro.com**: *Buchen.java*, *Liste.java*, *Overall.java*, *Speichern.java*, *buchen.jsp*, *confirm.jsp*, and *liste.jsp*. Download and add each of these files to the *FlightWeb* project. To add them to the project, use Windows Explorer to copy and paste the .java files to the *source* folder, and the .jsp pages to the *webContent* folder. Select the *Navigator* tab in the J2EE perspective to paste the files into the Web project.

While the code for each page is too extensive to list and explain line by line, the most important piece is method *saveBook* in the booking JavaBean, shown in **Figure 32**. The code is divided into two parts. The first demonstrates how to save the booking data directly to the database, as per our initial non-EJB approach. It uses table SBOOK_BA according to our original specification. The second part shows how to instantiate and pass data to the session bean called *FlightBookingProcessorBean*. This bean in turn instantiates the entity bean *FlightBookingBean*, which then creates the database records in method *ejbCreate*. Recall that we mapped the entity bean to table SBOOK_CMP, so when you run the application and book a flight, you should see it appear identically in both tables.

Building, Deploying, and Debugging the EJB-Enhanced Sample Application

Deploying a multi-project EJB application is similar to deploying a simple JSP application: all of the components get bundled by the enterprise application project into a single EJB Java archive (JAR) file that gets deployed to the J2EE server.

First, choose *Build EJB JAR file* from the context menu of the EJB project. This compiles the EJB components into a JAR file that SAP NetWeaver Developer Studio needs in order to add the EJBs to the final EAR file.

Figure 32 Code Required to Save a Booking (Reservation) to the Database

```
public void saveBook() throws Exception {

/* Begin code that inserts the booking directly into the    */
/* database. This is the regular, non-EJB approach.         */
         params = new String[15];
         PreparedStatement ps = conn.prepareStatement("Insert into sbook_ba (carrid, " +
     "connid, fldate, bookid, passform, passname, forename, passbirth, custtype, " +
     "fclass, smoker, reserved, order_date, loccuram, loccurkey) " +
     "values (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ? )");
        SimpleDateFormat sdfInput = new SimpleDateFormat("yyyy-MM-dd");
         java.sql.Date d_bdat = new
java.sql.Date(bdat.getYear(),bdat.getMonth(),bdat.getDate());
         java.sql.Date d_birth = new
java.sql.Date(passbirth.getYear(),passbirth.getMonth(),passbirth.getDate());
         BigDecimal d_preis = new BigDecimal(preis);
         java.util.Date hv_date = sdfInput.parse( fldate );
           d_fldate = new java.sql.Date(hv_date.getYear(), hv_date.getMonth(),
hv_date.getDate());
     ps.setString(1, carrid);
     ps.setString(2, connid);
     ps.setDate(3, d_fldate);
     ps.setString(4, bookid);
     ps.setString(5, passform);
     ps.setString(6, passname);
     ps.setString(7, forename);
     ps.setDate(8, d_birth);
     ps.setString(9, custtype);
     ps.setString(10, fclass);
     ps.setString(11, smoker);
     ps.setString(12, reserved);
     ps.setDate(13, d_bdat);
     ps.setBigDecimal(14, d_preis);
     ps.setString(15, curr);
     ps.executeUpdate();

/* Begin code that passes the booking data to our session EJB, */
/* FlightBookingProcessorBean), which in turn instantiates an  */
/* entity EJB that saves the booking record to the database   */

     try {

               Context ctx = new InitialContext();

               FlightBookingProcessorLocalHome bookingProcessorHome =

                     (FlightBookingProcessorLocalHome) ctx.lookup(

                            "java:comp/env/ejb/FlightBookingProcessorBean");

            FlightBookingProcessorLocal bookingProcessor = bookingProcessorHome.create();
            bookingProcessor.save_booking(bookid,
                   carrid,
                   connid,
                   d_fldate,
                   custtype,
                   fclass,
```

(continued on next page)

Figure 32 (continued)

```
                    d_preis,
                    curr,
                    d_bdat,
                    smoker,
                    reserved,
                    passform,
                    passname,
                    forename,
                    d_birth
                    );


          } catch (CreateException e) {

                 e.printStackTrace();}

      // end EJB stuff
      updateSeats(carrid, connid, fclass);
```

Next, we need to update the enterprise application project to include the EJB project. Mark the *FlightApp* project and choose *Add modules* from the context menu. A pop-up appears with available projects. Choose the *FlightEJB* project.

Finally, rebuild and redeploy the EAR file and test-drive the application again. Debugging the EJB-based application is the same as before: just activate the debugger, set a breakpoint, and run the application. When the debugger encounters a breakpoint while running a server-side component, the debugger window will pop up on your PC.

Summary

SAP NetWeaver Developer Studio is a powerful tool for building Web-based enterprise applications using open technologies including J2EE 1.3, Web Dynpro, and Web services. This article has shown you how easy SAP NetWeaver Developer Studio makes it to develop, debug, and deploy enterprise applications using any combination of the JavaServer Pages (JSP), JavaBeans, and Enterprise JavaBeans (EJB) technologies.

With its broad collection of wizards, editors, and visual tools, and its SAP integration capabilities, SAP NetWeaver Developer Studio greatly simplifies Java development for the business programmer — you only need to learn one tool to leverage any number of technologies. The J2EE toolset is a good starting point for learning how to use SAP NetWeaver Developer Studio, which takes full advantage of the underlying Eclipse open source platform. My hope is that this article will help you use the tool to ease your own application development, without sacrificing the advantages offered by newer technologies.

Karl Kessler studied computer science at the Technical University in Munich. He joined SAP in 1992. In 1994, Karl became a member of the ABAP Workbench product management group, and in 1996, he became product manager for business programming languages and frameworks. In 2003, Karl assumed responsibility for product management of the SAP NetWeaver platform, including SAP Web Application Server, SAP NetWeaver Developer Studio, SAP Enterprise Portal, Web services, J2EE, and ABAP. He can be reached at karl.kessler@sap.com.

Appendix: Source Code for the Example Application’s Selection Page and Cascading Style Sheet

Figure A Source Code for the Static HTML Portion of the Example Application’s Selection Page

```
<%@ page language = "java" %>

<HTML>
<HEAD>
<TITLE> Flight Booking – Selection Screen </TITLE>
</HEAD>
<BODY BGCOLOR="#FFFFFF">
<h2 align="center"> Welcome at Flight Booking</H2>
<img src="Flugzeug.gif" align="left" width="200">
<img src="Flugzeug_r.gif" align="right" width="200">

<form name="auswahl" action="list.jsp">
<table align="center">
<tr>
<th class="highlight">Selection Criteria for Flights</TH>
</tr>
<TR>
<td class="light">
   <TABLE>
    <tr><td>Flight Carrier</TD>
        <td><select name="carrid" size="1">
        <option value="LH" /> Lufthansa </option>
        <option value="UA" /> United Airlines </option>
            </select>
        </TD>
    </tr>
    <tr><td>Flight Number</TD>
        <td><select name="connid" size="1">

            <option> 3100 </option>
      <option> 3200 </option>
            </select>
        </TD>
    </tr>
    <tr>
    <td><label for="von">Flight date from:</label></td>
    <td><input type="text" id="von" name="von" >
```

(continued on next page)

Figure A (continued)

```
         <label for="bis">until</LABEL>
         <input type="text" id="bis" name="bis" >
     </td>
     </tr>
     </table>



</td>
</tr>
</table>
<br>
  <center><input type="submit" class="button" value=" Display Flights "></center>
<br>
</form>
</BODY>

</HTML>

</jsp:useBean>
```

Figure B The Cascading Style Sheet Used for the Example Application's JSP Pages

```
/* DATEI: style.css */
/* Stylesheets for JSP pages */
h1,h2,h3,h4,p,ul,ol,li,div,td,th,address,blockquote,nobr,b,i {
 font-family:Arial,sans-serif; }

h2 { font-size:16px; margin-bottom:18px; color:#70A0E0 }

p,ul,ol,li,div,td,th,address,blockquote { font-size:13px; }

.highlight { background-color:#B1C9E2; font-style:bold; text-align:left; }

.light { background-color:#EFF6FB; padding:5px; border:solid 1px #B1C9E2; }

.blue { color:#70A0E0; }

td { padding:7px; }

td.text { padding:0px; }

input.button { background-color:#C1D3E0; border-left:0px; border-top:0px;
               border-right:1px solid #70A0E0; border-bottom:1px solid #70A0E0 }

.error { color:red; font-style:bold }
```