Handling Transactions with BAPIs in Web Dynpro

SAP NetWeaver 04
Copyright

© Copyright 2004 SAP AG. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or for any purpose without the express permission of SAP AG. The information contained herein may be changed without prior notice.

Some software products marketed by SAP AG and its distributors contain proprietary software components of other software vendors.

Microsoft, Windows, Outlook, and PowerPoint are registered trademarks of Microsoft Corporation.

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

Oracle is a registered trademark of Oracle Corporation.

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

Citrix, ICA, Program Neighborhood, MetaFrame, WinFrame, VideoFrame, and MultiWin are trademarks or registered trademarks of Citrix Systems, Inc.

HTML, XML, XHTML and W3C are trademarks or registered trademarks of W3C®, World Wide Web Consortium, Massachusetts Institute of Technology.

Java is a registered trademark of Sun Microsystems, Inc.

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

MaxDB is a trademark of MySQL AB, Sweden.

SAP, R/3, mySAP, mySAP.com, xApps, xApp, SAP NetWeaver, and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP AG in Germany and in several other countries all over the world. All other product and service names mentioned are the trademarks of their respective companies. Data contained in this document serves informational purposes only. National product specifications may vary.

These materials are subject to change without notice. These materials are provided by SAP AG and its affiliated companies ("SAP Group") for informational purposes only, without representation or warranty of any kind, and SAP Group shall not be liable for errors or omissions with respect to the materials. The only warranties for SAP Group products and services are those that are set forth in the express warranty statements accompanying such products and services, if any. Nothing herein should be construed as constituting an additional warranty.
Icons in Body Text

<table>
<thead>
<tr>
<th>Icon</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Caution Icon]</td>
<td>Caution</td>
</tr>
<tr>
<td>![Example Icon]</td>
<td>Example</td>
</tr>
<tr>
<td>![Note Icon]</td>
<td>Note</td>
</tr>
<tr>
<td>![Recommendation Icon]</td>
<td>Recommendation</td>
</tr>
<tr>
<td>![Syntax Icon]</td>
<td>Syntax</td>
</tr>
</tbody>
</table>

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

Typographic Conventions

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

Task

The following tutorial demonstrates the basic steps needed to design, implement, deploy, and run a Web Dynpro application that handles transactions with BAPIs (Business Application Programming Interfaces).

In your sample application, you will carry out your remote calls to the backend ABAP system using an Adaptive RFC model. To write data into the table in the ABAP system, you will make use of existing functions in the form of writing BAPIs that are implemented as RFC function modules. For each function module that you need in your Web Dynpro application, the corresponding Java proxy classes will be generated. All the generated proxy classes and interfaces are bundled together with the corresponding Dictionary information from the ABAP system in the RFC model and are treated as part of your Web Dynpro project.

For the purpose of this tutorial, you will design a simply structured Web application, which will check flight availability and book or reserve a flight. The user interface for this Web application will consist of one view in which the user will be able to enter the flight booking request and trigger the booking in the ABAP system. First the application will check if the flight connection is available (1). The user can then reserve, confirm, or cancel the flight booking via a confirmation dialog that appears (2). As a result of the flight booking or reservation, the flight confirmation data will be written to the back-end system. Some of the data retrieved is also displayed in the result area of your Web Dynpro application (3).
**Transaction Model of the Sample Application**

Since the Web Dynpro example application is to be used to write data to the database of the ABAP back-end system, it is essential that the programming on the Web Dynpro side (client) is transactional. Furthermore, the BAPI transaction model explicitly requires that the transaction control be enabled for the client. For a combination of BAPIs, the client can generally decide itself at which point it sets the **COMMIT WORK** or the **ROLLBACK WORK**.

In this case (see graphic), the transaction is already defined with a single BAPI call (`executeBAPI_FlightBooking()`). The call of this writing BAPI sets the start of the LUW (Logical Unit of Work). The transaction is ended by calling the special service BAPI `BapiService.TransactionCommit`.

---

**Objectives**

By the end of this tutorial, you will be able to:

- Create a model that is used to access the SAP back-end system from within the Web Dynpro project
- Know which Service BAPIs should be used to handle transactions in Web Dynpro applications
- Implement custom controllers for specific tasks that cannot be assigned to a single view, especially regarding transaction handling
- Create contexts for the custom controller and bind them to the model
- Create view contexts and map them to the custom controller context
- Implement RFC calls as a part of a transaction in order to write new data into the ABAP back-end system
Prerequisites

Systems, Installed Applications, and Authorizations

- The SAP NetWeaver Developer Studio is installed on your computer.
- You have access to the SAP J2EE Engine (Release 6.40).
- You have access to a remote SAP back-end system.

To test this example application successfully, you must also make sure that this SAP system contains the function modules `BAPI_FLIGHT_CHECKAVAILIBILITY`, `BAPI_FLBOOKING_CREATEFROMDATA`, `BAPI_TRANSACTION_COMMIT`, `BAPI_TRANSACTION_ROLLBACK` and the appropriate data from the flight data model. (This data is stored in the tables SFLIGHT and SBOOK.)

To access functions within an SAP system, a user must provide this system with valid credentials by means of the logon process. Since you are using the Web Dynpro Adaptive RFC Layer, the user ID defined in the Web Dynpro Content Administrator of your J2EE Engine will take care of the connection automatically.

- The SAP System Landscape Directory (SLD) and the SLD bridge are configured and running.

The SLD contains component information about all SAP software modules, the system landscape description and the name reservation service.

The SLD bridge is used to transform the system data to the correct format for the SLD server.

For more information, refer to the Post-Installation Guide for the SAP System Landscape Directory on SAP Web AS Java 6.40 or read the documentation on the SAP System Landscape Directory.

- In order to connect the logical systems defined in the Adaptive RFC model with a physical SAP system, you need to perform two configuration tasks:

**Setting Up the SLD Connection**

To establish a connection to the SLD server, you have to specify the HTTP connection parameters. You will carry out this task in the J2EE Engine Administrator while the J2EE Engine is running.

Proceed as follows:

1. Start the J2EE Engine if you have not already done so.
2. Start the engine administrator console using the path `<Drive>:\usr\sap\<System ID>\JC00\j2ee\admin\go.bat`.
3. Choose Connect and log on to the J2EE Engine. You need administrator rights for the login process.
4. After login, choose the Cluster tab on the left, expand the node Server, Services, and choose SLD Data Supplier.
5. Choose CIM Client Generation Settings and specify the necessary HTTP connection parameters to establish a connection to the SLD.
6. Save the data by choosing Save.
7. Choose the CIM Client Test button to check the CIM client connectivity.
8. If the test was successful, you can close the J2EE Engine Administrator Console. Otherwise, correct your connection parameters.

Now you have defined the connection parameters for the SLD server used to run your application.
Maintaining the JCO Destinations in the Web Dynpro Content Administrator

The logical system names used in the model declaration need to be associated with an actual SAP system defined in the SLD before this application can be executed. You will carry out this task after deploying your example application in the Web Dynpro Content Administrator.

If you do not perform this step, you will only be able to build and deploy the application, but you will be unable to run it.

The procedure is described in the last step of this tutorial.

Knowledge

☐ You have acquired some experience with Web Dynpro applications, for example by working through the Welcome tutorial (Creating Your First Web Dynpro Application) and the FlightList tutorial (Creating a Web Dynpro Application Accessing ABAP Functions).

☐ You have experience in Java programming.

☐ You are familiar with the use of Remote Function Modules and BAPIs.

Next step:
Importing a Project Template
Importing a Project Template

To restrict the development of this example application to the actual content covered, there is a predefined Web Dynpro project template available in the SAP Developer Network (SDN) [http://sdn.sap.com](http://sdn.sap.com) (Web Application Server area | Web Dynpro | Samples and Tutorials Quicklink).

In the remainder of this tutorial, you will gradually add to this initial project template until it is a complete Web Dynpro project, which is also available for separate download.

- **TutWD_FlightBooking_Init.zip**: Initial Web Dynpro project template
- **TutWD_FlightBooking.zip**: Complete Web Dynpro project with access to ABAP functions

Prerequisites

- You have access to the SAP Developer Network ([http://sdn.sap.com](http://sdn.sap.com)) with a user ID and password.
- The SAP NetWeaver Developer Studio is installed on your computer.

Procedure

Importing the Project Template into the SAP NetWeaver Developer Studio

1. Call the SAP NetWeaver Developer Network using the URL [http://sdn.sap.com](http://sdn.sap.com) and log on with your user ID and the corresponding password. If you do not have a user ID, you must register before you can log on.
2. Navigate to the Web Application Server area and then to the Samples and Tutorials section.
3. Download the ZIP file **TutWD_FlightBooking_Init.zip**, which contains the initial Web Dynpro project **TutWD_FlightBooking_Init** and save it in a local directory or directly in the work area of the SAP NetWeaver Developer Studio.
4. Unzip the contents of the ZIP file **TutWD_FlightBooking_Init.zip** into the work area of the SAP NetWeaver Developer Studio or into a local directory.
5. Call the SAP NetWeaver Developer Studio.
   a. Import the Web Dynpro project **TutWD_FlightBooking_Init**. To do this, choose File → Import.
   b. In the next window, choose Existing Project into Workspace and choose Next to confirm.
   c. Choose Browse, open the folder in which you saved the project **TutWD_FlightBooking_Init**, and select the project.
   d. Choose Finish to confirm.
6. The Web Dynpro project **TutWD_FlightBooking_Init** appears in the Web Dynpro Explorer for further processing and completion of the tutorial.

Initial Project Structure

After you have imported the Web Dynpro project template **TutWD_FlightBooking_Init**, the following project structure is displayed in the Web Dynpro Explorer:
Web Dynpro project: TutWD_FlightBooking_Init

Web Dynpro application: FlightBookingApp

The application FlightBookingApp displays the interface view of the Web Dynpro component FlightBookingComp in the browser window.

Web Dynpro component: FlightBookingComp

This is the Web Dynpro component that contains our entire application.

Custom controller: FlightBookingCust

We will extend this controller with implementation steps, which are not specific to a particular view.

View: FlightBookingView

In this view, the user can enter the flight request in the appropriate input fields and trigger the flight availability check and eventual flight reservation or booking.

On the result area, some flight confirmation data will be displayed.

Initially the project contains some UI elements, context elements, and an event-handler:

<table>
<thead>
<tr>
<th>Layout:</th>
<th>RootUIElementContainer [TransparentContainer = RootUIS]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GroupInput [Group - Child]</td>
</tr>
<tr>
<td></td>
<td>GroupInput_Header [Caption - Header]</td>
</tr>
<tr>
<td></td>
<td>Template_CONTAINER [TransparentContainer - Child]</td>
</tr>
<tr>
<td></td>
<td>HorizontalGutter [HorizontalGutter - Child]</td>
</tr>
<tr>
<td></td>
<td>Button [Button - Child]</td>
</tr>
<tr>
<td></td>
<td>GroupOutput [Group - Child]</td>
</tr>
<tr>
<td></td>
<td>GroupOutput_Header [Caption - Header]</td>
</tr>
</tbody>
</table>

Context:

- **Context**: Context
  - **UIElements**: GroupHeader, GroupVisible

Methods:

<table>
<thead>
<tr>
<th>Name</th>
<th>Return type</th>
</tr>
</thead>
<tbody>
<tr>
<td>onActionBook</td>
<td>void</td>
</tr>
<tr>
<td>onActionBookFlight</td>
<td>void</td>
</tr>
<tr>
<td>onActionCancel</td>
<td>void</td>
</tr>
<tr>
<td>onActionReserve</td>
<td>void</td>
</tr>
</tbody>
</table>

Window: FlightBookingWindow

Next Step:

Developing the Example Application – Steps
Developing the Example Application – Steps

You now need to execute some steps to extend the initial project template:

1. Implementing access to the back end and handling transactions
   a. Create a new model that is based on Adaptive RFC
   b. Create the context of the custom controller and the view controller
   c. Bind new model nodes in the custom controller context to the model
   d. Implement transaction handling within the custom controller

2. Complete the predefined UI layer
   a. Complete the view layout
   b. Implement the view controller

Next step:
Creating a Model
Creating an Adaptive RFC Model

You use a model to specify where the data for your Web Dynpro application is to come from. For the purpose of our sample application, you will import the following function modules from ABAP system:

- **BAPI_FLIGHT_CHECKAVAILIBILITY**: Checks if the flight connection is available in the system.
- **BAPI_FLBOOKING_CREATEFROMDATA**: Creates a new booking for a flight. A reservation can also be created (flight booking with the status **Reserved**).
- **BAPI_TRANSACTION_COMMIT**: This method executes a COMMIT WORK command in the ABAP system. It is required for transactions developed externally in the ABAP system that change data in the system via BAPI calls. When you call BAPIs in your client application that change data in the ABAP system, you must then call this method to write the changes to the database.
- **BAPI_TRANSACTION_ROLLBACK**: This method executes a ROLLBACK WORK command in the ABAP system. If you call BAPIs in your client application that change data, this method can prevent these changes being written to the database, provided that these changes have not already been passed to the database with a COMMIT WORK command.

For the sake of completeness we will also add the rollback service BAPI to the model, even though we do not make use of it in our sample scenario.

**Prerequisites**

- You can log on to the ABAP system.
- You have ensured that the function modules mentioned above exist in this system.

**Procedure**

1. In the project structure, expand the node **Web Dynpro → Models**.
2. Select **Models** and choose **Create Model** from the context menu. The corresponding wizard appears.
3. Choose the **Import Adaptive RFC Model** option, followed by **Next**.
4. Enter the model name **FlightBookingModel** and the package name **com.sap.tut.wd.flightbooking.model**.

When importing an Adaptive RFC model, you have to specify the logical system names for model instances and RFC metadata:

- Default logical system name for model instances: **WD_MODELDATA_DEST**
  Change it to **WD_FLIGHTBOOKING_MODELDATA_DEST**
- Default logical system name for RFC metadata: **WD_RFC_METADATA_DEST**
  Change it to **WD_FLIGHTBOOKING_RFC_METADATA_DEST**

The logical systems will later be configured separately using the JCO Connections screen of the J2EE Engine Web Dynpro Content Administrator (see **Building, Deploying, Configuring, and Running Your Application**).
5. Accept the suggested values and choose Next.

6. Enter the appropriate data for logging on to the SAP system and choose Next.

When logging on, you can choose one of two options: Either choose a single application server or address the system and log on using Load Balancing.

7. Enter the start of the names of the function modules BAPI_* in the appropriate field and choose Search.

8. Select the following function modules from the list that appears:
   - BAPI_FLBOOKING_CREATEFROMDATA
   - BAPI_FLIGHT_CHECKAVAILIBILITY
   - BAPI_TRANSACTION_COMMIT
   - BAPI_TRANSACTION_ROLLBACK
9. Choose Next.

This automatically triggers the generation process. The import process is logged in a detailed description, which you can see in the next dialog box.


**Result**

The Java proxies are generated and a new model node FlightBookingModel is inserted into project structure. In addition, a specific logical dictionary FlightBookingModel is also added to the project.
The newly created Adaptive RFC model \textit{FlightModel} can now be used in any component in the current project.

\textbf{Next step:}\n
Creating a Custom Controller Context and Binding it to the Model

\section*{Creating the Contexts of the Custom and View Controllers Using the Data Modeler}

The custom controller \textit{FlightBookingCust} is responsible for writing the flight booking to the SAP system, so it needs to be able to map the corresponding input and output to the flight model. To establish this correspondence between the custom controller and the model, you will create an appropriate controller context which will be bound to the model structure by using the Data Modeler. In this way, you can ensure that the model data is stored and manipulated in a central location.

To display and change the data which is stored in the back end, you also need to create an appropriate context in the \textit{FlightBookingView} which is mapped to the \textit{FlightBookingCust} context.
Procedure

Adding a Model to the Web Dynpro Component

1. In the project structure, expand the tree up to the node Web Dynpro → Web Dynpro Components → FlightBookingComp.
2. Open the context menu of FlightBookingComp and choose Open Data Modeler.

The Data Modeler is opened for FlightBookingComp.

3. In the left toolbar, select the icon and click into the field Used Models.
4. In the list that appears, select the model FlightBookingModel and confirm by choosing OK.

By doing so, you specify that all views and controllers of FlightBookingtComp have a dependency relationship with the model FlightBookingModel.

Creating a Context for the Custom Controller

5. In the left toolbar, select the icon to Create a data link. Click on the field FlightBookingCust and drag it into the field FlightBookingModel.

The dialog box Edit Model Binding appears.
6. Drag Bapi_Flbooking_Createfromdata_input from the right to create new context elements and bind them to the left tree.

7. In the dialog that appears, select the model nodes Bapi_Flbooking_Createfromdata_Input, Booking_Data, and Output. Under the node Output, select the nodes Return and Ticket_Price.

8. Rename the node Output to Output_Flbooking and the node Return to Return_Flbooking by clicking in the corresponding field Name on the right.
9. Confirm with OK.

The corresponding model is created.

10. Drag the following model classes to the context of FlightBookingCust in the same way:

   - Bapi_Flight_Checkavailability_Input and select the following model nodes including their model attributes:
     - [ ] Bapi_Flight_Checkavailability_Input
     - [ ] Output (rename to Output_Availability)
     - [ ] Return (rename to Return_Availability)

   - Bapi_Transaction_Commit_Input and select the following model node including its model attributes:
     - [ ] Bapi_Transaction_Commit_Input

11. Confirm with Finish.
Creating a Context for the View Controller

12. In the left toolbar, select the icon to Create a data link. Click on the field FlightBookingView and drag it into the field FlightBookingCust.

The dialog box Edit Context Mapping appears.

13. Proceed as described above and map new context elements in the FlightBookingView to the following model nodes including their attributes:

- Bapi_Flbooking_Createfromdata_Input
  - Booking_Data
  - Output_Flbooking
    - Return_Flbooking
    - Ticket_Price
- Bapi_Flight_Checkavailability_Input
  - Output_Availability
  - Return_Availability


15. Save your work by choosing (Save All Metadata) from the toolbar.

Result

Using the model definition as your starting point, you have created a context for the custom controller FlightBookingCust and bound the appropriate context model elements for the input and output structures to the corresponding model classes.

You have created the necessary view contexts and mapped them to the custom controller context.

Next step:

Binding Input Objects to the Model Nodes
Binding Input Objects to the Model Nodes

Here you will define new objects that represent the input of the BAPIs at runtime. These objects must be bound to the model nodes that were declared at design time.

In the flight booking application, you are to be able to check flight availability and make new flight bookings are reservations via Commit. For this reason, the following objects are required, which are bound to the relevant context nodes:

- **inputAvailibility** (of the type: Bapi_Flight_Checkavailability_Input)
- **input** (of the type: Bapi_Flbooking_Createfromdata_Input)
- **inputCommit** (of the type: Bapi_Transaction_Commit_Input)

Procedure

1. Open the Controller Editor for the custom controller **FlightBookingCust**.
2. Choose the Implementation tab.
3. In the standard method **wdDoInit()**, add the following lines of code between //@@begin wdDoInit() and //@@end:

   ```java
   public void wdDoInit()
   {
      //@@begin wdDoInit()

      //Create a new element in node Bapi_Flbooking_Createfromdata_Input
      Bapi_Flbooking_Createfromdata_Input input = new Bapi_Flbooking_Createfromdata_Input();
      wdContext.nodeBapi_Flbooking_Createfromdata_Input().bind(input);

      //Create a new element in node Booking_Data
      input.setBooking_Data(new Bapisbonew());

      //Create a new element in node Bapi_Flight_Checkavailability_Input
      Bapi_Flight_Checkavailability_Input inputAvailibility = new Bapi_Flight_Checkavailability_Input();
      wdContext.nodeBapi_Flight_Checkavailability_Input().bind(inputAvailibility);

      //Create a new element in node Bapi_Transaction_Commit_Input
      Bapi_Transaction_Commit_Input inputCommit = new Bapi_Transaction_Commit_Input();
      wdContext.nodeBapi_Transaction_Commit_Input().bind(inputCommit);

      //@@end
   }
   ```

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

5. Save your work by choosing (Save All Metadata) from the toolbar.
Next step:
Implementing the Custom Controller
Implementing the Custom Controller

The actual BAPIs are called via the `execute()` method call of the model object currently stored in the context model node. This already contains the flight booking data entered by the user (through data binding and context mapping).

The data stored in the view context is a copy of the data stored in the model – that is, the one does not directly reference the other. Therefore, the view context does not yet contain the returned results of the BAPI call executed previously and stored in the model.

For this reason, you therefore need to explicitly invalidate the output model nodes (these are contained as inner nodes). The output data most recently stored in the model is then transmitted to the corresponding context node element.

Procedure

Declaring the Methods

1. Open the editor for the custom controller `FlightBookingCust` again.
2. Choose the Methods tab.
3. Choose New.
4. Select the Method option and choose Next.
5. In the wizard screen that appears, enter the name `executeBapi_CheckAvailability` for this new method and assign it the return type `void`. Choose Finish.
6. In the same way, create the following methods:
   - `executeBapi_FlightBooking`
   - `executeBapi_Commit`

Implementing the Controller

In order to establish the back-end connection, you must implement the methods created before.

7. In the method `executeBapi_CheckAvailability()`, add the following lines of code:

```java
executeBapi_CheckAvailability()
public void executeBapi_CheckAvailability( )
{
    //@@begin executeBapi_CheckAvailability()
    IWDMessagManager manager = wdComponentAPI.getMessageManager();
    try {
        //Calls remote function module BAPI_Flight_Checkavailability
        wdContext.currentBapi_Flight_Checkavailability_InputElement().modelObject().execute();
        //Synchronise the data in the context with the data in the model
        wdContext.nodeOutput_Availibility().invalidate();
    } catch (WDDynamicRFCExecuteException ce) {
        manager.reportException(ce.getMessage(), false);
    }
    //@@end
}
```

8. In the method `executeBapi_FlightBooking()`, add the following lines of code:
executeBapi_FlightBooking()

```java
public void executeBapi_FlightBooking() {
    //@@begin executeBapi_FlightBooking()
    IWDDMessageManager manager = wdComponentAPI.getMessageManager();
    try {
        //Calls remote function module BAPI_FLBOOKING_Createfromdata
        wdContext.currentBapi_Flbooking_Createfromdata_InputElement().modelObject().execute();
        //Synchronise the data in the context with the data in the model
        wdContext.nodeOutput_Flbooking().invalidate();
    } catch (WDDynamicRFCExecuteException ce) {
        manager.reportException(ce.getMessage(), false);
    }
    //@@end
}
```

9. In the method `executeBapi_Commit()`, add the following lines of code:

```java
executeBapi_Commit()

```java
public void executeBapi_Commit() {
    //@@begin executeBapi_Commit()
    IWDDMessageManager manager = wdComponentAPI.getMessageManager();
    try {
        //Calls remote function module BAPI_Transaction_Commit
        wdContext.currentBapi_Transaction_Commit_InputElement().modelObject().execute();
    } catch (WDDynamicRFCExecuteException ce) {
        manager.reportException(ce.getMessage(), false);
    }
    //@@end
}
```

We could also write the appropriate rollback method `executeBapi_Rollback( )` in the same way. However, rollback is not a part of our sample scenario.

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

7. Save the new metadata by choosing `Save All Metadata` from the toolbar.

**Next step:**

[Completing the Layout of FlightBookingView Using Templates](#)
Completing the Layout of FlightBookingView

Initially, the project TutWD_FlightBooking_Init already contains some user interface elements:

The form field for the user input, which is added using a template, is to be displayed in the GroupInput. GroupInput already contains a button bound to the action BookFlight.

Now you need only make a few additions to the second group GroupOutput, which is used for outputting a result area. Here you will use a template to add some data for the flight booking or flight reservation confirmation to the view. However, GroupOutput is only to be displayed if the flight booking or reservation is successful. For this reason, the visibility property of GroupOutput is bound to the context element GroupVisible. Since the header of GroupOutput depends on the user input (reservation or booking), GroupOutput_Header is bound to the context element GroupHeader.

The action BookFlight and the context elements GroupHeader and GroupVisible are defined in the implementation section of the FlightBookingView.

Procedure

1. In the project structure, double-click the node for the view FlightBookingView.
   The View Designer for the FlightBookingView appears.
2. Open the context menu of the transparent container Template_CONTAINER to apply the template input fields and choose Apply Template.
3. In the dialog that appears, choose Form and continue with Next.
4. Select the model node Booking_Data with its model attributes and confirm with Next.
5. Change the Name from _Class to Class.
6. Continue with Finish.

7. In order to display the booking confirmation, open the context menu of the group **GroupOutput** and choose **Apply Template**.

8. Choose Form and continue with Next.

9. Under the model node **Output_FLbooking**, choose the model attributes **Airlineid** and **Bookingnumber**. Under the model node **Ticket_Price**, choose the model attributes **Price** and **Curr**. Confirm by choosing Next.

10. In the next dialog box, change all **InputFields** in the **Editor** column to **TextViews**.

11. Confirm with Finish.

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

**Result**

The **Apply Template** function adds the corresponding UI elements to the view layout.
Next step:

Implementing the FlightBookingView Controller
Implementing the FlightBookingView Controller

Implementing the view controller in this scenario should implement the following process:

If the user presses the Book button, the system first checks whether the desired flight is even available. If the flight specified in the input form exists, a dialog box is displayed containing the buttons Reserve Flight, Book Flight, and Cancel. If it does not exist, a warning message is displayed. Once the user has booked or reserved a flight – and thus completed the transaction successfully – the flight booking/reservation data (for example, the booking number) is displayed in the GroupOutput in the view.

The actions Book, Reserve, and Cancel for the corresponding buttons already exist in the initial project template. These actions are bound to the event handlers onActionBook(), onActionReserve(), and onActionCancel() in the view controller FlightBookView.java.

Procedure

Implementing wdDoInit()

1. Open the FlightBookingView and choose the Implementation tab.
2. In the wdDoInit() method, insert the following line of code to make the UI element GroupOutput invisible:

```java
public void wdDoInit()
{
    //@@begin wdDoInit()
    wdContext.currentUIElementsElement()
        .setGroupVisible(WDVisibility.NONE);
    //@@end
}
```

Triggering the flight booking

1. Choose the Properties tab.
2. Choose Add.
3. Select FlightBookingComp and confirm with OK.
4. Choose the Implementation tab.
To check whether the desired flight is available, the user input for the flight booking must be written to the context for the `CheckAvailability` BAPI.

5. To do so, add the following source code to the event handler `onActionBookFlight()`:

```java
public void onActionBookFlight(
    com.sap.tc.webdynpro.progmodel.api.IWDCustomEvent wdEvent )
{
    //@@begin onActionBookFlight(ServerEvent)
    wdContext.currentBapi_Flight_Checkavailability_InputElement()
        .setAirlineid(
            wdContext.currentBooking_DataElement().getAirlineid());
    wdContext.currentBapi_Flight_Checkavailability_InputElement()
        .setConnectionid(
            wdContext.currentBooking_DataElement().getConnectionid());
    wdContext.currentBapi_Flight_Checkavailability_InputElement()
        .setFlightdate(
            wdContext.currentBooking_DataElement().getFlightdate());
    //@@end
}
```

To check the flight availability, the system calls the method `executeBapi_CheckAvailability()`, which was previously implemented in `FlightBookingCust`. If the flight is available, a `Success` message is written to the context. The message type “S” (success) is stored in the context attribute `Type`.

If the flight is available, a dialog box is displayed in which the user can either book or reserve the flight, or cancel the flight booking altogether. However, if the flight is not available, a warning message is displayed.

6. To implement this function, add the following program code to the end of the view method `onActionBookFlight()`:

```java
public void onActionBookFlight(
    com.sap.tc.webdynpro.progmodel.api.IWDCustomEvent wdEvent )
{
    ...
    wdContext.currentBapi_Flight_Checkavailability_InputElement()
        .setFlightdate(
            wdContext.currentBooking_DataElement().getFlightdate());
    wdThis.wdGetFlightBookingCustController().executeBapi_CheckAvailability(
    ) ;
    if (wdContext.currentReturn_AvailibilityElement().getType().equals("S")){
        IWDConfirmationDialog dialog;
        String dialogText = "The flight is available! ";
        dialog = wdComponentAPI.getWindowManager()
            .createConfirmationWindow(
                dialogText,
            wdControllerAPI.getControllerInfo().findInEventHandlers("Reserve"),
            "Reserve Flight" );
```
Further information about dialog boxes are available in the tutorial Dialog Boxes in Web Dynpro Applications.

**Implementing the event handlers onActionReserve, onActionBook, and onActionCancel**

1. In the method `onActionReserve()`, add the following lines of code:

```java
public void onActionReserve(com.sap.tc.webdynpro.progmodel.api.IWDCustomEvent wdEvent)
{
    //@@begin onActionReserve(ServerEvent)
    wdContext.currentBapi_Flbooking_Createfromdata_InputElement().setReserve_Only(true);
    wdThis.wdGetFlightBookingCustController().executeBapi_FlightBooking();
    wdThis.wdGetFlightBookingCustController().executeBapi_Commit();
    wdContext.currentUIElementsElement().setGroupHeader("Successful Flight Reservation");
    wdContext.currentUIElementsElement().setGroupVisible(WDVisibility.VISIBLE);
    //@@end
}
```

To reserve a flight, the context attribute `Reserve_Only` is set to `true`. The system then executes the methods `executeBapi_FlightBooking()` and `executeBapi_Commit()`, which are defined in FlightBookingCust.

To display GroupOutput, the context attributes `GroupHeader` and `GroupVisible` are changed accordingly.

2. In the method `onActionBook()`, add the following lines of code:
To book a flight, the context attribute `Reserve_Only` is set to `false`. The system then executes the methods `executeBapi_FlightBooking()` and `executeBapi_Commit()`, which are defined in `FlightBookingCust`.

To display the UI element `GroupOutput`, the context attributes `GroupHeader` and `GroupVisible` are changed accordingly.

3. In the method `onActionCancel()`, add the following lines of code:

```java
onActionCancel()

public void onActionCancel(com.sap.tc.webdynpro.progmodel.api.IWDCustomEvent wdEvent )
{
    //@@begin onActionCancel(ServerEvent)
    wdComponentAPI.getMessageManager().reportWarning(
        "The flight was not booked."
    );
    //@@end
}
```

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

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

**Next step:**

Building, Deploying, Configuring and Running Your Application
Building, Deploying, Configuring, and Running Your Application

You have now reached the last stage in the development of your example application. However, some preparation is still essential before you can successfully deploy and run the application on the J2EE Engine. Go through each of the following prerequisites carefully.

Prerequisites

☐ You have made sure that the relevant SAP basis system, which you will be accessing remotely to retrieve and write the flight data, is currently available and contains flight data.

☐ The SAP J2EE Engine has been started.

To do this, refer to Starting and Stopping the SAP J2EE Engine.

☐ You have checked that the configuration settings for the J2EE server are entered correctly in the Developer Studio.

To check the server settings, choose the menu path Window → Preferences → SAP J2EE Engine.

☐ The connection parameters for the used SLD are defined in the J2EE Visual Administrator.

Procedure

Building the Project

1. If you have not already done so, save the metadata for your project in its current state.

2. In the Web Dynpro Explorer, choose Rebuild Project from the context menu of the project node TutWD_FlightBooking_Init.

Make sure that the Tasks view does not display any errors for your project.

Deploying the Project

1. In the Web Dynpro Explorer, select the project node TutWD_FlightBooking_Init and choose Create Archive from the context menu.

2. Choose Deploy Project from the context menu of the project node.

Creating the JCO Connections in the Content Administrator

The logical system names used in the model declaration must be associated with an actual SAP system defined in the SLD before this application can be executed. For this reason, you need to create JCO connections in the Web Dynpro Content Administrator. To use an Adaptive RFC model in your Web Dynpro application, you need to define two connections:

- A connection to get the required (Dictionary) metadata information
- A connection to read the application data

To create the new JCO connections proceed as follows:

1. Open the Web Dynpro Content Administrator on your J2EE Engine using the URL:


2. If you are using the Content Administrator for the first time, you need to perform the self registration.
3. Select the Browse tab and navigate to the application node local → TutWD_FlightBooking_Init → Applications → FlightBookingApp.

4. Choose the JCO Connections tab.

   Two destinations named WD_FLIGHTBOOKING_MODELDATA_DEST and WD_FLIGHTBOOKING_RPC_METADATA_DEST are shown in this view and both of them have the status red. This means that the JCO connection is not yet maintained in the assigned SLD.

5. Select WD_FLIGHTBOOKING_MODELDATA_DEST and choose Create to configure a new JCO connection.

   a. Go through the steps of the JCO connection creation wizard and specify the connection data.

   b. After making the required entries, choose Finish.

   c. Test your configuration data by pressing the Test button.

6. Do the same for WD_FLIGHTBOOKING_RPC_METADATA_DEST.

7. If the tests were successful, you can close the Content Administrator. Otherwise correct your connection entries.

**Launching the Application**

1. In the Web Dynpro Explorer, open the context menu for the application object FlightBookingApp.

2. Choose Run.

**Result**

The Developer Studio automatically launches your application in the Web browser. Test the sample application by entering valid data in the Flight Booking form and choosing Book. Your application will then check if the requested flight is available. If it is, a confirmation dialog box will be displayed.
If the user chooses *Book Flight* in the dialog box, the application tries to write the flight booking data to the ABAP system. If the transaction was successful, some data (booking number, price, and so on) retrieved from the BAPI call is also displayed in the result area of your application.
Checking the Result

You do not have to carry out the following check. It is only necessary if you work with the ABAP Workbench.

To make sure that the booking data is written into the database table, you can start the ABAP Workbench and use the SE16 transaction to check the content of the SBOOK table.
## Handling Transactions with BAPIs in Web Dynpro

### Data Browser: Table SBOOK Select Entries

<table>
<thead>
<tr>
<th>MANDT</th>
<th>CARRID</th>
<th>CONNID</th>
<th>FLDATE</th>
<th>BOOKID</th>
<th>CUSTOMID</th>
<th>CUSTTYPE</th>
<th>SMOKER</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>LH</td>
<td>2407</td>
<td>14.03.2004</td>
<td>00020101</td>
<td>00000168</td>
<td>B</td>
<td></td>
</tr>
</tbody>
</table>

[See the SAP Online Help](https://help.sap.com) for more information on handling transactions with BAPIs in Web Dynpro.