How to…
Build a Web Dynpro Application

PUBLIC

ASAP "How to…" Paper

January 2004

.

Copyrights

Content

Preface

What is Web Dynpro?

From a technological point of view, SAP's Web Dynpro for Java is a revolutionary step in the development of web-based user interfaces. It is completely unlike any design paradigm ever used by SAP before and represents a quantum leap in the development of web-based, ERP applications.

What is the Design Philosophy Behind Web Dynpro?

Web Dynpro applications are built using declarative programming techniques based on the Model View Controller (MVC) design paradigm. That is, you specify what user interface elements you wish to have on the client, and where those elements will get their data from. All the code to create the user interface is then generated automatically within a standard runtime framework. This relieves you from the repetitive coding tasks involved in writing HTML and then making it interactive with JavaScript.

Terminology

Certain terms such as “component”, “element”, “context” and “interface” have specific meanings within the context (sic!) of the Web Dynpro for Java toolset. Therefore, to avoid ambiguity, such words will only be used when their Web Dynpro meaning is intended.

Screenshots

For the sake of brevity, various graphical figures in this document have been cropped or resized. Therefore, when you look at the corresponding screen in your installation of the SAP NetWeaver Developer Studio, it may very well be larger than the image displayed in this document.

Prerequisites

Hardware

If the SAP NetWeaver Developer Studio and the J2EE Engine are to be installed on the same machine, you will need at least 512Mb of RAM. This configuration however, will be slow due to frequent paging by the operating system. Therefore, for efficient operation, 1Gb of RAM is recommended.

If the SAP NetWeaver Developer Studio and the J2EE Engine are to be installed on different machines, then 512MB of RAM on each machine is acceptable.

Installed Software

You must install the current version of the SAP NetWeaver Developer Studio and the latest version of the J2EE Engine.

You need a working installation of SAP GUI and a valid user ID to log on to an SAP system with sufficient authorisation to perform an RFC call. In this example, the Web Dynpro application will call `BAPI_FLIGHT_GETLIST`. If your SAP system does not contain the correct information for this BAPI to operate, then you can call any BAPI that uses both `IMPORTING` structures and has a standard select option table as a `TABLES` parameter. Your replacement BAPI should return information in a `TABLES` parameter.

SAP System

Your J2EE Engine must have access to an SAP system in which `BAPI_FLIGHT_GETLIST` can be found. You must also be able to log on to this system with a user having sufficient authority to make an RFC call.

Introduction

Improved User Experience

The Web Dynpro technology provides a development and runtime environment for Web applications and enhances classical Web development to build easily adaptable user interfaces. SAP's Web Dynpro technology closes significant gaps between typical Web development tools and the needs of cost-effective, responsive, easy-to-use, maintainable, and professional browser-based user interfaces for business solutions. Web Dynpro uses design principles similar to SAP's Dynpro technology, but it is a completely new technology geared exclusively towards Web applications. Its main features include the following:

- Usability
- Abstract modelling
- Personalization and customization
- Separation of presentation layers and business logic
- Generic services
- Portability

Web Dynpro applications run in the SAP Enterprise Portal.

Building a High Fidelity Web User Interface

The main aims are to avoid coding as far as possible and achieve independence from the back-end platform and front-end technology. Web Dynpro delivers a declarative metamodel to develop user interfaces while writing less programming code. Web Dynpro uses this abstract definition to create a ready-to-run Web application for runtime platforms.

Additionally, Web Dynpro has a graphical toolset and an embedded IDE (Integrated Development Environment) that help developers to create the Web Dynpro metadata.

How is Web Dynpro Different From Other Web Development Tools

Many differences could be mentioned here, but from a developer’s point of view, one of the most fundamental differences is this: In other web development tools, such as Java Server Pages for instance, the unit of development is the web page, and your application consists of a set of connected pages that together, supply the required business functionality.

Not so with Web Dynpro! In the Web Dynpro world, the unit of development is the “component”. A component is a set of related Java programs that together, form a reusable unit of business functionality.

A component however, can have *multiple* views. A view is also a set of Java programs that function as a subordinate unit within a component. A view cannot exist outside the scope of its parent component, but a component could have many views.

For those of you already familiar with server page based web interfaces, you will find that the Web Dynpro architecture requires that you make a fundamental shift in your thinking.

Building a Web Dynpro Application

Introduction

To design a Web Dynpro application you need a good understanding of the business requirements. This understanding must then be translated into a set of functional requirements that can be met by the capabilities of the Web Dynpro toolset.

Since Web Dynpro is a declarative programming toolset, it handles all the mundane aspects of programming the UI for you. So you can shift your attention away from the specific details of coding the user interface and onto the flow of data through the business process itself.

For instance, if you want a list of suppliers for a particular product displayed in a dropdown list, you just declare a dropdown list and specify where that UI element will get its data from in a data storage area known as the context. To populate the context you have to write some coding but Web Dynpro will generate the coding for the UI and data transport for you.

The only code you have to write is the implementation of those actions that cannot be described in a declarative manner. Such actions would include silently logging on to an SAP system, executing a BAPI call or implementing an action listener event.

Web Dynpro Project Structure

Overview

A Web Dynpro project is the container within which you can build one or more applications. Here, the project is called `FlightListProject`. Figure 1 below shows an expansion of the different units which, taken together, make up this particular Web Dynpro project.

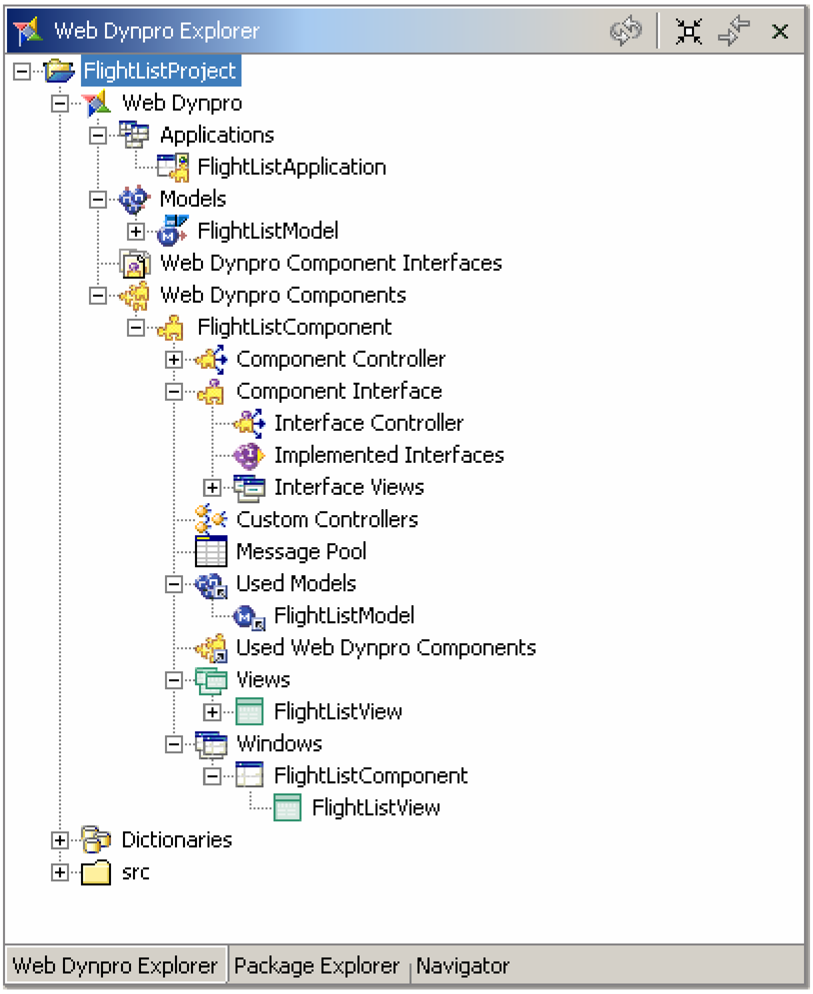


Figure 1: Project Structure

The root node called `FlightListProject` is the parent for the following items:

- Data dictionary definitions of simple (scalar) data types
- Source tree (reference only)
- Web Dynpro program(s)

Web Dynpro Project Units

Project

A project is the container within which all your related Web Dynpro dictionary structures, applications and components live.

Application

An application defines an entry point into a Web Dynpro component. Components, in themselves, cannot be accessed directly by the client software. Instead, they must be accessed via an application. The application associates a URL with a standard entry point in the component interface known as a plug. There is a one-to-one relationship between an application and a component entry point. Therefore, if you wish to give your component multiple entry points, you must define multiple applications.

Model

A model is any layer of code that encapsulates some business functionality external to the Web Dynpro environment. A model provides access to functionality such as BAPI calls or Web services, and can be implemented as a set of proxy objects or Enterprise Java Beans and so on. You create models at project level and each component in the project uses them as needed.

Component Interface

The Web Dynpro component interface consists of two parts: a visual one and a programmatic one, and defines the set of publicly accessible entry points to the component.

You can implement coding to validate user parameters received in the component interface controller. If the component functionality is sufficiently complex, you can pass user parameters to the configuration controller described below.

Configuration Controller

The configuration controller is the component responsible for adapting the behaviour of the overall Web Dynpro application. It is within this controller that configuration parameters should be processed and behavioural modifications made to the application as a whole.

Controllers in general mediate between the user interface and the business logic. They

- Handle events in event handlers
- Send actions to the business logic
- Collect data to be displayed on the user interface

Web Dynpro has two types of controller:

- View controllers
- Custom controllers: Configuration controllers and component controllers are types of custom controller.

Web Dynpro Component

A component is the unit of a project that contains actual functionality, and if written carefully, can either be reused by other projects or within other components of the same project.

Window

Whenever you create a component, a window is automatically created for that component. Normally, this window will have at least one view embedded within it, and becomes the default interface with which the user interacts. However, you can create what is known as a faceless component, that is, a component that has no user interaction. Such faceless components will only work in combination with normal components that contain views, since the whole point of the Web Dynpro toolset is to provide a browser-based user interface to business functionality.

View

A view is the visual component inside a window with which a user interacts. To be visible a component, it must have at least one view. A window can have multiple views arranged in different layouts, for example grid-, flow- or matrix layout, or even have views nested within views.

Implementation

What follows is a step-by-step guide to creating a simple Web Dynpro application that will pass search parameters to the function module `BAPI_FLIGHT_GETLIST` in an SAP system and then display the results in a table.

Creating a Project

Choose *File -> New -> Project*. The dialog box shown in Figure 2 below is displayed.

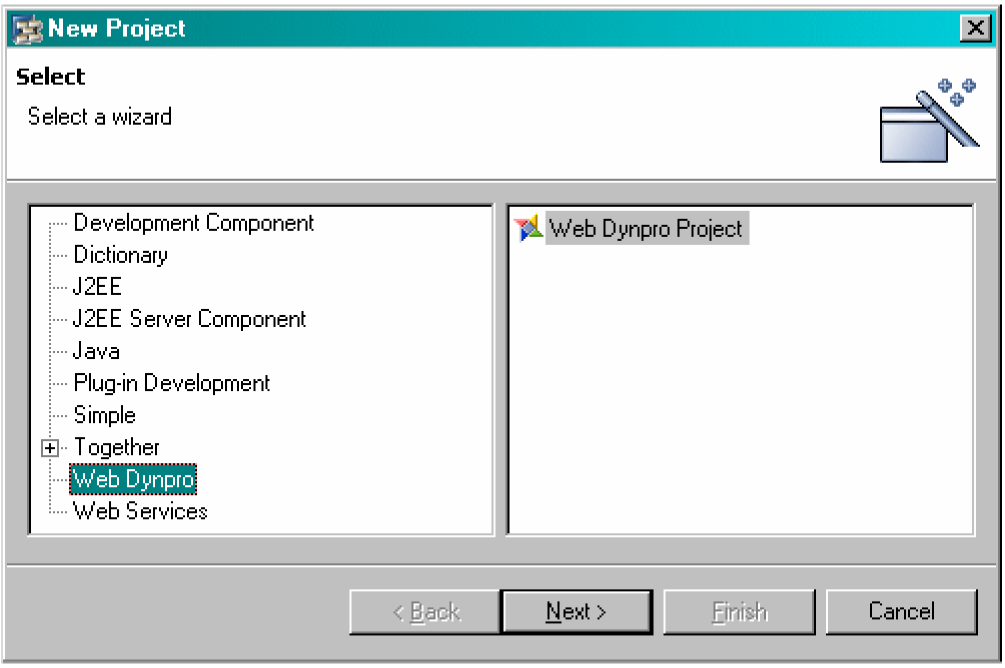


Figure 2: Creating a New Project. Step 1

Select *Web Dynpro* and then *Web Dynpro Project* and choose *Next*.

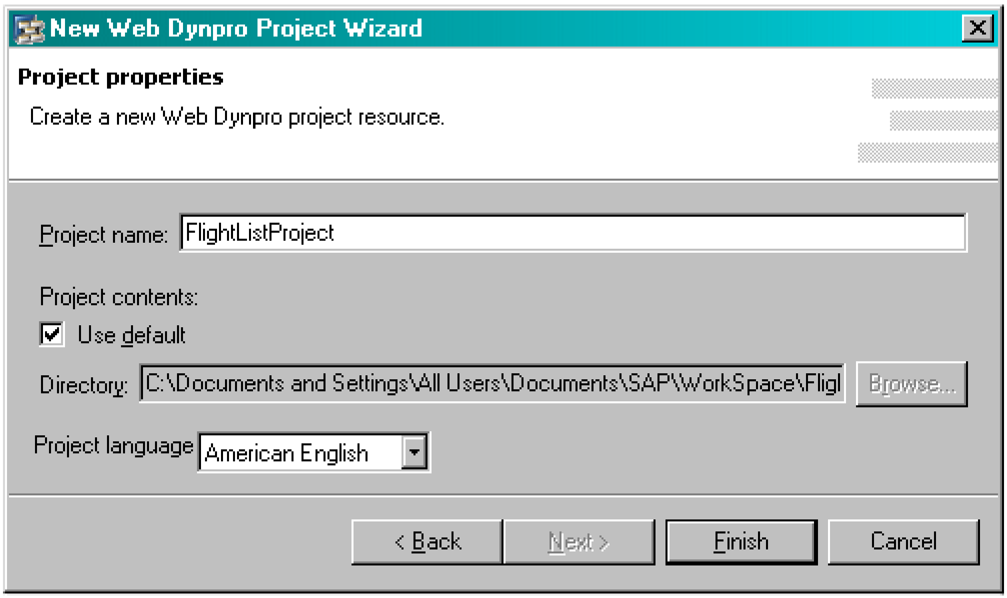


Figure 3: Create a New Project. Step 2

Figure 3 above shows the screen where you enter the name of your project.

Enter **`FlightListProject`** and press *Finish*. The SAP NetWeaver Developer Studio generates the required files in which the project metadata will be stored.

The Web Dynpro Explorer window is user-specific, so you will not see the exact contents of Figure 4 below. Nonetheless, your new project will look like the expanded tree node below.

Figure 4: Structure of a New Project

Creating a Web Dynpro Component

A component is the part of the Web Dynpro application that contains the actual functionality. Click on the tree node  `Web Dynpro Components` with the alternate mouse button and the context menu shown in Figure 5 below appears.

Figure 5: New Web Dynpro Component Menu

You now have to specify the details of your new component.

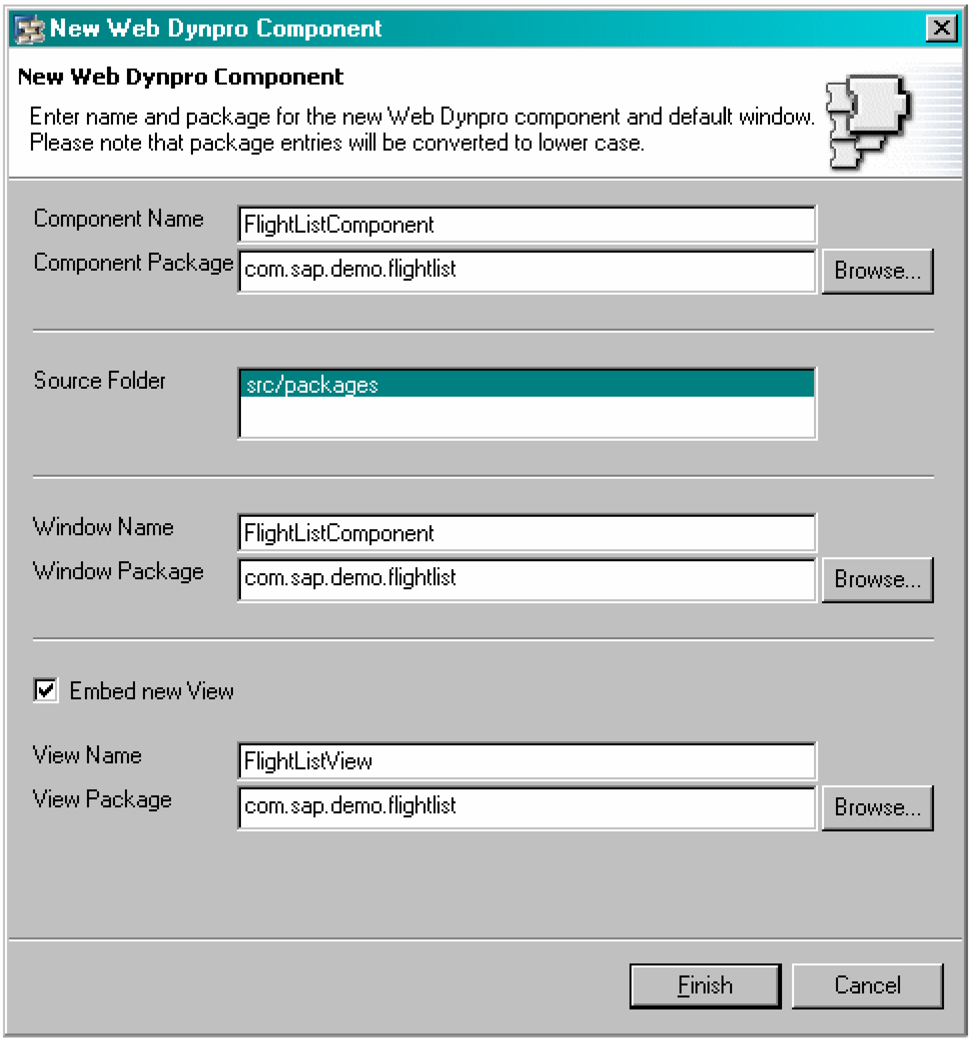


Figure 6: Create a New Web Dynpro Component.

Figure 6 above shows the screen where you specify the names of the component and the Java package. Enter the values shown above.

The values in the fields *Window Name* and *Window Package* default to automatically generated values. You can override them if required.

Notice also that a view will be created automatically if the *Embed new View* checkbox is selected. It is useful to leave this checkbox selected since it will automate a process you would otherwise have to perform manually.

Change the *View Name* field to `FlightListView` because there is no particular need to include the word `Component` in a view name.

When you choose *Finish* the SAP NetWeaver Developer Studio generates all the necessary metadata and Java files.

On the toolbar press the `Save All Metadata` icon .

Your Web Dynpro project will now look like Figure 7 below.

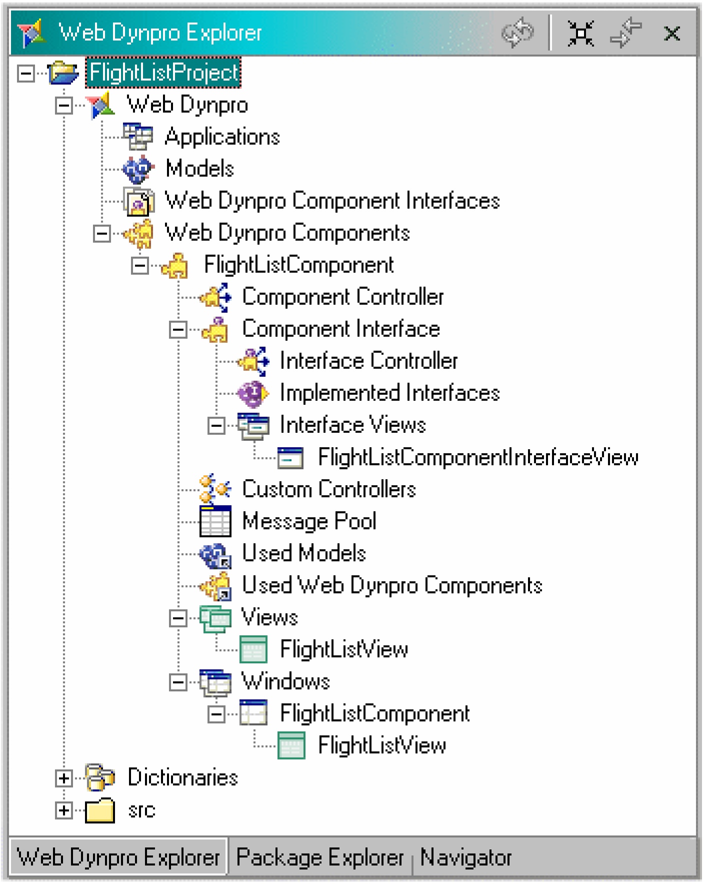


Figure 7: The Structure of a New Web Dynpro Component

Once you have created the component called **`FlightListComponent`**, the following programs are automatically created:

- A component controller
- A component interface controller
- A configuration controller
- A default window
- A view embedded in the default window

The controllers are all separate Java programs, yet they function together to behave as a single component.

Component Interface

Underneath the component interface is a controller. The methods and attributes of this controller provide the only publicly accessible entry point into the component. The methods and attributes of all the other controllers making up this component are completely encapsulated, and so invisible to the outside world.

If the functionality of your component is embedded in another Web Dynpro component, the parent component only has access to those methods declared in the interface controller.

The interface controller will, by default, have a view automatically defined. This is because components usually have a graphical user interface. The implementation of the interface view does not normally require any work beyond simply declaring the existence of the component. Unless you say otherwise, whatever view you first embed into the component's window will become the default component interface view.

All controllers can have defined entry and exit points known as plugs. Plugs are the standard entry and exit points to and from all controllers. This means that when the Web Dynpro runtime environment invokes a controller for the very first time, a special inbound plug must be present that acts as the standard entry point. This plug must be of type **`startup`**, and is normally called **`Default`**.

You may change the name of plug **`Default`** if you wish, but if you change its type to something other than **`startup`**, then when the Web Dynpro runtime fires the event to start the application, an unhandled event exception occurs.

Since you have a standalone component that does not embed any other components, and is not embedded in another component, you can ignore the interface controller.

Configuration Controller

You can use a configuration controller if you want your component to modify its behaviour on the basis of parameters it receives from the user, or the contents of configuration files.

A configuration controller is always present, even if it is not used.

Default Window

The default window is usually needed since almost all components have some sort of visual interface. However, if you are creating a faceless component, the default window will still be present, but just unimplemented.

Component Controller

The component controller is the main controller for the entire component and *never has a visual interface*. This controller is the main repository for the entire component's data and is the heart of the component's processing logic. Different subsets of the data held within the component controller can be supplied to view controllers for visualisation or custom controllers for some specific processing task.

Creating a Model

You do not have to create the model exactly at this point in the sequence of events. You can create it either before or after creating the component. However, it is good practice to create models first, because you have to declare their usage within a component. In other words, a component is a consumer of a model.

Specifying the Model Type

Directly under the `Web Dynpro` node of the project tree is a node called `Models`. Click on this node with the alternate mouse button and choose *Create Model.* The screen in Figure 8 below appears.

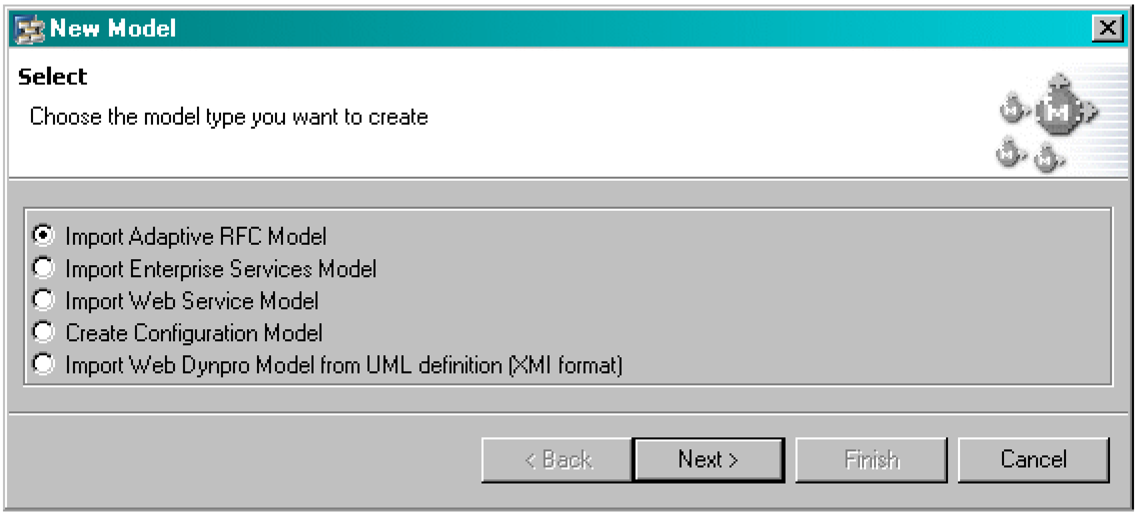


Figure 8: Creating a New Model. Choosing the model type

A model needs to be created that represents a BAPI call. This is done by selecting the *Import Adaptive RFC Model* option and pressing *Next*. Now you are asked for the name and package of your model. You don't have to end the model name with the word `model` but it helps later on when you use the generated code.

Important: In order for this type of RFC call to work, the System Landscape Directory (SLD) must have been correctly configured in your J2EE Engine. If this step is not performed, then you will be able to build and deploy this application, but you will be unable to run it.

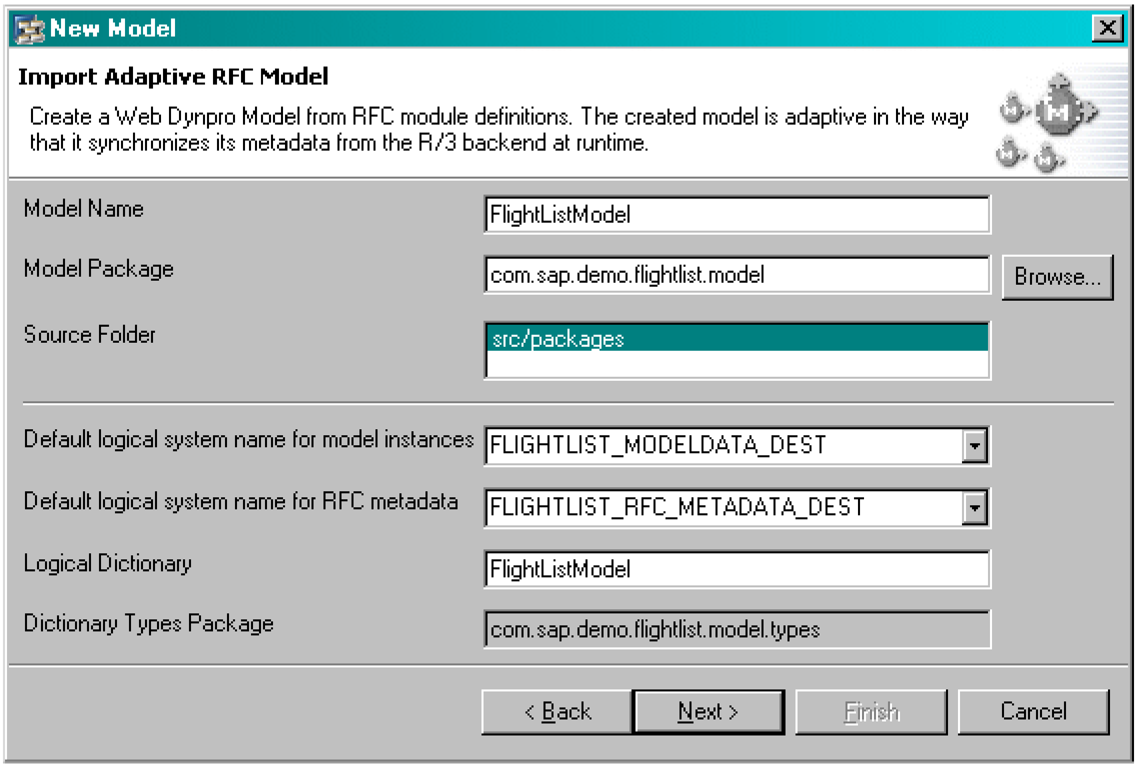


Figure 9: Create a New Model. Specify model details.

Figure 9 above shows the completed screen. Now choose *Next*.

Generating a Model from an ABAP Function Module Interface

Since you have selected the adaptive RFC model, the model creation process now requires logon information for the SAP system from which the BAPI interface will be derived. An example of this is shown in Figure 10 below. Please substitute the host name, system name and user details that are applicable for your installation.

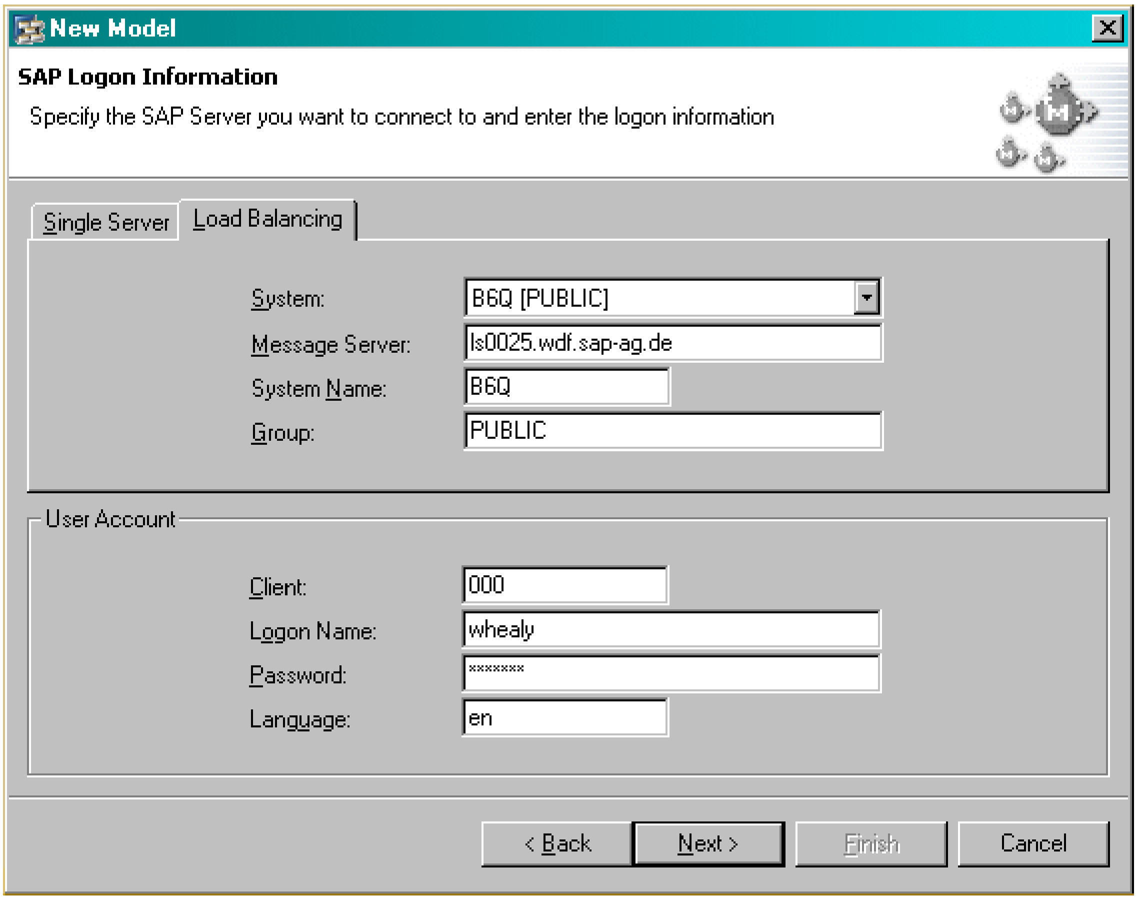


Figure 10: Create a New Model. Logon to SAP.

Figure 10 above shows the *Load Balancing* tab. You should use this tab if you are using logon load balancing for connecting to your SAP system. However, if you do not have group logon configured for your system, select the *Single Server* tab and choose a suitable SAP system from the same list of systems used by the SAP Logon program.

You must supply the details for a valid SAP user with authorization to perform a remote function call. If the user ID you specify has insufficient authorisation in this area, then the Connectivity Builder will be unable to proceed.

After the Connectivity Builder has connected to the target system, you will be presented with an *empty* list of BAPIs. This is not a mistake, but a deliberate design feature!

Since the average SAP system has more than 15,000 remote callable function modules, it would be very wasteful of system and network resources to transmit and process this volume of information, only for the user to select one or two items from the list. Therefore, for performance reasons, the list initially appears empty.

You may now enter either the exact name of the BAPI or a standard SAP search string. Enter the value `bapi_flight_*` in the field *Function Name* and press the button *Search*.

The results are shown below in Figure 11 below.

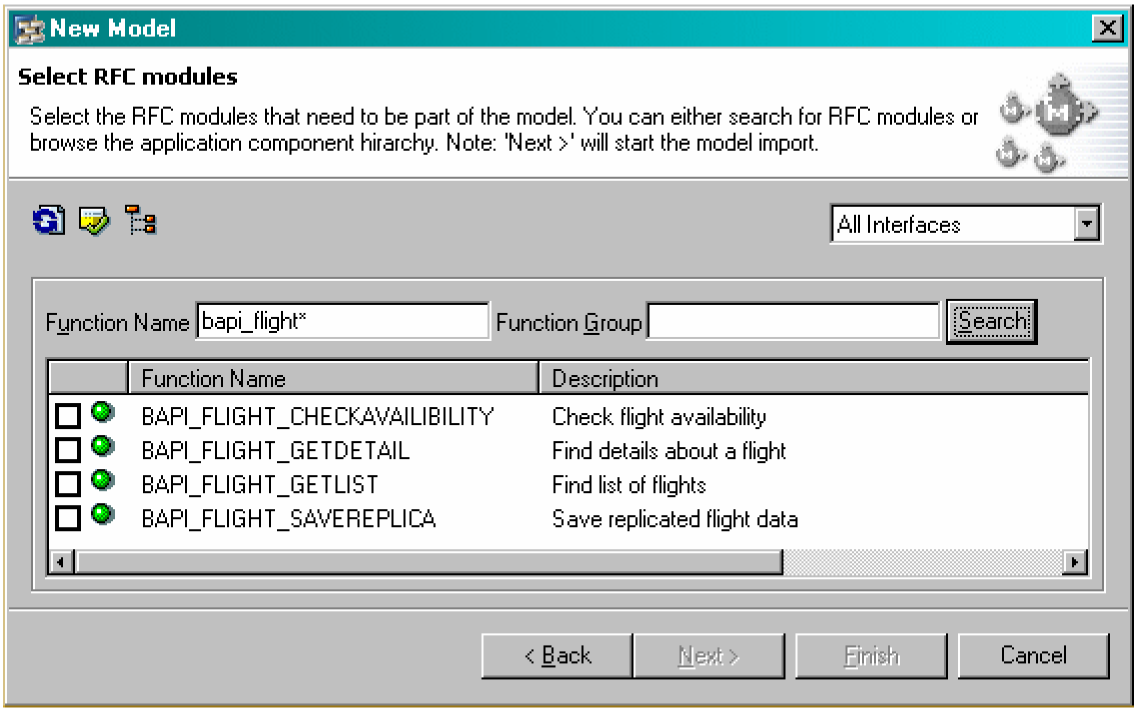


Figure 11: Create a New Model. Choose function module.

Check the box next to `BAPI_FLIGHT_GETLIST` and choose *Next* and then choose *Finish* in the next dialog box to start the proxy generation process. Depending on the number of BAPIs selected, and the number of interfaces, the interface generation process may take several minutes to complete.

The next screen you see shows the log file of the model creation process. Press *Finish*.

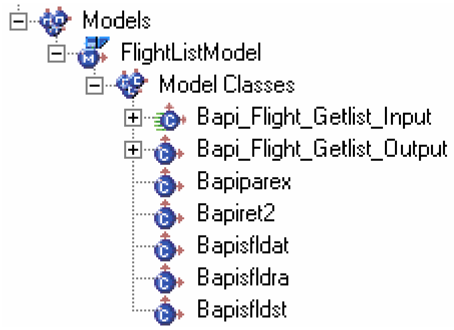
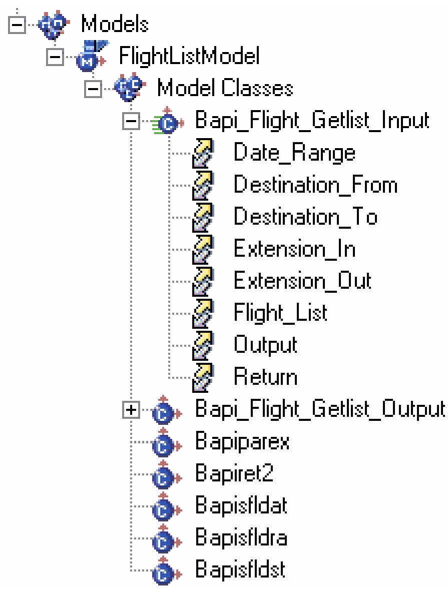


Figure 12: Structure of Generated Model. Part 1

After the proxy generation process has completed, the  models section of your project tree will now have a new node under it called `FlightListModel` as shown in Figure 12 above.

Now save your work using the  icon on the toolbar.

Explanation of Generated Model Classes



The main controlling class for the RFC call is the one ending in `_Input`. To call the function module, you first have to create an instance of the `Bapi_Flight_Getlist_Input` class. All the inbound parameters required by the BAPI are added to this `_Input` instance, that is, the `IMPORTING` and `TABLES` parameters. The expansion of this class is shown in Figure 13.

If you compare the model object expanded in Figure 13 with the structure of the ABAP interface for function module `BAPI_FLIGHT_GETLIST` in Textbox 1, you will notice that the single field `IMPORTING` parameters are missing from the model class hierarchy. This is because this model class hierarchy only shows you the dictionary structures that make up the interface, not the single fields. The names of these structures are derived from the ABAP Data Dictionary.

Don't worry, these single fields have not been forgotten! You will see them later when you come to use this model object in the component controller's context.

Figure 13: Structure of Generated Model. Part 2

```
FUNCTION BAPI_FLIGHT_GETLIST.
*"----------------------------------------------------------------------
*"*"Lokale Schnittstelle:
*"  IMPORTING
*"     VALUE(AIRLINE) LIKE  BAPISFLKEY-AIRLINEID OPTIONAL
*"     VALUE(DESTINATION_FROM) LIKE  BAPISFLDST STRUCTURE  BAPISFLDST
*"       OPTIONAL
*"     VALUE(DESTINATION_TO) LIKE  BAPISFLDST STRUCTURE  BAPISFLDST
*"       OPTIONAL
*"     VALUE(MAX_ROWS) LIKE  BAPISFLAUX-BAPIMAXROW OPTIONAL
*"  TABLES
*"      DATE_RANGE STRUCTURE  BAPISFLDRA OPTIONAL
*"      EXTENSION_IN STRUCTURE  BAPIPAREX OPTIONAL
*"      FLIGHT_LIST STRUCTURE  BAPISFLDAT OPTIONAL
*"      EXTENSION_OUT STRUCTURE  BAPIPAREX OPTIONAL
*"      RETURN STRUCTURE  BAPIRET2 OPTIONAL
*"----------------------------------------------------------------------
```

Textbox 1: Interface for the BAPI_FLIGHT_GETLIST Function Module

However, each of the `TABLES` parameters is defined by a dictionary structure. The names of these dictionary structures become the names of the Java classes generated by the proxy generator.

In general, when an interface to any remote callable function module is created by the proxy generator, the result will be an `<RFCname>_input` class, an `<RFCname>_output` class and then as many structure classes as there are unique `TABLES`, `IMPORTING` or `EXPORTING` structures.[1]

Here, there are 5 `TABLES` parameters, and 4 `IMPORTING` parameters. Therefore, you have a total of 9 extra structure classes in addition to the `<RFCname>_input` and `<RFCname>_output` classes. These structure classes will be named after the corresponding SAP data dictionary structures, not the name of the parameters they define in the BAPI interface.

You will also notice that the `<RFCname>_input` class appears to have an extra parameter called `Output`. This is the standard mechanism for obtaining the results of an RFC call. The `Output` parameter is of type `<RFCname>_output`, and an instance of this class is created by executing the

---

[1] Another Java class is created called `<RFCname>_porttype`, but it is only used internally.

BAPI. The execute() method belongs to the <RFCname>_input class, and when this method is called, an instance of the <RFCname>_output is created.

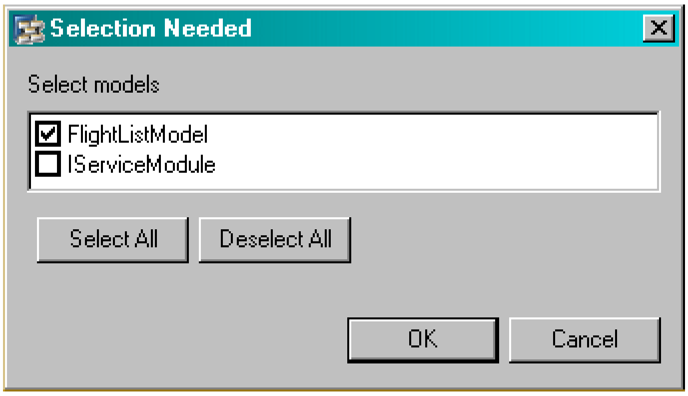
The reason for this apparent duplication is that the TABLES parameters of a function module permit two way data transfer. You can enter data into a TABLES parameter and pass it to a function module as input, but then the function module can update the table and send a modified version back as output. Therefore, an input and output version of every TABLES parameter could exist. To preserve the before and after images of the TABLES parameters, separate classes exist for input and output, each holding their own version of the TABLES parameters.

If you compare the list of parameters to the <RFCname>_input and <RFCname>_output classes, you will notice both similarities and discrepancies. The <RFCname>_input class has as children the IMPORTING and TABLES parameters; whereas the <RFCname>_output class has the EXPORTING and TABLES parameters. Since the TABLES parameters exist on both the input and output side of the interface, a naming conflict will arise due to duplicates parameters names, but this will be dealt with later.

Declaring the Use of a Model

A model is not simply a small part of a Web Dynpro component. It is an independent development object. The use of a model must be declared explicitly within a component.

Now that you have created the model object,[2] any component in the current project can use it.



1. Underneath the FlightList-Component tree node, click the Used models node with the alternate mouse button.
2. Choose *Add*.
3. Select the FlightListModel model as shown in Figure 14 and press *OK*. You can ignore the **IServiceModule** model.

Figure 14: Defining the model usage

Now the model's metadata is available to all controllers in the current component. This can be seen by the presence of a new entry under the Used Models node of your component. See Figure 15.

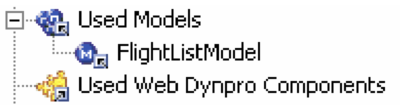


Figure 15: Model Usage in a Component

[2] As stated earlier, the model can be created at any time prior to it being needed by a component

Preparing the Component Controller

As mentioned earlier the component controller is the central point of control for all the functionality found within a Web Dynpro component.

In this example, you will be calling a BAPI by means of a model object. A component controller has no view associated with it, so to display the data obtained from the model, you will code the component in such a way that this data travels through the context of the component controller to the context of a view controller. The view then visualizes the data.

What is a Context?

All controllers, whether view controllers, custom controllers or component controllers, have a dynamic data storage area known as a context. The context consists of a basic static structure, to which you can add both data and metadata at runtime. The actual context structure obtained at runtime is dependent upon the declarations you make at design time.

The context holds both metadata and actual data. However, the context is not a passive data storage area; rather it is a highly dynamic, hierarchical data storage class whose metadata is defined at design time, but whose actual contents (and possibly even extra metadata) are not known until runtime.[3]

You can think of the context as the central, dynamic class that supplies a controller with both the actual data upon which it acts, and the metadata to describe that data.

Context Structure

The context is a hierarchy consisting of two basic types of entity – nodes and attributes. The only difference between the two is that a node may have children, and an attribute may not.

Nodes and attributes that have the context root node as their immediate parent are referred to as *independent*. Nodes and attributes that have some other node as their parent are referred to as *dependent*.

The difference between independent and dependent nodes does not feature in this example. But in complex situations where a child component must map one of its context nodes to a node living in a parent component, the difference becomes significant.

Design Time/Runtime Differences

At design time, you define the metadata that describes what the actual data will look like. However, this actual data (described by the metadata) is only available at runtime. Based on the declarations made at design time, the SAP NetWeaver Developer Studio generates the appropriate methods in your context that allow the creation of actual data.

It is crucial that you understand the difference between the structure of the context at design time and its structure at runtime.

---

[3] It is perfectly possible to modify the structure of the context at runtime, but this requires a detailed understanding of the Web Dynpro Runtime programming model. This is a more advanced form of Web Dynpro programming, and is therefore beyond the scope of this document.

Using the Model Object in the Component Controller's Context

Double-click on the component controller node in the project tree. This loads that controller into the editor which appears in the top right of the SAP NetWeaver Developer Studio.

Notice that the tabstrip at the bottom of Figure 16 below contains no *Layout* tab. Component controllers never have any direct visualisation, but in accordance with the MVC design philosophy, delegate this functionality to a view controller. Click on the *Context* tab and you will see the screen shown in Figure 16 below.

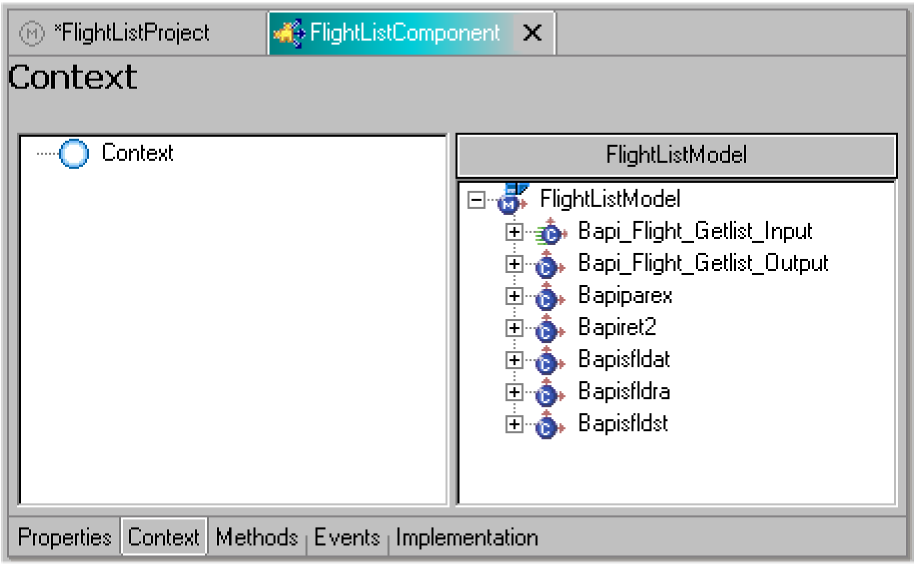


Figure 16: Component Editor - context

The screen is split into two halves. On the left is the as yet, undefined context for the component controller, and on the right, by virtue of your declared model usage, the metadata from the model. If you delete the usage of model `FlightListModel` under `Used Models`, then when you next call up this window, the right hand half will have disappeared.

According to the MVC design philosophy, a view should only be concerned with displaying data, and then handling the immediate user response. An MVC component, on the other hand, is designed to act as the interface layer between the view and the model. Therefore, although it is technically possible, it is not good design practice to have a Web Dynpro view controller making direct use of a model. A much more stable and understandable architecture is achieved if you only make use of models from within component or custom controllers.

Therefore, although the data from the model object is ultimately going to be displayed by the view controller, we will first hold that data in the component controller.

1. Click on the context root node with the alternate mouse button, and the menu shown in Figure 17 appears.

Figure 17: Context menu

2. Choose *New*. The following sidebar menu appears
3. Select *Model Node*. This is because you are creating a node that obtains its metadata from a model, rather than being user-defined.

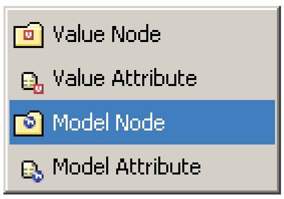


Figure 18: Context sidebar menu

4. Give the context node the same name as the corresponding model node.

You could choose any arbitrary name for the context model node, but to ensure that you don't lose track of the correspondence between the node in the model and the node in the context, it simplifies matters if you give the context node the same name as the corresponding model node.

As a consequence of adopting this naming convention, when you come to look at the generated classes for the context, you will find that the names in the context correspond to the names found in the model interface.

Once you have clicked on *Model Node*, enter the name of the context model node as in Figure 19 below and press *Finish*.

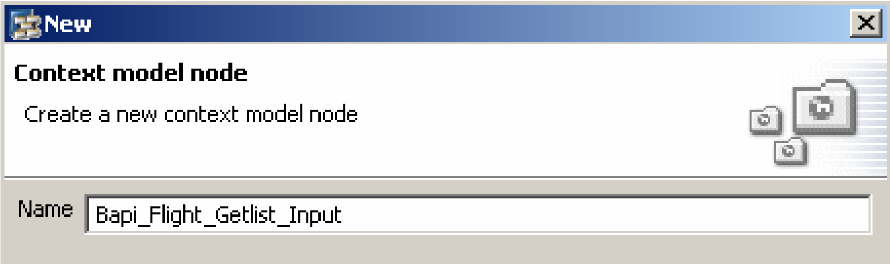


Figure 19: Naming the Model Node

You now have a model node in the component controller's context. However, this merely declares the existence of a node that *can* be associated with the metadata found in a model. You must now manually associate the context model node and the model metadata. This is a process known as model binding. Proceed as follows:

In the context screen, click with the alternate mouse button on the model node that you just created. Choose *Edit Model Binding*.

You will now see the screen shown in Figure 21. The context model node you have just created must correspond to the input side of the BAPI interface defined in the model. Select `Bapi_Flight_Getlist_Input` and press *Next*.

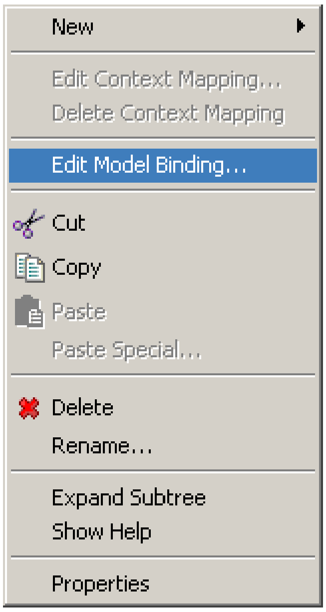


Figure 20: Model Binding

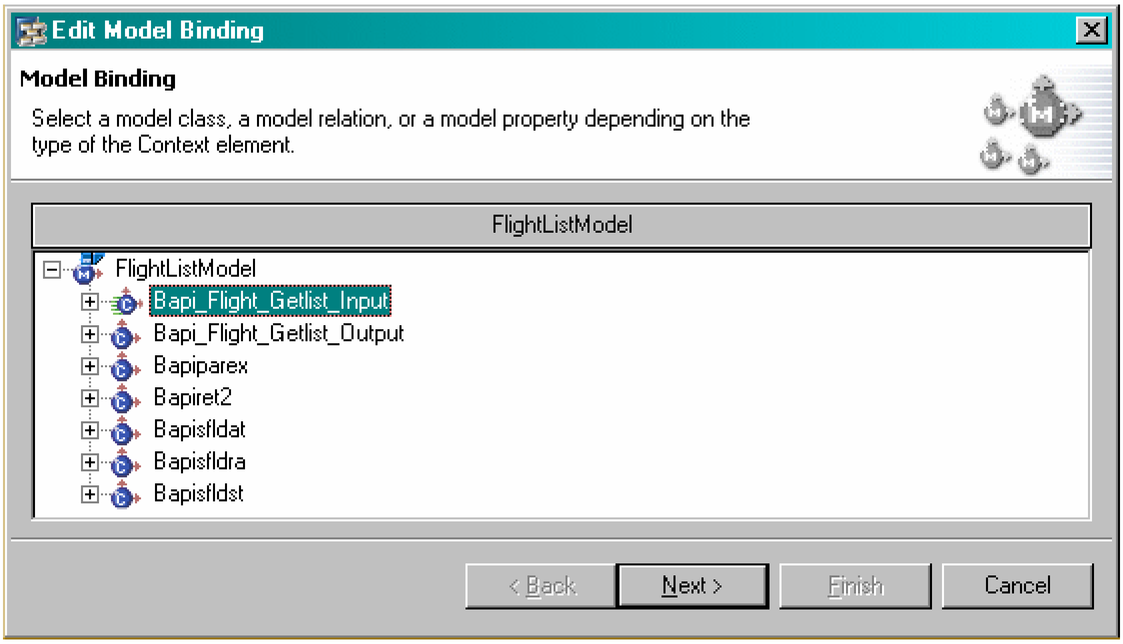


Figure 21: Select Model Binding

One of the design constraints of the context is that all nodes must have unique names - irrespective of their position within the hierarchy. Consequently, the screen is displayed with lots of red error markers all over it. These markers indicate that a name duplication problem exists in the model object hierarchy.

Since TABLES parameters perform two way communication with function modules, their names will occur once as input parameters to the `<RFCname>_input` class, and once again as parameters to the `<RFCname>_output` class. The naming problem exists because the `<RFCname>_output` class is always present as the parameter `Output`, under the `<RFCname>_input` class.

The easiest way to fix this problem is to open the `Output` node on the left hand side of the screen, and then one by one, append `_Output` (or some other suitable text) to the end of each of the node name using the edit fields on the right of the screen.

Once this is complete, the *Edit Model Binding* screen will be as in Figure 22 below.

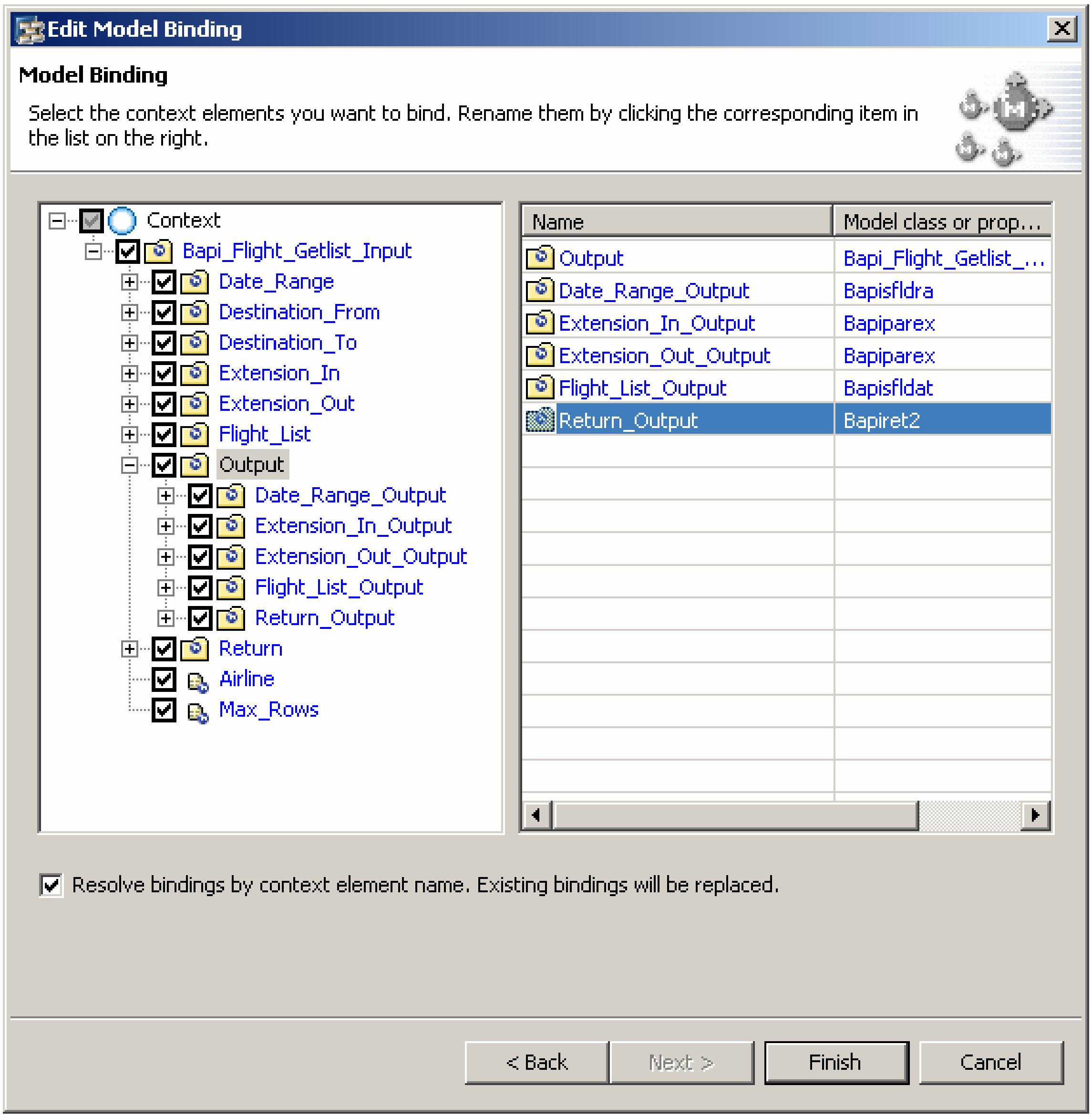


Figure 22: Model Binding After Name Correction

Now that you have resolved all the name conflicts, the *Finish* button becomes active.

Press *Finish*, and you will see that due to the model binding, a duplicate tree structure has appeared under the model node in the component controller's context as in Figure 23 below.

The process of model binding causes the hierarchical metadata found in the model object to be duplicated into the context of the component controller.

Now that the metadata exists in the component controller's context, the Web Dynpro component can manipulate both the input and output parameters of the BAPI simply by manipulating objects in its own context. It does not need to manipulate the model object directly.

The model binding process ensures that data can be transported automatically between the component controller's context and the model object. This applies both to the input parameters travelling from the Web Dynpro component into the BAPI interface, and the output parameters returning from the executed BAPI back into the component controller's context.

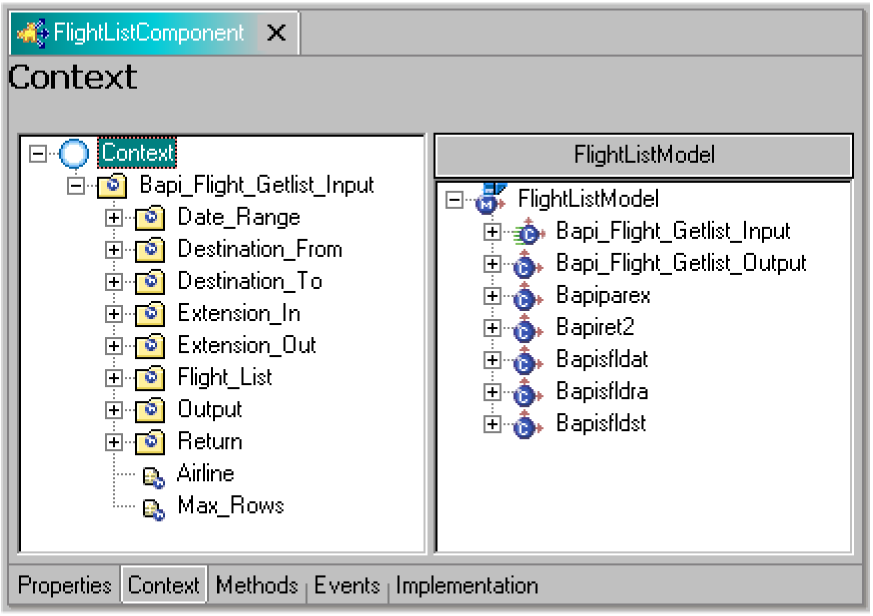


Figure 23: Completed Model Binding

Now you will leave the component controller for the time being, because all the declarative work has been done. However, you must come back to this controller and add some manual coding to complete the required functionality.

Editing the View

You have now come to the point where you have to add fields to the view `FlightListView`.

Double-click on the view name `FlightListView` under either the *Views* node or the *Window* node (both entries refer to the same object).

In the top right of the SAP NetWeaver Developer Studio the view editor appears with 7 tabstrips underneath it. By default, the *Layout* tab is selected when you first edit a view. Figure 24 below shows the View Editor window.

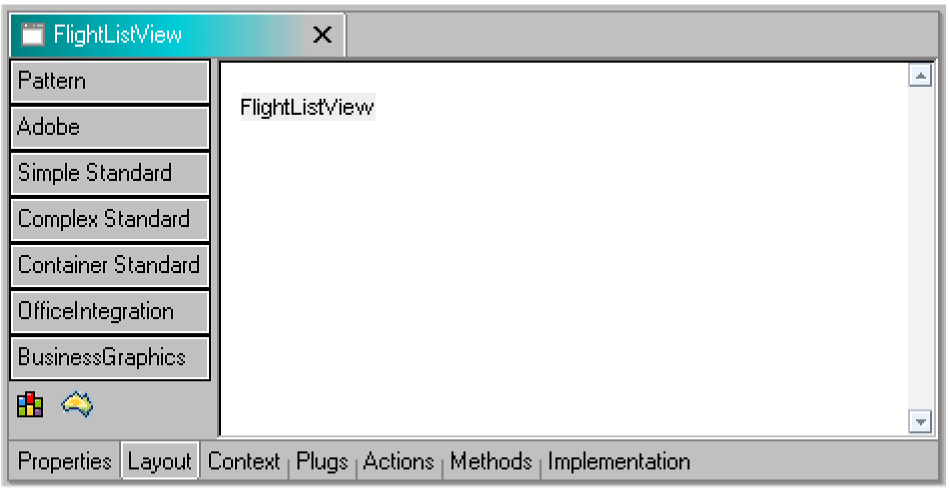


Figure 24: View Layout Editor

As shown in Figure 25 below, the name of the view has been used to create a text view UI element. Such a UI element will always be present in any newly created view. It is there to remind you which view you are looking at!

The View Outline window is in the bottom left of the studio window. Figure 25 below shows the hierarchical arrangement of UI elements in the current view. You do not need the *Default Text View* UI element so you can delete it.

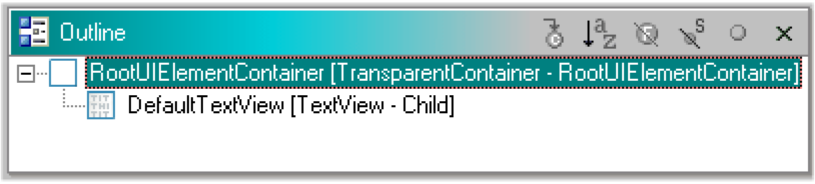


Figure 25: View Outline Editor

Declaring the Use of One Controller by Another

There are two important concepts that you need to understand.

1. A view is a controller in its own right, and possesses a context that is totally independent of the context in the component controller. Unless you say otherwise, there will be ***no*** communication of data between a view controller's context and a component controller's context.

2. A UI element can only be bound to a context node or attribute of the view controller to which it belongs. It is impossible to bind a UI element to a context node or attribute that belongs to a component controller, a custom controller, or some other view controller.

This appears to present a problem. In this example, the component controller's context acts as the link between the functionality of the Web Dynpro component and the business functionality provided by a BAPI call via the model. We need to get the data provided by the BAPI call, out of the model, into the component controller's context, then into the view controller's context, and finally onto the screen in some UI element. The first part of this chain of connections has already been established, but there is one vital link still missing.

In the tabstrip underneath the view editor window, there is a tab called *Properties*. Click here and you will see the screen shown in Figure 26 below. The section entitled *Required Controllers* is currently empty. In order for the component controller and the view controller to communicate with each other, you must add the name of the component controller as a required controller within the view.

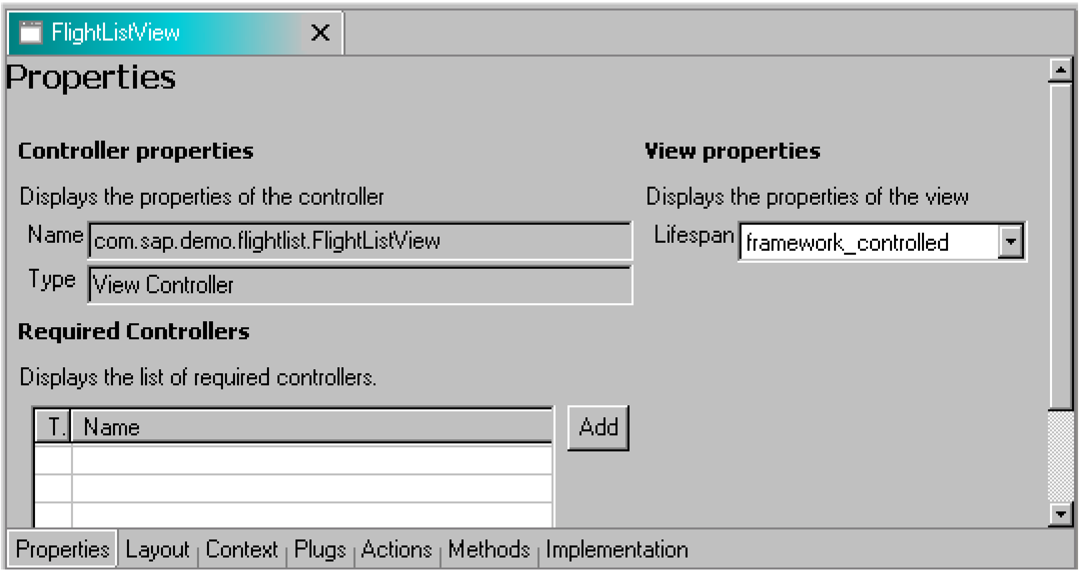


Figure 26: View Properties – Initial Values

Click on the *Add* button and you will see the screen in **Figure 27** below.

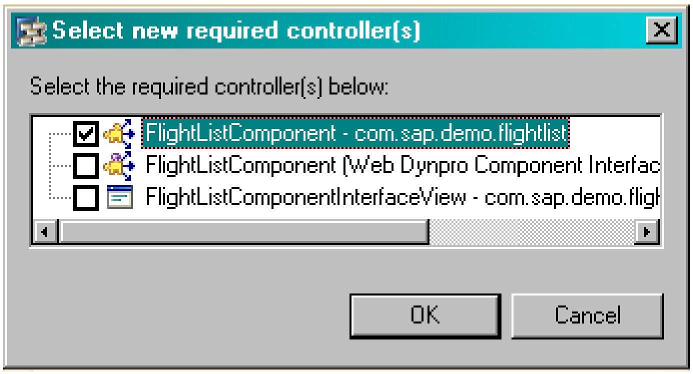


Figure 27: Add Component Controller

Select the component controller as shown in **Figure 27** above and click OK. Now that you have done this, when you click on the view controller's context tab, the hierarchy of the component controller's context will now become available to you.

At this point, all you have said is that the data in the component controller's context is now available to the view controller's context. However, the link has still not actually been established.

This is done through a process known as context mapping.

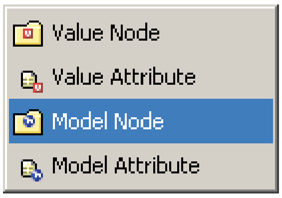
Context Mapping

Due to the model usage declaration in the component controller, the metadata of the model is automatically available to all the subordinate controllers. However, the contents of the component controller's context are not available in a view controller without an explicit declaration. In the previous step, you declared that the component controller's context was available for use in the view controller. However, this merely states that a link *can* exist, not that is *does*.

Since the data you require in the view comes from a model, you must create a model node in the view controller's context. Remember that a model node is simply a context node that inherits its metadata from a model object; whereas a value node is a context node where the metadata is user-defined.

Click on the *Context* node with the alternate mouse button to display the menu shown above in Figure 28 below.

Figure 28: Context menu



You need to create a model node with the same name as the model node in the component controller, that is, `Bapi_Flight_Getlist_Input`. To do this, choose *Model Node* from the *New* sidebar menu.

Figure 29: Context sidebar menu

There is a difference now between creating a model node in the component controller and creating a model node in the view controller. Click the alternate mouse button on the model node you have just created to display the context menu. Now compare the context menu you see on the screen with the one you saw when editing the component controller.

The context menu for the context in the view controller now has the *Edit Context Mapping* option activated. This option only becomes available when the context of another controller is available to map to. Since you have declared that the component controller will be used by the view controller, everything defined in the component controller's context is now available to the view controller's context.

Now it is possible to establish a mapping connection between the model node you have just created in the view controller, and the model interface via the component controller's context.

Select *Edit Context Mapping*. This time you are making a connection with a node that lives in another context, not a node that lives in a model.

Connections between contexts are known as mappings, whereas connections between a controller context and a model, or a UI element and a view controller's context are known as bindings.

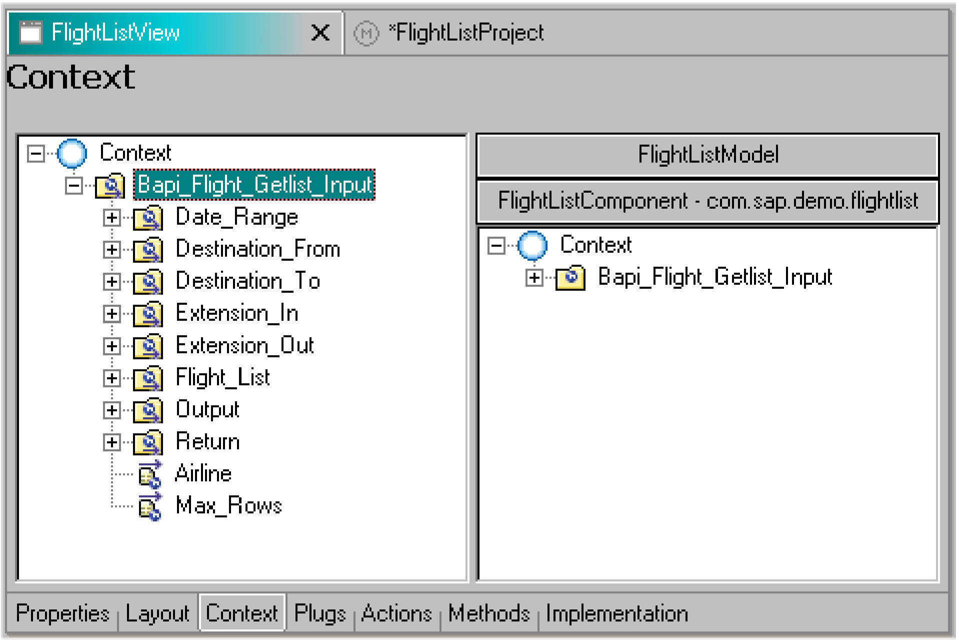


Figure 30: Context selection

Earlier, during the model binding process, you had to correct the names of the sub-nodes in the component controller context. Now that this one-off task has been completed, the node names inherited by the view controller through the mapping process will have the corrected names.

Select the node *Bapi_Flight_Getlist_Input* and choose *Next.* In the next dialog box, select all the checkboxes and choose *Finish*.

As a consequence of this mapping declaration, your view controller now has automatic access to any data found in the model interface. The transport of data between the model and the view is handled by a single method call which will be dealt with later.

Adding a Value Node and Value Attributes to the Context

Since your application allows you to call `BAPI_FLIGHT_GETLIST` multiple times, you must make sure that the user input from the previous BAPI call does not interfere with the input values for the current BAPI call. This only affects values being placed into BAPI `TABLES` parameters. The interface you are going to build will provide the user with four input fields: `CityFrom`, `CityTo`, `DateFrom` and `DateTo`. The `CityFrom` and `CityTo` fields correspond to the BAPI `IMPORTING` parameters `DESTINATION_FROM-CITY` and `DESTINATION_TO-CITY`. Web Dynpro treats `IMPORTING` structure as context nodes with a cardinality of `0..1`. This means that there can be at most, one element in the node collection. Therefore, for `IMPORTING` parameters, you just have to create a single element and each time the user enters an input value, it will overwrite the previous value.

With `TABLES` parameters however (particularly `SELECT-OPTION` tables), Web Dynpro represents these parameters as nodes with a cardinality of `0..n`. In other words, any number of elements are permitted in the collection. For the `DateFrom` and `DateTo` parameters, these corresponding to the `TABLES` parameter fields `DATE_RANGE-LOW` and `DATE_RANGE-HIGH` respectively. Therefore, any node elements holding old parameter values need to be removed before new elements are added.

Choose the  *Context* node with the alternate mouse button and in the context menu choose *New -> Value Node*. Enter the name `InputValues` and press *Finish*.

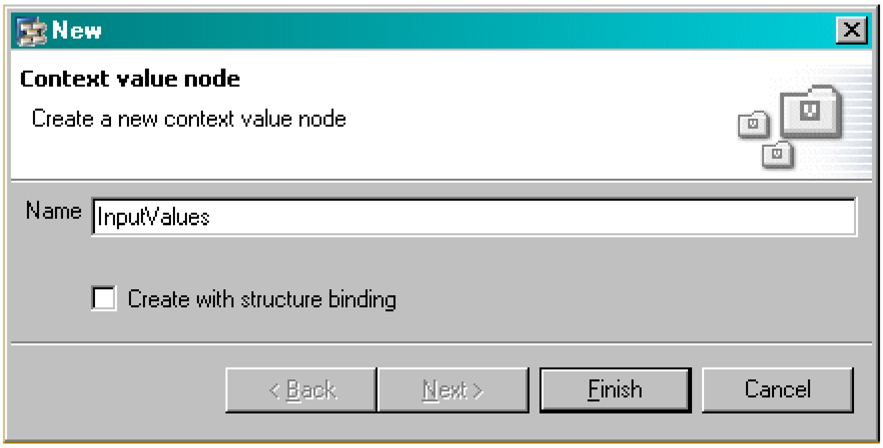
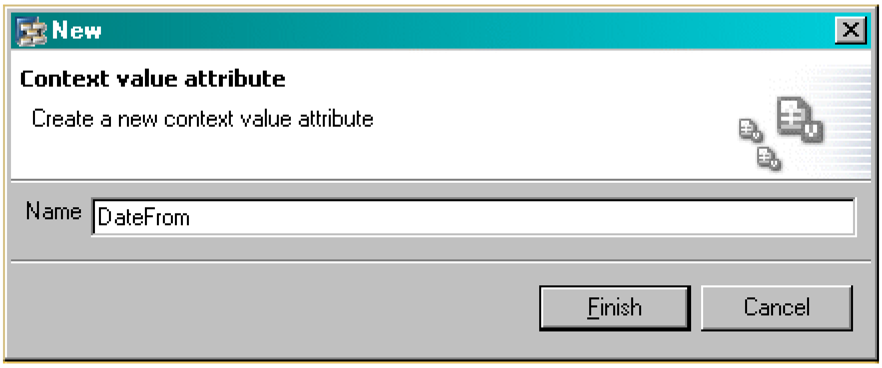


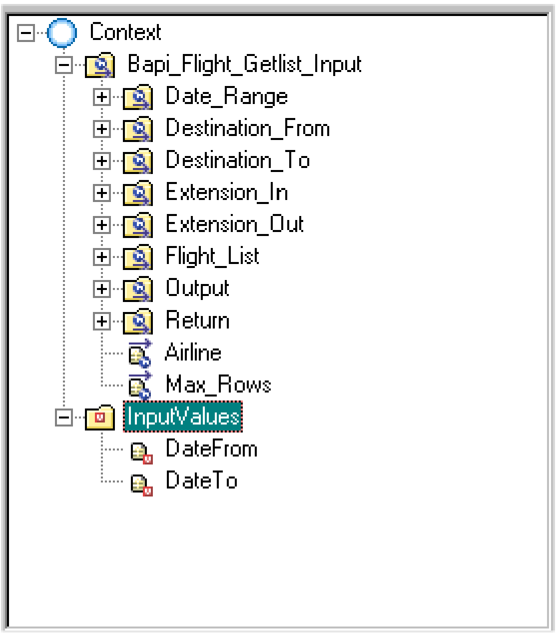
Figure 31: Add context Value Node

Having created the value node, make sure it is selected in the context window, and change the `cardinality` property to `1..1`.



Click the `InputValues` value node with the alternate mouse button and in the context menu choose *New -> Value Attribute*. Enter the name `DateFrom` in the name field as shown in Figure 32 and choose *Finish*.

Figure 32: Add Context Value Attribute



Repeat the above step for the field `DateTo`.

Your context should now look like Figure 33.

When you add attributes to a context node, unless you say otherwise, the data type of these attributes is `string`. This would work for the `DateFrom` and `DateTo` fields, but you would have to convert the `String` value to a `Date` value. It is much easier however, to take advantage of the built in data type `date`.

If you say that a context attribute is of type `date`, then any input field bound to that context attribute will inherit a date picker pop-up window, and any input to that field will automatically be checked to see if it is a valid date.

You don't have to code this.

Figure 33: Finished context for View Controller

Removing the Default Text View UI Element

The default text view element can be deleted in one of two ways. You can either:

- Click on the name `FlightListView` in the view editor layout window and press *Delete* or
- Click with the alternate mouse button on the name `DefaultTextView [TextView]` in the view outline window and select *Delete* from the context menu shown in Figure 34.

Either way, this default text view needs to be removed.

Figure 34: UI Element Menu

Adding UI Elements to a View

Return to the layout editor by clicking on the tab *Layout*. There are two ways of adding UI elements to a view.

Adding UI Elements Using the Outline Window

If you refer back to Figure 25, you will see that the root UI element in the view outline window is called `RootUIElementContainer`. You are going to create two input fields, each with an identifying label field. With the alternate mouse button, click on this element and select *Insert Child*.

The screen shown in Figure 35 is asking for the name of the UI element and its type. There is no mandatory naming convention required here, but it is good practice to use a descriptive name, followed by the UI element type. For instance, if you are creating UI elements for a field called `SalesOrderDate`, then the input field should be called `SalesOrderDateInput` and the corresponding label field should be called `SalesOrderDateLabel`. This will make programming easier because the generated classes and methods associated with these fields will now have readily identifiable names.

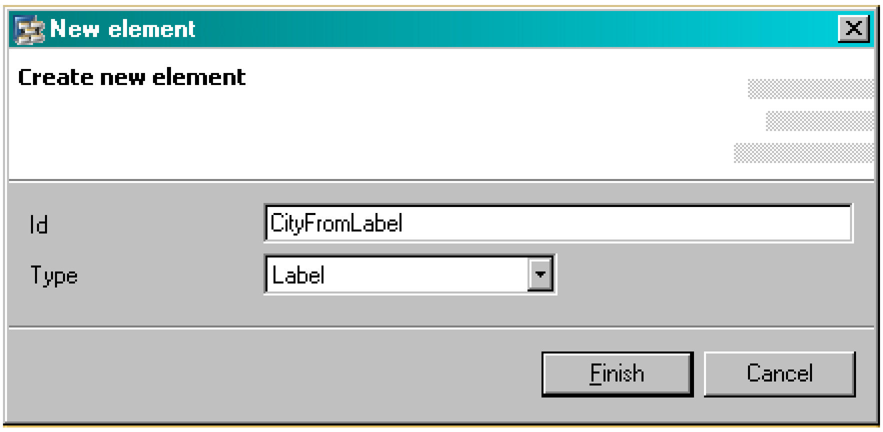
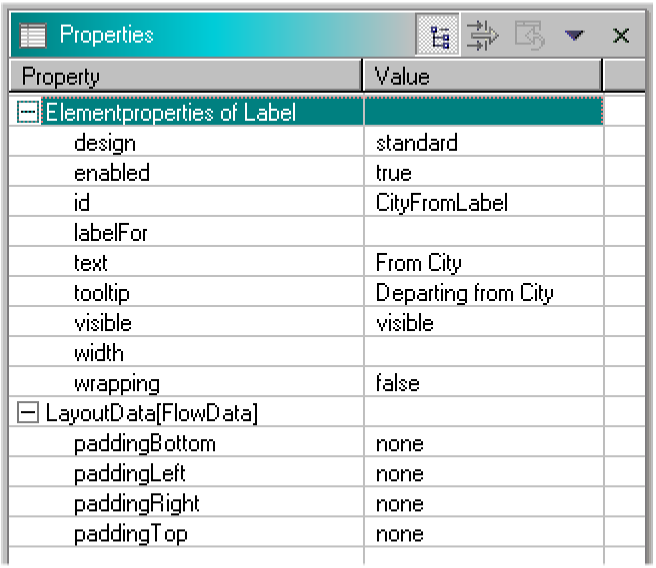


Figure 35: Add New UI Element

Using the outline window to add UI elements, we will add an input and label field for the context attribute *CityFrom*. Therefore, the input field will be called `CityFromInput`, and the label field will be called `CityFromLabel`.



Properties

Property	Value
Elementproperties of Label	
design	standard
enabled	true
id	CityFromLabel
labelFor	
text	From City
tooltip	Departing from City
visible	visible
width	
wrapping	false
LayoutData[FlowData]	
paddingBottom	none
paddingLeft	none
paddingRight	none
paddingTop	none

Figure 36: Properties of label UI Element

Once you have set the field to be of type label, press *Finish*. You will now see the *Label* field in the layout window of the view editor. Immediately after the label field has been added, it will appear in the layout editor, but it will have no text associated with it. In order to define the text, you must do the following.

Select the label `UI element`, and select the properties tab in the bottom right pane of the IDE. You will now see the properties of the currently selected UI element. Change the `text` property to read `From City`, the tooltip to read `Departing from City`, and set the design property to `light`. The properties of this label will now look like Figure 36.

Adding UI Elements Using the Graphical Layout Editor

Alternatively, you can add UI elements using the drag and drop tools in the graphical layout editor. Here, you will add the input field for the *CityFrom* context attribute using these tools.

1. Open the required set of tool bar buttons by clicking on *Simple Standard* on the left of the graphical layout editor.
2. Click on the *Standard~InputField* button.
3. After you have clicked on the input field button, drag and drop the UI element to the right of the label field. As you drag the input field towards the label field, you will see a small box appear in the bottom right hand corner of the label field. This indicates that the input field will appear to the right of the label field.
4. Since the input field was generated automatically, you were not given the opportunity to give the field a name. Select `id` property and rename the field to `CityFromInput`. For the time being, ignore the error indicator against the `Value` property.
5. Now that you have a label/input pair of fields, you can go back to the label field and set the value of its `labelFor` property to `CityFromInput`. In your example, setting this property has no particular effect on the functionality of the view, but it is a good habit to get into. The `labelFor` property tells the Web Dynpro runtime that a particular label field is being used to describe an input field.

Repeat the above steps using either editing technique and add four more label/input pairs; a pair for the fields `CityTo`, `DateFrom`, `DateTo`, and lastly `ReturnMessage`.

Make sure that the `Id` properties of these fields have the correct values using the naming convention described above. Also, change the `text` and `tooltip` properties of the label fields to something appropriate. The one exception here is that the field `ReturnMessage` will be an output field. Therefore, end the name of the input UI element with `Output` and set the `readOnly` property to **`true`**.

Link the corresponding label and input fields together by means of the label's `labelFor` property.

After you have completed these steps, your layout window should look like Figure 37 below. The horizontal alignment of the UI elements will vary depending on the width of the view editor window in your IDE.

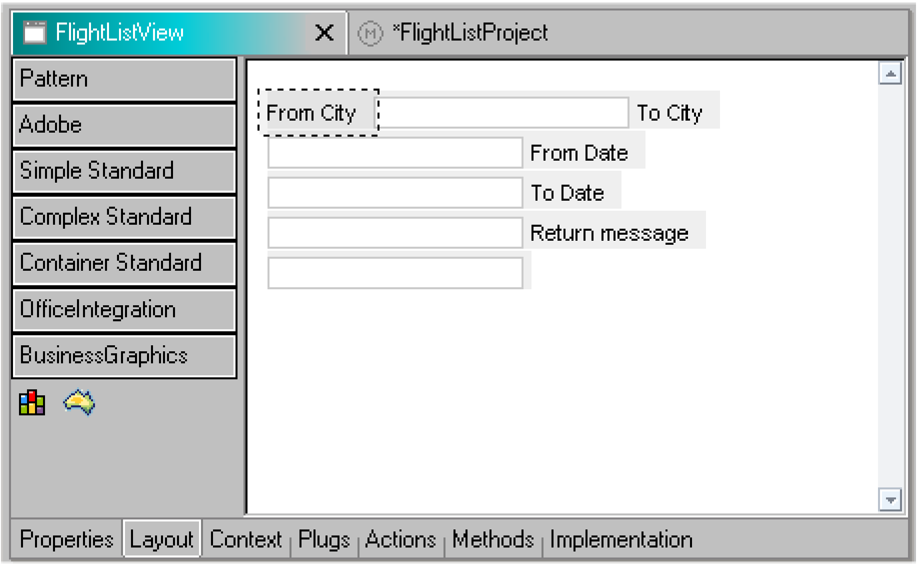


Figure 37: Label and Input Fields, Part 1

The fields are arranged horizontally across the screen. This is because the parent UI element for the entire view (which is **always** called `RootUIElementContainer` and is **always** of type `TransparentContainer`) has its `layout` property set to `FlowLayout` by default.

Change the layout as follows:

1. Select the `RootUIElementContainer` node in the outline window.
2. In the properties window, change the `layout` property from `FlowLayout` to `GridLayout`. A new set of properties appears that relates only to `GridLayout` containers.
3. Change the `colCount` property from 1 to 4 and press Enter. The fields will now be arranged in the correct manner.
4. Since the `ReturnMessage` field in the BAPI interface is so long (220 bytes), it will be fine to set the length field to a smaller value such as 70. This will do nothing more than truncate the displayed data.
5. Change the `colspan` property of the `ReturnMessage` field to 3 so that the field stretches across three columns.

Once these steps have been completed, the layout window should look like Figure 38 below.

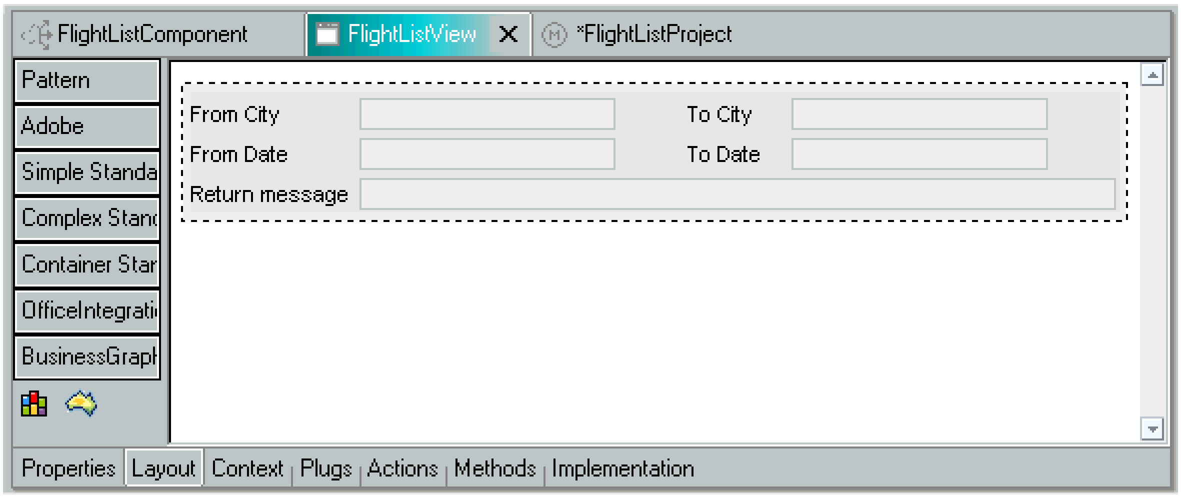


Figure 38: Label and Input Fields, Part 2

Adding a Button UI Element

Button UI elements are the simplest means of telling the Web Dynpro runtime that it should respond to some sort of user action. In this case, you want to call a BAPI when the user presses the **Execute** button.

1. Return to the Outline window.
2. Click `RootUIElementContainer` with the alternate mouse button.
3. Choose *Insert Child* from the dialog box.
4. Enter the name `FlightListButton` and change the type to **`Button`**.

Having added the button, you now have to make sure that the Web Dynpro runtime knows how to respond when this button is pushed. Therefore, you must create an action. Once you have created this action, you will associate it with the button UI element. Once you have made this association the Web Dynpro runtime will invoke the functionality contained within the action when the button is pressed.

At the bottom of the view editor pane, there is a tabstrip with seven tabs. Click on the tab *Actions*. Click here. You can see that currently, no actions have been defined for this view controller. Click on the button *New* to the right of the table of actions.

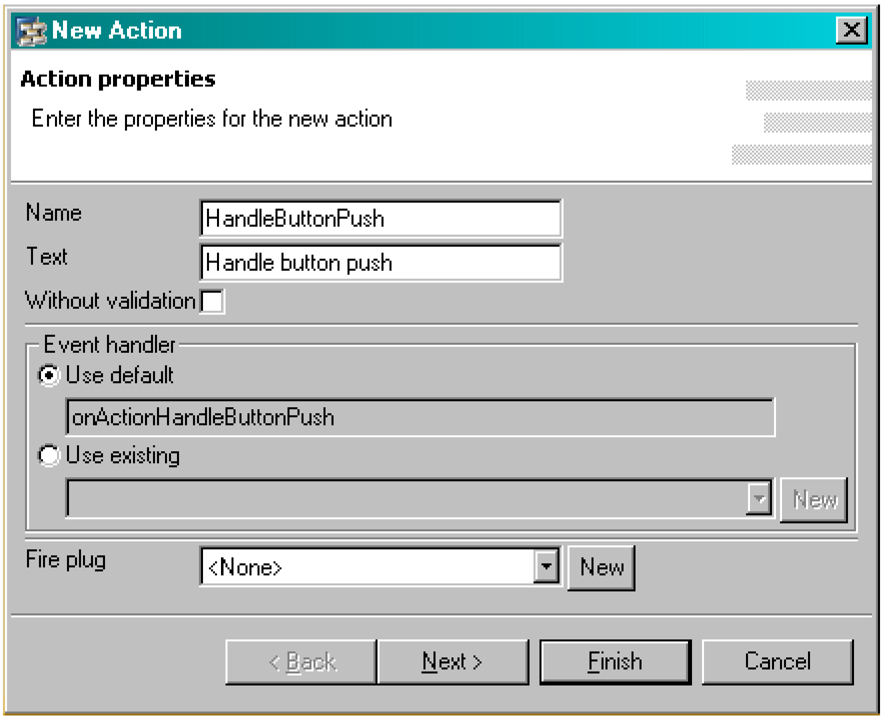


Figure 39: Define Action Properties

Notice that the action is named after the event that triggers it, not the functionality that results as a consequence of its execution. The reason for this is that actions can be made generic. It is perfectly possible to assign the same action to multiple UI elements. This means that the action handler must behave in a generic manner. Although this example only uses one action, it is good to form the habit of naming the action after the type of event that triggers it, rather than the functionality that occurs.

Fill in the action screen as shown in Figure 39 above and press *Finish*.

Now that you have created the action, you can assign it to the button.

1. Return to the layout view.
2. Select the button.
3. Find the property `onAction` under the `Event` node in the properties window.

4. Click in the value column, and three different buttons will appear on the right.
5. Click on the down arrow button. The action you have just created will appear in a dropdown list.
6. Select the action HandleButtonPush and press Enter.

Notice that the text you assigned to the action automatically becomes the button text. Since you have created a generic action, this text needs to be overridden. Click on the text property and add the text Get Flight List.

Even though you have defined an action and associated it with the button, there is no declarative way of specifying what functionality should take place when the button is pushed. Therefore, you will have to code the event handler manually. All Web Dynpro can do for you in this situation is to create a standard method called onAction<action_name> within which you must place your code.

Adding an Invisible UI Element

Since you have specified that the RootUIElementContainer should use the GridLayout layout manager and configured it to have 4 columns, you need to ensure that all UI elements occur in horizontal groups of 4. You have just inserted a button into column 1 of the grid. Anything added subsequently will automatically be placed into column two. The next visible UI element you will add is the table to hold the search results, but you don't want it to appear in column 2, so you must fill columns 2 to 4 with an invisible element.

1. Go to the Outline window.
2. Click RootUIElementContainer with the alternate mouse button.
3. Select *Insert Child*.
4. Enter the name **InvisibleElement1**.
5. Select the element type InvisibleElement.
6. Change the value of the visible property to none.
7. Change the colspan property to 3.

Adding a Table UI Element

The next UI element to add is a table. You can either use the outline menu or the graphical drag or drop editor. If you wish to use the drag and drop editor, then you must select the toolbar group called *Complex Standard* in order to find the table button.

After you have added the table UI element, you will need to change some of its properties.

First, because you have switched the Layout attribute of the RootUIElementContainer to GridLayout, and set the columns property to 4, the table you have just added will appear only in the left hand column of the grid layout. Therefore, change the colspan property of the table UI element from 1 to 4.

If you added the table using the drag and drop editor, then you will also need to rename the table's Id property to FlightListTable.

Save your work using the  icon on the toolbar.

Binding UI Elements to the View Controller's Context

To display a value or a set of values in a UI element, that UI element must be bound to a corresponding attribute or node in the view controller's context.

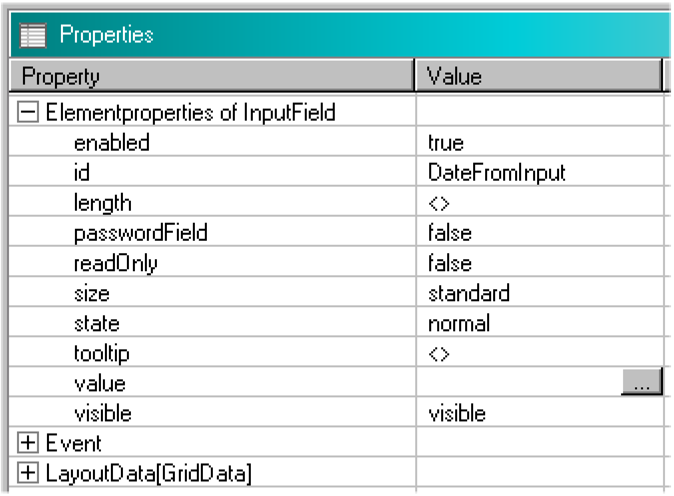
`Important:` You cannot bind a UI element to the context of any controller other than the view controller to which the UI element belongs.

The binding process is the vital declarative step that ensures your UI element will be able to communicate with the Web Dynpro runtime. Once a UI element's binding has been declared, no further action on the part of the programmer is required to transport the data from the context to the screen and back again. These tasks are handled automatically.

You currently have six UI elements on the screen whose bindings need to be declared; four input fields, one output field, and a table.

You do not have to bind the pushbutton in the same way as an input field because the purpose of the button is to trigger an event, rather than to display data.

Binding an Input Field UI Element



You may have noticed that the `Value` properties of the input fields have a red error indicator against them. This means that the UI element is currently unbound. If you were to leave it in this state, that UI element would be unusable. The Web Dynpro IDE does not permit such a situation and the Web Dynpro application would not build successfully.

To remove this error, select the `DateFromInput` input field UI element, and then click in the value column of the `Value` attribute in the properties window. A small button appears to the right of the value field with an ellipsis on it.

Figure 40: Define Input Field context Value

Click on the ellipsis button. The following hierarchy of the view controller's context is displayed.

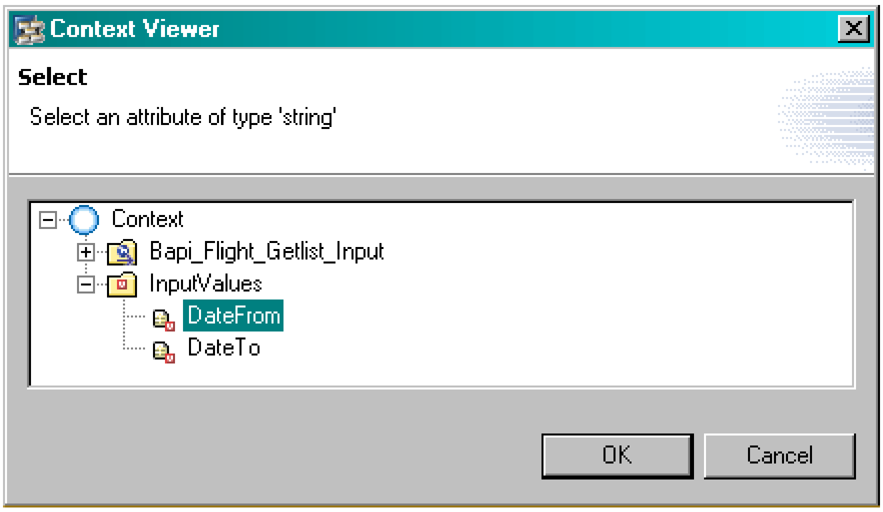


Figure 41: Bind Input UI Element to the context

The nodes and attributes displayed here belong to the view controller, and are the only ones to which UI elements can be bound. The OK button will remain inactive until an attribute in the context is selected that is compatible with a UI element of type **`Input Field`**. In this case, you need to bind the input field to the `DateFrom` field underneath the `InputValues` node.

When the binding is declared correctly, there will be a green tick next to the value assignment.

Repeat the above procedure for the `DateToInput` field. You must bind the `ReturnMessageOutput` field to the `Message` attribute underneath the `Output.Return_Output` context node.

Under the node `Bapi_Flight_Getlist_Input` are the nodes `Destination_To` and `Destination_From`. Bind the `CityTo` and `CityFrom` fields to the respective `City` attributes.

Binding a Table UI Element

Binding a table UI element is more involved than the single binding declaration required for an input UI element. This is because a table has multiple columns, each of which needs its own binding declaration.

Important: The different columns of a table UI element should only ever be bound to attributes that belong to the *same* context node. Do not attempt to bind different columns a table to attributes belonging to *different* context elements – the results could be chaotic!

First, select the table UI element either by clicking on its name in the Outline window, or by clicking on the top of the table area in the graphical editor window. Once the table UI element is selected, the properties window in the bottom right will show the properties of that object.

Instead of having a `value` property, table UI elements have a `dataSource` property. This defines the superset of columns from which the table will obtain its data.

In contrast to an input field UI element, a table's `dataSource` attribute must be bound to a *node* in the view controller's context, not an *attribute*. This is because there is a one-to-one relationship between a context node and the entire table UI element.

Each element in the node collection corresponds to a table row, and each attribute within the node element corresponds to a table column.

Click on the value column of the `dataSource` property and an ellipsis button will appear.

Click on the button and a dialogue box will appear displaying the view controller's context.

Since you are performing the binding for a table UI element, and not a simple input field UI element, you must select a context node, not a context attribute.

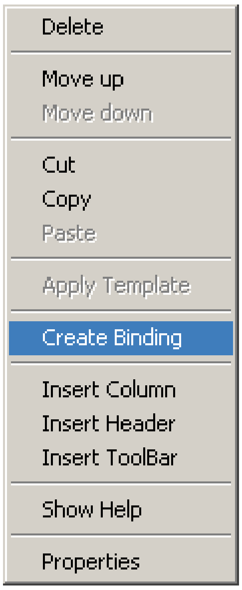


Figure 42: Context menu

Since function modules can both receive input and send output results in the same TABLES parameters, you must make sure you bind the table UI element to the correct side of the interface. That is, the `FlightListTable` as it will be *after* the BAPI call has been executed, not before.

Therefore, you must bind the table UI element to the context node called `Flight_List_Output` that occurs under the `Output` node of the context hierarchy.

Declaring the `dataSource` of the table UI element is only the first part of the binding declaration. You now have to declare the bindings for the individual table columns and you can only do this in the outline window.

Right click on the `FlightListTable` entry in the outline hierarchy, and click on the *Create Binding* entry as shown in Figure 42.

If you click on the *Create Binding* option *before* a value for the `dataSource` property has been defined a dialog box with an empty table view is displayed.

1. To enter a `dataSource` value, click on the ellipsis button. A dialog box will appear. Specify the context node as described above.
2. Select `Flight_List_Output` from the list and make sure that the subordinate checkboxes are all selected. Press *Finish*.

All the attributes in the `Flight_List_Output` context node now appear as columns in the `FlightListTable` as shown in Figure 43 below.

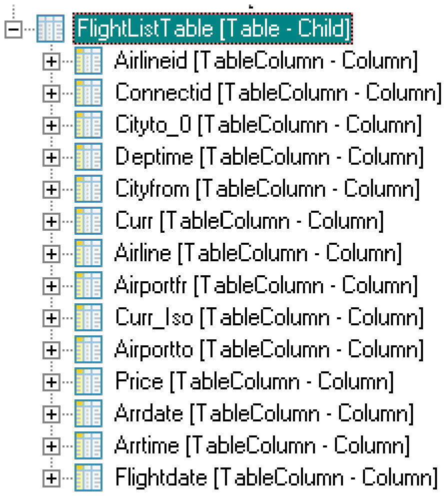


Figure 43: Outline View after Table Binding

Save your work using the  icon on the toolbar.

You have now completed all the declarative design steps. The remainder of the required functionality must be coded manually.

Java Programming with Web Dynpro

So far you have been working in an entirely declarative manner. That is, you have only defined the relationships that exist between the various elements of a Web Dynpro component. You have not had to write any Java code.

However, you have now reached the point in the design where all the declarative work has been done. This has eliminated at least 80% of the coding effort. However, there are certain remaining tasks that you cannot specify in a declarative manner. For instance, you cannot state declaratively what should happen when the user presses a pushbutton. Nor can you declare the creation of BAPI input fields for your search criteria.[4]

You have to carry out these coding tasks manually.

Structure of the Generated Code

In this example, you will only deal with the coding that is generated for the component controller, the view controller and their respective contexts. Coding that exists in the other controllers will not be discussed in this document.

In general, there are only two types of controller: view controllers and custom controllers. A view controller is concerned solely with handling data that is to be visualised on some form of output device, and a custom controller is concerned with all other aspects of component control and interaction. A component controller is a specialised type of custom controller.

To display the generated code for a particular controller, click on the tab *Implementation*. You then see various progress messages in a dialog box telling you what stage the code generator has reached. After a few seconds, you will see the Java editor for that particular controller.

The Context

Each controller uses a set of hierarchically arranged classes known as a context. As previously explained, a context is composed of a base set of known classes and methods, to which are added a set of dynamically generated classes and methods defined according to the declarations made by the programmer at design time. So there is no such thing as a **`standard`** context.

The Relationship between a Controller and its Context

From the point of view of a controller, there is always a one-to-one relationship between itself and its context. In the controller's constructor (which cannot be modified!), you will always find at least the code shown in the coding extract below.

```
private final IPrivate<controllerName> wdThis;
private final IPrivate<controllerName>.IcontextNode wdcontext;

public <controllerName>(IPrivate<controllerName> wdThis) {
  this.wdThis = wdThis;
  this.wdcontext = wdThis.wdGetcontext();
}
```

Textbox 2: Controller Constructor

The constructor of any controller always instantiates the member variables `wdThis()` and `wdcontext()`. These objects are the Web Dynpro self reference and the reference to the context of the current controller respectively. You are not permitted to modify a controller's constructor.

---

[4] This is due to a property of the input fields called cardinality with which you have not yet dealt.

The use of the standard Java self-reference `this`

In any standard Java program, there is always a self reference variable available called `this`. However, in Web Dynpro, the standard self reference `this` exists, but should never normally be used. The reasons for this are beyond the scope of an introductory document. Suffice it to say that you should use always the generated Web Dynpro self reference `wdThis()`.

For a detailed explanation of this point, see the Web Dynpro Programmers Pocket Reference Guide.

The Structure of the Context

The context of any controller always has a root node which acts as the parent node for all other context nodes and attributes. The root node always exists and always contains exactly one element. The child attributes and nodes of the root node element are defined by the declarations you make at design time.[5]

Directly underneath the root node, you may define any number of nodes and/or attributes. As with any hierarchically arranged structure, a node may itself have nodes and attributes as children. There is no predefined limit imposed on the number of levels the context hierarchy may contain.

Several things are important to remember about the context:

- The only difference between a node and an attribute is that a node can have children and an attribute cannot.
- Any node or attribute that has the root node as its immediate parent is referred to as being **independent**; whereas nodes and attributes occurring anywhere else in the context hierarchy are referred to as being **dependent**.
- At design time, the context contains only *metadata*.
- All context nodes are collections – irrespective of the number of elements they are permitted to contain. Therefore, at runtime the node collection could contain multiple elements; thus, the flat, two dimensional context hierarchy seen at design-time takes on a third dimension due to the presence of the node elements.

Structure of the Generated Context Classes and Methods

Since the structure of the context is entirely dependent on the declarations you make at design time, certain conventions have been adopted to make the structure more predictable. For instance, if you know the name of a context node declared at design time, then you can call a method that retrieves the node object directly. This method will have the name of the node embedded within it, and is the normal programming approach for accessing data in the context.

However, in more advanced programming situations, you may have to process the contents of a context whose structure you do not know. Here you can still perform any processing that is required but you have to use the generic interface to the context.

[5] Or in more advanced programming situations, you can define context nodes and attributes dynamically at runtime.

In general, the creation of context node <cn> at design time, leads to the creation of an interface class called I<cn>Node. The following set of methods and classes are created as a consequence of the declarations made at design time. Assume that a context node <cn> has been created that has a context attribute <ca> as its child.

wdcontext()	
↳ .create<cn>Element()	Creates a new collection element for node <cn>
↳ .set<ca>()	Setter method for the node attribute <ca>
↳ .get<ca>()	Getter method for the node attribute <ca>
↳ .current<cn>Element()	Returns the current element of node <cn>
↳ .set<ca>()	Setter method for the current node attribute <ca>
↳ .get<ca>()	Getter method for the current node attribute <ca>
↳ .node<cn>()	Returns a node object for <cn>
↳ .getElementAt(i)	Returns the node element at index i

For instance if you know that your context contains a node called SalesOrders that was declared at design time, then you know that you will be able to create an instance of that node by calling wdContext.nodeSalesOrders().

Knowing that the design time context node is called SalesOrders immediately tells you that the following methods are available from the wdcontext class.

createSalesOrdersElement()	Create a new element for the node SalesOrders
currentSalesOrdersElement()	Get the current element from the node SalesOrders
nodeSalesOrders()	Create a new, empty SalesOrders node

In your case, you are using a BAPI interface supplied to your context via a model. Consequently, almost all of your context nodes and attributes will inherit their metadata (which includes their name) from the interface defined in the model. Therefore, the names defined in the interface for BAPI_FLIGHT_GETLIST and the corresponding names of the interface's data dictionary structures will be the names that appear here as generated methods.

How Does This Work in Your Component Controller's Context?

In your component controller, you have already created a context model node called Bapi_Flight_Getlist_Input and bound it to the model class of the same name. The act of binding these items together causes the model node in the context to obtain its metadata from the model object. In so doing, your context now has a generated method called:

```
wdcontext.nodeBapi_Flight_GetList_Input()
```

A call to this method will return a model object of class Bapi_Flight_Getlist_Input. Notice that the name of the generated method obeys the naming convention described above.

Even though the metadata in the BAPI interface is arranged hierarchically, all the subclasses within Bapi_Flight_Getlist_Input are created as distinct node<contextNodeName> methods directly under wdcontext. This explains why you had to correct all the names of the TABLES parameters on the output of the BAPI interface. Irrespective of their position in the context hierarchy, all context nodes can be created by calling the method wdContext.node<contextNodeName>.

However, the hierarchical relationship of nodes in the context has not been lost. If you go back to the context tab of FlightListView, you will see that node Bapi_Flight_Getlist_Input has a subordinate node called Output, which in turn, has a subordinate node called Flight_List_Output. The hierarchical relationship has been preserved in the fact that from method nodeBapi_Flight_Getlist_Input you can reference the method nodeOutput, which in turn gives access to the method nodeFlight_List_Output. For example, the following coding is possible:

```
wdcontext.nodeBapi_Flight_GetList_Input().nodeOutput().nodeFlight_List_Output()
```

The Cardinality Property

Every context node has a property called `cardinality` and the possible values are listed below in the table below:

Cardinality	Meaning
`0..1`	Zero or one
`0..n`	Zero or more
`1..1`	One and one only
`1..n`	One or more

Table 1: Cardinality values and their meanings

This property is a combination of two values that describe firstly, the number of elements this particular node collection will contain when your application starts, and secondly, how many elements the node collection may have in total.

The first part of the cardinality is stored internally as a boolean value called `Mandatory`, and describes whether or not the existence of element zero in the node collection is mandatory.

The second part of the cardinality is also stored internally as a boolean value called `Multiple` and describes whether or not the node collection may have more than one element.

Taken together, there are four permutations of these values and they are listed above. So a node with a cardinality of `0..n` will start life as an empty collection, but during the lifespan of the program may accumulate as many elements as are required.

Similarly, a node of cardinality `1..1` will be a collection containing exactly one element. You cannot delete this element and you cannot add a new element to the node collection.

It is the convention within Web Dynpro that all model nodes have a cardinality of `0..n`. Therefore, when the program starts, any nodes with this cardinality will be empty - element zero will not exist. Therefore, before any data can be stored in the node collection, element zero must first be created manually.

User Coding Within the Standard Web Dynpro Framework

Within any controller, a set of predefined locations exist where you can add your code. These are known as hook methods and have standard names such as `wdDoInit()` or `wdDoExit()`.

You are not permitted to change either the name or the parameters of these hook methods.

There is always a pair of special comment markers within any generated method, between which you can write code or comments. Any coding outside the `//@@begin` and `//@@end` comment markers will be deleted during the regeneration process.

The coding extract below illustrates how the `wdDoInit()` method will appear immediately after initial code generation.

```
//@@begin javadoc:wdDoInit()
/** Hook method called to initialize controller. */
//@@end
public void wdDoInit() {
  //@@begin wdDoInit()
  //@@end
}
```

Textbox 3: Generated code

What are the Standard Hook Methods For?

When a controller class is created, there are certain standard hook methods that are always present irrespective of whether or not you implement them. The coding extract below shows only the standard methods of a controller class. The class also contains certain other attributes, but these have been omitted because they are not relevant to the current discussion. The following points should be understood about controllers.

- Each controller is a separate, dynamically generated Java class.
- The dynamically generated controller class contains a subordinate dynamic class of type `IPrivate<controller_name>.IContext`. This is the controller's context class as defined by your declarations at design time.
- Controllers normally have only one constructor. Since there are no `//@@begin` and `//@@end` comment markers within the constructor, don't even think about trying to modify it!
- Normally, the Web Dynpro runtime instantiates the controller classes automatically.

```
public FlightListComponent(IPrivateFlightListComponent wdThis) {
  this.wdThis = wdThis;
  this.wdcontext = wdThis.wdGetcontext();
}

public void wdDoInit() {
  //@@begin wdDoInit()
  //@@end
}

public void wdDoExit() {
  //@@begin wdDoExit()
  //@@end
}

//@@begin others
//@@end
}
```

Textbox 4: Methods common to all controllers

The methods `wdDoInit()` and `wdDoExit()` are always present in all controllers and serve the following purposes.

- The method `wdDoInit()` is called immediately after the controller has been instantiated. Preparatory work should be performed in this method.
- The method `wdDoExit()` is called immediately before the controller is garbage collected. You should write code to perform clean up tasks in this method.

Just prior to the class's closing curly bracket, there is a pair of `//@@begin other` and `//@@end` comment markers. Between these comment markers, you are permitted to create entirely new static methods for the controller or to declare class variables. Any variables or methods declared here are completely invisible to the Web Dynpro runtime environment.

The coding extract below shows the methods specific to a component controller.

```
public void wdDoPostProcessing(boolean isCurrentRoot) {
  //@@begin wdDoPostProcessing()
  //@@end
}

public void wdDoBeforeNavigation(boolean isCurrentRoot) {
  //@@begin wdDoBeforeNavigation()
  //@@end
}
```

Textbox 5: Methods specific to a component controller

- The method `wdDoPostProcessing()` is a hook for error handling during context validation.

- The method `wdDoBeforeNavigation()` is a hook that allows you to implement your own navigation processing if you want to override the standard navigation.

The coding extract below shows the methods specific to a view controller.

```
public static void wdDoModifyView(
                     IPrivateFlightListView wdThis,
                     IPrivateFlightListView.IcontextNode wdcontext,
                     com.sap.tc.webdynpro.progmodel.api.IWDView view,
                     boolean firstTime) {
  //@@begin wdDoModifyView
  //@@end
}
```

Textbox 6: Methods specific to a view controller

- The method `wdDoModifyView()` is called immediately before a view is rendered. This is the only place where you can to modify the view's UI elements.

Dynamically-Created View Controller Methods

You have already created a pushbutton UI element and associated it with an action which you called `HandleButtonPush`. As a result of creating this action, a new method is created within the view controller called `onActionHandleButtonPush()` as shown in the coding extract below.

```
public void onActionHandleButtonPush(
              com.sap.tc.webdynpro.progmodel.api.IWDCustomEvent wdEvent) {
  //@@begin onActionGetFlightList(ServerEvent)
  //@@end
}
```

Textbox 7: Dynamically created method

The Web Dynpro runtime will automatically call this method when you press the associated button.

Coding Added to the Component Controller

Logging On and Off the SAP System

In order to gain access to any functionality within an SAP system, a user must provide the SAP system with some valid credentials by means of the log on process. This holds true irrespective of whether a user or a program requires access. 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.

There are two configuration tasks to be performed in order to connect the logical systems defined in the Adaptive RFC model with a physical SAP system.

1. The physical SAP system must first be defined in the System Landscape Directory (SLD). See the J2EE Engine Administration Guide for instructions on how to do this.
2. 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. This task is done in the JCO Connections screen of the J2EE Engine Web Dynpro Content Administrator.

Good Design Practice for Web Dynpro components

As stated in the preface of this document, the Web Dynpro environment is based on the existing Model View Controller (MVC) design philosophy. This design philosophy ensures that the different coding units within your application perform distinct, cooperative roles.

A Web Dynpro component is structured in a way that follows the MVC design philosophy quite closely – but there are a few differences.

Each time you create a component, you are creating a complete MVC triad in its own right. However, implementation of the model part is optional, and the view part is a reference to the visual interface of some other view controller. Also, when you create a view controller, you are creating another MVC triad that is hierarchically below the component controller. Therefore, a view and a component both have their own, distinct controllers.

According to the MVC design philosophy, a view should only be concerned with displaying data on the client, and receiving the subsequent user input. The view itself should not need to derive the data it is to display. This is the controller's job.

Therefore, although it is technically possible, a Web Dynpro view controller should not directly interface to a model. Interaction with a model should take place in a custom controller or the component controller. Only those values required on the user interface should be passed through to the view controller.

You will implement this architecture in this example. The model interface has already been established in the component controller when you bound the context node *Bapi_Flight_Getlist_Input* to the model object. Then when you created the model node in the view controller (also called *Bapi_Flight_Getlist_Input*) and mapped it to the component controller, you established exactly the type of architecture required for good MVC design.

In the next part of this tutorial, you will create a method with the component controller that calls the BAPI, and then you will execute this method from within the action handler method in the view controller. This way, you are not violating good MVC design by calling the BAPI directly from the view controller.[6]

[6] Technically, it is possible to invoke the functionality of any model from any Web Dynpro controller. However, to maintain an architecture that is easier to understand, model functionality should not be invoked from view controllers.

Making use of the Model in the Component Controller's Context

When you created the model node `Bapi_Flight_Getlist_Input`, you associated it with a model object by means of a process called Model Binding. This context node represents the entire input side of the BAPI interface.

Remember that all context nodes are collections, and that model nodes representing either the entire BAPI interface, or TABLES parameters, are collections of cardinality `0..n`. Therefore, when the component controller for `FlightListComponent` is first instantiated, the node collection called `Bapi_Flight_Getlist_Input` will be completely empty. So before we can make use of any of the input parameters to the BAPI itself, we must first create an element for the node `Bapi_Flight_Getlist_Input`.

In order to do this, you must identify the object type of node `Bapi_Flight_Getlist_Input`.

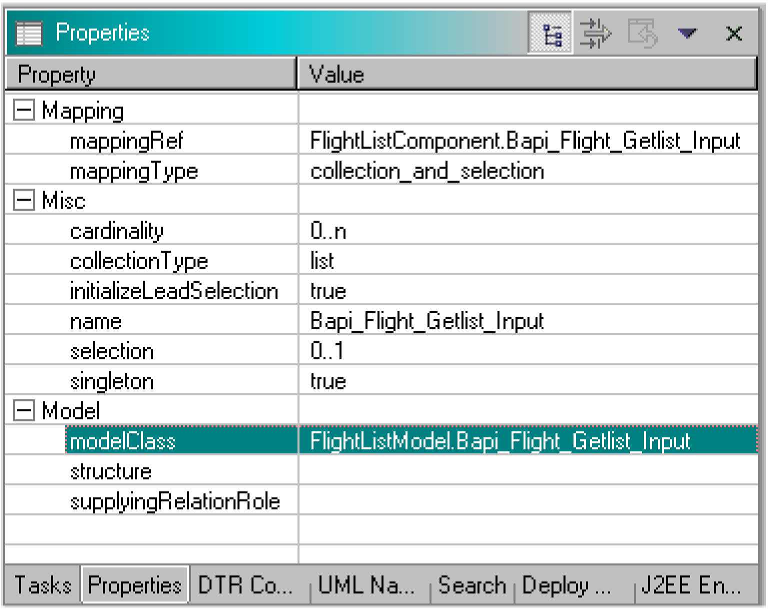


Figure 44: Model class of BAPI input context node

You do this by editing the component `FlightListView` component and selecting the *Context* tab. First, select the node `Bapi_Flight_Getlist_Input`. In the *Properties* windows in the lower right hand corner of the IDE, select the *Properties* tab. Here you will see a property called *modelClass*. In your case, this will be `FlightListModel.Bapi_Flight_Getlist_Input`: this is the object type to be instantiated.

All context nodes have a `bind()` method with three signatures that can be used in one of the following ways. The object being bound to the node collection causes either:

1. The whole node collection to be thrown away and replaced with a single element.
2. The whole node collection to be thrown away and replaced with a new element collection.
3. A new element to be appended to the end of the existing collection.

In our case, all the model nodes in our context are completely empty, and we want to create a new element that represents all the parameters to be passed to the BAPI. Therefore it is quite acceptable to use the bind() method that causes option 1 to take place. The coding extract in Textbox 8 shows how this is achieved.

```
public void wdDoInit() {
  //@@begin wdDoInit()
  bapiInput = new Bapi_Flight_Getlist_Input();

  wdContext.nodeBapi_Flight_Getlist_Input().bind(bapiInput);
```

Textbox 8: Create the BAPI input object and bind it to the context node

Now that this has been done, an actual object exists as element 0 of node collection `Bapi_Flight_Getlist_Input`. Now you have a place to store user input.

Notice that this code is within the `wdDoInit()` method. This means that it will be executed only once, when the component controller is instantiated.

Two more tasks need to be performed in this method.

Firstly, if you wish to display a message on the client, then object that performs this task is known as the Message Manager.

Important: There is only one Message Manager per component!

Since all view controllers are hierarchically subordinate to the component controller, and the component controller is responsible for constructing the window that becomes the component's interface view, there is no need for each view to have its own message manager. Therefore, to gain access to the Message Manager object, you must first access the component controller's API. This is done as follows:

```
// Get a reference to the message manager via the component
// controller's API,
msgMgr = wdThis.wdGetAPI().getMessageManager();
```

Textbox 9: Get a reference to the message manager

The next block of code creates the elements for the `Destination_To` and `Destination_From` nodes. Remember that these node correspond to `IMPORTING` parameters and have a cardinality of `0..1`.

```
// Create new elements in the Destination_From and Destination_To nodes
bapiInput.setDestination_From(new Bapisfldst());
bapiInput.setDestination_To(new Bapisfldst());
```

Textbox 10: Create model object instances for IMPORTING parameters

Since `msgMgr` object will be needed by the method that calls the BAPI (which you haven't written yet), this object is declared as an instance variable between the `//@@begin others` and `//@@end` comment markers.

```
//@@begin others
IWDMessageManager msgMgr;
Bapi_Flight_Getlist_Input bapiInput;
//@@end
```

Textbox 11: Declare a message manager object

All the code from Textbox 8 to Textbox 10 is contained within the `wdDoInit()` method of the component controller.

Resolving unknown references in the source code

Some statements in the coding may have a wavy red line placed under them by the dynamic syntax checker. This indicates that you are referencing a class that has not yet been defined. You can fix this either by manually adding the required import statement, or by using an automatic name resolution feature of the SAP NetWeaver Developer Studio.

To call up the context menu click in the coding window using the alternate mouse button and then choose *Source -> Organize Imports*.

The SAP NetWeaver Developer Studio then automatically tidies up import statements.

If there are several classes that match the one used by your program, then a selection dialog box will be displayed, and you must choose the correct class to be imported.

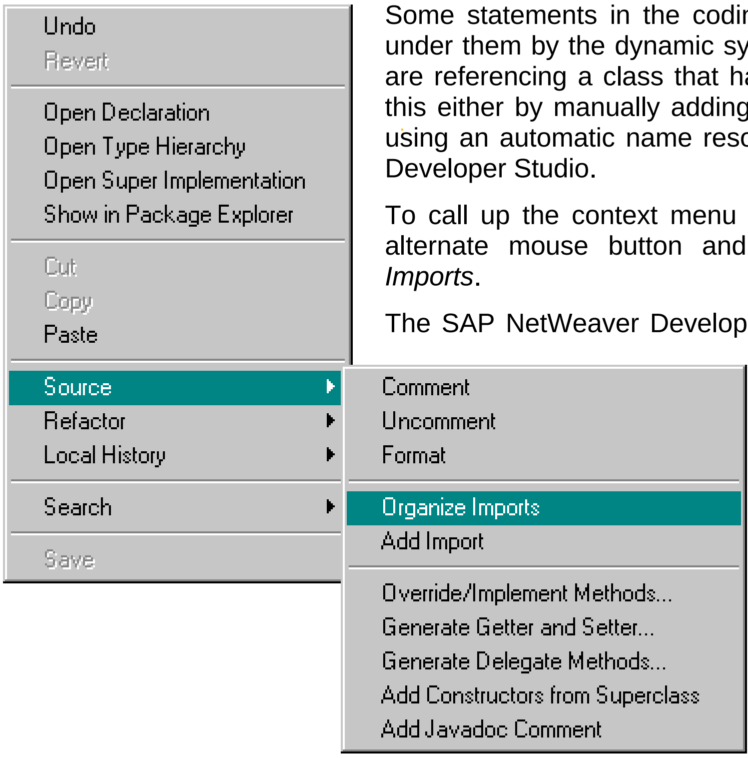


Figure 45: Organize Imports

Creating a method in the Component Controller

Now you need to create the method in the component controller that will assign the user's input values to the BAPI interface, and then call the BAPI.

Edit the component controller by double clicking on the name *Component Controller* underneath *FlightListComponent* in the Web Dynpro Explorer project tree. Select the *Methods* tab. To the right of the screen, press the *New* button.

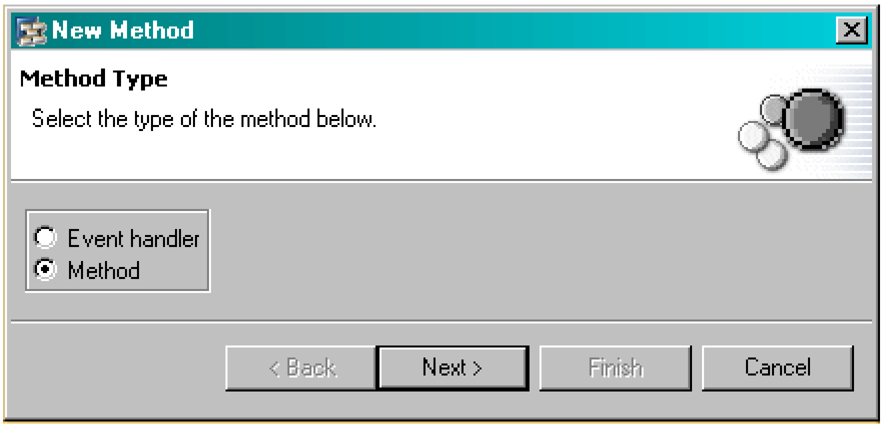


Figure 46: Create a new method. Part 1

Select the method radio button as shown in Figure 46 and press *Next*. On the next screen, you will need to give the method a name. Enter `callBapiFlightGetlist` and press *Next* again.

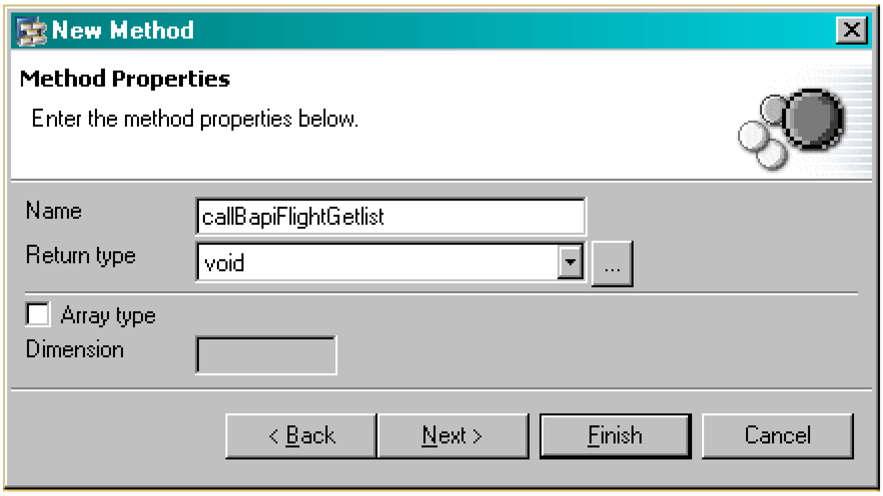


Figure 47: Create a new method. Part 2

At this point you could simply press *Finish*, but that that would leave your method without the ability to receive parameters. Therefore to pass the user's input values from the view controller to the component controller, you are going to transfer them as method parameters.

You could just as easily have done this by creating a context node in the component controller of the same structure as the *InputValues* node in the view controller context (see Figure 33), and then connecting the two by context mapping. Nonetheless, you will use parameter mapping in order to illustrate how to access the context programmatically.

You have to repeat the following procedure twice. Press the *New* button to create a new method parameter. Enter the name of the parameter, and set the type to `date`.

Parameter name	Data Type
dateFrom	date
dateTo	date

Table 2: Create method parameters.

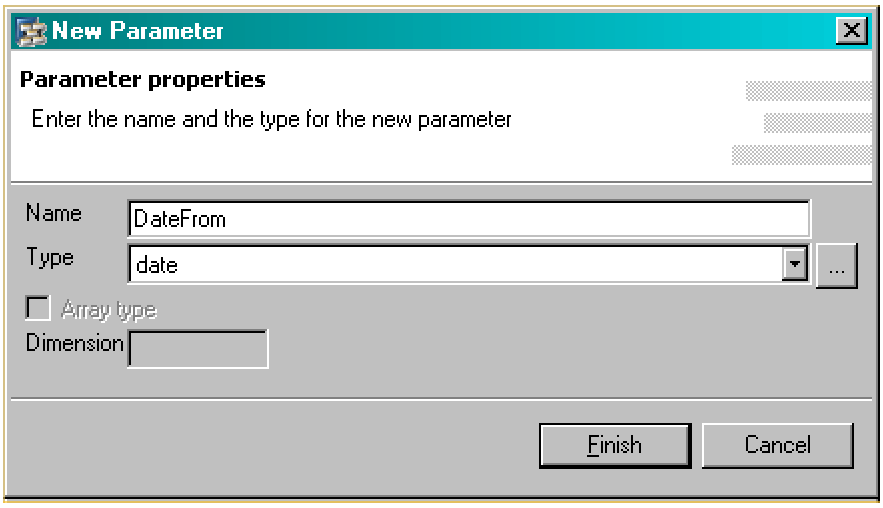


Figure 48: Create a new method parameter. Part 1

Once you have completed the creation of all both parameters, your method creation window should look like Figure 49.

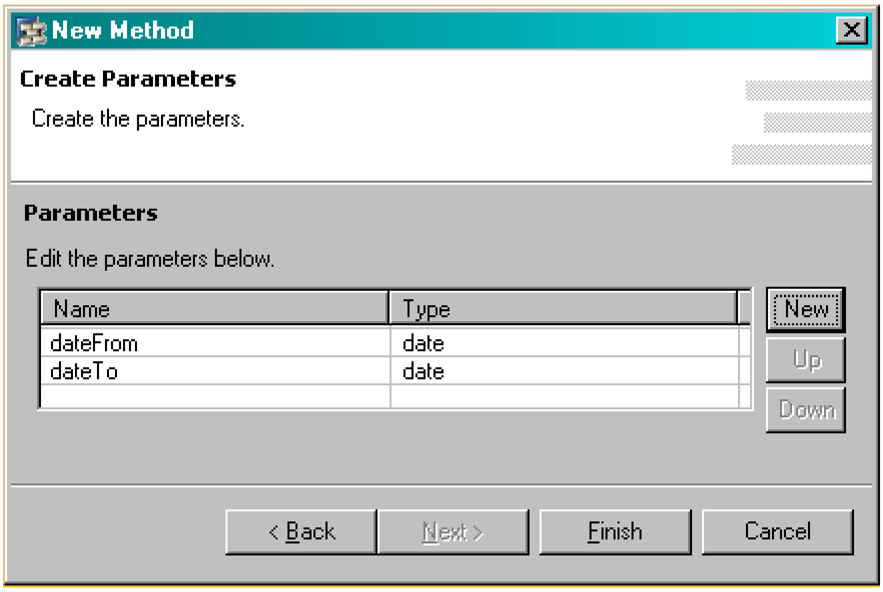


Figure 49: Create a new method parameter. Part 2

Press *Finish*.

Now that the method and its parameters have been declared, you need to implement the actual BAPI call.

Identifying Model Node Object Types

Before you can write the following code, you need to be able to identify which objects should be instantiated. You can do this by editing the `FlightListComponent` component and selecting the *Context* tab. First, select the node `Date_Range`. In the *Properties* windows in the lower right hand corner of the IDE, select the *Properties* tab. Here you will see a property called ModelClass. In your case, this will be `FlightListModel.Bapisfldra`.

The entire input side of the BAPI interface is represented by the model node called `Bapi_Flight_Getlist_Input`. Select this node in the component controller context window and make a note of the *modelClass* property.

Coding the callBAPIFlightGetList() Method of the Component Controller

The following coding shows all the coding necessary to take the two parameter values from the view controller, add them to the appropriate node element objects, and then call the BAPI. The success or failure of the BAPI call is reported back to the user through the Message Manager.

```
public void callBapiFlightGetlist(java.sql.Date dateFrom,
                                  java.sql.Date dateTo ) {
  //@@begin callBapiFlightGetlist()
  // Create new mode node objects to hold the user input
  Bapisfldra dateRangeObj = new Bapisfldra();

  // Store user input in element objects
  if (dateFrom != null) {
    dateRangeObj.setLow(dateFrom);
    dateRangeObj.setSign("I");

    if (dateTo != null) {
      dateRangeObj.setHigh(dateTo);
      dateRangeObj.setOption("BT");
    }
    else
      dateRangeObj.setOption("EQ");

    bapiInput.addDate_Range(dateRangeObj);
  }

  // Call BAPI
  try {
    wdContext.
      nodeBapi_Flight_Getlist_Input().
        currentBapi_Flight_Getlist_InputElement().
          modelObject().
            execute();
  }
  catch (Exception ex) {
    msgMgr.reportException(ex.getLocalizedMessage(), false);
  }

  // Resynchronise the data in the context with the data in the model
  wdContext.nodeOutput().invalidate();

  // Remove the input table rows so that successive user inputs
  // do not accumulate
  bapiInput.removeDate_Range(dateRangeObj);

  // Report success message through the message manager
  msgMgr.reportSuccess("Success");
  //@@end
}
```

Textbox 12: The Coding for method callBapiFlightGetlist()

The first thing to do is create an appropriate model object for the `DATE_RANGE TABLES` parameters. The class of this object can be determined from looking at the *modelClass* property when the appropriate node is selected in the context viewer.

Since the `dateTo` and `dateFrom` parameters are being placed into a standard ABAP `SELECT-OPTIONS` table, the table contents must vary depending on the user input. Therefore, it is necessary to examine whether or not the user has supplied any data for these fields.

```
// Store user input in element objects
if (dateFrom != null) {
  dateRangeObj.setLow(dateFrom);
  dateRangeObj.setSign("I");

  if (dateTo != null) {
    dateRangeObj.setHigh(dateTo);
    dateRangeObj.setOption("BT");
  }
  else
    dateRangeObj.setOption("EQ");

  bapiInput.addDate_Range(dateRangeObj);
}
```

Textbox 13: Examining user input before adding it to the SELECT-OPTIONS table

If the user has entered a value in the `dateFrom` field, then we know that at least the `DATE_RANGE-LOW` field will need to be populated. The `dateFrom` value is assigned to the `SELECT-OPTION LOW` field by calling the standard mutator method `setLow()`. The `SELECT-OPTION SIGN` field will always be set to "`I`" for include.

If the user has entered a value in the `dateTo` field, then an inclusive range is being specified.[7] Therefore, you must not only assign the `dateTo` value to the SELECTION-OPTION HIGH field, but you must also change the `SELECTION-OPTION OPTION` field to "BT" for between.

If the user leaves the `dateTo` field empty, then the `SELECTION-OPTION OPTION` field should be set to "`EQ`" for equals.

Having created the `DATE_RANGE TABLES` object, and assigned the correct values to it, the entire object must be attached to the object that represents the input side of the BAPI. Failure to do this step will cause the BAPI to execute with no `DATE_RANGE` values. In other words, it will completely ignore any values typed into the date fields!

This is why it is necessary to call `bapiInput.addDate_Range(dateRangeObj)`.

Now you are actually ready to call the BAPI. This must be done inside a standard try/catch construct because the BAPI execution could raise an exception if a network error occurs.

```
// Call BAPI
try {
  wdContext.
    nodeBapi_Flight_Getlist_Input().
      currentBapi_Flight_Getlist_InputElement().
        modelObject().
          execute();
}
catch (Exception ex) {
  msgMgr.reportException(ex.getLocalizedMessage(), false);
}
```

Textbox 14: Calling the BAPI

The statement to call the BAPI is a little lengthy, and requires some explanation!

[7] It would be perfectly possible to extend the functionality here to allow the full use of ABAP's `SELECT-OPTION` capability. But that will be left for you to do on your own!

When the wdDoInit() method created the bapiInput object, it was creating an instance of an object whose metadata is derived from an Adaptive RFC model. This metadata also includes all the information necessary to execute this model's functionality. However, since the Bapi_Flight_Getlist_Input context node has a cardinality of 0..n, it is perfectly possible to have multiple BAPI input objects stored as elements within this node collection. Each BAPI input object could point to a different SAP system (one of the big advantages of the Adaptive RFC Layer), so in order to execute the BAPI, you must reference the exact element of the context node that holds the BAPI input information.

Therefore, you have to chain through the context (wdContext) to the BAPI input node (node-Bapi_Flight_Getlist_Input), and then to the currently selected element of that node (cur-rentBapi_Flight_Getlist_InputElement). Now you have a reference to exactly the right BAPI input object (in your case, there is only one element in this collection!) and from here you can access the model functionality itself via modelObject(). After this, it is simply a matter of calling the execute() method to fire the BAPI call.

If the BAPI call experiences a network problem, then an exception will be raised and the catch clause will invoke the message manager to notify the user of the problem.

```
  // Resynchronise the data in the context with the data in the model
  wdContext.nodeOutput().invalidate();

  // Remove the input table rows so that successive user inputs do not ac-
cumulate
  bapiInput.removeDate_Range(dateRangeObj);

  // Report success message through the message manager
  msgMgr.reportSuccess("Success");
```

Textbox 15: Final processing steps after BAPI call

Assuming that the BAPI is successful, the Output node of the context must now be invalidated. The invalidate method of any node resets that node's collection to the minimum number of elements specified by the cardinality – that is, zero or one. Then it examines the node's metadata to see how the node should be populated. In the case of a model node, the data for the context node is retrieved from the model object. This process guarantees that the data held in the context is always synchro-nized with the model.

Since you are allowing the user to call the BAPI multiple times, you must ensure the old output stored in the context is destroyed before attempting another BAPI call. In your case, you only have one con-text node that is affected in this way; therefore, you can arbitrarily delete the only element in the Date_Range context node.

To clean up the import statements as before call up the context menu by clicking in the coding win-dow using the alternate mouse button and then choose *Source -> Organize Imports*.

Lastly, report a success message through the message manager.

Coding Added to the View Controller

The following coding applies to the view `FlightListView`. Double-click on the view name in the project hierarchy, either under the `Views` nodes or under the `Window` node.

Once the view has been opened for editing, select the tab *Implementation*.

Implementing the `onActionHandleButtonPush()` Method

The only coding that needs to be added to the view controller is the implementation of the action handler for the `HandleButtonPush` UI action. In general, for every action `<act>` that you define in a view controller, a method called `onAction<act>()` will be created.

You will need to reorganize the imports for the `Date` object. You will be asked to select from `java.util.Date` and `java.sql.Date`. You must select `java.sql.Date`.

```
public void onActionHandleButtonPush(
             com.sap.tc.webdynpro.progmodel.api.IWDCustomEvent wdEvent) {
  //@@begin onActionHandleButtonPush(ServerEvent)
  Date dateFrom = wdContext.currentInputValuesElement().getDateFrom();
  Date dateTo   = wdContext.currentInputValuesElement().getDateTo();

  wdThis.
    wdGetFlightListComponentController().
      callBapiFlightGetlist(dateFrom, dateTo);
  //@@end
}
```

Textbox 16: Implementing the onAction event handler

The first thing to do is retrieve the two date values from the context.

Important: One of the things about Web Dynpro programming that is very different from other types of web interface is that when retrieving values entered by the user, you ***do not*** refer to the UI element objects seen on the screen. Since all UI elements are bound to nodes or attributes in the view controller's context, you can always retrieve data directly from the context.

If you refer back to Figure 41, you will see that the two UI elements `DateFromInput` and `DateToInput` were bound to context attributes `InputValues.DateFrom` and `InputValues.DateTo` respectively.

The first two statements in `onActionHandleButtonPush()` create some member variables that obtain their values directly from the context. Since the `InputValues` context node was created with a cardinality of `1..1`, you know for certain that it will contain exactly one element. Therefore, the generated method `wdContext.currentInputValuesElement()` will refer to precisely one node element. The attributes `DateFrom` and `DateTo` are then accessed via the standard accessor methods.

The last thing to do is call the method `callBapiFlightGetlist()`. However, this method belongs to the component controller, and the `onActionHandleButtonPush()` method you are currently coding is part of the view controller. These two controllers are separate Java programs, but have been connected together by the *Required Controllers* declaration made earlier. Refer back to Figure 26 if you've forgotten this important step!

Save your work using the icon on the toolbar.

You have now completed all the manual coding.

Creating an Application

Now that you have completed the declarations and coding required for this Web Dynpro component, the last step is to create a Web Dynpro application. As stated earlier the only possible entry point into a component is through an application.

Figure 50 Application Menu

You now have a fully functional Web Dynpro component, but you need to define a means of invoking it.

In the Web Dynpro Explorer window in the top left of the SAP NetWeaver Developer Studio right click on the `Applications` node under the `Web Dynpro` node of your project. From the menu shown above, select *Create Application*.

You will now be asked for the name of your application, and the Java package to which you want it assigned. Figure 51 below shows the required values for your flight list example.

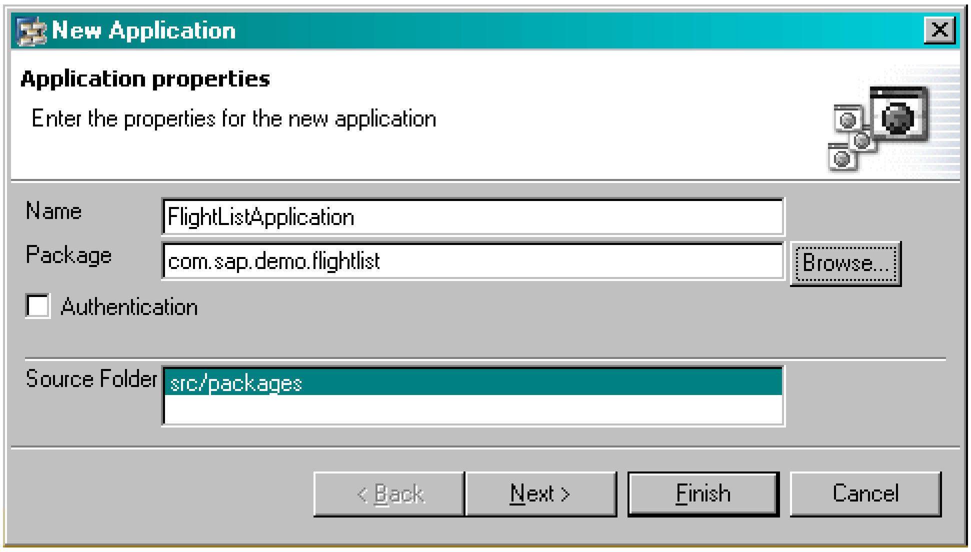


Figure 51: Create an Application

Choose *Next*. Since you have already created the component for which this application will act as an entry point, you can leave the radio button selection shown in Figure 52 below in its default position, and press *Next* again.

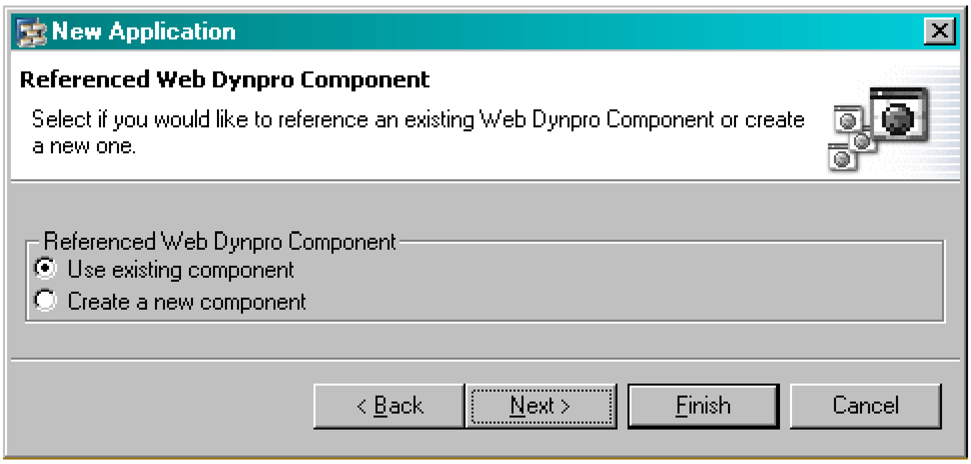


Figure 52: Referenced Component

On the next screen specify what component you would like this application attached to. Since your project only has a single component, all dropdown list entries are set to their correct values.

This is shown in Figure 53 below.

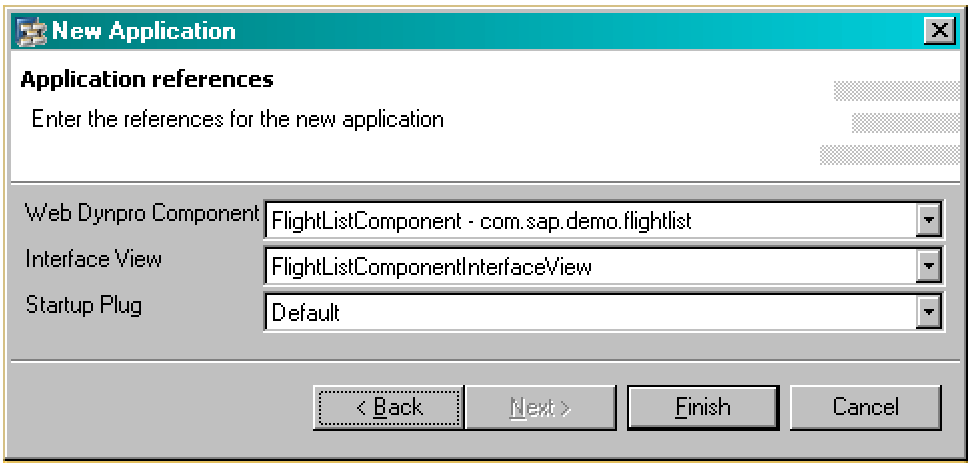


Figure 53: Application entry point

After you press *Finish*, the project tree will look like Figure 54 below. Save your work using the  icon on the toolbar.

Figure 54: The Defined Application

Deployment of Application

To run your Web Dynpro application you must deploy it to the J2EE Engine. Applications can be deployed in one of two ways. Either:

- Click on the application *FlightListApplication* with the alternate mouse button and select *Deploy new archive and run* as shown in Figure 55 below. With a single click, you will be able to both deploy the application and run it. This is the most useful option when testing during development.

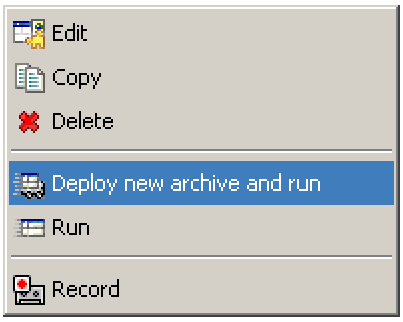


Figure 55: Application menu

Or

- Click with the alternate mouse button on the project node itself in the Web Dynpro Explorer window, and then select *Deploy* from the menu shown in Figure 56 below. This will create a new archive and deploy it to the J2EE Engine, but it will not run it. To run the application, you must now click on the application node with the alternate mouse button in the project tree, and then select *Run* from the menu.

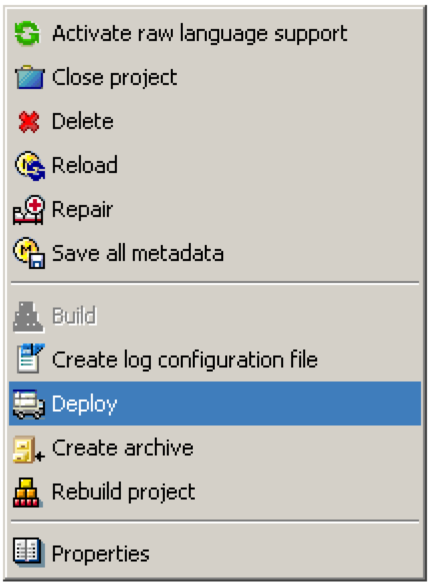


Figure 56: Application menu

Results

After you have deployed and run your application, you should see the screen shown in Figure 57 below in your browser.

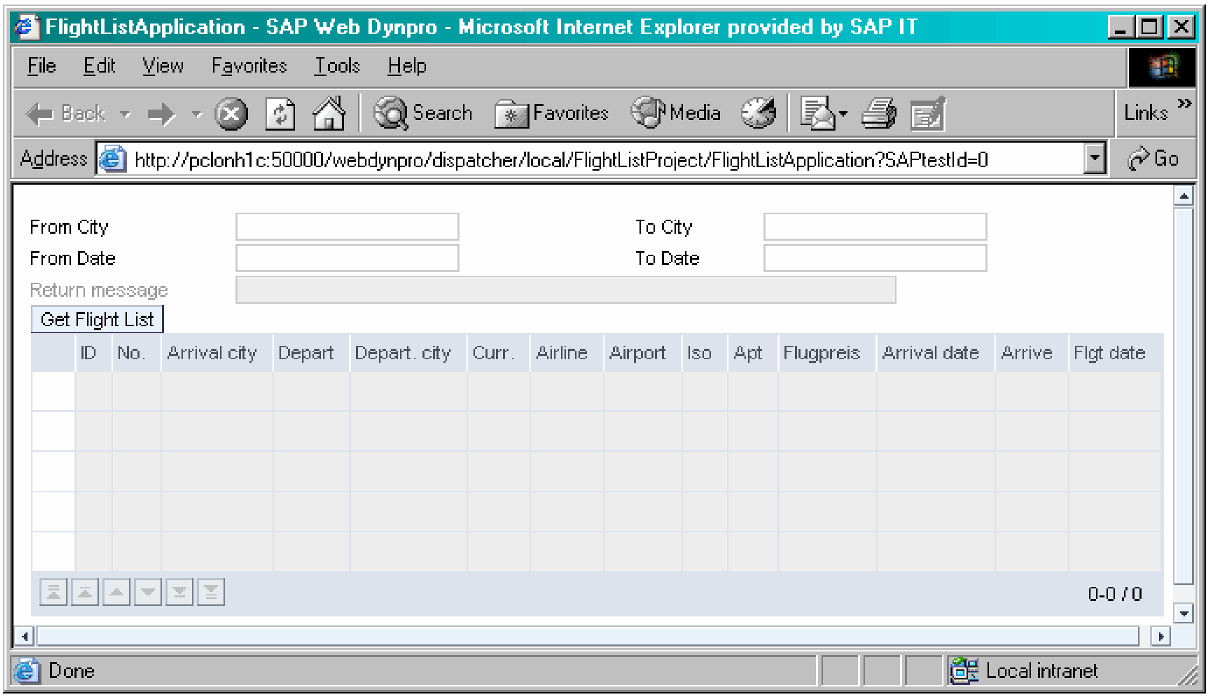


Figure 57: Initial Screen

Here, you can enter any valid search string you like. For instance, if you enter New York in the *From City* field you may see something like Figure 58 below.

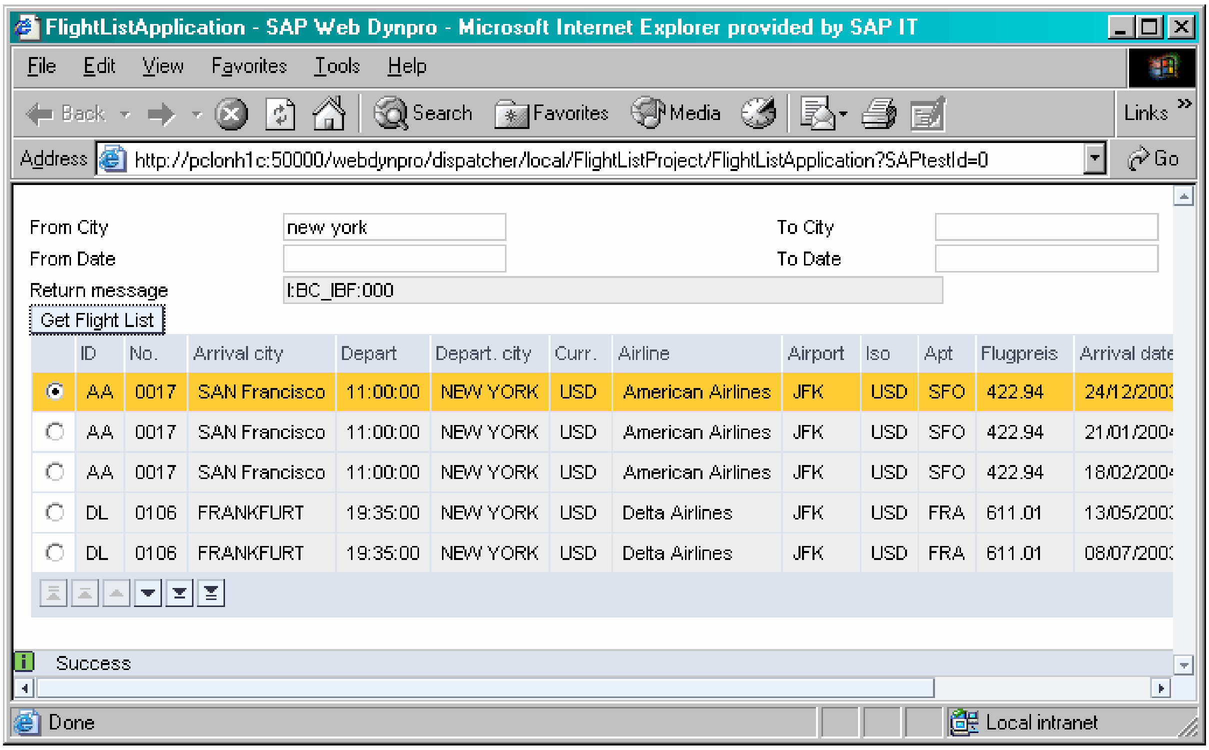


Figure 58: Search results for flights leaving from New York

Using the date picker

Referring back to Figure 41 you will remember that the `DateFromInput` and `DateToInput` UI elements in your view were bound to context attributes of type `date`. As a result of this binding, the UI element will automatically have a date picker associated with it. You do not need to do anything here other than associate the input field with a context attribute of type `date`.

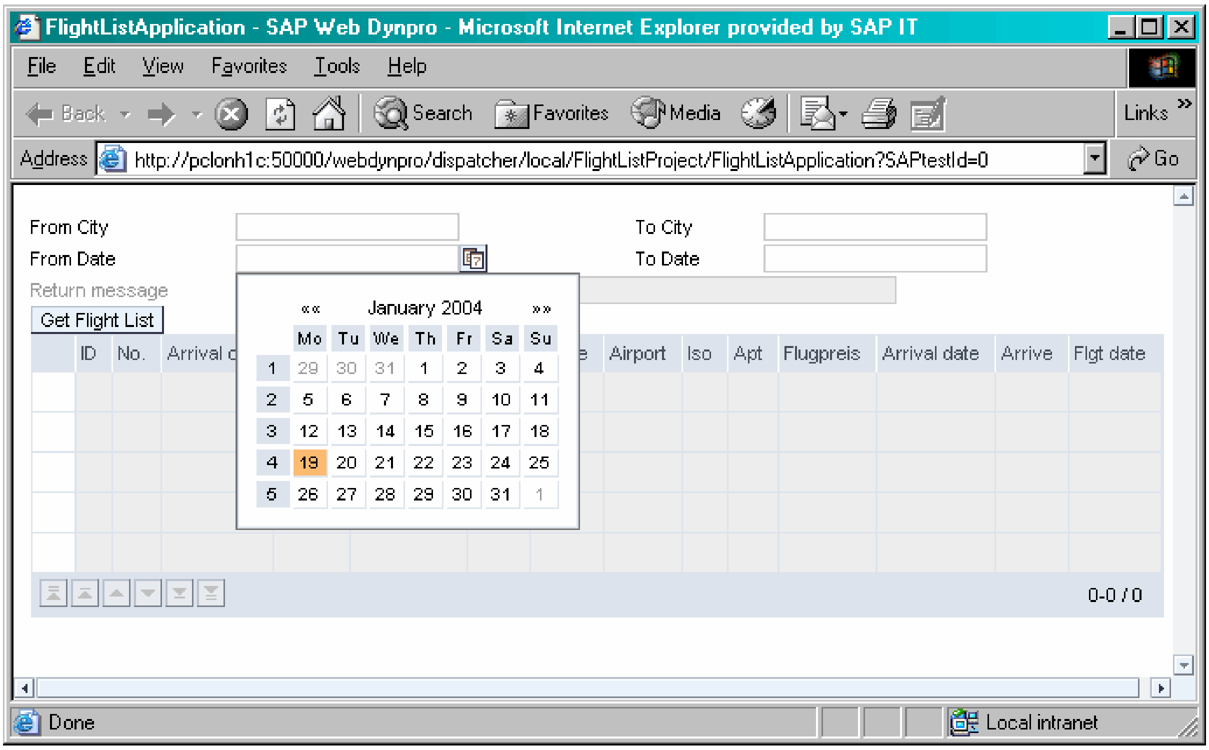


Figure 59: Date picker automatically associated with input fields of type date

By clicking on the little calendar icon to the right of the input field, the date picker window shown in Figure 59 above will be displayed.

You can enter any valid date into the `From Date` or `To Date` fields. The date format assigned to this input field is specified by the locale information of the Adaptive RFC user. In this case, the user ID specified in the JCO Connections of the J2EE Engine for the logical destination `FLIGHTLIST_MODELDATA_DEST`, has the date format DD/MM/YYYY. Therefore, when the selected date has been copied from the date picker window to the input field, it appears as 19/1/2004.

Should you choose some selection criteria that do not match any flights, then you will see the following type of output.

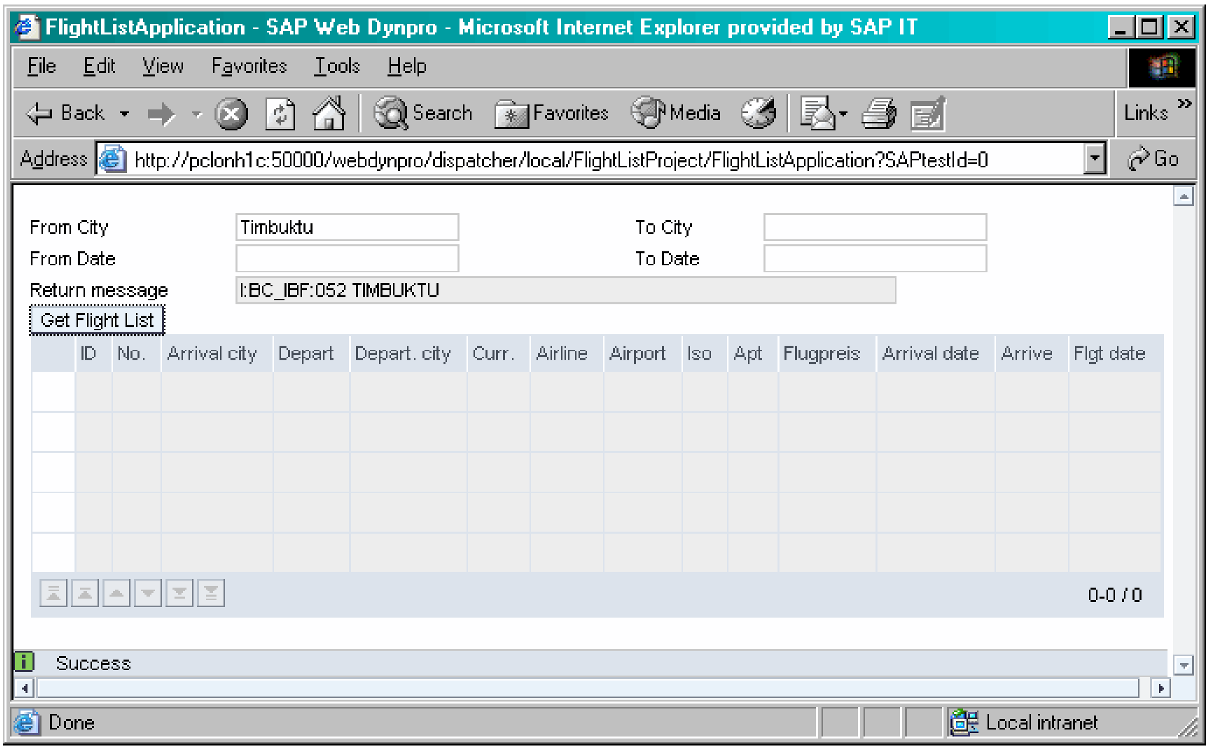


Figure 60: No flight information found

Summary

The following list summarizes the main steps for creating this application. These steps assume that you have already added your SAP system to the System Landscape Directory (SLD).

1. Create a project and give it a name.
2. Within the project, create a component by specifying a name and a Java package name.
3. Create a model based on the function module `BAPI_FLIGHT_GETLIST`.
   - Select the correct logical system names for the Adaptive RFC layer
     - Make sure these logical system names are correctly configured in the JCO Connections section of the Content Administrator in the J2EE Engine.
   - Select which SAP system to connect to and then log on
   - Select the required BAPI (or BAPIs)
4. Declare the use of the model you have just created in your component under *Used Models*.
5. Create a model node in the component controller's context. For simplicity, give the model node the same name as the input side of the BAPI interface.
6. Bind the context model node to the model object.
   - Resolve any name conflicts
7. Declare instance variables between the `//@@begin others` and `//@@begin end` markers.
8. Edit the `wdDoInit()` method of the component controller
   - Create an instance of the model object and bind it to the context model node.
   - Create a message manager object
   - Create model objects for context model nodes of cardinality `0..1`.
9. Edit the `callBapiFlightGetlist()` of the component controller adding the code to:
   - Check whether the user has entered any date values, and build the correct input parameters for the `SELECT-OPTIONS` table `DATE_RANGE`.
   - Within a try/catch construct, call the BAPI.
   - If the BAPI call fails, use a message manager to display the error message.
   - Invalidate the `Output` node of the component controller's context.
   - Arbitrarily remove the element from the `Date_Range` context node.
   - Report a BAPI success message using the message manager.
10. In the properties tab of the view controller, declare the use of the component controller under required controllers.
11. Create a model node in the view controller's context
    - For simplicity, give this model node the same name as the model node in component controller's context.
    - Use context mapping to connect the model node in the view controller's context to the model node in the component controller's context.
12. In the view controller's context, create a value node called `InputValues` of cardinality `1..1` under which are two attributes of type `date`. These attributes will act as temporary input fields for value passed to the BAPI's `TABLES` parameters.
13. Add a pair of label/input UI elements to the view layout for each of the input fields.

- Give these fields meaningful names using the recommended naming convention.
- Set the `labelFor` property of the label fields.
- If any fields are to be output only, then set the `readOnly` property to true.
- Bind the `value` property of the input fields to the appropriate attributes in the context.
- Set the `layout` property of the `RootUIElementContainer` to `GridLayout`, and then set the `colCount` property to 4.
- Adjust the `colSpan` property of any required fields.

14. Add a pushbutton to the view layout.
15. Create an action in the view to be triggered when the user pushes the button.
    - Associate the action with the button UI element.
16. Add an invisible UI element to fill the empty cell to the right of the pushbutton.
17. Add a table UI element.
    - Set the `colSpan` property to 4.
    - Set the `dataSource` property to point to the `Flight_List` node on the `Output` side of the BAPI interface in the context.
    - Define the binding properties for each of the table columns.
18. Edit the view controller's `onActionHandleButtonPush()` method.
    - Get the two date fields from the view controller's context.
    - Call the method `callBapiFlightGetlist()` in the component controller and pass the two date fields as parameters.
19. Create an application
20. If this is the first time you have created an application using the logical destinations specified in step 3, then you should only *Deploy* the application – not *Deploy and run*.
    - Configure your application's logical destination (JCO Connections) in the Content Administrator of the J2EE Engine. This is a one off task.
21. If you have already configured the logical destinations of your application, then you can *Deploy and Run* your application.