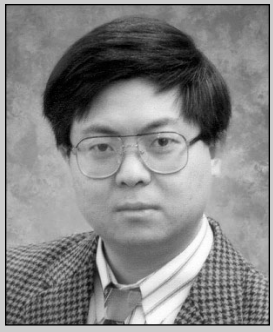
Develop More Extensible and Maintainable Web Applications with the Model-View-Controller (MVC) Design Pattern

Ken Huang and Markus Wieser



Ken Huang, Senior Programmer/Analyst, Varian Medical Systems

Markus Wieser, Senior Developer, SAP Labs

(complete bios appear on page 84)

With Release 6.10 of the SAP Web Application Server (Web AS), SAP introduced the Business Server Pages (BSP) technology, a significant advancement in SAP web application development. Developers could quickly and easily produce dynamic and content-rich web applications by combining markup languages such as HTML, XHTML, and XML with ABAP and JavaScript code. While the basic BSP approach works well for smaller-sized applications, without a defined design pattern and a clear separation of layout and business logic, it can quickly become cumbersome as applications inevitably increase in size and complexity.

The Model-View-Controller (MVC) design pattern is ideal for handling large, complicated applications. It is an architecture that compartmentalizes the different types of logic that structure an application, making it more extensible and maintainable. The MVC architecture has long been a software development standard and is now gaining popularity in the web application development community — it has become the recommended model for Sun's Java 2, Enterprise Edition (J2EE) platform, for instance. With Release 6.20 of the Web AS, SAP added support for the MVC design pattern to address the challenges of handling large-scale and complex BSP applications.

Unfortunately, there isn't a lot of detailed information on building BSP applications with the MVC design pattern. MVC is mentioned only briefly in SAP's training class for BSP programming, and when we ourselves started building an MVC-based BSP application, we found that the SAP documentation didn't cover all the details needed to guide an ABAP developer through a real-world implementation. As SAP employees, we had the luxury of consulting our colleagues, internal newsgroup discussions, and the BSP framework team. For an ABAP

developer outside of SAP, however, the available resources are limited — only a small number of published books and articles are dedicated to the concept, and most of the information currently available is not presented in a tutorial way.

This article aims to close this information gap. It examines the MVC design pattern in detail and shows ABAP developers and web application architects how to apply the pattern to BSP application development.[1] After reading this article and following the step-by-step example provided, you should be able to quickly start building your own MVC-based web applications and have a good foundation for exploring more advanced uses of the pattern.

> ✓ ***Prerequisites***
>
> *To implement the example detailed in this article, you need SAP Web AS 6.20. This article also assumes that you are familiar with the basics of ABAP Objects and BSP programming.*

Introducing the MVC Design Pattern

With the basic BSP development approach, an application's layout and business processing logic are mingled together in the BSP page, leaving the structuring of the application to the experience and style of the code developer. For instance, a BSP application might require the ability to search for business partners in different parts (i.e., pages) of the application. A developer who lacks knowledge of, or experience with, modern architecture design might implement the search directly in a BSP page's event handler, using attributes that are specific to that particular page. Let's say this developer then needed to add the search function to another page with slightly different attribute definitions. Since the search is implemented directly in the event handler of the first page, it is nearly impossible to reuse the page's business logic — the second page cannot access the implementation, and even if the logic is moved to a dedicated class, for example, you would still need to deal with the differing attribute definitions. So the same logic must be implemented again and again, requiring repeated tests, bug fixes, and changes in several (sometimes forgotten!) places. Clearly, the lack of a defined structure requires significant specification and design effort to keep an application's architecture consistent.

If the MVC architecture had been used in the previous example, the developer could have implemented the search function in a model, rather than directly in the BSP page, which could also then be used to build view layouts for the required searches. With this architecture, not only is the business logic reused, the same search function (with the same labels, field length specifications, etc.) is available to all views. The model can even be reused in other applications.

So how does the MVC design pattern make this possible? It separates an application's presentation logic from its business logic and defines the responsibilities of the individual components, enabling you to:

- **Modify the application's user interface logic without affecting the business logic, and vice versa.** The separation of these two elements makes it easier to incorporate changing requirements, like presenting different kinds of details to users, reading data from different database tables, or enhancing business rules. For example, without ever touching the user interface, you can change the business rules of a sales application so that some services or discounts apply only to customers with a certain amount of purchases. With the standard BSP architecture, you would first have to search through the embedded business logic of all the BSP pages in the application, and then modify the code page by page — which is

[1] Two previously published *SAP Professional Journal* articles — "A Developer's Guide to Creating Powerful and Flexible Web Applications with the New Web Application Builder" (January/February 2002) and "Build More Powerful Web Applications in Less Time with BSP Extensions and the MVC Model" (March/April 2003) — provide a good introduction to BSP, MVC, how to apply MVC design to BSP development, and how to use the Web Application Builder tool.

not only time-consuming, but also increases the risk of altering the layout code.

- **Modularize and reuse business logic components.** With the MVC architecture, the business logic is its own component, separate from the user interface, making it very easy to reuse in other BSP pages. With the standard BSP architecture, there is no defined component that provides a BSP page's business logic. The business logic could be implemented in any of several different ways (depending on the page's design and the developer's experience, for example): the business logic could be in a separate class, which is fairly easy to reuse, but it could just as easily be implemented directly in a BSP page's event handler and be tightly linked to the page attributes. Even worse, it could be embedded in the BSP page's layout. You can imagine how difficult it is to reuse such logic, and how much effort it requires to change even simple things like adding a search function or changing field labels.
- **Minimize the maintenance of increasingly complex applications.** With MVC's compartmentalized structure, corrections or enhancements can be applied centrally, so you don't have to search through different BSP page layouts and event handlers to make the required changes, as is the case with the standard BSP architecture.
- **Leverage the specialized skill sets of application developers.** Because the user interface and the business logic are separate, it is much easier for developers with different specializations to work on the same application. For example, a web designer can concentrate on the page layout while an ABAP developer can focus solely on the business logic. The traditional BSP architecture requires BSP developers to be well versed in both web design and ABAP programming.

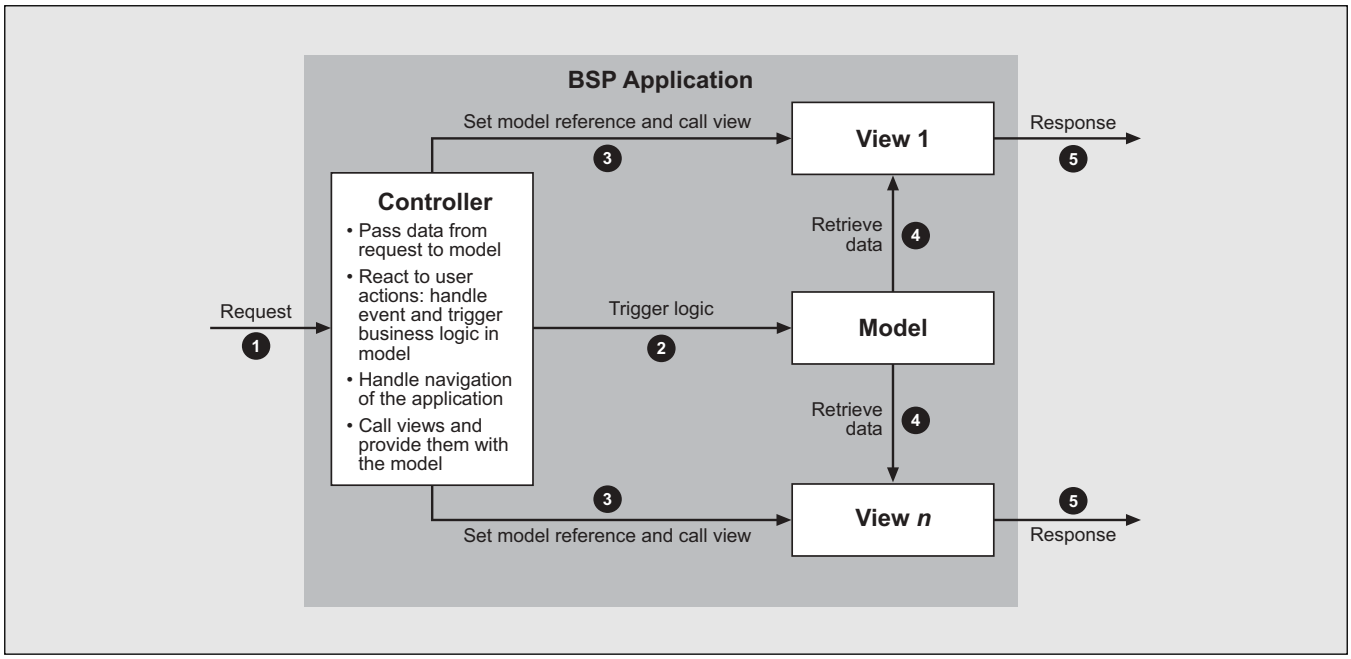
You might be thinking, "This sounds great! Why didn't someone think of it before?" Believe it or not, the MVC design pattern is not a new idea. It was originally invented in the 1970s by PARC (Xerox's Palo Alto Research Center) to decouple the GUIs of Smalltalk applications from the code that actually executes the application's business logic. Since then, the pattern has become a key element of object-oriented design that is used in a variety of languages for GUI and web application development.

The MVC design pattern provides three logical components for the design of an application or a subsection of a complex application (a search function that is reused in other parts of the application, for example):

- The **model** contains the logical structure of the application's business data. It is implemented as an ABAP Objects class and is built on business rules without regard for how data is presented to the user. Since a model can serve multiple views, it is possible to display the same data in different formats, such as HTML, XML, and WML. A view gets the model reference from the controller, but how the data is displayed is up to the view.
- The **view** is responsible for the visual presentation of the data in the user interface. The design and implementation of the view is independent from how the business data is retrieved, processed, and stored — this is determined by the controller (see the next bullet item). A view can use multiple models, and the same model can be used by multiple views. Let's say we have a model that provides airline flight schedules. The controller decides which view should render the flight schedule based on user requests — if the flight schedule is queried from a web browser, the controller will call an HTML view; if the query is from a cell phone, it calls a WML view.
- The **controller** facilitates the communication between the model and the view. It translates user requests from the view into actions that the model will perform. When the user clicks on a button in the view,[2] the controller's event handler processes the event and triggers the corresponding model logic, or simply navigates to a different view. Based on the user request and the outcome of the model's actions, the controller selects an

[2] The execution of a BSP application always starts with a controller, which is accessed either directly or via a BSP page that calls it. A BSP application without a controller or a BSP page cannot be displayed.

Figure 1 Architecture of a Simple MVC-Based BSP Application



appropriate view. In other words, a controller is responsible for how the application should respond to the request.

An MVC-based BSP application is constructed in the Web Application Builder in transaction *SE80*,[3] just like a regular BSP application (except for models, which are implemented with the Class Builder via transaction *SE24*). You simply create the application framework, its models, its views, and its controllers. You don't have to implement or specify anything else — there are no extra settings or options to select. The controller takes care of the process flow, the model handles the business logic, and the views provide the presentation (see **Figure 1**).

As you can see in the diagram, the MVC design pattern clearly separates control of the process (❶, ❷, and ❸), the data model of the application (❹), and the presentation of the application data (❺).

[3] In Web AS 6.20, the Object Navigator (transaction *SE80*) serves as the starting point for ABAP Workbench tools such as the ABAP Editor, Screen Painter, and Web Application Builder. When you double-click on a component of a BSP application in the Object Navigator, for example, the environment becomes the Web Application Builder.

Building a BSP Application Using the MVC Architecture

To demonstrate how to build a BSP application using the MVC design pattern, we will walk through the steps required to put together a very simple BSP application that searches a database table for airline flights and presents the search results. It contains two views: a search screen and a result screen. As we proceed, we will examine in detail how the components of the MVC architecture are being used to structure the application.

For the implementation, we will use ABAP Objects,[4] BSP pages, and *htmlb* tags, which are one of the BSP extensions provided with Web AS 6.20 (see the sidebar on the next page for more on extensions).

[4] ABAP Objects is an extension of the ABAP programming language that enables programmers to mix regular ABAP code with object-oriented commands. Modeled on Java and C++, ABAP Objects includes functionalities such as single inheritance (the ability to derive a class from an existing one), encapsulation of data and logic, and polymorphism, to name a few. For more information, navigate to *SAP Library → mySAP Technology Components → SAP Web Application Server → ABAP Programming and Runtime Environment → ABAP Programming → ABAP Objects*.

Introducing BSP Extensions

The Business Server Pages (BSP) technology, introduced with Release 6.10 of the SAP Web Application Server (Web AS), brought relief to web developers everywhere by enabling them to combine HTML and ABAP in BSP pages to create SAP web applications. However, as a web application's design inevitably becomes more complicated — by adding intricate elements like tables or increasing the number of input fields and buttons, for example — you are bound to end up with complex and repetitive HTML coding.

In Web AS 6.20, SAP introduced an enhancement to BSP programming called "BSP extensions." A BSP extension is a collection of BSP elements, which can be thought of as custom tags that encapsulate commonly used HTML code. With this encapsulation, user interface elements like table views can be added to a BSP page with only a few lines of code, making BSP application code cleaner, more compact, and easier to write and maintain.

Web AS 6.20 includes a set of predefined extensions, such as HTML Business for BSP (*htmlb*), and you can also easily define your own custom extensions in the Object Navigator (transaction *SE80*). As you will see in the example detailed in the article, to create the layout of the BSP application's views, we use the *htmlb* tags, which encapsulate common user interface elements like input fields and pushbuttons.

For an overview of all available BSP extensions and their elements, simply click on the *Tag Browser* button in the left pane of the Object Navigator. You will find SAP-delivered extensions, along with any transportable custom extensions, under *BSP Extensions* → *Transportable*. You can view locally created custom extensions that belong to a non-transportable "package" (formerly known as a "development class") by selecting *BSP Extensions* → *Local*.

For a detailed explanation of BSP extensions and their use, see Karl Kessler's article, "Build More Powerful Web Applications in Less Time with BSP Extensions and the MVC Model," in the March/April 2003 issue of this publication. You can also find more information under *SAP Library* → *mySAP Technology Components* → *SAP Web Application Server* → *Business Server Pages* → *Web Application Server* → *Web Applications and Business Server Pages* → *Programming Model* → *BSP Extensions*.

Over the remainder of this article, we will take a detailed look at the following tasks involved in building a BSP application using the MVC architecture:

- Creating the framework
- Creating a model
- Creating views
- Creating a controller
- Testing the application
- Enhancing the application
- Varying the MVC design pattern

Let's start by building the framework.[5]

[5] The example detailed in this article was created using a CRM 4.0 system running on Web AS 6.20. An R/3 Enterprise system running on Web AS 6.20 will work as well.

✓ *Following the Example*

Other than Web AS 6.20 and familiarity with ABAP Objects and BSP programming, there are no additional prerequisites to following the discussions in this article — we either build components from scratch or use standard SAP components.

✓ *Naming Conventions*

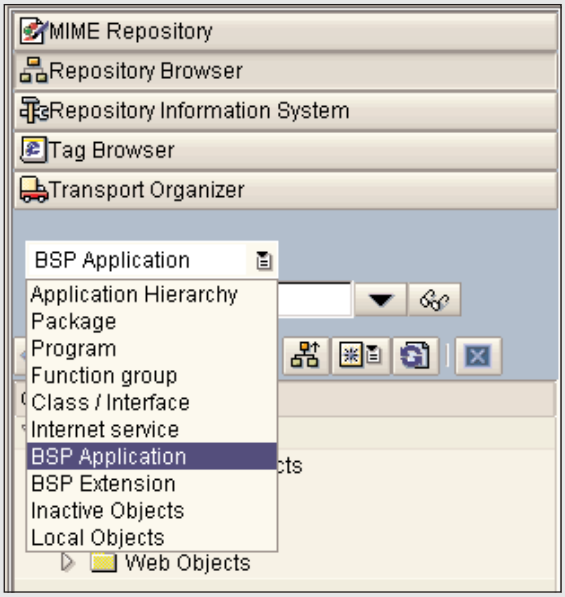
For the example, we followed SAP's standard naming conventions for customer development (see SAP Note 16466 "Customer name range for SAP objects"), so the example objects start with Z, and classes, which normally start with CL_, start with ZCL_. Where possible, we have also included in the names the theme of the example ("MVC" and "FLIGHTS") and grouping information ("CTRL" for "controller").

Creating the Framework

To create the framework for a simple BSP application, follow these steps:

1. Log on to your SAP system and start the Object Navigator via transaction *SE80*.
2. Select *BSP Application* from the dropdown menu, as shown in **Figure 2**.
3. Specify an application name (*z_mvc_flights* in the example) with a maximum length of 15 characters (see the note "Activating a Service for the Application" below), and click on the display button ( ).
4. Since the application doesn't yet exist, the system will ask you if you would like to create it. Choose *Yes*.
5. In the following dialog box, enter a short description for the application — e.g., *Sample application displaying flight information* (see **Figure 3**) — and click on the create button ( ).

Figure 2 Create a BSP Application



✓ *Activating a Service for the Application*

As of Web AS 6.20, each BSP application needs an active node (referred to as a "service") in the service tree of the Internet Communication Framework (ICF) in order to work. You can manually activate or deactivate a service — thus determining whether or not a BSP application is available — via transaction SICF. When a 6.20 BSP application is created, a service entry is automatically created and activated for that application in the ICF. The same is true when a 6.10 BSP application is opened and modified in 6.20, as long as the name of the BSP application does not exceed 15 characters (see step 3 above). For more information on the ICF and the BSP application architecture, navigate to SAP Library → mySAP Technology Components → SAP Web Application Server → Business Server Pages → Web Application Server.

Figure 3 Enter a Short Description of the Application

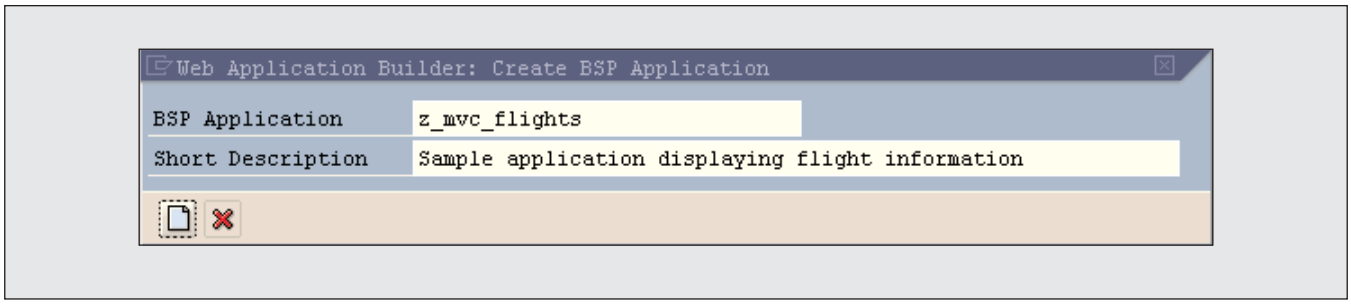
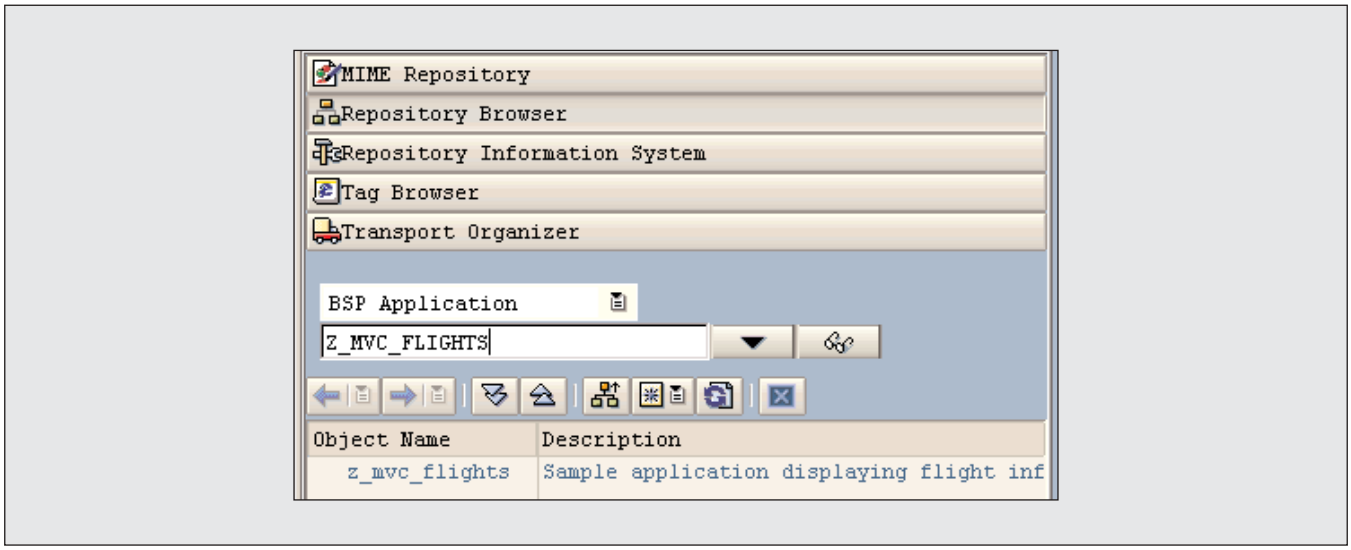


Figure 4 A BSP Application Framework in the Object Navigator

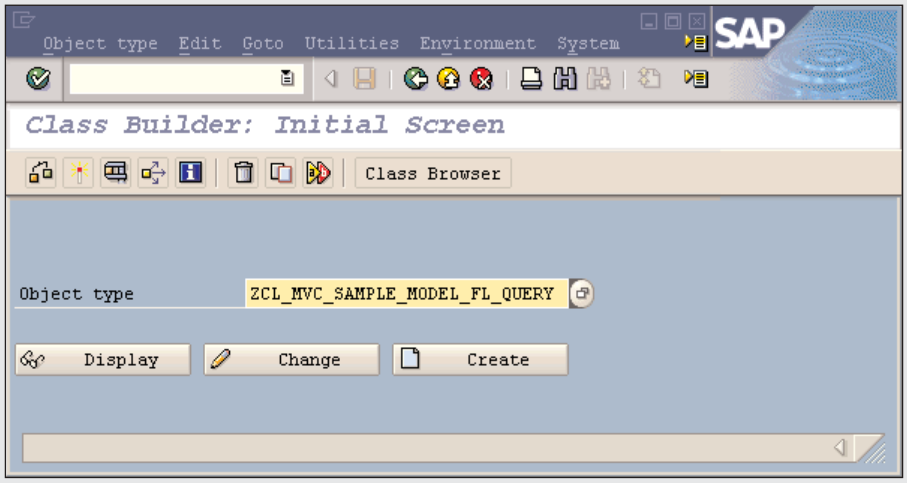


That's it — it's that simple. As you can see in **Figure 4**, you now have the framework of a simple BSP application. The next component we need to create is the model that will contain the application logic for performing the search and storing the search result. Once the model is created, we will use it to create the views and the controller for our application.

Creating a Model

The model provides access to the application's business data, which in our case is a database table containing flight information. The model's implementation is based on CL_BSP_MODEL — the base class for models. When our completed example application is launched, an instance of the controller will be created, which in turn will create an instance of the model (see the upcoming section "Creating a Controller" on page 73). The controller passes data like search criteria and search results between the views and the model. Since we are going to use data binding (see the sidebar on pages 70-71 for more on this), this data exchange will be fully automatic. Without data binding, you would have to use custom coding to transfer data between the user interface and the application's business logic.

Figure 5 Class Builder Initial Screen



To create the model class for our BSP application, follow these steps:

1. Start the Class Builder (transaction *SE24*), enter *ZCL_MVC_SAMPLE_MODEL_FL_QUERY* in the *Object type* field of the Class Builder initial screen (see **Figure 5**), and click on the *Create* button.

2. In the following dialog box (**Figure 6**), provide a description (e.g., *Model class for flight query*) and click on the *Save* button, which takes you to the main Class Builder screen.

✓ *"Create Class" Screen Settings*

Do not change anything other than the description on the "Create Class" screen — the default settings are correct. For more information on instantiation types, class types, and the other options on this screen, navigate to SAP Library → mySAP Technology Components → SAP Web Application Server → ABAP Programming and Runtime Environment → ABAP Programming → ABAP Objects.

3. Go to the *Properties* tab and click on the *Superclass* button. Enter the name of the base class (*CL_BSP_MODEL*) as shown in **Figure 7**, and save. All the public and protected components defined in the base class, such as methods and attributes, are now available in our new model class via inheritance.

4. Go to the *Attributes* tab. We want our application to search for flights and present the results, so we

Figure 6 Create Class Screen

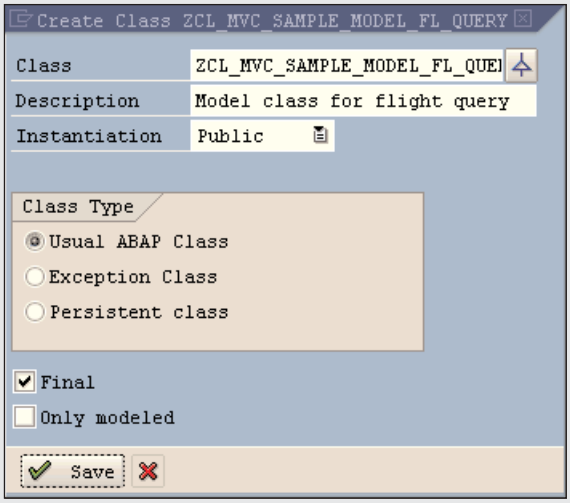


Figure 7 *Specify the Superclass for the New Model Class*

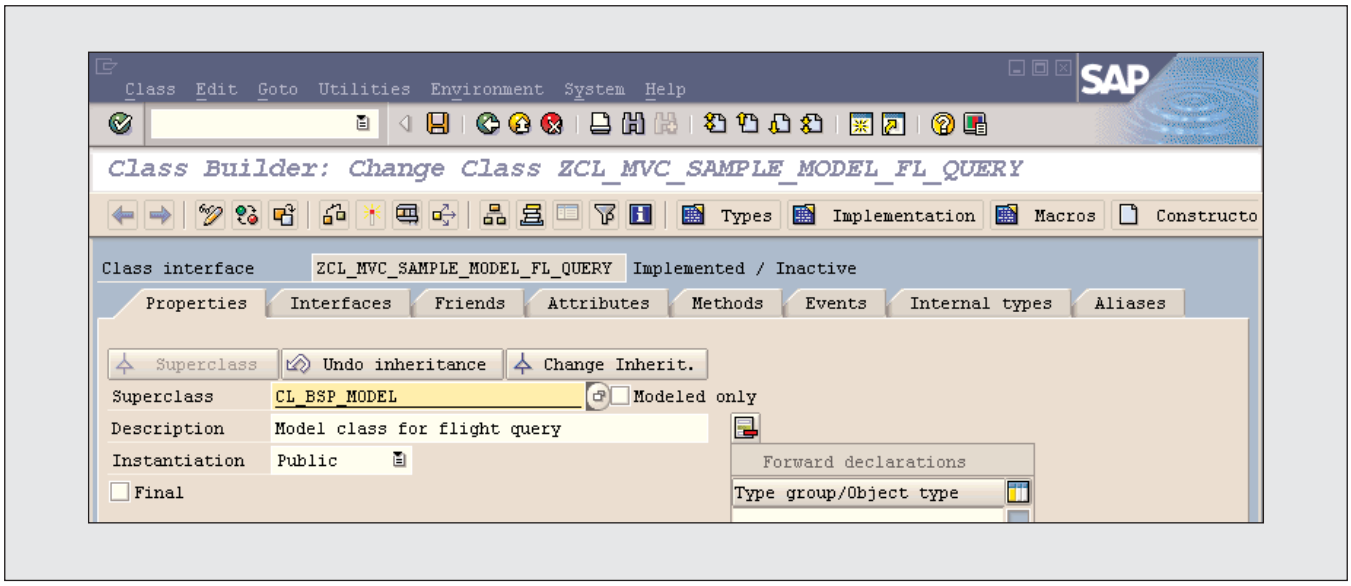
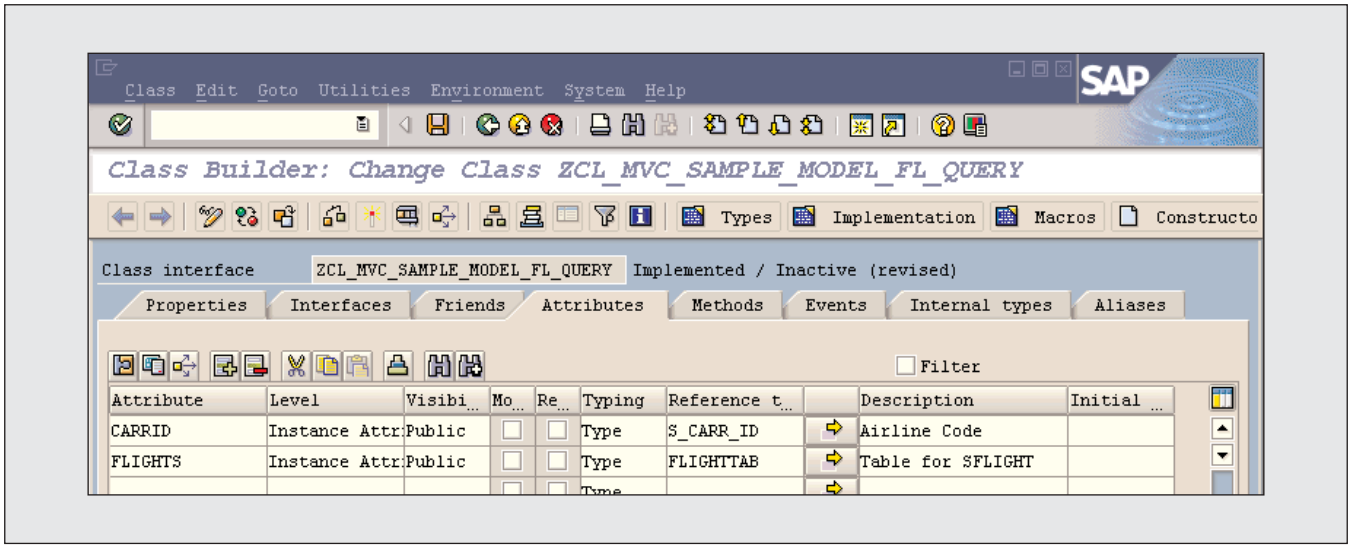


Figure 8 *Add Attributes to the New Model Class*

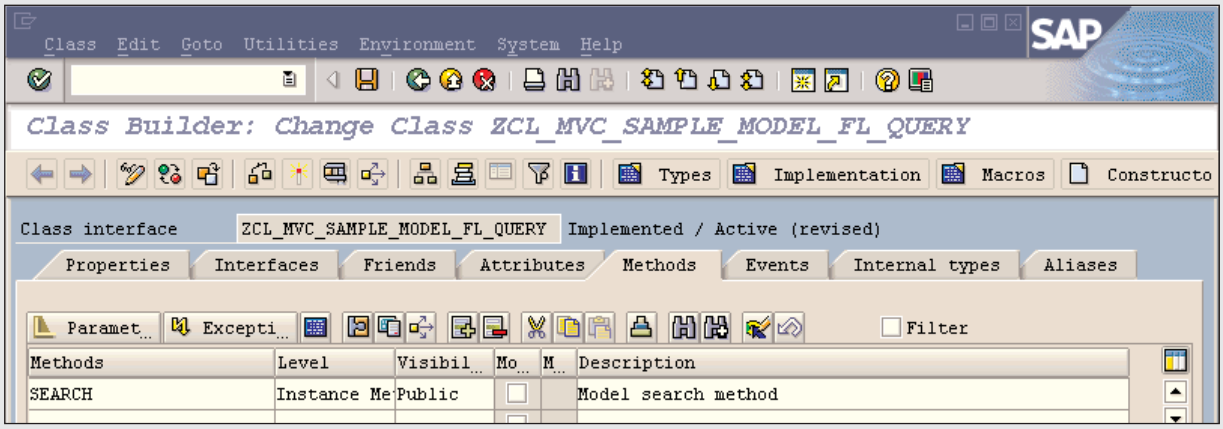


add the attribute *CARRID*, which will be used as the search criteria, and the attribute *FLIGHTS*, which will contain the results of the search (see **Figure 8**). Both attributes must be declared with *Public* visibility so that views and controllers[6] can access them. You must also enter a reference type — for the example, we use the standard SAP reference types *S_CARR_ID* and *FLIGHTTAB* for attributes *CARRID* and *FLIGHTS*, respectively.

Entering a description is optional, but it is good practice to provide one, at least for public methods and attributes. The remaining options shown here keep their default values.

[6] Remember that a BSP application can have multiple models, views, and controllers. Models can be used by multiple views and controllers, and each view/controller can use multiple models.

Figure 9 Add a Method to the New Model Class



✓ *Number of Attributes*

While there is no limit to the number of attributes you can define for a model, numerous attributes may be a sign that the model is too complex and used for too many different purposes. If we wanted to also provide business partner information in our application, for example, rather than extend our flight query model, we would create a second model dedicated to processing business partner data. Models can be reused in different applications, so it makes sense to build several with clearly defined responsibilities.

5. Go to the *Methods* tab. We want the application to use our model to look up flights, so we add the method *SEARCH*, as shown in **Figure 9**.

✓ *Number of Methods*

As with attributes, there is no limit on the number of methods you can define — it is restricted only by reusability and responsibility considerations.

Like the attributes, the method must be declared with *Public* visibility so that it can be accessed by controllers. Again, entering a description is optional. Save your settings.

6. Click on the edit icon ( ) in the *Methods* tab toolbar, which takes you to the method editor screen. When you saved the settings in the previous step, the system created the following empty implementation for the *search* method:

```
method search .
endmethod.
```

We want to add code in order to search for flights based on search criteria and store the result, so change the implementation by adding the code shown in bold in **Figure 10**.

The *select* statement will read all entries from database table *sflight*[7] that match the criteria defined in the *where* clause — i.e., all flights corresponding to the carrier ID (*carrid*) will be loaded and stored in table attribute *flights*.

We have not yet added any code that transfers the

[7] You can check the content of, and add entries to, table SFLIGHT via transaction *SE16* (Display Table Contents).

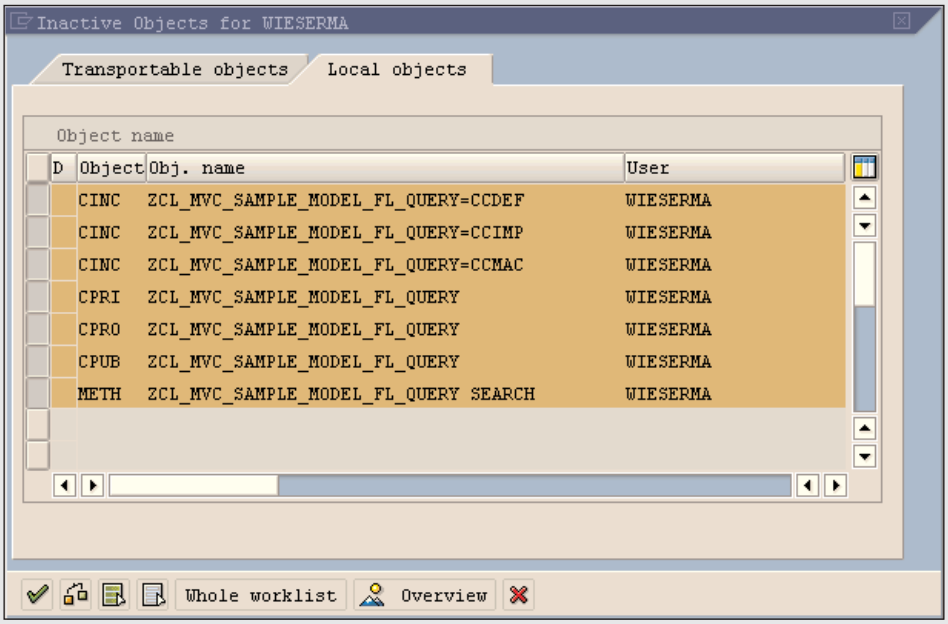
Figure 10 Modified Method Implementation

```
method search .

* Attribute carrid will have been set by data binding
* Select flights based on the carrier id
  select * from sflight
           into table me->flights
           where carrid = me->carrid.

endmethod.
```

Figure 11 Activation Dialog for the New Model Class



search criteria entered in the search view to attribute *carrid*, or that returns the search result stored in table attribute *flights* to the result view. In the next section, "Creating Views," you will see how view elements like input fields are "bound" to model attributes to enable data transfer. The controller sits between the model and the view, connecting the two and performing the actual data transfer. Data entered in the view's input field is automatically transferred to the bound model attribute, and if the model attribute is changed, the input field is automatically updated. To use data binding, a model's attributes must be defined with public visibility. The sidebar on pages 70-71 provides a more detailed discussion of the data binding concept.

7. Finally, save ( ), return to the main Class Builder screen ( ), and activate ( ) the newly created model class. Make sure all of the class components are selected in the activation dialog (**Figure 11**).

✓ Activating Class Components

A class consists of several components, including private (CPRI), protected (CPRO), and public (CPUB) components. Each method (METH) implementation is also a component. When a class is created, all of its components are inactive. To activate the class, all of its components must be selected in the dialog box shown in Figure 11 — if not, you will get error messages. If the visibility of a method or an attribute is not specified in the Class Builder, it will default to private. This will not affect the activation of the model class, but the controller and view will not be able to access it, and you will get error messages.

Let's now examine how to construct our flight query application's layout and how to implement its two views: the search view (*query.htm*) and the result view (*flights.htm*).

Creating Views

The layout of views can be implemented via the Web Application Builder in the Object Navigator (transaction *SE80*), which is the method we will use for our example application, or you can use a WebDAV-compliant[8] external design tool. For details on using an external WebDAV tool for BSP development, navigate to *SAP Library → mySAP Technology Components → SAP Web Application Server → Business Server Pages → Web Application Server → Web Applications and Business Server Pages → Programming Environment → BSP Development Tools → Implementing External Tools with WebDAV*.

Creating the Search View

We want our application to search for flights based on the carrier ID, so we need to create a view in which the user can enter a carrier ID and trigger the search.

To create the search view, follow these steps:

1. In the Object Navigator (transaction *SE80*) place the cursor on the name of the BSP application,

[8] WebDAV (World Wide Web Distributed Authoring and Versioning) is a set of standard HTTP extensions that allow users to collaboratively edit and manage files via the Internet. For more information, visit **www.webdav.org**.

Figure 12 Create the View

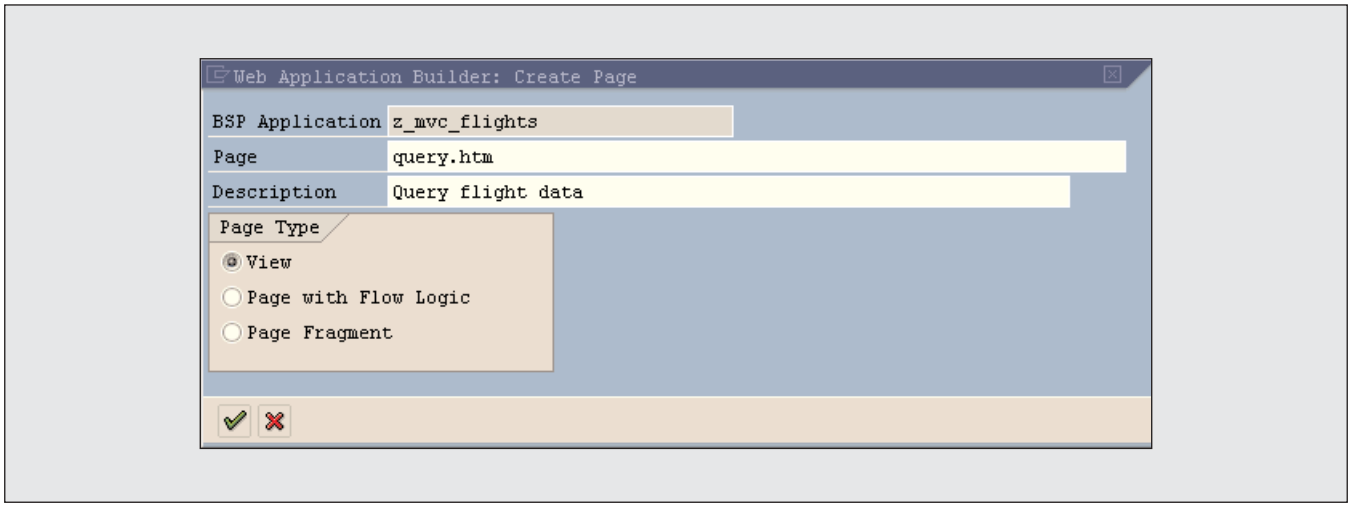


Figure 13 View Attributes

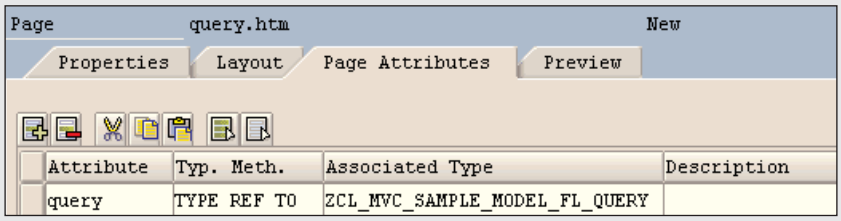


Figure 14 Default Layout Code for the Search View (query.htm)

```
 1 <%@page language="abap"%>
 2 <%@extension name="htmlb" prefix="htmlb"%>
 3
 4 <htmlb:content design="design2003">
 5   <htmlb:page title = "test ">
 6     <htmlb:form>
 7
 8       <htmlb:textView     text          = "Hello World!"
 9                           design        = "EMPHASIZED" />
10
11       <htmlb:button       text          = "Press Me"
12                           onClick       = "myClickHandler" />
13
14     </htmlb:form>
15   </htmlb:page>
16 </htmlb:content>
```

right-click, and select *Create* → *Page* from the context menu.

2. In the dialog box (**Figure 12**), enter a name for the page (*query.htm* in the example), a description (*Query flight data*), and select *View* as the page type. Click on the continue button ( ), which takes you to the view attributes screen.

3. Go to the *Page Attributes* tab and add a *query* attribute with *TYPE REF TO* as the typing method and *ZCL_MVC_SAMPLE_MODEL_FL_QUERY* (the model class we created in the previous section) as the associated type (see **Figure 13**). The *query* attribute will represent our model in the view.

4. Switch to the *Layout* tab. The layout editor will contain the default code shown in **Figure 14**. Delete lines 8-12 in Figure 14 and add the code

Figure 15 Modified Layout Code for the Search View (query.htm)

```
 1 <%@page language="abap"%>
 2 <%@extension name="htmlb" prefix="htmlb"%>
 3
 4 <htmlb:content design="design2003">
 5  <htmlb:page title = "Query flight data ">
 6     <htmlb:form>
 7
 8       <htmlb:label for="//query/carrid"/>
 9
10       <htmlb:inputField value="//query/carrid"/>
11
12       <htmlb:button text    = "Search"
13                     onClick = "onSearch" />
14
15     </htmlb:form>
16   </htmlb:page>
17 </htmlb:content>
```

shown in bold in **Figure 15** to create an input field for the search criteria, a label for the input field, and a button to trigger the search.

In line 8, we add a label for the carrier ID input field. The *for* attribute of the <*htmlb:label*> tag[9] is bound to the model attribute *carrid* to provide the text to be displayed (see the sidebar on pages 70-71 for a detailed discussion of data binding).

In line 10, we similarly add an input field to process the carrier ID — the *value* attribute of <*htmlb:inputField*> is also bound to the *carrid* attribute of the *query* model, so that the input field displays the content of the *carrid* attribute. If the user changes the value in the carrier ID input field, the value of the corresponding model attribute also changes.

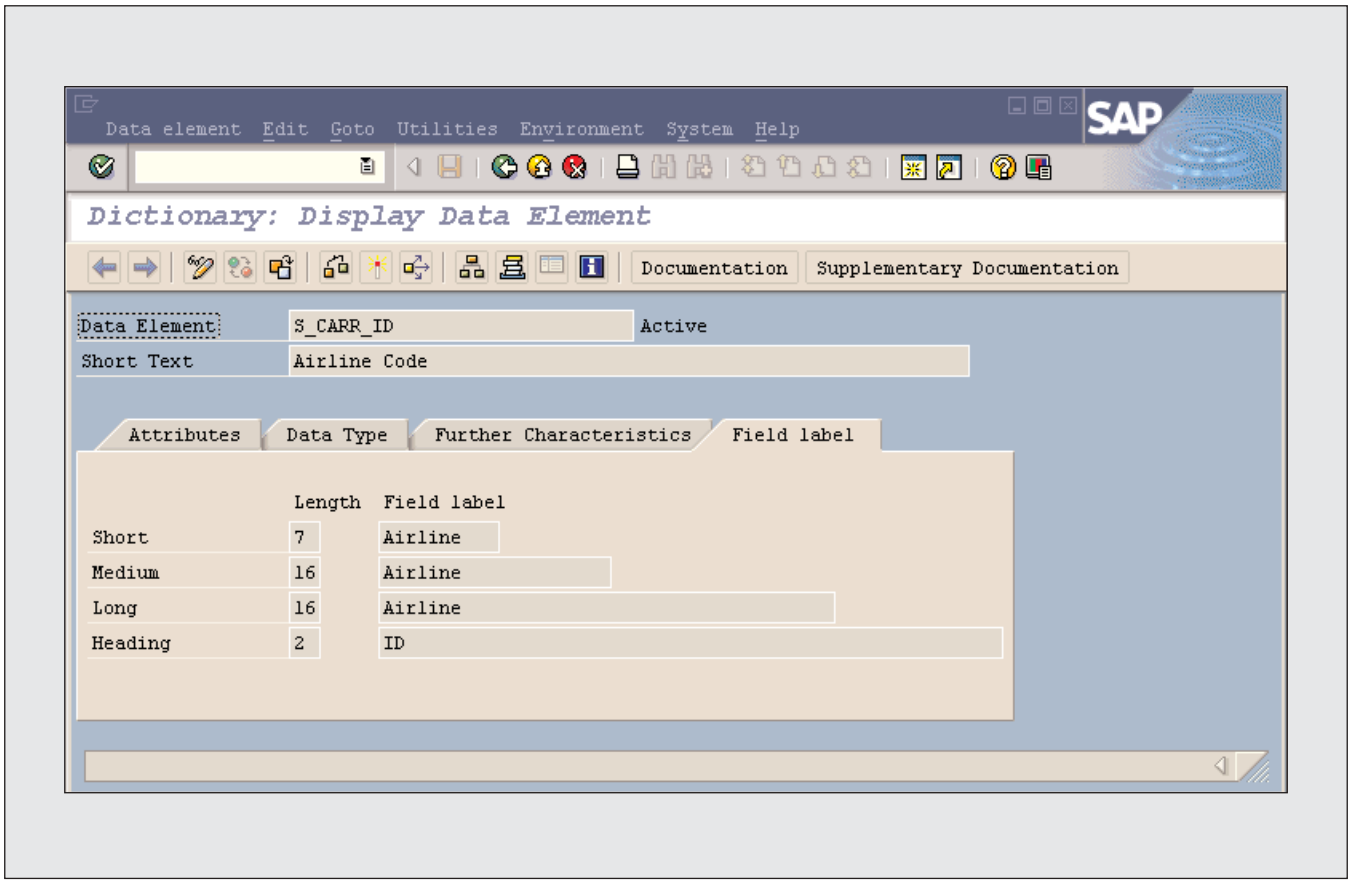
In lines 12-13 we add a button that, once clicked, will trigger the search. In the example, we hard-code the *text* attribute of the button to display *Search*. The content of the *onClick* attribute is important — the event handler defined in our controller's DO_HANDLE_EVENT method (see the upcoming section "Creating a Controller") will check to ensure that the event was caused by the *Search* button, and only then will trigger the search. The *onClick* attribute essentially links the button to the corresponding logic in the event handler, so be sure to specify *onClick* exactly as shown in line 13 and verify that the value is correct in the controller. Our application contains only one button, the clear source of the event, but most applications contain several — such as *Exit*, *Delete*, *Save*, *Cancel*, and so on — which makes differentiating them very important.

As you can see in Figure 15, both the label and the input field are bound to the carrier ID of the *query* model. Without data binding, the <*htmlb:label*> tag would look like the following:

```
<htmlb:label id ="carridLabel"
             text="Airline"
             for="carrid"/>
```

[9] See the sidebar "Introducing BSP Extensions" on page 59 for more on the *htmlb* extension.

Figure 16 Field Labels for the S_CARR_ID Data Type



✓ *Field Labels and Data Binding*

When using data binding, the field label displayed in the view is not the value of the model attribute — it is the value of the field label defined for the attribute's underlying data type. Our carrid model attribute has an SAP-defined data type, S_CARR_ID. The Data Dictionary (DDIC), transaction SE11, lists the field labels in ***Figure 16*** *for S_CARR_ID. By default, the system first tries to retrieve the "Medium" field label. If there is no "Medium" field label (defining field labels for a data type is optional), the system will next try to retrieve the "Long" field label, and if there is no "Long" field label defined, it will try the "Short" field label. If no field label texts are defined, no text is displayed for the field label in the view.*

In this case, the <*htmlb:label*> tag requires specific values for its *id* and *text* attributes. The *id* attribute identifies the BSP element and the *text* attribute contains the label text. The *for* attribute specifies the input field (in our example, the carrier ID) for which the label should be displayed. With data binding, the values of the *id* and *text* attributes can be automatically derived from the content of the *for* attribute.

Data Binding with the MVC Design Pattern

"Data binding" is one of the most important advantages of building a BSP application based on the MVC architecture — data binding enables data transfer between views and models in a generic way, so that you don't have to add or change controller coding to make it work with your models. To better explain the advantage of data binding, let's look at what happens after a user enters data in our example application and clicks on the *Search* button.

The BSP runtime ensures that the search request triggered by the user action (clicking the *Search* button) is sent to the corresponding controller. Before the controller can process the request, however, it must receive the user's input data, so the BSP runtime calls the controller's DO_HANDLE_DATA method to retrieve the data. The controller's DO_REQUEST method then sends the request to the event handler and creates a view filled with data (the search results). (These two methods, along with others, are discussed in detail in the "Creating a Controller" section of this article.)

Data binding is the missing link between the user interface (the view) and the business logic (the model). The binding is defined in a view's layout and determines which attributes of a model correspond to which screen elements of a view, such as input fields and tables (data binding can be used only for attributes, not for methods). So, for example, in the search view of our example flight query application, the model's *carrid* attribute is bound to an input field in which a user can enter a carrier ID. Data entered in the view is automatically transferred to the bound model attribute, and if the model attribute is changed, the corresponding input field is updated as well. The BSP runtime sends the data binding information to the controller, along with the user input data, where it is processed by the controller's DO_HANDLE_DATA method. Without data binding, you would have to add coding to the controller's DO_HANDLE_DATA and DO_REQUEST methods for each individual attribute to be passed between a view and the model, and each time you altered an attribute, you would have to alter the controller methods.

Let's explore the details of data binding a little further by taking a look at how we can use data binding in our example flight query application.

Using Data Binding

In order to pass the carrier ID search criteria from the search view of the example application to the model, and to then receive the search result in the result view, we have to bind attributes of the model to elements of the view layout.

In the layout of the search view, for example, we use the <*htmlb:inputField*> tag* to build the input field for the carrier ID (see line 10 of Figure 15). We want to bind the *value* attribute of <*htmlb:inputField*> to the attribute *carrid* of our *query* model** so we fill *value* with the following path specification:

```
<htmlb:inputField value="//query/carrid"/>
```

This example code creates an input field that takes its initial value from the *carrid* attribute of the *query* model. When a user enters data in the input field, it is automatically transferred to the model during the request processing.

---

* See the sidebar "Introducing BSP Extensions" on page 59 for more on the *htmlb* extension.

** Remember from step 3 of "Creating the Search View" (on page 67 of the article), the model is defined as a *query* attribute in the page attributes of the search view (the same is true for the result view). The *query* attribute represents the model in the view.

Depending on the type of attribute, the path specification will have one of the following syntaxes:

Simple Attribute	
Syntax	`value="//<model name>/<property name>"`
Example	`value="//query/carrid"`
Structure Attribute	
Syntax	`value="//<model name>/<structure name>.<field name>"`
Example	`value="//query/single_flight.carrid"`
Table Attribute	
Syntax	`value="//<model name>/<table name>[<row index>].<field name>"`
Example	`value="//query/flights[1].carrid"`

The search results for our example flight query application — all the flights that correspond to the search criteria specified by the user in the input field — are stored in a table attribute (*flights* in the example) that you can bind to a view layout element to display the results.

Using Setter and Getter Methods in Addition to Data Binding

If simple data binding is not sufficient — for example, if you want to convert the data in a certain way or set default values — you can also define setter and getter methods in the model class and use custom code to access model attributes. For each attribute accessed, an additional getter method must be defined that provides that attribute's metadata (information about length, data type, texts, etc.) to the BSP runtime — you do not need to call this method yourself, the BSP runtime calls it automatically.

The base class CL_BSP_MODEL includes templates for defining the various setter/getter methods and the metadata getter methods. These templates were automatically added to our example model class ZCL_MVC_SAMPLE_MODEL_FL_QUERY when it was created (via inheritance from base class CL_BSP_MODEL).

Using the templates is a good idea, since you can copy the method signatures — saving you time and helping prevent errors in the interface definitions (due to typos, for example). The section "Enhancing the Application" (on page 79 of the article) describes in detail the creation of a setter method and a metadata getter method. Keep in mind that your setter/getter methods must adhere to the following naming conventions; otherwise, the controller will not execute them:

Simple Attribute	Structure Attribute	Table Attribute
• `set_<attribute>`	• `set_S_<attribute>`	• `set_T_<attribute>`
• `get_<attribute>`	• `get_ S_<attribute>`	• `get_ T_<attribute>`
• `get_M_<attribute>` (metadata getter)	• `get_ M_S_<attribute>` (metadata getter)	• `get_ M_T_<attribute>` (metadata getter)

Once you have implemented a setter or getter method in the model class, you are done. You don't have to make any modifications in your views or in the data binding. The BSP runtime takes care of either executing the simple data binding or using your new setter/getter methods.

Creating the Result View

In addition to searching for specific flights, we want our application to display the results in a table. The result view will host this table, which gets its content from the model.

To create the result view, follow these steps:

1. Once the first view is created (the search view we created in the previous section), the BSP application automatically contains a new folder called *Views*, which you can use to create any additional views for this application. Simply right-click on the *Views* folder and select *Create*.

2. Create a new view in the same way that you created the previous view, but instead using the name *flights.htm* and the layout shown in **Figure 17**. Modify the default layout code with the coding shown in bold, as we did in Figures 14 and 15. With the binding of the *table* attribute to the *flights* attribute of the *query* model (line 9), the <*htmlb:tableView*> is automatically filled with the search result. Tags <*htmlb:tableViewColumn*> and <*htmlb:tableViewColumns*> specify which fields of the *flights* table are displayed. Without any restrictions, all the fields of the table would be shown.

> ✓ ***Don't Forget…***
>
> *Be sure to add the model as a query attribute of the result view, as we did for the search view in the previous section, so that the result view can access the result list.*

Figure 17 Modified Layout Code for the Result View (flights.htm)

```
 1 <%@page language="abap"%>
 2 <%@extension name="htmlb" prefix="htmlb"%>
 3
 4 <htmlb:content design="design2003">
 5   <htmlb:page title = "Flights display ">
 6     <htmlb:form>
 7
 8       <htmlb:tableView id="myFlights"
 9                        table="//query/flights"
10                        visibleRowCount="10" width="10">
11        <htmlb:tableViewColumns>
12           <htmlb:tableViewColumn columnName="carrid"/>
13           <htmlb:tableViewColumn columnName="connid"/>
14           <htmlb:tableViewColumn columnName="fldate"/>
15           <htmlb:tableViewColumn columnName="seatsmax"/>
16           <htmlb:tableViewColumn columnName="seatsocc"/>
17         </htmlb:tableViewColumns>
18       </htmlb:tableView>
19
20     </htmlb:form>
21   </htmlb:page>
22 </htmlb:content>
```


✓ *Text Translation and Data Binding*

Texts displayed in the view can be hard-coded, taken from Data Dictionary (DDIC) definitions, or retrieved from the Online Text Repository (OTR). OTR texts can be created either from directly within a BSP page or via transaction SOTR_EDIT. View elements that use OTR texts contain references called "aliases" that specify which texts should be used. OTR and DDIC texts are created, stored, and translated independently of view layouts. During runtime, the data binding specifications and the OTR aliases are resolved, providing the view layout elements with text in the user's logon language, so you don't have to provide coding for different languages. Text translation and OTR are not covered in this article; for more details, navigate to SAP Library → mySAP Technology Components → SAP Web Application Server → Business Server Pages → Web Application Server → Web Applications and Business Server Pages → Programming Environment → Internationalization and Translation.

✓ *Adding BSP Extension Tags*

While you can add BSP extension tags to your view layout code manually, you can also drag and drop tags from a list, which is much easier than trying to add them from memory. Simply click on the Tag Browser button in the Object Navigator (transaction SE80) and select BSP Extensions → Transportable to display SAP-defined and any transportable custom extensions and tags (select BSP Extensions → Local to view locally created custom extensions and tags that belong to a non-transportable package). You can then drag and drop tags from the list into the layout editor.

Creating a Controller

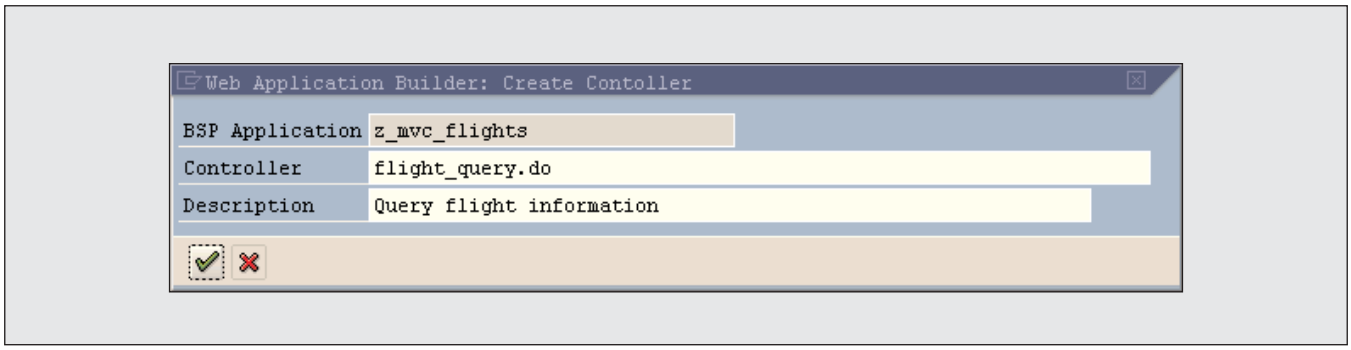
A controller processes requests and determines how the application should react by communicating with models and calling views (remember that a controller can work with multiple models and views). In our example application, for instance, the controller first calls the search view (*query.htm*), then uses the model (via the *query* attribute stored in the search view's page attributes) to search for flights. Finally, the controller calls the result view (*flights.htm*), which displays the selected flights via the *query* attribute (i.e., the model) stored in the result view's page attributes. In a BSP application, a controller is the starting point of each navigation of the application — the user actions in the view trigger a request, but it is the controller's task to decide what happens next.

A controller is derived from the base class CL_BSP_CONTROLLER2. The derivation happens automatically when the controller is created using the Web Application Builder in the Object Navigator (transaction *SE80*).

The core methods of the base class CL_BSP_CONTROLLER2, which are used to initialize the controller's data, transfer data, and react to user-triggered events, include the following:

- DO_INIT — Creates an instance of the model (the ZCL_MVC_SAMPLE_MODEL_FL_QUERY class in the example).
- DO_HANDLE_DATA — Transfers data from the request to the model attributes. Since we are using data binding for our example application, we do not need to modify this method; we can just keep the coding provided by CL_BSP_CONTROLLER2. Without data binding, you would have to add custom code to the method to handle the data transfer.

Figure 18 *Create a Controller*



- DO_REQUEST — Dispatches the request to the event handler, determines and creates a response view, sets the view data, and calls the response view. We will use this method to call the search and result views for our example application, as well as to provide the views with a reference to the model.

- DO_HANDLE_EVENT — Processes events like button clicks and triggers the corresponding model logic (in the example, this method processes the click on the *Search* button).

For our example flight query application, we want the controller to call the search view (*query.htm*), execute the search for flights, and then call the result view (*flights.htm*) to display the selected flights. Follow these steps:

1. In the Object Navigator (transaction *SE80*), place the cursor on the newly created application (*z_mvc_flights*) and select *Create* → *Controller* from the context menu. On the pop-up (**Figure 18**), enter the controller name (*flight_query.do* in the example) and a description (*Query flight information*), and click on the continue button ( ), which takes you to the controller attributes screen (**Figure 19**).

2. We next need to create an ABAP Objects class for the controller logic and assign it to the controller we are creating. In the *Controller Class* field, enter *ZCL_MVC_SAMPLE_CTRL_FLIGHT* as the name of our new class.[10] All other settings should keep their default values. Save and double-click on the *Controller Class* field. The system automatically creates the specified controller as a derivation of base class CL_BSP_CONTROLLER2 and opens the Class Builder.

3. Switch to change mode ( ) and select the *Attributes* tab (see **Figure 20**). Add the attributes *QUERY*, which is a reference to the model, and *VIEW_NAME*, which will be used to store the name of the response view. Assign the *QUERY* attribute *Public* visibility (so the view can access the model), give it a *Type Ref* typing (to enable referencing), and for the reference type enter *ZCL_MVC_SAMPLE_MODEL_FL_QUERY* (the model class we created earlier). The description is derived from the model class definition.

   Assign the *VIEW_NAME* attribute *Private* visibility (it is only needed by the controller) and enter *STRING* as the reference type, so we can pass it to method *create_view* (*create_view* will be used in the DO_REQUEST method, which defines the view name parameter as a string). The remaining settings can keep their default values.

4. Switch to the *Methods* tab, which automatically lists all the methods that controller class

[10] If you are reusing an existing controller, or if you created the controller in the Class Builder, enter the name of the existing class here.

Figure 19 Controller Attributes

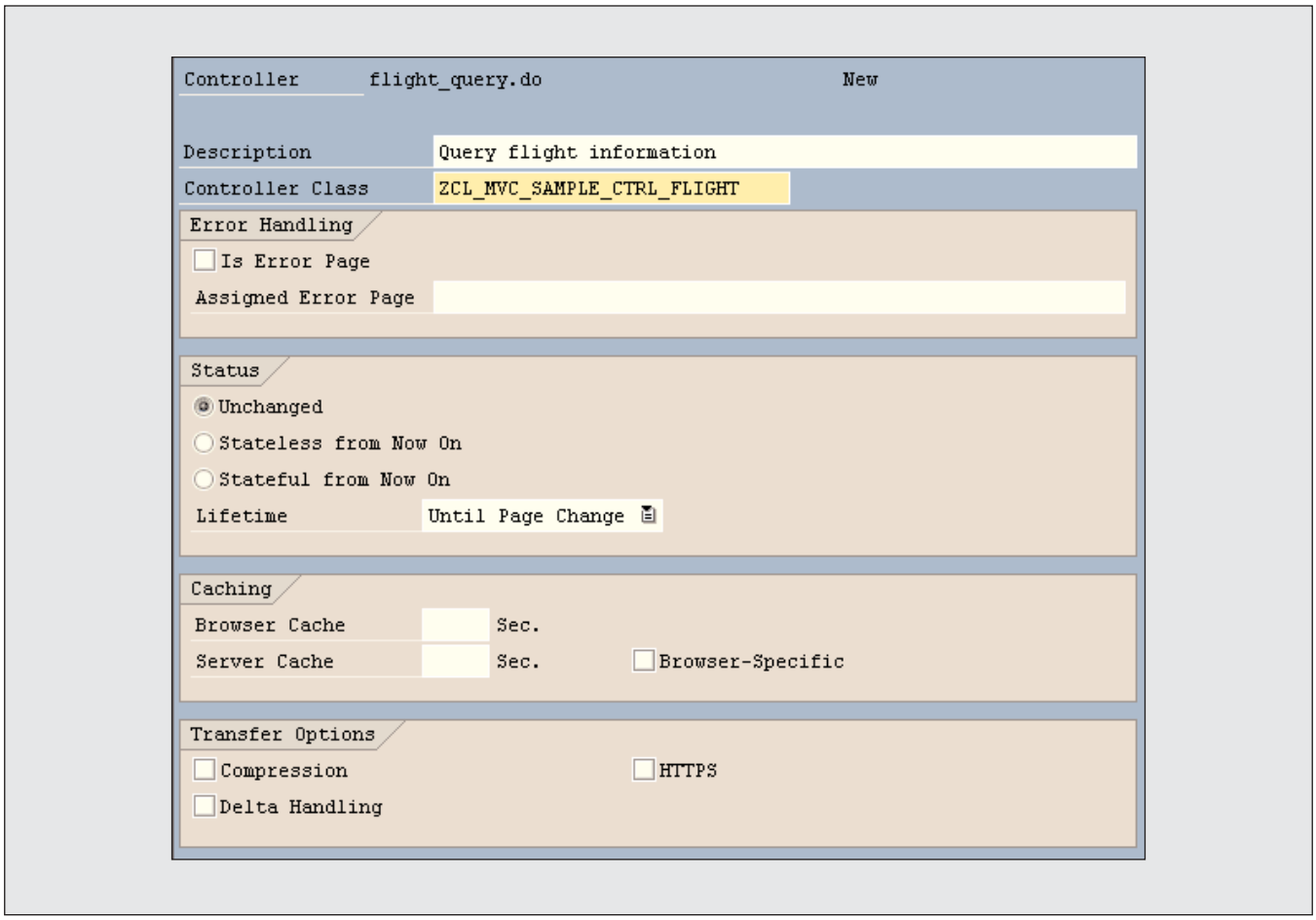
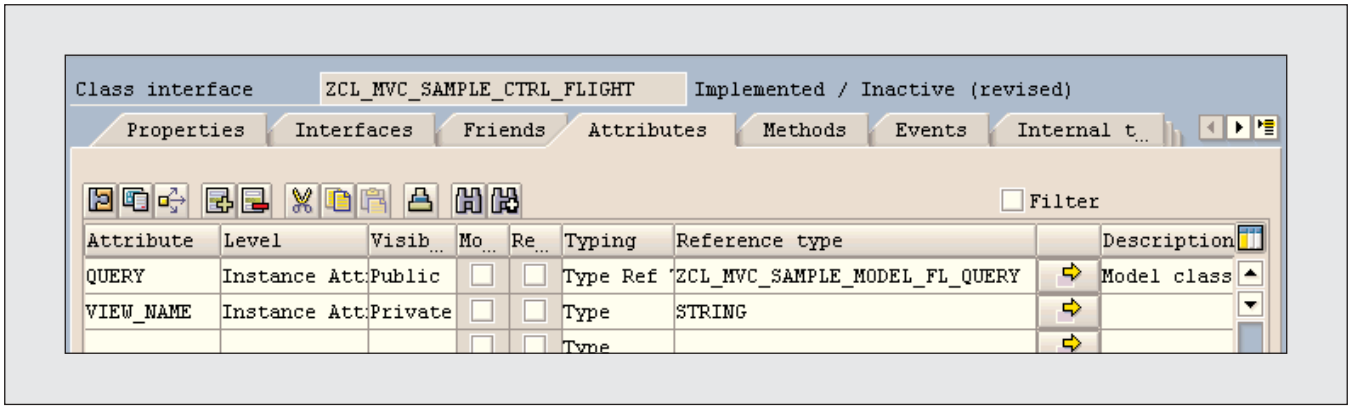


Figure 20 Add Attributes to the New Controller Class



ZCL_MVC_SAMPLE_CTRL_FLIGHT inherited from base class CL_BSP_CONTROLLER2. Place the cursor on method DO_INIT, click on the redefine button

✓ *Redefining the Controller Class Methods*

To use our model and views in controller class ZCL_MVC_SAMPLE_CTRL_FLIGHT, we have to redefine some of the methods inherited from base class CL_BSP_CONTROLLER2 using the Class Builder's redefine function ( ). The redefinition steps automatically add commented coding (shown in bold in the DO_INIT example below):

```
method do_init .
* CALL METHOD SUPER->DO_INIT.
* .
endmethod.
```

If you uncomment those lines, you would execute the method implementation of the parent class. This is helpful when you want to carry out the parent logic and some additional custom logic. We don't want to reuse functionality of the parent class in the example, however, so be sure to delete the commented code before adding the new code.

( ), and replace the existing commented code with the code shown in bold in **Figure 21**.

The added code tells the controller to create a model instance of type *ZCL_MVC_SAMPLE_MODEL_FL_QUERY*, identify it as *query*, and assign it to the *query* attribute created for the controller in the previous step. After *do_init* is executed (this happens automatically when the controller is called for the first time), the model's search functionality can be accessed and used.

5. Return to the *Methods* tab, place the cursor on method DO_REQUEST, click on the redefine button ( ), and replace the existing commented code with the code shown in bold in **Figure 22**.

   In line 8 we dispatch the handling of the event (i.e., the click of the *Search* button) to method *do_handle_event*. If any flights are found, *view_name* will change to *flights.htm*. In lines 12-14 we ensure that *view_name* contains the name of the search view (*query.htm*) if no flights are found or *do_request* is executed for the first time (remember that we are using one controller for both views). The next view is created in line 17. In line 21 we set a reference to the *query* model in the view. A view is internally represented by an automatically generated ABAP Objects class, and the view's page attributes are defined as private attributes of the generated class. The *name* parameter of method *set_attribute* must match the entry in the page attributes — in this case, *query* (see step 3 in "Creating the Search View"). In line 22 we call the view.

6. Return to the *Methods* tab, place the cursor on

Figure 21 Add the Model Instantiation

```
1 method do_init .
2
3 * Create an instance of the search model
4   query ?= me->create_model(
5                class_name = 'ZCL_MVC_SAMPLE_MODEL_FL_QUERY'
6                model_id   = 'query' ).
7
8 endmethod.
```


Figure 22 Add the View Processing

```
 1 method do_request .
 2
 3 * Define a variable for the view we are about to create
 4   data view type ref to if_bsp_page.
 5
 6 * Dispatch the input so that the event handler can process
 7 * events like when the agent clicks on the Search button
 8   me->dispatch_input( ).
 9
10 * The attribute me->view_name can be set by the event handler
11 * to change navigation to the flights display
12   if me->view_name is initial.
13     me->view_name = 'query.htm'.
14   endif.
15
16 * Create the response view
17   view = me->create_view( view_name = me->view_name ).
18   if view is bound.
19 *   Add a reference to the model instance to the view,
20 *   so that the view can access the model
21     view->set_attribute( name = 'query' value = me->query ).
22     me->call_view( view ).
23   endif.
24 endmethod.
```

Figure 23 Add the Event Handling

```
 1 method do_handle_event .
 2
 3 * Check that the event has been triggered by the search button
 4   if htmlb_event is bound and
 5      htmlb_event->server_event = 'onSearch'.
 6 *   Execute the search implemented in the query model
 7     me->query->search( ).
 8
 9 *   Flights have been found, new target is the hit list display
10     if not me->query->flights[] is initial.
11       me->view_name = 'flights.htm'.
12     endif.
13   endif.
14
15 endmethod.
```

method DO_HANDLE_EVENT, click on the redefine button (), and replace the existing commented code with the code shown in bold in **Figure 23**.

The added code processes the click on the *Search* button. In line 5 we verify that the event was triggered by a click on the *Search* button defined in the search view (remember from step 4 of the section "Creating the Search View," we set the *onClick* attribute of the *Search* button to *onSearch*, so make sure the check in line 5 and the *onClick* definition match, both in terms of value and also capitalization). In line 7 we execute the *search* method of the model. In lines 10-12 we specify the next visible view by changing *view_name* to *flights.htm*, in case flights that match the entered carrier ID are found.

7. Return to the *Methods* tab and activate ( ) the controller class.[11]

> ✓ ***Supporting Multiple Views***
>
> *The number of views a controller supports is highly dependent on the complexity of the logic needed to react to events, set the model reference, trigger and execute business logic, and call views. However, too many views will overload the controller and make it difficult to maintain, so bear this in mind when building your own MVC-based applications.*

[11] Since we are using data binding, you do not need to redefine the DO_HANDLE_DATA method — data binding enables the controller to use the default implementation provided by base class CL_BSP_CONTROLLER2.

8. Return to the controller attributes screen ( ), double-click on the BSP application name (*z_mvc_ flights*), and activate ( ) the application.

Now that all the components of the application have been created and activated, we are ready for a test run.

Testing the Application

To test our application, start the Object Navigator (transaction *SE80*), right-click on the application name, and select *Test* from the context menu. To start, a BSP application needs either a page with flow logic or a controller. If there are multiple controllers and/or pages, you have to specify a default start page or select the component you want to start with. Our application has only a single controller, so that controller will be started by default, which in turn triggers the execution of method DO_INIT.

Enter *LH* as the search criteria, as shown in **Figure 24**, and click on *Search*. At this time, data binding kicks in and transfers *LH* to the model via the controller's DO_HANDLE_DATA method. DO_REQUEST and DO_HANDLE_EVENT trigger the SEARCH method to complete the process, which results in **Figure 25**.

Now that we've confirmed that our application is running smoothly, let's look at how we can build on its functionality a bit.

Figure 24 Enter Search Criteria to Test the Application

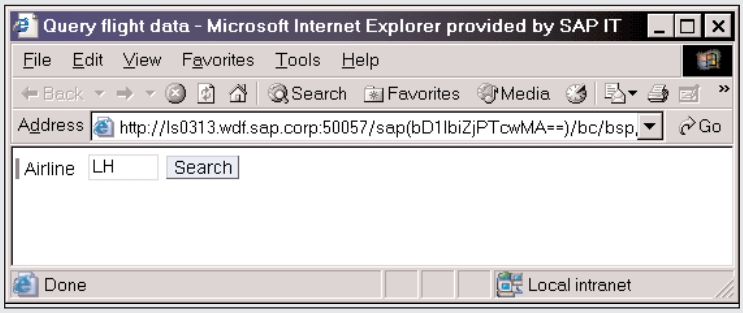
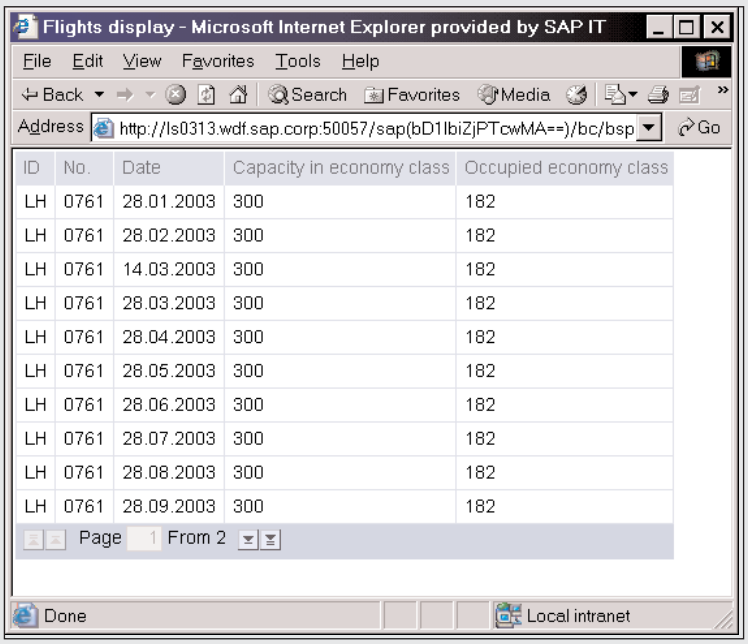


Figure 25 Search Results



Enhancing the Application

In our simple implementation, if no search criteria is entered, no flights will be selected. Let's look at how we can change the data binding from a simple transfer to a more complex one with data processing that will enable us to create a default search criteria. To accomplish this, we need two more methods — a setter method to set a default search criteria in the model, and a getter method to retrieve the metadata of the carrier ID attribute (remember from the sidebar on pages 70-71 that any attribute associated with a getter or setter method requires an additional getter method that provides the attribute's metadata). Follow these steps:

1. Start the Class Builder (transaction *SE24*), enter model class *ZCL_MVC_SAMPLE_MODEL_FL_QUERY* in the *Class* field, and switch to change mode ( ).

2. Go to the *Methods* tab and copy method IF_BSP_MODEL_SETTER_GETTER~_SET_XYZ (inherited from base class CL_BSP_MODEL) to create a new method with the name *SET_CARRID* (see the sidebar on pages 70-71 for more on setter/getter method naming conventions).

3. Click on the edit icon ( ) to change the implementation of the new setter method (SET_CARRID). First remove all lines except for the first and last, and then add the code shown in bold in **Figure 26**.

Figure 26 Implement SET_CARRID

```
 1 method set_carrid .
 2
 3   me->carrid = value.
 4
 5 * Pre-fill carrid if no value
     has been provided
 6   if value is initial.
 7     me->carrid = 'LH'.
 8   endif.
 9
10 endmethod.
```

Figure 27 Metadata Getter Method (GET_M_CARRID)

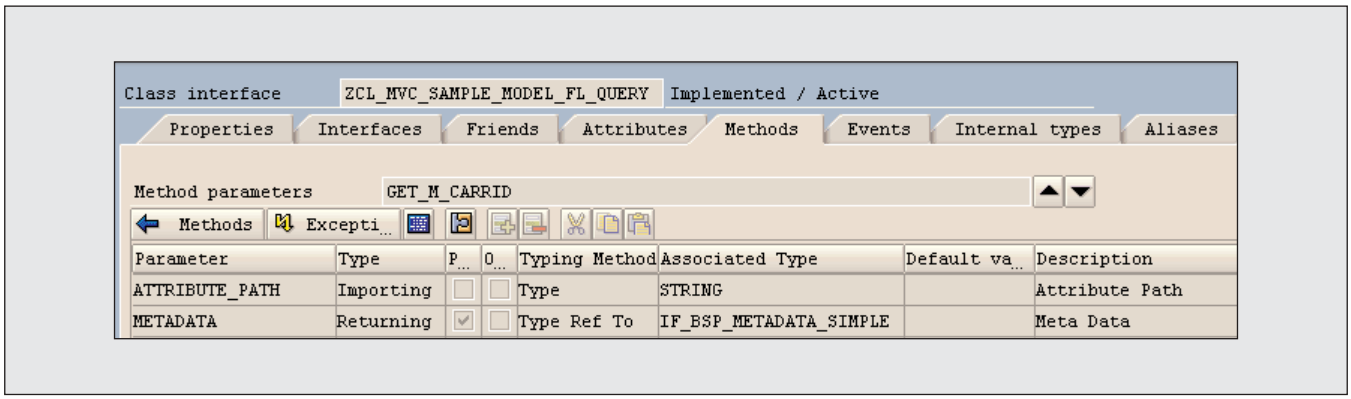


Figure 28 Implement the GET_M_CARRID Metadata Getter Method

```
 1 method get_m_carrid .
 2
 3   data: type_definition  type ref to cl_abap_elemdescr,
 4         ddic_description type        dfies.
 5
 6
 7 * Get the runtime type definition for carrier id attribute
 8   type_definition ?=
 9      cl_abap_typedescr=>describe_by_name( 'S_CARR_ID' ).
10
11 * Get the Data Dictionary definition of the carrier id from
12 * the runtime type definition
13   ddic_description = type_definition->get_ddic_field( ).
14
15 * Create a metadata description instance based on the
16 * DDIC information
17   create object metadata type cl_bsp_metadata_simple
18      exporting info = ddic_description.
19
20 endmethod.
```

SET_CARRID takes the *value* (i.e., the user input) that was transferred from the view (line 3), checks if it is initial (line 6), and if so sets *LH* as the default value in the *carrid* attribute (line 7).

4. Save and return to the *Methods* tab. Copy method IF_BSP_MODEL_SETTER_GETTER~_GET_M_XYZ (inherited from base class CL_BSP_MODEL) to create a new method with the name *GET_M_CARRID*. Your method's signature will look like **Figure 27**.

5. Click on the edit icon ( ) to change the implementation of the new metadata getter method (GET_M_CARRID). Remove all lines except for the first and last and add the code shown in bold in **Figure 28**.

✓ *Using Description Objects to Create Metadata*

Each data type (elementary, like strings, or complex, like tables or data references) and object type (classes and interfaces) has exactly one description object. Static methods of CL_ABAP_TYPEDESCR, such as DESCRIBE_BY_NAME or DESCRIBE_BY_OBJECT_REF, can be used to create individual description objects that provide information about length, texts, and other properties, which can in turn be used to create the metadata in metadata getter methods. Since carrid is a simple attribute, the created metadata object is of type CL_BSP_METADATA_SIMPLE. If we had to build a metadata getter method for a structure or table attribute, we would instead use classes CL_BSP_METADATA_STRUCT and CL_BSP_METADATA_TABLE accordingly.

The metadata getter has to create a metamodel to describe properties such as data type, field labels, display length, and internal length for the *carrid* attribute. Since the *carrid* attribute is bound to a label as well as an input field, it will be executed twice — once to provide information for the input field (e.g., display length) and once to provide the label text. To create the metamodel, we use Runtime Type Identification (RTTI) to determine the *carrid* attribute's Data Dictionary (DDIC) definition.

In line 3 we define variable *type_definition* to contain the RTTI. Variable *ddic_description* in line 4 is of type *dfies* and will be used to store the retrieved DDIC attribute information. In lines 8-9 we create an actual RTTI for the *carrid* attribute by passing *S_CARR_ID*, the SAP-defined type of attribute *carrid*. The constructor of class *CL_BSP_METADATA_SIMPLE* requires an import parameter of type *dfies*, so in line 13 we retrieve the DDIC information into variable *ddic_description* from *type_definition*. Finally, in lines 17-18 the *metadata* for *carrid* is created based on the retrieved DDIC definition. As you can see in Figure 27, the variable *METADATA* is defined as a return parameter, so the code in line 17 will ensure that the description of *carrid* is returned.

6. Save ( ), return to the main screen of the Class Builder ( ), and activate ( ) the model class.

From now on, each time the search is triggered without providing a search criteria, the carrier will default to *LH*.

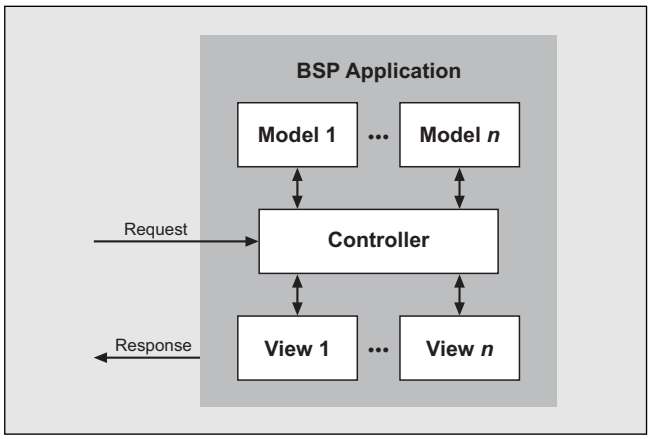
Some additional enhancements you could try would be to extend the search criteria — currently the search is limited to the carrier ID. To do this, you would need to extend the model and the view layout. You could also experiment with the setter/getter methods by ensuring that the carrier ID is always entered in uppercase — currently, the search does not find any flights if the criteria is entered in lowercase.

So far we have taken a detailed look at how to build a BSP application, how to create a model, a view, and a controller for the application, and how to use data bindings to automate data transfer between the model and the view. Let's now take a brief look at some variations of the MVC design pattern that can enhance the maintainability of your BSP applications.

Varying the MVC Design Pattern

There are several configurations of the MVC pattern. The simplest consists of one controller, which acts as the central dispatcher, and several views and models. This is the configuration we used for our example application. Among other tasks, this controller is responsible for handling all input requests, model

Figure 29 Using a Single Controller



initializations, data transfers between views and models, and determination of the response view. **Figure 29** is a graphical overview of this basic configuration.

As you can see, an application that uses a single controller can lead to a very cumbersome controller as more and more views and models are added to the expanding application. It would quickly become too difficult to maintain the central controller.

This problem can be solved by coupling each view with its own "view controller" (see **Figure 30**). Instead of one central controller serving all views, dedicated view controllers handle only their associated views. Now that the application consists of components with less intricate logic and more well-defined responsibilities, it is easier to apply changes because enhancements affect smaller areas, thus reducing the risk of errors.

Using multiple controllers for multiple views is still not ideal for large applications where data has to be shared between several views and controllers, however. Now that we have multiple controllers, we no longer have a central instance that can access all the application data.

This dilemma becomes obvious when adding a view that needs data from two different models belonging to two different controllers. This predicament is resolved by adding "custom controllers," which have no associated view, are accessible from each controller,[12] have their own data models, and will exist until they are explicitly destroyed.[13] In keeping with the principle of small, well-defined building blocks, you are not limited to one custom controller — you can create as many as necessary.

Another problematic aspect of using multiple controllers for multiple views is the handling of models. The model-related tasks of a controller include creating instances, handling models that belong to a custom

[12] Each custom controller must be instantiated in the main controller.

[13] This only applies to stateful applications, where the application keeps track of the state of the user interaction. Note that in this case, the lifetime of the controller must be set to *session*.

Figure 30 Using Multiple Controllers for Multiple Views

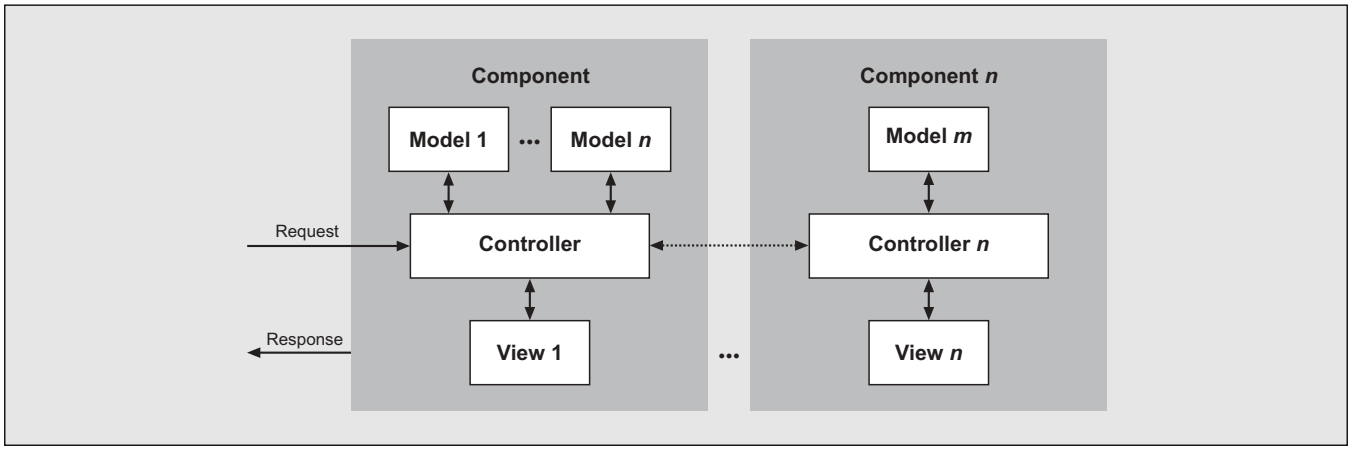
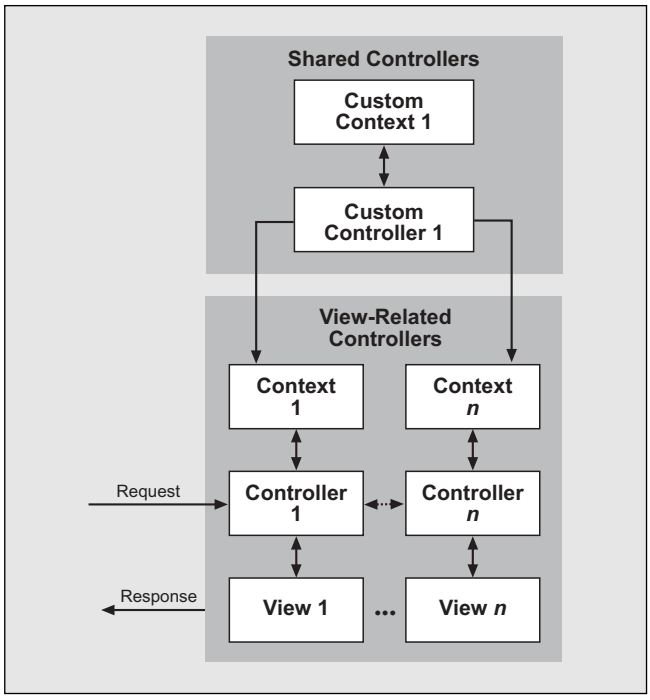


Figure 31 Using Custom Controllers and Controller Contexts



controller, and so on. In bigger applications, this becomes a complexity issue. The solution is to add a "controller context," which bundles the model-related activities into one block,[14] adding a separation between the controller and the models.

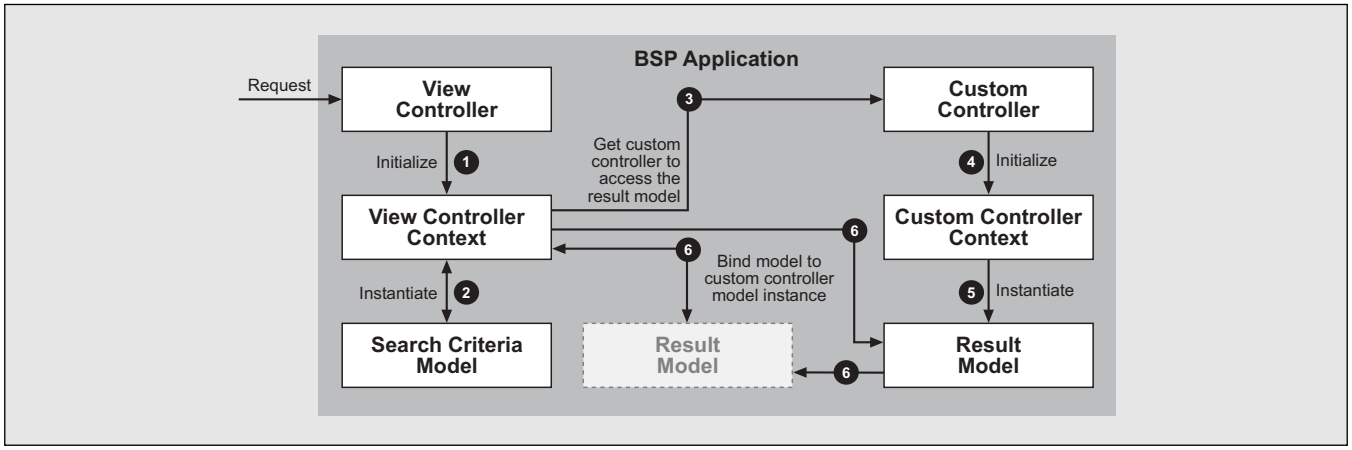
[14] The controller context is created as a separate class. The controller contains an attribute with a reference to this additional class.

Figure 31 shows a graphical overview of using custom controllers and controller contexts.

To illustrate the advantage of using custom controllers and controller contexts, let's look at our example application. The search criteria and the result list belong to the same model, which in turn belongs to one controller. Wouldn't it be nice if we could reuse the result list in other controllers and views?

Let's say that we've reconfigured our example application so that the search view contains a view controller that has a controller context, which in turn defines two models, one for the search criteria and another for the search result. **Figure 32** shows a graphical overview of how such an architecture would work. Since other views and controllers will need access to the result model and its data, the model is defined as part of the custom controller. Instead of being handled directly by the controllers, the models are handled by the controller contexts. Let's take a closer look at how this works. When the view controller is initiated by the BSP runtime, it first initializes its controller context (❶). The view controller context then creates a new instance of the search criteria model (❷) and gets a custom controller (❸), which triggers the creation of a custom controller context (❹) and the result model (❺). The view controller context then takes the result model instance created by the custom controller context and binds it to its own result model (❻), so that the view controller, and

Figure 32 Advanced MVC Pattern Example



therefore the view, is unaware of the result model's origins. Because of this unawareness, data binding and processing can happen as with any other model — with the exception that changes to attributes will also affect any custom controller contexts and any view controller contexts that bind the result model.[15] In our example, each view using the result model will automatically show the actual data — you don't have to worry about synchronizing different data sources.

The Interaction Center Web Client application, part of mySAP CRM, uses the advanced MVC pattern extensively, combining hundreds of views, view controllers, custom controllers, controller contexts, and models. The advanced MVC architecture encapsulates view layout, flow control, content data processing, and business data modeling, but even the simpler variants combined with data binding can help you keep your applications easier to maintain and understand.

Conclusion

If properly applied, the MVC design pattern will result in a very maintainable and extensible application. In a simple application, the intricate architecture can be a downside of using the MVC approach, but for large, complicated enterprise application design, the MVC approach has major advantages, such as the ability to encapsulate and reuse business logic and support for building well-structured, maintainable, and reliable applications. Using the MVC design pattern requires in-depth design and planning before the development work can start, but for complex and long-term applications, the hard work will pay off.

[15] Custom controllers and controller contexts are advanced MVC development techniques. A detailed discussion of their implementation and use is beyond the scope of this article.

Ken Huang is a senior software engineer/ consultant living in the Silicon Valley. He has more than 10 years of professional application software development experience. His wide-ranging experience spans SAP R/3, distributed architectures, multi-tiered Internet/intranet systems, call center application systems, wireless/mobile application systems, and interactive voice response (IVR) technology. His consulting clients include HP, Compaq, SAP, Wells Fargo Bank, and First Data Corp. He can be reached at ken_huang88@yahoo.com.

Markus Wieser has an engineering degree in software development and business organization from the Higher Technical Training and Testing Institute of St. Poelten, Austria. Markus joined SAP AG in 1996 and transferred to SAP Labs, Palo Alto, in 1998. Since then he has been working on the Interaction Center application of mySAP CRM. He is currently focusing on the Interaction Center Web Client, an application based on SAP BSP technology and the MVC architecture. Before joining SAP, Markus worked for four years at Siemens Austria, developing graphical planning tools for signaling networks. He can be reached at markus.wieser@sap.com.

