Model View Controller (MVC)

- Model View Controller (MVC)
  - MVC Design Pattern
  - Using MVC for BSP
    - Creating a Controller
    - Creating a View
    - Testing Controllers
    - Calling (Sub) Controllers
    - Calling a View
    - Creating Error Pages
  - From Pages to Controllers
  - Call Options of BSP Components
  - Navigation
  - Lifetime
  - Data Binding
    - Calling the Model Class by the Controller
  - Components
    - Process Flow
    - Creating Your Own Components
      - Creating the Top-Level Controller
      - Creating Components
      - Calling Components

Determining Input Processing

Class CL_BSP_CONTROLLER2

Examples of Architecture

BSP Application with Controllers and Views

BSP Application with Several Views per Controller

Combination of the Previous Examples

Calling Controllers of Other Applications

Calling Several Controllers from a View

Model View Controller Tutorial

Creating a Controller

Creating a View

Calling a Controller

Model View Controller (MVC)

Use

SAP Web Application Server 6.20 has implemented the **Model View Controller** (MVC) design pattern, which is widely used in the user interface programming field and which has proved its worth, as an extension of the previous BSP implementation model. Its controller-based use ensures an even clearer distinction between application logic and presentation logic in BSP applications. You can structure graphical user interfaces clearly and organize them in logical units, even with complex applications.

Using the MVC design pattern has the following advantages:

- Structuring BSP applications is simplified, since the view is cleanly separated from the controller and the model. This not only facilitates changing BSP applications, but also considerably improves their maintenance.
- You have the option of generating program-driven layout. The HTML/XML output is therefore created by program code instead of a page with scripting.

- Navigation using the **<bsp:goto>** element and call using the **<bsp:call>** element. The advantage of using **<bsp:goto>** navigation over redirect is that there is no additional network traffic. Furthermore, you remain in the same work process, which can have advantages for creating objects and memory space. The call using **<bsp:call>** element is more variable than adding them using **INCLUDE**directive, since it is triggered at runtime.

  With the call option using **<bsp:call>**, you can also distribute the user interface into **components**.

- Optimized performance due to fewer redirects.

- Intuitive and east-to-use interface for application development.

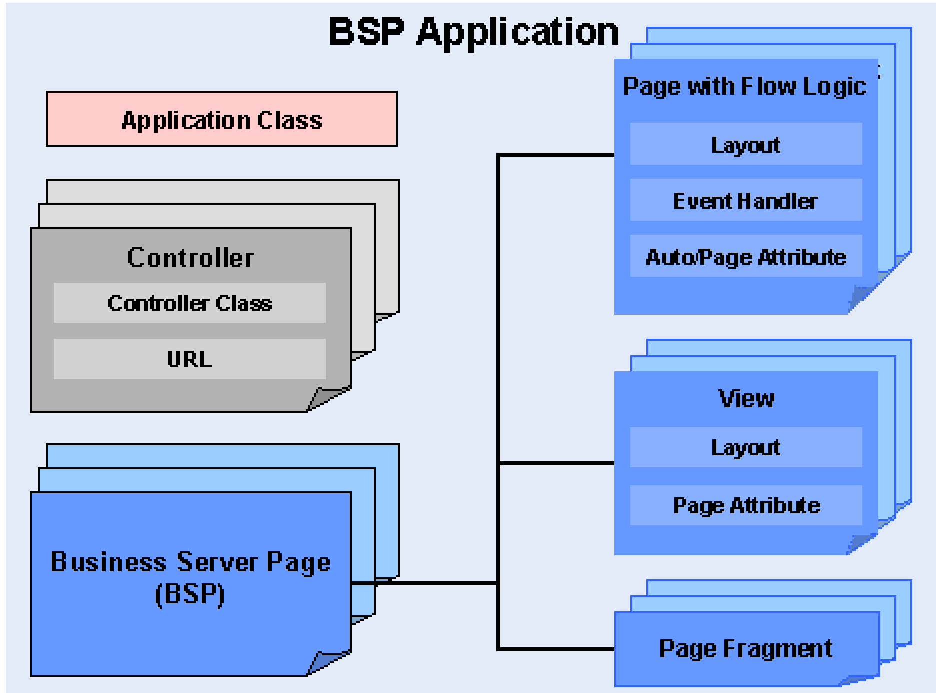
Previous BSP applications, that is, BSP applications without MVC, can still be executed without requiring any changes. MVC does, however, have various advantages with more complex applications. See **Using MVC for BSP**.

Integration

The MVC design pattern is integrated in the BSP programming model and the Web Application Builder of the ABAP development environment (Transaction **SE80**) from SAP Web Application Server 6.20.

Functions

A BSP application can consist of one or more controllers and Business Server Pages, as well as known elements such application classes, MIME objects and themes. A BSP can have different characteristics; it is either a page with flow logic (as before), or a view or a page fragment:



Within a BSP application, there can be several controllers, several views and several pages with flow logic.

Controllers

A controller is the instance of a central controller class. In the BSP-MVC environment, each controller is directly or indirectly derived from the same base class **CL_BSP_CONTROLLER2**, where the central method is **`DO_REQUEST`**.

There is a URL for every controller that can be addressed externally, such as using a browser. A controller can therefore be used as the initial point of entry to a BSP application. The mapping of the URL to the controller class is determined in the BSP application.

A controller is the controlling instance in the MVC design pattern, where it also acts as the controlling mechanism. It carries out the following tasks:

- It provides the data
- It is responsible for selecting the correct layout
- It triggers data initialization
- It executes input processing
- It creates and calls a view instance

Layout selection

A controller will usually call a view instance for creating the HTML / XML output. The controller can call a view that is created using a factory method. The theme or the browser variant, for example, can be used here as the selection criteria. If a controller passes the control to a view, it can – and should – set attributes to the view. These attributes may just be data, or a reference to one (or, in extreme cases, several) model(s). A reference to the controller is automatically transferred.

A controller has access only to views in its own application.
A controller can, however, delegate processing to another controller, and this controller can be located in a different application.

A controller should not work with too many views, since all of these requests are processed centrally. On the other hand, the controller should jump to all views that have the same or very similar input processing.

Data provision

Although a controller does not have any pre-defined attributes, they can be set and read using generic methods. However, a controller should provide a method **`init_attributes`**, which is responsible for filling the attributes. There is a service method that facilitates filling the attributes.

Event handling

The controller also takes care of event handling. It takes on all of the tasks that were executed in the previous BSP programming model by the **event handlers**: It carries out initialization and request processing, manages data transfer and is responsible for managing views and controlling a view's lifetime.

Redirects from the controller or page to the controller or page can be easily implemented. See also **Navigation**

If it cannot be decided until input processing which page should follow, we recommend that you let the controller branch to different views (for example, if it is checked internally whether the user has registered as a customer, and the corresponding data is then queried).

A controller can also be used to delegate control over screens to the sub-controller. A controller can delegate the control for a whole screen or a screen section to one or more different sub-controllers. This can result in a complex tree structure of controllers and **components** can be formed (that consist of both cascading controllers as well as their corresponding views).

You can find information about the life cycle of controllers in **Lifetime**.

View

Views are only responsible for the layout; they visualize the application data. Views are very much like pages, although they do not have event handlers nor auto-page attributes, nor their own URL. Unlike auto-page attributes, normal page attributes can be used, which are then filled by the controller. Controllers should control calling views and communicate with a model.

If the type of controller class is known for a view (see the *Properties* tab for the view), the view can also access the attributes of the controller class.

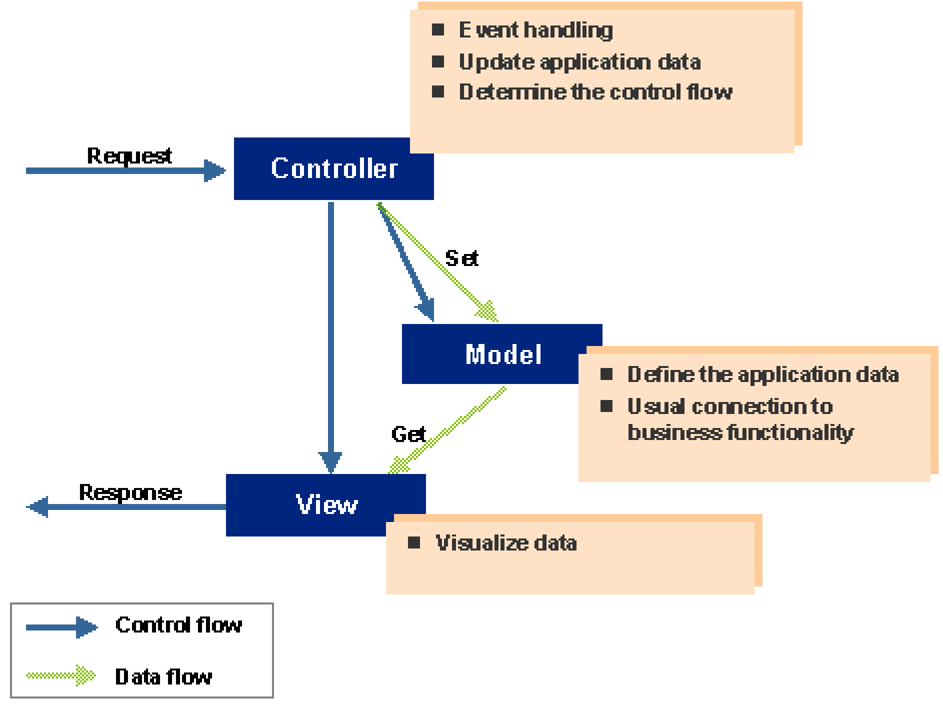
You can find information about the life cycle of views in **Lifetime**.

Models

The model is used to obtain all necessary application data from the database. It represents the internal data structures and corresponds to the **application class** used in the remaining BSP programming model. The model is responsible for carrying out the central actions of reading, modifying, blocking and saving data.

When used with controllers, this controller can create a reference to a class that is used as a model. Class **CL_BSP_MODEL** is available for this (see also **Data Connection**).

MVC in BSP Applications



For more information, refer to:

Using MVC for BSP

Class CL_BSP_CONTROLLER2

Navigation

Lifetime

BSP Component Call Options

Components

Activities

Creating a Controller

Creating a View

Calling a Controller

Calling a View

Creating Error Pages

A simple **Tutorial** is available for your first steps with the MVC design pattern.

Example

You can find an example of MVC in the system in BSP application **BSP_MODEL**.

Furthermore, the following **Architecture Examples** are outlined:

- **BSP Application with Controllers and Views**
- **BSP Application with Several Views per Controller**
- **Combination of the Previous Examples**
- **Calling Controllers of Other Applications**
- **Calling Several Controllers from a View**

MVC Design Pattern

The **Model View Controller** (**MVC**) design pattern contains a clear distinction between processing control, data model and displaying the data in the interface. These three areas are formally distinguished from each other by three objects: **model**, **view** and **controller**. As a result, you can easily split Web applications into logical units.

The **model** is used as an application object of the application data administration. It responds to information requests about its status, which usually come from the view, as well as to statements for status changes, which are usually sent by the controller. In this way, only the model is used to process data internally, without making reference to the application and its user interface.
There can be different views for a model, which can be implemented using different view pages.

The **view** handles the graphical and textual output at the interface and therefore represents the input and output data in each interface element, such as pushbuttons, menus, dialog boxes and so on. The view takes of visualization. To visualize the status, the view queries the model, or the model informs the view about possible status changes.

The **controller** interprets and monitors the data that is input by the user using the mouse and the keyboard, causing the model or the view later to change if necessary. Input data is forwarded and changes to the model data are initiated. The controller uses the model methods to change the internal status and then informs the view about this. This is how the controller determines reactions to the user input and controls processing.

The view and the controller together form the user interface.

Since the model does not recognize either views or the controller, internal data processing is detached from the user interface. As a result, changes to the user interface have no effect on internal data processing and the data structure. You also have the option, however, of displaying the data in different formats; you can display election results as a table, a bar chart or as a pie chart.

You can find additional information about the MVC design pattern on the Internet and in current specialist literature.

Using MVC for BSP

Uses

All BSP applications that you created with SAP Web AS 6.10 can also be executed without MVC. In general, you do not need to change anything.

The previous BSP implementation model gives you the option of controlling event handling and navigation using redirects.

The MVC design pattern provides you with various advantages, so that you can consider converting to MVC in the following cases:

- If your pages are dynamically composed of several parts (components)

  A controller can assemble a page from several views. As a result, the layout is **componentized**.

- If input processing is so complex that it should be subdivided into different methods

  A controller offers great flexibility, especially during input processing, since you can create and call new methods.

  If the system cannot decide which page comes next until input processing, we recommend that you let the controller branch to different views.

- If redirects using navigation can lead to performance problems (such as slow diversion)
- If visualization logic is fairly important, since you can use MVC to separate the logic from the layout
- If the layout from a different person is being processed as the visualization logic
- If parts of the layout should be created by the program, such as by a generating program or an XSLT processor

Combination of MVC with BSP

You can combine the technology of the previous implementation model for BSPs with the new MVC design pattern.

- In an application, there may be pages with flow logic as well as controllers and views

- The views can only be called by the controllers.

- Redirects from pages to controllers and back can take place with the help of redirect using the navigation methods.

- In the page layouts you can use the `<bsp:call>` element or the `<bsp:goto>` element to call a controller. You cannot use these elements to call pages.

Process

- Use the top controller as a point of entry to your BSP application and its process flow. First create a controller (see **Creating Controllers**).
- Then a call a view from this top controller. Next create a corresponding view (see **Creating Views**).

- Now **test** your controller.
- Then call the controller or the sub-controller (see **Calling Controllers**), and then the view (see **Calling Views**).
- If necessary, you can also create **error pages**.

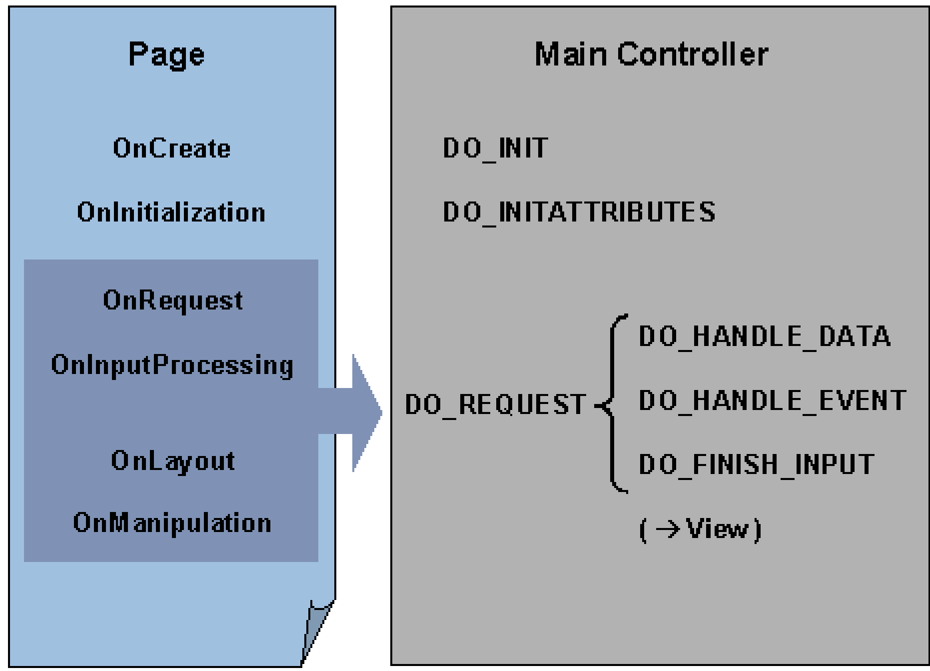
From Pages to Controllers

With a "normal" page, the presentation is determined by the layout, whilst in the MVC design pattern, views specify the presentation. With normal BSPs, predefined event handlers are available to process events. With MVC on the other hand, events are handled by controllers.

Normal pages are different from controllers especially with regard to event handling and programming. The events of the pages can be matched with the controller methods:

- Page events and main controller methods
- Page events and sub-controller methods

Page events and main controller methods

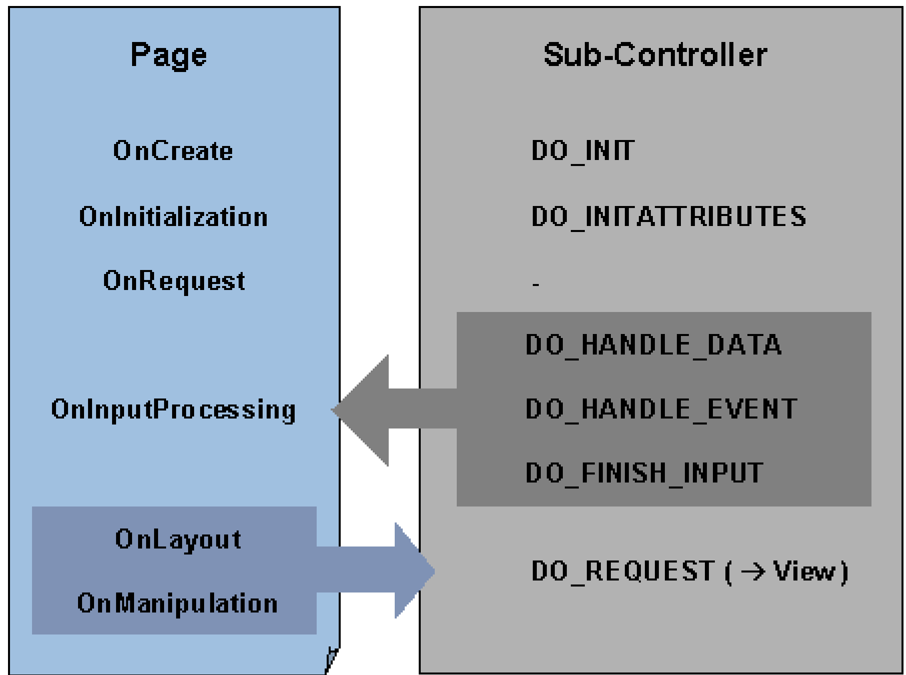


A main controller handles both input and output processing, where it uses the central method **DO_REQUEST** to call the methods specializing in input processing: **DO_HANDLE_DATA**, **DO_HANDLE_EVENT** and **DO_FINISH_INPUT**. In method **DO_REQUEST**, input processing must be triggered using **DISPATCH_INPUT**. This corresponds to the processing steps in the purely page-based BSP programming model that are executed using events **OnRequest**, **OnInputProcessing**, **OnManipulation** and **Layout**.

Page events and sub-controller methods

From method `DO_REQUEST`, the three following methods required for input processing are called:

- `DO_HANDLE_DATA`
- `DO_HANDLE_EVENT`
- `DO_FINISH_INPUT`



Call Options of BSP Components

In general, views can only be called from controllers. The only exceptions are error pages.

Controllers can be called from controllers, or from the layout methods of pages and views.

The calls can be considered as "forwarding a request" or as "adding a page fragment".

Calls that are made from a controller are identical. Only the environment is different, particularly depending on whether or not data was already written in the HTTP response, and whether additional data is subsequently added in the HTTP response.

The calls are different from views or pages. You can use the following elements of BSP extension `bsp` to branch from a view or a page to a controller:

- `<bsp:goto>`

  Forward

- `<bsp:call>`

Insert

These two elements are based on the same technology as when a controller is called by a controller. As inner elements, both can only have elements of type **`<bsp:parameter>`**. You hereby determine the parameters that are passed to the controller.

Navigation

There are two options for navigating to a different URL:

- **`navigation->goto_page`** for a page or a controller
- **`<bsp:goto>`** element for a controller.

navigation->goto_page

With **goto_page**, there is a **redirect** to the specified page (or to the controller), that is, the browser is informed that it may request a different page. There is then a new browser request to the destination page.

This has the following effect:

- The browser recognizes the URL for the destination page, since it requested it itself. In the page does not run in a frame, its URL is displayed in the address line of the browser.
- An additional browser request is required, which leads to increased network load. Even if the amount of data is extremely small, this may slow down performance with very slow networks (such as via satellite).
- In a stateless case, all temporary data from previous processing is lost.

<bsp:goto> Element

With the **`<bsp:goto>`** element, the new controller or view is **called** to provide the content for the current request.

This means that:

- The browser does not recognize the URL for the destination page, but tries to communicate with the existing page.
- No additional browser request is required.
- If no **`target`** has been entered in the **`form`** of the target page, the request that results from sending the **`form`** (or also from a **`refresh`**) is sent to the requesting page. As a result, the target page only has the task of creating the HTML (view) and does not usually have to worry about input. The calling page is responsible for this and thereby takes over the controller functionality.
- The work process does not change, that is, the context remains the same even in a stateless application and you can therefore access the data and objects that have already been created.

- The controller must be able to use the input to decide on its current "status" if it should display several views after each other. It should not store this status on the server, otherwise the "Back" processing would not function correctly. With different URLs, this is easier using redirects.

- When you use different views, the URL does not change. As a result, you cannot use bookmarks on these pages.

What does page_name point to?

runtime->page_name always points to the externally addressed page or the controller. You get the name and URL using **page->get_page_name()** or **page->get_page_url()**.

Lifetime of the View that is Called

The lifetime of the view that is called is limited to the call **only**. For more information, see Lifetime and Note 545847.

Lifetime

Controllers

You determine the lifetime of components or their controllers in the usual way using the *Properties* tab page of the controller. You can specify the lifetime as one of the following three options in *Status*:

- To page change
- For the duration of the request
- For the duration of the session

The setting is usually *To page change*.

By default, the lifetime of controller instances is limited to the one call. If the controller instance is passed with **id**, then its lifetime is the same as that specified in the controller's properties (*Properties* tab). The **id** can be specified as follows:

- From a page or a view: **<bsp:call/goto comp_id = "id">**
- From a controller or a page event: **create_controller(…controller_id='id')**

The **controller_id** or the **comp_id** of the **<bsp:call>** element must **not** contain an underscore ('_ ').
The underscore is reserved as a separator between controllers.

The lifetime of the top-level controller ranges from the first **CREATE_CONTROLLER** for a sub-controller to **DELETE_CONTROLLER** for the sub-controller.

If sub-controllers should occasionally be hidden, so that they are not involved in event handling for a while, then use method **CONTROLLER_SET_ACTIVE**. A controller that is set to inactive in this way will not be called for input processing.

The controller is only hidden; the controller instance is **not** deleted.

Views

This section concerning the lifetime of views concerns the use of MVC in SAP Web AS 6.20 up to and including Support Package 9.

Unlike controllers or pages, views have only a very short lifetime. Their life cycle looks as follows:

1. They are created.
2. They are supplied with parameters.
3. They are called.
4. They are now no longer required and therefore expire, since views **cannot** be reused.

Views therefore only exist for the duration of the call, that is, they are destroyed after they have been called.

As a developer of BSP applications with MVC, you must explicitly recreate the view, since you cannot reuse it in a controller. The following provides an example of correct and incorrect coding:

Use:	**Do not** use:

`DATA: view TYPE REF TO if_bsp_page.`
`view = create_view( view_name = 'main.htm' ).`
	`DATA: view TYPE REF TO if_bsp_page.`
	`IF view IS NOT BOUND. " or IS NOT INITIAL.`
	`    view = create_view( view_name = 'main.htm' ).`
	`ENDIF.`

In any case, the view must be explicitly recreated before it is used.

```
DATA: view TYPE REF TO if_bsp_page.
view = create_view( view_name = 'main.htm' ).
" ... set attributes ....
call_view( page ).
CLEAR view.
```

Data Binding

To make programming easier for you with the MVC design, the framework for the model of an application provides you with basic class CL_BSP_MODEL, which you can use in your own model class as a class that passes on its properties. The model class represents the data context of your application and, as a result, receives a copy of the data (or references to the data) that are relevant for the view from the database model.

The model class provides:

The data that are used for the views, with the corresponding type and data Dictionary information.

Input conversions

Information about input errors that occurred for which data

A controller can instantiate a model class or even several model classes (see also **Calling the Model Class Using the Controller**). The controller has a list of all model instances, analogous to the list of sub-controllers.

The controller assigns unique IDs to each model instance.

If you are using the MVC Design Pattern, you do not need to use the **Application Class**. Instead of the usual application class with purely page-based BSP applications, you should use controllers and model classes in the MVC environment.

Data binding is particularly important with **`HTMLB`**extension elements **`inputField`** and **`label`** (see also the documentation for these elements in the system). It is also implemented for **`HTMLB`** elements **`dropdownListBox`**, **`radioButtonGroup`**, **`checkbox`**, **`textEdit`** and **`tableView`**.

A model class can either be designed quite simply or it can be used with more complex application cases.

Simple Model Class

Data binding is important for transmitting values for output data. Add the data that is required by the view to your model class as attributes. These attributes are all from the visibility range *public* and can be as follows:

Simple variables

1. Structures
2. Tables

In the most simple case, the model class has these attributes only and so can easily be used for data binding as part of a BSP application.

This type of simple model class provides the following functionality:

The controller can create a model instance and initialize the attributes, since they are public attributes.

The controller transmits a reference to the model instance to the view.

The data binding to the model is specified in the view for each view element using a path expression (//...).

Example:

A BSP application contains an input field that is implemented using **`HTMLB`**, in which users can write data.

`model` is defined as a page attribute. For the input field you can then write:

`<htmlb inputField … value=”//model/<Attribut>”`

This ensures that the content of **`value`** is bound to the corresponding attribute of the model class.

The process flow is now as follows:

1. The content of the attribute is assigned to the input field value using the above statement.
2. This generates the ID from the model.
3. Additional attributes are also generated, for example, one that determines whether fixed values exist.
4. User input is transferred to the model class at the next request.
5. Data conversions including connection to the Dictionary (conversion exists, for example) are automatically executed by the base model class.

In the default case, if a conversion exit for a field exists in the ABAP Dictionary, this conversion exit is called. All data contained in the ABAP Dictionary structure for the field are available. If, however, separate setter/getter methods (see the following section) are written, the conversion exit can be switched off.

If necessary you can also add your own methods to your model class for further processing attributes.

Complex Model Class

It is possible that simple model classes are not sufficient for your requirements. This may be the case, for example, if you are working with generic data or if you need special methods for setting (**SET**) and getting (**GET**) attributes. You can therefore use these methods to determine your own implementations that are important for your specific application.

In these types of applications, the base class contains copy templates for the setter and getter methods: **_SET_<attribute>**and **_GET_<attribute>**. All of these templates begin with _. The naming conventions for the actual methods are as follows:

Naming convention for setter methods:

SET_<attribute> for a field
SET_S_<attribute> for a structure
SET_T_<attribute> for a table

Naming convention for getter methods:

GET_<attribute> for a field
GET_S_<attribute> for a structure
GET_T_<attribute> for a table

An example of implementing a getter method for structure fields/structure attributes:

```
method GET_S_FLIGHT .
field-symbols: <l_comp> type any.
  assign component of structure flight to <l_comp>.
  value = <l_comp>.
  if component eq 'CARRID'.
    translate value to lower case.
  endif.
  if component eq 'CONNID'.
    shift value left deleting leading '0'.
```

```
  endif.
endmethod.
```

The ABAP keyword **assign component** assigns the structure component **component** for the structured field **flight** (with reference type **sflight**) to the field symbol **<l_comp>**. The value of **<l_comp>** is output as follows: If the structure component **component** points to an airline (**CARRID**), the name of the airline is translated in lowercase. If the structure component **component** points to a single flight connection (**CONNID**), then some of the introductory zeros may be deleted.

As soon as a setter or a getter method is set, it is used automatically.

Data binding is automatically available because the name is the same. In method **DO_HANDLE_DATA** (see also **Process Flow**) of class **CL_BSP_CONTROLLER2** all controllers automatically fill the form fields with data.

The path specifications for the model data have the following syntax:

Simple field attribute

```
value="//<field name>"
```

Structure attribute

```
value="//<structure name>.<field name>"
```

Table attribute

```
value="//<table name>[<line index].<field name>"
```

Calling the Model Class by the Controller

Uses

A model class is called or managed by a controller, that is, a controller can hold one or several model classes (or instances). The controller class provides methods for creating, getting, setting and deleting this type of model class. There are also methods for passing incoming data on to the correct model instance, which is identified by the **model_id**.

Components

Use

Complex BSP applications that are based on the **MVC Design Pattern** have many extensive components. Each individual part, consisting of a complex BSP application, contains precise application logic and well thought out presentation logic. It makes sense to create the individual BSP components as reusable modules. These reusable modules are:

- Controllers
- One or more views

- A Model

Together they form a **component**.

Components are only available for **stateful** BSP applications.

Integration

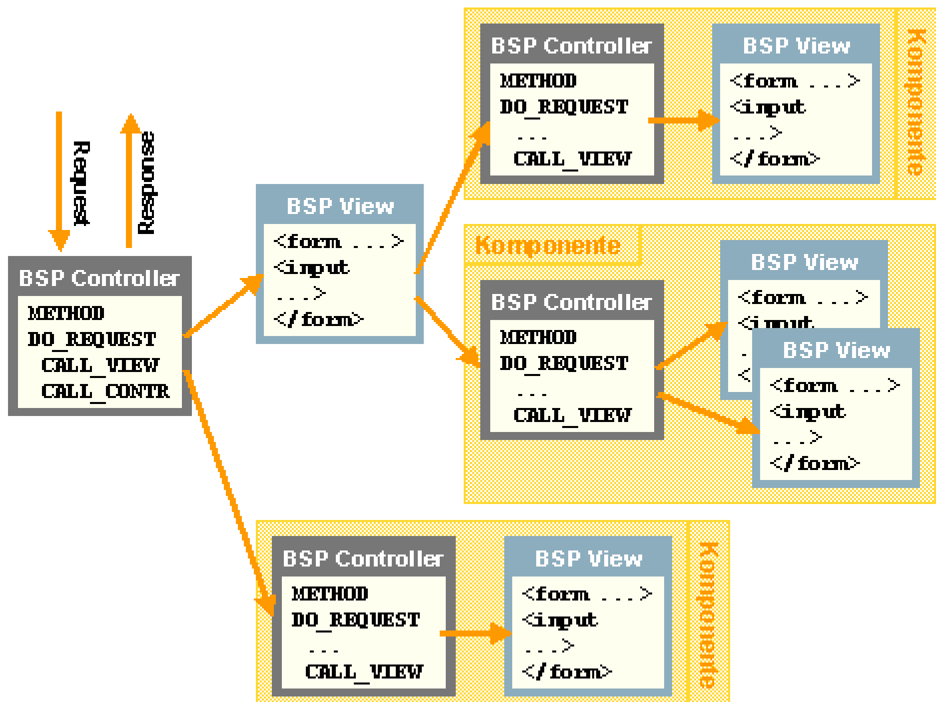
The use of components is integrated in the MVC design pattern.

Prerequisites

You are in SAP Web AS 6.20.

Functions

A component consists of a controller, whose class is derived from **CL_BSP_CONTROLLER2**, as well as one or more views, which can result in regular nesting. This is outlines in the following graphic:



Central features of components are:

1. • With components, there are complex call sequences during an HTTP request.
2. • The individual parts, of which a page in the browser consists, are dynamically assembled during runtime.

3. • One component can call a different component. It should therefore be placed in a view. This is done using the <**bsp:call**> element.

1. • Initialization can be called by the controller using method **create_controller**. This method is available for all controller classes. It creates a controller or finds an existing one.

- The parent controller contains a list of the individual sub-controllers and forwards all input to the relevant controller. This is done by prefixing all IDs with the path of the controller IDs.
- The controller has a hierarchical tree. Every controller controls its view or views, its model as well as the list of sub-controllers.
- Basis class **CL_BSP_CONTROLLER2** controls the sub-controllers. The controller developer is responsible for controlling the view and the model.

If you want to use data binding functionality, you can add a model class to your component. For more information see **Data Binding**.

Activities

1. ...

1. 1. **Creating the top-level controller**
2. 2. **Creating a Component**
3. 3. **Calling the Component**
4. 4. **Determing the Input Processing**

Process Flow

Uses

The methods of class **Class CL_BSP_CONTROLLER2** are used to create components as part of the Model View Controller design pattern.

The whole hierarchy level is processed with every request.
The hierarchy itself is defined at output.

Process

1. First call **DO_INIT**.
2. Then call **DO_INITATTRIBUTES**.
3. Then call **DO_REQUEST**.

With a main controller, **DO_REQUEST** takes care of both input and output processing.

a. Input processing

The browser request is sent directly to the top-level controller. This dispatches the input to the sub-controllers. Service function **DISPATCH_INPUT** is available for this.

DISPATCH_INPUT reads the form fields from the request and dispatches them to the sub-controller. Prefixes are added to the form fields.

The prefixes are written automatically for BSP elements, for example, by BSP extension **HTMLB**.
If, however, you have pure HTML or HTML tags, then you must add the name of the controller as a prefix to your input data. In this case, service function **GET_ID** is available for adding prefixes.

All data that do not belong to one of the sub-components must be processed using method **DISPATCH_INPUT** in the main controller. The following methods are called:

- **DO_HANDLE_DATA**
- **DO_HANDLE_EVENT**
- **DO_FINISH_INPUT**

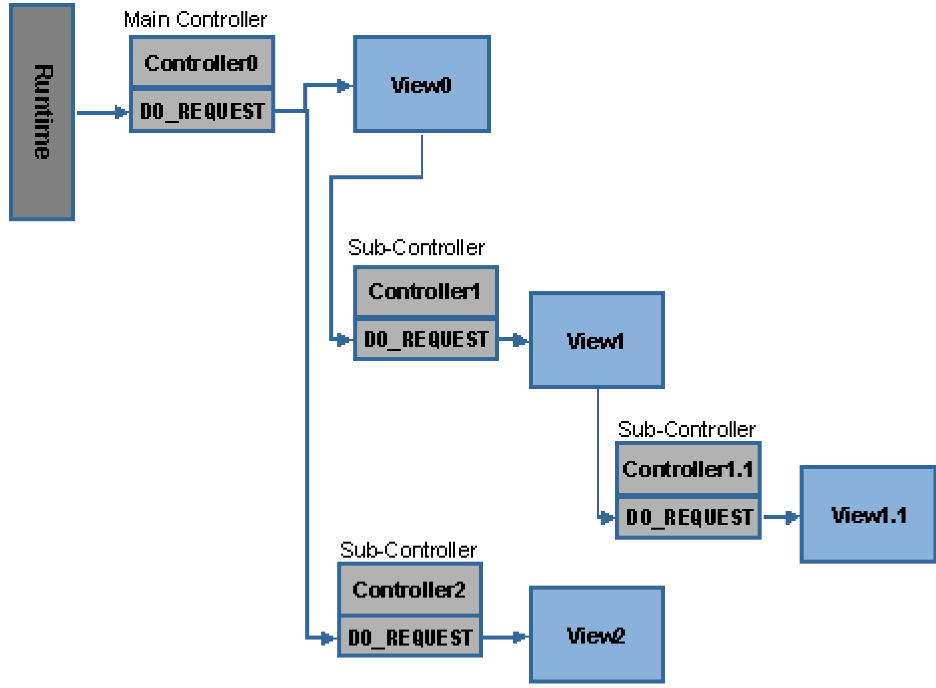
These three methods are called by the parent controller only with the form fields for the current controller.

a. Output processing

Determining output processes contains the output for the next page. A view is created and displayed. Depending on the status of the top-level controller, you can also set a sub-controller to inactive or create new controllers.

The process flow of the output is displayed in the following graphic:

Page Output



At output, **DO_REQUEST** carries out the following tasks:

i. **DO_REQUEST** determines whether data must be fetched from the model or from the global attributes.
ii. **DO_REQUEST** fetches the table with the object keys from the top-level controller.
iii. **DO_REQUEST** requests a view.
iv. **DO_REQUEST** sets the correct attributes for the view.
v. **DO_REQUEST** calls the view.

Handling events

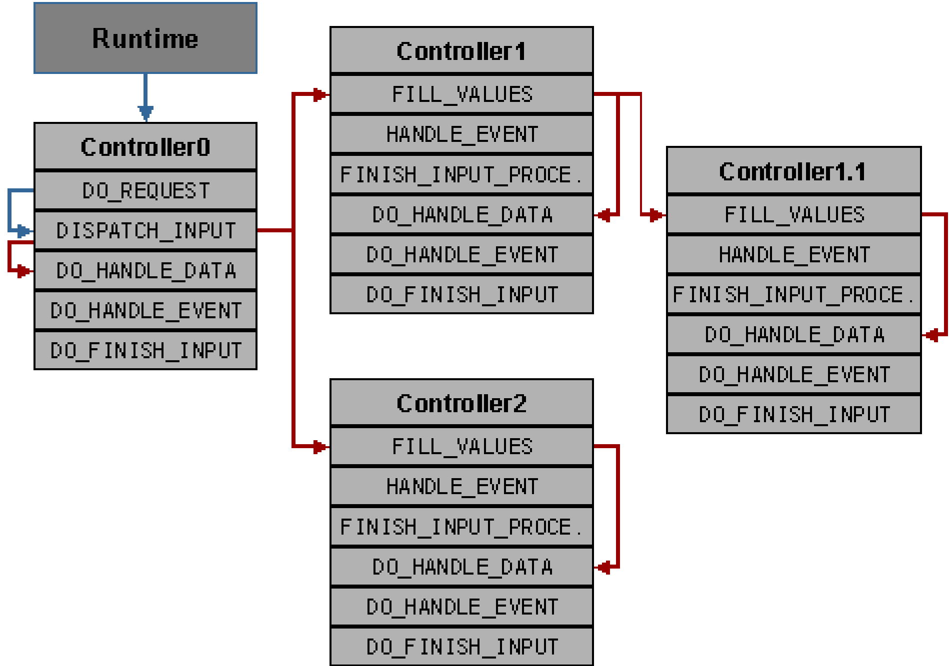
If a component contains events, **DISPATCH_INPUT** calls the HTMLB manager. The HTMBL manager collects the relevant information, including the ID, that is, the ID of the object that triggered the event.

DISPATCH_INPUT then calls method **DO_HANDLE_DATA**. **DO_HANDLE_DATA** is called by all controllers (that is, for all active components), that is, by the top-level controller as well as by all sub-controllers. The model class is filled with **DO_HANDLE_DATA** (see also **Data Binding**): The system transfers form fields and messages for the global messages object (see below).

If your model class is based on **CL_BSP_MODEL** and you have defined your setter and getter methods accordingly, the form fields are filled automatically.

The process flow with **DO_HANDLE_DATA** is displayed in the following graphic:

Page Input (DO_HANDLE_DATA)



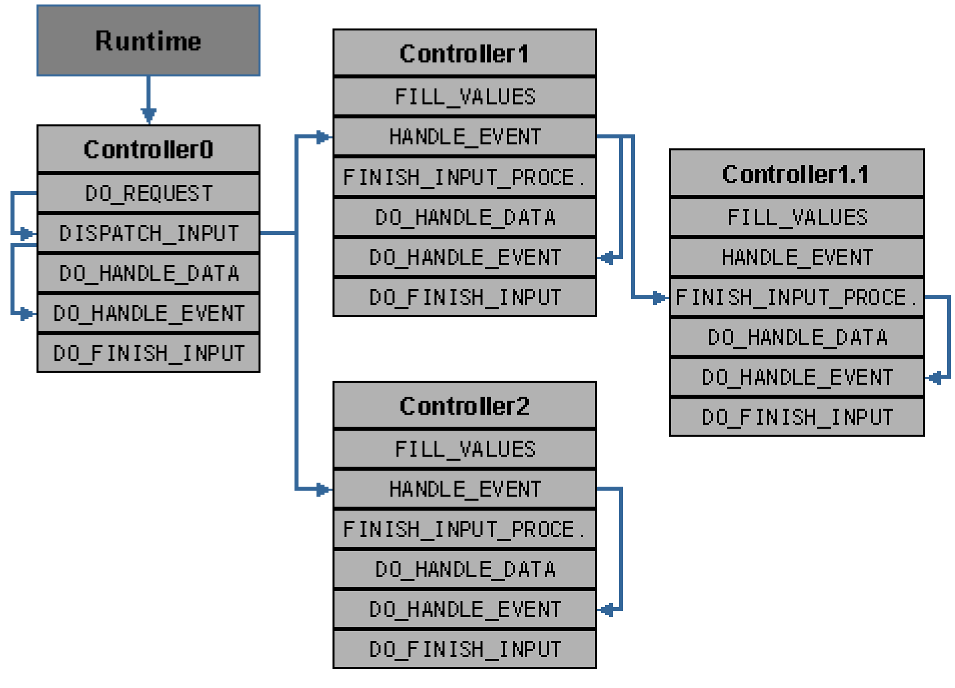
Once **DO_HANDLE_DATA** has filled all data, method **DO_HANDLE_EVENT** is called for the controller that is responsible for the input event. This also states the event ID and the event is dispatched to the controller. **DO_HANDLE_EVENT** also outputs parameter **GLOBAL_EVENT** (a string). If the event is an **HTMLB** event, object **HTMLB_EVENT** is filled accordingly.

Events are only dispatched to the relevant controller if the element ID was assigned to the **HTMLB** element (attribute **id**).

DO_HANDLE_EVENT also has access to the global messages object and can carry out additional steps if necessary. For example in the case of an error, this method can have data displayed again.

The process flow with **DO_HANDLE_DATA** is displayed in the following graphic:

Page Input (DO_HANDLE_EVENT)

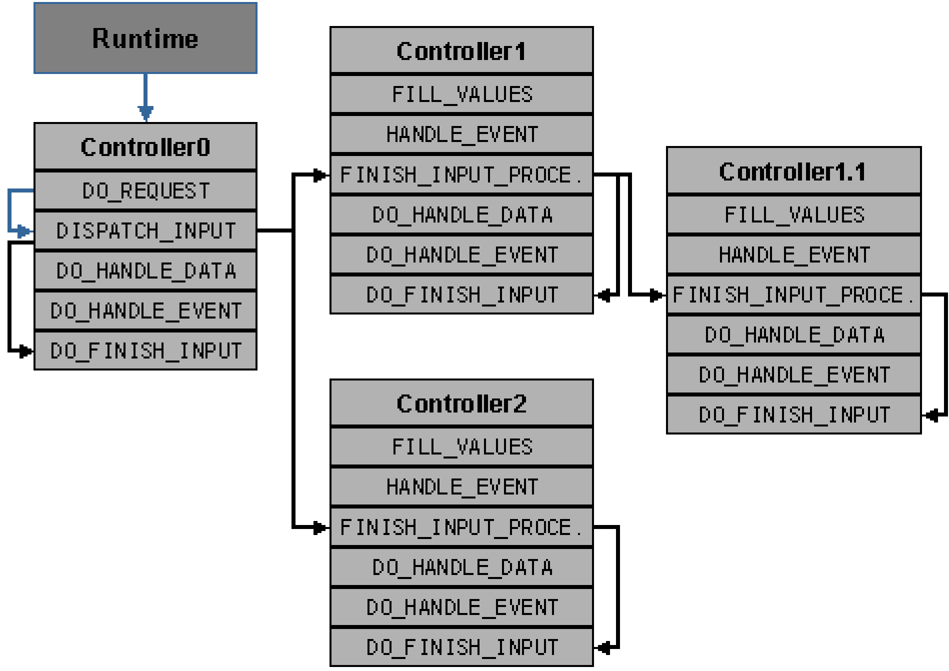


Note that only a sub-controller is called here.

Method **DO_FINISH_INPUT** is always called (for every controller, that is, for all active components). You can use it to react to events in a component that occur in a different component. To do this, use parameter **GLOBAL_EVENT**, which is set in method **DO_HANDLE_EVENT**. Using this global event, at the end of input processing each component should know exactly which events are present and how to react to them.

The process flow with **DO_FINISH_INPUT** is displayed in the following graphic:

Page Input (DO_FINISH_INPUT)



Global Messages

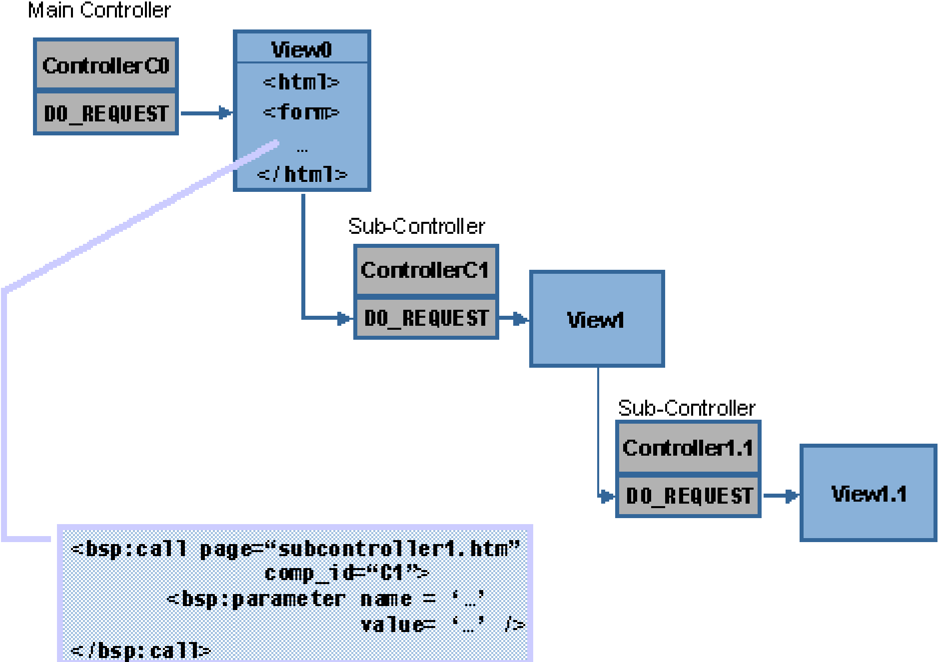
Parameter **GLOBAL_MESSAGES** is shared by all components. Use this parameter to handle incorrect user input, for example to display that an error occurred, or that the end date entered by the user is before the start date, and so on.

The main controller creates the global messages and forwards them to all sub-controllers. On the other hand, the messages object is local. If the local messages object is now filled in a controller, then you can forward this information to the global messages object and react to it using any component you like.

Controllers and Their IDs

Usually there is a main controller, a top-level view as well as different sub-controllers and additional views.

A main controller first calls **CREATE_VIEW**, then **SET_ATTRIBUTE** for the view, and then **CALL_VIEW**. The top level controller can also create sub-controllers. This is done using the **<bsp:call>** element, which has the attributes **PAGE** and **COMP_ID**. Furthermore, the embedded element **<bsp:parameter>** can also specify parameters for name and value.



The attribute output takes place either in the view or in the top-level controller.

COMPONENT_ID always identifies the controller. The **COMPONENT_ID** has a reference to the controllers concerned.
In method **CREATE_CONTROLLER** this reference is parameter **COMPONENT_ID**, and in the **<bsp:call>** element it is attribute **COMP_ID**:
When a controller is created, a reference is sent to the parent controller, which has a list of all the sub-controllers that belong to it.
Every sub-controller can query its parent controller for the **COMP_ID** of each additional sub-controller.

Creating Your Own Components

Uses

You create components to use them independently as well as together with other components for BSP applications. You can create a component for a search in an online shop, for example, and create an additional one for the detail display of the article that was found.

When you develop components, you can also form teams, so that one team is responsible for developing controllers, a different team is responsible for the views, and a third team is responsible for developing the models.

Process

1. Create the top-level controller
2. Create a component
3. Call the component
4. Determine the input processing

Creating the Top-Level Controller

Procedure

1. Create a BSP application and declare it as stateful.

   You can find the checkbox for stateful on the *Properties* tab page as the ID *Stateful*.

2. Save your BSP application.
3. Create a controller within this BSP application.

   a. Enter a unique class name for the controller.
   b. Set the lifetime in the *Status* field to *Session*.

4. Save your controller.
5. Double-click on the controller class name.
6. The following dialog box asks if you want to create the class. Answer it with *Yes*.

   You branch to the Class Builder.

7. Save your class.
8. On the *Properties* tab page, check that your class inherits properties from **CL_BSP_CONTROLLER2**. If this is not the case, for example if your class inherits properties from CL_BSP_CONTROLLER, then change the data for the class that passes on the properties.
9. Branch to the *Methods* tab page.

   a. In change mode, overwrite method **DO_REQUEST** using the icon (Redefine):

```
method DO_REQUEST .

   data: main_view type ref to if_bsp_page.

* if input is available, dispatch this input to subcomponent.
* this call is only necessary for top-level controllers.
* ( if this is not a top-level controller or no input is present,
*   this call returns without any action)
  dispatch_input( ).

* if any of the controllers has requested a navigation,
* do not try to display, but leave current processing
  if is_navigation_requested( ) is not initial.
    return.
  endif.
```

```
* output current view
  main_view = create_view( view_name = 'main.htm' ).
  call_view( main_view ).

endmethod.
```

   b. If necessary, overwrite method **DO_INIT**.
   c. In order to react to user input, overwrite methods **DO_HANDLE_DATA** and **DO_HANDLE_EVENT**.

10. Activate your class.
11. Create a view within your BSP application.

   a. In the following example, the view is called **main.htm**.
   b. Fill the view layout with HTML coding or HTMLB coding.
   c. Save the view.

12. Activate and test your finished BSP application.

Continue by **Creating Components**.

Creating Components

Procedure

1. Create a controller (including classes) in a BSP application.

   This controller may belong to an already existing BSP application or it can be located in its own BSP application.

   Note that the basic class of this controller and the top-level controller is class **CL_BSP_CONTROLLER2** (see also **Creating Top-Level Controllers and Views**).

2. If this controller should always be used as the component controller, then change method **DO_REQUEST** so that only views can be displayed. If not, **DO_REQUEST** would look exactly the same as the **DO_REQUEST** from the top-level controller.

   a. Overwrite methods **DO_HANDLE_DATA**, **DO_HANDLE_EVENT** and/or **DO_FINISH_INPUT**.
   These methods are called by the parent controller only with the form fields for the current controller. All components share parameter **GLOBAL_MESSAGES**. **GLOBAL_MESSAGES** is used to handle incorrect input.
   Parameter **GLOBAL_EVENT** is set by method **DO_HANDLE_EVENT** and is used in **DO_FINISH_INPUT**. The component developers should device how these values should be set.
   Methods **DO_HANDLE_DATA** and **DO_FINISH_INPUT** are called for all active components. **DO_HANDLE_EVENT** is only called by the controller that is responsible for the input event.

   b. For every attribute that should be passed to this controller, create a *public* attribute or a method.

1. Create one or several views.
2. Activate the views.

Continue by **Calling the Components**.

Determining Input Processing

Use

The browser sends its request to the top-level controller. This main controller dispatches the input to the appropriate sub-controller. This is why it is necessary to call method **DISPATCH_INPUT** in the top-level controller.

Procedure

Input processing consists of three steps:

1. Filling data

   For every controller, method **DO_HANDLE_DATA** is called with a list of form fields that should be handled by this method.
   If an error occurs during the data conversion, then this method can also pass one or more messages to the global error object (**global_messages**).

2. Handle event

   Method **DO_HANDLE_EVENT** is called for exactly one controller. The event is passed on and object **htmlb_event** is filled if it is an **HTMLB**-event. Method **DO_HANDLE_EVENT** has access to object **global_messages**, in order to determine the additional steps that are necessary, depending on the error. For example, in the case of an error, you can specify that you want to display the data again. You can also set a **global_event** using method **DO_HANDLE_EVENT**.

3. Finish input processing

   For every controller, method **DO_FINISH_INPUT** is called with a global event, which is set by the event handler method. The closing input processing is carried out here.

Class CL_BSP_CONTROLLER2

Overview

Class **CL_BSP_CONTROLLER2** is used to create controllers and **components**. Every controller class automatically inherits all methods and attributes from this central basic class.

If the basic class of your controller class displays **CL_BSP_CONTROLLER** instead of **CL_BSP_CONTROLLER2**, change the inheritance hierarchy accordingly.

Class **CL_BSP_CONTROLLER2** enables you to:

- Retain a list of sub-controllers
- Create unique IDs for the sub-controllers, where the sub-controller is assigned the controller ID prefix
- Use models
- Forward data to the correct controller as well as fill model classes (if they exist)

Methods

Below you can find an overview of all methods in a controller class. Processing Process provides details on the most important methods.

The individual methods can be separated into different categories:

Functions where overwriting is required

DO_REQUEST is the central method in a controller class.

You **must** overwrite this method.

In **DO_REQUEST** you specify the request processing, that is, this method is called for every request. This method does the "main work"; in particular it should branch to the correct view.

DO_REQUEST can be used in two different areas:

- If it is the top-level controller of a component, then this method handles both input and output processing.

- If it is a sub-controller of a component, then this method only handles output processing.

Functions where overwriting is recommended

You **should** overwrite these methods in order to determine input processing.

Method	**Description**
DO_HANDLE_DATA	Reacts to user input. Processes data input for this component.

DO_HANDLE_EVENT	Reacts to user input. Processes events if the component contains them. Exactly one view controller is called to handle the event, which contains an event such as a save button, for example.
DO_FINISH_INPUT	Ends the input processing.

Functions where overwriting is possible

You **can** overwrite these methods in order to determine input processing.

Method	**Description**
DO_INIT	This method is called once at the start and is used for initialization. This method behaves like a constructor method.
DO_INITATTRIBUTES	This method is called with every request and is used to initialize the attributes. The parameters are read from the request. In this method, you can also execute initializations that are required for each request. You can also use this method to set additional attributes. This method is not absolutely necessary, since you can use `DO_REQUEST` to solve everything that you can (theoretically) handle here.

Service functions

You can call these methods:

Method	**Description**
CREATE_VIEW	Creates or fetches a view instance Use either the name of the view, or the **object navigation**. A view must always belong to the same BSP application as its controller.
CALL_VIEW	Calls the request handler of the view instance.
CREATE_CONTROLLER	Creates or fetches a controller instance

CALL_CONTROLLER	Calls the request handler (method **DO-REQUEST**) of the controller instance.
GET_ATTRIBUTE	Returns the specified page attributes. Generic method for reading an attribute value.
GET_LIFETIME	Returns the lifetime of this page (only for the top-level controller)
GET_PAGE_URL	Returns the URL of the page or the current controller
SET_ATTRIBUTE	Sets the specified page attributes. Generic method for setting an attribute value.
SET_LIFETIME	Changes the lifetime of this page (only for the top-level controller)
TO_STRING	Creates a formatted string
WRITE	Writes a formatted string in the output
GET_OUT	Fetches the current output writer
SET_MIME_TYPE	Changes the MIME type of the page or the content type of the header field
INSTANTIATE_PARAMETER	Instantiates the parameter from the request using the request data
SET_CACHING	Changes the caching values There are two types of caching: • Browser cache • Server cache See also **Caching BSPs**. You can only use limited caching here. Note that the server cache is not user-specific. If you change the page, you should reset the cache that may be set.
DISPATCH_INPUT	Dispatches the input processing (only for the top-level controller). For each input, **DISPATCH_INPUT** calls the correct methods in the correct sequence. This method fetches data from the request.

	This method does not have any attributes.
GET_ID	Calculates the ID from the specified ID and the component ID
SET_MODEL	Creates and registers a model instance
CREATE_MODEL	Creates and registers a model instance
GET_CONTROLLER	Fetches a sub-controller
CONTROLLER_SET_ACTIVE	Sets a controller to active/inactive. This is relevant with input processing, since you can use it to hide a controller. See also **Lifetime**
DELETE_MODEL	Deletes a model instance
FILL_MODEL_DATA	Fills the model data
DELETE_CONTROLLER	Deletes a sub-controller
GET_MODEL	Fetches a model instance
IS_TOPLEVEL	Is this controller a top (main) controller (0: no, 1: yes)?
IS_NAVIGATION_REQUESTED	Has a controller requested a navigation (0: no, 1: yes)?

Framework functions

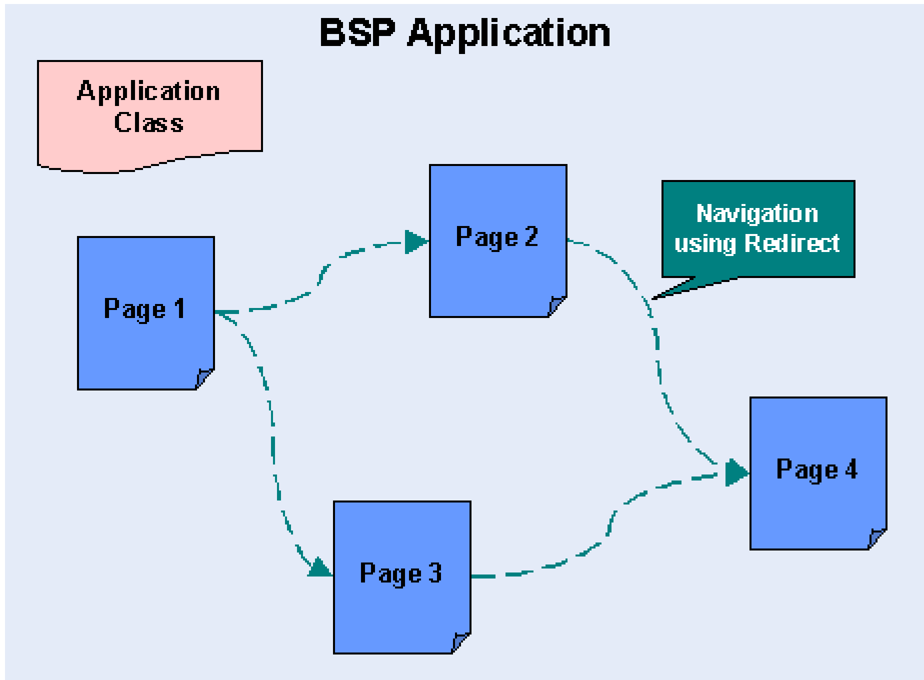
These methods are provided as part of the framework and are only included here for the sake of completeness. They are not usually relevant for application development.

Method	Description
IF_BSP_DISPATCHER~REGISTER	Registers a sub-components
IF_BSP_CONTROLLER~FINISH_INPUT_PROCESSING	Processes or dispatches: end of input processing.
IF_BSP_CONTROLLER~FILL_VALUES	Processes or dispatches: handling values
IF_BSP_CONTROLLER~HANDLE_EVENT	Processes or dispatches: Handle event
GET_FIELD_COMPONENT	Finds components for a field name
GET_FIELD_MODEL	Finds model for a field name

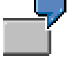
Methods `DO_DESTROY` and `SUBSCRIBE` are not relevant.

Previous BSP Application

With SAP Web AS 6.10, normal BSP applications usually consisted of an application class and several BSPs. Navigation between the pages was controlled using redirects.

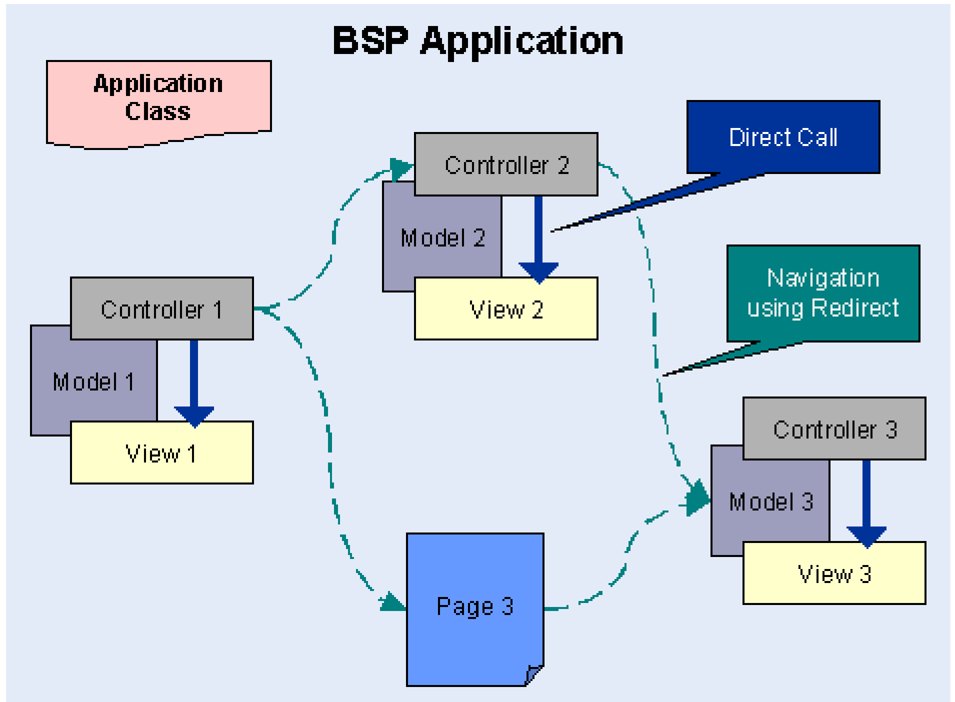


This is how it looks with SAP Web AS 6.20: **BSP Application with Controllers and Views**

 BSP Application with Controllers and Views

From SAP Web AS 6.20, you can combine controllers and views in a BSP application. You navigate between the controllers and any pages that exist using redirect. Each controller can have one (or several) model(s).

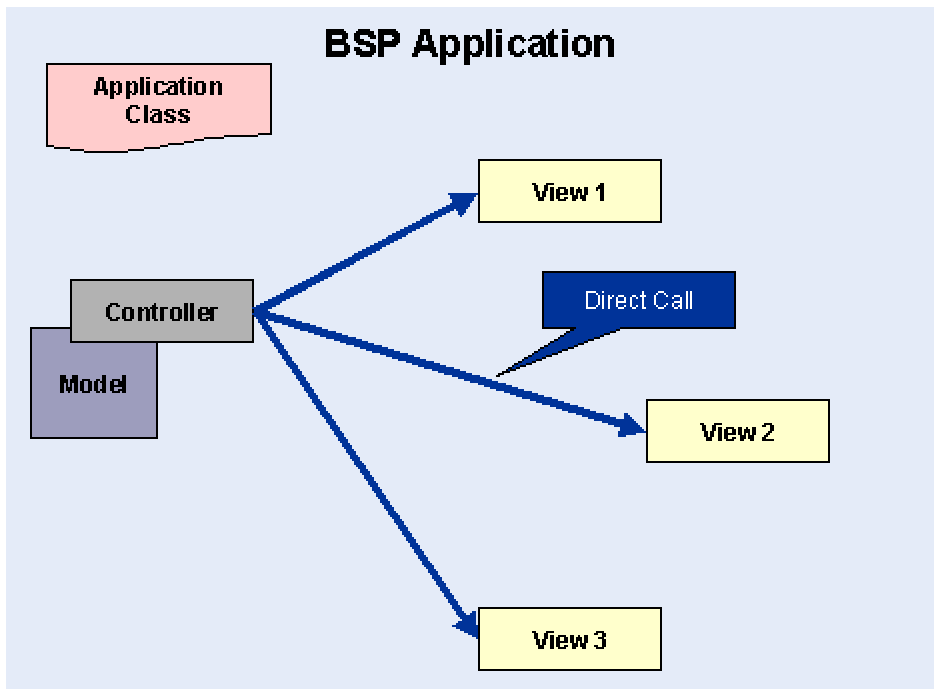
You can use redirect to navigate between the controllers. You call the views using a call.



With several views for a controller, the whole thing looks as follows: **BSP Application with Several Views per Controller**

BSP Application with Several Views per Controller

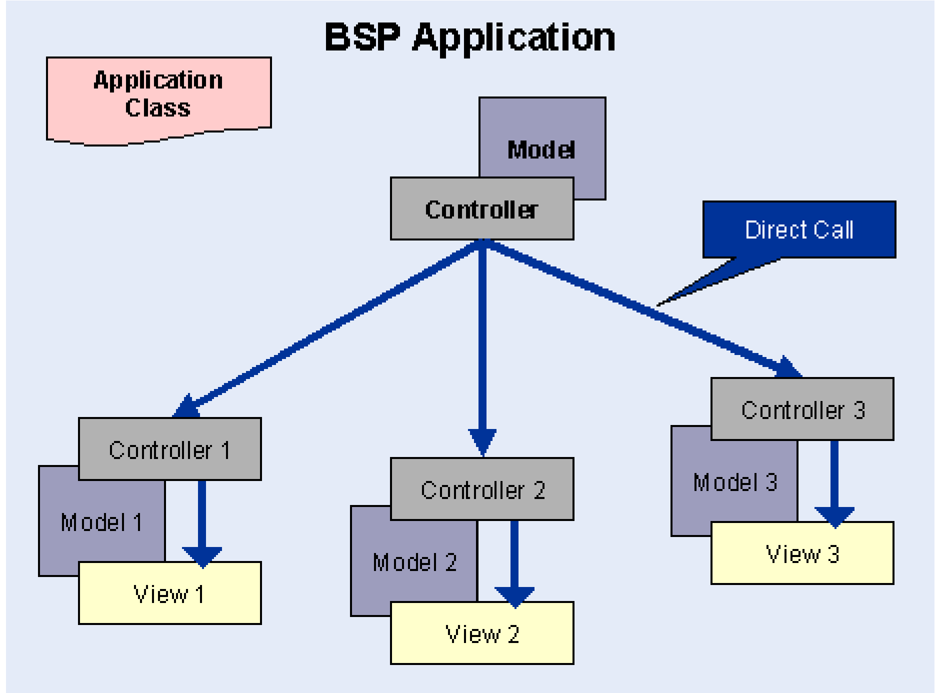
With a **BSP application with controllers and views**, an individual controller can also call several views – either sequentially after each other or alternately. With this example, you always access it using a controller.



What does a combination of these two examples look like (this one here and **BSP application with controllers and views**)? **Like this**.

Combination of the Previous Examples

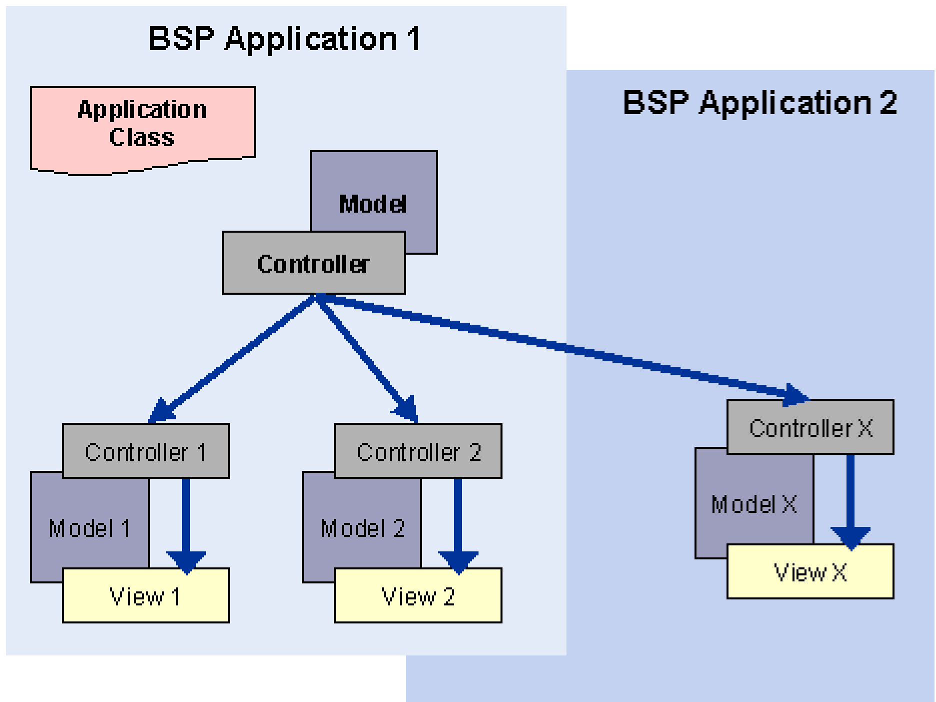
You can combine the examples of a **BSP application with several views per controller** with a **BSP application with controllers and views** so that each view is controlled by exactly one controller. Central distribution is carried out by a superior controller.



Does this also work with several BSP applications? **Of course**!

Calling Controllers of Other Applications

You can also call controllers of other BSP applications:

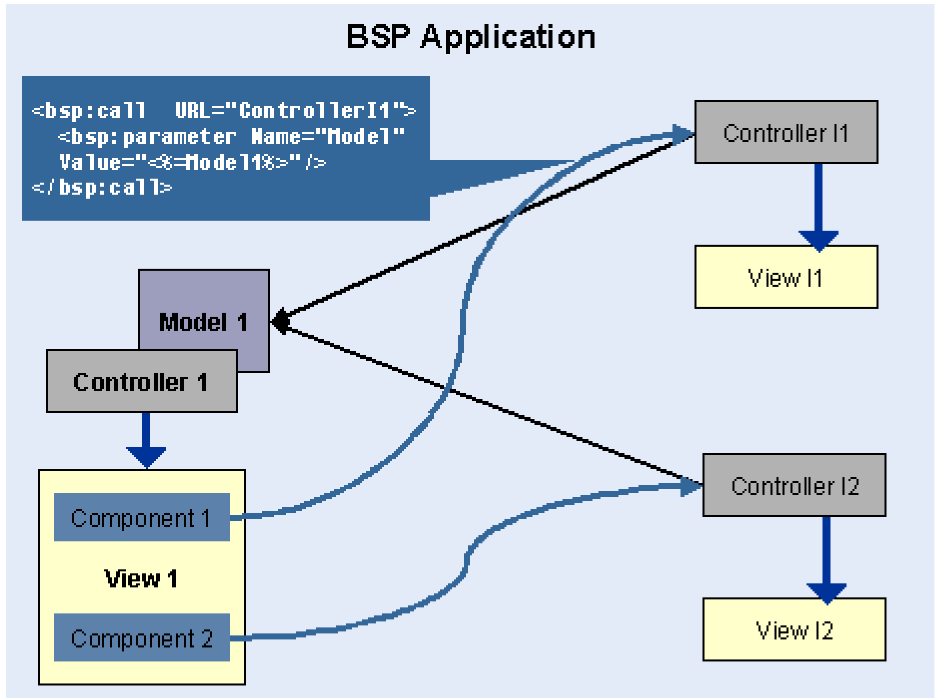


Calling Several Controllers from a View

With this example, several controllers are called from a view using a BSP element.

There can also be a page in place of the calling controller and view (in the graphic on the left-hand side). There cannot be a page, however, at the level of the called areas (in the graphic on the right-hand side).

With help from the views that are allocated, these controllers provide the contents for a sub-area of the main views. A reference to the model can also be specified with the call.



Model View Controller Tutorial

Uses

In this tutorial you can use a simple example to run through the first steps with the **Model View Controller** design pattern for BSP.

Prerequisites

- You are in an SAP Web AS 6.20 system
- You know how to use **MVC for BSPs**

Functions

Creating a Controller

Creating a View

Calling a Controller

Creating a Controller

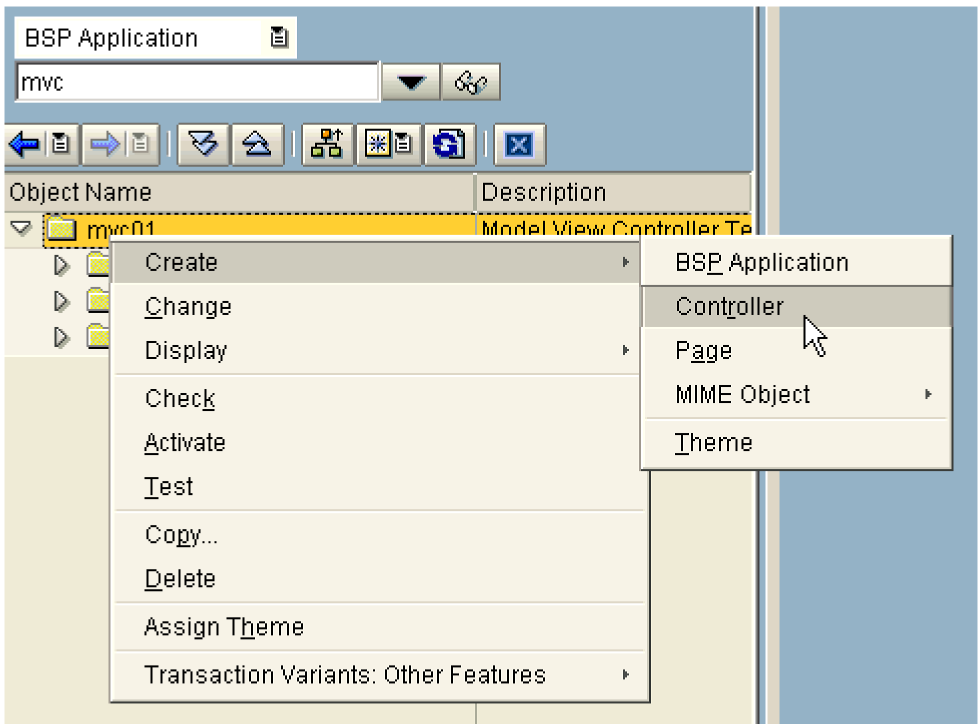
Prerequisites

You have created an empty BSP application for this tutorial.

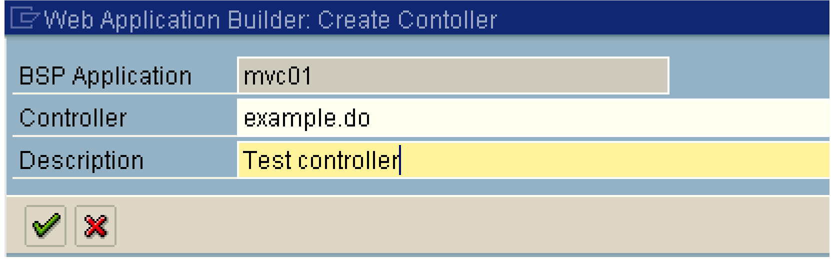
Procedure

1. Create a controller within your BSP application.

   To do this, choose *Create* → *Controller.*

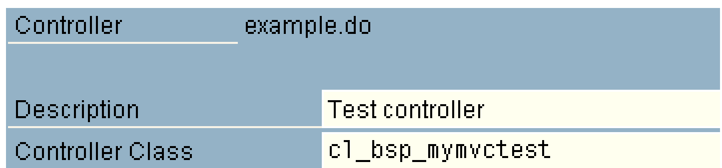


2. On the following dialog box, give the controller a name and add a short description.



3. Choose ✔.
4. On the following screen, assign a class name to the controller.

   The class does not have to exist yet.

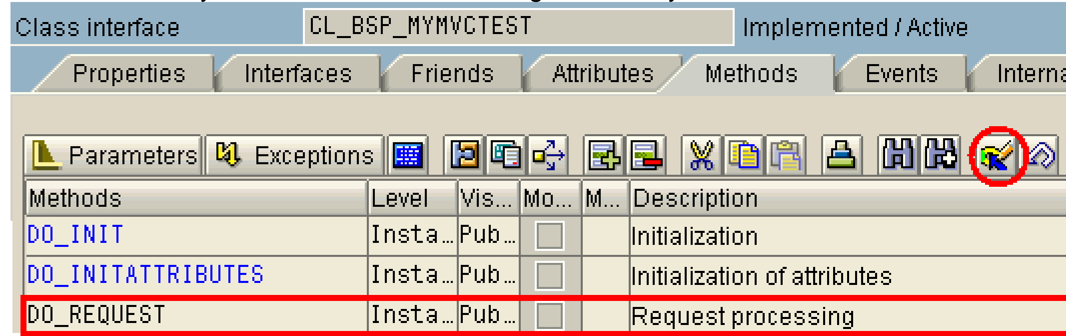


5. You navigate to the Class Builder by double-clicking on the controller class.

   If the class does not already exist, the system asks you if you want to create it. Choose *Yes* so that you create a class with the specified name that is derived from **CL_BSP_CONTROLLER2**.

   Each controller class must be derived directly or indirectly from **CL_BSP_CONTROLLER2**.

6. Choose the symbol to branch to the change mode in your class.

7. Select method **DO_REQUEST** and choose symbol to overwrite the methods.

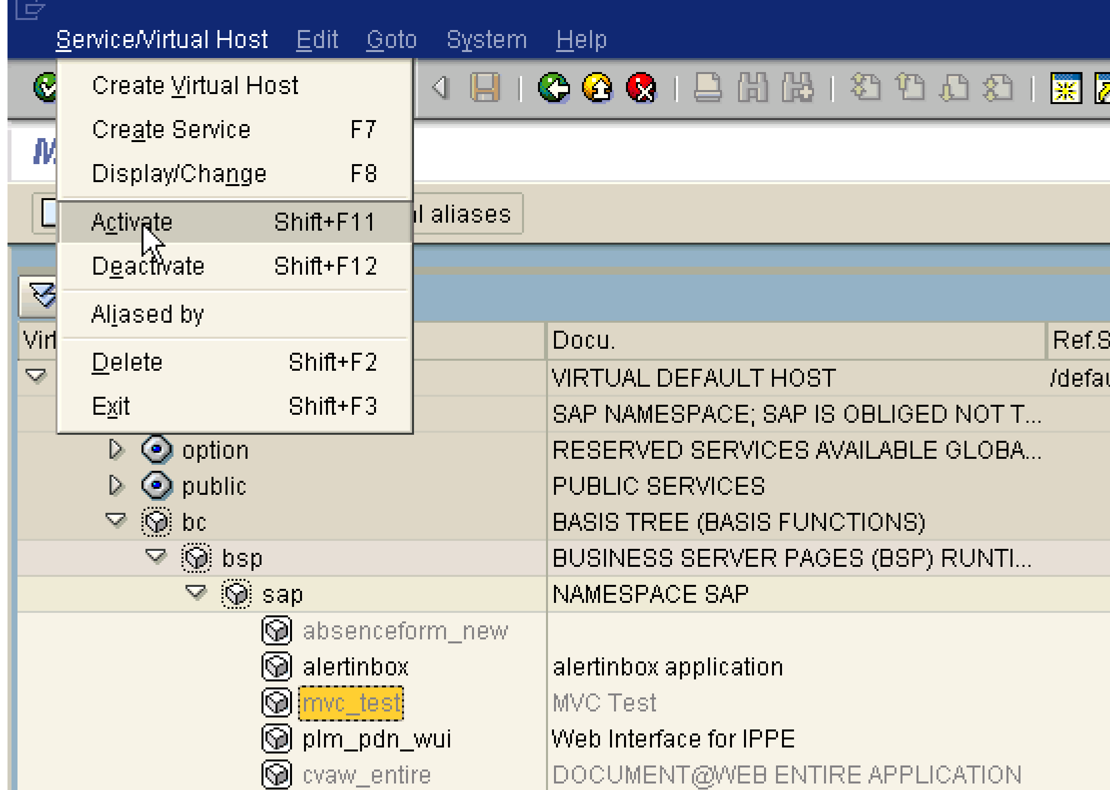
8. Generate the required output.

   In this example, it is simple HTML:

```
method DO_REQUEST .

write( '<html><body><H1>' ).

write( 'This is my very first controller' ).

write( '</H1></body></html>' ).

endmethod.
```

9. Activate your class and your BSP application.
10. Before you can test the controller, in Transaction **SICF** you must also activate the new entry that was automatically created for your BSP application (see also **Activating and Deactivating an ICF Service**).

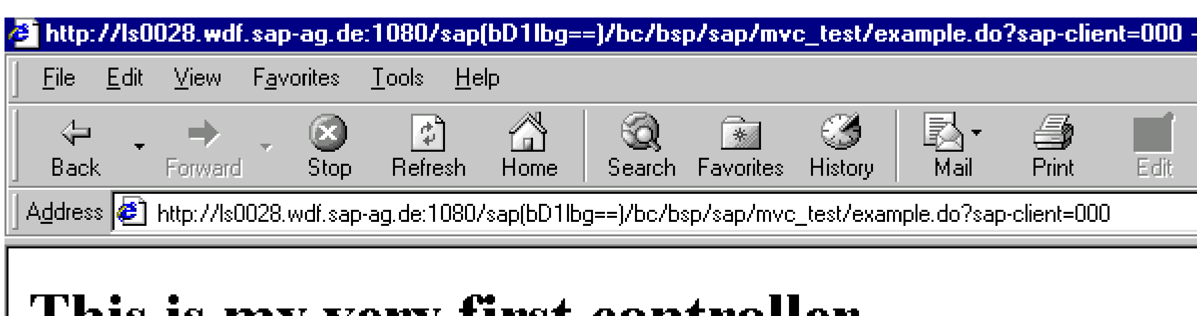
    In Transaction **SICF**, select the entry for your BSP application and choose *Service/Virt.Host → Activate.*



Confirm the following confirmation prompts.

11. You can now test the new controller page that you have created.

Result



Continue by **creating a view**.

Creating a View

Use

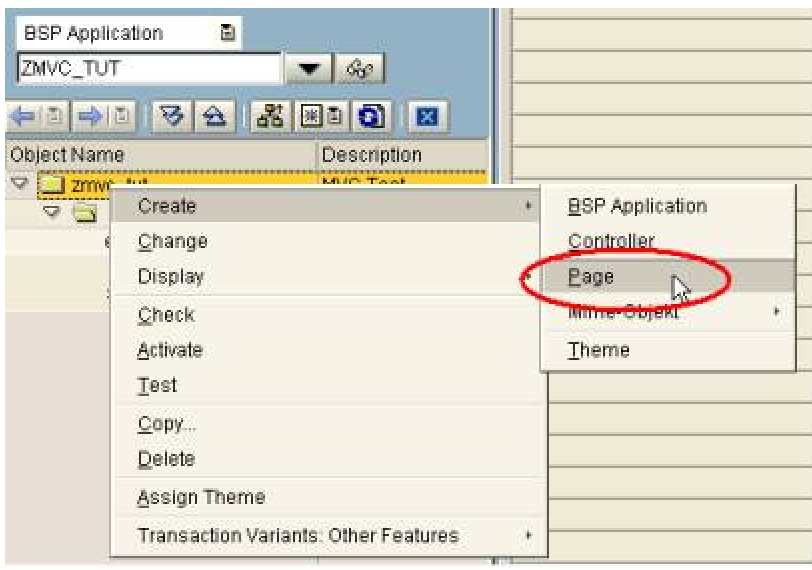
If you do not always want to use the `write` function to create the HTML page content (as described in **Creating a Controller**), and you want to create it as pure HTMLO layout instead, then create a view that you can call from the controller.

Procedure

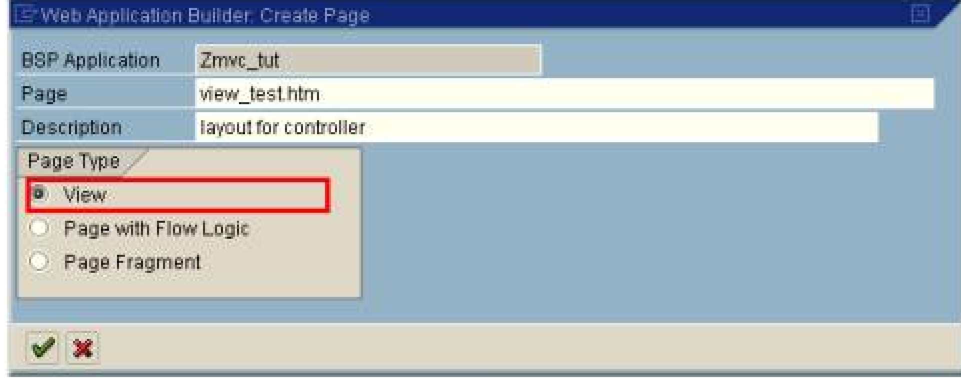
2. ...

5. 1. Begin as if you are creating a normal page with flow logic in your BSP application.

To do this, choose *Create* → *Page.*



6. 2. In the following dialog box, enter a name and short description of the view and select *View* as the page type:

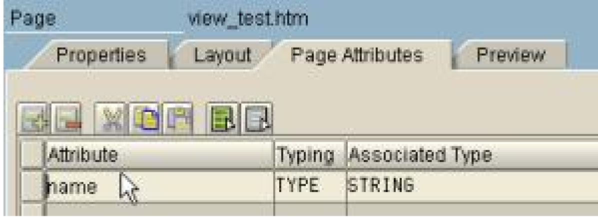


7. 3. Choose ✔.

8. 4. Create the attributes for the variable parts of the view.

You cannot define auto-page attributes, since views cannot be called directly from the browser.

Create the following attribute:



9. 5. Define the layout as usual:

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

<html>

  <head>

    <link rel="stylesheet" href="../../sap/public/bc/bsp/styles/sapbsp.css">

    <title> Layout for Controller </title>

  </head>

  <body class="bspBody1">

    <H1>View Example</H1>

    <H3>Hello, user <%= name%></H3>

  </body>

</html>
```

10. 6. Activate the view.
11. 7. Finally, adjust the **DO_REQUEST** method to the controller class.

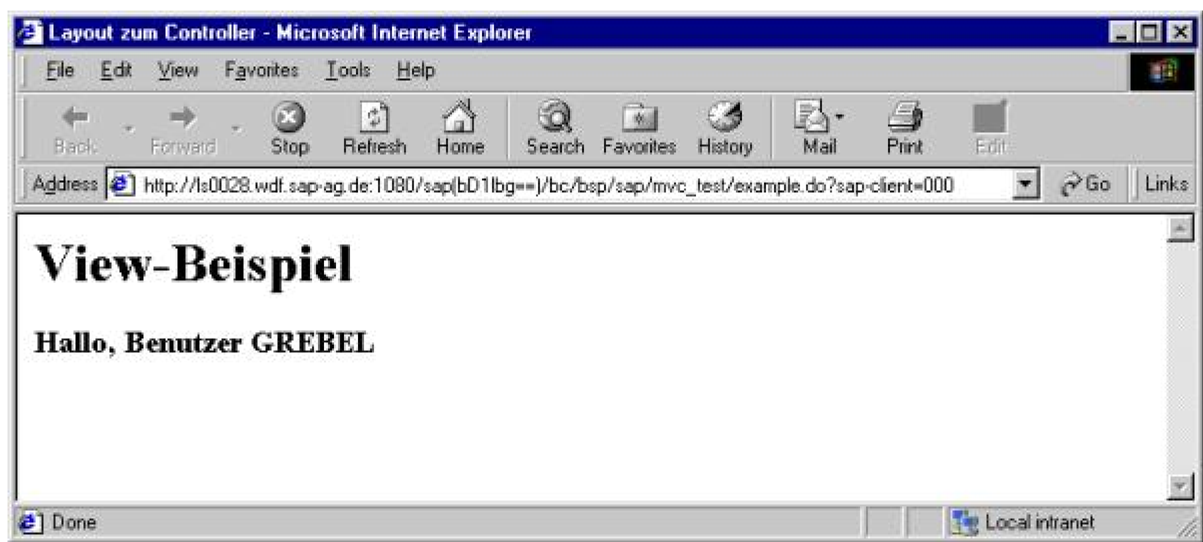
Here, the schema is always the same. First you create the view, then you set the attributes, and then you call the view. (For the time being you can ignore the warning concerning exception **CX_STATIC_CHECK**, or you can set a **try-catch** block around the calls):

```
method DO_REQUEST .

    data: myview type ref to if_bsp_page.

  myview = create_view( view_name = 'view_test.htm' ).

  myview->set_attribute( name = 'name' value = sy-uname ).

  call_view( main_view ).

endmethod.
```

12. 8. Activate your class and test your controller.

Result

You have created your own view for the layout.



Continue by Calling the Controller.

Calling a Controller

Use

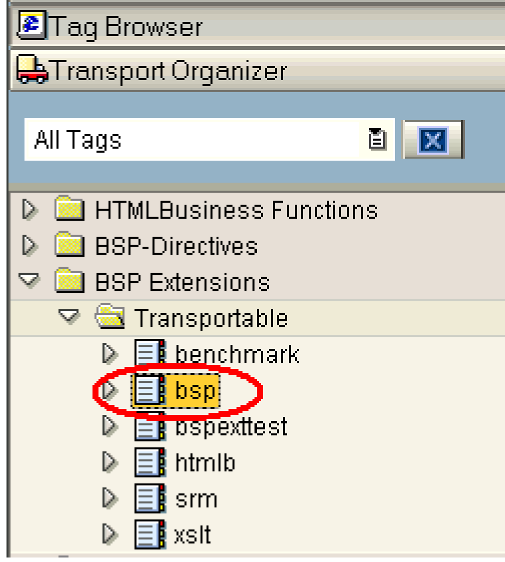
You can call a controller from a page with flow logic, or from a view.

Procedure

1. Create a page within your BSP application.

   Ensure that you select *Page with Flow Logic* as the page type.

2. In the Tag Browser, select BSP extension **`bsp`** and drag it to the second line of your BSP's layout under the page directive.



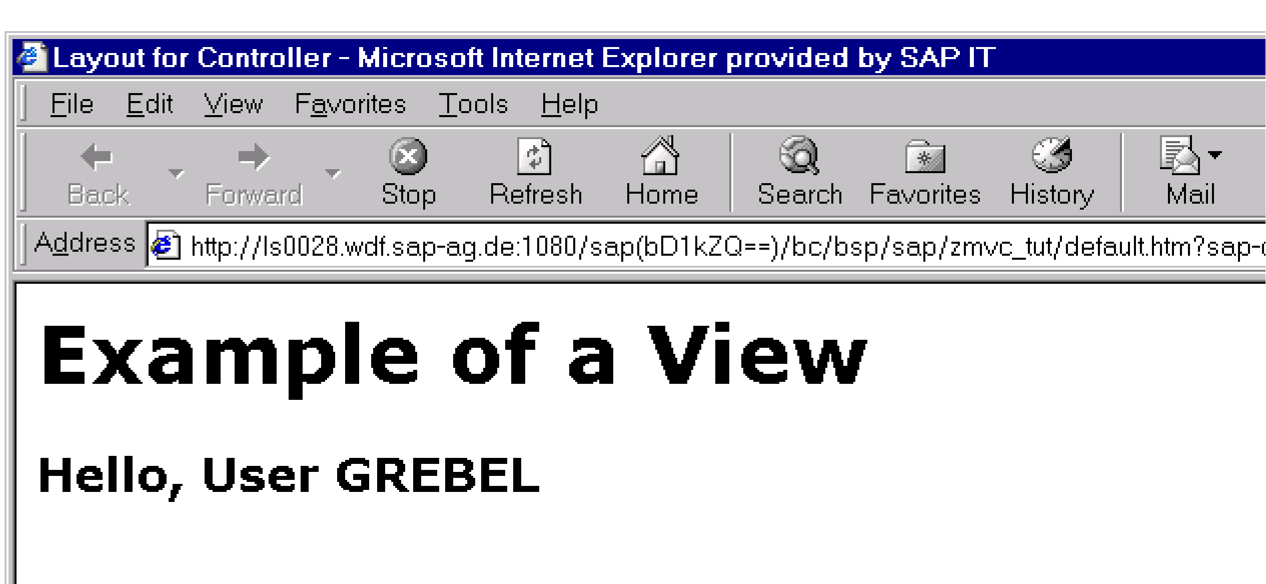
3. Now drag the **<bsp:goto>** directive of BSP extension **bsp** to the body of the HTML layout and add the URL parameter.

   The source now looks as follows:

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

<%@ extension name="bsp" prefix="bsp" %>

<html>

  <head>
    <link rel="stylesheet" href="../../sap/public/bc/bsp/styles/sapbsp.css">
    <title> Initial page </title>
  </head>

  <body class="bspBody1">
    <bsp:goto url="example.do"></bsp:goto>
  </body>

</html>
```

4. You can now activate and test the page.

   The page looks as follows:



Ensure that the page you have tested looks exactly the same as when you tested the controller. The URL is different, however. You can use *View Source* in the browser to see that nothing remains of the HTML text from the BSP, but that only the content of the view is displayed:

```
<html>
 <head>
  <link rel="stylesheet" href="../../sap/public/bc/bsp/styles/sapbsp.css">
  <title> Layout for Controller </title>
 </head>
 <body class="bspBody1">
 </head>
 <body class="bspBody1">
  <H1>View Example</H1>
  <H3>Hello, User GREBEL</H3>
 </body>
</html>
```

5. You can now try out the difference between the **<bsp:goto>** element and the **<bsp:call>** element.

If you use the `<bsp:call>` element instead of the `<bsp:goto>` element, the calling page text remains the same. In the view that is inserted, you should therefore delete the HTML text available on the outline page, otherwise these texts will be transferred twice.

6. You can add another attribute to the controller. This is a *public* class attribute.

   It is set using the `<bsp:parameter>` element. You can use it for example to control which view is called, or this value can be passed to the view.