NET200

SAP Web Application Server: Developing BSP Applications

mySAP Technology

Date	
Training Center	
Instructors	
Education Website	

Participant Handbook

Course Version: 2004 Q4
Course Duration: 5 Day(s)
Material Number: 50069796

An SAP course - use it to learn, reference it for work

About This Handbook

This handbook is intended to complement the instructor-led presentation of this course, and serve as a source of reference. It is not suitable for self-study.

Typographic Conventions

American English is the standard used in this handbook. The following typographic conventions are also used.

Type Style	Description
Example text	Words or characters that appear on the screen. These include field names, screen titles, pushbuttons as well as menu names, paths, and options. Also used for cross-references to other documentation both internal (in this documentation) and external (in other locations, such as SAPNet).
Example text	Emphasized words or phrases in body text, titles of graphics, and tables
EXAMPLE TEXT	Names of elements in the system. These include report names, program names, transaction codes, table names, and individual key words of a programming language, when surrounded by body text, for example SELECT and INCLUDE.
`Example text`	Screen output. This includes file and directory names and their paths, messages, names of variables and parameters, and passages of the source text of a program.
`Example text`	Exact user entry. These are words and characters that you enter in the system exactly as they appear in the documentation.
`<Example text>`	Variable user entry. Pointed brackets indicate that you replace these words and characters with appropriate entries.

Icons in Body Text

The following icons are used in this handbook.

Icon	Meaning
💡	For more information, tips, or background
➡	Note or further explanation of previous point
⚠	Exception or caution
1 2 3	Procedures
🖥	Indicates that the item is displayed in the instructor's presentation.

Contents

Course Overview

In this course, you will become familiar with the architecture of the SAP Web Application Server and learn how to program a Web application using Business Server Pages. You will first set up the layout of a business server page using HTML and use the ABAP scripting language to generate some dynamic parts of the layout. You will also learn how to include MIME objects in the application, provide BSP applications in multiple languages, and use topics to adjust the layout of BSP applications without making modifications. The course also discusses how to include data from SAP systems by calling BAPIs in the system and how to log on to the SAP Web Application Server You will learn how BSP extensions can be used to design the layout. The use of the BSP extension HTMLB is covered in particular detail.

Target Audience

This course is intended for the following audiences:

- Developers who want to create Web applications based on BSP applications

Course Prerequisites

Required Knowledge

- Basic knowledge of ABAP (BC400 – ABAP Workbench: Foundations and Concepts)
- Basic knowledge of HTML and HTTP

Recommended Knowledge

- NET050 - Web Application Development: Foundations
- Knowledge of object-oriented programming (preferably ABAP Objects)
- Knowledge of transaction programming

Course Goals

This course will prepare you to:

- Describe the system architecture of the SAP Web Application Server
- Develop Web applications that are based on the Business Server Page programming model

Course Objectives

After completing this course, you will be able to:

- Describe the system architecture of the SAP Web Application Server
- Describe the request/response cycle
- Name the components of a Business Server Page and a BSP application and describe their use
- Develop Web applications based on Business Server Pages
- Implement the layout of Business Server Pages using HTMLB elements
- Implement language-specific BSP applications
- Explain how to assign a desired corporate identity design without modification by assigning a topic
- Use data from other SAP systems by calling BAPIs in your BSP applications

SAP Software Component Information

The information in this course pertains to the following SAP Software Components and releases:

Unit 1

The SAP Web Application Server

Unit Overview

This unit provides an overview of the architecture of the SAP Web Application Server. You will learn how an HTTP request is received and processed by the SAP Web Application Server and how the HTTP response is created.

Unit Objectives

After completing this unit, you will be able to:

- Describe the system architecture of the SAP Web Application Server
- List the most important steps of an HTTP request/response cycle
- Describe the features and use of the transactions SMICM and SICF

Unit Contents

Lesson: System Architecture of the SAP Web Application Server

Lesson Overview

This lesson provides an overview of the system architecture of the SAP Web Application Server and describes an HTTP request/response cycle. You will become familiar with the transactions SMICM and SICF.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the system architecture of the SAP Web Application Server
- List the most important steps of an HTTP request/response cycle
- Describe the features and use of the transactions SMICM and SICF

Business Example

For certain BSP applications, Web developers must store specific information (such as logon data, error pages, and so on) in the system. Furthermore, Web developers must be aware of the errors and warnings that can occur when starting BSP applications and be able to look up system settings for communication using HTTP.

System Architecture

The classical SAP R/3 is implemented as a three-tier, client-server architecture with a presentation level, application level, and database level. SAP R/3 is scalable at presentation and application server level. This is an important prerequisite for a system because it allows many users to work simultaneously. The **SAP Web Application Server** is a further development of the classical client/server technology. The SAP kernel has been extended to include a new process, the **Internet Communication Manager** (ICM). It allows you to process requests from the Internet or intranet **directly** - such as those made through a browser using the HTTP protocol.

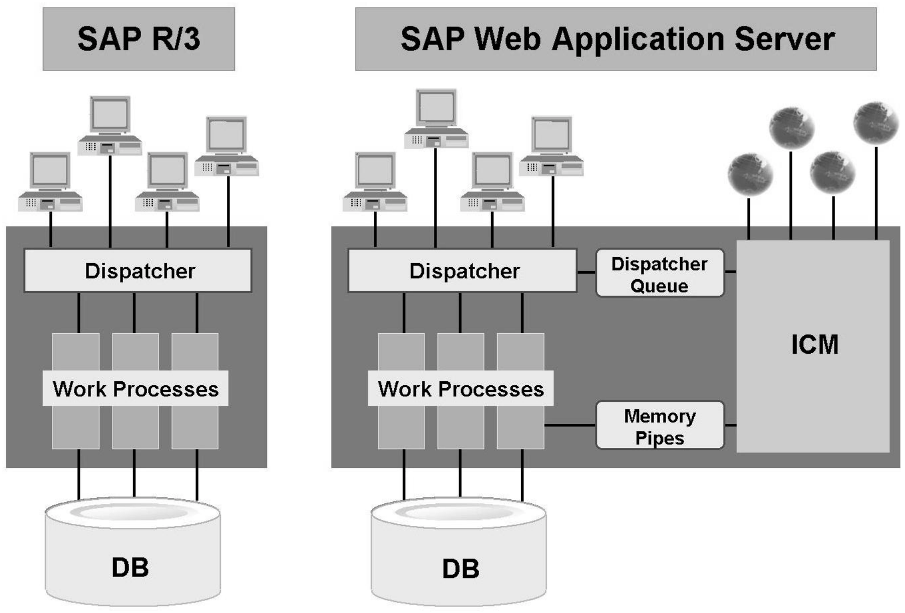


Figure 1: Architecture of the SAP Web Application Server

Hint: If a request (HTTP, HTTPs, SMTP) is received, the SAP Web Application Server acts as a **server**. If a request (HTTP, HTTPs, SMTP) is sent, the SAP Web Application Server acts as a **client**. Both roles are possible. Do not confuse the client role of the SAP Web Application Server with the role of sending a response.

Note: In an SAP system consisting of several application servers, the load is distributed using the message server (MS). This principle also applies to incoming requests from the Internet. By means of logon balancing the request is assigned to the application server that has the smallest load at that particular moment, using an HTTP REDIRECT. As of SAP Web Application Server Release 6.20, you can also distribute the load using a predefined Web switch, the SAP Web Dispatcher.

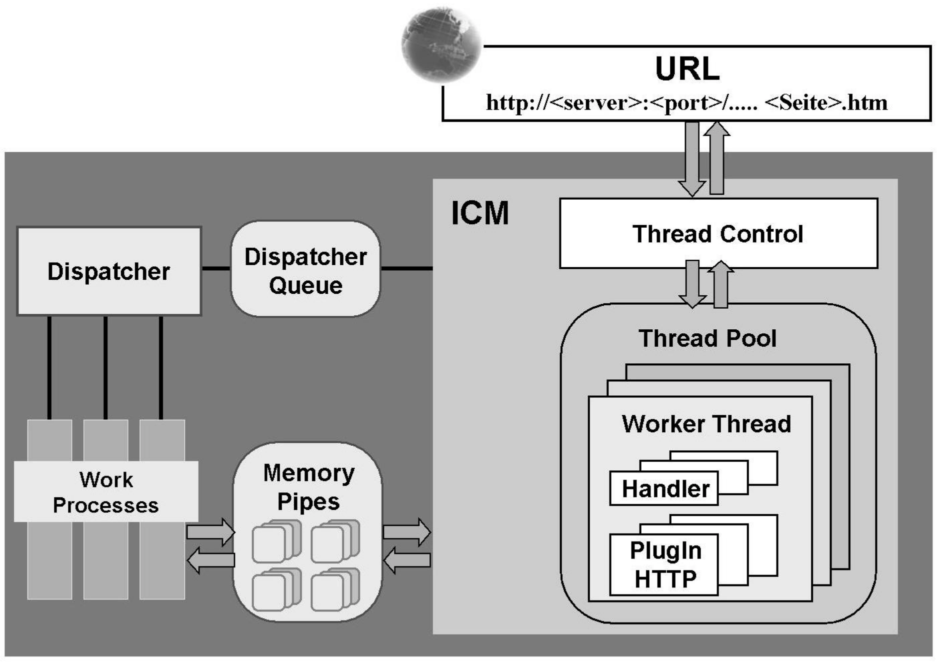


Figure 2: ICM Details

The **Internet Communication Manager (ICM)** enables communication between the SAP Web Application Server and the outside world using the protocols HTTP, HTTPS, and SMTP. The ICM is implemented as a process that is started, stopped, and monitored by the SAP Web Dispatcher. Profile parameters are used to configure the ICM.

As a server, the ICM can process requests from the Internet, whose URL contains the correct server/port combination pointing to the ICM. The ICM then calls the appropriate local handler, depending on the URL. Requests to the ICM are processed in worker threads. Because several worker threads are available (that is, in a thread pool), the load can be distributed. A special thread (thread control) distributes the requests to the available worker threads. A worker thread contains an I/O handler, for network input and output, and plug-ins for protocol-specific tasks.

The requests can sometimes be processed entirely in the ICM, but a data exchange with the SAP system is usually required to process the business logic. In this case, the data is passed on to a work process of the SAP system and a user context is created. The data is exchanged between the ICM and the work process using memory pipes.

Hint: You can display the monitor for the ICM by choosing *Tools -> Administration -> System Monitoring -> Monitor -> Internet Communication Manager* or by starting the transaction SMICM.. You can display the monitor for the work processes in the SAP System by choosing *Tools -> Administration -> System Monitoring -> Monitor -> Process Overview* or by starting transaction SM50.

The HTTP Plug-In

The HTTP plug-in handles HTTP requests and HTTP responses. Depending on the URL prefix, different local handlers are addressed. Profile parameters are used to define the assignment between the URL prefix and the local handler. The following operations are possible:

Logging handler
Recording of HTTP requests

Server cache handler
If the queried object is in the server cache, it is read from there. Objects that are sent back are stored in the server cache.

File access handler
Data in the file system of the SAP system is read directly.

HTTP redirect handler
An incoming HTTP redirect is passed on to another server.

SAP system handler
This handler passes the request on to the ABAP server of the SAP system, where it is processed by a work process.

J2EE handler
This handler passes the request on to the J2EE Engine.

If no J2EE server is configured, all requests that are not processed by the HTTP redirect handler or file access handler, or cannot be processed by the server cache handler, are passed on to the SAP system handler. However, if a J2EE Server is configured, only those requests whose URLs match the entries in the URL prefix table are passed on to the SAP system handler. The entries in this table are created from the structure of the HTTP service tree, which can be displayed and edited using transaction SICF.

The Internet Server Cache

The Internet server cache stores HTTP objects before they are sent to the client. If a new HTTP request is made for the object, it can be loaded from the server cache. The objects in the server cache have an expiration time, which can be set differently for each object. The server cache consists of a working memory cache and a hard disk cache. Both cache areas exist only once for each worker thread. MIME objects (static objects) are automatically stored in the server cache. The names of objects that cannot be found and hence cannot be stored in the server cache are included in the list of unfound objects (UFO). At each new request, the system checks the UFO list to see whether the object has already been requested but could not be found. Only then does the system try to load the object from the server cache.

Each object can be uniquely identified by a key (cache key) that results from the request URL. Using transaction SMICM, you can display the objects stored in the server cache and remove them from the cache if necessary.

A special feature of the Internet server cache is that it allows you to invalidate individual objects, or remove them from the cache, from within the applications.

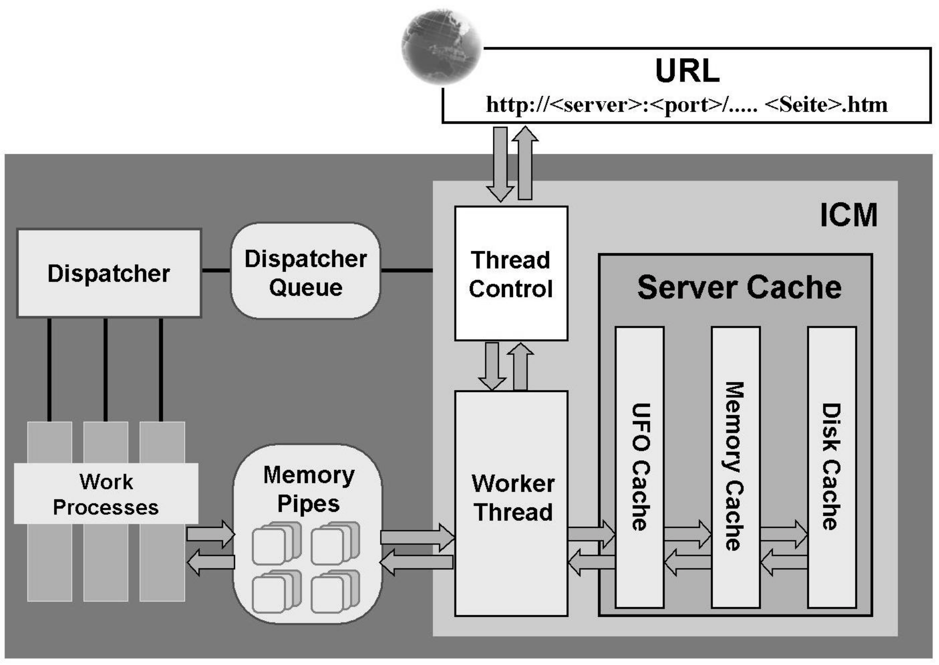


Figure 3: The ICM Server Cache

Internet Communication Framework

The Internet Communication Framework (ICF) provides the environment for handling an HTTP request in a work process of a SAP System (server and client). The ICF consists of ABAP classes and interfaces on the basis of which objects can be instantiated. These ABAP classes and interfaces, in turn, enable access to the request response data. The IF_HTTP_SERVER and IF_HTTP_CLIENT interfaces are of central importance here. The following lists the most important steps of an HTTP request/response cycle **(interaction model)** in the server case.

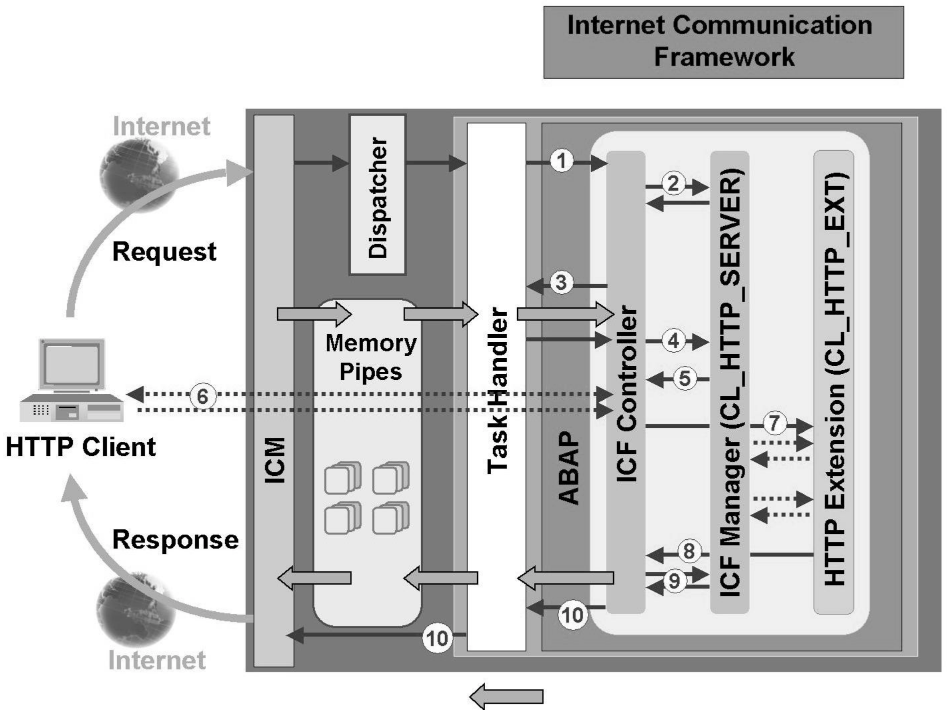


Figure 4: Flow of a Request/Response Cycle (Interaction Model)

The graphic shows the flow of an HTTP request/response cycle. The first two steps are not displayed.

- The HTTP request is received by the ICM.
- The SAP system handler processes the request (based on the URL prefix).
- The function module HTTP_DISPATCH_REQUEST is called (ICF controller) **(1)**.
- The server object (of the class CL_HTTP_SERVER) is created using the function module (server control block - here, ICF Manager) **(2)**.
- The request data is passed from the memory pipes to attributes of the server control block **(3)** and **(4)**.
- The HTTP request handler is selected **(5)**. The HTTP request handler is defined by the structure of the URL using the HTTP service tree.

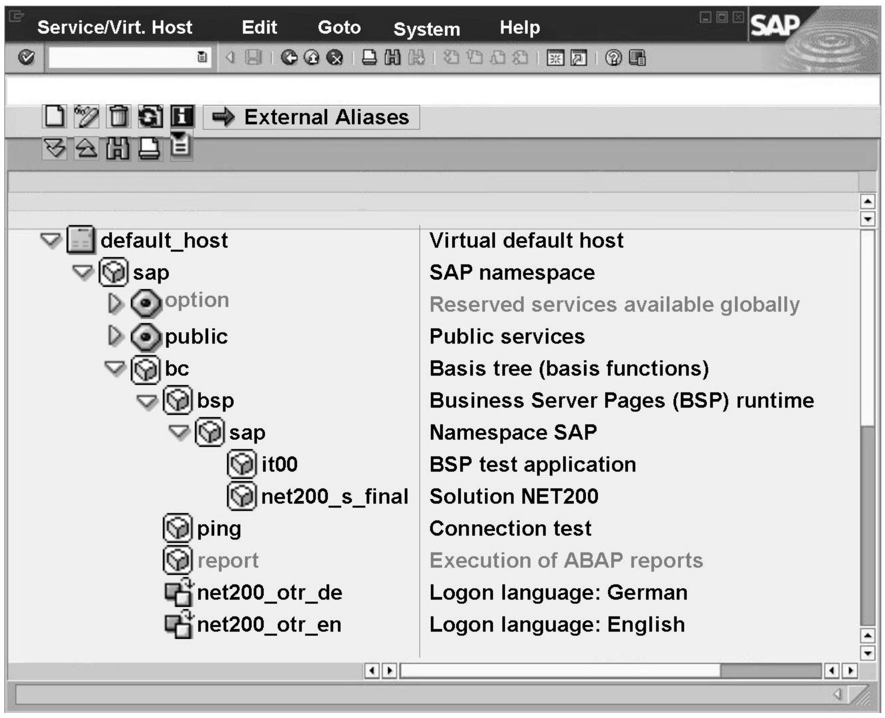
  Note: SAP provides the class CL_HTTP_EXT_BSP for starting BSP applications.
- Client logon (optional) **(6)**
- The HTTP request handler is called **(7)**. The HTTP request handler can use the server control block to access the request data and define the response data.
- Control is returned to the ICF controller **(8)**, which calls other handlers if necessary (according to the definitions in the HTTP service tree).
- The HTTP response is created and the data is written back to the memory pipes **(9)**.
- The data is sent back using the ICM and SAP System handler **(10)**.

The HTTP Service Tree

You define HTTP request handlers in the ABAP Workbench using the Class Builder. For this purpose, the handler class must implement the HANDLE_REQUEST method of the IF_HTTP_EXTENSION interface because this method is called by the server control block. In the HANDLE_REQUEST method, you can then access the server object.

The URL defines which request handler is called by the server control block. The request handler is assigned to the URL using transaction SICF. You can initially define several virtual Web servers that have different IP addresses, aliases (host header names), or ports. Requests whose URLs are not suited to any of the defined virtual Web servers are received by the default host.

For each virtual Web server, you can create a service tree whose node sequence correlates to the URL prefix. Each node can represent either a service for which an HTTP request handler and additional service-specific data (specification of the logon procedure for starting the service, explicit answer pages for error cases, and so on) can be stored or a reference to an existing service (alias). The stored logon data is accumulated using the tree. The services (including all subnodes) can be activated and deactivated using transaction SICF.



default_host	Virtual default host
sap	SAP namespace
option	Reserved services available globally
public	Public services
bc	Basis tree (basis functions)
bsp	Business Server Pages (BSP) runtime
sap	Namespace SAP
it00	BSP test application
net200_s_final	Solution NET200
ping	Connection test
report	Execution of ABAP reports
net200_otr_de	Logon language: German
net200_otr_en	Logon language: English

Figure 5: Transaction SICF: The HTTP Service Tree

SAP provides an HTTP service tree for the default host. The **BSP** service, which is used for calling BSP applications, is assigned to the node *sap/bc*. For all BSP applications, the system uses the HTTP request handler CL_HTTP_EXT_BSP, which is connected to this service.

Lesson Summary

You should now be able to:

- Describe the system architecture of the SAP Web Application Server
- List the most important steps of an HTTP request/response cycle
- Describe the features and use of the transactions SMICM and SICF

Unit Summary

You should now be able to:

- Describe the system architecture of the SAP Web Application Server
- List the most important steps of an HTTP request/response cycle
- Describe the features and use of the transactions SMICM and SICF

Unit 2

Business Server Pages: Programming Model

Unit Overview

In this unit, you will learn about the Business Server Pages programming model. You will learn how to create BSP applications and Business Server Pages, how data is managed in BSPs, how to navigate within a BSP application, and how states are managed in BSP applications.

Unit Objectives

After completing this unit, you will be able to:

- Describe the components of a BSP application
- Create BSP applications
- Create Business Server Pages
- Edit the layout of a Business Server Page
- Describe the components that make up a BSP and the tasks that each of them has
- Describe the event concept for BSPs with flow logic
- List the various options, and define and use the types and data objects in BSPs with flow logic
- Enable users to enter information
- Define static navigation between BSPs
- Define dynamic navigation between BSPs
- Enable data transfer between BSPs
- React to errors in transmitted data
- Work with global objects
- List criteria for deciding to use stateful or stateless programming
- Implement techniques for retaining data in a stateless BSP application

Unit Contents

Lesson: Business Server Pages: Introduction

Lesson Overview

This unit starts with a general overview of BSP applications. It explains which components can be included in a BSP application and in what contexts these components can be used. Finally, it discusses the creation of Business Server Pages with server-side scripting.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the components of a BSP application
- Create BSP applications
- Create Business Server Pages
- Edit the layout of a Business Server Page

Business Example

Now that a decision has been made to develop a Web application using the BSP programming model, you need to create a BSP application and several Business Server Pages.

BSP Application

A **BSP application** is a Web application that is functionally self-contained and is implemented using Business Server Pages (BSPs). BSP applications and BSPs are standalone development objects, which are developed using the Web Application Builder on the SAP Web Application Server. BSP applications are completely integrated in the SAP Web Application Server. This means, for example, that BSP applications can read data from the database or call BAPIs. The BSP application is assigned to a package and thus linked to the Transport Organizer. A BSP application can consist of the following components:

- Business Server Pages (BSPs)
- Application class
- MIME objects
- Theme
- Navigation structure
- Controller

There are three types of BSPs: pages with flow logic, page fragments, and views. You do not have to include all the above components in your BSP application. Simple applications can consist only of BSPs.

The use of controllers and views is supported as of SAP Web Application Server Release 6.20 in connection with the **Model View Controller design pattern**.

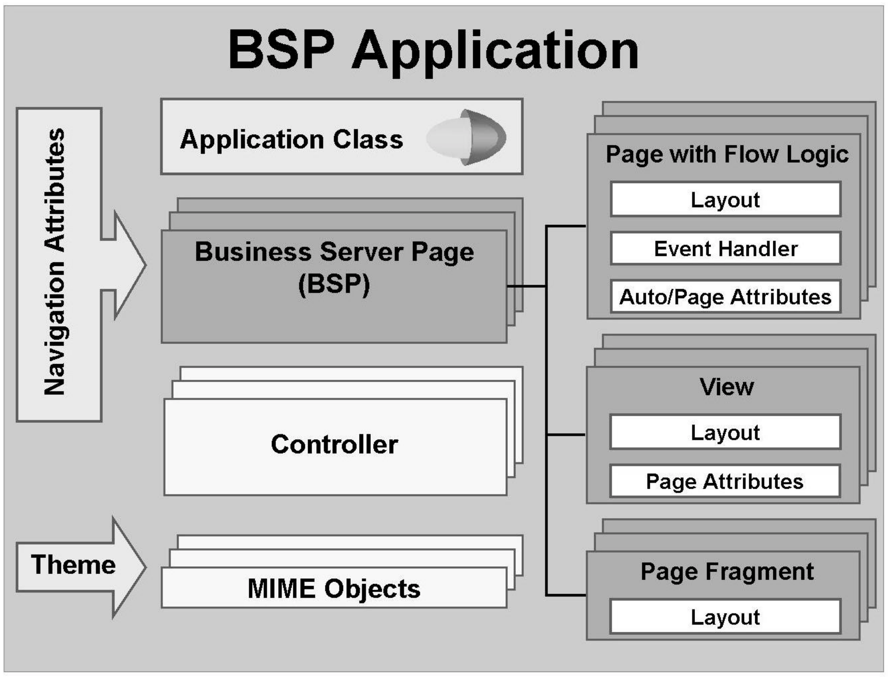


Figure 6: Components of a BSP Application

Business Server Pages define the Web pages that are displayed to the user in the Web browser when a BSP application is launched. The **layout** of a BSP can contain only static HTML source code. However, you can also display information in the layout that can be ascertained only at runtime, for example, through database selection. This means that the layout of a BSP usually consists of a static part (such as text or images) and a dynamic part (data from a database table). The static part of a BSP can be defined using hypertext markup language (HTML) or extensible markup language (XML). The dynamic part is implemented using **server-side scripting**. You can use either **ABAP** or **JavaScript** as your scripting language.

In addition to layout, the page also allows you to save application logic in event handlers (**page with flow logic**).

Conversely, **page fragments** always contain the layout of a Business Server Page. Page fragments are used in particular for modularization. (For example, if you want the same page header to appear on every page, you can define this as a page fragment and include it in all pages.) If a BSP is characterized as a page fragment in its attributes, it cannot be executed. This means that it can be neither addressed through its own URL nor used as the destination for a link from another page.

Views are used to display data within the framework of the Model View Controller design pattern.

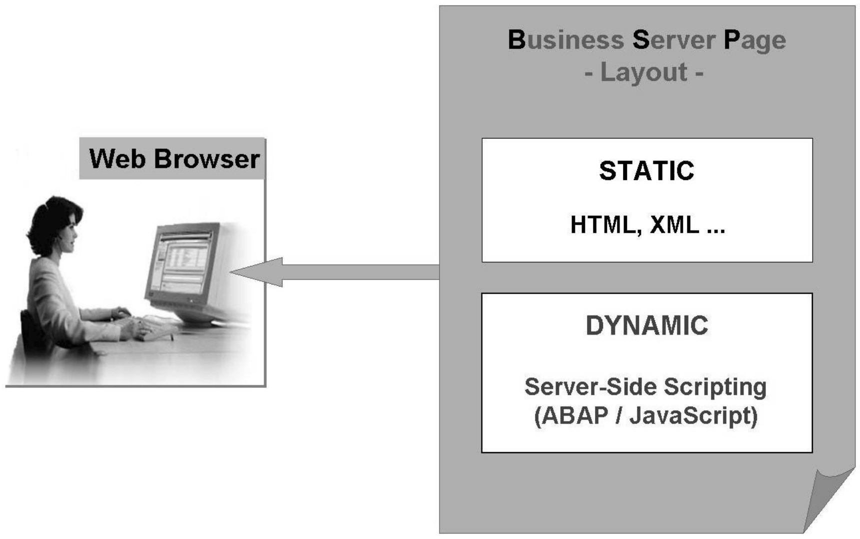


Figure 7: Layout of a Business Server Page

BSP directives are statements that need to be interpreted by the server at runtime. They are denoted in the layout using the scripting tag - that is, you enclose the statements between <% and %>.

The scripting language in the BSP is specified using the **page directive** as follows:

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

Hint: The page directive must be in the first line and first column of the layout. You cannot mix scripting languages within a single BSP. This applies in particular when you include page fragments in a page.

Further important directives are summarized in the following:

Include directive

Includes a page fragment.

Example: `<%@ include file='fragment.htm' %>`

Inline code

Flagging of ABAP or JavaScript source code in the layout of the BSP.

Example: `<% data: wa type spfli. %>`

Output of variable values

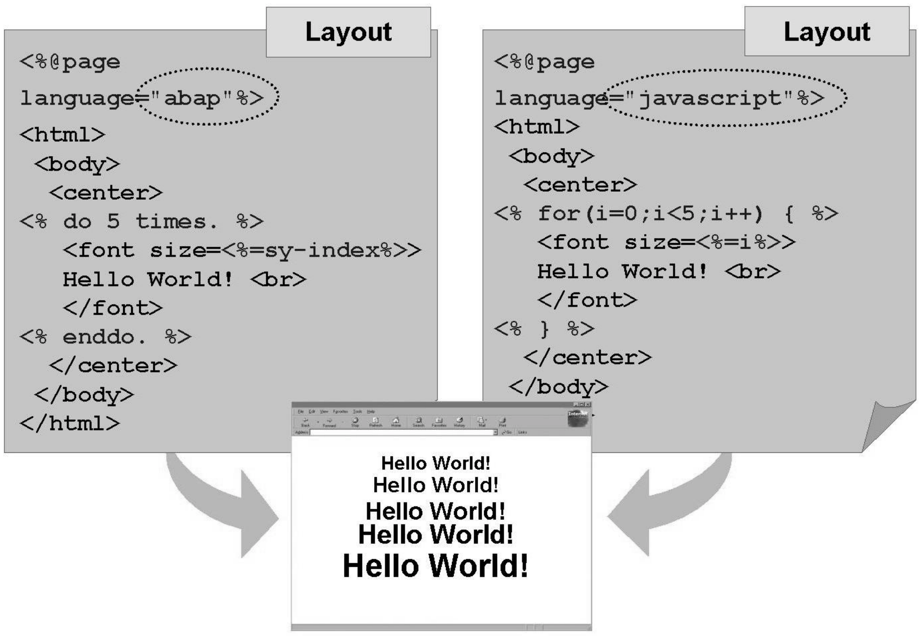
Displays the value of an elementary variable at the time of page processing.

Example: `<%= wa-carrid %>`

Comment

Server-side comment. Source-code between the relevant scripting tags is not interpreted by the server.

Example: `<%-- Loop at itab into wa. --%>`



Layout

```
<%@page
language="abap"%>
<html>
 <body>
  <center>
<% do 5 times. %>
   <font size=<%=sy-index%>>
   Hello World! <br>
   </font>
<% enddo. %>
  </center>
 </body>
</html>
```

Layout

```
<%@page
language="javascript"%>
<html>
 <body>
  <center>
<% for(i=0;i<5;i++) { %>
   <font size=<%=i%>>
   Hello World! <br>
   </font>
<% } %>
  </center>
 </body>
```

```
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

Figure 8: Example of Server-Side Scripting

Caution: The layout should be used only to specify the appearance of the page. It should be clearly separated from the business logic (for example, reading data from a database table).

Caution: For ABAP statements, the constraints for ABAP Objects apply.

Hint: Naturally, you can use JavaScript for client-side scripting, regardless of whether the server-side scripts are written in ABAP or JavaScript.

You can assign an **application class** to a BSP application. This class encapsulates business logic needed in the application. You can access the methods and attributes of the class from any page in the BSP application. For this purpose, the system automatically creates an object instance of this application class, which can be accessed using the instance name **application**. The application class must be created as a global class in the Class Builder (in the customer namespace Z_CL_).

The application class also allows you to define global attributes for the whole BSP application.

Hint: You can assign an application class to several BSP applications.

MIME stands for Multipurpose Internet Mail Extension. In the SAP Web Application Server, all **MIME objects** (such as graphics, style sheets, audio files, and video files) are stored in the MIME Repository. The MIME repository can be accessed from the Object Navigator (transaction: SE80).

A **theme** is a container for MIME objects. Using themes lets you customize the layout of pages, without having to change the layout source code. You can assign a theme to several BSP applications.

The term **navigation structure** refers to the sequence of BSPs within an application. Navigation requests, which allow you to follow navigation paths in the BSP application, are assigned to each path. The navigation paths are maintained in a table as a property of the BSP application.

Exercise 1: Creating a BSP Application

Exercise Objectives

After completing this exercise, you will be able to:

- Create a BSP application
- Create Business Server Pages
- Use ABAP scripting to create the layout for a BSP

Business Example

Now that a decision has been made to develop a Web application using the BSP programming model, you need to create a BSP application and several business server pages.

Task:

Create the the BSP application ZNET200_##_01 (where ## is your group number), with a BSP named start.htm. Sample solution for this exercise: NET200_S_01.

1. Create the package ZNET200_##.
2. Create the BSP application ZNET200_##_01 and assign it to your package ZNET200_##.
3. In the BSP application ZNET200_##_01, create a BSP (page with flow logic) with the name `start.htm`.
4. Edit the BSP layout. Make sure that a text (in this case, "Hello, World!") appears centered on the page.
5. Change the layout of your BSP to insert its dynamic parts using server-side ABAP scripting.

   Carry out one of the following two tasks:

   a) Change (for example, increase) the font size dynamically

   b) Make the text color striped

Solution 1: Creating a BSP Application

Task:

Create the the BSP application ZNET200_##_01 (where ## is your group number), with a BSP named start.htm. Sample solution for this exercise: NET200_S_01.

1. Create the package ZNET200_##.

   a) Start the Object Navigator (transaction: SE80). Start the Repository Browser and choose *Package*. Enter the package name in the appropriate field. Create the package using forward navigation. Enter a short description. Specify the application component **`CA`** and the software component **`HOME`**. The name of the transport layer depends on the system you are using.

2. Create the BSP application ZNET200_##_01 and assign it to your package ZNET200_##.

   a) Start the Repository Browser and choose *BSP Application*. Enter the name of your application and choose *Display*. The system now takes you through the steps of creating a BSP application. Save your application. When you save your application, assign the BSP application to your package ZNET200_## and to the transport request created for you.

3. In the BSP application ZNET200_##_01, create a BSP (page with flow logic) with the name **`start.htm`**.

   a) Display your BSP application in the Object Navigator. Position the cursor on the application and, using the right mouse button, choose *Create -> Page*. Enter the page name. Choose *Page with Flow Logic* as a property of your page. Save the page.

Continued on next page

4. Edit the BSP layout. Make sure that a text (in this case, “Hello, World!”) appears centered on the page.

   a)

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

<html>

  <head>
    <title>
      Simple example: Scripting in ABAP
    </title>
  </head>

  <body>
    <center>

<!----------------->
<!-- Static      -->
<!----------------->
      <h2>
        Static
      </h2>
      Hello World! <br>

    </center>
  </body>

</html>
```

5. Change the layout of your BSP to insert its dynamic parts using server-side ABAP scripting.

   Carry out one of the following two tasks:

   a) Change (for example, increase) the font size dynamically

Continued on next page

b) Make the text color striped

a)

```
<!------------------------------->
<!-- With scripting: Solution A -->
<!------------------------------->

    <h2>
      Solution A
    </h2>
    <% do 5 times. %>
      <font size="<%=sy-index%>">
        Hello World! <br>
      </font>
    <% enddo. %>
```

b)

```
<!------------------------------->
<!-- With scripting: Solution B -->
<!------------------------------->

    <h2>
      Solution B
    </h2>
    <% data: color type string,
             int   type i.
       do 6 times.
         int = sy-index mod 2.
         if int < 1.
           color = '#0000DD'.
         else.
           color = '#00DD00'.
         endif. %>
         <font color="<%=color%>">
           Hello World! <br>
         </font>
    <% enddo. %>
```

Lesson Summary

You should now be able to:

- Describe the components of a BSP application
- Create BSP applications
- Create Business Server Pages
- Edit the layout of a Business Server Page

Lesson: Layout of a Business Server Page

Lesson Overview

This lesson explains in detail the components that make up a Business Server Page (BSP). We will present the event concept of Business Server Pages for the classic programming model (page with flow logic). Events represent points in time during the processing of a BSP. Finally we will explain how data is managed in a BSP.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the components that make up a BSP and the tasks that each of them has
- Describe the event concept for BSPs with flow logic
- List the various options, and define and use the types and data objects in BSPs with flow logic

Business Example

Now that you have created the BSPs, the business logic must be implemented. The decision has been made **not** to use the Model View Controller Design Pattern. This means that the control flow, the business logic, and the layout form an object (BSP with flow logic). Now you need to decide the source code that you want to execute at each event. As part of this , the appropriate data objects must be created and have types assigned to them.

Components of a Business Server Page

BSPs can be made up of the following components, depending on the page type (A = page with flow logic, V = view, and S = page fragment):

- Properties (A, V, S)
- Layout (A, V, S)
- Page attributes (A, V)
- Type definitions (A)
- Event handlers (A)

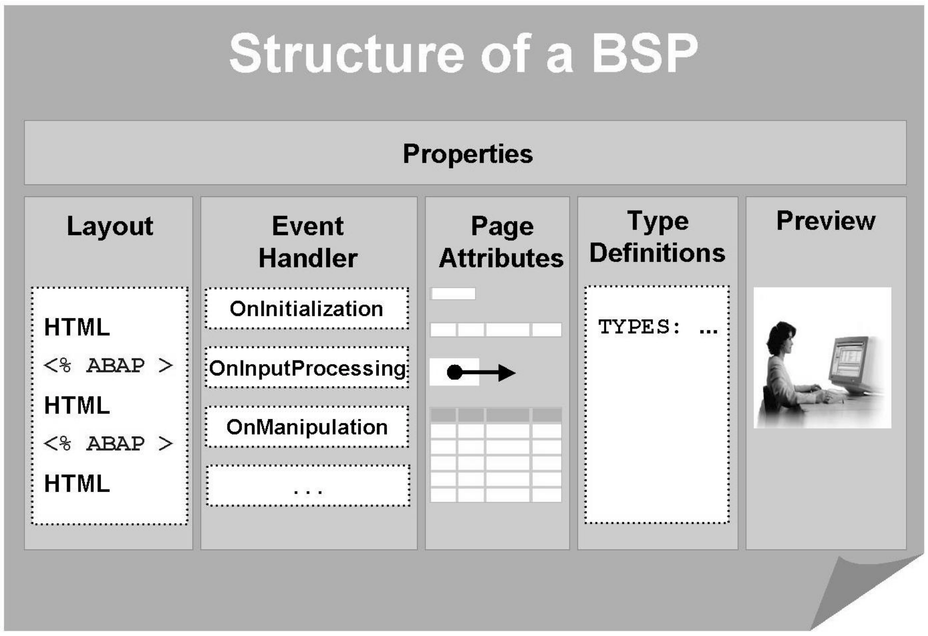


Figure 9: Layout of a Business Server Page

The **properties** of a BSP include the page type.

An error page can be assigned to views and pages with flow logic. If a runtime error occurs, this error page is then sent to the Web client.

The source code of the layout can be compressed.

Pages with flow logic have their own URL. You can specify that certain pages are to be executed using HTTPs only. In addition, the data from the HTTP response can be compressed on the server so that the HTTP request/response cycle time is shorter, and the network load lower.

In the **Layout**, you specify how the page will appear in the browser. This can consist of both static HTML elements - like text display - and dynamic elements. You implement the dynamic elements using ABAP or JavaScript.

There are predefined **event handlers** that are processed in a defined sequence when a Business Server Page is interpreted. The handlers thus reflect the time required for processing a page. Event handlers allow separation of static (layout) and dynamic (business logic) source code.

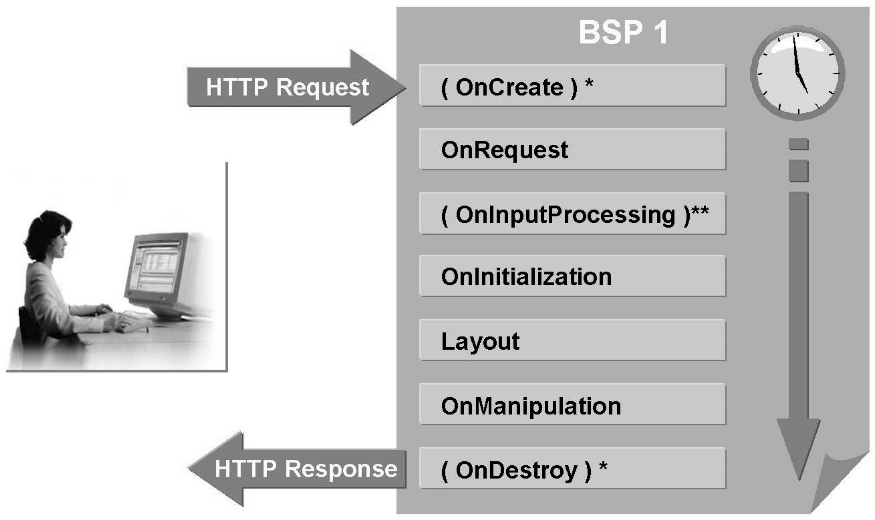
The **page attributes** serve two purposes: They provide the external interface for the Business Server Page (automatic page attributes), and they are used to define the fields, work areas, and internal tables whose definitions are always visible within this page (non-automatic page attributes).

In the **type definition**, you can define ABAP types that can be accessed by any event associated with the page. In particular, page attributes can be set for pages on the basis of these types.

The **Preview** displays a preview of the static HTML elements of the page.

Event Concept for Business Server Pages

In each BSP source code can be defined, that is related to standard handlers, known as event handlers. These event handlers are executed in a predefined sequence.



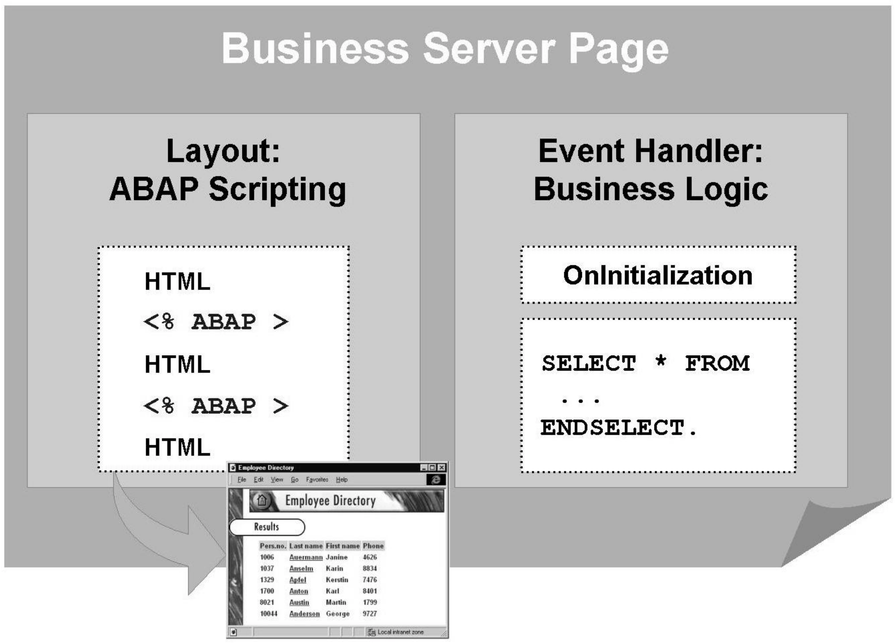
( ) * Different for stateless and stateful BSP applications

( ) Executed only under certain conditions**

Figure 10: Components of a Business Server Page

The event handlers are written in ABAP. All the constraints that apply to ABAP Objects programming also apply to these event handlers. You program the business logic of a BSP in the event handlers - for example, reading from the database, or calling function modules or BAPIs.

Note: The system generates an ABAP Objects method for each event handler that contains the code stored in the handler.



```
HTML
<% ABAP >
HTML
<% ABAP >
HTML
```

```
SELECT * FROM
  ...
ENDSELECT.
```

Pers.no.	Last name	First name	Phone
1006	Auermann	Janine	4626
1037	Anselm	Karin	8834
1329	Apfel	Kerstin	7476
1700	Anton	Karl	8401
8021	Austin	Martin	1799
10044	Anderson	George	9727

Figure 11: ABAP Programming Within a BSP

The following list shows the event handlers with their attributes and includes examples for their use.

OnCreate

- Executed first
- Will be executed only once for "stateful"; otherwise at each call of a BSP
- Therefore appropriate only for use with "stateful"
- Initializes data and creates objects

OnRequest

- Executed every time a BSP is accessed
- Restores the internal data structures from the request
- Should always be used if logic is to be executed independently of further navigation and independently of whether the application is stateful or stateless

OnInitialization

- Is run through after OnRequest
- Used mainly for data retrieval (that is, reading from database tables, filling internal tables)
- Should always be used if processing of the source code is to take place only if the layout will be processed as well

OnInputProcessing

- Executed under certain conditions (after a user dialog)
- Used for processing user inputs and subsequent navigation to a next page

OnManipulation

- Used for manipulating the HTTP data stream according to the layout

OnDestroy

- Is the last to be executed
- Only executed for "stateful" if the page is destroyed
- Therefore appropriate only for use with "stateful"
- Used for deleting information at the end of a BSP application

Data Definition and Visibility Within a Business Server Page

The process of defining data for a BSP follows simple rules. One important criterion is the visibility necessary.

If the data object has to be available **at several times** - that is, in several event handlers - then you must define this data object as a **page attribute** of the Business Server Page.

If the data object needs to be available only **during a single event**, then you create the data object in the appropriate event handler.

All data objects can be typed using ABAP Dictionary types or the predefined ABAP types. In the same way as the page attributes, you can define types for a Business Server Page. These types are then visible at each event (in each event handler). It is possible to refer to these types also when creating page attributes. If types are created within an event handler, they are visible only in this event handler.

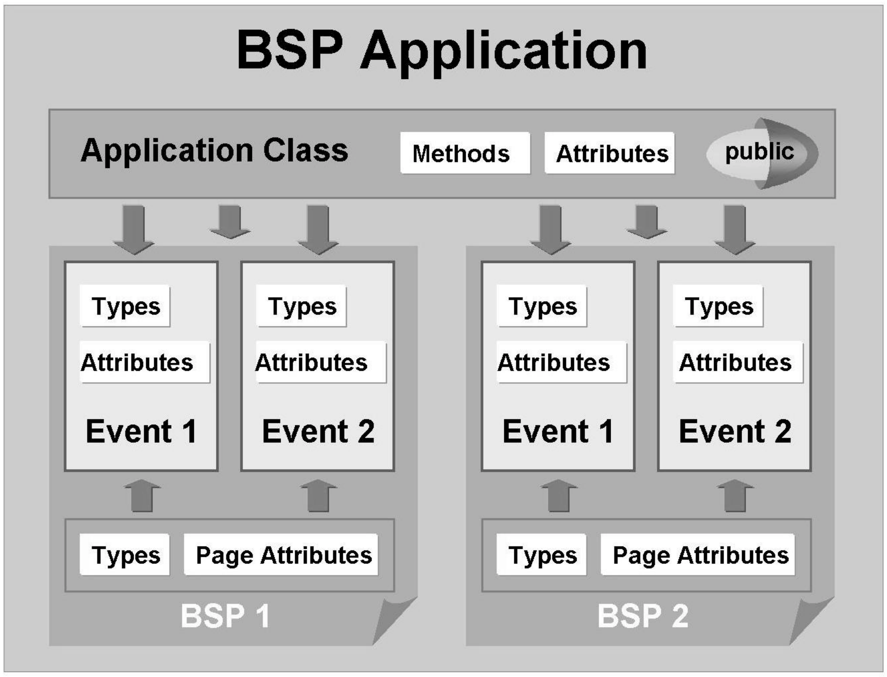
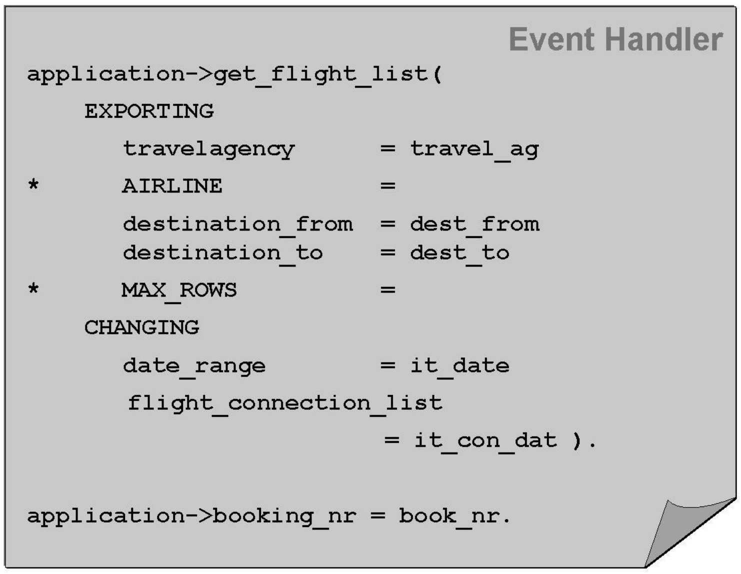


Figure 12: Visibility of Types and Attributes in a Business Server Page

Finally, the public types and attributes of the application object are visible in all BSPs of the BSP application and in all events of a BSP. To have this, you must enter the application class in the respective field in the tab *Attributes* of the BSP application. Whenever a BSP of the application is called, an instance of this class with the name *APPLICATION* is automatically created. In case the BSP application is executed "stateful", the attributes of the application class can be used for exchanging data between two BSPs since the object *APPLICATION* is kept after sending the HTTP response. Using the application class, you can encapsulate business logic and make it accessible through public methods.

BSP Application with Application Class


Event Handler

```
application->get_flight_list(
    EXPORTING
       travelagency      = travel_ag
*      AIRLINE           =
       destination_from  = dest_from
       destination_to    = dest_to
*      MAX_ROWS          =
    CHANGING
       date_range        = it_date
       flight_connection_list
                          = it_con_dat ).

application->booking_nr = book_nr.
```

Figure 13: Using an Application Class

Exercise 2: Creating a BSP Application

Exercise Objectives

After completing this exercise, you will be able to:

- Incorporate a page fragment in a BSP that includes flow logic
- Implement source code in the event handlers of a BSP
- Define types and data objects that are visible in every event of a BSP or in only one event of a BSP

Business Example

You need to create a self-service flight booking application. To do this, create the BSP application and individual BSPs. The first page of the BSP application should always display last-minute offers. This means that you must make sure that the most up-to-date data is read from the database before the data is sent to the browser.

Task:

Create a new BSP application with 5 BSPs. You should implement the page header of each BSP by incorporating a page fragment. The first page should display flight data for last-minute offers.

1. Create a BSP application named ZNET_##_02, where ## is your group number. If you have a a single-digit group number, always enter it in the form 0#. Assign the BSP application and the subobjects you create in the following exercises to your package ZNET200_##. Sample solution for this exercise: NET200_S_02.
2. Create five BSPs, giving them the following names and descriptions. Save the Business Server Page.

Page name	**Title**
public/start.htm	Select flight
public/flights.htm	Flight connections
public/details.htm	Flight details
protected/customer.htm	Display customer data
protected/confirm.htm	Confirm booking

Continued on next page

3. Create the page fragment, *header.htm*. Give it the description `Page Header Include`. In the Layout section, create a page header that will appear at the top of each page. For example, display the name of your travel agency and add a horizontal separator. Insert the page fragment in the Business Server Pages *public/start.htm* , *public/flights.htm*, and *public/details.htm*. Activate the BSP application and test the pages.

4. The following tasks pertain to the BSP **public/start.htm**. The first page should display last-minute offers, which can be booked by travel agency number **110**. Use the method *GET_LAST_MINUTE_FLIGHTS* of the application class *CL_NET200S_FINAL* to read the flights between today and exactly 3 weeks after today from the database. (Use the parameter *i_range*.) Limit the number of records returned by the method to 5 (using the parameter *i_max_rows*). Create a suitable page attribute (with the type of an internal table) for the corresponding interface parameter of the method (*e_flights*).

5. Display the last-minute offers in an HTML table on the page. Assign a column header to the columns you display. Display the following fields:

Column	**Description**
flightdate	Flight date
cityfrom	Departure location
cityto	Arrival Location
flightconn	Flight connection number
numhops	Number of individual connections
flighttime	Total duration of flight

6. Improve the appearance of your pages by defining style attributes. For example, specify the background color and font for the whole page. Refer to the model solution. Store the style attributes in a page fragment and insert this fragment in the *<head>...</head>* section of all your BSPs.

Solution 2: Creating a BSP Application

Task:

Create a new BSP application with 5 BSPs. You should implement the page header of each BSP by incorporating a page fragment. The first page should display flight data for last-minute offers.

1. Create a BSP application named ZNET_##_02, where ## is your group number. If you have a a single-digit group number, always enter it in the form 0#. Assign the BSP application and the subobjects you create in the following exercises to your package ZNET200_##. Sample solution for this exercise: NET200_S_02.

    a) Start the Object Navigator (transaction: SE80). Start the Repository Browser and choose *BSP Application*. Enter the name of your application and choose *Display*. The system now takes you through the process of creating a BSP application. Save your application. When doing so, assign the BSP application to your package ZNET200_## and to the transport request created for you.

2. Create five BSPs, giving them the following names and descriptions. Save the Business Server Page.

Page name	**Title**
public/start.htm	Select flight
public/flights.htm	Flight connections
public/details.htm	Flight details
protected/customer.htm	Display customer data
protected/confirm.htm	Confirm booking

    a) Create the BSPs using these names. Choose *Page with Flow Logic*. Two subnodes will appear: public and protected, containing all the pages. The titles are generated automatically in the pages.

3. Create the page fragment, *header.htm*. Give it the description `Page Header Include`. In the Layout section, create a page header that will appear at the top of each page. For example, display the name of your travel agency and add a horizontal separator. Insert

Continued on next page

the page fragment in the Business Server Pages *public/start.htm* , *public/flights.htm,* and *public/details.htm*. Activate the BSP application and test the pages.

a) Create a BSP logo.htm and check the indicator *Page Fragment*.

logo.htm - LAYOUT

```
<h2>
  SAP Travel Agency: Book Your Flights Online!
</h2>
<hr>
```

b) Add the Include directive to the following pages:

public/start.htm, public/flights.htm, and public/details.htm - LAYOUT

```
<%@page language="ABAP"%>
...
  <body>
<!------------------------------------------->
<!-- Page Fragment with the page header     -->
<!------------------------------------------->
    <%@ include file="Header.htm" %>
    ...
```

4. The following tasks pertain to the BSP **public/start.htm**. The first page should display last-minute offers, which can be booked by travel agency number **110**. Use the method *GET_LAST_MINUTE_FLIGHTS* of the application class *CL_NET200S_FINAL* to read the flights between today and exactly 3 weeks after today from the database. (Use the parameter *i_range*.) Limit the number of records returned

Continued on next page

by the method to 5 (using the parameter *i_max_rows*). Create a suitable page attribute (with the type of an internal table) for the corresponding interface parameter of the method (*e_flights*).

a) Assign the application class *CL_NET200S_FINAL* to your BSP application. Use the pattern function to call the method *GET_LAST_MINUTE_FLIGHTS* at the *OnInitialization* event. The object name is **`APPLICATION`**. Navigate to the method source code by double-clicking the method name. Display the interface parameters and their types by choosing the button *Signature*. On the page *public/start.htm,* create a page attribute of the same type. Assign the following values to the method parameters:

Parameter	Values
i_max_rows	Number of records to be read
i_range	Time (in days) from the system date
i_travelagency	Travel agency number
e_flights	Flight list with last-minute flights

public/start.htm - PAGE ATTRIBUTES

it_last_minute TYPE net200_tt_bapiscodat

public/start.htm - OnInitialization

```
* Data Retrieval for Last Minutes Offers
CALL METHOD application->get_last_minute_flights
  EXPORTING
    i_range        = 21
    i_max_rows     = 5
    i_travelagency = '00000110'
   IMPORTING
    e_flights      = it_last_minute.
```

5. Display the last-minute offers in an HTML table on the page. Assign a column header to the columns you display. Display the following fields:

Continued on next page

Column	Description
flightdate	Flight date
cityfrom	Departure location
cityto	Arrival Location

Continued on next page

flightconn	Flight connection number
numhops	Number of individual connections
flighttime	Total duration of flight

a)

public/start.htm - LAYOUT

```
<!--------------------------------------->
<!-- Last Minute Offers               -->
<!--------------------------------------->
<h3>
  Last Minute Offers
</h3>

<table>
  <thead>
    <tr>
      <td> Flight Date</td>
      <td> Departure</td>
      <td> Destination</td>
      <td> Connection ID</td>
      <td> Number of Hops</td>
      <td> Total Flight Time</td>
    </tr>
  </thead>
  <tbody>
  <% data wa_last_minute like line of it_last_minute.
     loop at it_last_minute into wa_last_minute.%>
       <tr>
         <td><%= wa_last_minute-flightdate%></td>
         <td><%= wa_last_minute-cityfrom%></td>
         <td><%= wa_last_minute-cityto%></td>
         <td><%= wa_last_minute-flightconn%></td>
         <td><%= wa_last_minute-numhops%></td>
         <td><%= wa_last_minute-flighttime%></td>
       </tr>
  <% endloop. %>
  </tbody>
</table>
```

Continued on next page

6. Improve the appearance of your pages by defining style attributes. For example, specify the background color and font for the whole page. Refer to the model solution. Store the style attributes in a page fragment and insert this fragment in the *<head>...</head>* section of all your BSPs.

   a)

styles.htm - LAYOUT (EXAMPLE)

```
<style>
  body      {background-color:rgb(204,204,255);
             font-family     :Arial}
  table     {border:solid;
             border-collapse:collapse;
             empty-cells:show;
             width:100%}
  thead     {font:bold}
  tr        {border:solid}
  td        {border:solid;
             border-width:1px;
             padding:3px;
             background-color:rgb(204,204,204)}
  hr        {height:5;
             background-color:rgb(0,0,0)}
  .noborder {border:none;
             background-color:rgb(204,204,255)}
</style>
```

All BSPs - LAYOUT

```
<head>
  ...
  <%@ include file = "styles.htm" %>
</head>
```

Lesson Summary

You should now be able to:

- Describe the components that make up a BSP and the tasks that each of them has
- Describe the event concept for BSPs with flow logic
- List the various options, and define and use the types and data objects in BSPs with flow logic

Lesson: Processing User Input

Lesson Overview

This lesson outlines how you enable users to enter information on a Business Server Page and how you have this information processed in the OnInputProcessing event. The lesson also describes the options for navigating statically or dynamically to the next page, and how to pass data to the next page.

Lesson Objectives

After completing this lesson, you will be able to:

- Enable users to enter information
- Define static navigation between BSPs
- Define dynamic navigation between BSPs
- Enable data transfer between BSPs
- React to errors in transmitted data
- Work with global objects

Business Example

It should be possible to navigate between different pages in a Web application. The user input should also be analyzed. If an error occurs (such as an invalid date), an appropriate user dialog should take place. Data (user input or page attributes) should be exchanged between the pages of the application (which is stateless for the time being).

Making User Input Possible with HTML Forms

To allow users to enter data on an HTML page, you must define input fields in an HTML form.

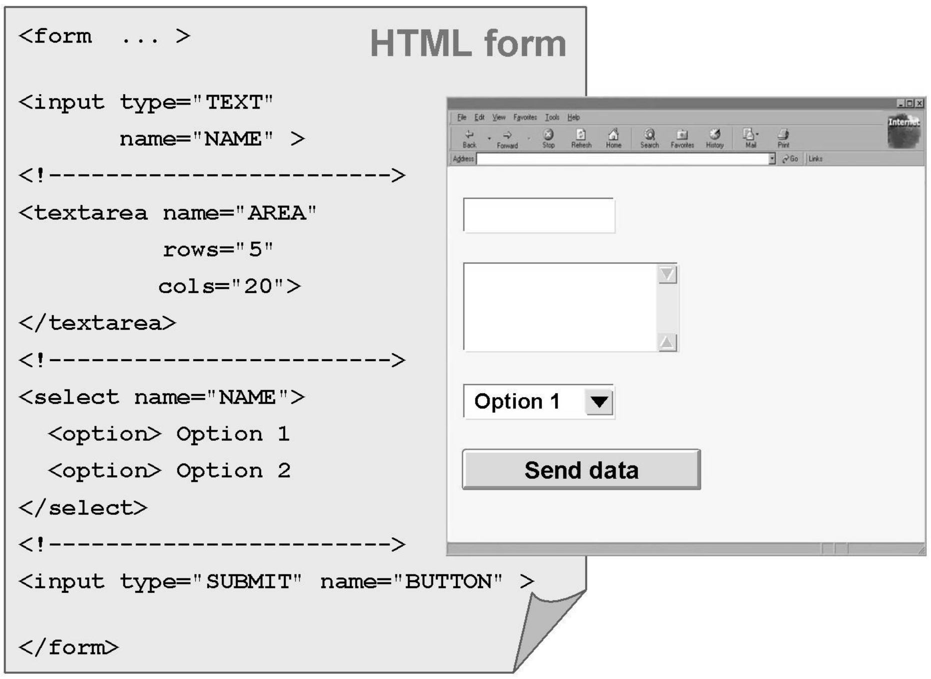
You can define forms at any point on an HTML page using **<FORM> ... </FORM>**. You can provide different elements – such as input fields, output fields, selection lists, or checkboxes – on the form. An HTML page can contain several forms. Forms cannot be nested. The following table summarizes the most important tags used in HTML forms.

Important Elements in HTML Forms

Field type	HTML Tag	Comments
Input/output field	`<input type="text" name="A" value="B">`	The field is filled with the value B.
Password field	`<input type="password" name="A">`	
Checkbox	`<input type="checkbox" name="A" value="X">`	The name/value pair `A=X` is then sent to the receiver in the HTTP request, if the checkbox is ticked.
Radio button	`<input type="radio" name="A" value="X">`	All selection buttons in a group have the same name (but different values).
Dropdown list	`<select name="A"> <option value="B"> Disp </option> ... </select>`	The user sees the values specified between `<option>` and `</option>` (in this example, Disp).
Input area with more than one line	`<textarea name="A" rows="..." cols="...">`	
Send button	`<input type="submit" name="A" value="Send">`	The text specified after the `value` addition appears on the button as a label.

The simplest way to send an HTML form to the server is to press a Send button, which must be in the form. When the user does this, the Web browser searches the whole form for input elements and combines them to form a **query string**. The query string consists of a **name/value pair** for each element in the form. Each name/value pair is separated by an ampersand (**&**). The name of a name/value pair is specified by the *name* attribute, while the value is specified either by the *value* attribute or by the user input in the field.

An alternative to sending the form is for the user to click an HTTP hyperlink. Hyperlinks are defined by the **<a href="...">** tag and by the corresponding closing tag **</a>**. A hyperlink can be either a picture or a text enclosed between the opening and closing tags. It does not matter to the hyperlink whether it is defined in the form or outside it. The query string is not created automatically. Instead, the URL is used as defined in the *href* attribute of the HTML element. There is another way of using a hyperlink to call a JavaScript function defined on an HTML page. This approach is very common in Web programming. You use JavaScript to generate a query string from the form data and then send this to the receiver with the next request.



```
<form  ... >

<input type="TEXT"
       name="NAME" >
<!------------------------>
<textarea name="AREA"
          rows="5"
          cols="20">
</textarea>
<!------------------------>
<select name="NAME">
  <option> Option 1
  <option> Option 2
</select>
<!------------------------>
<input type="SUBMIT" name="BUTTON" >

</form>
```

Figure 14: Example of an HTML Form

On the server, the data is generally passed to a program, which analyzes the query string and starts an appropriate action. In this way it is possible to navigate to different subsequent pages, depending on the data that has been sent (**dynamic navigation**). You can also define the next page statically. This means that the next page is specified on the HTML page itself, not by the program analyzing the data passed in the query string. This is known as **static navigation**. We will examine these two types of navigation more closely in the following sections.

Specifying the Next Page Statically

The definition of the HTML form includes attributes that specify how the form is to be processed. You list these attributes in the opening form tag.

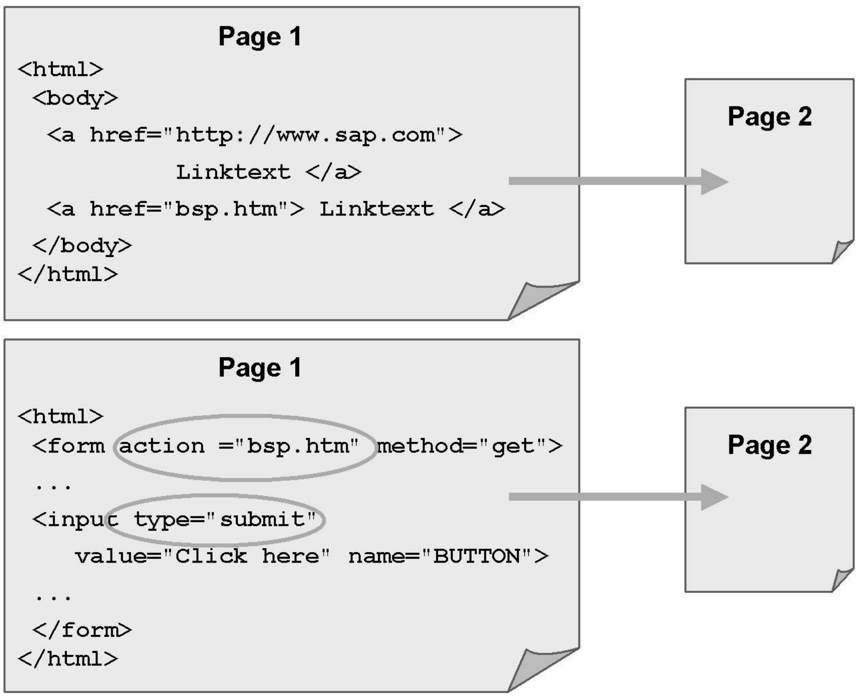
```
<FORM method=... action=... >
```

Attributes Used to Process Forms

Attribute	Description
method="type"	Specifies how the system sends the form to the server (that is, which method is used). **GET** The query string is appended to the URL, separated by a question mark (?). **Note:** This means that the query string appears in the address bar in the browser. The maximum length of the query string is 4 kilobytes (KB). This is the **default method**. **POST** The data is sent to the server in the HTTP body, which means that it does not appear in the browser address bar. The data is not buffered in an HTTP cache. There is no limit to the length of the query string.
ACTION="execution"	Specifies the program that is to be executed on the server.

To specify the destination statically in a BSP application (that is, the next BSP), assign the name of this next BSP to the *action* attribute in the HTML form. The SAP Web Application Server will process the next page whenever the user sends the form to the server using a Send button. The query string will either be appended to the URL or sent in the HTTP body, depending on the method. You can work with several forms on one HTML page to enable the user to navigate to different next pages. Forms cannot be nested.

Alternatively, you can navigate to the next BSP using a hyperlink. To do this, the attribute *href* must contain the name of the next BSP. Whenever the user chooses this link, the system will take him or her to this same target page. If you want to pass form data to the next BSP using a hyperlink, you must implement this data transfer in a JavaScript function because the query string is not automatically constructed when a hyperlink is selected.



```
<html>
 <body>
  <a href="http://www.sap.com">
            Linktext </a>
  <a href="bsp.htm"> Linktext </a>
 </body>
</html>
```

```
<html>
 <form action ="bsp.htm" method="get">
 ...
 <input type="submit"
    value="Click here" name="BUTTON">
 ...
 </form>
</html>
```

Figure 15: Specifying the Next Page Statically

Specifying the Next Page Dynamically

Processing the OnInputProcessing Event Handler

If you want the next page to be dependent on the consistency or values of the input data, this data must be analyzed before navigation continues. In the BSP programming model (without the implementation of the Model View Controller design pattern) this is realized by passing the data to the BSP, which was used to process the actual HTML page (containing the form). The event handler *OnInputProcessing* is available to analyze the data and navigate to the next page.

This raises the question of how processing of the *OnInputProcessing* event is to be triggered.

For this to happen, the name **OnInputProcessing** must appear in the query string.

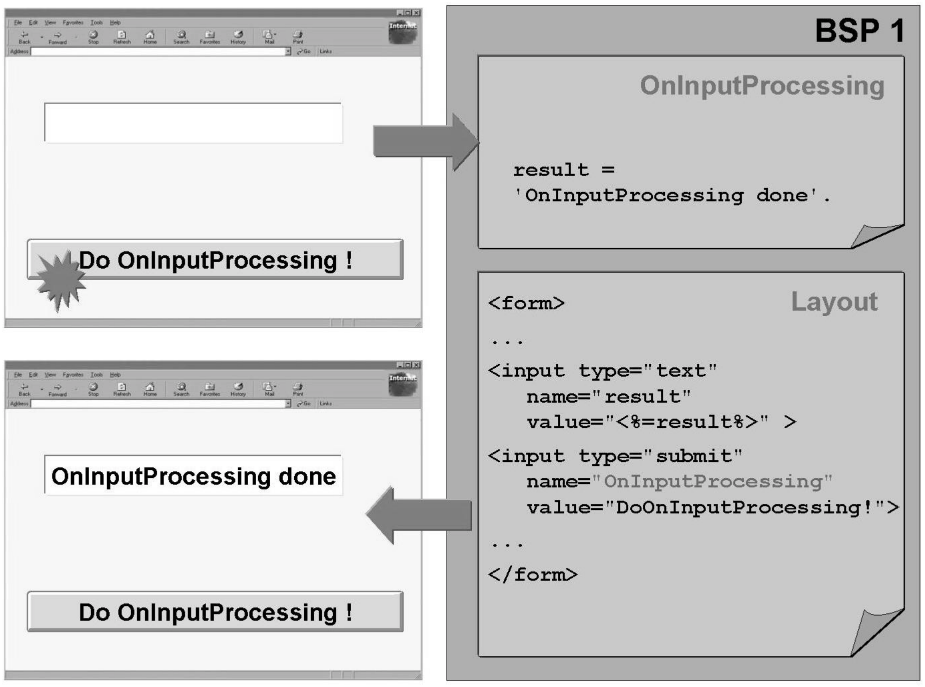
If you use a Send button, you assign this value to the attribute *name*. The *action* attribute can be deleted in the form tag, since the HTTP response is sent to the same BSP on which the current HTML page is based.

```
<form>
...
  <input type="submit"
         name="OnInputProcessing"
```

```
        value="Label">
...
</form>
```

If you use a hyperlink, the name *OnInputProcessing* must be added to the query string. Because the request is passed to the last page to be processed, you can omit the URL completely.

```
<a href="?OnInputProcessing">
```



```
result =
'OnInputProcessing done'.
```

```
<form>
...
<input type="text"
   name="result"
   value="<%=result%>" >
<input type="submit"
   name="OnInputProcessing"
   value="DoOnInputProcessing!">
...
</form>
```

Figure 16: Processing OnInputProcessing

Case distinction using *EVENT_ID*

To enable the user to trigger different actions within one HTML form, you need to provide a different Send button for each possible action. You must assign the name *OnInputProcessing* to each of these Send buttons, so that the OnInputProcessing event handler will be triggered when the button is chosen. The text **<label>** is assigned to the value attribute of a send button in order to be displayed on the button. In this way, the name/value pair *OnInputProcessing=<label>* appears for each query string.:

```
<input type="submit"
       name="OnInputProcessing"
       value="To_Page_1">
```

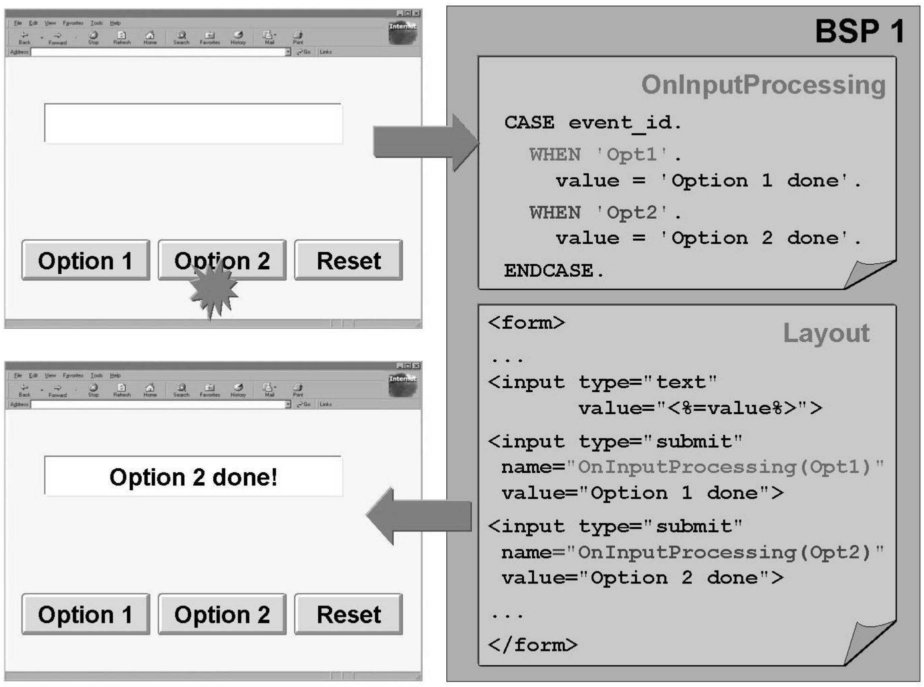
A case distinction in the event handler *OnInputProcessing* can now be made because the value following the name *OnInputProcessing* in the query string is automatically passed to an attribute with the name **EVENT_ID**. This attribute is created automatically in the BSP environment. Depending on the button, another text may also be passed to this attribute. Case distinction is possible with the help of a `CASE event_id. ... ENDCASE.` structure.

Hint: The value of *EVENT_ID* is case-sensitive.

It is impractical to use the label of a button for case distinction, since it may contain special characters and is usually language-specific. This is why you have the option of sending a unique text fragment with the name *OnInputProcessing* and the Send button label. This text fragment must be appended to the *OnInputProcessing* name in parentheses:

```
<input type="submit"
       name="OnInputProcessing(Opt1)"
       value="To_Page_1">
```

There is now a name/value triplet for the Send button in the query string. In such cases, the text fragment, not the label, is passed to the *EVENT_ID* attribute.



```
CASE event_id.
  WHEN 'Opt1'.
    value = 'Option 1 done'.
  WHEN 'Opt2'.
    value = 'Option 2 done'.
ENDCASE.
```

```
<form>
...
<input type="text"
       value="<%=value%>">
<input type="submit"
 name="OnInputProcessing(Opt1)"
 value="Option 1 done">
<input type="submit"
 name="OnInputProcessing(Opt2)"
 value="Option 2 done">
...
</form>
```

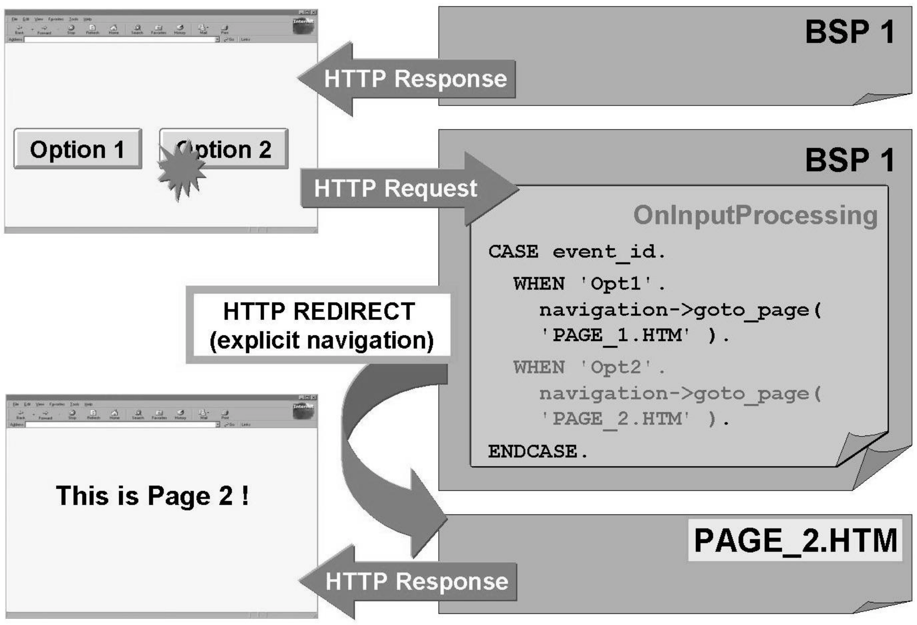
Figure 17: Case Distinction for OnInputProcessing

If you implement dynamic navigation using hyperlinks, the query string must contain the name/value pair **OnInputProcessing=Opt1**. The value following *OnInputProcessing* (here: Opt1) is passed to the attribute *EVENT_ID* if a form is included.

```
<a href="?OnInputProcessing=Opt1">
```

Navigation to the Next Page

Typically, you specify the next page at the OnInputProcessing event. So that you now process the appropriate next page depending on the value of *EVENT_ID*, use the global object, **NAVIGATION**. The interface *IF_BSP_NAVIGATION* provides a template for the global object *NAVIGATION*. Both this interface and the class that implements it, *CL_BSP_NAVIGATION*, are part of the Internet Communication Framework (ICF). Among other things, they offer the methods **NEXT_PAGE** and **GOTO_PAGE** for specifying the next page.



```
CASE event_id.
  WHEN 'Opt1'.
    navigation->goto_page(
    'PAGE_1.HTM' ).
  WHEN 'Opt2'.
    navigation->goto_page(
    'PAGE_2.HTM' ).
ENDCASE.
```

Figure 18: Specifying the Next Page Dynamically

next_page

If you have defined a navigation structure for your application, specify the name of the navigation request as the actual parameter. You define the navigation requests and structure in the BSP application attributes.

```
NEXT_PAGE( 'TO_PAGE2' )
```

goto_page

Specify the name of the next BSP as the actual parameter.

```
GOTO_PAGE( 'PAGE2.HTM' )
```

When you use the method *NEXT_PAGE* or *GOTO_PAGE*, an HTTP redirect is generated; the browser thus generates another HTTP request. This new request then causes the next page specified in the logic to be called (**explicit navigation**). If you do not generate an HTTP redirect, this is known as **implicit navigation**. In such cases, the other events on the current page are processed (beginning with OnInitialization).

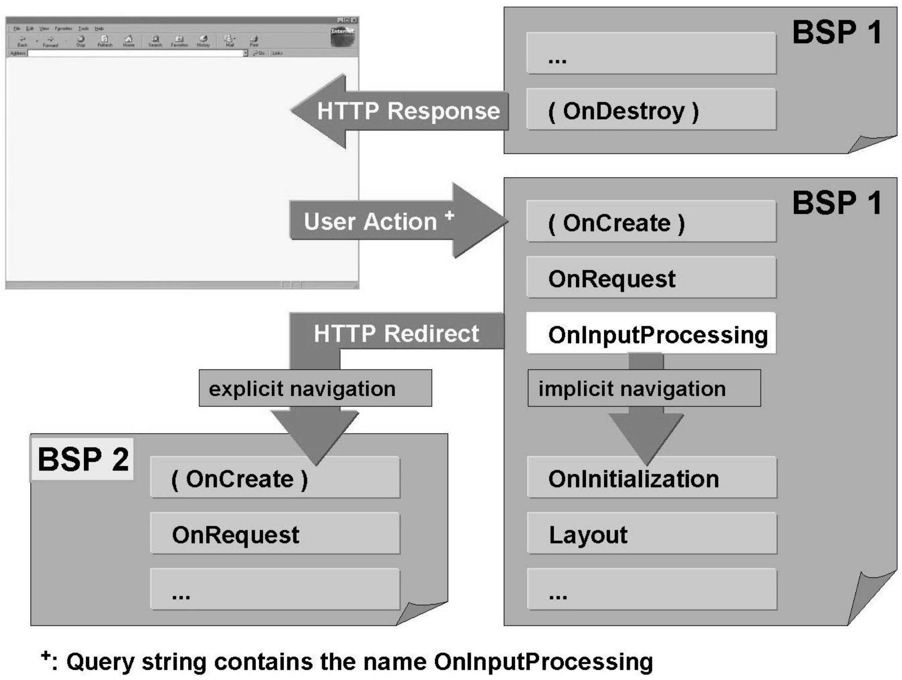
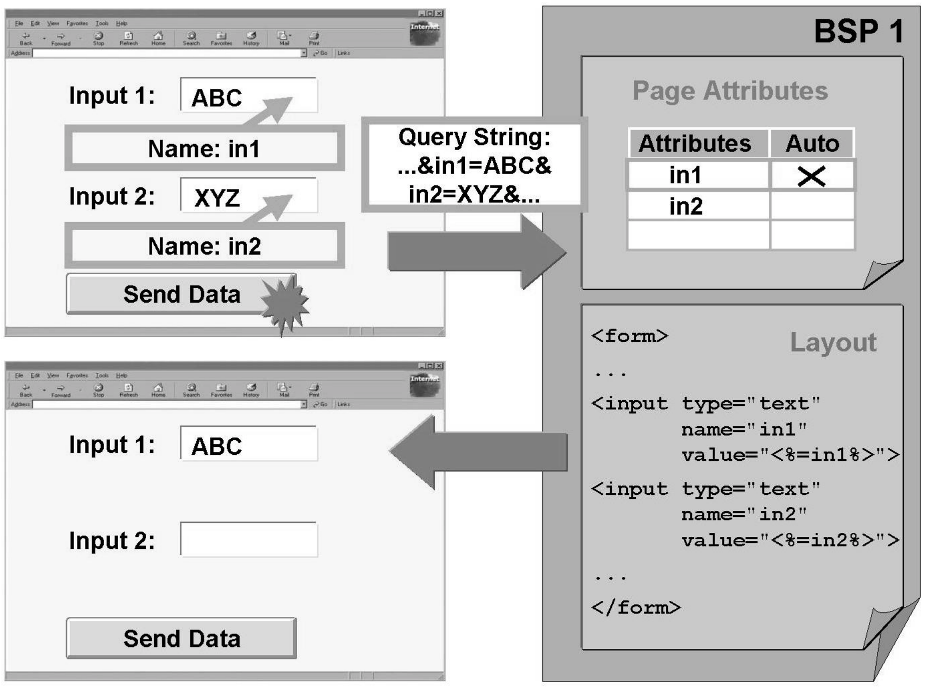


Figure 19: Sequence of Events in Implicit and Explicit Navigation

Data Transfer Between BSPs

Data entered by the user in the input fields in an HTML form is passed to the server as a query string with the HTTP request. For this data to be transferred to the BSP attributes, there must be an **identically-named page attribute** for each form field. This page attribute must also be flagged as an **auto** page attribute – that is, the "Auto" indicator must be set.



```
<form>
...
<input type="text"
       name="in1"
       value="<%=in1%>">
<input type="text"
       name="in2"
       value="<%=in2%>">
...
</form>
```

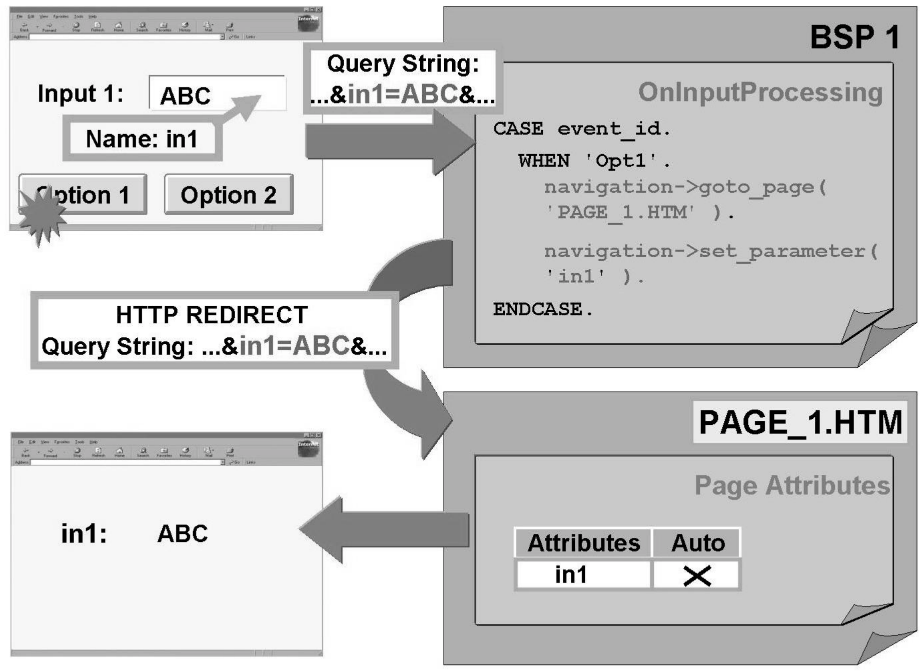
Attributes	Auto
in1	✕
in2	

Figure 20: Getting Data from a Query String

The *Auto* page attributes can be thought of as the data interface of the BSPs. The corresponding query string, which contains the data as name/value pairs, can be generated using either a Send button in an HTML form or a hyperlink.

If you use explicit navigation, it is also possible to transfer the name/value pairs from the HTTP request to the HTTP redirect and thus to pass the user input to the next page. For this you use the method **set_parameter** of the global object *navigation*. When this method is called, you need only specify, as a parameter, the name of the name/value pair that is to be added to the query string in the HTTP redirect. The **set_parameter** method must be executed once for each name/value pair.

```
navigation->set_parameter( name = ’in1’ ).
```

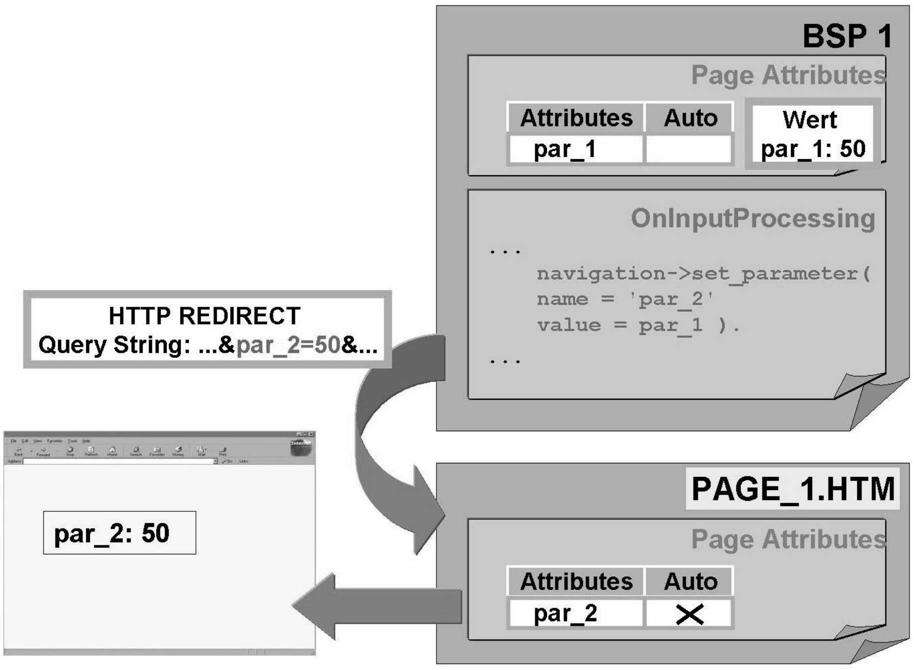


```
CASE event_id.
  WHEN 'Opt1'.
    navigation->goto_page(
    'PAGE_1.HTM' ).

    navigation->set_parameter(
    'in1' ).
ENDCASE.
```

Figure 21: Data Transfer When Using Explicit Navigation

The *set_parameter* method also permits character-like page attributes to be passed between two BSPs. Any data structure can be passed in this way. However, the maximum size of the dataset for the whole URL is 4 KB. It is easy to reach this limit because the query string is appended to the HTTP redirect in an encoded form, which makes the dataset much larger. In such general cases, you must specify the method call, the name, and the value of the name/value pair appended to the query string. If the BSP application is stateful, there are more elegant ways to pass page attributes between two BSPs.

```
navigation->set_parameter( name = 'par_2'
                           value = par_1 ).
```



```
...
    navigation->set_parameter(
    name = 'par_2'
    value = par_1 ).
...
```

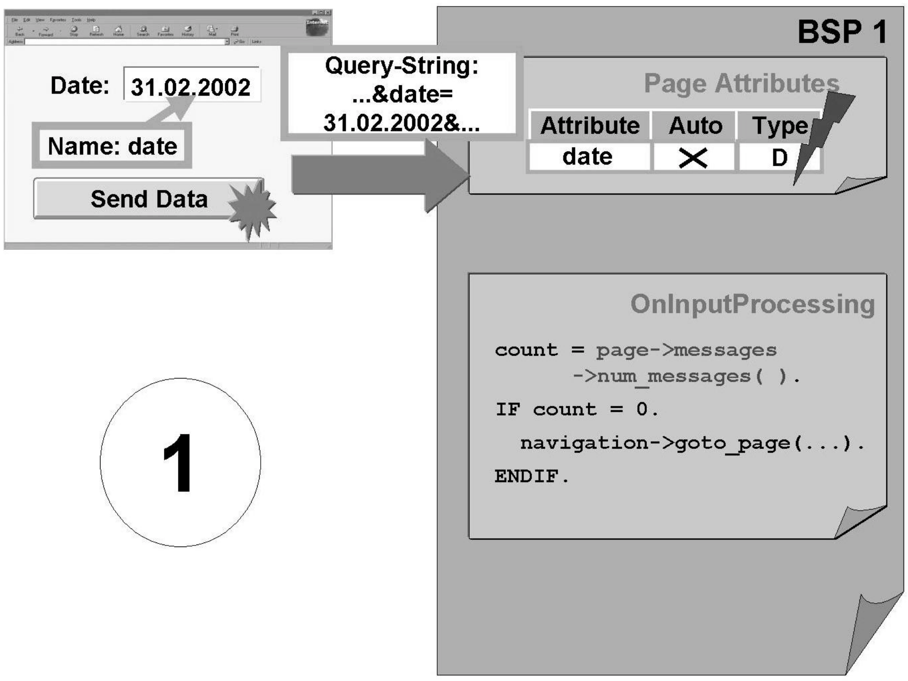
Figure 22: Data Transfer for character-like Page Attributes

Error Handling

The global object *MESSAGES* is provided so that you can handle errors. This global object contains a list of error messages and their severity. It also specifies the input associated with the error message (the condition). Entries in this list can be made either automatically (if a type conflict has been specified with the appropriate *Auto* page attribute by the system) or as part of the program logic. In the latter case, the error text, severity, and condition can be specified in the program. Errors should always be processed on the page where they occurred. The user should be able to navigate to the next page only if no errors occur.

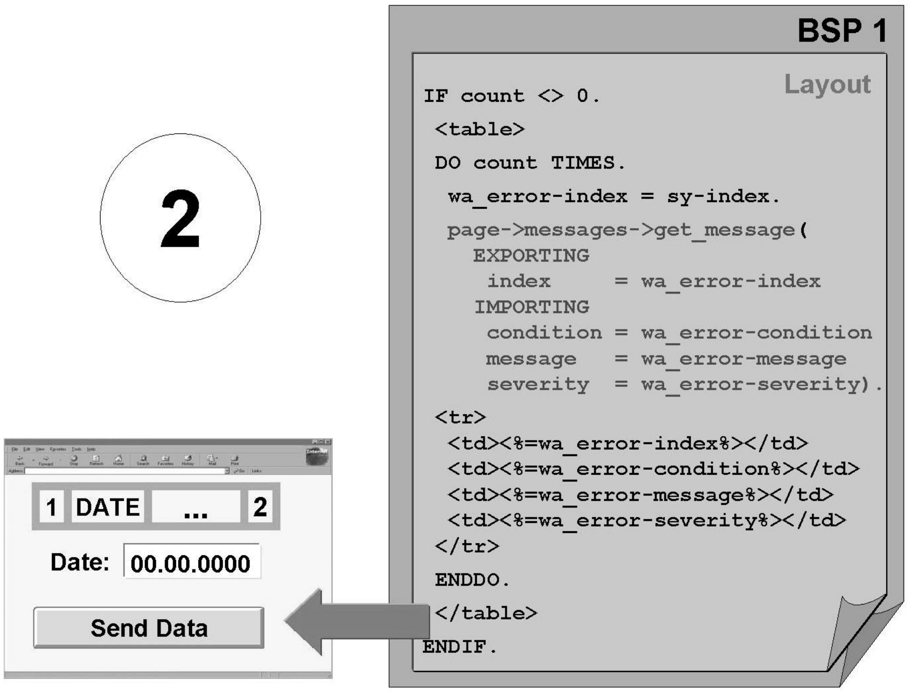
Hint: For page attributes of **type D**, the data is expected in the format specified in the user's default values and automatically converted to the internal format, YYYYMMDD.

For page attributes of **type T**, the data is expected in the format specified in the format HH:MM:SS and automatically converted to the internal format, HHMMSS.



```
count = page->messages
      ->num_messages( ).
IF count = 0.
  navigation->goto_page(...).
ENDIF.
```

Figure 23: Automatic Error Recognition due to a Type Conflict (1)

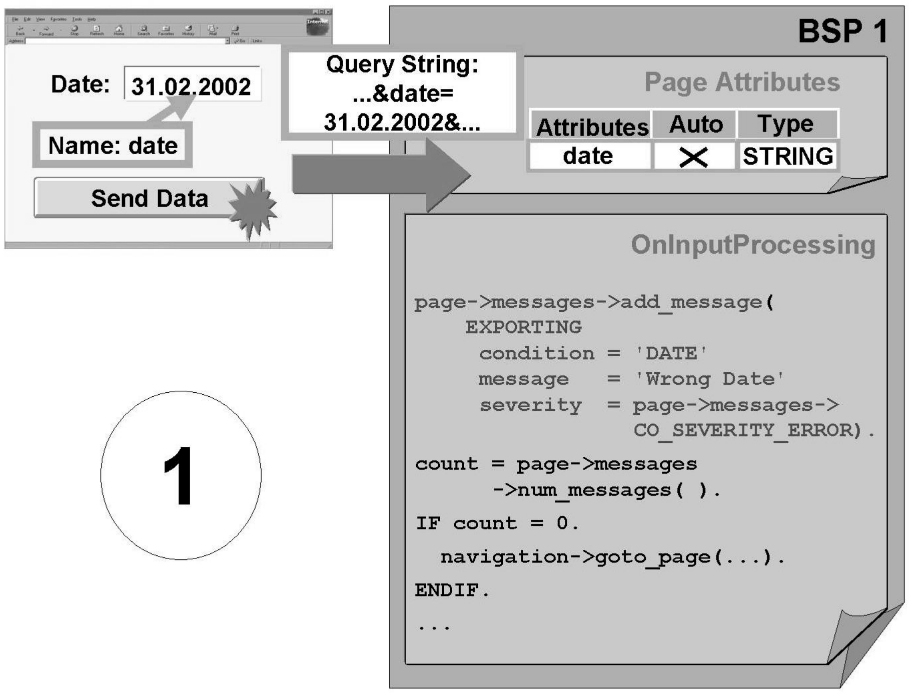


```
IF count <> 0.
 <table>
 DO count TIMES.
  wa_error-index = sy-index.
  page->messages->get_message(
    EXPORTING
     index     = wa_error-index
    IMPORTING
     condition = wa_error-condition
     message   = wa_error-message
     severity  = wa_error-severity).
 <tr>
  <td><%=wa_error-index%></td>
  <td><%=wa_error-condition%></td>
  <td><%=wa_error-message%></td>
  <td><%=wa_error-severity%></td>
 </tr>
 ENDDO.
 </table>
ENDIF.
```

Figure 24: Automatic Error Recognition due to a Type Conflict (2)

The disadvantage of having the type conflict recognized by the system is that the data is not automatically passed to the page attributes, and thus cannot be displayed again. However, in this case you can use the **GET_FORM_FIELD** method of the object **REQUEST** to pass the name of a name/value pair to an attribute of the type STRING. The **GET_FORM_FIELDS** method allows you to split the query string into name/value pairs.

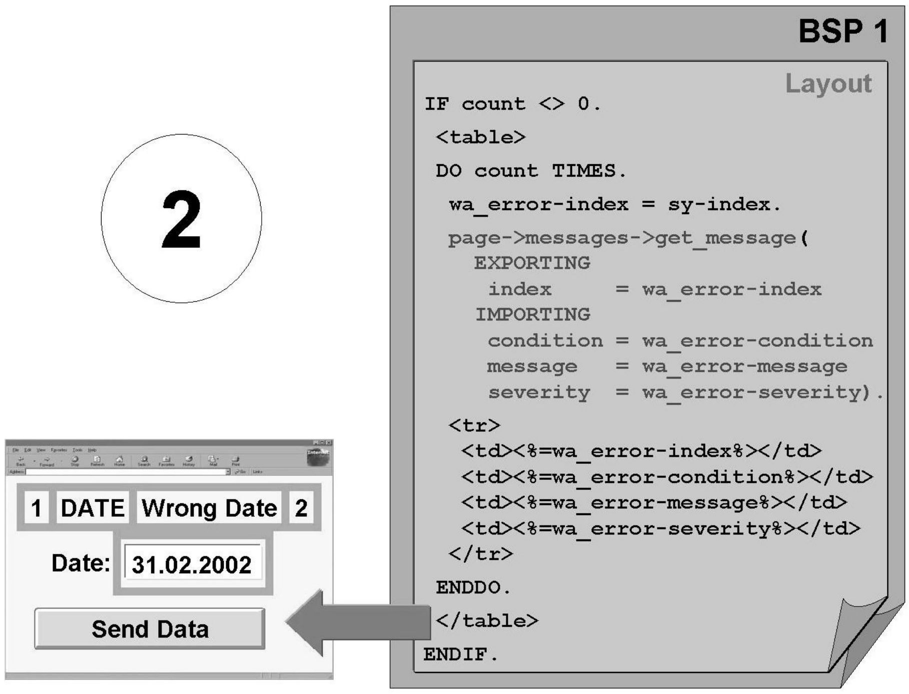
Alternatively, you can select all auto page attributes of the type STRING. In this case however, all type checks must be executed by the program logic.



Attributes	Auto	Type
date	X	STRING

```
page->messages->add_message(
    EXPORTING
      condition = 'DATE'
      message   = 'Wrong Date'
      severity  = page->messages->
                  CO_SEVERITY_ERROR).
count = page->messages
      ->num_messages( ).
IF count = 0.
  navigation->goto_page(...).
ENDIF.
...
```

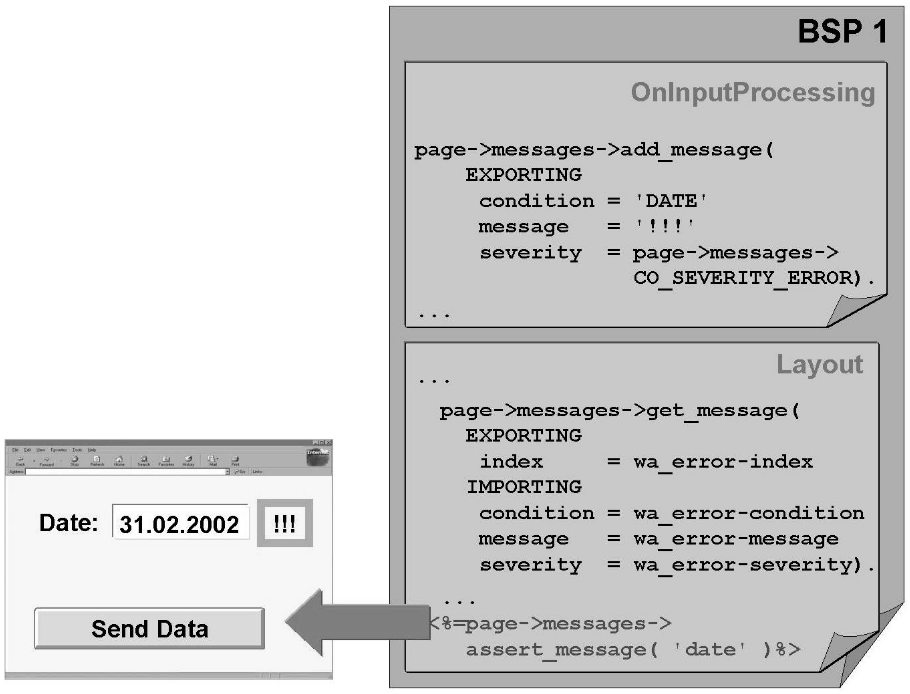
Figure 25: Error Recognition Using the Program Logic (1)



```
IF count <> 0.
 <table>
 DO count TIMES.
  wa_error-index = sy-index.
  page->messages->get_message(
    EXPORTING
     index     = wa_error-index
    IMPORTING
     condition = wa_error-condition
     message   = wa_error-message
     severity  = wa_error-severity).
  <tr>
   <td><%=wa_error-index%></td>
   <td><%=wa_error-condition%></td>
   <td><%=wa_error-message%></td>
   <td><%=wa_error-severity%></td>
  </tr>
 ENDDO.
 </table>
ENDIF.
```

Figure 26: Error Recognition Using the Program Logic (2)

Finally, an error message can be placed beside the field where the input error occurred using the *ASSERT_MESSAGE* method of the global object MESSAGES. To do this, you must pass the field name as an attribute of the ASSERT_MESSAGE method.



```
page->messages->add_message(
    EXPORTING
     condition = 'DATE'
     message   = '!!!'
     severity  = page->messages->
                 CO_SEVERITY_ERROR).
...
```

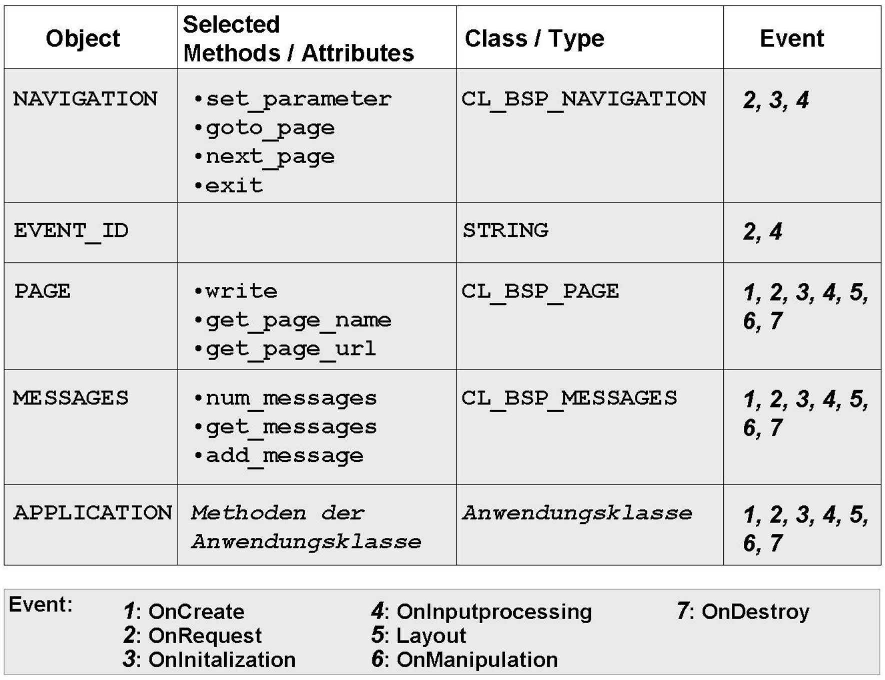
```
...
  page->messages->get_message(
    EXPORTING
     index     = wa_error-index
    IMPORTING
     condition = wa_error-condition
     message   = wa_error-message
     severity  = wa_error-severity).
  ...
<%=page->messages->
   assert_message( 'date' )%>
```

Figure 27: Context-Sensitive Display of Error Messages

Possible runtime errors should be caught by corresponding `TRY. ... ENDTRY.` statement blocks. However, it is also possible to assign an error page in the case of non-explicitly caught runtime errors. This is done by making an entry in the relevant field in the attributes of a BSP. The error page is selected from a dropdown list box. Here, all pages of the same BSP application are displayed for which the attribute **Is Error Page** is set. In the case of a runtime error, the error page is processed directly (no HTTP redirect) and the corresponding HTML page is sent to the client.

Global Objects

The following table summarizes some important global objects and selected methods. You cannot always access every object at every BSP event.



Object	Selected Methods / Attributes	Class / Type	Event
NAVIGATION	•set_parameter •goto_page •next_page •exit	CL_BSP_NAVIGATION	*2, 3, 4*
EVENT_ID		STRING	*2, 4*
PAGE	•write •get_page_name •get_page_url	CL_BSP_PAGE	*1, 2, 3, 4, 5, 6, 7*
MESSAGES	•num_messages •get_messages •add_message	CL_BSP_MESSAGES	*1, 2, 3, 4, 5, 6, 7*
APPLICATION	*Methoden der Anwendungsklasse*	*Anwendungsklasse*	*1, 2, 3, 4, 5, 6, 7*

Event: *1*: OnCreate *4*: OnInputprocessing *7*: OnDestroy
2: OnRequest *5*: Layout
3: OnInitalization *6*: OnManipulation

Figure 28: Selected Global Objects and Event Handlers

Hint: For detailed information on all global objects, see the online documentation.

You use the *NAVIGATION* global object to control the sequence of the BSPs and to set values for the next page.

You use the *EVENT_ID* global object to implement a case distinction for several pushbuttons.

You use the global object *PAGE* and the *write* method to control the formatting of the output on the HTML page.

```
<% page->write( value = wa-fldate ). %>
```

You use the global object *MESSAGE* to implement pages for error handling.

You can use the global object *APPLICATION* to access the methods of the application class associated with your BSP application. You store the business logic of your application in these methods. If the BSP application is executed as stateful, the attributes of the application class can be used for exchanging data between two BSPs.

Exercise 3: Enabling and Processing User Input and Navigating to the Next Page

Exercise Objectives

After completing this exercise, you will be able to:

- Enable users to enter information
- Process user input
- Specify the next page dynamically
- Implement data transfer between BSPs

Business Example

You will implement the first three BSPs in the self-service application for booking flights. The user selects a flight from the first page. On the next page (Flights), the user should see suitable offers in a table. The user can use a link to display the flight details.

Task:

Continue to work with your BSP application or copy the model solution from the last exercise. To do this, copy your BSP application ZNET200_##_02 or the BSP application NET200_S_02, giving it the name ZNET200_##_03, where ## is your group number. Adhere to the names given and always enter single-digit group numbers with a leading zero, 0#. Model solution for this exercise: NET200_S_03.

1. Make it possible for the user to enter the following information on the first page:

   Input fields on the first page

Name	Name of the HTML element
Departure location	depa
Arrival location	dest
Date of departure (Range from to)	day_low, month_low, year_low, day_high, month_high, year_high

   Use the model solution as a basis for the layout of your first page. You should present the departure interval as dropdown boxes for the day, month, and year. You can use the copy template for this purpose. This copy template is stored on the BSP **template_start.htm** in the BSP application **NET200_T_03**. Copy the code segment you have selected

Continued on next page

into the layout of your first page (using Ctrl + C, followed by Ctrl + V). Offer the user a pushbutton for sending the form to the server. Make sure that the OnInputProcessing event handler will be executed.

2. Make sure that the application displays the public/flights.htm BSP as the next page when the user clicks the pushbutton on the first page. On the public/flights.htm BSP (Flight Connections), offer another pushbutton to take the user back to the first page.

   Caution: Provide a way for the user to navigate to the next page dynamically.

3. Make sure that the data entered by the user on the first page is passed to the next page (public/flights.htm).

   Hint: Assign the type STRING to the page attributes that correspond to the form fields. Assign the type **S_CITY** to the `Departure Location` and `Arrival Location` page attributes.

4. Make sure that the application reads the appropriate flight connections from the database, based on what the user enters on the first page. The method used to read these flight connections, **get_flight_list**, is included in the application class CL_NET200S_FINAL, which is already assigned to your application. Implement this method.

   Hint: You can address methods included in the application class using the global object *APPLICATION*.

   Before the application calls the method, it must format the data correctly for the interface. To do this, use the function module **FIT_IN_BAPI_FLCONN_GETLIST**. The departure location, arrival location, and attributes for the range of departure dates are passed to this function module. Test the function module using transaction SE37. Find out about the types of the data structures filled by the function module. Call the function module before you call the method. Create auxiliary variables for the function module's actual parameters *dest_from*, *dest_to*, and *date*. Assign the following types to these auxiliary variables:

dest_from	bapiscodst
dest_to	bapiscodst
it_date	STANDARD TABLE OF bapiscodra

Continued on next page

Pass the function module parameters to the method *get_flight_list*. In addition, pass the travel agency number "00000110" to the method using the *TRAVELAGENCY* parameter. As a result you will get an internal table with the relevant flight connections (changing parameter: *flight_connection_list*). Assign the Data Dictionary structure BAPISCODAT as the table's line type (create the table type first). The internal table data is to be displayed in an HTML table on the BSP public/flights.htm. Display the following columns:

Column	Description
flightconn	Flight connection number
flightdate	Flight date
airportfr	Departure airport
cityfrom	Departure location
deptime	Departure time
airportto	Destination airport
cityto	Destination city
arrtime	Arrival time

5. Create a link from the public/flights.htm BSP to the public/details.htm BSP. Use the flight number as the link. So that your application can display the correct detail list for the flight connection, you must pass the appropriate parameters (name/value pairs) with the link.

Hint: In principle, links are structured as follows:

```
<a href="BSP.HTM?p1=value&p2=value"> text
</a>
```

Hint: The value of the attribute *href* in the hyperlink must not include any line breaks.

Pass the following information to public/details.htm:

Continued on next page

Parameter name	Information passed
travel_ag	’00000110’
connid	Flight connection number (flightconn)
fldate	Flight date (flightdate)

Caution: Do not change the names of these parameters.

6. Use the **PAGE** global object and the **WRITE** method to format the display of the date and time fields on the start and connections pages.

Hint: The flight duration field (column *FLIGHTTIME*) is not a field of type *T*. This means that this column cannot be formatted using the **WRITE** method.

7. Delete the public/details.htm BSP from your application. Copy the BSP public/details.htm from the copy template NET200_T03 to **template_details.htm** in your own application. You may have to change the name of the page fragment *Header.htm* in the include statement if you have used a different name in your solution.

Solution 3: Enabling and Processing User Input and Navigating to the Next Page

Task:

Continue to work with your BSP application or copy the model solution from the last exercise. To do this, copy your BSP application ZNET200_##_02 or the BSP application NET200_S_02, giving it the name ZNET200_##_03, where ## is your group number. Adhere to the names given and always enter single-digit group numbers with a leading zero, 0#. Model solution for this exercise: NET200_S_03.

1. Make it possible for the user to enter the following information on the first page:

Input fields on the first page

Name	Name of the HTML element
Departure location	depa
Arrival location	dest
Date of departure (Range from to)	day_low, month_low, year_low, day_high, month_high, year_high

Use the model solution as a basis for the layout of your first page. You should present the departure interval as dropdown boxes for the day, month, and year. You can use the copy template for this purpose. This copy template is stored on the BSP **template_start.htm** in the BSP application **NET200_T_03**. Copy the code segment you have selected into the layout of your first page (using Ctrl + C, followed by Ctrl + V). Offer the user a pushbutton for sending the form to the server. Make sure that the OnInputProcessing event handler will be executed.

a)

public/start.htm - LAYOUT

```
<!------------------------------------------->
<!-- Begin of Form                        -->
<!------------------------------------------->
    <form>

    <table class=noborder>
      <tr>
```

Continued on next page

```
  <td class=noborder>Departure:</td>
  <td class=noborder colspan=3>
    <input type=text name=depa></td>
</tr>
<tr>
  <td class=noborder>Destination:</td>
  <td class=noborder colspan=3>
    <input type=text name=dest></td>
</tr>
<tr>
  <td class=noborder>Destination date:</td>
  <td class=noborder>
    <% data: counter2(2) type n,
             counter4(4) type n. %>
    <select name="day_low">
      <% do 31 times.
        counter2 = sy-index. %>
        <option VALUE="<%=counter2%>">
          <%=counter2%>
        </option>
      <% enddo. %>
    </select>
    <select name="month_low">
      <% do 12 times.
        counter2 = sy-index. %>
        <option VALUE="<%=counter2%>">
          <%=counter2%>
        </option>
      <% enddo. %>
    </select>
    <select name="year_low">
      <% do 5 times.
        counter4 = sy-index + 2001. %>
        <option VALUE="<%=counter4%>">
          <%=counter4%>
        </option>
      <% enddo. %>
    </select>
  </td>
  <td class=noborder>
    to
  </td>
  <td class=noborder>
    <select name="day_high">
      <% do 31 times.
```

Continued on next page

```
              counter2 = sy-index. %>
              <option VALUE="<%=counter2%>">
                <%=counter2%>
              </option>
            <% enddo. %>
          </select>
          <select name="month_high">
            <% do 12 times.
              counter2 = sy-index. %>
              <option VALUE="<%=counter2%>">
                <%=counter2%>
              </option>
            <% enddo. %>
          </select>
          <select name="year_high">
            <% do 5 times.
              counter4 = sy-index + 2003. %>
              <option VALUE="<%=counter4%>">
                <%=counter4%>
              </option>
            <% enddo. %>
          </select>
        </td>
      </tr>
    </table>

    <br>
    <input type=submit
           name="OnInputProcessing(flights)"
           value="Show flights">

    </form>
```

2. Make sure that the application displays the public/flights.htm BSP as the next page when the user clicks the pushbutton on the first page. On the public/flights.htm BSP (Flight Connections), offer another pushbutton to take the user back to the first page.

   Caution: Provide a way for the user to navigate to the next page dynamically.

   a)

Continued on next page

public/start.htm - OnInputProcessing

```
CASE event_id.
  WHEN ’flights’.
* Navigation to the next page
    navigation->goto_page( ’FLIGHTS.HTM’ ).
ENDCASE.
```

public/flights.htm - OnInputProcessing

```
CASE event_id.
  WHEN ’back’.
* Navigation to the previous page
    navigation->goto_page( ’START.HTM’ ).
ENDCASE.
```

public/flights.htm - LAYOUT

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

<html>

  <head>
    <title> Flight Connections </title>
    <%@ include file = "styles.htm" %>
  </head>

  <body>
<!---------------------------------------------------->
<!-- Page Fragment with the page header             -->
<!---------------------------------------------------->
    <%@ include file="Header.htm" %>

<!---------------------------------------------------->
<!-- Begin of Form                                  -->
<!---------------------------------------------------->
    <form>
```

Continued on next page

```
      <input type=submit
             name="OnInputProcessing(back)"
             value="Back">
    </form>
<!---------------------------------------------------------->
<!-- End of Form                                         -->
<!---------------------------------------------------------->
  </body>

</html>
```

3. Make sure that the data entered by the user on the first page is passed to the next page (public/flights.htm).

   Hint: Assign the type STRING to the page attributes that correspond to the form fields. Assign the type **S_CITY** to the `Departure Location` and `Arrival Location` page attributes.

   a)

   public/start.htm - OnInputProcessing

```
CASE event_id.

  WHEN 'flights'.
* Setting attributes for the next page (Form fields entries)
    navigation->set_parameter( name = 'depa' ).
    navigation->set_parameter( name = 'dest'  ).
    navigation->set_parameter( name = 'day_low' ).
    navigation->set_parameter( name = 'month_low' ).
    navigation->set_parameter( name = 'year_low' ).
    navigation->set_parameter( name = 'day_high' ).
    navigation->set_parameter( name = 'month_high' ).
    navigation->set_parameter( name = 'year_high' ).
* Navigation to the next page
    navigation->goto_page( 'FLIGHTS.HTM' ).

ENDCASE.
```

Continued on next page

public/flights.htm: PAGE ATTRIBUTES

Attribute	Auto	Type	Reference Type	Description
day_high	X	TYPE	STRING	To (Day)
day_low	X	TYPE	STRING	From (Day)
depa	X	TYPE	S_CITY	Arrival location
dest	X	TYPE	S_CITY	Departure location
month_high	X	TYPE	STRING	To (Month)
month_low	X	TYPE	STRING	From (Month)
year_high	X	TYPE	STRING	To (Year)
year_low	X	TYPE	STRING	From (Year)

4. Make sure that the application reads the appropriate flight connections from the database, based on what the user enters on the first page. The method used to read these flight connections, **get_flight_list**, is included in the application class CL_NET200S_FINAL, which is already assigned to your application. Implement this method.

   Hint: You can address methods included in the application class using the global object *APPLICATION*.

   Before the application calls the method, it must format the data correctly for the interface. To do this, use the function module **FIT_IN_BAPI_FLCONN_GETLIST**. The departure location, arrival location, and attributes for the range of departure dates are passed to this function module. Test the function module using transaction SE37. Find out about the types of the data structures filled by the function module. Call the function module before you call the method. Create auxiliary variables for the function module's actual parameters *dest_from*, *dest_to*, and *date*. Assign the following types to these auxiliary variables:

dest_from	bapiscodst
dest_to	bapiscodst
it_date	STANDARD TABLE OF bapiscodra

Continued on next page

Pass the function module parameters to the method *get_flight_list*. In addition, pass the travel agency number "00000110" to the method using the *TRAVELAGENCY* parameter. As a result you will get an internal table with the relevant flight connections (changing parameter: *flight_connection_list*). Assign the Data Dictionary structure BAPISCODAT as the table's line type (create the table type first). The internal table data is to be displayed in an HTML table on the BSP public/flights.htm. Display the following columns:

Column	**Description**
flightconn	Flight connection number
flightdate	Flight date
airportfr	Departure airport
cityfrom	Departure location
deptime	Departure time
airportto	Destination airport
cityto	Destination city
arrtime	Arrival time

a)

public/flights.htm: TYPE DEFINITION

```
TYPES:
tab_bapiscodat type standard table of BAPISCODAT.
```

public/flights.htm: PAGE ATTRIBUTES

Attribute	Auto	Type	Reference Type	Description
it_con_dat		TYPE	TAB_BAPISCO-DAT	Internal table for flight connection data

public/flights.htm - OnInitialization

Continued on next page

```
* event handler for data retrieval

************************************************
* Prepare the form fields for the BAPI Interface
************************************************
DATA: dest_from TYPE bapiscodst,
      dest_to   TYPE bapiscodst,
      it_date   TYPE TABLE OF bapiscodra.

*****************************************
* Data Conversion for Method Call
*****************************************
CALL FUNCTION 'FIT_IN_BAPI_FLCONN_GETLIST'
  EXPORTING
    start      = depa
    end        = dest
    day_low    = day_low
    day_high   = day_high
    month_low  = month_low
    month_high = month_high
    year_low   = year_low
    year_high  = year_high
  IMPORTING
    dest_from  = dest_from
    dest_to    = dest_to
  TABLES
    date       = it_date.

****************************************.
* BAPI Call via Application class method
****************************************
CALL METHOD application->get_flight_list
  EXPORTING
    travelagency           = '00000110'
*   AIRLINE                =
    destination_from       = dest_from
    destination_to         = dest_to
*   MAX_ROWS               =
  CHANGING
    date_range             = it_date
    flight_connection_list = it_con_dat.
```

public/flights.htm - LAYOUT

Continued on next page

```
...
<body>

<!---------------------------------------------------->
<!-- Page Fragment with the page header            -->
<!---------------------------------------------------->
    <%@ include file="Header.htm" %>

<!---------------------------------------------------->
<!-- Begin of Form                                 -->
<!---------------------------------------------------->
    <form>
      <input type=submit
             name="OnInputProcessing(back)"
             value="Back">
    </form>
<!---------------------------------------------------->
<!-- End of Form                                   -->
<!---------------------------------------------------->

<!---------------------------------------------------->
<!-- Begin of Flight Table                         -->
<!---------------------------------------------------->
    <table>
      <thead>
        <tr>
          <td colspan="2">
          </td>
          <td colspan="3">
            Departure Info
          </td>
          <td colspan="3">
            Arrival Info
          </td>
        </tr><tr>
          <td>Connection</td>
          <td>Date</td>
          <td>Airport</td>
          <td>Departure</td>
          <td>Departure Time</td>
          <td>Airport</td>
          <td>Destination</td>
          <td>Arrival Time</td>
        </tr>
```

Continued on next page

```
      </thead>

      <tbody>
<!----------------------------------------------------->
<!--  Scripting                                      -->
<!----------------------------------------------------->
        <% data: wa_con_dat type BAPISCODAT.
        loop at it_con_dat into wa_con_dat. %>
          <tr>
            <td><%= wa_con_dat-flightconn %></td>
            <td><%= wa_con_dat-flightdate %></td>
            <td><%= wa_con_dat-airportfr %></td>
            <td><%= wa_con_dat-cityfrom %></td>
            <td><%= wa_con_dat-deptime %></td>
            <td><%= wa_con_dat-airportto %></td>
            <td><%= wa_con_dat-cityto %></td>
            <td><%= wa_con_dat-arrtime %></td>
          </tr>
        <% endloop. %>
      </tbody>

    </table>
<!----------------------------------------------------->
<!-- End of Flight Table                             -->
<!----------------------------------------------------->

  </body>
```

5. Create a link from the public/flights.htm BSP to the public/details.htm BSP. Use the flight number as the link. So that your application can display the correct detail list for the flight connection, you must pass the appropriate parameters (name/value pairs) with the link.

   Hint: In principle, links are structured as follows:

   `<a href="BSP.HTM?p1=`*`value`*`&p2=`*`value`*`"> text </a>`

   Hint: The value of the attribute *href* in the hyperlink must not include any line breaks.

   Pass the following information to public/details.htm:

Continued on next page

Parameter name	Information passed
travel_ag	’00000110’
connid	Flight connection number (flightconn)
fldate	Flight date (flightdate)

Caution: Do not change the names of these parameters.

a)

public/flights.htm: LAYOUT

```
<!----------------------------------------------------->
<!--  Scripting                                      -->
<!----------------------------------------------------->
  <% data: wa_con_dat type BAPISCODAT.
  loop at it_con_dat into wa_con_dat. %>
    <tr>
      <td>
<!-- The value of the href attribute has to be in a   -->
<!-- single line; line breaks are added here for      -->
<!-- technical reasons                                -->
        <a href="Details.htm?
                 travel_ag=00000110&
                 connid=<%=wa_con_dat-flightconn%>&
                 fldate=<%=wa_con_dat-flightdate%>">
          <%= wa_con_dat-flightconn %></a>
      </td>
      ...
    </tr>
  <% endloop. %>
```

6. Use the **PAGE** global object and the **WRITE** method to format the display of the date and time fields on the start and connections pages.

Hint: The flight duration field (column *FLIGHTTIME*) is not a field of type *T*. This means that this column cannot be formatted using the **WRITE** method.

Continued on next page

a)

public/flights.htm - LAYOUT

```
<!------------------------------------------------->
<!--  Scripting                                 -->
<!------------------------------------------------->
...
<% loop at it_con_dat into wa_con_dat. %>
  ...
  <td> <% page->write(
          value = wa_con_dat-flightdate ). %>
  </td>
  ...
  <td> <% page->write(
          value = wa_con_dat-deptime ). %>
  </td>
  ...
  <td> <% page->write(
          value = wa_con_dat-arrtime ). %>
  </td>
  ...
<%  endloop. %>
```

public/start.htm - LAYOUT

```
<% Loop at it_last_minute into wa_last_minute. %>
  ...
  <% page->write(
          value = wa_last_minute-fldate ). %>
  ...
<% Endloop. %>
```

Continued on next page

7. Delete the public/details.htm BSP from your application. Copy the BSP public/details.htm from the copy template NET200_T03 to **template_details.htm** in your own application. You may have to change the name of the page fragment *Header.htm* in the include statement if you have used a different name in your solution.

   a) Delete the public/details.htm BSP. Copy the BSP template_details.htm from the template to the BSP public/details.htm in your application. Check the include names. Activate and test the application.

Lesson Summary

You should now be able to:

- Enable users to enter information
- Define static navigation between BSPs
- Define dynamic navigation between BSPs
- Enable data transfer between BSPs
- React to errors in transmitted data
- Work with global objects

Related Information

- Online documentation on global objects
- Class documentation for the global class CL_BSP_NAVIGATION

Lesson: Session Handling

Lesson Overview

In this training unit, we will introduce and evaluate stateless and stateful programming. We will also explain techniques such as the use of hidden fields and cookies in a BSP application.

Lesson Objectives

After completing this lesson, you will be able to:

- List criteria for deciding to use stateful or stateless programming
- Implement techniques for retaining data in a stateless BSP application

Business Example

You need to consider which parts of a BSP application should be implemented statefully and which should be stateless. If part of an application is stateless, you also need to consider how data that may be needed in the future can be retained on the SAP Web AS and later identified correctly.

Stateful and Stateless BSP Applications

What does stateful programming mean?

Executing a BSP application **statefully** means that the application context is retained after the response and is available when the application continues executing. This, in turn, means that the application context is rolled into the work process if processing is continued. This is the case with classical SAP R/3 transactions at each SAP GUI screen change. In BSP applications, it means that the roll area is available even after a page has been transmitted to the browser for further requests from the same application.

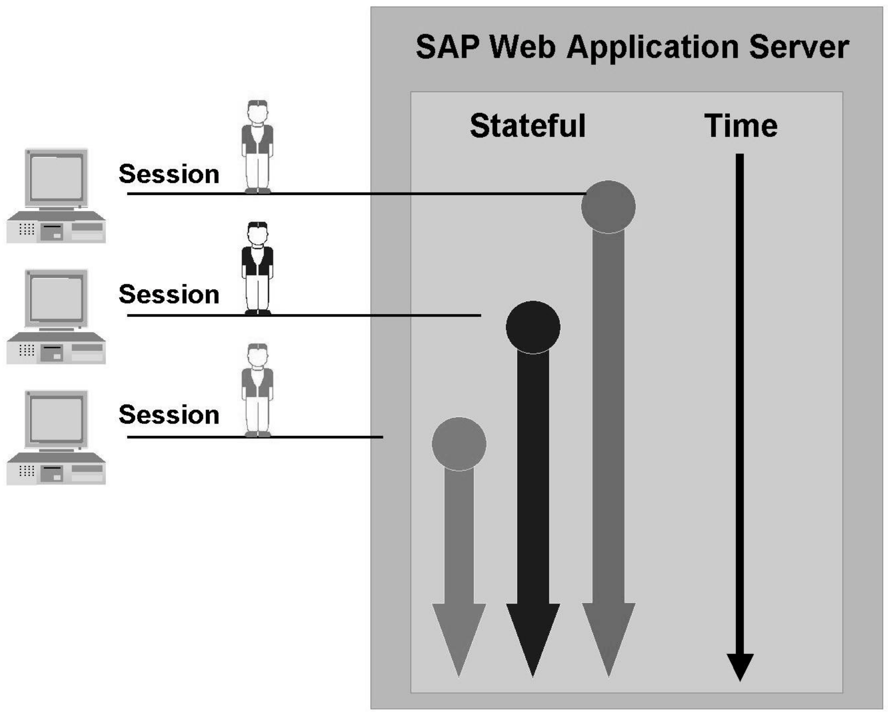


Figure 29: Stateful Programming

- A large load is created on the SAP Web Application Server: Web applications are generally used by **several** Internet and Intranet users. For each application, the context on the SAP Web Application Server is retained.
- Resources are retained for an unnecessarily long time on the SAP Web Application Server. The session is not deleted when the user navigates to another page in the browser. Thus, the program context is retained until a timeout mechanism releases it. Because this can normally take some time, limited resources are blocked longer than necessary.
- Applications are easier to program: Data is available at each new access.

What does stateless programming mean?

Executing a BSP application **statelessly** means that for each new request, a new application context (roll area) is created. In addition when the BSP application continues, the old context is no longer available (stateless execution or execution without memory). After an HTTP request/response cycle, all resources are released.

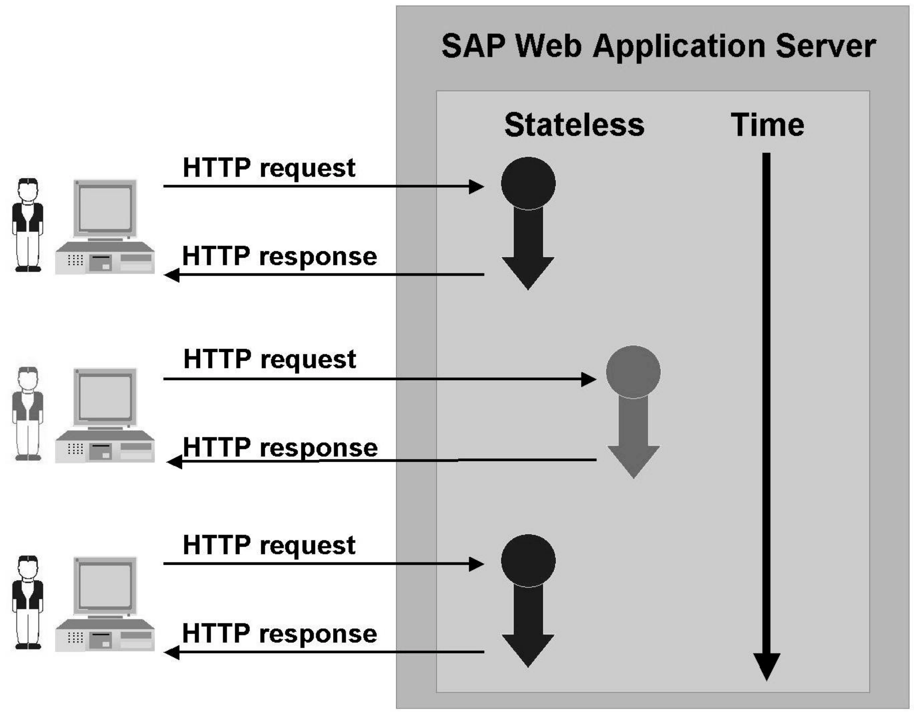


Figure 30: Stateless Programming

- Resources on the SAP Web Application Server are used only during HTTP request processing: Stateless programming usually results in good server scalability and is thus particularly suitable for Web applications (with many users).
- Data must be generated again: Data that is required over several BSPs must be read from the database, often several times in a row. This increases the load on the database and makes the program logic more complex.

Stateless programming poses the question of how data can be retained over several request/response cycles if the data must be available while the rest of the application executes.

Mixed Stateful and Stateless Programming

In actual applications, it is generally desirable to make some parts of a BSP application stateful, and others stateless. For this reason, SAP has made it possible for developers to change a part of the program from stateful to stateless (and vice versa) either statically or dynamically.

Firstly, you can specify statically (by setting a flag on the Properties tab) whether a BSP application should be executed statelessly or statefully. When a BSP in this application is called, you can also specify whether this mode should be retained or whether it should be changed from this point on. In addition, you can change this preset value at any time by setting the *KEEP_CONTEXT* attribute of the global object *RUNTIME*:

`runtime->keep_context = 0.`

After the BSP has been processed, the application is executed **statelessly**.

`runtime->keep_context = 1.`

After the BSP has been processed, the application is executed **statefully**.

Setting this property has the following effect, with regard to the session and the APPLICATION object:

- If the BSP application is executed statelessly, the session and thus (when using an application class) the APPLICATION object is instantiated anew for every HTTP request and then destroyed at the end of the HTTP request/response cycle.
- If the BSP application is executed statefully, the session and thus (when using an application class) the APPLICATION object is retained along with its attributes. A temporary cookie containing the session ID is sent to the Web browser. In order to restart a BSP application, an appropriate name/value pair has to be send in the query string **(sap-sessioncmd=open)**. In this case, the existing session is ended, a new session is started, and the associated session ID is sent to the Web browser using a cookie. The current session can be ended by sending **(sap-sessioncmd=close)** in the query string. Here, the next page is specified in the name/value pair **sap-exitURL=<URL>** in the query string. The BSP application can also be ended in the program logic. To do so, you use the method *exit* belonging to the global object *navigation*. In this case, the next page is passed as a parameter when the method is called:

  ```
  navigation->exit( 'http://www.sap.com' ).
  ```

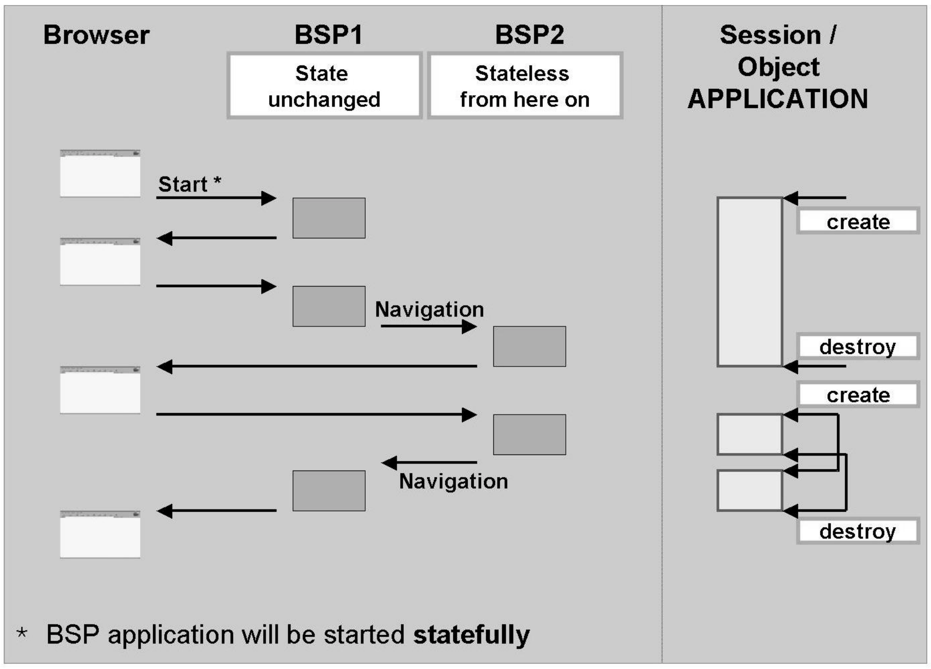


Figure 31: Lifetime of the Session and the Object APPLICATION (1)

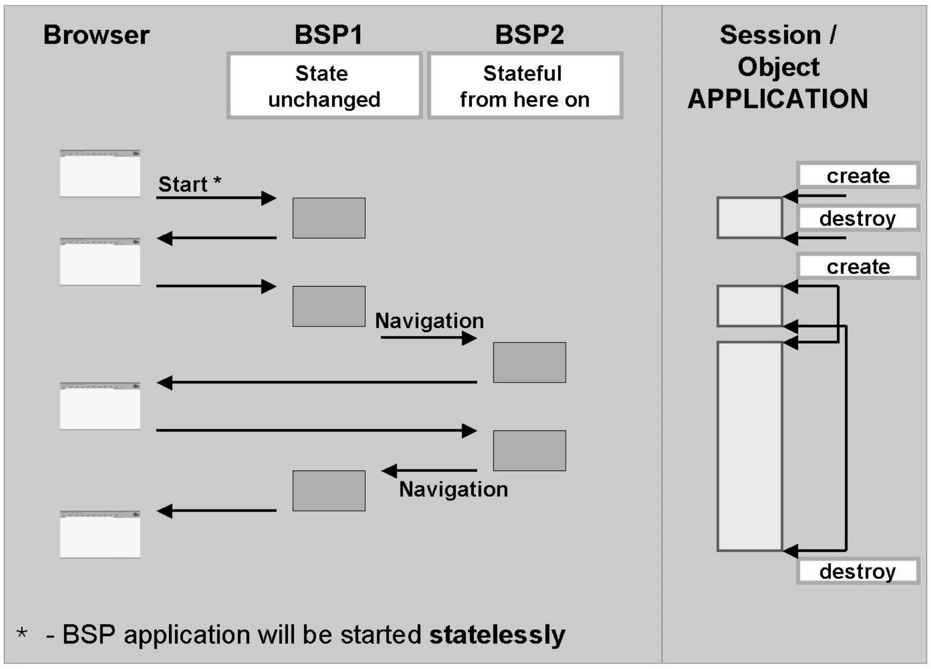


Figure 32: Lifetime of the Session and the Object APPLICATION (2)

If the BSP application is executed statefully, you can also let the individual BSPs (and thus the corresponding page attributes) live longer than a single HTTP request/response cycle. The lifetime of a BSP is defined statically (by choosing the appropriate value from a drop-down box). You have several options:

Lifetime: Session

The PAGE object for this BSP is not deleted until the BSP application is switched to stateless or ended explicitly:

```
navigation->exit( ... ).
```

Lifetime: Page change

The PAGE object for this BSP is not deleted until the BSP application is switched to stateless, ended explicitly, or a page change takes place as a result of explicit navigation:

```
navigation->goto_page( ... ).
```

or

```
navigation->next_page( ... ).
```

Lifetime: Request

The object PAGE for this BSP is deleted after each HTTP request/response cycle.

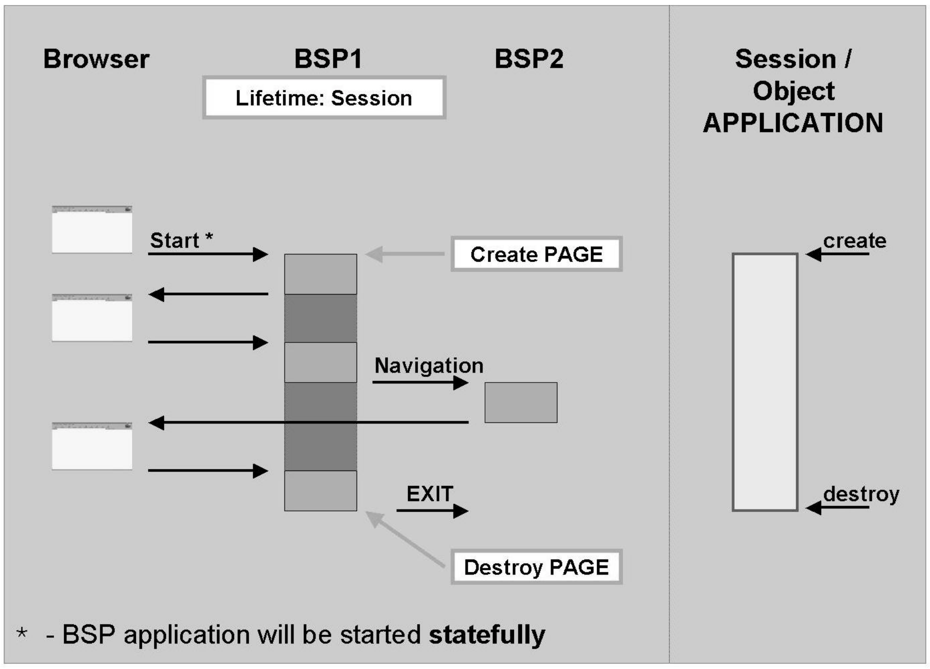


Figure 33: Lifetime of a BSP with the Property *Lifetime: Session*

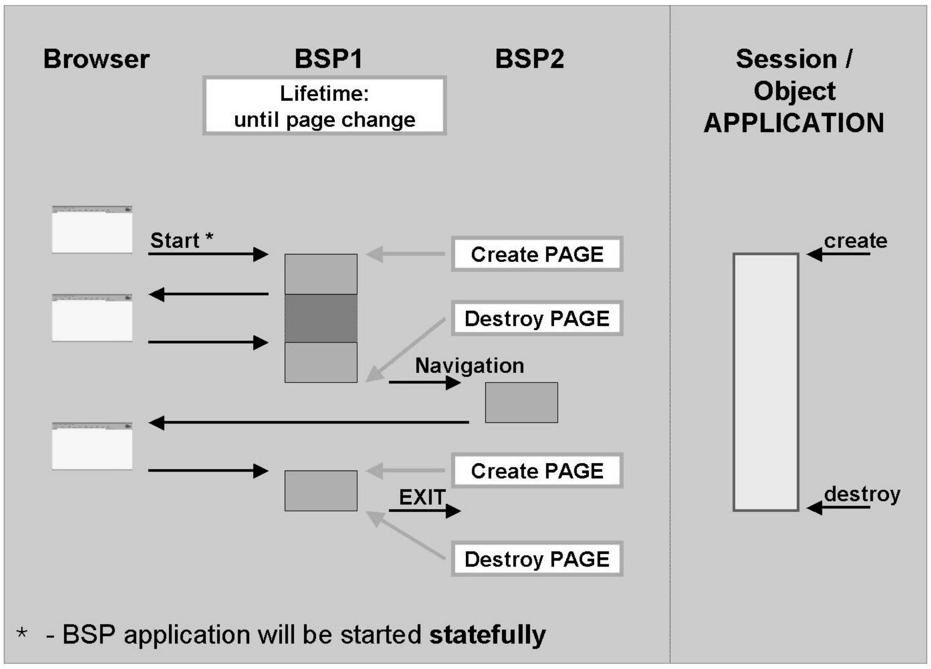

Browser
BSP1
Lifetime: until page change
BSP2
Session / Object APPLICATION
Start *
Create PAGE
Destroy PAGE
Navigation
Create PAGE
EXIT
Destroy PAGE
create
destroy
* - BSP application will be started **statefully**

Figure 34: Lifetime of a BSP with the Property *Lifetime: Until Page Change*

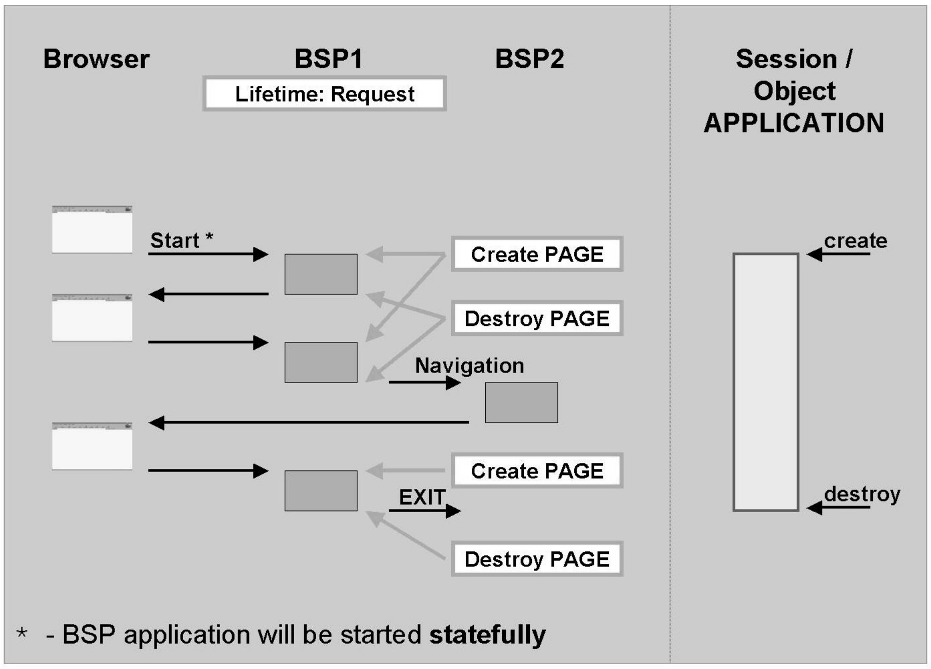
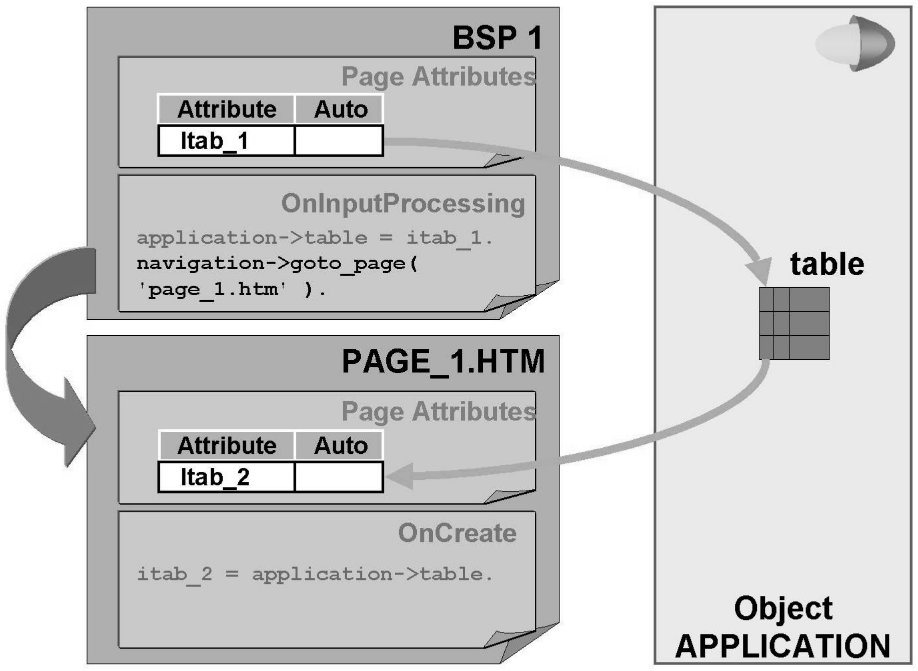


Figure 35: Lifetime of a BSP with the Property *Lifetime: REQUEST*

Data Transfer in Stateful BSP Applications

The visibility of page attributes in a BSP is always restricted to the execution of this BSP. This means that the page attributes of BSP1 cannot be addressed while BSP2 is being processed, regardless of the lifetime of BSP1. For you to be able to process data from BSP1 in BSP2, it must first be passed from BSP1 to BSP2. To do this, you use the method *SET_PARAMETER* of the global object *NAVIGATION*. However, transferring this kind of data is only practicable if it is made up of values from the query string of an HTTP request (such as form data). Even internal tables quickly push this procedure to its limits (because only a limited quantity of data can be transferred, and this data must be character-type).

If the BSP application is being executed statefully at the time of the navigation between it and a second BSP, you can use the attributes of the application class to transfer the data. In this approach, the data from BSP1 is transferred to attributes of the application class. When BSP2 is processed, it accesses these attributes. To use this option, you simply set the BSP application mode to stateful before you leave BSP1 and then restore it to stateless after BSP2 has been processed.



```
application->table = itab_1.
navigation->goto_page(
'page_1.htm' ).
```

```
itab_2 = application->table.
```

Figure 36: Data Transfer in a Stateful BSP Application

Canceling Stateful BSP Applications

Statefully programmed BSP applications should always be exited so that the session is also deleted on the application server. There are different ways of doing this:

The user exits the application using a hyperlink or a button on the HTML page. As a result of the corresponding HTTP request, the BSP is called again and the session is deleted there using `navigation->exit( '..' )`..

If the user closes the browser or enters the URL to the next page in the address line of the browser, this must lead to an HTTP request that also results in the exiting of the session. For this, the browser event **OnUnload** must be caught. The corresponding URL for exiting the session can be created in the source code of the relevant BSP using the following static method: `CL_BSP_LOGIN_APPLICATION=>GET_SESSIONEXIT_URL( page = page ).`

To ensure that the source code for catching the browser event OnUnload is available to the programmer in reusable form (so that the source code does not have to be manually added to layout of each BSP), the HTML frameset can be used. The frameset consists of a single frame. In this frame, the actual BSP application to be displayed is embedded. If the browser is closed or the next page is called by entering a URL in the browser address line, the frameset is also deleted. A corresponding template is available in the BSP application **system**. Name: **session_single_frame.htm**. This page is copied to your BSP application and is then the new start page. After copying, you only need to correct the name of the first BSP to be embedded (`Data` statement in the upper part of the layout).

Stateful BSP Applications Without the Use of Cookies

If a BSP application is started statefully, a temporary cookie, which contains the session ID, is sent by the BSP runtime environment. At the next HTTP request, the cookie is sent back to the creating server by the client. The server can read the session ID and this assign the session to the client.

However, if the use of temporary cookies is not supported by the browser, the session ID must be swapped between the server and the client in a different way. Otherwise, it would not be possible to execute stateful BSP applications. There is therefore a second possibility, with which the session ID is included in the URL in encrypted form. To use this procedure, you must start the BSP application using a URL that contains the query string **sap-syscmd=nocookie**. In this case, the session ID is inserted in the cache key - that is, the parenthetical expression in the URL.

Techniques for Retaining Data in a Stateless BSP Application

The program context of a stateless BSP application is not retained on the SAP Web Application Server beyond a single HTTP request /response cycle. To avoid repeated database reads (which adversely affect performance), you can implement alternative techniques. The following techniques are available:

- Hidden fields for invisible storage of data in a BSP
- Use of cookies (client and server-side).

A form can have **hidden fields** that are not displayed in the browser, but are sent back to the HTTP server as part of the query string when the next HTTP request is sent. This allows the HTTP server to hide information on the HTML page. It then uses this information to identify the user when the next request is sent.

```
<input type="hidden" name="counter" value="5">
```

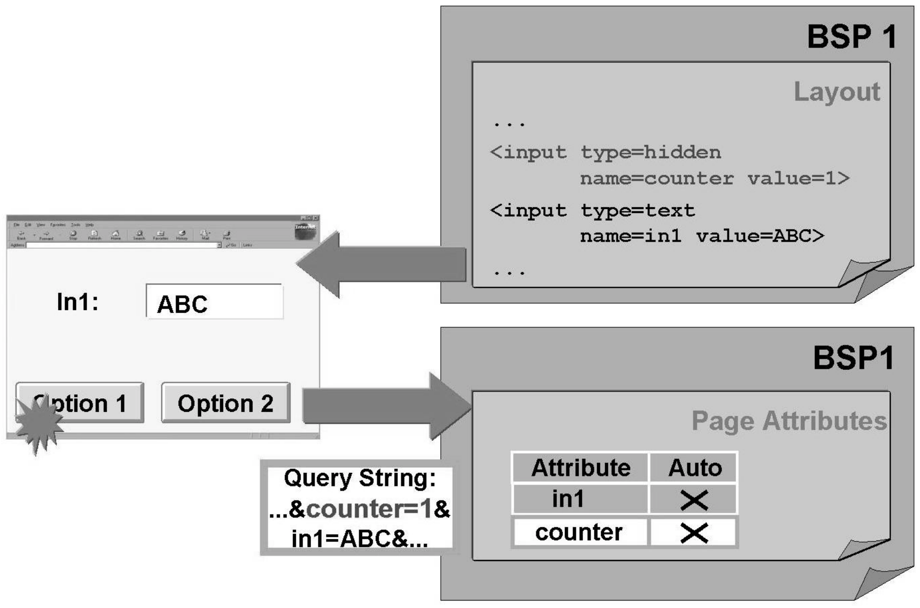


Figure 37: Passing Hidden Fields

The disadvantages of this techniques are:

- The data hidden on the page is stored as a string, that is, not in a defined data type.
- The values of the hidden fields must be elementary or character-type. If the latter, all the fields must be character-type. In particular, you cannot use nested structures or internal tables as values for HIDDEN FIELDS. For example, you cannot use:

  ```
  <input type="hidden"
         name="Table"
         value="<%= itab %>">
  ```

- The hidden information (such as state recognition) is part of the HTML page. Thus, this information is sent to the server only if the user triggers the next HTTP request using an element on the page that contains the hidden fields. For example, if a user fills a basket in an online store, then navigates to another page while the Web application is being processed, and does not use the browser's *Back* to return to the online shop, the items added to the basket are not saved.

Cookies are a general technique that can be implemented by an HTTP server to send and receive information. An HTTP server can send a cookie that is saved by the HTTP client (Web browser). The cookie contains the information to be saved and a description of valid URLs. At the next HTTP client request to an HTTP server whose URL is in the cookie's validity area, the cookie is sent to the server along with the request.

Cookies are used in Web applications to store user-related information and other data. The information in cookies makes it possible to address a user personally each time he or she repeatedly accesses a page or to manage the status of a shopping basket.

Another way of saving information for later in a stateless BSP is to buffer it on the server (using a **server-side cookie**). For clarity, the cookies sent from the HTTP server to the HTTP client are referred to as **client-side cookies** in the following sections.

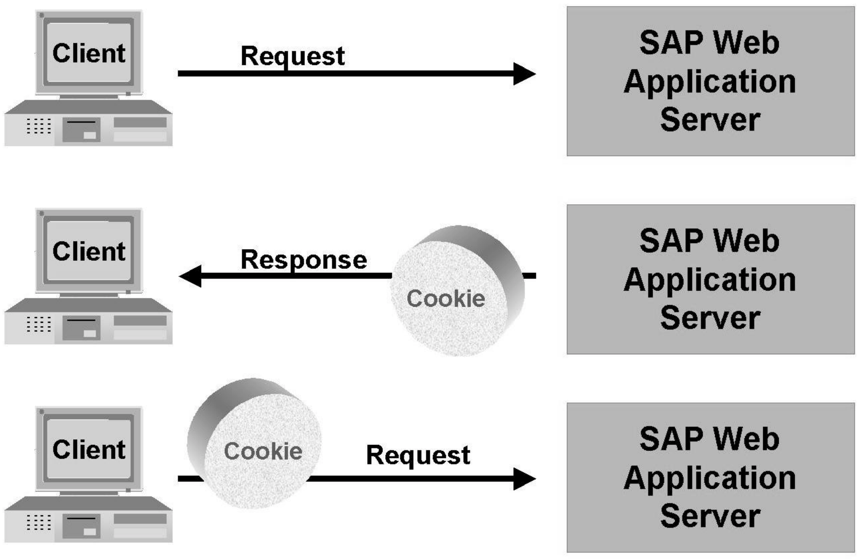

Client
Request
SAP Web Application Server
Client
Response
Cookie
SAP Web Application Server
Client
Cookie
Request
SAP Web Application Server

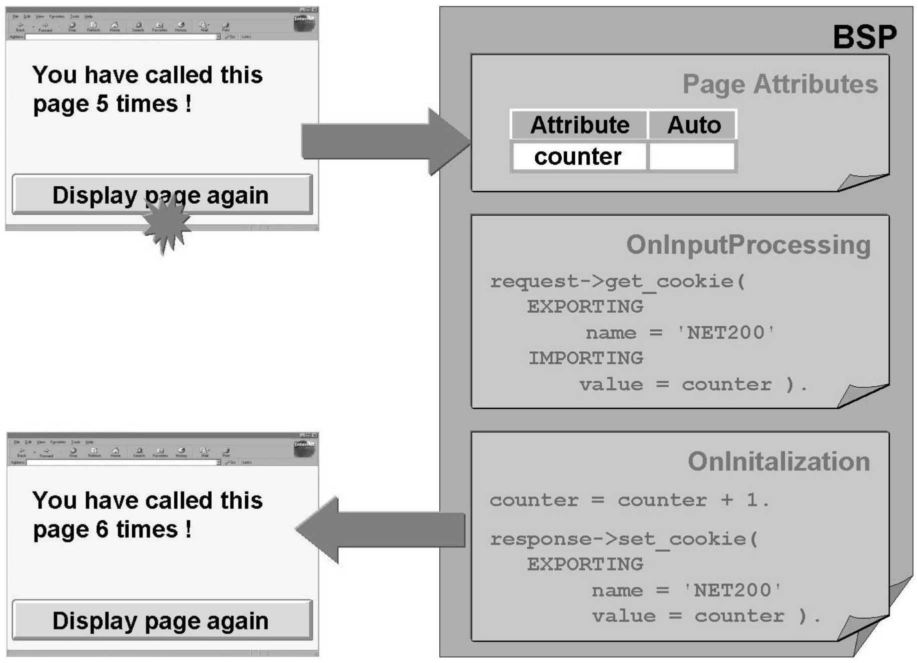
Figure 38: Client-Side Cookies

Client-Side Cookies in a BSP Application

The Internet Communication Framework (ICF) provides methods using the IF_HTTP_ENTITY interface for sending and receiving client-side cookies. This interface is implement in the classes CL_HTTP_REQUEST and CL_HTTP_RESPONSE (among others) from which the global objects *REQUEST* and *RESPONSE* are derived. You use these methods to define client-side cookies and to read cookies sent with the HTTP request.

You can control the lifetime of a client-side cookie. If you do not specify a valid expiration date, the client-side cookie is retained in memory and deleted when the browser is closed (**temporary client-side cookie**). If you wish to store the cookie on the client beyond a browser session, you must provide the cookie with a corresponding expiration date in the SET_COOKIE method. The cookie is then stored on the client hard drive (**persistent client-side cookie**). A cookie whose expiration date has passed will not be sent to the server.

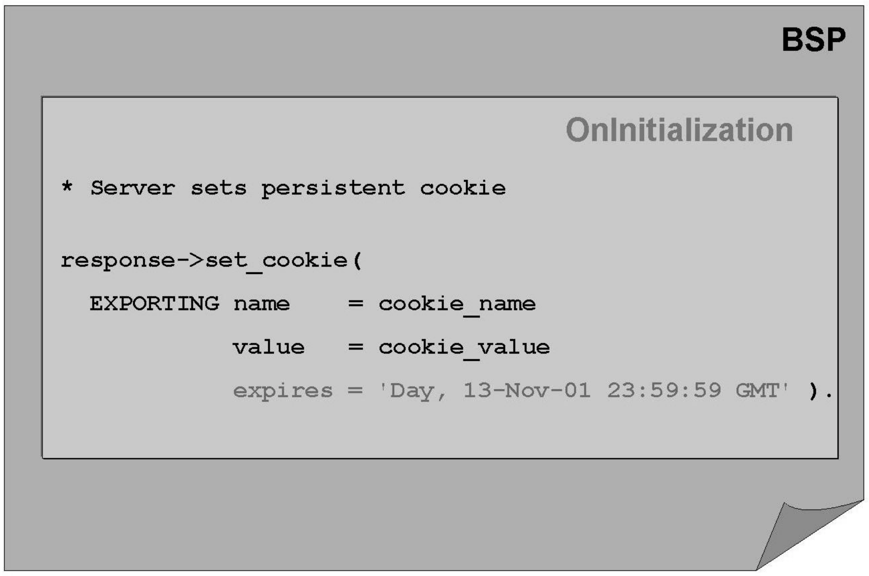
From the SAP Web Application Server, you can delete a client-side cookie. To do so, you use the method *DELETE_COOKIE_AT_CLIENT* belonging to the global object *RESPONSE*.



```
request->get_cookie(
   EXPORTING
        name = 'NET200'
   IMPORTING
       value = counter ).
```

```
counter = counter + 1.

response->set_cookie(
   EXPORTING
         name = 'NET200'
         value = counter ).
```

Figure 39: Setting a Temporary Client-Side Cookie: Example


BSP
OnInitialization

```
* Server sets persistent cookie

response->set_cookie(
  EXPORTING name    = cookie_name
            value   = cookie_value
            expires = 'Day, 13-Nov-01 23:59:59 GMT' ).
```

Figure 40: Setting Persistent Client-Side Cookies

Caution: You must set the expiry date in the format **Day, DD-Mon-YYYY HH:MM:SS GMT**. For more information, refer to the specifications for HTTP cookies.

Hint: In the BSP environment, temporary client-side cookies are used as follows:

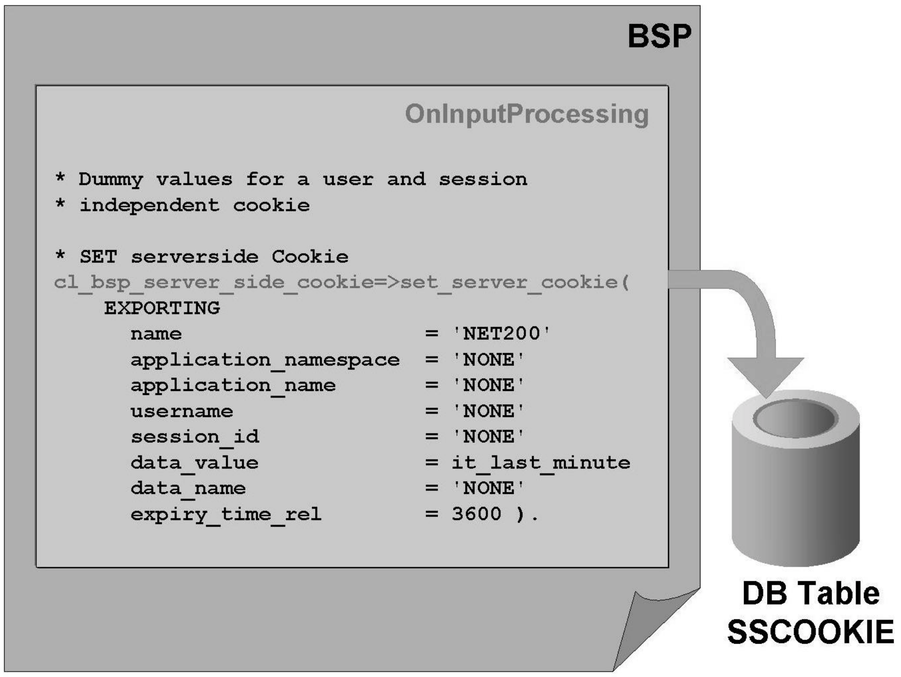
If a BSP application is executed statefully, the BSP runtime environment automtically creats a temporary cookie with which the session ID for this BSP applicaiton is exchanged. With this information, the program context buffered on the applicaiton server can be assigned to the client.

For statelessly and statefully executed BSP applications, the BSP runtime environment automtically creates a temporary cookie with which the system can identify whether a new browser session was started at the client. Whenever the user closes the browser session and starts a new one, the BSP runtime sends the cookie to the client with new contents. From within the application, you can ascertain the session ID of the browser session using the attribute `runtime->session_id`.

Server-Side Cookies in a BSP Application

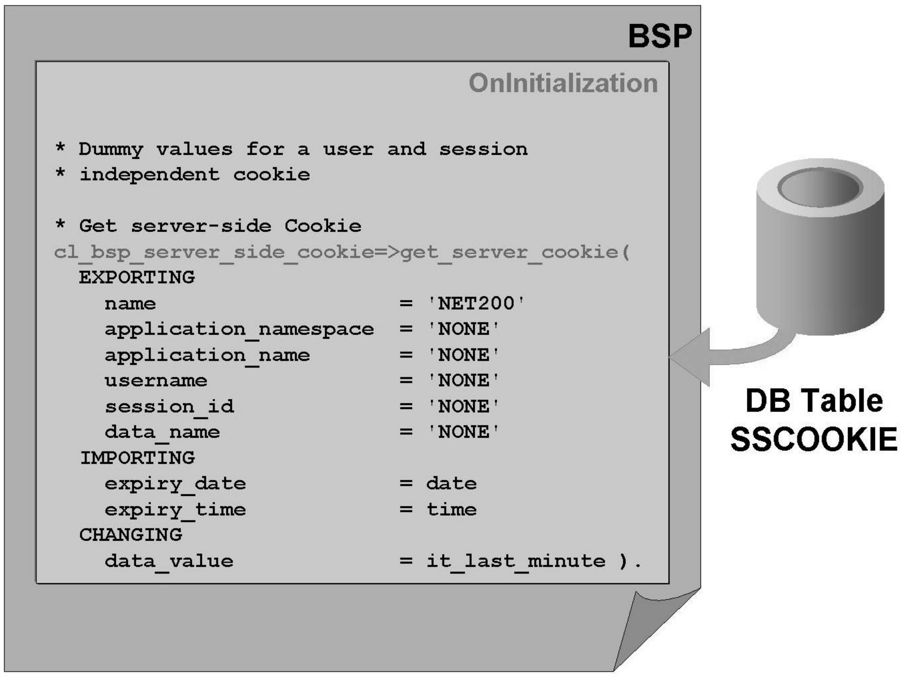
The CL_BSP_SERVER_SIDE_COOKIE class provides methods for working with server-side cookies. Server-side cookies are a form of **persistent data** - that is, they are stored in the database in the form of data clusters in the buffered table SSCOOKIE. A single cookie can be a data object of any complexity (such as a field, a structure, or a table).

Every server-side cookie can be identified by its name. You can also identify a cookie using the name of the BSP application, the name of the package containing the BSP, the session ID of the browser session (not to be confused with the session on the application server), the user name, or a combination of the above. The appropriate interface parameters must be filled when the method is called. However, if interface parameters are not necessary, they can be filled with dummy values.



```
* Dummy values for a user and session
* independent cookie

* SET serverside Cookie
cl_bsp_server_side_cookie=>set_server_cookie(
    EXPORTING
      name                      = 'NET200'
      application_namespace     = 'NONE'
      application_name          = 'NONE'
      username                  = 'NONE'
      session_id                = 'NONE'
      data_value                = it_last_minute
      data_name                 = 'NONE'
      expiry_time_rel           = 3600 ).
```

Figure 41: Saving Server-Side Cookies



```
* Dummy values for a user and session
* independent cookie

* Get server-side Cookie
cl_bsp_server_side_cookie=>get_server_cookie(
  EXPORTING
    name                      = 'NET200'
    application_namespace     = 'NONE'
    application_name          = 'NONE'
    username                  = 'NONE'
    session_id                = 'NONE'
    data_name                 = 'NONE'
  IMPORTING
    expiry_date               = date
    expiry_time               = time
  CHANGING
    data_value                = it_last_minute ).
```

Figure 42: Reading Server-Side Cookies

Server-side cookies offer the following advantages over client-side cookies:

- There is no limit to the number of cookies. (Web browsers can manage a maxiumum of 300 cookies, and only 20 in each domain.)
- There is no limit to the quantity of data that can be sent. (Client-side cookies can store a maximum of 4 KB of data.)
- Complex data can be stored according to type.

It is best to store sensitive data (such as customer data) as a server-side cookie.

A commonly used technique is the combination of a client-side cookie and a server-side cookie. In the server-side cookie, you store sensitive data (for example, customer information data, contents of shopping baskets); in the client-side cookie on the other hand, you store identification information (such as the session ID of the browser session) so that the server can find the appropriate server-side cookie again.

If you use a server-side cookie and do not repeatedly read the application data from the database, this generally improves the performance of the application (the table SSCOOKIE is buffered on the application server). The application data is usually compiled from several database tables through complex SELECT statements. However, if you store the read data to the database in the form of a server-side cookie (data cluster), the access is:

- Limited to the required data quantity
- Displayed in a simple, fully-specified access
- Buffered on the application server (single record buffering)

You can display the server-side cookies using the standard program BSP_SHOW_SERVER_COOKIES. In addition, the standard program BSP_CLEAN_UP_SERVER_COOKIES deletes the expired cookies. However, cookies whose expiration date has passed are not deleted automatically. Therefore, you should plan to execute the program BSP_CLEAN_UP_SERVER_COOKIES regularly.

Hint: To check whether the expiration date has passed in the standard program BSP_CLEAN_UP_SERVER_COOKIES, compare the expiration date with the current system date. If you want to compare the expiration time in hours and minutes, you must do so in the logic of the BSP application. Unfortunately, this check is not performed automatically by the method GET_SERVER_COOKIE.

Object	Selected Methods / Attributes	Class / Type	Event
`REQUEST`	•`get_cookie` •`get_cookies` •`get_data`	`IF_HTTP_ENTITY`	***2, 3, 4, 5, 6***
`RUNTIME`	•`keep_context` •`server` •`session_id` •`application_name` •`application_namespace` •`page_name` •`page_url`	`IF_BSP_RUNTIME`	***1, 2, 3, 4, 5, 6, 7***
`RESPONSE`	•`set_cookie` •`set_data`	`IF_HTTP_ENTITY`	***3, 5, 6***

Event:			
	1: OnCreate	***4***: OnInputprocessing	***7***: OnDestroy
	2: OnRequest	***5***: Layout	
	3: OnInitalization	***6***: OnManipulation	

Figure 43: Global Objects and Session Handling

Exercise 4: Session Handling

Exercise Objectives

After completing this exercise, you will be able to:

- Use hidden fields
- Use server-side cookies

Business Example

The travel agency number that was processed up to now as a constant ('00000110') should now be transferred from the first page to the following pages as a hidden name/value pair. In addition, the last-minute table should be saved using a server-side cookie to reduce unnecessary database accesses.

Task:

Continue to work with your BSP application or copy the model solution from the last exercise. To do this, copy your BSP application ZNET200_##_03 or the BSP application NET200_S_03, giving it the name ZNET200_##_04, where ## stands for your group number. Adhere to the names given and always enter single-digit group numbers with a leading zero, 0#. Sample solution for this exercise is NET200_S_04.

1. The first three BSPs (public/start.htm, public/flights.htm, and public/details.htm) require the travel agency number when reading database table entries. It is thus appropriate to define this value on the first of the three pages and then pass it to the subsequent pages. Because the BSP application is stateless, you should implement this as a hidden (form) field.

   On the BSPs *public/start.htm* and *public/flights.htm*, create the page attribute *travel_ag* of the type *STRING*. Verify that the page attribute is already defined in the BSP *public/details.htm*.

   At the appropriate event, assign the value *00000110* to the page attribute *travel_ag* on the BSP **`public/start.htm`**. Create a page fragment. Give it the name **`hidden.htm`**, define a hidden form field on it with the name **`travel_ag`** and assign it the value of the identically-named page attribute. Insert this page fragment in the pages *public/start.htm* and *public/flights.htm*. Replace the text literal with the page attribute wherever it appears on the first three pages where the travel agency number was previously hard-coded. Make sure that the name/value pair **travel_ag=00000110** is passed from

Continued on next page

the page *public/start.htm* to *public/flights.htm* when the user navigates to it. Which of the page attributes defined in this exercise must be of type *AUTO*?

2. The table of last-minute offers should not be filled by complex select statements every time the application starts. Instead, save it as a **server-side** cookie after it has been read for the first time. Do this so that a separate server-side cookie is generated for each application. This cookie is then used for the next 24 hours, independent of the user or the session. After a complete day has passed, the current last-minute offers must be reread from the database and the old cookie replaced by a new one.

   Use the static methods *CL_BSP_SERVER_SIDE_COOKIE* **=>get_server_cookie** and *CL_BSP_SERVER_SIDE_COOKIE* **=>set_server_cookie**.

   Give the cookie a group-specific name (for example, LAST_MINUTE_##). After the cookie has been read, check that it is still valid.

   Check whether the cookie has been set. To do this, use the BSP_SHOW_SERVER_COOKIES program.

Solution 4: Session Handling

Task:

Continue to work with your BSP application or copy the model solution from the last exercise. To do this, copy your BSP application ZNET200_##_03 or the BSP application NET200_S_03, giving it the name ZNET200_##_04, where ## stands for your group number. Adhere to the names given and always enter single-digit group numbers with a leading zero, 0#. Sample solution for this exercise is NET200_S_04.

1. The first three BSPs (public/start.htm, public/flights.htm, and public/details.htm) require the travel agency number when reading database table entries. It is thus appropriate to define this value on the first of the three pages and then pass it to the subsequent pages. Because the BSP application is stateless, you should implement this as a hidden (form) field.

   On the BSPs *public/start.htm* and *public/flights.htm*, create the page attribute *travel_ag* of the type *STRING*. Verify that the page attribute is already defined in the BSP *public/details.htm*.

   At the appropriate event, assign the value *00000110* to the page attribute *travel_ag* on the BSP **`public/start.htm`**. Create a page fragment. Give it the name **`hidden.htm`**, define a hidden form field on it with the name **`travel_ag`** and assign it the value of the identically-named page attribute. Insert this page fragment in the pages *public/start.htm* and *public/flights.htm*. Replace the text literal with the page attribute wherever it appears on the first three pages where the travel agency number was previously hard-coded. Make sure that the name/value pair **travel_ag=00000110** is passed from the page *public/start.htm* to *public/flights.htm* when the user navigates to it. Which of the page attributes defined in this exercise must be of type *AUTO*?

   a)

BSP	Attribute name	Type	Auto
public/start.htm	travel_ag	STRING	
public/flights.htm	travel_ag	STRING	X
public/details.htm	travel_ag	STRING	X

hidden.htm - LAYOUT

Continued on next page

```
<%@page language="abap" %>
<input type="hidden"
       name="travel_ag"
       value="<%=travel_ag%>">
```

public/start.htm - OnInitialization

```
* Hidden Fields
travel_ag = '00000110'.

...
* Data Retrieval for Last Minutes Offers
  CALL METHOD application->get_last_minute_flights
    EXPORTING
      i_range        = 21
      i_max_rows     = 5
      i_travelagency = travel_ag
    IMPORTING
      e_flights      = it_last_minute.
...
```

public/start.htm - OnInputProcessing

```
CASE event_id.

  WHEN 'flights'.

* Setting attributes for the next page
* (Form fields entries)
  navigation->set_parameter( name = 'travel_ag' ).
...
* Navigation to the next page
  navigation->goto_page( 'FLIGHTS.HTM' ).

ENDCASE.
```

Continued on next page

public/start.htm - LAYOUT

```
<!----  begin of form------ -->
<form>
  <%@ include file = "hidden.htm" %>

  ...

</form>
<!----  end of form------ -->
```

public/flights.htm - OnInitialization

```
...
****************************************.
* BAPI Call via Application class method
****************************************
CALL METHOD application->get_flight_list
  EXPORTING
    travelagency           = travel_ag
*   AIRLINE                =
    destination_from       = dest_from
    destination_to         = dest_to
*   MAX_ROWS               =
  CHANGING
    date_range             = it_date
    flight_connection_list = it_con_dat.
```

public/flights.htm - LAYOUT

```
...
<tbody>
<!------------------------------------->
<!--  Scripting                      -->
<!-----------------  ----------------->
  <% data: wa_con_dat type BAPISCODAT.
  loop at it_con_dat into wa_con_dat. %>
```

Continued on next page

```
    <tr>
      <td>
<!-- Travel agency number is now given  -->
<!-- by the value of the page attribute -->
<!-- travel_ag                          -->
<!-- NO LINE BREAK IN HREF !!!!!!!!!!!! -->
        <a href="Details.htm?
           travel_ag=<%=travel_ag%>&
           connid=<%=wa_con_dat-flightconn%>&
           fldate=<%=wa_con_dat-flightdate%>">
            <%= wa_con_dat-flightconn %>
        </a>
      </td>
      ...
    </tr>
  <% endloop. %>
</tbody>
...
```

public/details.htm - LAYOUT

```
<!----  begin of form------ -->
<form>
  <%@ include file = "hidden.htm" %>

  ...

</form>
<!----  end of form------ -->
```

2. The table of last-minute offers should not be filled by complex select statements every time the application starts. Instead, save it as a **server-side** cookie after it has been read for the first time. Do this so that a separate server-side cookie is generated for each application. This cookie is then used for the next 24 hours, independent of the user or the session. After a complete day has passed, the current last-minute offers must be reread from the database and the old cookie replaced by a new one.

Continued on next page

Use the static methods *CL_BSP_SERVER_SIDE_COOKIE* **=>get_server_cookie** and *CL_BSP_SERVER_SIDE_COOKIE* **=>set_server_cookie**.

Give the cookie a group-specific name (for example, LAST_MINUTE_##). After the cookie has been read, check that it is still valid.

Check whether the cookie has been set. To do this, use the BSP_SHOW_SERVER_COOKIES program.

a)

public/start.htm - OnInitialization

```
* event handler for data retrieval

* set number of travel agency to '00000110'
travel_ag = '00000110'.


******************************************************
* Check, whether last minute offers have allready
* been read and stored as server side cookie
******************************************************
DATA: dummy    TYPE string VALUE 'NONE',
      exp_date TYPE d,
      exp_time TYPE t.

CALL METHOD cl_bsp_server_side_cookie=>get_server_cookie
  EXPORTING
    name                  = 'NET200_COOKIE'
    application_name      = dummy
    application_namespace = dummy
    username              = dummy
    session_id            = dummy
    data_name             = dummy
  IMPORTING
    expiry_date           = exp_date
    expiry_time           = exp_time
  CHANGING
    data_value            = it_last_minute.

******************************************************
* Check, if cookie has expired or if cookie has
```

Continued on next page

```
* not yet been created
******************************************************
IF exp_date < sy-datum OR
   exp_date = sy-datum AND exp_time < sy-uzeit.

* Data Retrieval for Last Minutes Offers
  CALL METHOD application->get_last_minute_flights
    EXPORTING
      i_range        = 21
      i_max_rows     = 5
      i_travelagency = travel_ag
    IMPORTING
      e_flights      = it_last_minute.

* Create Cookie, to avoid complicated database
* operations

  CALL METHOD cl_bsp_server_side_cookie
             =>set_server_cookie
    EXPORTING
      name                  = 'NET200_COOKIE'
      application_name      = dummy
      application_namespace = dummy
      username              = dummy
      session_id            = dummy
      data_value            = it_last_minute
      data_name             = dummy
*     EXPIRY_TIME_ABS       =
*     EXPIRY_DATE_ABS       =
*     EXPIRY_TIME_REL       =
      expiry_date_rel       = 1.
ENDIF.
```

Lesson Summary

You should now be able to:

- List criteria for deciding to use stateful or stateless programming
- Implement techniques for retaining data in a stateless BSP application

Unit Summary

You should now be able to:

- Describe the components of a BSP application
- Create BSP applications
- Create Business Server Pages
- Edit the layout of a Business Server Page
- Describe the components that make up a BSP and the tasks that each of them has
- Describe the event concept for BSPs with flow logic
- List the various options, and define and use the types and data objects in BSPs with flow logic
- Enable users to enter information
- Define static navigation between BSPs
- Define dynamic navigation between BSPs
- Enable data transfer between BSPs
- React to errors in transmitted data
- Work with global objects
- List criteria for deciding to use stateful or stateless programming
- Implement techniques for retaining data in a stateless BSP application

Test Your Knowledge

1. You design the HTML page in the layout.

   Determine whether this statement is true or false.

   □ True

   □ False

2. You specify the chronological flow of a BSP using the processing sequence of the event handlers.

   Determine whether this statement is true or false.

   □ True

   □ False

3. You can add more events to a BSP by creating new events in the application class.

   Determine whether this statement is true or false.

   □ True

   □ False

4. The processing sequence of the event handlers can be influenced using client-side scripting.

   Determine whether this statement is true or false.

   □ True

   □ False

5. In a BSP, you can include links to other BSPs in the same application.

   Determine whether this statement is true or false.

   □ True

   □ False

6. In a BSP, you can insert links to other BSPs in other applications.

   Determine whether this statement is true or false.

   □ True

   □ False

7. In the ACTION attribute of an HTML form (FORM tag), you can store the names of several BSPs to be executed.

   Determine whether this statement is true or false.

   ☐ True

   ☐ False

8. The OnInputProcessing event of a BSP is always processed.

   Determine whether this statement is true or false.

   ☐ True

   ☐ False

9. The global object *NAVIGATION* is available for every event in a BSP.

   Determine whether this statement is true or false.

   ☐ True

   ☐ False

Answers

1. You design the HTML page in the layout.

   Answer: True

   In the BSP layout, you should store only the page's appearance in HTML, and the dynamic page elements in ABAP or JavaScript. In particular, do not include any of the business logic in the layout.

2. You specify the chronological flow of a BSP using the processing sequence of the event handlers.

   Answer: True

   The event handlers are executed in a predefined sequence.

3. You can add more events to a BSP by creating new events in the application class.

   Answer: False

   You cannot define other events apart from the predefined events.

4. The processing sequence of the event handlers can be influenced using client-side scripting.

   Answer: False

   The processing sequence of the event handlers is always fixed. All you can do is influence whether the events OnCreate, OnDestroy, and OnInputProcessing are to be executed.

5. In a BSP, you can include links to other BSPs in the same application.

   Answer: True

   All you need to do is to enter the name of the appropriate BSP in the HREF attribute of a hyperlink or in the ACTION attribute of the FORM tag.

6. In a BSP, you can insert links to other BSPs in other applications.

   Answer: True

   However, you must specify the name of the BSP application in the URL. You cannot simply supply the name of the BSP because the system will search only in the current application.

7. In the ACTION attribute of an HTML form (FORM tag), you can store the names of several BSPs to be executed.

   Answer: False

   You can store the name of only one BSP that is to be executed.

8. The OnInputProcessing event of a BSP is always processed.

   Answer: False

   The OnInputProcessing event will be processed if the name OnInputProcessing appears in the query string. This event can be initiated by the user clickng the Send button or clicking a link, or implemented using JavaScript functions.

9. The global object *NAVIGATION* is available for every event in a BSP.

   Answer: False

   The global object *NAVIGATION* is only available for OnRequest, OnInitialization, and OnInputProcessing events. You specify the next page with the available methods, *goto_page* and *next_page*.

Unit 3

Layout and Language

Unit Overview

In this unit, you will learn how MIME objects are used in BSP applications. In addition, you will learn how you can adjust the layout of a BSP application by assigning a Theme and without making modifications. Finally, you will learn how a BSP application can be presented to the Internet user in more than one language.

Unit Objectives

After completing this unit, you will be able to:

- Import MIME objects into the MIME Repository
- Include MIME objects in BSP applications
- Adapt a BSP application without making modifications
- Enable translation of text literals in Business Server Pages

Unit Contents

Lesson: Including MIME Objects

Lesson Overview

In this lesson you will be shown how MIME objects can be managed in the SAP Web Application Server and used in the layout of Business Server Pages.

Lesson Objectives

After completing this lesson, you will be able to:

- Import MIME objects into the MIME Repository
- Include MIME objects in BSP applications

Business Example

The layout of a BSP application for the Web is to be improved by adding graphics.

The MIME Repository

The **MIME Repository** serves as a storage repository for all MIMEs (style sheets, graphics, icons) in the SAP system. MIMEs are created as objects in the SAP database and can be referenced on pages of the BSP applications. MIME objects use the SAP development infrastructure; in particular, changes in the MIME Repository, such as importing new MIMEs, are written to a transport request.

The MIME Repository is visible through a browser. Here all the MIME objects are arranged hierarchically in directories in a tree diagram. When you start the browser, it overlaps the entire left navigation area in the Object Navigator. **For each BSP application**, a **folder with the same name** is created automatically in the MIME Repository. The BSP application folder is used as a storage area for all application-specific MIMEs. This particular folder cannot be deleted explicitly and is therefore part of the MIME Repository as long as the BSP application exists. However, as soon as the BSP application is deleted, the respective folder with all the MIMEs contained in it is deleted automatically as well. In addition to application-specific MIMEs, cross-application MIMEs for each BSP application are available in the Public folder.

When a BSP application is transported, the respective application folder in the MIME Repository is not automatically transported as well. Both the application folder and the MIME objects that are to be transported, and possibly also its subdirectories, must be assigned **explicitly** to a transport request. All the MIME Repository objects are assigned to one namespace. If there are customer developments, the corresponding directories in the customer namespace are accessed.

You can **import** any MIME objects you like into the SAP Web Application Server. For a detailed list of the formats supported, refer to the online documentation under the entry “Supported MIME Categories.” During import, the MIME objects are stored in database tables (SMIM* tables). To embed a MIME object in a Business Server Page, use the Drag&Drop function to drag the URL from the MIME Repository into the page layout. Then, all you need to do is enclose the URL within the appropriate HTML tag (for example, `<img src="URL">`).

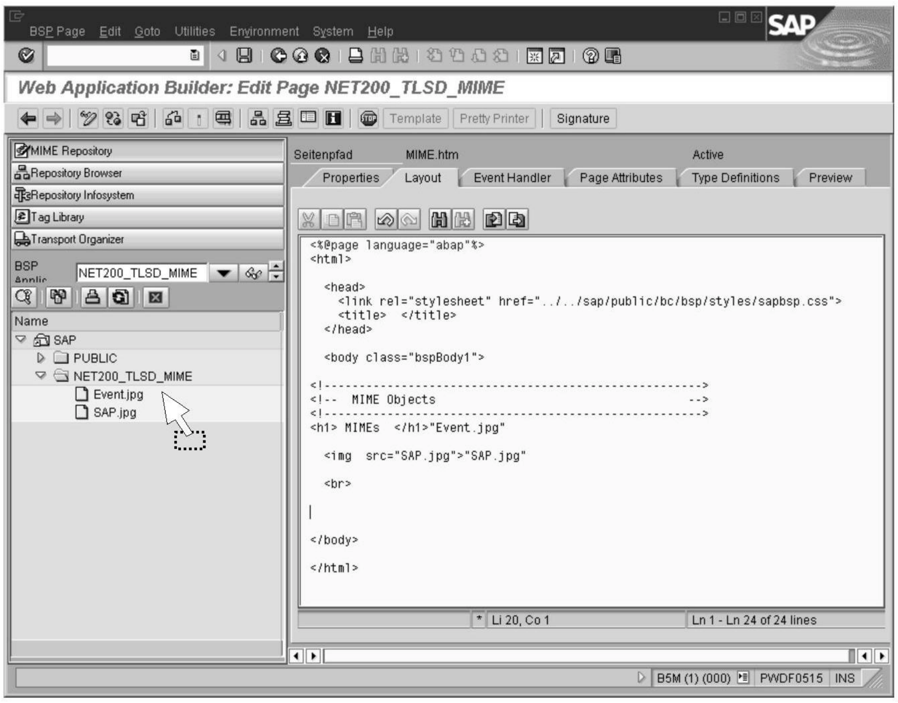
Hint: The directory structure *sap/<path>/<mime>* displayed in the MIME Repository represents a part of the complete URL for the corresponding MIME object *http://<host>:<port>/sap/bc/bsp/sap/<path>/<mime>*.

How to Import and Include MIME Objects

1. Switch to change mode in the page layout of the Business Server Page into which the MIME object is to be included.
2. Start the MIME Repository and select the node with the name of your BSP application.
3. In the context menu (right mouse-click), choose the menu path *Import -> MIME Objects*. aus. The file manager of your computer is started automatically and you can select the respective MIME object in the file system.
4. Complete the MIME properties and save your entries. Because the import into the SAP system occurs now, the dialog box for entering the object catalog appears. Assign a package to the MIME object and save your entries.
5. Drag the MIME object to the required position in the page layout. The URL is copied to the source text.
6. Enclose the URL with the appropriate HTML tag:

   ```
   <img src="URL">
   ```
7. Test the Business Server Page.

Caution: When you copy BSP applications, the folders, but not their contents, are copied to the MIME Repository. To do this, the objects need to be explicitly assigned to a transport request (right mouse-click, then choose *Additional Functions -> Write Transport Request*).



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

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

  <body class="bspBody1">

<!--------------------------------------------------->
<!--  MIME Objects                                 -->
<!--------------------------------------------------->
<h1> MIMEs  </h1>"Event.jpg"

  <img  src="SAP.jpg">"SAP.jpg"

  <br>



</body>

</html>
```

Figure 44: Including MIME Objects

The MIME Repository allows you to create new MIME objects also and to assign them directly to a folder in the MIME Repository. To do this, go to the context menu (right mouse-click) and choose the menu path *Create -> MIME Object*. You can choose from the MIME types maintained in the system (database table SDOKFEXT). To process the source code of the

MIME object, you can start the editor from the dialog box. To do this, you must first define the editor by choosing the menu path *Utilities -> Settings -> Business Server Pages -> External HTML Editor*.

Hint:

- You can convert MIME objects of the text category ("text/html", "text/plain", and so on) into dynamic pages of your BSP application (BSP with flow logic).
- You can create subdirectories for application folders to better structure the storage, for example, if you have a large set of MIME objects.
- MIME objects are not marked as relevant for translation in the standard version. However, if you wish to allow translating language-specific MIME objects, you only need to change the appropriate indicator in the development system (right mouse-click, then choose *Properties -> Log.Doc.Properties*)..
- If the relative URL to a MIME object is created using Drag&Drop, the path specification refers to the target object (Drop). However, if the target object is a page fragment, which is in turn included in other BSPs using the INCLUDE directive, the relative path specification may be incorrect. This is always the case if the page fragment and the including BSP are assigned to different directory levels.

Exercise 5: Including MIME Objects

Exercise Objectives

After completing this exercise, you will be able to:

- Include MIME Objects in BSP applications

Business Example

You wish to include graphics (MIME objects) in the page fragment header.htm. In addition, the source code of the page fragment *style.htm* is to be copied to a Cascading Style Sheet. Then the CSS is to be referenced in the BSPs of your application instead of the page fragment.

Task:

Continue to work with your BSP application or copy the model solution from the last exercise. To do this, copy your BSP application ZNET200_##_04 or the BSP application NET200_S_04, giving it the name ZNET200_##_05, where ## is your group number. Adhere to the names given and always enter single-digit group numbers with 0#. The model solution for this exercise is NET200_S_05.

1. You want to make the page header information more interesting using graphics. To do this, you include images (MIME objects) in the page fragment header.htm. You have two options:

   1) Either you use MIME objects from the PUBLIC/NET200 node in the MIME Repository (recommended) or

   2) You export the MIME objects from the MIME Repository node PUBLIC/NET200 and import them into the MIME node of your BSP application.

2. Create a new MIME object of the type *text/css* (Cascading Style Sheet). Give the MIME object the name **styles.css** and copy the source code of the page fragment **styles.htm**. In the MIME Repository, assign this object to your application. Delete the Include directive for including the page fragment *styles.htm* in the layout of all BSPs of your application. Instead, reference the respective MIME object from the `<head>...</head>` part of the pages.

Solution 5: Including MIME Objects

Task:

Continue to work with your BSP application or copy the model solution from the last exercise. To do this, copy your BSP application ZNET200_##_04 or the BSP application NET200_S_04, giving it the name ZNET200_##_05, where ## is your group number. Adhere to the names given and always enter single-digit group numbers with 0#. The model solution for this exercise is NET200_S_05.

1. You want to make the page header information more interesting using graphics. To do this, you include images (MIME objects) in the page fragment header.htm. You have two options:

   1) Either you use MIME objects from the PUBLIC/NET200 node in the MIME Repository (recommended) or

   2) You export the MIME objects from the MIME Repository node PUBLIC/NET200 and import them into the MIME node of your BSP application.

   a) Regarding 1):

      Start the Object Navigator (transaction SE80) and switch to change mode of the page fragment header.htm. In the object tree, choose the MIME Repository. Navigate to the node *PUBLIC → NET200*. Drag the required MIME object into the layout of the page fragment header.htm (Drag&Drop).

   b) Regarding 2):

      Start the Object Navigator (transaction SE80) and switch to change mode of the page fragment header.htm. In the object tree, choose the MIME Repository. Navigate to the node *PUBLIC →NET200*. Export a MIME object to your front end by right-clicking the function *Export*. Switch to the MIME node of your BSP application and import the MIME object. Drag the required MIME object into the layout of the page fragment header.htm.

   c) To include MIME objects, implement the following source code:

      ```
      <img src="<PATH>">
      ```

      Hint: The path specification <*PATH*> generated using Drag&Drop is a relative path specification. Since your pages are located in subdirectories (*public* or *protected*), this relative specification must be adjusted accordingly.

Continued on next page

2. Create a new MIME object of the type *text/css* (Cascading Style Sheet). Give the MIME object the name **styles.css** and copy the source code of the page fragment **styles.htm**. In the MIME Repository, assign this object to your application. Delete the Include directive for including the page fragment *styles.htm* in the layout of all BSPs of your application. Instead, reference the respective MIME object from the `<head>...</head>` part of the pages.

   a) Navigate to the layout of the page fragment *styles.htm*. Copy the entire source code into the clipboard (Ctrl + C). Make sure that the notepad is used as a local HTML editor. To do this, navigate to the respective dialog box using the menu path *Utilities → Settings → Business Server Pages*. Then enter **`Notepad`** in the field provided. Now start the MIME Repository. Navigate to the node of your application. Create a new MIME object by clicking the right mouse key on the function *Create → MIME Object* . As MIME type, choose *text/css*, and as object name *styles.css*. Start the editor by pressing the corresponding pushbutton or F8. Enter the contents of the clipboard (Ctrl + V). Save the file.

      One after the other, open the layout of the BSPs of your application and delete the Include directive for the page fragment *styles.htm*. Instead, fill in the line `<link rel=stylesheet href="<PATH>">` in the section `<head>...</head>`.

Lesson Summary

You should now be able to:

- Import MIME objects into the MIME Repository
- Include MIME objects in BSP applications

Lesson: Adjusting the Layout

Lesson Overview

In this lesson you will learn how to change the design of BSPs whose layout is structured with classic HTML, **not** with BSP extensions, without making modifications. For this purpose, this lesson also explains the use of **cascading style sheets** (CSS) and **themes**.

Lesson Objectives

After completing this lesson, you will be able to:

- Adapt a BSP application without making modifications

Business Example

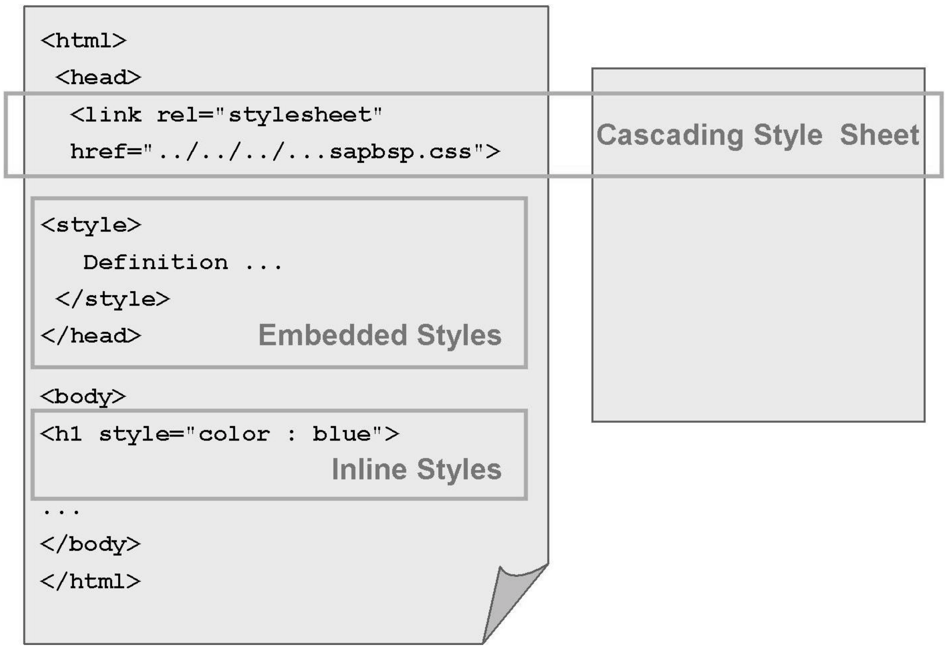
You have a BSP application whose layout you wish to adapt to the respective requirements, without making modifications.

Cascading Style Sheets (CSS)

A company's Web applications should all have a uniform appearance, if possible. This means that style elements such as colors, script types, and logos should always be implemented in the same way so that the user can find his or her way through the respective pages and also always recognize the product and its producer. Ensuring this uniform appearance over several pages involves much effort if the formatting information is maintained for each page or even for each tag. Therefore, it is common to separate the layout from the Web page functions and to store the layout data (such as the background color of a page) in a separate file. We call these files Cascading Style Sheets, or CSS for short. You can link a CSS with an HTML page using the LINK tag.

```
<LINK REL =STYLESHEET
      HREF="http://www.myserver.com/mysheets.css">
```

Style definitions are cascading, which means they can be defined by including a style sheet, but they can be overwritten with a new definition in the case of certain page objects - either locally for a page or locally for a tag. A hierarchy defines exactly which definition element determines the final layout of a page object. To find out which language elements are available in a CSS and how to use the style sheets appropriately, refer to the respective special documentation.



```
<html>
 <head>
  <link rel="stylesheet"
  href="../../../...sapbsp.css">
<style>
   Definition ...
 </style>
</head>
<body>
<h1 style="color : blue">
...
</body>
</html>
```

Figure 45: Styles in HTML Pages

The Theme Concept

To be able to adapt the layout of a BSP application to your own requirements, you use **themes**. Themes are independent Repository objects that are maintained in the Repository Browser (SE80). They represent a kind of replacement matrix. You use themes to specify for which MIME object a new version is created. When the theme is used, not the original version of the MIME object is used, but instead the version of the object defined with the theme.

You can assign the theme to one or more BSP applications by Customizing. In this way, the original design can be converted to the company-specific design, without having to modify the BSP application. However, this depends on whether the URLs to the MIME objects are correctly structured in the layout of the BSPs:

Each MIME object is requested by the browser in a new HTTP request. The URL to the MIME object must make clear which version of this object is to be returned. Therefore, when the BSP application is started, the BSP runtime environment changes the URL for the first BSP by adding a parenthetical expression containing the logon language and the theme in encrypted form. This parenthetical expression is called the **cache key**. It comes after the first directory name in the URL (usually */sap(...)*). The URL for the MIME objects to be included must now be structured in a way that the cache key comes after the first directory node. The simplest way of doing this is to use relative path specifications.

Hint: The same logon language and the same theme will always result in the same cache key.

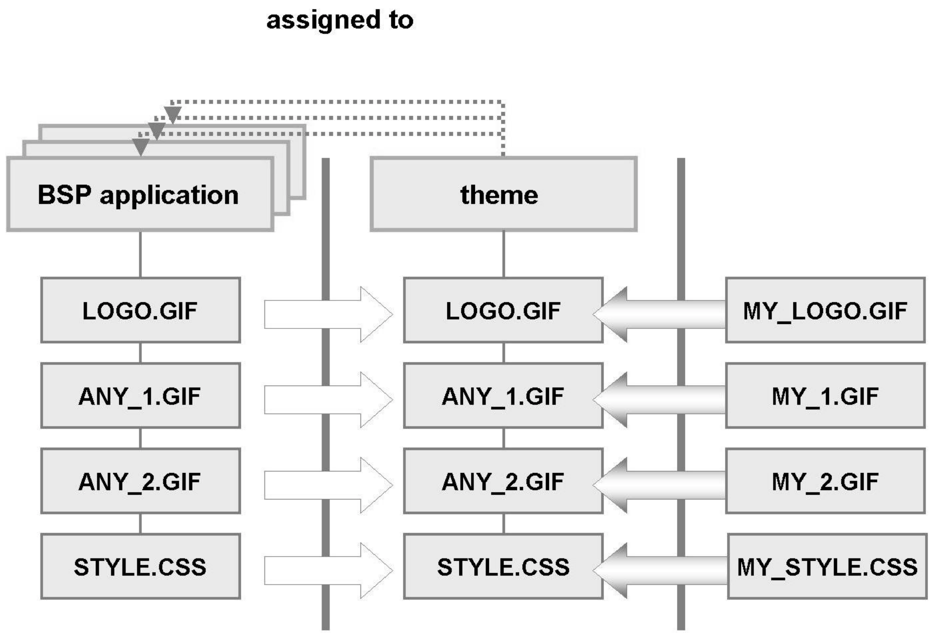


Figure 46: Theme Concept

After you have created the theme, the MIME objects, for which another version is to be created, are assigned to the theme via Drag&Drop (from the MIME Repository). Using the context menu for the MIME object ("Files" tab of the theme), you can then change the MIME object or import a new version from the file system.

To have the browser display a MIME object in its most recent form after a change has been made, you need to perform these steps:

- The browser must request the MIME object again when displaying the corresponding HTML page (into which the MIME object is incorporated). However, if the browser already displayed the MIME earlier, the object is reloaded from the browser cache for renewed display. To prevent this, the user needs to request the object again (REFRESH). Alternatively, you can set up the browser so that it does not use its cache.
- If the browser requests the object again, it sends its request to the SAP Web Application Server. The SAP Web Application Server also buffers objects that are stored in the MIME Repository and have already been requested. Each object in the server cache has an expiration date. Before this time period runs out, all the requests for this object are loaded from the server cache. Therefore, changes to an object that has already been buffered require that you remove the object from the server cache. To do this, you can use transaction SMICM. Choose the menu path *Goto -> HTTP Server Cache -> Display* to display all buffered objects.

Exercise 6: Adjusting the Layout

Exercise Objectives

After completing this exercise, you will be able to:

- Adapt the design of a BSP application to your needs by assigning a theme

Business Example

The layout of your BSP application is to be adapted to the corporate design and brand of the company.

Task:

Continue to work with your BSP application or copy the model solution from the last exercise. To do this, copy your BSP application ZNET200_##_05 or the BSP application NET200_S_05, giving it the name ZNET200_##_06, where ## stands for your group number. Adhere to the names given and always enter single-digit group numbers that begin with a leading zero as in 0#. The model solution for this exercise is NET200_S_06.

1. The graphic used in the page header is to be replaced using the theme concept. First create a theme for this and name it **`ZNET200_##`**. Assign the graphic included in your page fragment *header.htm* to the theme. In the MIME Repository, select an object from the existing graphics that you want to appear in the page header from now on. Export this graphic to the temporary file on your computer. Go back to editing the theme. Overwrite the content of the graphic displayed in the page header with the screen you exported in the last step. Save the theme. Assign the theme to your application byCustomizing.
2. Change the content of the file *style.css* in the same way as described in the first part of this exercise.

Solution 6: Adjusting the Layout

Task:

Continue to work with your BSP application or copy the model solution from the last exercise. To do this, copy your BSP application ZNET200_##_05 or the BSP application NET200_S_05, giving it the name ZNET200_##_06, where ## stands for your group number. Adhere to the names given and always enter single-digit group numbers that begin with a leading zero as in 0#. The model solution for this exercise is NET200_S_06.

1. The graphic used in the page header is to be replaced using the theme concept. First create a theme for this and name it **`ZNET200_##`**. Assign the graphic included in your page fragment *header.htm* to the theme. In the MIME Repository, select an object from the existing graphics that you want to appear in the page header from now on. Export this graphic to the temporary file on your computer. Go back to editing the theme. Overwrite the content of the graphic displayed in the page header with the screen you exported in the last step. Save the theme. Assign the theme to your application byCustomizing.

   a) In the Object Navigator, open the Repository Browser. Navigate to your package. Create the theme **ZNET200_##**. To do this, choose *Create -> Web Objects -> Theme* from the context menu (right mouse-click). Now start the MIME Repository in the navigation area. Navigate to the node of the application that is assigned to the graphic that has appeared until now in the page header of your application. Drag the graphic to the object list of your theme. Now look for another graphic in your MIME Repository. Export the graphic to the file *C:\temp* on your front end (right mouse-click *Export/Import -> Export as copy*). Now focus on the node of your theme having the name of the graphic to be replaced and import the new graphic (right mouse-click *Import*). Save the theme. Assign the theme to your application. To do this, use the correponding pushbutton (Shift + F8) that appears in the application toolbar above the theme. Start your BSP application.

      Hint: If you wish to change the composite definition of the graphic again, you must remove the graphic from the server cache so that the changed version is reloaded.

Continued on next page

2. Change the content of the file *style.css* in the same way as described in the first part of this exercise.

   a) Proceed as described in the first part of this exercise. Use the notepad as an external editor to change the source code of the CSS.

Lesson Summary

You should now be able to:

- Adapt a BSP application without making modifications

Lesson: Internationalization

Lesson Overview

In this lesson, you will learn how to manage multilanguage texts in Business Server Pages.

Lesson Objectives

After completing this lesson, you will be able to:

- Enable translation of text literals in Business Server Pages

Business Example

A BSP application is to be provided in different languages.

Internationalization Using the Online Text Repository (OTR)

The Online Text Repository (OTR) is a central storage area for texts and provides services for processing these texts. The Online Text Repository supports the entry and translation of texts and can be used for BSP applications. OTR directives for entry and translation of text can be used in the layout of a Business Server Page. Here a distinction is made between alias texts and long texts.

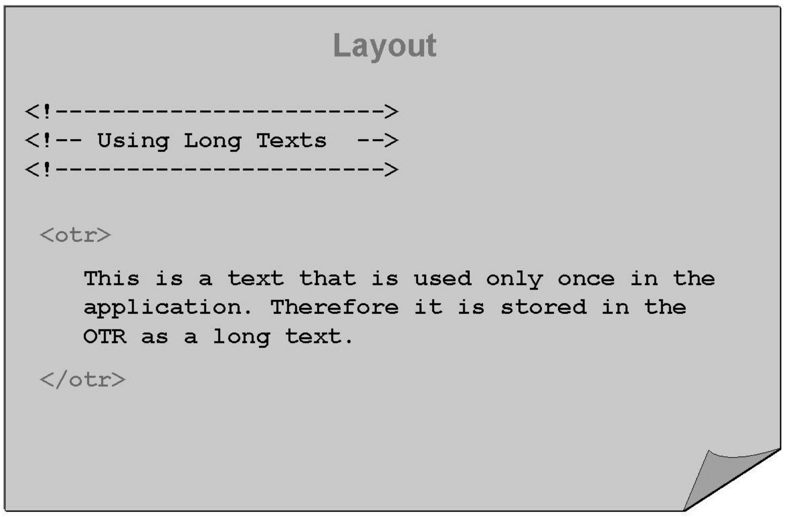
Alias texts are reusable text literals with a length of less than 255 characters. For identification purposes, an alias name is assigned to each text literal. With this name, the text from BSP applications can be addressed through the directive **<%=OTR(AliasName)%>**.

Long texts can be any length. Relevant text passages are enclosed by the tags **<OTR> ... </OTR>** and can thus be translated.

The decision whether to create a text literal as an alias text or as a long text is not based solely on the length of the text stored. A more important distinguishing feature is the frequency of use of a text: If a text appears only once in a BSP application of the package, this text is stored as a long text in the OTR. However, if a text is used frequently, the text is stored as a short text in the Online Text Repository under its alias name. With this alias name, the text can be addressed and used again. The text needs to be translated only once. When it is created, a OTR text that is not assigned to a local or private package is included in a transportable change request. This ensures that new or changed texts are also available in the subsequent systems. The connection to the Change & Transport System is thus guaranteed.

Creating Long Texts

You create a long text by entering the text between the opening and closing OTR tags in the layout of the Business Server Page and by activating the page. When you activate the page, the texts are passed to the Online Text Repository.



```
<!---------------------->
<!-- Using Long Texts  -->
<!---------------------->

 <otr>

    This is a text that is used only once in the
    application. Therefore it is stored in the
    OTR as a long text.

 </otr>
```

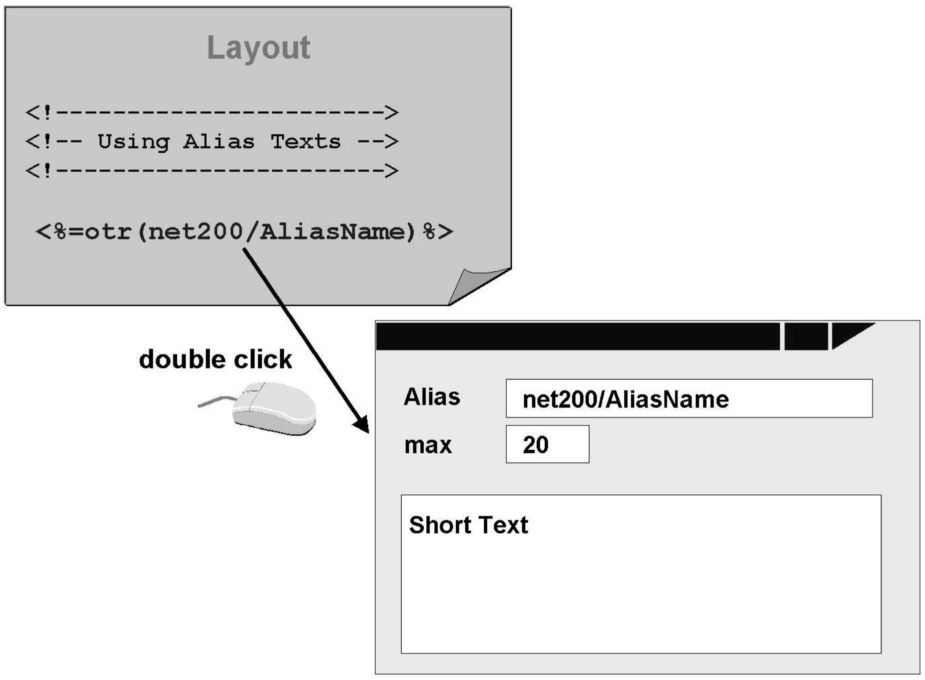
Figure 47: Including OTR Long Texts

Creating Alias Texts

You can create and use alias texts in one of three ways:

1. In the layout, you can use the OTR directive with an alias name that does not exist and then double-click the alias name. This takes you to a screen where you can maintain the alias name, the maximum text length, and the text. The alias name always starts with the name of the package to which the alias text is assigned. If you have forgotten the package name in the OTR directive, it will be added automatically. To prevent redundancies in the Online Text Repository, when you save the text, the system searches for similar alias texts that may be in use already and displays these alias texts.
2. You can first create the alias text and then refer to it afterwards. To do this, navigate out of BSP processing into the OTR browser (through the menu *Goto -> OTR Browser*). All the alias texts that already exist in the package (to which the currently processed BSP belongs) are displayed. In addition, all the texts belonging to the package SOTR_VOCABULARY_BASIC are displayed. This package contains alias texts that can be used for all packages. The alias texts are copied from the OTR browser using Drag&Drop.
3. You can create alias texts using the transaction SOTR_EDIT. First you need to select the language (original language) in which you want to create the alias text.

Alias texts that are stored in the OTR can be addressed not only in the layout of the BSP, but also from the event handlers. Here you use the method **GET_OTR_TEXT** belonging to the global object **RUNTIME**.



```
<!---------------------->
<!-- Using Alias Texts -->
<!---------------------->

 <%=otr(net200/AliasName)%>
```

Alias: net200/AliasName
max: 20

Figure 48: Creating Alias Text Using Forward Navigation

Hint: The Online Text Repository not only stores the texts, but also manages a context for each OTR object. The person who created the text, the other languages, and so on, are stored in the context.

Providing Multilanguage BSP Applications

To make a BSP application available in several languages, you should specify all translation-relevant texts as OTR texts (alias or long texts). The texts can be translated using transaction SE63.

When you start a BSP application, it is possible for the logon language to be determined by the corresponding setting in the Web browser (*Tools -> General -> Languages*) - as of SAP Web AS Release 6.20. The language stored in the Web browser is transmitted at the time of the first request in the HTTP header (*accept language* field in the header) to the SAP Web Application Server. When the HTML pages are generated from the BSPs, the language-dependent texts are inserted accordingly. This ensures that the statically stored language preferences of the client are automatically considered.

However, it is often desirable to be able to select a logon language for the SAP Web Application Server that is different from the standard language in the browser (for example, if this standard language is not provided on the SAP Web Application Server). To allow the selection of another language, the language must be communicated to the system when the BSP application is started. There are several options for this:

1. For each logon language, you can create an **alias** for the service you wish to start using transaction SICF. The logon language is stored as an attribute of the alias. A URL is assigned to the individual aliases according to their position in the SICF tree. Possible logon languages are therefore represented by a selection of different services pointing to a single BSP application. The selected language is passed to the following page through a field in the HTTP header (*accept language* field). In addition, the language is stored in the cache key in coded form. The cache key is incorporated in the URL (string in brackets) and serves to identify language and topic-dependent MIME objects in the HTTP server cache. This URL, and therefore also the cache key, is returned with the next request in the HTTP header to the server and is automatically evaluated by the server.

   Caution: When using an alias, you may have to adjust the relative path specifications for the MIME objects to be included, since they refer to the alias URL of the BSP. When structuring the URL to the MIMEs, you must ensure that the cache key is not lost, since it is used to identify the correct (langauge and theme-specific) version of the MIME object.

   If the *Logon Data Required* flag is set for the alias, the language stored for the alias cannot be changed.
2. You can extend the URL for calling the first page of a BSP application to include the query string **SAP-LANGUAGE=<LANGUAGE>**. This setting is also passed to the next page in the case of stateless applications. This parameter is evaluated only if the logon data for this service is not marked as required.
3. If the logon language is not defined either by a query string, the alias settings, or the language setting stored in the browser, the logon takes place in the language assigned to the user (fixed values) or in the default language of the system.

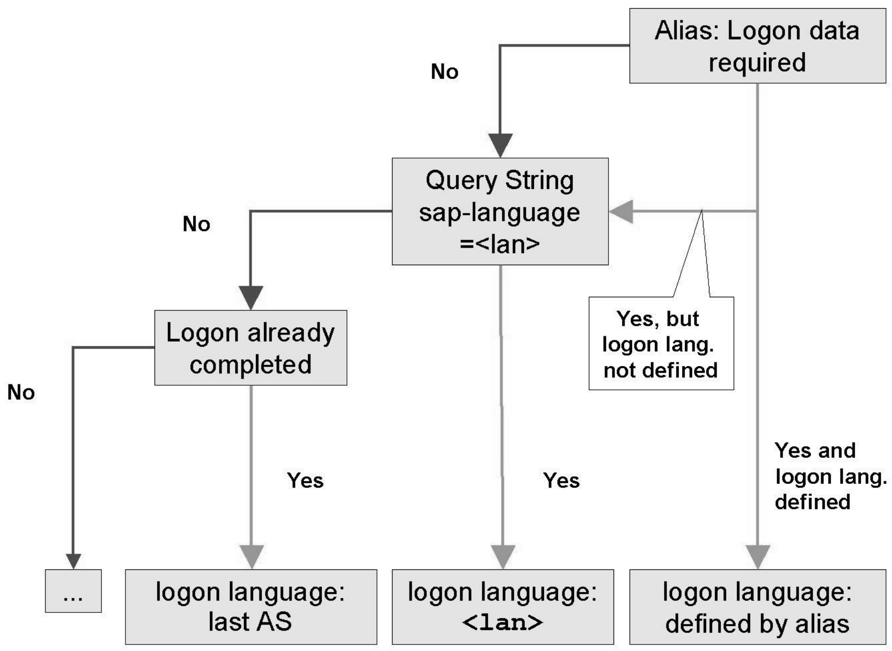

Alias: Logon data required
No
Query String sap-language =<lan>
Yes, but logon lang. not defined
No
Logon already completed
No
Yes
Yes
Yes and logon lang. defined
...
logon language: last AS
logon language: <lan>
logon language: defined by alias

Figure 49: Selection of Logon Language (1)

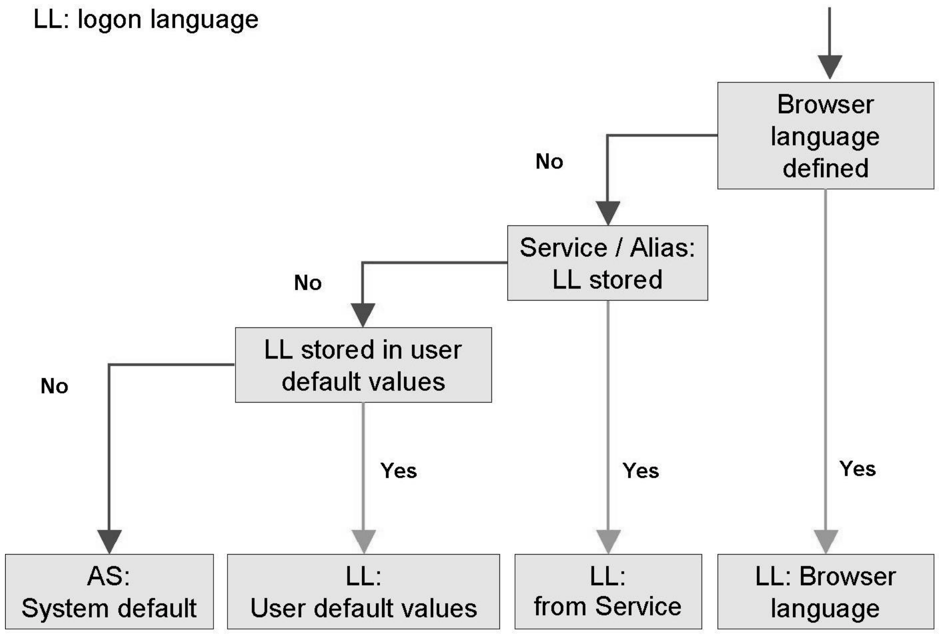
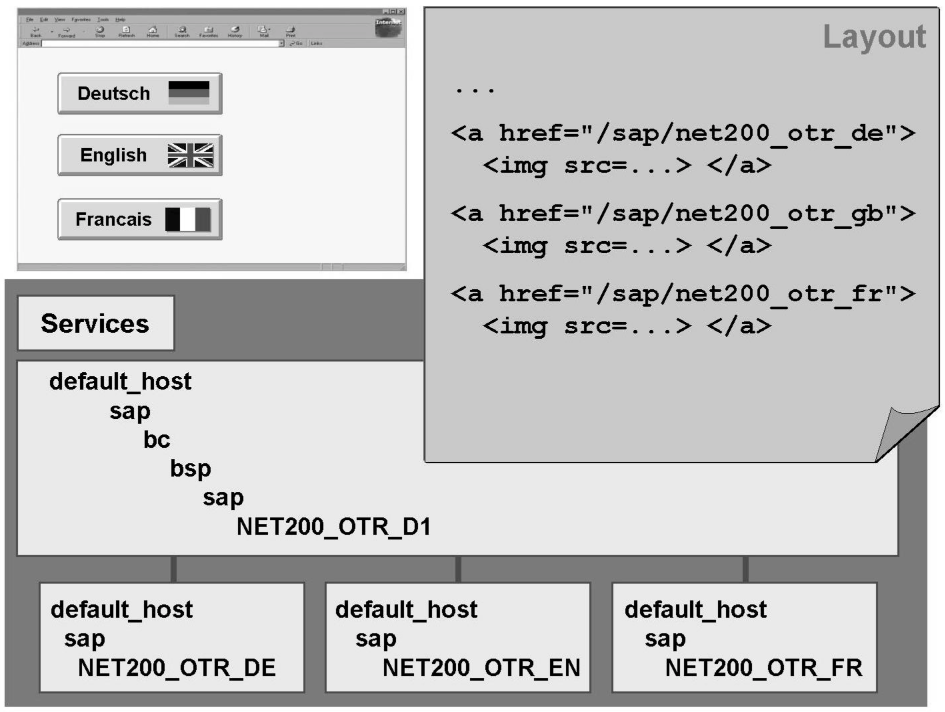


Figure 50: Selection of Logon Language (2)

To offer the user a simple selection of possible languages, you can transmit a start page before the actual BSP application. This page contains a hyperlink for each language.



```
...
<a href="/sap/net200_otr_de">
  <img src=...> </a>
<a href="/sap/net200_otr_gb">
  <img src=...> </a>
<a href="/sap/net200_otr_fr">
  <img src=...> </a>
```

Figure 51: Depiction of Language Dependency

Exercise 7: Internationalization

Exercise Objectives

After completing this exercise, you will be able to:

- Make translation-relevant texts translatable in the layout of BSPs
- Translate long texts and alias texts
- Provide a selection for the language in which the pages of a BSP application are to be displayed

Business Example

Your online flight-booking application is to be offered internationally. The default language for starting the BSP application should be the language stored in the browser. Using appropriate links on the start page, it should be possible to start the application in other provided languages.

Task:

Continue to work with your BSP application or copy the model solution from the last exercise. To do this, copy your BSP application ZNET200_##_06 or the BSP application NET200_S_06, giving it the name ZNET200_##_07. ## stands for your group number. Adhere to the names given and always enter single-digit group numbers with 0#. The model solution for this exercise is: NET200_S_07.

1. On the first three pages of your application, identify all the translation-relevant texts and replace these with alias texts or long texts. Decide, on the basis of the frequency of a text, which type of translation ability you want to choose. Use the tag browser to create the OTR directive in the layout for alias texts. First make all the texts on a BSP translatable.

   After activating the page, copy the OTR directives for texts already translated by dragging them from the OTR browser to the other BSPs.
2. Use transaction SE63 to translate the long text and alias texts you have created into another language.
3. Insert hyperlinks on the start page to make it possible to start the BSP application in the original language and in the language you selected in the second part of the task. Use either a text or a graphic from the MIME repository for the hyperlinks.

Solution 7: Internationalization

Task:

Continue to work with your BSP application or copy the model solution from the last exercise. To do this, copy your BSP application ZNET200_##_06 or the BSP application NET200_S_06, giving it the name ZNET200_##_07. ## stands for your group number. Adhere to the names given and always enter single-digit group numbers with 0#. The model solution for this exercise is: NET200_S_07.

1. On the first three pages of your application, identify all the translation-relevant texts and replace these with alias texts or long texts. Decide, on the basis of the frequency of a text, which type of translation ability you want to choose. Use the tag browser to create the OTR directive in the layout for alias texts. First make all the texts on a BSP translatable.

   After activating the page, copy the OTR directives for texts already translated by dragging them from the OTR browser to the other BSPs.

   a) Open the initial page of your application in the Web Application Builder. Set all the texts that are used on this page only between OTR tags (`<otr>...</otr>`). For column headings, use alias texts because they can also be used on the subsequent pages or in other applications. To do this, replace the translation-relevant texts with an OTR directive (`<%=otr(znet200_##/...)>`). The directive can be created in the layout using the tag browser (copy by dragging and dropping). After you double-click the directive, a dialog box appears. Here you can enter the text and the maximum length (relevant for translation). Save your entries. Activate your BSP.

      Proceed in the same way for the other two pages of your application. If you wish to include an alias text that was already created, copy this text from the Online Text Repository. To do this, start the OTR browser using the menu path *Goto -> Online Text Repository Browser*. You might need to refresh the browser to display the alias texts that you just created.

      Test your application. All texts will appear in your logon language. However, if you start the application in a different language (for example, by adding the query string `?sap-language=<lan>`, where `<lan>` is the language abbreviation of the logon language), no texts will appear.

Continued on next page

2. Use transaction SE63 to translate the long text and alias texts you have created into another language.

   a) *Translation -> OTR -> Short Texts / Long Texts*

3. Insert hyperlinks on the start page to make it possible to start the BSP application in the original language and in the language you selected in the second part of the task. Use either a text or a graphic from the MIME repository for the hyperlinks.

   a)

public/start.htm - Layout

```
...
<body>

<!------------------------------------------->
<!-- Page Fragment with the page header     -->
<!------------------------------------------->
  <%@include file="Header.htm" %>

<!------------------------------------------->
<!-- Last Minute Offers                     -->
<!------------------------------------------->
  <table class=noborder>
    <tr>
      <td class=noborder>
        <h3><otr>Last-Minute Angebote</otr></h3>
      </td>
      <td class=corner>
        <a href="?sap-language=en">
          <img src="../../public/net200/gb.gif">
        </a>
        <a href="?sap-language=fr">
          <img src="../../public/net200/fr.gif">
        </a>
        <a href="?sap-language=de">
          <img src="../../public/net200/de.gif">
        </a>
      </td>
    </tr>
  </table>
...
```

Continued on next page

style.css (Example)

```
body      {background-color:rgb(204,204,255);
           font-family     :Arial}
table     {border:solid;
           border-collapse:collapse;
           empty-cells:show;
           width:100%}
thead     {font:bold}
tr        {border:solid}
td        {border:solid;
           border-width:1px;
           padding:3px;
           background-color:rgb(204,204,204)}
hr        {height:5;
           background-color:rgb(0,0,0)}
.noborder {border:none;
           background-color:rgb(204,204,255)}
.corner   {border:none;
           background-color:rgb(204,204,255);
           text-align:right;
           vertical-align:top}
```

Lesson Summary

You should now be able to:

- Enable translation of text literals in Business Server Pages

Unit Summary

You should now be able to:

- Import MIME objects into the MIME Repository
- Include MIME objects in BSP applications
- Adapt a BSP application without making modifications
- Enable translation of text literals in Business Server Pages

Unit 4

BSP Extensions

Unit Overview

The BSP programming model, which is based on server pages technology, offers developers great freedom with regard to the HTML code they can create, beginning with an empty page to complex applications. However, the repeated creation of complex HTML code is a tedious process, especially if your goal is to design a uniform layout for a larger application.

Here an abstraction technique allows you to simply express the syntax and semantics of specific HTML code sections. This technology is known as **BSP extensions**. A BSP extension contains a collection of **BSP elements**. SAP provides an infrastructure that allows developers to include BSP extensions in their BSP applications.

SAP delivers a range of predefined extensions (for example, **HTML Business for BSP (HTMLB)**), which are available for use in any SAP Web Application Server 6.20 system. You can also define your own extensions to suit specific requirements. You can define your own extensions in the BSP environment using an editor that is included in the development environment (transaction SE80).

Furthermore, in the BSP environment, it is now also possible to develop applications on the basis of a programming model known across programming languages as the Model View Controller (MVC) concept. It enables the separation of the interface, flow logic, and business logic - that is, better modularization. Therefore, the development of reusable components is supported.

Unit Objectives

After completing this unit, you will be able to:

- Use elements of the BSP extension HTMLB to design the layout of Business Server Pages
- Process user input made using BSP elements
- Extract data from a query string
- Adapt the design of a BSP application based on the BSP extension HTMLB

- Render complex pages, consisting of several subpages, on the server side
- Create a new BSP extension
- Create new BSP elements
- Describe the class hierarchy for a BSP element
- Encapsulate combinations of different existing BSP elements in a new BSP element
- Describe the advantages of the MVC programming paradigm over classic BSP programming
- Create and call controllers, views, and models

Unit Contents

Lesson: BSP Extensions: HTMLB

Lesson Overview

In this lesson, you will learn how to use elements of the BSP extension HTMLB to design the layout of Business Server Pages.

Lesson Objectives

After completing this lesson, you will be able to:

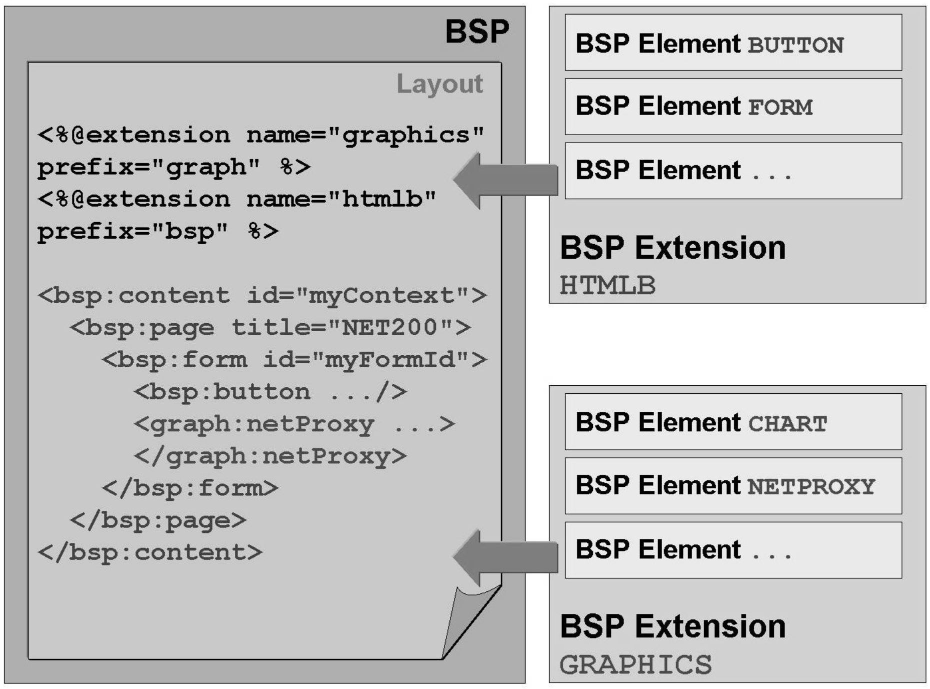
- Use elements of the BSP extension HTMLB to design the layout of Business Server Pages
- Process user input made using BSP elements
- Extract data from a query string
- Adapt the design of a BSP application based on the BSP extension HTMLB
- Render complex pages, consisting of several subpages, on the server side

Business Example

BSP elements are used to design the layout of a Business Server Page. For this purpose, SAP provides a range of libraries (HTMLB, XHTMLB, PHTMLB) that contain elements for implementing various screen elements. The classic HTMLB elements are stored in the in the HTMLB library. The application developer must learn how to use all available BSP elements.

BSP Extensions and BSP Elements

A BSP extension is a collection of BSP elements. BSP extensions represent an individual Repository object type. **BSP elements** are subobjects of this type. In transaction SE80, you can use the Tag Browser to see an overview of the existing BSP extensions. BSP elements appear as subobjects of the individual BSP extensions and can be copied to the layout of a BSP by dragging them to the layout. You can use the elements of several BSP extensions on one BSP. For this purpose, the names of the relevant BSP extensions are made known at the beginning of the layout of a BSP using an appropriate directive.



```
<%@extension name="graphics"
prefix="graph" %>
<%@extension name="htmlb"
prefix="bsp" %>

<bsp:content id="myContext">
  <bsp:page title="NET200">
    <bsp:form id="myFormId">
      <bsp:button .../>
      <graph:netProxy ...>
      </graph:netProxy>
    </bsp:form>
  </bsp:page>
</bsp:content>
```

Figure 52: Layout Design Using BSP Elements

In the BSP context, each element is assigned an ABAP class to represent the element function. If the BSP layout is now interpreted by the BSP compiler, pseudo-code is generated and executed every time the BSP element is called. In each case, one object of the corresponding ABAP class is instantiated and its methods are executed in a fixed sequence. Parameters that are listed in the BSP tag are assigned to appropriate attributes of this object. The function of the element defines the HTML output, which can also be browser-specific.

Using BSP elements offers the following advantages:

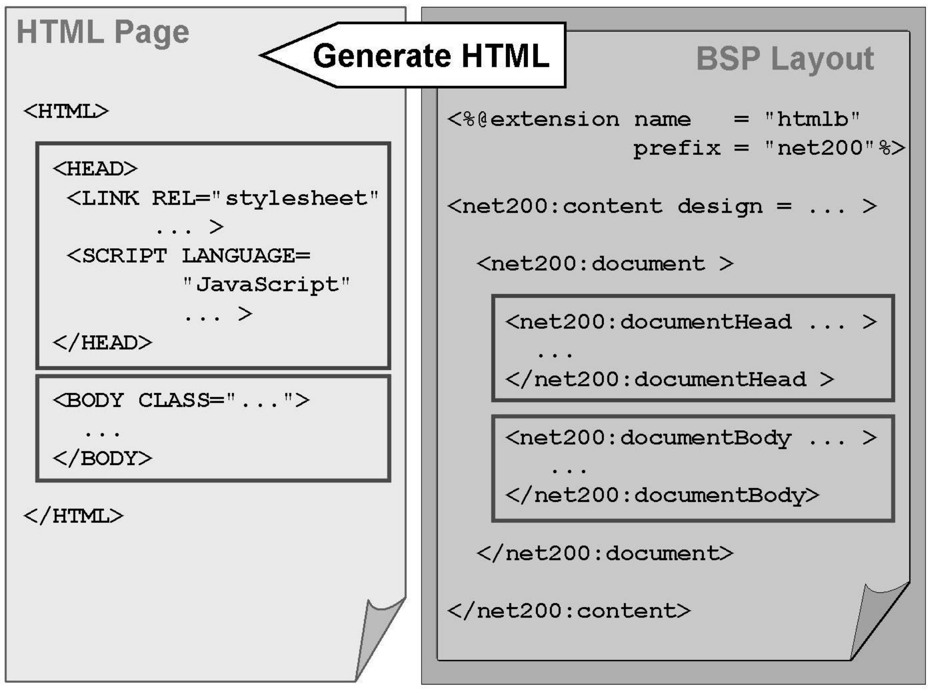
- BSP elements form an abstraction layer. This enables uniform page layout for different HTTP client classes.
- The code included in the ABAP classes for the BSP elements must be developed only once.
- The layout can be parsed and searched for syntax errors because it was designed using BSP elements, not HTML elements, which satisfy the XML standard.
- Achieving a uniform layout, especially for large Web applications, is simplified because the generated code contains appropriate references to style sheets.

The BSP Extension HTMLB

With the SAP Web Application Server Release 6.20, SAP provides many BSP extensions that can be used to design the layout. The BSP extension **HTMLB** contains a range of BSP elements that can be used to create typical screen elements, such as input fields, send buttons, tables, and so on. The BSP extension HTMLB is assigned to the package **SBSPEXT_HTMLB**. This package also contains example Web applications for testing the individual BSP elements and their numerous parameters.

To use the BSP elements for the layout of a BSP, you open the Tag Browser in the navigation area of transaction SE80. The entry HTMLB is listed under the transportable BSP extensions. When you click HTMLB, the system displays a list of the BSP elements that are assigned to the BSP extension HTMLB. You can see the attributes for any BSP element by clicking it. Double-clicking an element opens the documentation for it in a new window. You can copy an element by dragging it from the list to the layout of a BSP. Similarly, you can also copy the parameters by dragging them to the element tag.

There are certain dependencies you should consider when you use BSP elements. For example, if you want to use the tag to generate an input field, the field must be within a tag for generating a form. Every page has an essential basic structure.



```
<HTML>
  <HEAD>
   <LINK REL="stylesheet"
        ... >
   <SCRIPT LANGUAGE=
            "JavaScript"
            ... >
  </HEAD>
  <BODY CLASS="...">
    ...
  </BODY>
</HTML>
```

```
<%@extension name   = "htmlb"
             prefix = "net200"%>

<net200:content design = ... >

  <net200:document >

    <net200:documentHead ... >
      ...
    </net200:documentHead >

    <net200:documentBody ... >
      ...
    </net200:documentBody>

  </net200:document>

</net200:content>
```

Figure 53: Basic Structure of a BSP Based on BSP Elements

At runtime, the system creates the basic structure of an HTML page from the BSP elements.

<%@extension name="htmlb" prefix="net200" %>

This directive enables you to use elements of the BSP extension HTMLB in the corresponding layout. You can choose any prefix you wish and, for identification purposes, it must precede the element name for all elements of this extension (the reason for this is that you can work with several BSP extensions simultaneously and these may contain elements with identical names).

<net200:content ... design="...">...</net200:content>

The topmost element in the hierarchy. The *design* parameter defines which style sheets are referenced. Permitted values are **CLASSIC**, **DESIGN2002**, and **DESIGN2003** (as of SAP Web AS 6.20, SAP_BASIS/SAP_ABA SP34).

<net200:document>...</net200:document>

Is converted to the tag pair `<html> ... </html>`.

<net200:documentHead ...>...</net200:documentHead>

Generates the header of the HTML page. You can insert additional HTML code (for example, JavaScript functions, style definitions).

<net200:documentBody ...>...</net200:documentBody>

Creates the HTML body.

Hint: Instead of the elements `<document>`, `<documentHead>`, and`<documentBody>`, you can also use the element `<page>`. You can then insert additional code into the HTML header using the `<headInclude>` element.

When using `<documentHead>`, you can insert additional code directly between the opening and closing tags. After generating the page, the relevant source code segment appears **before** the automatically generated source code of the HTML header. If you want to swap the order, you can also use the element `<headInclude>` here.

The HTML body is constructed using the other elements. The elements of the BSP extension HTMLB can be split into three groups:

The **first group** of elements is converted to HTML source code at runtime. This code prevents the user from filling in any fields or sending HTTP requests (triggering server-side events). User actions are restricted to the triggering of user-defined JavaScript functions. One example of an element of this group is a simple text (`<textView>`).

The **second group** of elements allows you to trigger HTTP requests. The requests are usually handled by the event handler *OnInputProcessing* for that same page. A framework is provided that enables you to extract the attributes that are related to these elements.

Example: By pressing a send button that is created using the HTMLB element `<button .../>`, you can trigger an HTTP request. An HTML table created using the element `<tableView ...>` even provides several different ways of creating an HTTP request (for example, by selecting a row, clicking on a column header, or scrolling through the rows or columns of the dataset).

An HTTP request is created when the user clicks on the corresponding element. This action is caught using JavaScript, a corresponding query string is created, and the request is "fired". The mouse click is also referred to as an **event**. We differentiate between **server events**, which are predefined by the implementation of the element and usually lead to an HTTP request to the same page, and **client events**, for which the user defines the JavaScript function to be executed.

If an element supports the triggering of server events, this means that information connected to the event is automatically passed with the query string. For example, if the user selects a row in an HTML table, created using the element `<tableView ...>`, and thus triggers a server event, the row numbers of the selected rows are automatically passed to the server. Accessing these attributes from within the source code is supported by a corresponding framework (**event handling**).

The **third group** of elements allows the user to enter data. With the next HTTP request, this data is sent back to the server in the query string. You can usually use auto page attributes to extract the information. However, the data extraction can also be object-oriented (in some cases, it must be). This usually occurs at the time of *OnInputProcessing* (**data extraction**).

Example: You can use the `<textEdit ...>` element to create an area for inputting several lines of text. If the entered text is to be sent back to the server, this is usually done after a Send button is clicked.

Hint: Some elements can trigger events and receive user input.

Example: In the case of a checkbox, the developer can decide whether checking the checkbox triggers an event or the information is only sent to the server after a send button has been pressed.

The following lists all the elements that do not permit interaction and hence belong to the first of the above groups:

HTMLB Elements that Neither Trigger Server Events nor Permit User Input

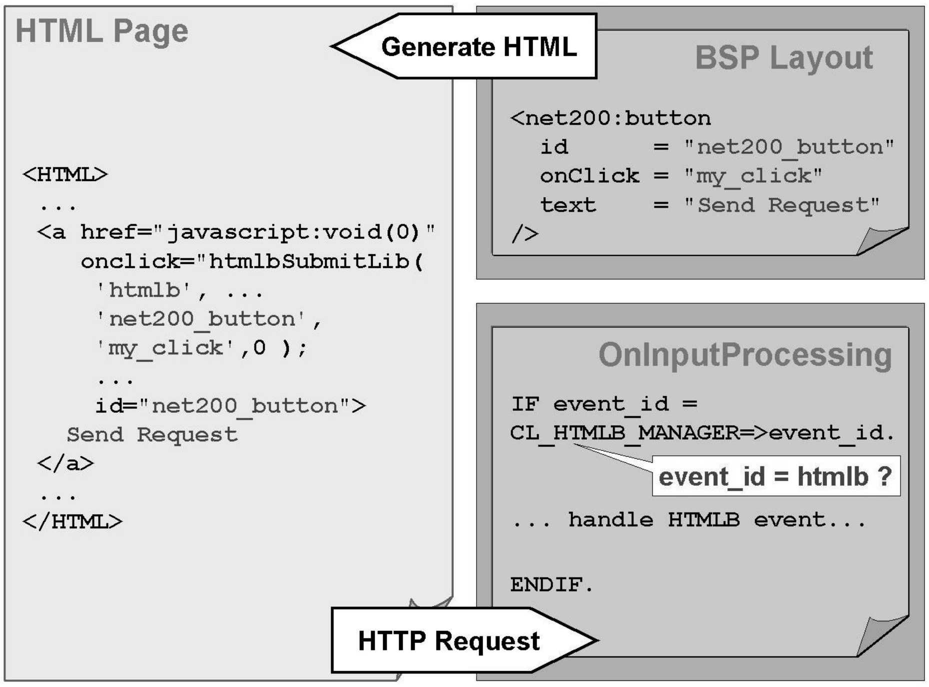
HTMLB element	Dependent elements
content	document page
document	documentHead documentBody
page	headInclude form All other elements for the HTML body.
documentHead	headInclude
documentBody	form All other elements for the HTML body.
form	All other elements for the HTML body.
chart	
gridLayout	gridLayoutCell
group	groupHeader groupBody
itemList	listItem
label	
textView	

Event Handling

There are two ways of analyzing the incoming events:

- The source code for the event handling is stored directly in the event handler *OnInputProcessing*
- The request data is passed to a specially created event handler class and handled there.

The following takes a closer look at the first technique.



```
<HTML>
 ...
 <a href="javascript:void(0)"
    onclick="htmlbSubmitLib(
     'htmlb', ...
     'net200_button',
     'my_click',0 );
     ...
     id="net200_button">
   Send Request
 </a>
 ...
</HTML>
```

```
<net200:button
  id      = "net200_button"
  onClick = "my_click"
  text    = "Send Request"
/>
```

```
IF event_id =
CL_HTMLB_MANAGER=>event_id.

... handle HTMLB event...

ENDIF.
```

Figure 54: Event Handling: OnInputProcessing (1)

If the HTMLB element can trigger server events itself, the field *event_id* is filled with the value **htmlb**. Therefore, the event handling usually starts with the system checking whether the *event_id* field has this value. If the field does have the value, the system next ascertains which element or user action triggered the event. For this, all attributes connected with the HTMLB element creating the event must be extracted from the query string. Different attributes are returned for each attribute, but there are four attributes that all elements have. These are:

server_event

Value assigned to the parameter responsible for triggering server events (for example, onClick=**my_click**).

id

Value assigned to the *id* parameter of the HTMLB element (for example, id=**net200_button**).

name

Type of HTMLB element that triggered the event (for example, **button**).

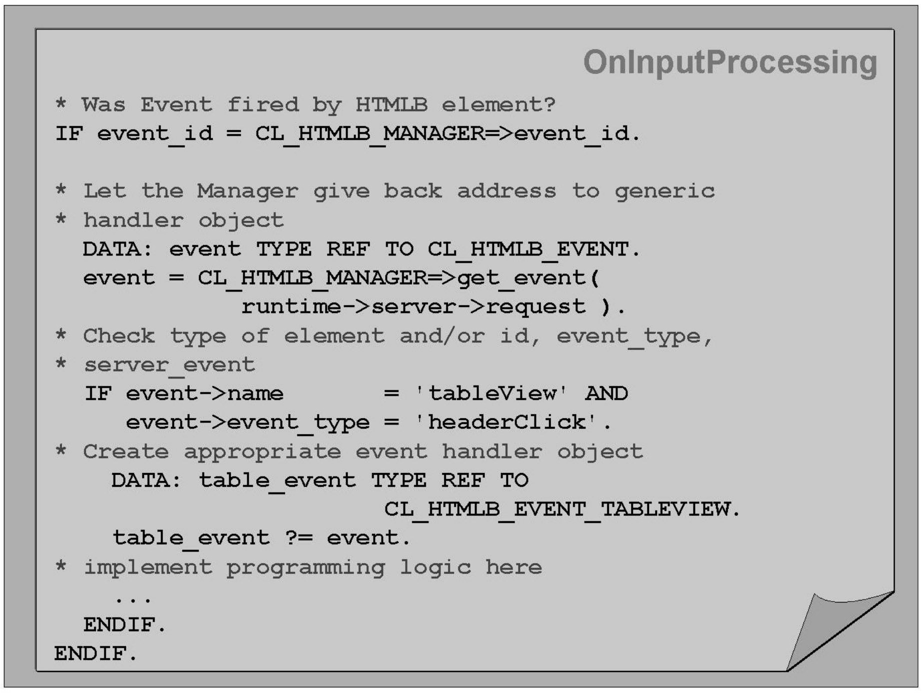
event_type

Type of server event (for example, **Click**).

The **server_event** attribute can, for example, be used to group objects that are to be linked with the same subsequent action (send buttons or hyperlinks that are in different positions on the page but are to have the same result).

To be able to read the element-specific attributes, an object of the corresponding handler class is then instantiated. Such a class exists for every HTMLB element that can trigger server events and returns special attributes. This class enables access to the special attributes.

From a technical point of view, the BSP runtime environment returns the address of a handler object on your request. This handler object generally enables access to all attributes of the element creating the event. The reference variable (to which the address is assigned) must merely be typed according to the correct handler class, but this information is usually not known until runtime. However, there is one common superclass for all handler classes. If the reference variable is typed according to this superclass, no error can occur when assigning the address to the handler object. Using the superclass, you can analyze the four common attributes and, based on this, create a further reference variable with the type of the correct handler class. The address of the generic reference variable is then assigned to the correctly typed reference variable (Widening Cast).


OnInputProcessing

```
* Was Event fired by HTMLB element?
IF event_id = CL_HTMLB_MANAGER=>event_id.

* Let the Manager give back address to generic
* handler object
  DATA: event TYPE REF TO CL_HTMLB_EVENT.
  event = CL_HTMLB_MANAGER=>get_event(
               runtime->server->request ).
* Check type of element and/or id, event_type,
* server_event
  IF event->name       = 'tableView' AND
     event->event_type = 'headerClick'.
* Create appropriate event handler object
    DATA: table_event TYPE REF TO
                         CL_HTMLB_EVENT_TABLEVIEW.
    table_event ?= event.
* implement programming logic here
    ...
  ENDIF.
ENDIF.
```

Figure 55: Event Handling: OnInputProcessing (2)

The following table provides an overview of all HTMLB elements that support the triggering of server events. The name of each HTMLB element is identical to the generic attribute *name*. The second column contains the server events that can be triggered (the name of the parameter of the HTMLB element and the value of the *event_type* attribute, which it sets). Finally, the third column contains the names of the corresponding handler classes, which enable you to access the element-specific attributes.

HTMLB elements that support the triggering of server events

HTMLB element / *name*	**Parameter ->*event_type***	**event_handler *CL_HTMLB_EVENT_..***
breadCrumb	onClick -> click	BREADCRUMB
breadCrumbItem	-	-
button	onClick -> click	BUTTON
checkBoxGroup	-	-
checkbox	onClick -> click	CHECKBOX
dateNavigator	onDayClick -> dayClick onMonthClick -> monthClick onWeekClick -> weekClick onNavigate -> previous / next	DATENAVIGATOR
days	-	-
week	-	-
month	-	-
dropdownListBox	onSelect -> select	SELECTION
image	onClick -> click	-
link	onClick -> click	LINK
radioButtonGroup	-	-
radioButton	onClick -> click	RADIOBUTTON
tabStrip	-	-
tabStripItem	onSelect -> select	TABSTRIP
tabStripHeader	-	-
tabStripBody	-	-
tableView	onFilter -> filter	TABLEVIEW

HTMLB element *I name*	**Parameter ->*event_type***	**event_handler** *CL_HTMLB_EVENT_..*
	onHeaderClick -> headerClick	
	onNavigate -> navigate	
	onRowSelection -> rowSelection	
tableViewColumns	-	-
tableViewColumn	onCellClick -> cellClick	TABLEVIEW
	onItemClick -> itemClick	
tray	onCollapse -> collapse	TRAY
	onEdit -> edit	
	onExpand -> expand	
	onRemove -> remove	
trayBody	-	-
tree	onTreeClick -> click	TREE
treeNode	onNodeClick -> click	TREE

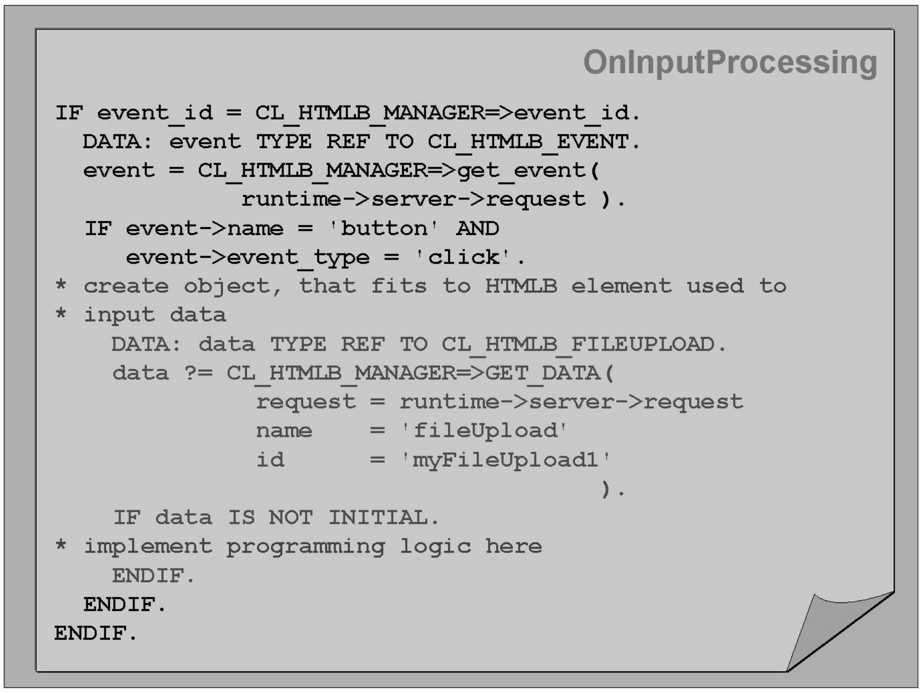
Data Extraction

If the user can make entries using the HTMLB element, this raises the question of how the data can be transferred from the query string to page attributes of the BSP. In most cases, an auto page attribute can be used, whose name matches the ID of the relevant HTMLB element. However, this is not possible in some cases. Example: Information sent by the user to the server using the ***fileUpload*** element (application data, size of object, file type).

Therefore, a concept exists for all elements the user can use to make entries that enables the extraction of the relevant data from the query string.

Similarly to event handling, the user can request the address of an object from the BSP runtime, which generally enables access to all data in the query string. The restriction to data of a specific form field is achieved by passing the type and ID of the element with the relevant method call using the interface. The returned address is assigned to a reference variable that must be correctly typed (to suit the HTMLB element whose data is to be extracted).

Example: Using a *fileUpload* element, the user can select a file to be loaded to the server with the next HTTP request. However, this element does not support the triggering of server events. Therefore, the form is sent using an HTMLB *button*. The static method *GET_DATA* of the HTMLB Manager is used to extract the data of the *fileUpload*. The address is assigned to a reference variable of the type *CL_HTMLB_FILEUPLOAD*. This reference variable can then be used to access all the attributes of this object.


OnInputProcessing

```
IF event_id = CL_HTMLB_MANAGER=>event_id.
  DATA: event TYPE REF TO CL_HTMLB_EVENT.
  event = CL_HTMLB_MANAGER=>get_event(
              runtime->server->request ).
  IF event->name = 'button' AND
     event->event_type = 'click'.
* create object, that fits to HTMLB element used to
* input data
    DATA: data TYPE REF TO CL_HTMLB_FILEUPLOAD.
    data ?= CL_HTMLB_MANAGER=>GET_DATA(
               request = runtime->server->request
               name    = 'fileUpload'
               id      = 'myFileUpload1'
                                   ).
    IF data IS NOT INITIAL.
* implement programming logic here
    ENDIF.
  ENDIF.
ENDIF.
```

Figure 56: Data Extraction: OnInputProcessing

HTMLB elements for which data extraction is supported

HTMLB element / *name*	Element class (*CL_HTMLB_..*)	Extractable attributes / also using auto page attribute
checkBox	CHECKBOX	checked / X
dropdownListBox	DROPDOWNLIST-BOX	selection / X
fileUpload	FILEUPLOAD	file_name / file_content / file_length / file_content_type /
inputField	INPUTFIELD	value / X
listBox	LISTBOX	selections / +

HTMLB element / *name*	**Element class (*CL_HTMLB_..*)**	**Extractable attributes / also using auto page attribute**
radioButtonGroup	RADIOBUTTON-GROUP	selection / X
tabStrip	TABSTRIP	selection / X
textEdit	TEXTEDIT	text / X
tray	TRAY	iscollapsed /

+ If only one entry is selected from the list box, it can also be extracted using an auto page attribute. If several entries are selected, the data extraction concept is required.

If the data extraction is executed as described above, you do not need automatic page attributes. Instead, the data is read directly from the HTTP request.

Adjusting the Design

The theme concept is not used for adjusting the design in the HTMLB environment. Instead, the path specification for the included style sheets is influenced by specifying the parameter *themeRoot* of the *content* tag. Proceed as follows:

1. For each desing *design_1 ... design_n*, you must first copy the original SAP style sheets and the graphics and icons referenced in them in the MIME Repository.

   To do this, you must first create a directory *designs* in the MIME Repository as a subfolder of the node of any BSP application *bsp_app*. Then define a further subnode for each design (*design_1 ... design_n*).

   The original objects are in the folders *sap/public/bc/htmlb* and *sap/public/bc/xhtmlb*. Copy these two directories with all their contents (subdirectories and MIME objects) to the target node created in the last step (*bsp_app/design/design_1 ... bsp_app/design/design_n*).

2. To reference the copied style sheets, extend the parameter *themeRoot* of the *content* tag in the layout of the BSP as follows (here for the design *design_1*):

```
<htmlb:content design    = "design2002"
               themeRoot = "/sap/bc/bsp/sap/bsp_app/
                              designs/design_1">
```

Specifying the ***design*** **parameter** affects which of the style sheets stored in the node */sap/bc/bsp/sap/bsp_app/designs/design_1* is referenced. The tag processing is also influenced: Depending on the value of the *design* parameter, different class references are generated in the resulting HTML source code.

3. Finally, the copied style sheets, images, and icons in the new designs must be adjusted. To do this, you must first ascertain the names of the style sheets in the HTML source code and then navigate to them in the MIME Repository. Using the right mouse button, you can open and edit the CSS.

   In the HTML source code, you must now determine the names of the classes that are responsible for the design of specific HTML elements. The classes are defined in one of the opened CSS. After adjusting the relevant style sheet attributes and saving the CSS in the MIME Repository, you can check the changes by restarting the application (in the stateless case, refreshing the page is suffient). To do this, you must remove any buffered versions of the edited objects from the client cache and the server cache.

Demos, Documentation, and Notes

More information on the topics covered in this lesson is available from the following sources:

The **Web Application Builder** (SE80 -> Tag Browser -> BSP Extensions -> Transportable) features extensive documentation on the BSP extensions HTMLB, XHTMLB, and PHTMLB. XHTMLB and PHTMLB are extensions that provide further useful elements for structuring Web pages.

The online documentation contains detailed documentation on the SAP Web Application Server and BSP applications. This is accessed in the SAP Help Portal as follows: http://help.sap.com -> SAP NetWeaver -> Release '04 -> SAP NetWeaver -> Application Platform -> ABAP Technology -> UI Technology -> WEB UI Technology -> Business Server Pages

As of Release 6.20, the SAP Web AS features the **BSP applications** *HTMLB_SAMPLES* (HTMLB), *SBSPEXT_XHTMLB* (XHTMLB), and *SBSPEXT_PHTMLB* (PHTMLB). These allow you to edit the parameters of the individual BSP elements and directly test the effect on element generation.

In the **Software Developer Network (SDN)**, there is a discussion forum for current BSP topics. Individual questions are partly dealt with by the BSP development team. The forum is accessed via: http://sdn.sap.com -> menu entry *Forums* -> Forums -> SDN Forums -> Web Application Server -> Business Server Pages.

In the **OSS**, you can create problem messages and find notes under the area *BC-BSP*.

Exercise 8: BSP Extension HTMLB

Exercise Objectives

After completing this exercise, you will be able to:

- Use the BSP extension HTMLB to define the layout of BSPs

Business Example

In a number of steps, you will define the layout of individual pages of a BSP application using elements of the BSP extension HTMLB.

Task:

Continue to work with your BSP application or copy the model solution from the last exercise. To do this, copy your BSP application ZNET200_##_07 or the BSP application NET200_S_07, giving it the name ZNET200_##_13, where ## is your group number. Adhere to the names given and always enter single-digit group numbers that begin with a leading zero as in 0#. Model solutions for this exercise: NET200_S_08 (exercise 1) ... NET200_S_13 (exercise 6 with optional parts).

1. Create the first page of your BSP application using elements of the BSP extension HTMLB. Proceed as follows:

   Copy the start page *public/start.htm* to the page *public/start_htmlb.htm*. The following specifications refer to the newly created page. Delete the layout. First, define the frame of the page using the elements *content*, *document*, *documentHead*, and *documentBody*. Choose *DESIGN2002* as the design for the page. Copy the title from the `<head>...</head>` section of the page *public/start.htm*. Activate and test the new page.

   Hint: To be translatable, texts must be created as alias texts. OTR long texts cannot be used with BSP extensions.

2. Output elements:

   Convert the pure output elements using HTMLB elements. Start with the page header. Copy the page *header.htm* to the page *header_htmlb.htm*. Delete the layout on the target page. Create the layout (graphic with adjacent text) using the elements *image* and *textView*. A horizontal line or rule across the page should appear below these elements. To align the graphic, the text, and the line, you can use the elements *gridLayout* and *gridLayoutCell*. Activate the new page header.

Continued on next page

Now include the page header in the page *public/start_htmlb.htm*. Use HTMLB elements to include the following HTML elements on the start page: The text "Last Minute Flights:" and links to the same page in different languages should appear below the page header. Refer to the model solution **NET200_S_09**. Use the HTMLB elements *textView*, *link*, and (if the links are to appear as images) *image* to design the layout of the page. To align the elements, you can use the elements *gridLayout* and *gridLayoutCell*.

Hint: If an image is to be used as a hyperlink, you cannot use the *text* attribute of the *link* element. The HTMLB element *image* must be within the opening and closing tags of the HTMLB element *link*.

3. Form:

On the page *public/start_htmlb.htm*, define a form with input fields for the departure city (*depa*) and destination city (*dest*). Define two more input fields for a date interval (*date_low* and *date_high*). Define a date interval for these fields. Limit the length of the date fields to that required for entering a date. Include an input help option for the date fields that allows the user to select dates from a calendar. All fields should be preceded by language-specific field labels.

Include the hidden field *travel_ag*, which is inserted using a page fragment, using HTMLB elements. Copy the page fragment *hidden.htm* to the page fragment *hidden_htmlb.htm*. Delete the layout of the new page. Define a hidden form field for the attribute *travel_ag* using the BSP extension HTMLB. Now activate the page fragment and include it in the form of the page *public/start_htmlb.htm*.

Hint: Use the HTMLB elements *form*, *label*, and *inputField* to align the elements *gridLayout* and *gridLayoutCell*.

Hint: Do not define a Send button here.

4. Event handling:

Define a button in the form of the page *public/start_htmlb.htm*. Make the button text translatable. Pushing the button is to trigger the server event *click*.

Navigate to the event handler OnInputProcessing. Find out which action is responsible for the execution of the event handler. To do so, examine the contents of the *event_id* field. If the field value shows

Continued on next page

that an HTMLB element triggered the event, find out what type of element it was. It is not necessary to examine the attributes connected with the event, because only one button exists on the page. If you are certain that the button on the form triggered the processing of the event handler, navigate to the subsequent page. The name of the next page is *public/flights_htmlb.htm*. Pass on the departure city, destination city, and the date interval in the HTTP redirect. For this purpose, adapt the methods *navigation->set_parameter* for the date information.

Copy the page *public/flights.htm* to the page *public/flights_htmlb.htm*. Adapt the page attributes to receive the date information. Navigate to the event handler OnInitialization. Previously, the function module *FIT_IN_BAPI_FLCONN_GETLIST* was used here for format conversion. Since the data is now no longer split into day, month, and year, but is a value instead, the format conversion must be adapted. Use the function module *FIT_IN_BAPI_FLCONN_GETLIST2* for this. Finally, ensure that the BSP *public/start_htmlb.htm* reappears if the user presses the send button to navigate back.

5. Tables:

Now create the table of last-minute flights on the page *public/start_htmlb.htm* using the BSP extension HTMLB. Display all columns and set the width of the table to 100% of the page width. Hide the footer with the scroll buttons.

Hint: Use the HTMLB element *tableView*.

If you have the time, use HTMLB elements to also create the table in the layout of the BSP *public/flights_htmlb.htm*. For this purpose you must convert the entire layout of the page to HTMLB elements. Set the table width to 100% of the page width. Restrict the number of simultaneously displayed table rows to 10 and make it possible for the user to scroll through the rows. Only display the columns that were previously displayed. Make it possible for the user to sort the table by flight date, departure city, and destination city.

Hint: Use the HTMLB elements *tableView*, *tableViewColumns*, and *tableViewColumn*.

Hint: To allow the user to sort the table, you must, amongst other things, enable the triggering of the server event *headerClick*.

Continued on next page

To ensure that you can still use the connection number to navigate to the next page, you must add a column to the corresponding internal table. This column must contain the relevant URL for navigating to the next page. Proceed as follows:

Create an additional table type that contains all the previous columns (line type *BAPISCODAT*) and another column of the type STRING for the URL. The name of this additional column is **URL**.

Hint: To copy the fields of a row type t_wa2 in a row type t_wa1, use the following statements:

```
TYPES: BEGIN OF t_wa1,
         ... .
INCLUDE TYPE t_wa2.
TYPES:  ....
       END OF t_wa1.
```

Create an additional page attribute for this extended table type. Navigate to the event handler *OnInitialization*. Extend the source code: The flight connection data, which was read using the application class *GET_FLIGHT_LIST*, must be transferred to the extended internal table. In the URL column, store the character string that was previously defined as the target for the link in the layout of the public/flights.htm page. Construct the corresponding query string using the ABAP statement *CONCATENATE*.

Navigate to the page layout. Set the type of the column *FLIGHTCONN* to the value **LINK** and assign the name of the column that contains the URL (*URL*) to the attribute **linkColumnKey**.

To eventually return to the start page by pressing the send button, you must adjust the source code in the event handler *OnInputProcessing* accordingly. To do this, refer to the last part of the exercise.

6. If you have the time to, create another layout.

   Using elements of the BSP extension HTMLB, also create the layout of the details page. Copy the page *public/details.htm* to the page *public/details_htmlb.htm*. Delete the layout on the new page and create the layout using the BSP extension HTMLB. You must also adapt the source code of the event handler *OnInputProcessing*. Ensure that the links on the page *public/flights_htmlb.htm* now take you to the page *public/details_htmlb.htm*. Refer to the relevant page in the model solution **NET200_S_13**.

Solution 8: BSP Extension HTMLB

Task:

Continue to work with your BSP application or copy the model solution from the last exercise. To do this, copy your BSP application ZNET200_##_07 or the BSP application NET200_S_07, giving it the name ZNET200_##_13, where ## is your group number. Adhere to the names given and always enter single-digit group numbers that begin with a leading zero as in 0#. Model solutions for this exercise: NET200_S_08 (exercise 1) ... NET200_S_13 (exercise 6 with optional parts).

1. Create the first page of your BSP application using elements of the BSP extension HTMLB. Proceed as follows:

   Copy the start page *public/start.htm* to the page *public/start_htmlb.htm*. The following specifications refer to the newly created page. Delete the layout. First, define the frame of the page using the elements *content*, *document*, *documentHead*, and *documentBody*. Choose

Continued on next page

DESIGN2002 as the design for the page. Copy the title from the `<head>...</head>` section of the page *public/start.htm*. Activate and test the new page.

Hint: To be translatable, texts must be created as alias texts. OTR long texts cannot be used with BSP extensions.

a) The BSP elements and their attributes are copied from the Tag Browser using Drag&Drop. The attribute values are added manually. If you double-click a BSP element, a dialog box appears that describes the element and the attributes. The model solution for this section is **NET200_S_08**.

public/start_htmlb.htm - Layout

```
<%@extension name="htmlb" prefix="htmlb" %>

<htmlb:content design="DESIGN2002" >
  <htmlb:document>
    <htmlb:documentHead title="<%=otr(NET200/HEADERTEXT)%>" >
    </htmlb:documentHead>
    <htmlb:documentBody>
    </htmlb:documentBody>
  </htmlb:document>
</htmlb:content>
```

2. Output elements:

Convert the pure output elements using HTMLB elements. Start with the page header. Copy the page *header.htm* to the page *header_htmlb.htm*. Delete the layout on the target page. Create the layout (graphic with adjacent text) using the elements *image* and *textView*. A horizontal line or rule across the page should appear below these elements. To align the graphic, the text, and the line, you can use the elements *gridLayout* and *gridLayoutCell*. Activate the new page header.

Now include the page header in the page *public/start_htmlb.htm*. Use HTMLB elements to include the following HTML elements on the start page: The text “Last Minute Flights:” and links to the same page in different languages should appear below the page header. Refer to the model solution **NET200_S_09**. Use the HTMLB elements *textView*,

Continued on next page

link, and (if the links are to appear as images) *image* to design the layout of the page. To align the elements, you can use the elements *gridLayout* and *gridLayoutCell*.

Hint: If an image is to be used as a hyperlink, you cannot use the *text* attribute of the *link* element. The HTMLB element *image* must be within the opening and closing tags of the HTMLB element *link*.

a) The BSP elements and their attributes are copied from the Tag Browser using Drag&Drop. The attribute values are added manually. If you double-click a BSP element, a dialog box appears that describes the element and the attributes. The model solution for this section is **NET200_S_09**.

header_htmlb.htm - Layout

```
<%@extension name="htmlb" prefix="htmlb" %>

<%-- Alignment of elements                              --%>
<htmlb:gridLayout columnSize  = "2"
                  rowSize     = "2"
                  cellSpacing = "10"
                  width       = "100%"
                  style       = "TRANSPARENT" >


  <%-- Cell 1 in Row 1                                  --%>
  <htmlb:gridLayoutCell columnIndex = "1"
                        rowIndex    = "1" >
    <%-- Display image                                  --%>
    <htmlb:image src="../../PUBLIC/NET200/Reisen01.jpg" />
  </htmlb:gridLayoutCell>


  <%-- Cell 2 in Row 1                                  --%>
  <htmlb:gridLayoutCell columnIndex       = "2"
                        rowIndex          = "1"
                        verticalAlignment = "MIDDLE" >
    <%-- Display text                                   --%>
    <htmlb:textView text   = "<%= otr(net200/headertext) %>"
                    layout = "native"
                    design = "header1" />
```

Continued on next page

```
  </htmlb:gridLayoutCell>


  <%-- Horizontal ruler in Row 2                        --%>
  <htmlb:gridLayoutCell columnIndex = "1"
                        rowIndex    = "2"
                        colSpan     = "2" >
    <hr>
  </htmlb:gridLayoutCell>

</htmlb:gridLayout>
```

public/start_htmlb.htm - Layout

```
<%@extension name="htmlb" prefix="htmlb" %>

...
  <%-- Begin of BODY section                               --%>
    <htmlb:documentBody>

    <%-- Include header file                               --%>
      <%@include file="header_htmlb.htm" %>

    <%-- Element alignment                                 --%>
      <htmlb:gridLayout columnSize  = "4"
                        rowSize     = "1"
                        width       = "100%"
                        cellSpacing = "10" >

      <%-- Element 1 in row                                --%>
        <htmlb:gridLayoutCell columnIndex = "1"
                              rowIndex    = "1" >
          <htmlb:textView text   = "<%= otr(NET200/LASTMINUTE) %>"
                          design = "header2" />
        </htmlb:gridLayoutCell>

      <%-- Element 2 in row                                --%>
        <htmlb:gridLayoutCell columnIndex         = "2"
                              rowIndex            = "1"
                              horizontalAlignment = "RIGHT"
                              width               = "35" >
          <htmlb:link id        = "link1"
```

Continued on next page

```
                    reference = "?sap-language=en" >
          <htmlb:image src="../../public/net200/gb.gif" />
        </htmlb:link>
      </htmlb:gridLayoutCell>

    <%-- Element 3 in row                                    --%>
      <htmlb:gridLayoutCell columnIndex          = "3"
                            rowIndex             = "1"
                            horizontalAlignment  = "RIGHT"
                            width                = "35" >
        <htmlb:link id        = "link1"
                    reference = "?sap-language=fr" >
          <htmlb:image src="../../public/net200/fr.gif" />
        </htmlb:link>
      </htmlb:gridLayoutCell>

    <%-- Element 4 in row                                    --%>
      <htmlb:gridLayoutCell columnIndex          = "4"
                            rowIndex             = "1"
                            horizontalAlignment  = "RIGHT"
                            width                = "35"  >
        <htmlb:link id        = "link1"
                    reference = "?sap-language=de" >
          <htmlb:image src="../../public/net200/de.gif" />
        </htmlb:link>
      </htmlb:gridLayoutCell>

    </htmlb:gridLayout>

  </htmlb:documentBody>

...
```

3. Form:

   On the page *public/start_htmlb.htm,* define a form with input fields for the departure city (*depa*) and destination city (*dest*). Define two more input fields for a date interval (*date_low* and *date_high*). Define a date interval for these fields. Limit the length of the date fields to that required for entering a date. Include an input help option for the date fields that allows the user to select dates from a calendar. All fields should be preceded by language-specific field labels.

Continued on next page

Include the hidden field *travel_ag*, which is inserted using a page fragment, using HTMLB elements. Copy the page fragment *hidden.htm* to the page fragment *hidden_htmlb.htm*. Delete the layout of the new page. Define a hidden form field for the attribute *travel_ag* using the BSP extension HTMLB. Now activate the page fragment and include it in the form of the page *public/start_htmlb.htm*.

Hint: Use the HTMLB elements *form*, *label*, and *inputField* to align the elements *gridLayout* and *gridLayoutCell*.

Hint: Do not define a Send button here.

a) The BSP elements and their attributes are copied from the Tag Browser using Drag&Drop. The attribute values are added manually. If you double-click a BSP element, a dialog box appears that describes the element and the attributes. Using Drag&Drop, copy the new page fragment from the navigation window to the layout of the page *public/start_htmlb.htm*. The model solution for this section is **NET200_S_10**.

public/start_htmlb.htm - Layout

```
<%@extension name="htmlb" prefix="htmlb" %>
...
<htmlb:documentBody>
  ...
<%-- Begin of HTML form                          --%>
  <htmlb:form>

<%-- Inclusion of hidden fields                  --%>
    <%@ include file = "hidden_htmlb.htm" %>

<%-- Element alignment                           --%>
    <htmlb:gridLayout columnSize  = "4"
                      rowSize     = "3"
                      width       = "100%"
                      cellSpacing = "10" >

 <%-- first row: departure city                --%>
      <htmlb:gridLayoutCell columnIndex = "1"
                            rowIndex    = "1" >
        <htmlb:label for  = "depa"
```

Continued on next page

```
                  text = "<%= otr(net200/cityfrom) %>" />
     </htmlb:gridLayoutCell>
     <htmlb:gridLayoutCell columnIndex = "2"
                           rowIndex    = "1"
                           colSpan     = "3" >
       <htmlb:inputField id="depa" />
     </htmlb:gridLayoutCell>

<%-- second row: destination city              --%>
     <htmlb:gridLayoutCell columnIndex = "1"
                           rowIndex    = "2" >
       <htmlb:label for  = "dest"
                  text = "<%= otr(net200/cityto) %>" />
     </htmlb:gridLayoutCell>
     <htmlb:gridLayoutCell columnIndex = "2"
                           rowIndex    = "2"
                           colSpan     = "3" >
       <htmlb:inputField id="dest" />
     </htmlb:gridLayoutCell>

<%-- third row: date range selection           --%>
     <htmlb:gridLayoutCell columnIndex = "1"
                           rowIndex    = "3" >
       <htmlb:textView text="<%= otr(net200/flightdate) %>" />
     </htmlb:gridLayoutCell>
     <htmlb:gridLayoutCell columnIndex = "2"
                           rowIndex    = "3" >
       <htmlb:inputField id       = "date_low"
                         type     = "DATE"
                         showHelp = "TRUE"
                         size     = "10"
                         value    = "20030101" />
     </htmlb:gridLayoutCell>
     <htmlb:gridLayoutCell columnIndex = "3"
                           rowIndex    = "3" >
       <htmlb:textView text="<%= otr(net200/to) %>" />
     </htmlb:gridLayoutCell>
     <htmlb:gridLayoutCell columnIndex = "4"
                           rowIndex    = "3" >
       <htmlb:inputField id       = "date_high"
                         type     = "DATE"
                         showHelp = "TRUE"
                         size     = "10"
                         value    = "20040101"/>
     </htmlb:gridLayoutCell>
```

Continued on next page

```
      </htmlb:gridLayout>
    </htmlb:form>
  </htmlb:documentBody>
  ...
```

hidden_htmlb.htm - Layout

```
<%@extension name="htmlb" prefix="htmlb" %>
<htmlb:inputField visible = " "
                  id      = "travel_ag"
                  value   = "<%= travel_ag %>" />...
```

4. Event handling:

   Define a button in the form of the page *public/start_htmlb.htm*. Make the button text translatable. Pushing the button is to trigger the server event *click*.

   Navigate to the event handler OnInputProcessing. Find out which action is responsible for the execution of the event handler. To do so, examine the contents of the *event_id* field. If the field value shows that an HTMLB element triggered the event, find out what type of element it was. It is not necessary to examine the attributes connected with the event, because only one button exists on the page. If you are certain that the button on the form triggered the processing of the event handler, navigate to the subsequent page. The name of the next page is *public/flights_htmlb.htm*. Pass on the departure city, destination city, and the date interval in the HTTP redirect. For this purpose, adapt the methods *navigation->set_parameter* for the date information.

   Copy the page *public/flights.htm* to the page *public/flights_htmlb.htm*. Adapt the page attributes to receive the date information. Navigate to the event handler OnInitialization. Previously, the function module *FIT_IN_BAPI_FLCONN_GETLIST* was used here for format conversion. Since the data is now no longer split into day, month, and year, but is a value instead, the format conversion must be adapted. Use the function module *FIT_IN_BAPI_FLCONN_GETLIST2* for this. Finally, ensure that the BSP *public/start_htmlb.htm* reappears if the user presses the send button to navigate back.

Continued on next page

a) The BSP elements and their attributes are copied from the Tag Browser using Drag&Drop. The attribute values are added manually. If you double-click a BSP element, a dialog box appears that describes the element and the attributes. The model solution for this section is **NET200_S_11**.

public/start_htmlb.htm - Layout

```
    ...
<%-- Element alignment                           --%>
  <htmlb:gridLayout columnSize  = "4"
                    rowSize     = "4"
                    width       = "100%"
                    cellSpacing = "10" >
    ...
    <htmlb:inputField id       = "date_high"
                      type     = "DATE"
                      showHelp = "TRUE"
                      size     = "10"
                      value    = "20040101" />
  </htmlb:gridLayoutCell>
  <htmlb:gridLayoutCell columnIndex = "1"
                        rowIndex    = "4"
                        colSpan     = "4" >

      <%-- Submit button                         --%>
      <htmlb:button text    = "<%= otr(net200/display_flights) %>"
                    onClick = "flights" />

    </htmlb:gridLayoutCell>
  </htmlb:gridLayout>
</htmlb:form>
...
```

public/start_htmlb.htm - OnInputProcessing

```
* event handler for checking and processing user input and
* for defining navigation

* Is event triggerd by HTMLB element?
```

Continued on next page

```
IF event_id = cl_htmlb_manager=>event_id.

* What kind of element did trigger event
  DATA: event TYPE REF TO cl_htmlb_event.
  event = cl_htmlb_manager=>get_event( runtime->server->request ).
  IF event->name = 'button' AND event->server_event = 'flights'.

* No further investigation of event related attributes
*   DATA: button_event TYPE REF TO cl_htmlb_event_button.
*   button_event ?= event.

* Setting attributes for the next page (Form fields entries)
    navigation->set_parameter( name = 'depa' ).
    navigation->set_parameter( name = 'dest'  ).
    navigation->set_parameter( name = 'date_low' ).
    navigation->set_parameter( name = 'date_high' ).
    navigation->set_parameter( name = 'travel_ag' ).
* Navigation to the next page
    navigation->goto_page( 'FLIGHTS_HTMLB.HTM' ).

  ENDIF.
ENDIF.
```

public/flights_htmlb.htm - PAGE ATTRIBUTES

The following page attributes that are no longer required

Attribute	Auto	Type	Reference type	Description
day_low	X	TYPE	String	from: day
day_high	X	TYPE	String	to: day
month_low	X	TYPE	String	from: month
month_high	X	TYPE	String	to: month
year_low	X	TYPE	String	from: year
year_high	X	TYPE	String	to: year

Continued on next page

New page attributes

Attribute	Auto	Type	Reference type	Description
date_low	X	TYPE	D	from: date
date_high	X	TYPE	D	to: date

public/flights_htmlb.htm - OnInitialization

```
* event handler for data retrieval

************************************************
* Prepare the form fields for the BAPI Interface
************************************************
DATA: dest_from TYPE bapiscodst,
      dest_to   TYPE bapiscodst,
      it_date   TYPE TABLE OF bapiscodra.

*****************************************
* Data Conversion for Method Call
*****************************************
CALL FUNCTION 'FIT_IN_BAPI_FLCONN_GETLIST2'
  EXPORTING
    start       = depa
    end         = dest
    date_low    = date_low
    date_high   = date_high
  IMPORTING
    dest_from   = dest_from
    dest_to     = dest_to
  TABLES
    date        = it_date.

...
```

public/flights_htmlb.htm - OnInputProcessing

```
CASE event_id.
  WHEN 'back'.
* Navigation to the previous page
```

Continued on next page

```
        navigation->goto_page( 'START_HTMLB.HTM' ).
    ENDCASE.
```

5. Tables:

   Now create the table of last-minute flights on the page *public/start_htmlb.htm* using the BSP extension HTMLB. Display all columns and set the width of the table to 100% of the page width. Hide the footer with the scroll buttons.

   Hint: Use the HTMLB element *tableView*.

   If you have the time, use HTMLB elements to also create the table in the layout of the BSP *public/flights_htmlb.htm*. For this purpose you must convert the entire layout of the page to HTMLB elements. Set the table width to 100% of the page width. Restrict the number of simultaneously displayed table rows to 10 and make it possible for the user to scroll through the rows. Only display the columns that were previously displayed. Make it possible for the user to sort the table by flight date, departure city, and destination city.

   Hint: Use the HTMLB elements *tableView*, *tableViewColumns*, and *tableViewColumn*.

   Hint: To allow the user to sort the table, you must, amongst other things, enable the triggering of the server event *headerClick*.

   To ensure that you can still use the connection number to navigate to the next page, you must add a column to the corresponding internal table. This column must contain the relevant URL for navigating to the next page. Proceed as follows:

   Create an additional table type that contains all the previous columns (line type *BAPISCODAT*) and another column of the type STRING for the URL. The name of this additional column is **URL**.

   Hint: To copy the fields of a row type t_wa2 in a row type t_wa1, use the following statements:

```
TYPES: BEGIN OF t_wa1,
          ... .
```

Continued on next page

```
INCLUDE TYPE t_wa2.
TYPES:  ....
       END OF t_wa1.
```

Create an additional page attribute for this extended table type. Navigate to the event handler *OnInitialization*. Extend the source code: The flight connection data, which was read using the application class *GET_FLIGHT_LIST*, must be transferred to the extended internal table. In the URL column, store the character string that was previously defined as the target for the link in the layout of the public/flights.htm page. Construct the corresponding query string using the ABAP statement *CONCATENATE*.

Navigate to the page layout. Set the type of the column *FLIGHTCONN* to the value **LINK** and assign the name of the column that contains the URL (*URL*) to the attribute **linkColumnKey**.

To eventually return to the start page by pressing the send button, you must adjust the source code in the event handler *OnInputProcessing* accordingly. To do this, refer to the last part of the exercise.

a) The BSP elements and their attributes are copied from the Tag Browser using Drag&Drop. The attribute values are added manually. If you double-click a BSP element, a dialog box appears that describes the element and the attributes. The model solution for this section is **NET200_S_12**.

public/start_htmlb.htm - LAYOUT

```
<htmlb:documentBody>
  ...
  <htmlb:form>

    <htmlb:gridLayout columnSize  = "4"
                      rowSize     = "5"
                      cellSpacing = "10"
                      width       = "100%" >
      <htmlb:gridLayoutCell columnIndex = "1"
                            rowIndex    = "1"
                            colSpan     = "4" >

        <%-- LAST MINUTE OFFERS                         --%>
        <htmlb:tableView id            = "last_minute"
                         table         = "<%= it_last_minute   %>"
                         width         = "100%"
                         footerVisible = "FALSE" >
```

Continued on next page

```
        </htmlb:tableView>
        <hr>
      </htmlb:gridLayoutCell>
      <htmlb:gridLayoutCell columnIndex = "1"
                            rowIndex    = "2" >
        <htmlb:label for  = "depa"
                     text = "<%= otr(net200/cityfrom) %>" />
    ...
  </htmlb:form>
</htmlb:documentBody>
```

public/flights_htmlb.htm - Type Definition

```
* Line Type
types:  begin of bapiscodat_ext.
          include type bapiscodat.
types:    url     type string,
        end   of bapiscodat_ext,

* Table Type
        tab_bapiscodat_ext type table of bapiscodat_ext.
```

public/flights_htmlb.htm - Page Attributes (Changes)

Attribute	Auto	Type	Reference Type	Description
it_con_dat_ext		TYPE	tab_bapis-codat_ext	Table with additional column for URL

public/flights_htmlb.htm - OnInitialization

```
****************************************.
* BAPI Call via Application class method
****************************************
CALL METHOD application->get_flight_list
  EXPORTING
```

Continued on next page

```
    travelagency           = travel_ag
*   AIRLINE                =
    destination_from       = dest_from
    destination_to         = dest_to
*   MAX_ROWS               =
  CHANGING
    date_range             = it_date
    flight_connection_list = it_con_dat.

  ...

DATA: wa1 LIKE LINE OF it_con_dat,
      wa2 LIKE LINE OF it_con_dat_ext.

LOOP AT it_con_dat INTO wa1.
  MOVE-CORRESPONDING wa1 TO wa2.
  CONCATENATE 'Details.htm?travel_ag='
               travel_ag
              '&connid='
               wa2-flightconn
              '&fldate='
               wa2-flightdate
              INTO wa2-url.
  APPEND wa2 TO it_con_dat_ext.
ENDLOOP.
```

public/flights_htmlb.htm - OnInputProcessing

```
* event handler for checking and processing user input and
* for defining navigation

IF event_id = cl_htmlb_manager=>event_id.

  DATA: event TYPE REF TO cl_htmlb_event.
  event = cl_htmlb_manager=>get_event( runtime->server->request ).
  IF event->name = 'button' AND event->server_event = 'back'.
    navigation->goto_page( 'START_HTMLB.HTM' ).
  ENDIF.

ENDIF.
```

Continued on next page

public/flights_htmlb.htm - LAYOUT

```
<%@extension name="htmlb" prefix="htmlb" %>
<htmlb:content design="DESIGN2002" >
  <htmlb:document>
    <htmlb:documentHead
           title="<%= otr(net200/flightconnections) %>" >
    </htmlb:documentHead>
    <htmlb:documentBody>
      <%@include file="Header_htmlb.htm" %>
      <htmlb:form>
        <htmlb:gridLayout columnSize  = "1"
                          rowSize     = "2"
                          cellSpacing = "10"
                          width       = "100%">
          <htmlb:gridLayoutCell columnIndex = "1"
                                rowIndex    = "1" >
            <htmlb:button
                   text = "<%= otr(SOTR_VOCABULARY_BASIC/BACK) %>"
                   onClick = "back"/>
          </htmlb:gridLayoutCell>
          <htmlb:gridLayoutCell columnIndex = "1"
                                rowIndex    = "2" >
            <htmlb:tableView
                   id              = "flight_table"
                   table           = "<%= it_con_dat_ext      %>"
                   width           = "100%"
                   sort            = "SERVER"
                   onHeaderClick   = "sort"
                   visibleRowCount = "15"
                   navigationMode  = "BYLINE" >
              <htmlb:tableViewColumns>
                <htmlb:tableViewColumn
                       columnName    = "flightconn"
                       width         = "200"
                       type          = "LINK"
                       linkColumnKey = "url" >
                </htmlb:tableViewColumn>
                <htmlb:tableViewColumn columnName = "flightdate"
                                       sort       = "TRUE" />
                <htmlb:tableViewColumn columnName="airportfr" />
                <htmlb:tableViewColumn columnName = "cityfrom"
                                       sort       = "TRUE" />
                <htmlb:tableViewColumn columnName="deptime" />
```

Continued on next page

```
                <htmlb:tableViewColumn columnName="airportto" />
                <htmlb:tableViewColumn columnName = "cityto"
                                       sort       = "TRUE" />
                <htmlb:tableViewColumn columnName="arrtime" />
              </htmlb:tableViewColumns>
            </htmlb:tableView>
          </htmlb:gridLayoutCell>
        </htmlb:gridLayout>
      </htmlb:form>
    </htmlb:documentBody>
  </htmlb:document>
</htmlb:content>
```

6. If you have the time to, create another layout.

   Using elements of the BSP extension HTMLB, also create the layout of the details page. Copy the page *public/details.htm* to the page *public/details_htmlb.htm*. Delete the layout on the new page and create the layout using the BSP extension HTMLB. You must also adapt the source code of the event handler *OnInputProcessing*. Ensure that the links on the page *public/flights_htmlb.htm* now take you to the page *public/details_htmlb.htm*. Refer to the relevant page in the model solution **NET200_S_13**.

   a)

   public/details_htmlb.htm - LAYOUT

```
<%@extension name="htmlb" prefix="htmlb" %>
<htmlb:content design="DESIGN2002" >
  <htmlb:document>
    <htmlb:documentHead
           title="<%= otr(net200/flightconnections) %>" >
    </htmlb:documentHead>
    <htmlb:documentBody>
      <%@include file="Header_htmlb.htm" %>
      <htmlb:form>
        <%@include file="hidden_htmlb.htm" %>
        <htmlb:gridLayout columnSize  = "1"
                          rowSize     = "3"
                          width       = "100%"
                          cellSpacing = "10" >
          <htmlb:gridLayoutCell columnIndex = "1"
```

Continued on next page

```
                      rowIndex    = "1" >
  <htmlb:textView
         text  = "<%= otr(net200/Einzelverbindungen) %>"
         design = "HEADER2" />
  <htmlb:tableView
         id            = "flight_hop_list"
         table         = "<%= it_flight_hop_list   %>"
         width         = "100%"
         footerVisible = "FALSE" >
    <htmlb:tableViewColumns>
      <htmlb:tableViewColumn columnName="hop" />
      <htmlb:tableViewColumn columnName="airlineid" />
      <htmlb:tableViewColumn columnName="connectid" />
      <htmlb:tableViewColumn columnName="cityfrom" />
      <htmlb:tableViewColumn columnName="cityto" />
      <htmlb:tableViewColumn columnName="deptime" />
    </htmlb:tableViewColumns>
  </htmlb:tableView>
</htmlb:gridLayoutCell>
<htmlb:gridLayoutCell columnIndex = "1"
                      rowIndex    = "2" >
  <htmlb:textView
         text   = "<%= otr(net200/Verfuegbarkeit) %>"
         design = "HEADER2" />
  <htmlb:tableView
         id            = "flight_availibility_list"
         table         = "<%= it_availibility  %>"
         width         = "100%"
         footerVisible = "FALSE" >
    <htmlb:tableViewColumns>
      <htmlb:tableViewColumn columnName="hop" />
      <htmlb:tableViewColumn columnName="firstfree" />
      <htmlb:tableViewColumn columnName="businfree" />
      <htmlb:tableViewColumn columnName="econofree" />
    </htmlb:tableViewColumns>
  </htmlb:tableView>
</htmlb:gridLayoutCell>
<htmlb:gridLayoutCell columnIndex = "1"
                      rowIndex    = "3" >
  <htmlb:button text    = "<%= otr(NET200/FIRSTFREE) %>"
                onClick = "first_book" />
  <htmlb:button text    = "<%= otr(NET200/BUSINFREE) %>"
                onClick = "busi_book" />
  <htmlb:button text    = "<%= otr(NET200/ECONOFREE) %>"
                onClick = "econ_book" />
```

Continued on next page

```
          </htmlb:gridLayoutCell>
        </htmlb:gridLayout>
      </htmlb:form>
    </htmlb:documentBody>
  </htmlb:document>
</htmlb:content>
```

public/details_HTMLB.htm - OnInputProcessing

```
IF event_id = cl_htmlb_manager=>event_id.

  DATA: event TYPE REF TO cl_htmlb_event.
  event = cl_htmlb_manager=>get_event( runtime->server->request ).

  CASE event->server_event.
    WHEN 'back'.
      ...
    WHEN 'busi_book'.
      ...
    WHEN 'econ_book'.
      ...
  ENDCASE.

  navigation->goto_page( '../protected/customer.htm' ).

ENDIF.
```

public/flights_htmlb.htm - OnInitialization

```
...
LOOP AT it_con_dat INTO wa1.
  MOVE-CORRESPONDING wa1 TO wa2.
  CONCATENATE 'Details_htmlb.htm?travel_ag='
               travel_ag
              '&connid='
               wa2-flightconn
              '&fldate='
               wa2-flightdate
```

Continued on next page

```
              INTO wa2-url.
  APPEND wa2 TO it_con_dat_ext.
ENDLOOP.
```

Lesson Summary

You should now be able to:

- Use elements of the BSP extension HTMLB to design the layout of Business Server Pages
- Process user input made using BSP elements
- Extract data from a query string
- Adapt the design of a BSP application based on the BSP extension HTMLB
- Render complex pages, consisting of several subpages, on the server side

Lesson: Composite Elements

Lesson Overview

When you create BSP applications with BSP extensions, it can happen that using BSP elements, which are generally easy, is not as trivial as you think. To generate the required layouts, you often need a number of special elements, and for each element possibly also a number of parameters.
In such cases, to simplify handling the many special elements and to minimize the development effort involved for the creator of the BSP application or its layouts, you can create what are known as composite BSP elements. These elements implicitly call functions of several BSP elements that already exist and thus represent a type of shell for using a certain combination of other elements.

Lesson Objectives

After completing this lesson, you will be able to:

- Create a new BSP extension
- Create new BSP elements
- Describe the class hierarchy for a BSP element
- Encapsulate combinations of different existing BSP elements in a new BSP element

Business Example

When you create different BSP applications, forms need to be created again and again. These consist of a series of input/output fields, corresponding field labels, and a range of pushbuttons. Here the aim is to have a uniform design - that is, in the different forms, the spacing, colors, fonts, and so on, should be identical. Therefore, it is a good idea to define a new BSP element that encapsulates these functions using existing BSP elements.

Creating BSP Elements

As a rule, the functions of BSP elements are implemented by calling the elements (by using the appropriate tag) in the layout of a page. You are not interested at this point in what happens at runtime when the layout is processed. The situation is quite different, however, if you wish to use the functions of a BSP element explicitly from within the ABAP source code. This is the case whenever new BSP elements are created that re-use the functions of existing BSP elements.

In this lesson, we will focus on creating composite elements. Here, the functions of existing BSP elements are used from within the functions of the newly created elements. From an organizational point of view, BSP elements are assigned to a BSP extension.

You will find the option of creating a new BSP extension - for example - in the context menu of the package (submenu Web objects) in the navigation area of the ABAP workbench (SE80). The **BSP extension** represents a container for an arbitrary number of individual BSP elements. When creating the extension, you can assign a default prefix. This is used in the *extension* directive and, in the tags for the BSP elements, it is usually placed directly in front of the element name, separated by a colon. In addition, it is possible to assign a BSP element basis class. This is then used as a superclass for all BSP elements. If no BSP element basis class is explicitly assigned, the class *CL_BSP_ELEMENT* will be used as superclass.

BSP elements can be created as subobjects of a BSP extension. When you create an element, both the element name and the name of the element handler class must be specified (customer namespace). This contains the entire range of element functions and thus represents the BSP element at the technical level. The element handler class is automatically created when you generate the BSP element. It has the following relationship to the BSP element basis class:

> On the basis of the properties and attributes of the BSP element, a **class is generated** to which the BSP element basis class is assigned as a superclass. The functions of the BSP element basis class are thus enhanced by the attributes of the BSP element and the four methods *CONSTRUCTOR*, *CLASS_CONSTRUCTOR*, *FACTORY*, and *FACTORY_CLEAR*. These methods are automatically filled with source code, depending on the properties of the element. The name of the generated basis class is ***ZCLG_<extension>_<element>*** , whereby *<extension>* is the name of the BSP extension and *<element>* is the name of the BSP element.
>
> The BSP handler class is derived from the generated basis class *ZCLG_<extension>_<element>*. In this way, it is ensured that manual changes to the element handler class will not be lost when the element properties and attributes are changed and there is thus the need to generate the basis class again.

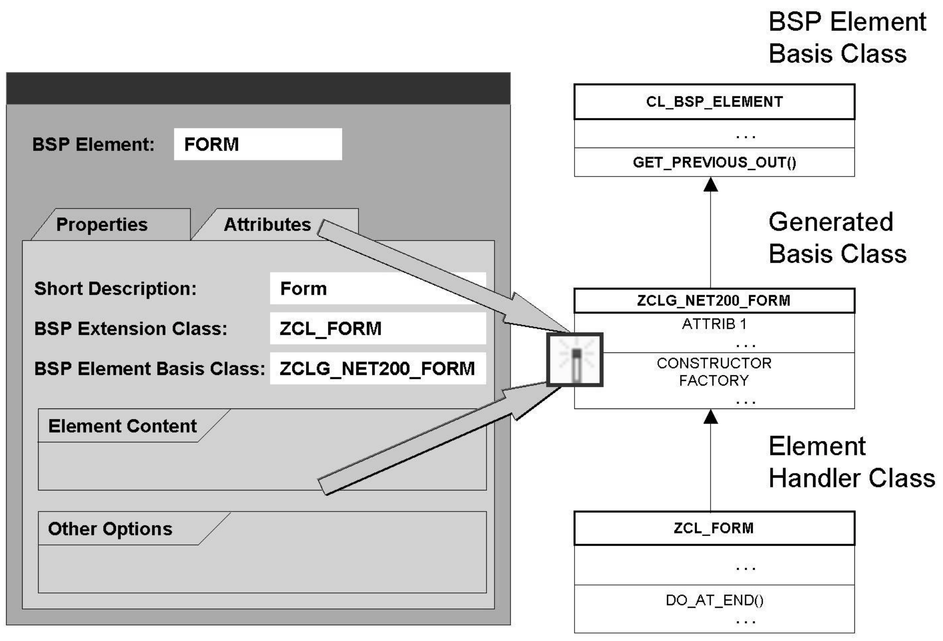


Figure 57: Class Hierarchy for BSP Elements

After the BSP element has been created, you need to define the properties and attributes of the element. Through the attributes, it is possible to pass information to the BSP element - information needed for element processing. These attributes therefore form the interface to the caller and are visible when used in the layout as element parameters.

You can make the following settings through the properties of the BSP element:

Element content:

- The element has no content - that is, between the opening and the closing tag, no content is allowed. Example: *<htmlb:button/>*

- Element contains solely other BSP elements. Example: *<htmlb:tableView> ... </htmlb:tableView>*

- Element can contain HTML source code in addition to BSP elements. Example: *<htmlb:form> ... </htmlb:form>*

User-defined validation

With the methods *COMPILE_TIME_IS_VALID* and *RUNTIME_IS_VALID* in the element handler class, user-defined validation of the element content can be performed at compilation time and/or runtime, provided this setting is activated. Example: *<htmlb:inputField>*

Iteration through element content

If a source text section is to be run through several times, you can have this done by setting this parameter. In this case, the method *DO_AT_ITERATION* of the element handler class is processed so often until the return parameter *RC* is set to the value *CO_ELEMENT_DONE*.

Manipulation of element content

By setting this parameter, you make it possible to subsequently change the HTML source code generated in the element. The source code generated in such an element must be explicitly passed to the surrounding element. For this purpose, you need to implement the method *DO_AT_END*. If this does not happen, the source code will be discarded. Example: *<htmlb:gridLayoutCell>* passes the content to the element *<htmlb:gridLayout>*.

'Page Done' is not returned at end of BSP element.

After the BSP element has been processed, other BSP elements are processed. This is the standard setting.

BSP Elements: Functionality

The element handler class of a BSP element contains a range of methods that are inherited from the element basis class and are called by the BSP runtime in certain circumstances and in a certain sequence. The developer's task is to assign the element-specific source code to the BSP element by redefining the appropriate methods. The following methods can be redefined:

COMPILE_TIME_IS_VALID

This method can be used to recognize incorrect transfers to the attributes of the BSP element. This takes place during the syntax check. Prerequisite for having this method processed: The *user-defined validation* setting is activated.

RUNTIME_IS_VALID

This method can be used to recognize incorrect transfers to the attributes of the BSP element at runtime. Prerequisite for having this method processed: The *user-defined validation* setting is activated.

DO_AT_BEGINNING

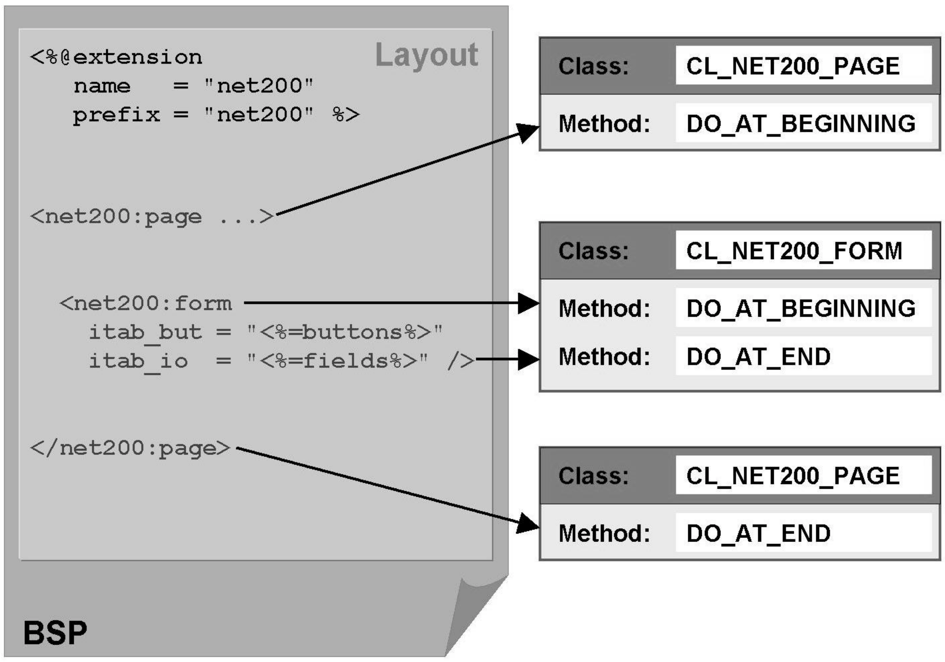
Encapsulates the source code that is to be processed when the opening tag (layout) for the BSP element is to be processed.

DO_AT_END

Encapsulates the source code that is to be processed when the closing tag (layout) for the BSP element is to be processed.

DO_AT_ITERATION

Encapsulates the source code for the BSP element that is to be run through several times in a loop.



```
<%@extension
   name   = "net200"
   prefix = "net200" %>


<net200:page ...>


  <net200:form
    itab_but = "<%=buttons%>"
    itab_io  = "<%=fields%>" />


</net200:page>
```

Figure 58: Processing BSP Elements

The following rules apply to method processing:

1. At the end of the method ***DO_AT_BEGINNING*** , further processing is influenced by the return parameter **RC**, which can accept the values **CO_ELEMENT_DONE** (RC=0) and **CO_ELEMENT_CONTINUE** (RC=1). This determines whether the element content is processed (RC=1) or not (RC=0).

   Example: If the list of the texts to be displayed is passed as table to the BSP element *<htmlb:breadCrumb>*, the element content is not evaluated. If the individual texts, on the other hand, are defined statically through the element *<htmlb:breadCrumbItem>*, the element content is evaluated.
2. At the end of the method ***DO_AT_ITERATION*** , further processing is influenced by the return parameter **RC**, which can accept the values **CO_ELEMENT_DONE** (RC=0) and **CO_ELEMENT_CONTINUE** (RC=1). This determines whether the element content is processed (RC=1) or not (RC=0) again.
3. At the end of the method ***DO_AT_END*** , further processing is influenced by the return parameter **RC**, which can accept the values **CO_PAGE_DONE** (RC=0) and **CO_PAGE_CONTINUE** (RC=1). Page processing for RC=CO_PAGE_DONE ends only when the parameter *No return of 'Page Done' at the end of the BSP element* is not set for the element properties.
4. To create the HTML source code from the above mentioned methods, the source code must be passed to what are known as bodywriters. The bodywriter of the current element must, in this case, pass the content to the bodywriter of the surrounding element. In this way, the outputs of all elements are accumulated. You need to distinguish between two techniques, depending on whether the **parameter *Manipulation of Element Content*** is set or not:

   Parameter is not set: Using the method ***ME->PRINT_STRING(<string>)*** , you can pass a previously created string to the surrounding bodywriter. Example: *<htmlb:button>*

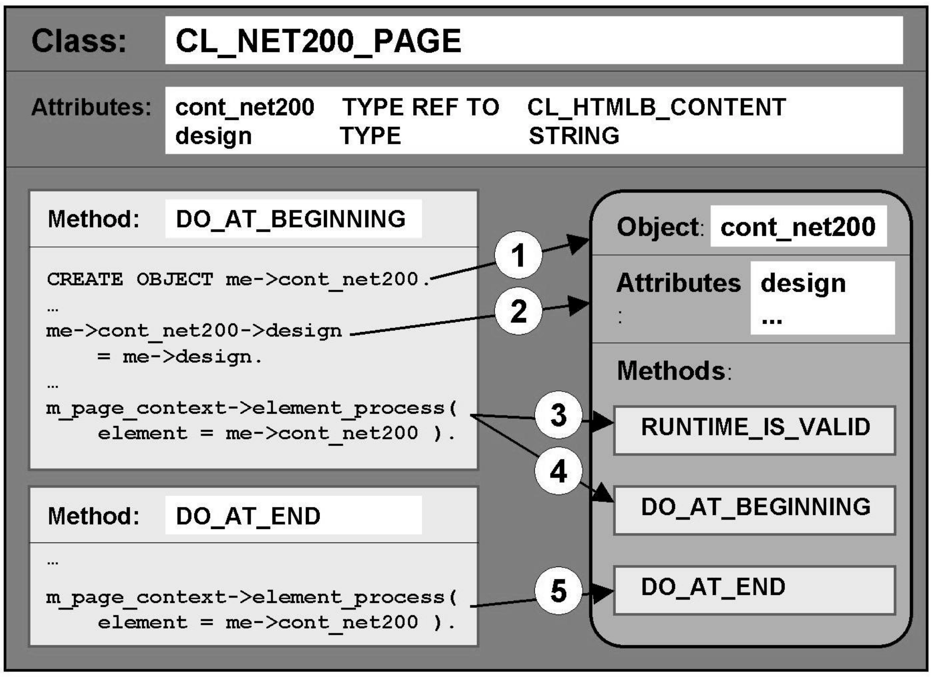
   Parameter is set: In the method ***DO_AT_END*** , the bodywriter of the current element must be explicitly passed to the bodywriter of the surrounding element. There is a reference to the current bodywriter available through the attribute *M_OUT* (however, only if the parameter *Manipulation of Element Content* is set!). Access to the bodywriter of the surrounding element can be shown using the example of the element *<htmlb:tabStripItemHeader>*.

Composite Elements

The content of the new BSP elements cannot be created solely through direct definition of the HTML source code in the corresponding element handler class, but also through processing of existing BSP elements. For this purpose, the methods of the element handler classes of existing BSP elements must be processed from within the methods of the element handler class of the new BSP element. Many actions that run automatically when a BSP element is processed from within the layout must, however, be programmed here. To do this, proceed as follows:

> For each BSP element that is to be processed, an appropriate object (*obj*) must first be instantiated. The respective element handler class serves as a reference class.
>
> The attributes of this object must be filled explicitly. Reminder: The attributes of the object correspond to the parameters of the BSP element when called from within the layout.
>
> The correct methods of the object *obj* are processed by the following call:
> `m_page_context->element_process( element = me->obj ).`

The method *M_PAGE_CONTEXT->ELEMENT_PROCESS(...)* decides independently, depending on the call time and the properties of the element to be processed, which method(s) need to be processed.



```
CREATE OBJECT me->cont_net200.
…
me->cont_net200->design
    = me->design.
…
m_page_context->element_process(
    element = me->cont_net200 ).
```

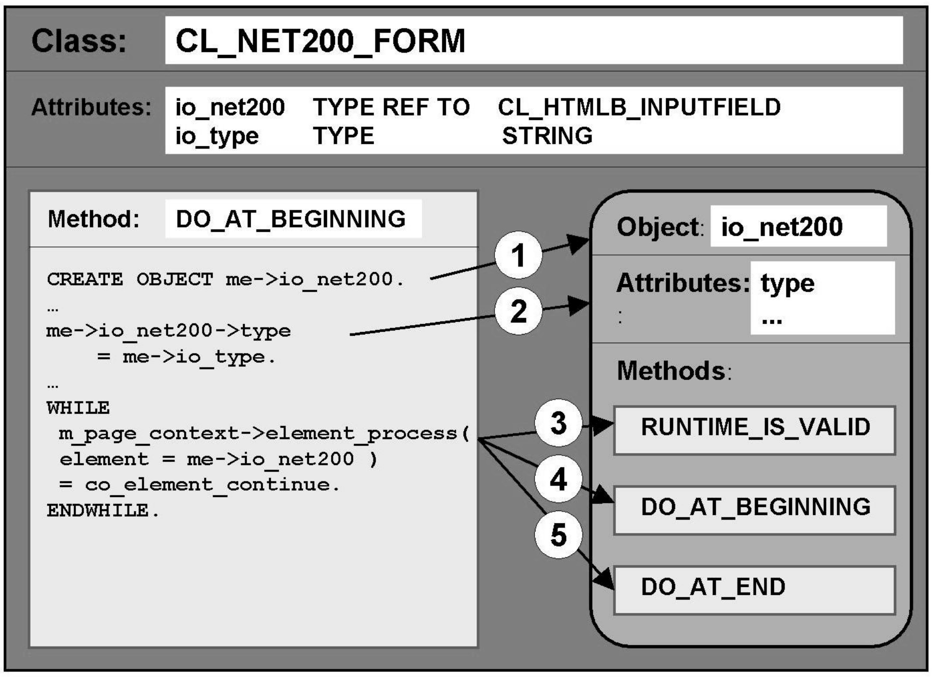
```
…
m_page_context->element_process(
    element = me->cont_net200 ).
```

Figure 59: Processing Existing BSP Elements

If element processing is started from within the method *DO_AT_BEGINNING*, the methods *DO_AT_ITERATION* and *DO_AT_END* of the element to be processed, however, are not run by default. Depending on the implementation of the already existing BSP element, it can be necessary to have these methods run as well before the next BSP element is processed.

Example: If a BSP element is to be processed without element content (*<htmlb:inputField>*, *<htmlb:label>* ...), all the methods of this object must be run before another element can be processed.

To achieve this, the method *M_PAGE_CONTEXT->ELEMENT_PROCESS(...)* is run as often as it takes to cover all the methods of the element to be processed. For this purpose, you can analyze the return value of the functional method *M_PAGE_CONTEXT->ELEMENT_PROCESS(...)*. This gives you information on whether element processing is completed (return value = 0). In particular, with this type of call, the caller does not need any knowledge of the exact implementation of the BSP element to be processed.

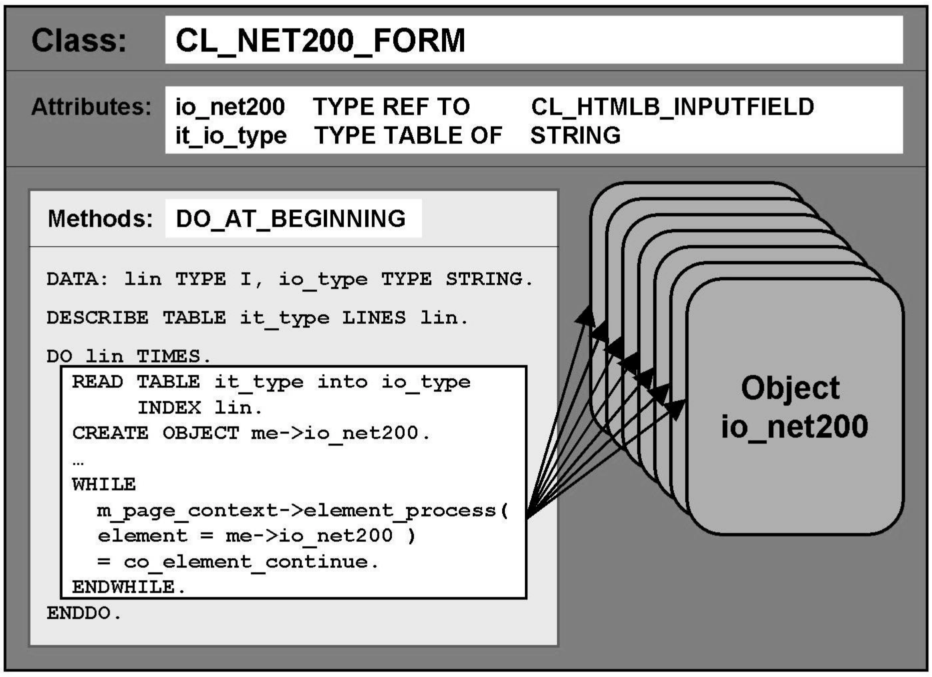


```
io_net200   TYPE REF TO   CL_HTMLB_INPUTFIELD
io_type     TYPE          STRING
```

```
CREATE OBJECT me->io_net200.
…
me->io_net200->type
    = me->io_type.
…
WHILE
 m_page_context->element_process(
 element = me->io_net200 )
 = co_element_continue.
ENDWHILE.
```

Figure 60: Complete Processing of Existing BSP Elements

As soon as all the methods of the object *obj* are processed, the object cannot be processed a second time. If this is necessary, the object must first be instantiated a second time. Explicit deletion of the reference variable*obj* is not necessary before the renewed instantiation of the object.



Class: CL_NET200_FORM

Attributes:

io_net200	TYPE REF TO	CL_HTMLB_INPUTFIELD
it_io_type	TYPE TABLE OF	STRING

Methods: DO_AT_BEGINNING

```
DATA: lin TYPE I, io_type TYPE STRING.
DESCRIBE TABLE it_type LINES lin.
DO lin TIMES.
  READ TABLE it_type into io_type
       INDEX lin.
  CREATE OBJECT me->io_net200.
  …
  WHILE
    m_page_context->element_process(
    element = me->io_net200 )
    = co_element_continue.
  ENDWHILE.
ENDDO.
```

Figure 61: Multiple Processing of Existing BSP Elements

Lesson Summary

You should now be able to:

- Create a new BSP extension
- Create new BSP elements
- Describe the class hierarchy for a BSP element
- Encapsulate combinations of different existing BSP elements in a new BSP element

Lesson: Model View Controller for BSPs

Lesson Overview

This lesson describes the advantages of using the Model View Controller (MVC) and the implementation of this programming paradigm.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the advantages of the MVC programming paradigm over classic BSP programming
- Create and call controllers, views, and models

Business Example

You must create a BSP application based on the Model View Controller (MVC) programming paradigm.

MVC Design Pattern

The **Model View Controller (MVC)** design pattern clearly distinguishes between process control, the data model, and the display of data on the user interface. The formal separation of these three areas is realized with the three objects **model**, **view**, and **controller**. You can thus simply divide complex Web applications into logical units.

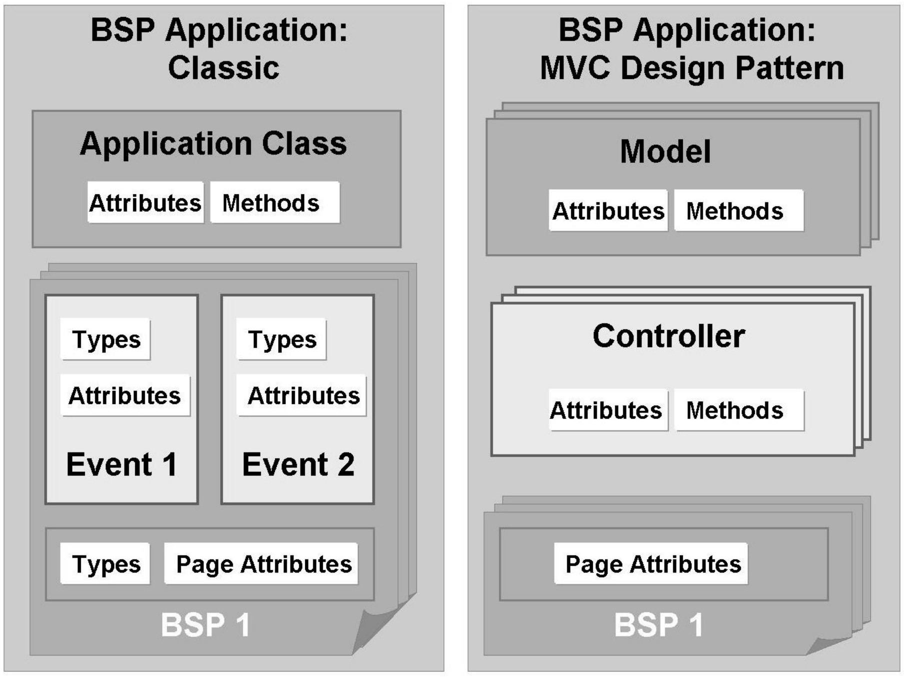


Figure 62: Classic BSP Application - MVC Design Pattern

The **model** serves as an application object for managing the application data. The model responds to both information requests regarding its status, which usually come from the view, and instructions on changing state, which usually come from the controller. In this way, the model serves the sole purpose of internal data processing, without referring to the application and its user interface. A model can have different views, which are implemented using different view sets.

The **view** handles the graphical and textual output on the user interface and hence displays the input and output data in the relevant interface elements. These can be buttons, menus, or dialog fields, for example. The view therefore takes care of the visualization. To display the state, the view requests information from the model, or the model informs the view about any changes of state.

The **controller** interprets and checks the user's mouse and keyboard input and ensures the model or view layer changes if required. Input data is passed on and changes to the model data are initiated. The controller uses the methods of the model to change the internal state and then informs the view. The controller thus defines the reactions to user inputs and controls the processing.

The view and the controller form the user interface. Since the model does not know the views or the controllers, the internal data processing is separated from the user interface. Therefore, changes to the user interface do not affect the internal processing of the data and the data structure.

However, this makes it possible to display the data in different formats simultaneously; for example, you can display the numeric percentage results of a vote as a table, bar chart, or pie chart.

Uses

The BSP programming model allows you to control the flow logic, event handling, and navigation using redirects. Using the MVC design pattern offers various advantages, so you should consider using MVC in the following cases:

- If your pages (BSPs) are made up of several parts (components) dynamically:

  A controller can construct a page that consists of several views. This allows you to split the layout into components.
- If the input processing is so complex that it should be subdivided into different methods.

  A controller offers a great level of flexibility, particularly with input processing, because you can create and call new methods
- If only the input processing can decide which page is next, we recommend that you let the controller branch to different views.
- If navigation using HTTP redirects causes performance problems (for example, slow connection).
- If the display logic is relatively complex, you can clearly separate the logic from the layout using Model View Controller.
- If the layout and the display logic are edited by different people.
- If parts of the layout are to be created using programs - for example, using a generator or XSLT processor.

Combining MVC with Existing BSPs

You can combine the techniques of the BSP programming model with the newly integrated MVC design pattern: A BSP application can contain pages with flow logic as well as controllers and views. The views can be called only by the controllers. Transitions from pages to controllers and back can take place using redirects with the navigation methods. In the page layouts, you can call a controller using the `<bsp:call>` element or the `<bsp:goto>` element. However, you cannot use these elements to call views.

Architecture Examples

The following figures illustrate how you can use controllers, views, and models to redesign a classic BSP application (without the MVC design pattern) consisting of three BSPs. Which of the models is used depends on the complexity of the flow logic, the business logic, and the desired display (all information on one HTML page or spread out over several HTML pages).

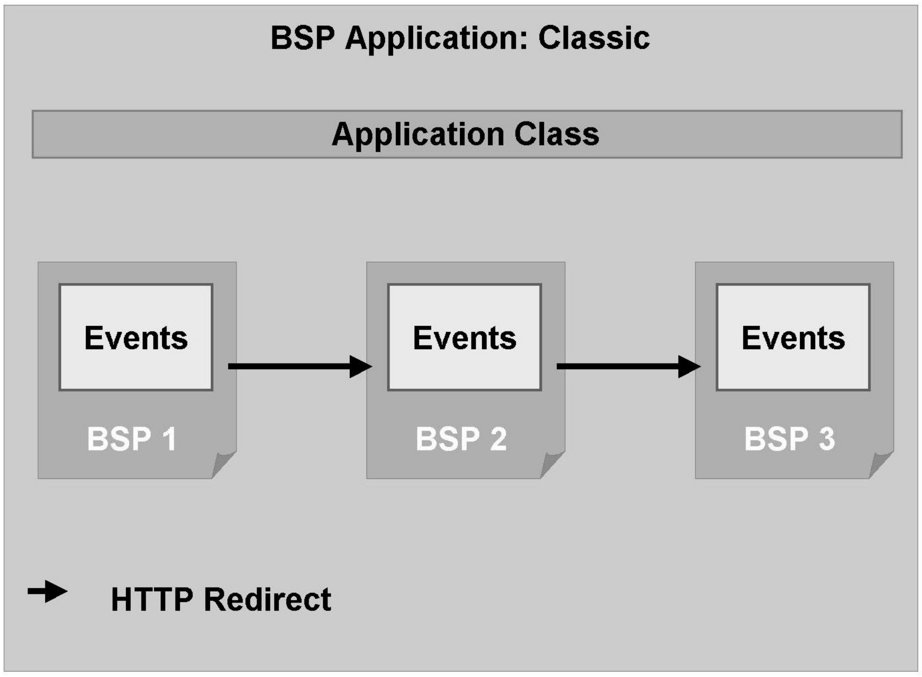


Figure 63: Classic BSP Application with Three BSPs

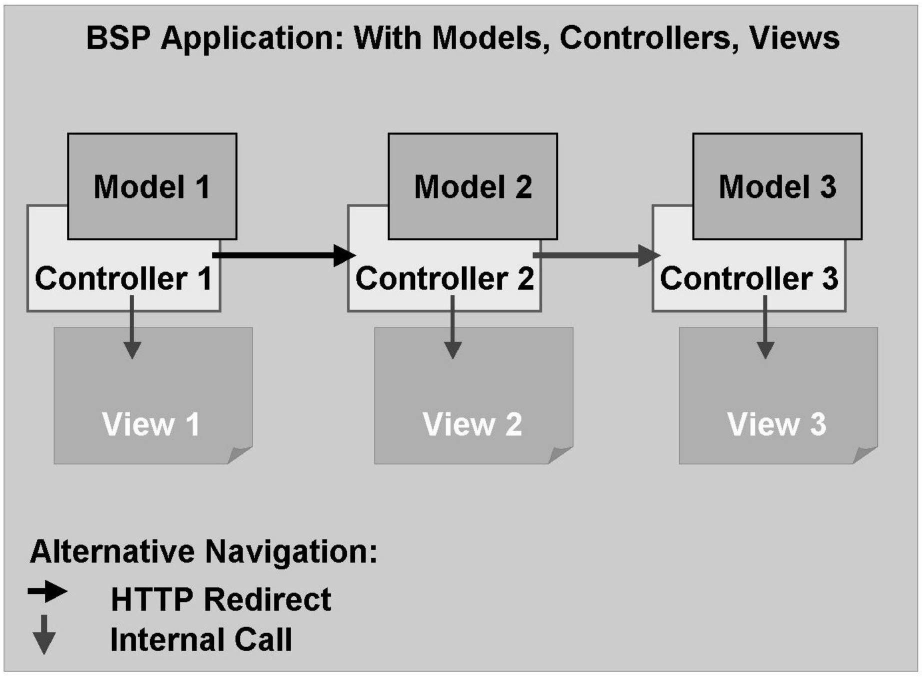


Figure 64: Replacing Classic BSPs with Models, Views, and Controllers

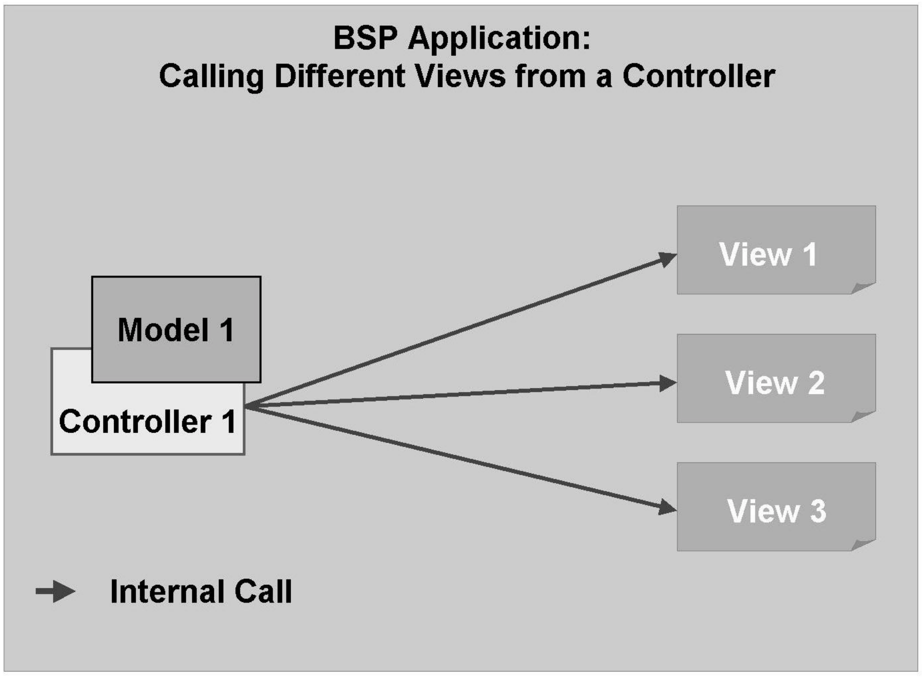


Figure 65: Calling Different Views from One Controller

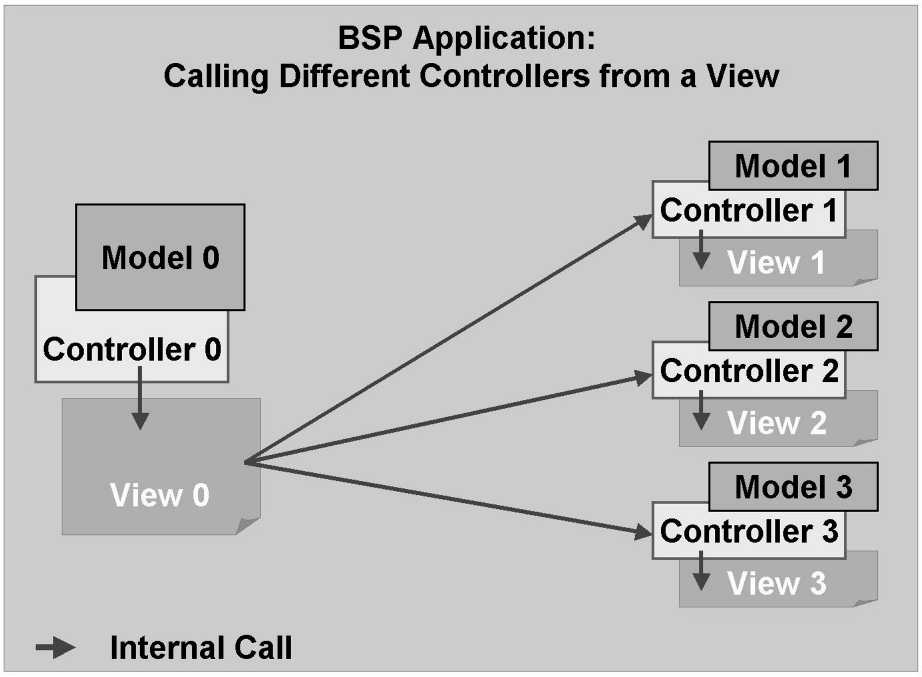


Figure 66: Calling Different Controllers from One View

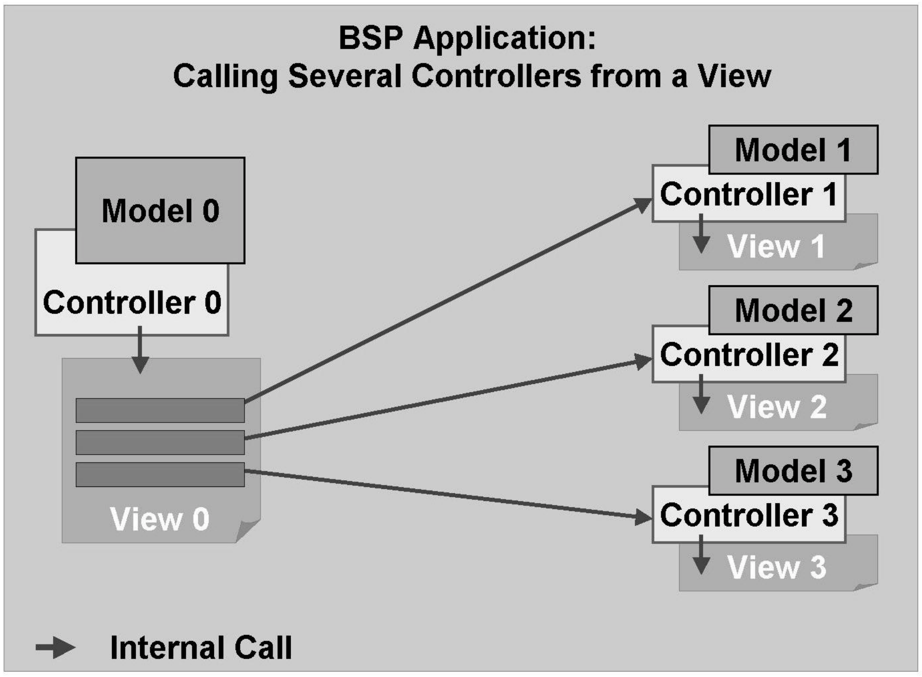


Figure 67: Calling Several Controllers from One View Simultaneously

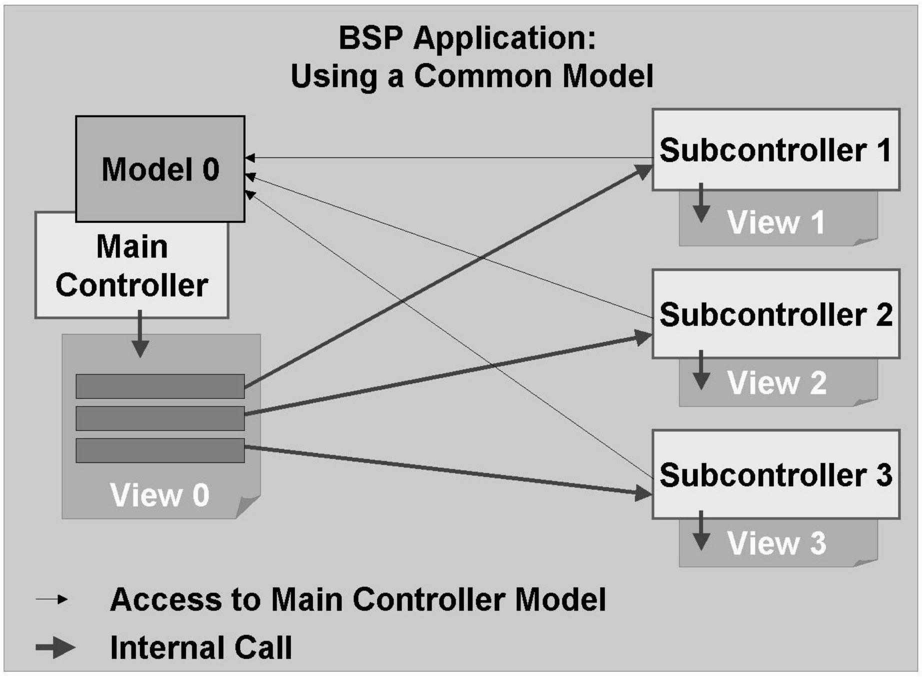


Figure 68: Using One Model for an Entire BSP Application (No Application Class Required)

Model, View, and Controller: Definition and Properties

A **view** is created as a subobject of a BSP application in the Web Application Builder. The layout is usually edited using the BSP extension HTMLB. Page attributes represent the interface with the controller that calls the view. Because both objects must know the types of these page attributes, the types are assigned in the Data Dictionary. Therefore, views do not have a tab for creating types, which could be used for specifying types of page attributes.

A **model** is also created using the Object Navigator, in the Class Builder. As a class, the model class is an independent Repository object and is not assigned to a BSP application as a subobject. To simplify programming with the MVC design pattern, the framework provides a basis class CL_BSP_MODEL for the model of an application. The model class can then be derived from this basis class. The model class represents the data context of the application and hence contains a copy (or reference to) the data from the database model that is relevant to the view.

The model class provides the following:

- The data used for the views with the relevant type and Data Dictionary information
- Input conversions
- Information about which data had input errors

Data binding is especially useful for the value transfer of input and output data. For this purpose, the data required by the view is added to the model class as attributes. These attributes are visible to the public and can be any of the following:

- Simple variables
- Structures
- Tables

In the simplest case, the model class has only these attributes and can therefore be easily used for data binding in a BSP application. A simple model class like this offers the following functions:

- The controller can create a model instance and initialize the attributes because they are public attributes.
- The controller passes a reference pointing to the model instance to the view.
- In the view, the data binding to the model is expressed as a path (//...) for each view element.

Example: A BSP application has an input field that was inserted using HTMLB; the user can make entries in this input field. The reference variable *model* is defined as a page attribute. In the model, there is an attribute *io_field* that corresponds to the input field. You can then write the following for the input field: `<htmlb:input_field ... value="//model/io_field"/>`. As a result, the contents of *value* are passed to the relevant attribute in the model class. The process flow is now as follows:

- Using the statement above, the content of the attribute is assigned the value of the input field.
- The ID is generated from the model.
- Additional properties are also generated, for example, whether input help (F4) is available or fixed values exist.
- With the next request, the user input is transferred to the model class.
- Data conversions, including those using the Data Dictionary (for example, conversion exits) are automatically executed by the basis model class.

If a conversion exit exists for a field in the Data Dictionary, this conversion exit is called by default. All data in the Data Dictionary structure for the field is available. If necessary, you can also add your own methods to your model class for further processing of the attributes.

Views and Controllers: Creating and Using

The following describes how to create and use the components view, and controller.

A **controller** is created as a subcomponent of a BSP application in the Web Application Builder. Controllers have the file extension **.do**. The controller class is the most important attribute to specify.

The controller class is always a subclass of the class **CL_BSP_CONTROLLER2** delivered with the SAP Standard. If the controller class has not been created (which is the default case), you can still enter the class name in the relevant field and create the controller class by forward navigation.

The controller class inherits a number of attributes and methods from the super class *CL_BSP_CONTROLLER2*. These attributes and methods are used when processing user input or connecting to models, for example. You can add other attributes to the controller class, depending on the created BSP application. Additional attributes are one way of passing data to the controller when it is called. The programmer will also redefine a range of methods to intialize attributes (which may be required), handle events, or process form data. Just as the event handlers are processed in a predefined sequence in a classic BSP with a processing block, a range of methods in the controller class is also processed in a predefined order.

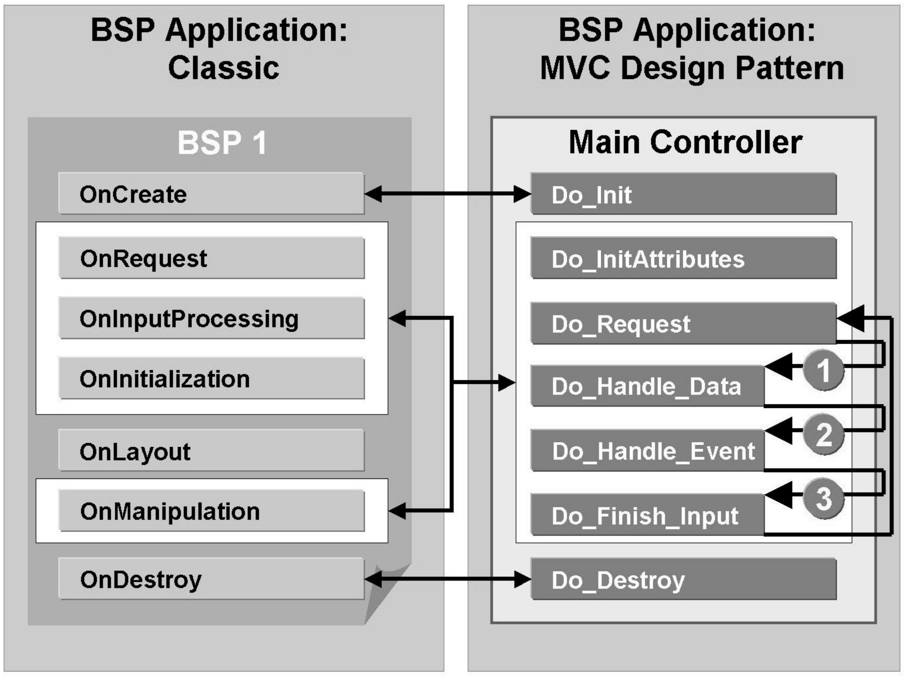


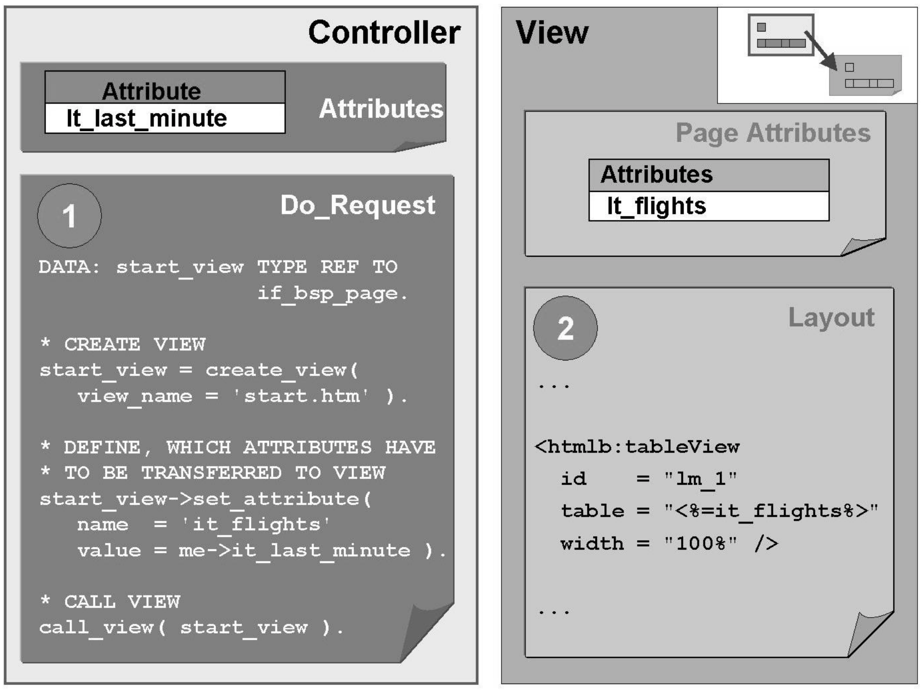
Figure 69: Comparison: Event Handlers in BSPs with Flow Logic and Methods of the Controller Class

Calling a View from a Controller

To call a view from within a controller, you must first create an interface reference variable of the type *IF_BSP_PAGE*. Using the *CREATE_VIEW* method, a view instance is then created and the address is passed back to the interface reference variable. When creating the view instance, the name of the assigned view is also passed.

Attributes that were previously set in the controller (for example, in the method *DO_INIT* or *DO_INITATTRIBUTES*) can be passed to the view instance using the method *SET_ATTRIBUTE(...)*. Values of attributes or references to objects are passed the interface parameters of the view.

Finally, the view is called and hence the layout is processed.



```
DATA: start_view TYPE REF TO
                if_bsp_page.

* CREATE VIEW
start_view = create_view(
   view_name = 'start.htm' ).

* DEFINE, WHICH ATTRIBUTES HAVE
* TO BE TRANSFERRED TO VIEW
start_view->set_attribute(
   name  = 'it_flights'
   value = me->it_last_minute ).

* CALL VIEW
call_view( start_view ).
```

```
...

<htmlb:tableView
  id    = "lm_1"
  table = "<%=it_flights%>"
  width = "100%" />

...
```

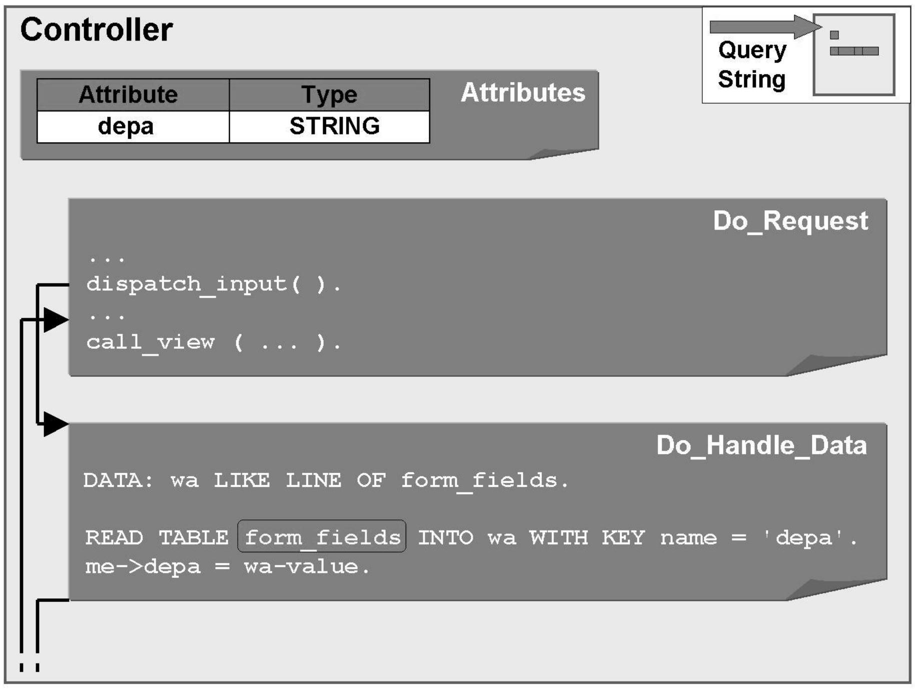
Figure 70: Calling a View from a Controller and Passing Attributes

Receiving User Input

After the user has sent a form, the information is passed to the main controller as a query string. Calling the ***DISPATCH_INPUT( )*** method within the controller method *DO_REQUEST( )* ensures that, in the case of a non-empty query string, the methods *DO_HANDLE_DATA*, *DO_HANDLE_EVENT*, and *DO_FINISH_INPUT* are processed.

The user input is processed in the *DO_HANDLE_DATA* method. The BSP runtime fills a two-column internal table, in which the name/value pairs of the query string are stored. The name of this internal table is ***FORM_FIELDS*** . If the form fields are not bound to attributes of a model, the individual values must be read from the table, checked for type errors, and passed to attributes of the controller.

Hint: The user entries are stored unformatted in the internal table *FORM_FIELDS*. Any required conversion to the internal format must be programmed (for example, for a date, user input would be 13/12/2003; internal format would be 20031213).



Attribute	Type
depa	STRING

```
...
dispatch_input( ).
...
call_view ( ... ).
```

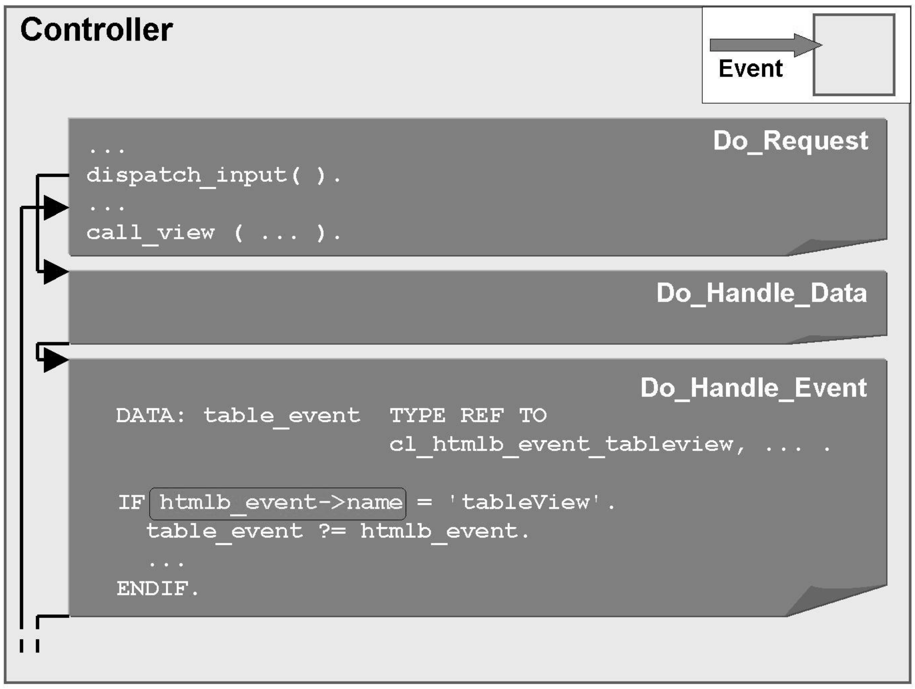
```
DATA: wa LIKE LINE OF form_fields.

READ TABLE form_fields INTO wa WITH KEY name = 'depa'.
me->depa = wa-value.
```

Figure 71: Reading the Form Data in the Event Handler *DO_HANDLE_DATA*

Event Handling

If the user triggers an HTMLB event in the Web browser (for example, by pressing the send button), this information is passed to the main controller as part of the query string (name/value pair *oninputprocessing=htmlb*). After the user entries have been processed (*DO_HANDLE_DATA*), the **DO_HANDLE_EVENT** method is started.

Unlike event handling for BSPs with flow logic, the runtime environment here creates a suitable handler object depending on the triggering HTMLB element - therefore, you do not have to ascertain the event type using the HTMLB Manager.



```
...
dispatch_input( ).
...
call_view ( ... ).
```

```
  DATA: table_event  TYPE REF TO
                     cl_htmlb_event_tableview, ... .

  IF htmlb_event->name = 'tableView'.
    table_event ?= htmlb_event.
    ...
  ENDIF.
```

Figure 72: Handling Server Events using the Event HandlerDO_HANDLE_EVENT

Method *DO_FINISH_INPUT*

In same cases, the user entries have to be processed and the event handling has to be executed before the boudary conditions for selecting data from the database are defined.

Example: In a form, the user can select a data record in a table. Via two send buttons, the user can either navigate to the previous page or to the next page, where the details for the selected data record are displayed. Therefore, before the detail data is read, the system must first check whether details are to be displayed (*DO_HANDLE_EVENT*) and, if so, for which of the displayed data records (*DO_HANDLE_DATA*). Therefore, after *DO_HANDLE_EVENT*, a further method (***DO_FINISH_INPUT***) is processed, in which the corresponding subsequent action can be defined.

Main Controller and Subcontroller

The main controller and subcontrollers are used for structuring the functions in large applications. For example, you can define components that can be combined with other components in large applications.
Of course, the individual components can be assigned to different applications.

If the main controller and subcontroller are to be used, the main controller is always the object requested from the browser using the URL. Therefore, the corresponding instance to the controller class of the main controller is created from within the BSP runtime environment.

Subcontrollers, on the other hand, are instances of controller classes generated from within the source code of the main controller instance. Subcontrollers are therefore attributes of the main controller. Subcontrollers can be created in two ways:

1. The subcontrollers are created explicitly before the main view is processed.

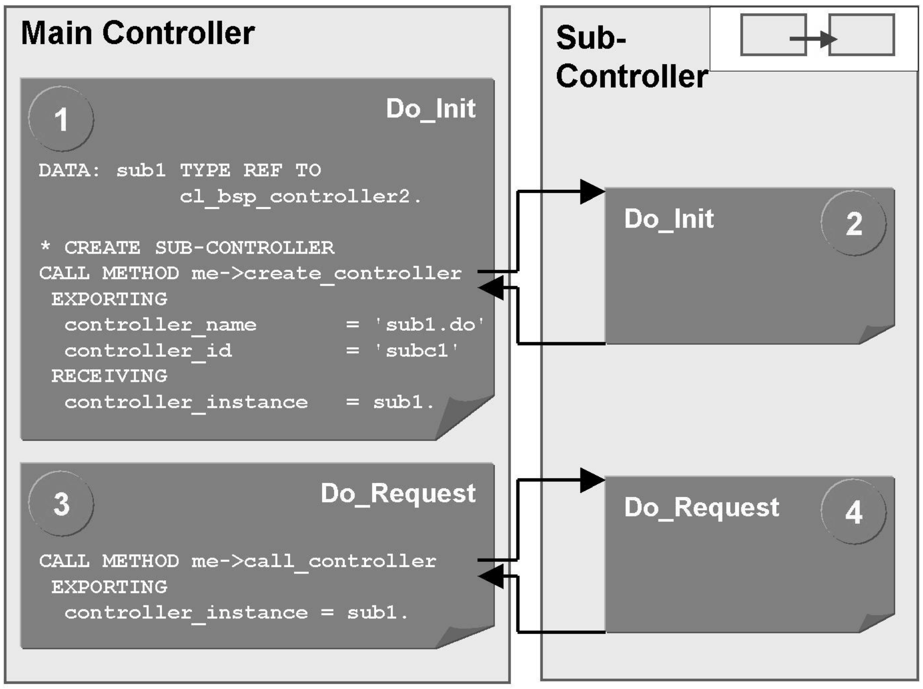
   There are different possibilities for this: The methods *DO_INIT*, *DO_INITATTRIBUTES*, and *DO_REQUEST*. If the application is executed statefully, the event handler *DO_INIT* would be best, since this source code is only processed once. However, if the conditions, which indicate which subcontroller is to be created, are not known till later, a later time could be more suitable.
2. The subcontrollers are created implicitly when the main view is processed. This represents the latest possible time.

Explicit Creation of Subcontrollers

Using the *CREATE_CONTROLLER (...)* method, you can create an instance of a subcontroller class from the source code of the higher-level controller. Each subcontroller is assigned an ID (*CONTROLLER_ID*).

When you create a subcontroller, only its method *DO_INIT* is executed.

The subcontroller can be processed from the source code of the main controller using the *CALL_CONTROLLER(...)* method. In this case, only the *DO_REQUEST* method of the subcontroller is processed, but *DO_INITATTRIBUTES* is not (irrespective of what event handler of the main controller is used to call the subcontroller).



```
DATA: sub1 TYPE REF TO
           cl_bsp_controller2.

* CREATE SUB-CONTROLLER
CALL METHOD me->create_controller
 EXPORTING
  controller_name       = 'sub1.do'
  controller_id         = 'subc1'
 RECEIVING
  controller_instance   = sub1.
```

```
CALL METHOD me->call_controller
 EXPORTING
  controller_instance = sub1.
```

Figure 73: Creating the Subcontroller Explicitly and Calling it from the Main Controller

From within the source code of a view, a subcontroller is called using the BSP element `<bsp:goto ...>` or `<bsp:call ...>`; the controller ID is used to identify the controller to be processed. These elements are part of the BSP extension *BSP*, which must already have been included in the page. If you use `<bsp:goto ...>`, any output processed up to this point is discarded. The HTML page will then only contain what was created after you started using this language element. In contrast, `<bsp:call ...>` allows you to combine several views. In this way, you can construct HTML pages that are composed of several areas whose contents are defined using corresponding views.

Hint: For the main controller and subcontroller, the relevant view is usually called in the *DO_REQUEST* method. As a result of the type of call, the results of the view processing are nested in one another. Therefore, you must ensure that the nested processing of the views results in a valid HTML page.

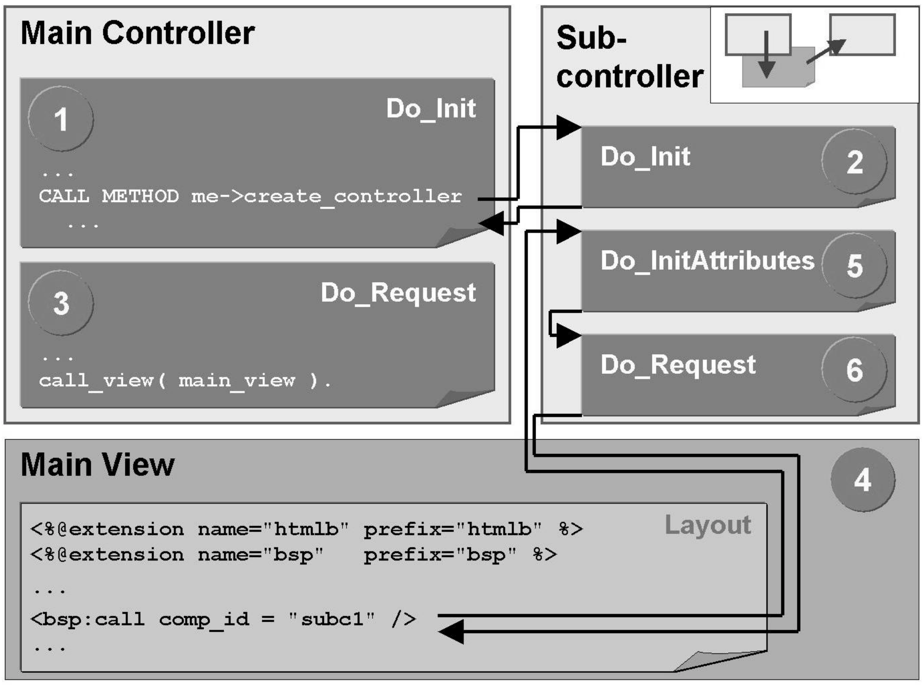

Main Controller
1
Do_Init
...
CALL METHOD me->create_controller
...
3
Do_Request
...
call_view(main_view).
Sub-controller
Do_Init
2
Do_InitAttributes
5
Do_Request
6
Main View
4
Layout
<%@extension name="htmlb" prefix="htmlb" %>
<%@extension name="bsp" prefix="bsp" %>
...
<bsp:call comp_id = "subc1" />
...

Figure 74: Creating the Subcontroller Explicitly and Calling it from the Main View

Implicit Creation of Subcontrollers

When calling a subcontroller using the *<bsp:call>* tag or the *<bsp:goto>* tag, and this subcontroller has not yet been created, it is created implicitly at this point. For this, the parameter ***url*** must also be specified along with the parameter *comp_id*. The former specifies the name of the subcontroller and the path to the subcontroller (if the subcontroller is assigned to a different BSP application, for example).

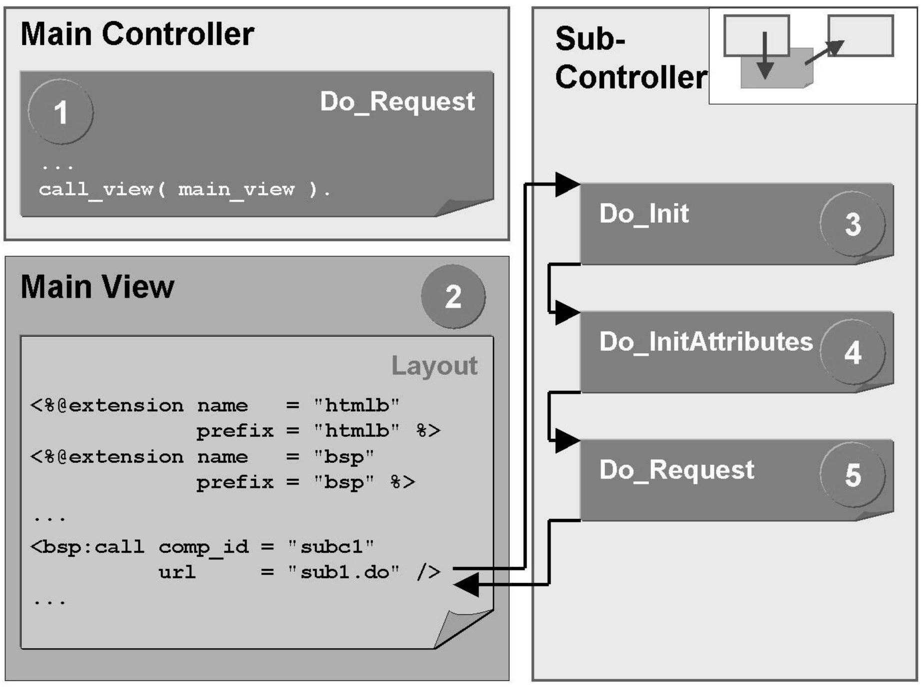


Figure 75: Creating the Subcontroller Implicitly when Calling it from the Main View

Program Flow when Processing User Entries

When using a main controller with a main view and subcontroller with corresponding views, the resulting HTML page consists of a combination of the processed views. Example: Each view of a subcontroller defines a part of a complex form. The individual form sections are joined together by the view of the main controller. If the user now fills in the form fields and sends the form, each controller must contain the data connected to its view. Furthermore, one of the controllers must take care of the event handling. Which controller this is depends on which view contains the HTML element that triggered the event. This view was called by the controller for which event handling is now started. Therefore, the handling of the query string proceeds as follows:

In the main controller, the *DISPATCH_INPUT( )* method must be called to pass the data of the query string to the individual controllers for handling. Which data is to be passed to which controller depends on the names of the relevant name/value pairs in the query string: Each controller is assigned an ID (**CONTROLLER_ID**) when it is created. This ID, followed by an underscore, must be the start of the name in a name/value pair, for that pair to be passed on to the relevant controller. Name/value pairs that cannot be assigned to a subcontroller are passed on to the main controller for processing. Therefore, when creating a form, ensure that all field names have an appropriate prefix so that this assignment is possible. If you use elements of the BSP extensions HTMLB, XHTMLB, and PHTMLB, this is done automatically.

Hint: The *CONTROLLER_ID* cannot contain an underscore, since this is interpreted as a separator between the *CONTROLLER_ID* and the form field.

After the incoming data has been passed on to the individual controllers, their methods are processed as follows:

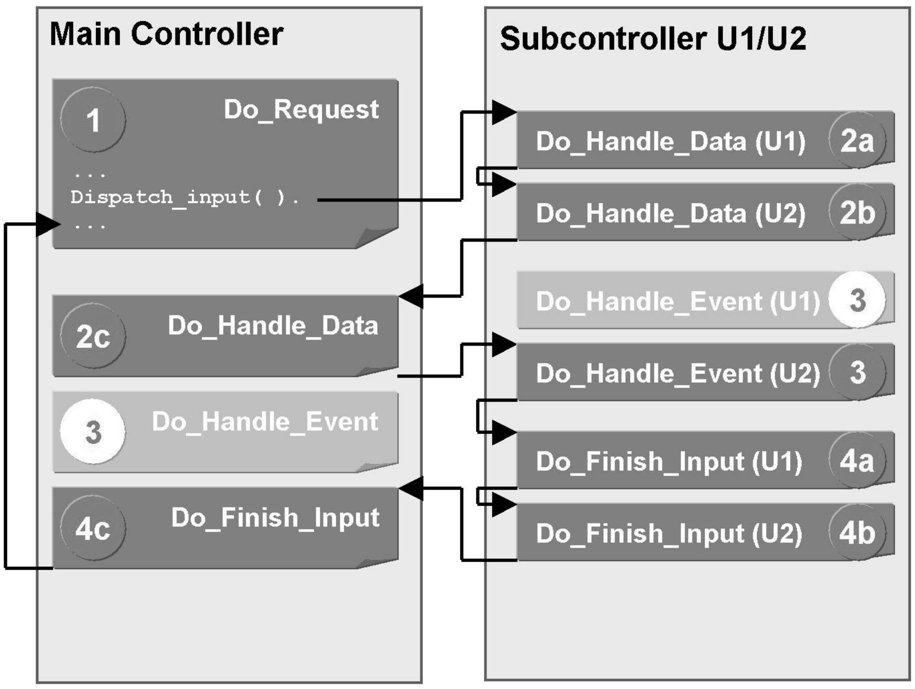


Figure 76: Query string handling for a main controller and two subcontrollers. The HTTP request is triggered using an element assigned to the view of subcontroller U2.

First, the ***DO_HANDLE_DATA*** methods of the individual subcontrollers are called. Then the *DO_HANDLE_DATA* method of the main controller is called. Each controller is assigned its form data using the table *form_fields*.

After this, the ***DO_HANDLE_EVENT*** method is called for one of the subcontrollers or for the main controller. Which controller this is, depends on which view the event was triggered.

Finally, the ***DO_FINISH_INPUT*** method is called first for all the subcontrollers and then for the main controller.

Processing of the *DO_REQUEST* method of the main controller is then continued directly after the *DISPATCH_INPUT( )* call.

Accessing the Data of a Controller

When working with main controllers and subcontrollers, you often have to access the data of one subcontroller from within another. To be able to do this, the main controller must provide you with a reference to the desired subcontroller, so that you can then use the reference variable to access the public attributes and methods of the relevant subcontroller. The main controller holds a table containing the references to all subcontrollers, the contents of which can be ascertained using the ***GET_CONTROLLER*** method of the main controller. In this case, you must also pass the *CONTROLLER_ID*. The address of the main controller is known to every subcontroller (private attribute ***m_parent***).

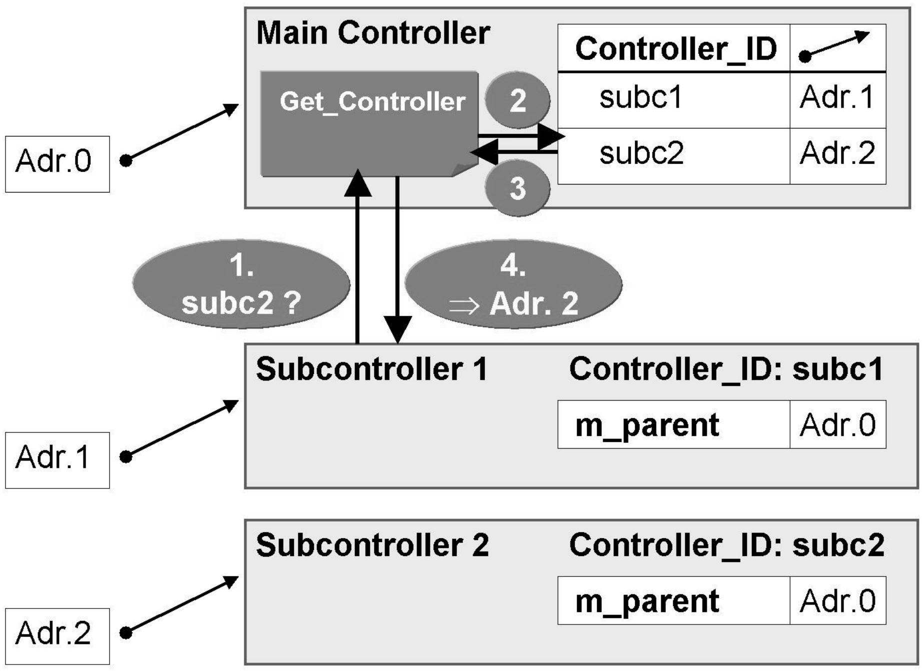
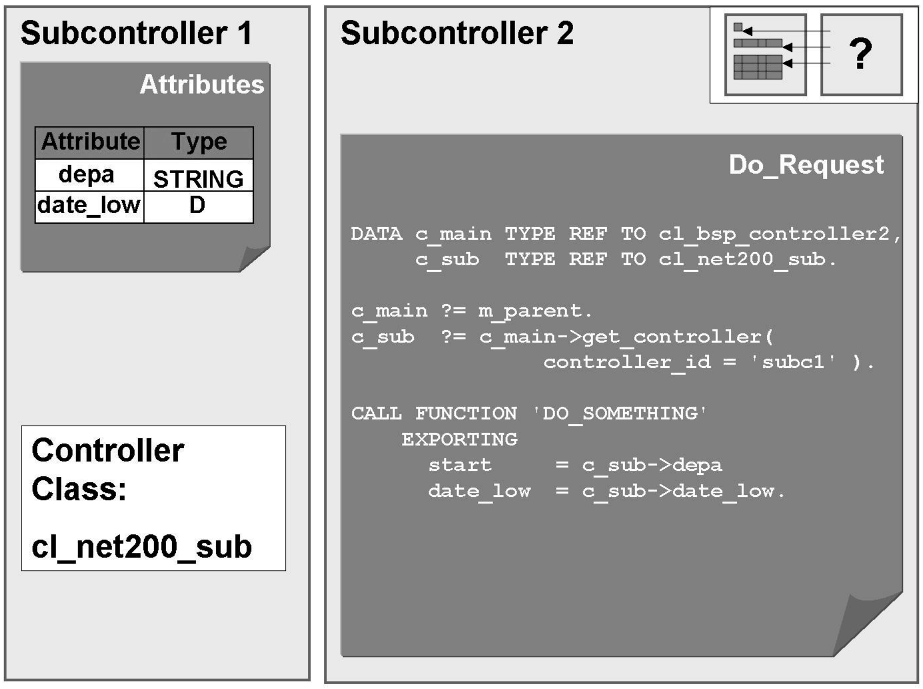


Figure 77: Ascertaining the Address of a Subcontroller



```
DATA c_main TYPE REF TO cl_bsp_controller2,
     c_sub  TYPE REF TO cl_net200_sub.

c_main ?= m_parent.
c_sub  ?= c_main->get_controller(
               controller_id = 'subc1' ).

CALL FUNCTION 'DO_SOMETHING'
    EXPORTING
      start     = c_sub->depa
      date_low  = c_sub->date_low.
```

Figure 78: Accessing the Attributes of One Subcontroller from Within Another

Models

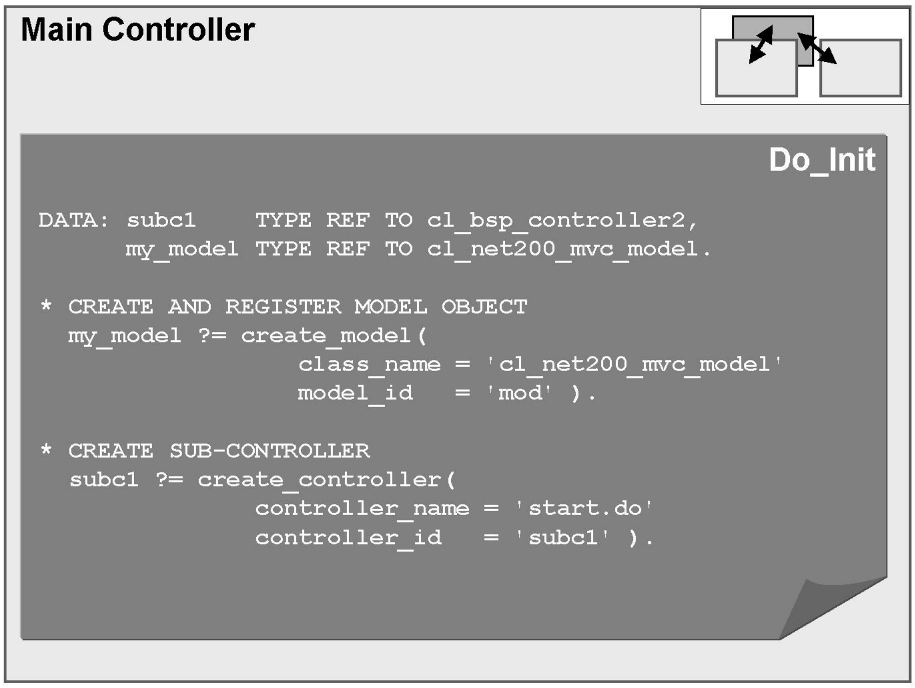
Creating and Using a Model

Using one central model for each BSP application offers great advantages over the data storage for each controller:

- The model serves as a central container for all attributes of the application.
- When transferring data, the controller and the view must exchange only the reference pointing to the model. This means that there is one generic interface for all attributes.
- The model attributes can be addressed from all controllers in the same way.
- Data binding is implemented between the fields of a form that is assigned to a view and the attributes of the model: The user entries are converted to the internal format using the conversion exits defined in the Data Dictionary, a type check is executed, the system checks for fixed values, and the input data is passed to the model attributes.

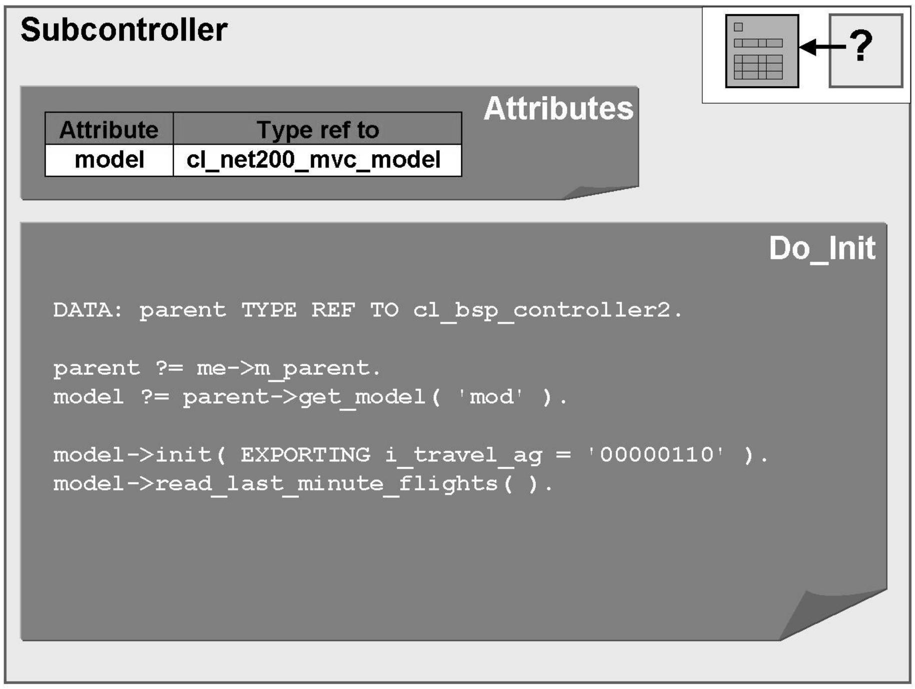
To make use of these advantages, proceed as follows:

1. First, the model class must be created as a global class (Class Builder). The superclass is ***CL_BSP_MODEL*** .
2. All attributes that are connected with the business logic must be added to the model. Additionally, you must add the desired methods to the model, which work with the attributes previously created (for example, a method for filling an internal table, which is then displayed in a view).
3. You must create an object of the model class in the main controller of the application. As is the case when creating a subcontroller, a *MODEL_ID* is assigned when you create a model.
4. To ensure that the subcontrollers can access the attributes of the model created in the main controller, you must ascertain the reference to the model object in all controllers. This can be done in the *DO_INIT* method. Store the reference to the model as an instance attribute of the class. The model address is ascertained in the same way as the address of a controller. However, the ***GET_MODEL*** method is used instead of the *GET_CONTROLLER* method.
5. If data binding between the attributes of the model and the form fields of a view is desired, you must pass a reference pointing to the model object to the view. The values of the form fields must refer to the (public) attributes of the model.



```
DATA: subc1    TYPE REF TO cl_bsp_controller2,
      my_model TYPE REF TO cl_net200_mvc_model.

* CREATE AND REGISTER MODEL OBJECT
  my_model ?= create_model(
                  class_name = 'cl_net200_mvc_model'
                  model_id   = 'mod' ).

* CREATE SUB-CONTROLLER
  subc1 ?= create_controller(
               controller_name = 'start.do'
               controller_id   = 'subc1' ).
```

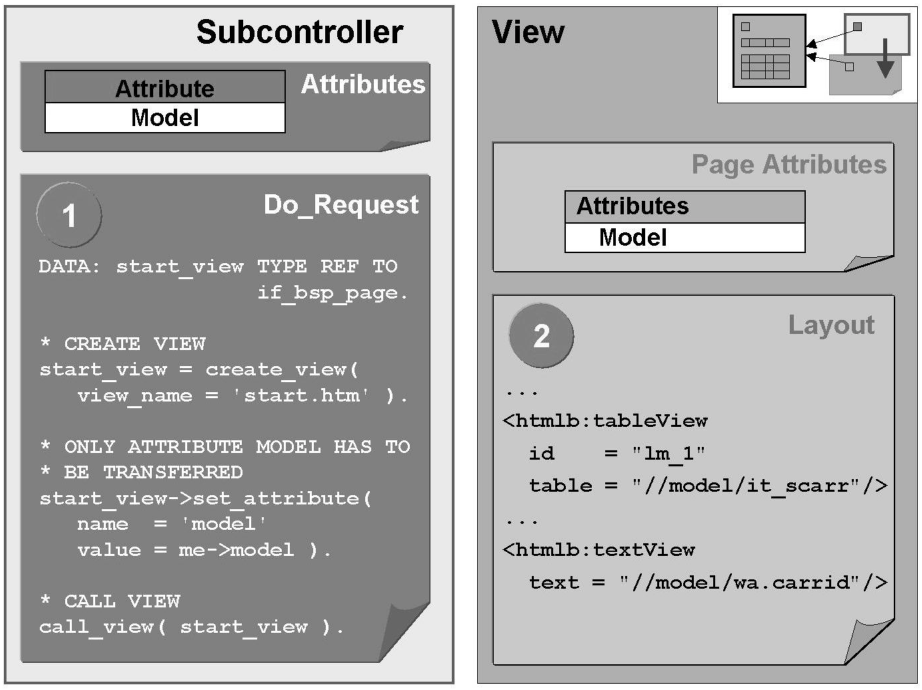
Figure 79: Creating a Model Object



Attribute	Type ref to
model	cl_net200_mvc_model

```
DATA: parent TYPE REF TO cl_bsp_controller2.

parent ?= me->m_parent.
model ?= parent->get_model( 'mod' ).

model->init( EXPORTING i_travel_ag = '00000110' ).
model->read_last_minute_flights( ).
```

Figure 80: Ascertaining the Reference to the Model Object of the Main Controller in a Subcontroller



```
DATA: start_view TYPE REF TO
                 if_bsp_page.

* CREATE VIEW
start_view = create_view(
   view_name = 'start.htm' ).

* ONLY ATTRIBUTE MODEL HAS TO
* BE TRANSFERRED
start_view->set_attribute(
   name  = 'model'
   value = me->model ).

* CALL VIEW
call_view( start_view ).
```

```
...
<htmlb:tableView
  id    = "lm_1"
  table = "//model/it_scarr"/>
...
<htmlb:textView
  text = "//model/wa.carrid"/>
```

Figure 81: Transferring the Reference to the Model Object to a View and Data Binding

Error Handling

Errors that occur during the processing of the source text of a controller can be managed using a message object, as when using the classical BSP programming model. For this purpose, every controller has the attribute *messages*.

When you are working without a model, all errors - including the errors that occur when the user entries are passed to the controller attributes - must be handled manually. In the event handler *DO_HANDLE_DATA*, you can use the method *MESSAGES->ADD_MESSAGE* to add a new error message to the message object of the controller. The message can then be displayed in the view conditionally (*PAGE->MESSAGES->ASSERT_MESSAGE(...)*). For this purpose, the reference to the message object of the controller must be passed to the view.

The main controller and subcontrollers always work with the same message object - that is, all errors that occur in a component are managed by one message object. Since an identically-named form field can appear multiple times on different views of the component, you must, when collecting the messages, ensure that these can be uniquely identified (for example, the field *condition* should contain the field name with a controller prefix).

If the form fields are bound to a model, then the check for the appropriate type of the input takes place in the model (where it is carried out by *GETTER methods*). If a type error occurs, the corresponding error message is managed by the model (private attribute *errors*). In the relevant controller, the global message object is also assigned an error message, which indicates the general problem (problem during data transfer). Additional errors, resulting from the business logic, can be added from the methods of the model using the method call *ERRORS->ADD_MESSAGE*.

If the main controller and subcontroller share a model, the error messages are managed centrally by this model. If each controller works with its own model instance, the error messages are separated according to the controller via the model instances.

To be enable you to reuse the models, you can proceed as follows:

By calling the *GET_ERRORS* method of the model, you can set the reference to the message object of the model. With this reference, you can obtain the individual error messages using the *GET_MESSAGE* method. If the error messages are to be displayed conditionally, do not use the *ASSERT_MESSAGES* method. Instead, you should call the method *IF_BSP_MODEL_BINDING~IS_ATTRIBUE_VALID* of the model. This method returns whether there was an error message for the relevant attribute of the model (*is_valid = 0*). If the message object of the model contains an error message for the relevant attribute, this message is also returned.

Demos, Documentation, and Notes

More information on the topics covered in this lesson is available from the following sources:

The online documentation contains detailed documentation on the SAP Web Application Server and BSP applications. This is accessed in the SAP Help Portal as follows: http://help.sap.com -> SAP NetWeaver -> Release '04 -> SAP NetWeaver -> Application Platform -> ABAP Technology -> UI Technology -> WEB UI Technology -> Business Server Pages

As of Release 6.20, the SAP Web AS features the **BSP application** *BSP_MODEL*. It shows the data binding for the different UI elements. It also shows the error handling in the context of input and output fields.

In the **Software Developer Network (SDN)**, there is a discussion forum for current BSP topics. Individual questions are partly dealt with by the BSP development team. The forum is accessed via: http://sdn.sap.com -> menu entry *Forums* -> Forums -> SDN Forums -> Web Application Server -> Business Server Pages.

In the **OSS**, you can create problem messages and find notes under the area *BC-BSP*.

Lesson Summary

You should now be able to:

- Describe the advantages of the MVC programming paradigm over classic BSP programming
- Create and call controllers, views, and models

Unit Summary

You should now be able to:

- Use elements of the BSP extension HTMLB to design the layout of Business Server Pages
- Process user input made using BSP elements
- Extract data from a query string
- Adapt the design of a BSP application based on the BSP extension HTMLB
- Render complex pages, consisting of several subpages, on the server side
- Create a new BSP extension
- Create new BSP elements
- Describe the class hierarchy for a BSP element
- Encapsulate combinations of different existing BSP elements in a new BSP element
- Describe the advantages of the MVC programming paradigm over classic BSP programming
- Create and call controllers, views, and models

Unit 5

Special Topics

Unit Overview

In this unit you will learn about the options for authentication in the SAP Web Application Server. You will also learn how to search for and use BAPIs to acquire data from other SAP systems. This unit also covers using external editors to adjust the layout. Finally, this unit introduces applications for Smart Forms, sending e-mails from BSP applications, mobile business, and the client functions of the SAP Web Application Server.

Unit Objectives

After completing this unit, you will be able to:

- Describe different logon procedures
- Define services in transaction SICF and there set the logon procedure to be used
- Create anonymous users for a service
- Set up applications with public and protected areas
- Create Internet users dynamically
- Give an overview of RFC and BAPIs
- Give an overview of the important tools in the RFC/BAPI environment
- Call a BAPI in an SAP R/3 back-end system
- Implement utilities, such as external tools, to efficiently develop BSP applications
- Describe which steps are required to send mails from BSP applications, integrate SAP Smart Form documents into BSP applications, and create Web application for mobile devices.

Unit Contents

Lesson: User Concepts and Logon Procedures

Lesson Overview

The SAP Web Application Server supports a range of different logon procedures. Which of the procedures is used depends on the service in question. Furthermore, it is possible to change the user context at runtime.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe different logon procedures
- Define services in transaction SICF and there set the logon procedure to be used
- Create anonymous users for a service
- Set up applications with public and protected areas
- Create Internet users dynamically

Business Example

Certain parts of an Internet application (for example, catalogs) are usually available for all users, whereas a logon is required for other parts (for example, purchase orders).

Support for Different Logon Procedures

The SAP Web Application Server supports different logon procedures:

- The user does not need to log on, that is, the application is accessible. For this purpose, you must preset a user for the appropriate service in transaction SICF.
- The user is queried for his or her user ID and password in a dialog box in the browser. The language and client can be preset or they can be defined at runtime.
- In an accessible page, the HTTP fields sap-language, sap-client, sap-user or sap-alias, and sap-password are filled.
- The users are identified through Single Sign-On (SSO) logon procedures or X 509 client certificates.

If you are using SSO, your users should protect their PCs, when leaving their workplace, against access by other users.

The sequence in which the above mentioned logon procedures are used by the system is preset by the system itself.

- First, the system checks in the corresponding service in SICF whether one of the flags *Logon Data Required* or *Security Requirements: Client Certificate with SSL* is selected. In this case, the logon takes place using the logon data stored, or a logon using the logon with client certificate is expected. Other logon procedures will not be attempted.
- If none of the above mentioned logon procedures is preset, the system will then check whether the fields sap-client, sap-language, sap-user and/or sap-password are evaluated as part of the query string. If all the information is provided, the logon takes place using this data.
  If the BSP application is executed stateful, a temporary cookie is transmitted after successful logon. This cookie contains the session ID. It is used for authentication, unless one of the above procedures is automatically forced. Renewed logon is required only if the Web application is again executed stateless, or the session is terminated explicitly.
- Then the system checks whether an **SSO cookie** is sent with the request. The released BSP application **System** can be used to create this cookie.
- Afterwards, the system checks whether the user has already authenticated himself using **Basic Authentication**. If so, a new logon takes place with this information. The logon information is kept in the HTTP header. This logon data can only be discarded if you close all your browser sessions. For information on how to log on using Basic Authentication, refer to the last item below.
- Finally, the system checks whether the user has got a client certificate and executes the logon using this.
- As a last option, error HTTP 401 is transmitted as an HTTP response to the browser. This means that the browser shows a dialog box in which the user can enter his or her name and password. Specification of the language and client, however, is not possible here.

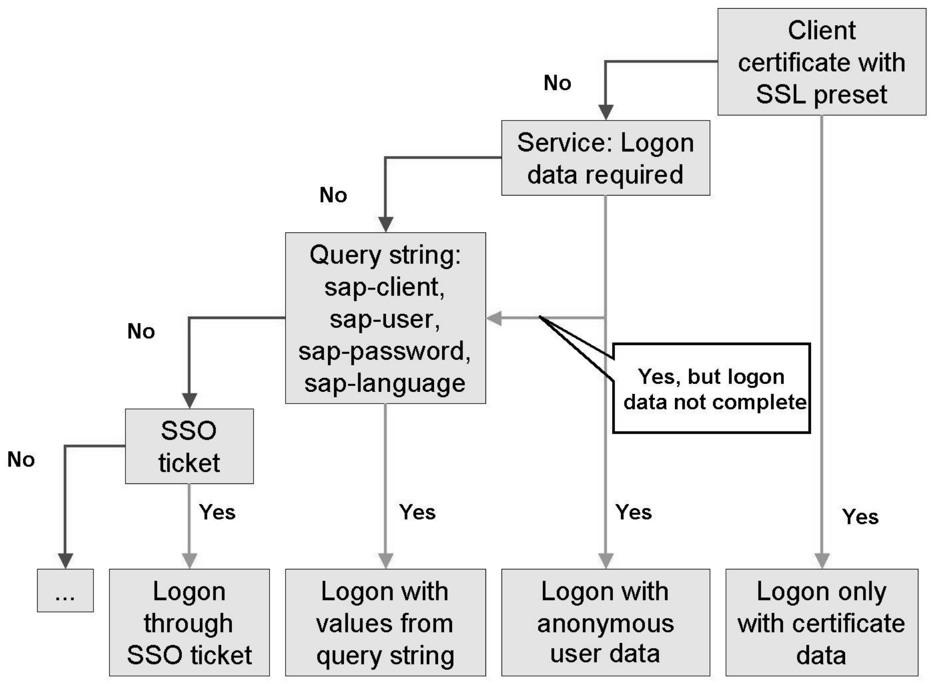

Client certificate with SSL preset
No
Service: Logon data required
No
Query string: sap-client, sap-user, sap-password, sap-language
No
Yes, but logon data not complete
SSO ticket
No
Yes
Yes
Yes
Yes
...
Logon through SSO ticket
Logon with values from query string
Logon with anonymous user data
Logon only with certificate data

Figure 82: Determining the Logon Procedure (1)

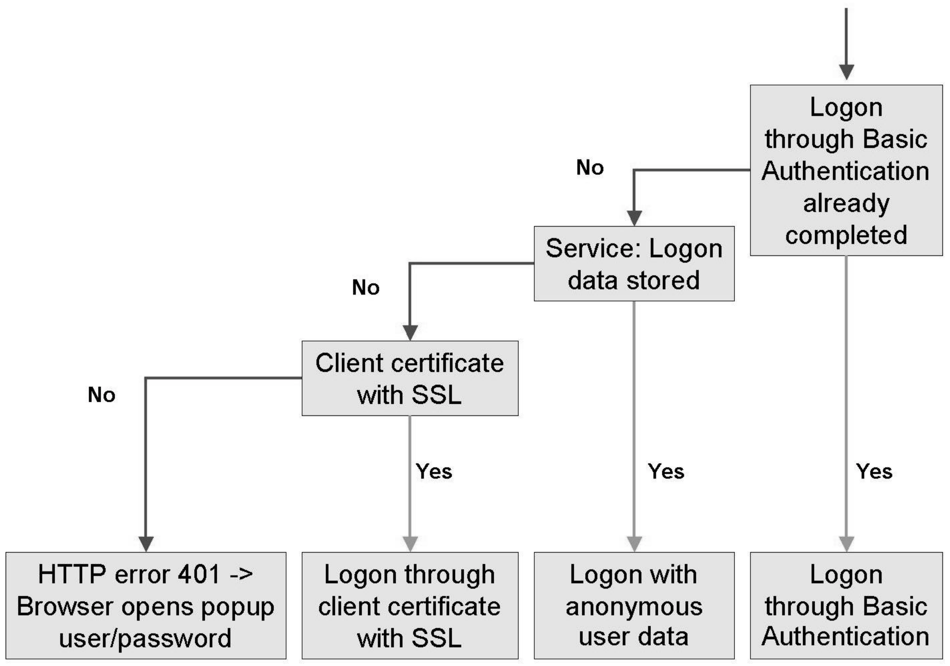

Logon through Basic Authentication already completed
No
Service: Logon data stored
No
Client certificate with SSL
No
Yes
Yes
Yes
HTTP error 401 -> Browser opens popup user/password
Logon through client certificate with SSL
Logon with anonymous user data
Logon through Basic Authentication

Figure 83: Determining the Logon Procedure (2)

Hint: The procedure described above applies to the logon fields Client, User, and Password. The definition of the language also depends on browser settings, user defaults, and the system default language.

Hint: Since SAP Web Application Server Release 6.20, the selected client is kept through a temporary cookie. If the logon procedure Basic Authentication is selected, the client no longer needs to be stored "hard-coded" in the service in SICF, but can be transmitted in the query string (sap-client=<client>) at the first call of a BSP application. This is especially important if one and the same BSP application is to be made available in more than one client.

Service Options

The services are structured hierarchically (tree structures). As of SAP Web Application Server Release 6.20, an appropriate service is created automatically for each BSP application. Not only does the service enable you to enter logon information and to choose a logon procedure, it also provides service-specific authorization. An administrator sets this authorization for the authorization object S_ICF in the SERVICE field (for example, S_ICF-ICF_FIELD = ’SERVICE’ and S_ICF-ICF_VALUE = ’CHECK’) if CHECK is defined as authorization in the service. Of course, further authorization checks can also be executed in the application itself.

Defining an Anonymous User for a Service

If the logon information for client, user, and password are stored in the service, logon can take place at the system without explicit authentication requirements. The user stored in the service is thus also referred to as an anonymous user.

- For this reason, there should be a user set in transaction SU01 as a **service user** and used solely for this purpose. Only then is it possible to change the user context (user switch) when switching to a protected area where explicit authentication is required.
- Assign to the anonymous user only authorizations that are absolutely necessary.
- So that the user cannot log on with a user different from the preset one, the logon data should be marked as required.

Hint: Logon data is not copied to other systems during data transfer, but must be maintained subsequently in each case.

Hint: If you wish to have cascading style sheets, screens, and so on accessible, you must also define an anonymous user for the corresponding services, to which you then assign these MIME objects.

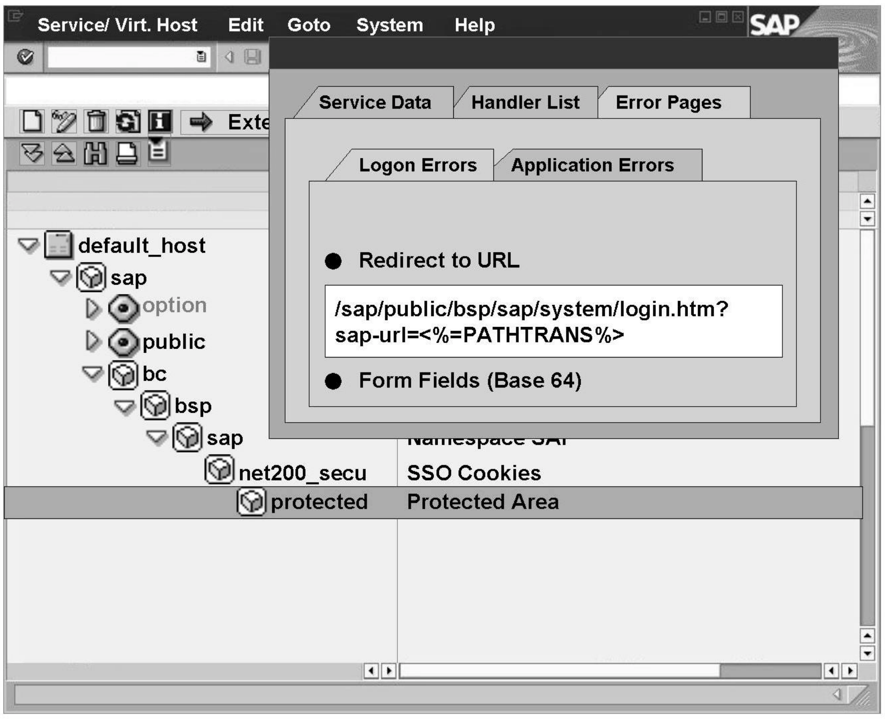
Applications with Public and Protected Areas

One way of splitting up an application into a public and a protected area consists of implementing the application as two BSP applications. The corresponding service for the first BSP application is configured with an anonymous user, while the service for the second BSP application requires a logon.

The second way of splitting up an application into a public and a protected area is to store the BSPs of the application in two subtrees; one branch is public and the other is protected through logon. To do this, you create a subservice for the public area in the appropriate service for the BSP application and store an anonymous user there. Whether or not a subnode needs to be created for the protected area depends on the choice of logon procedure.

One option is to choose the logon procedure Basic Authentication. For this you do not need to create a further node in SICF. As a result, when a page of the protected area is called, a dialog box (HTTP error 404) appears. Here the user enters his or her user name and password. The data is then kept using HTTP header fields. To log on again, all the browser windows need to be closed.

It is also possible to perform logon using **SSO cookies**. To create SSO cookies, you can use the BSP application *SYSTEM*. To do this, you create a node for the protected area of your own BSP application. Under the tab *Error Pages -> Logon Errors*, mark *Redirect to URL* and, in the corresponding field, enter the text literal **`/sap/public/bsp/sap/system/login.htm?sap-url=<%=PATHTRANS%>`**. The result is that, when a BSP assigned to the protected area is called for the first time, an HTTP Redirect to the BSP *login.htm* of BSP application *SYSTEM* will take place. The user, password, and client are queried through a form and, after this data is transmitted, a corresponding SSO cookie is created and passed to the client. Afterwards, through a further HTTP Redirect, the page that was originally requested is called. If more protected pages are called, the system checks whether the SSO cookie is also sent. In this case, a further explicit authentication is not required. Deleting the cookies and, therefore, explicit logoff is possible by using the page *sessionexit.htm* in the BSP application *System*.



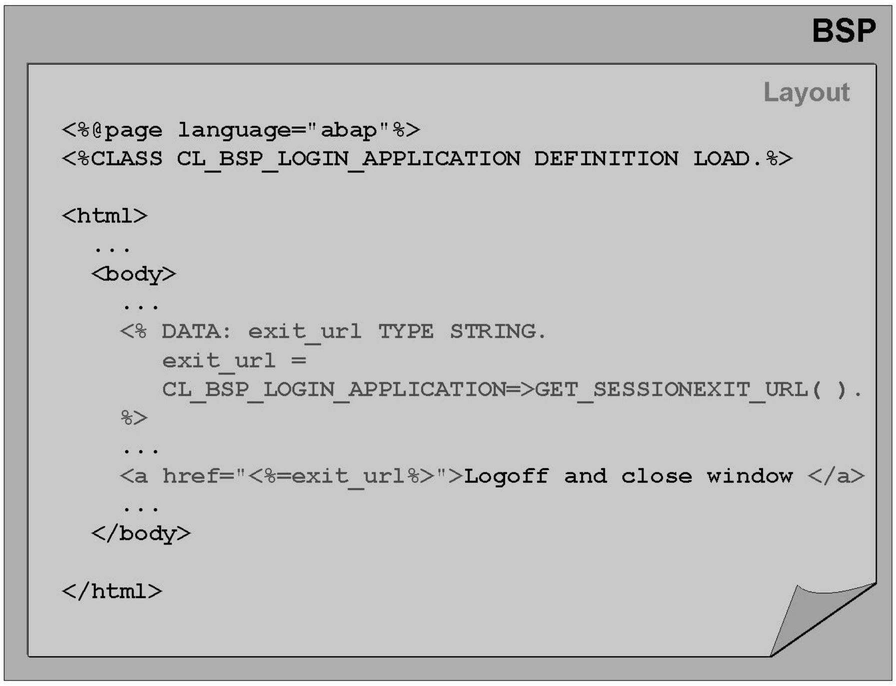
● Redirect to URL

/sap/public/bsp/sap/system/login.htm?
sap-url=<%=PATHTRANS%>

● Form Fields (Base 64)

net200_secu	SSO Cookies
protected	Protected Area

Figure 84: Logon with SSO Cookies



```
<%@page language="abap"%>
<%CLASS CL_BSP_LOGIN_APPLICATION DEFINITION LOAD.%>

<html>
  ...
  <body>
    ...
    <% DATA: exit_url TYPE STRING.
       exit_url =
       CL_BSP_LOGIN_APPLICATION=>GET_SESSIONEXIT_URL( ).
    %>
    ...
    <a href="<%=exit_url%>">Logoff and close window </a>
    ...
  </body>

</html>
```

Figure 85: SSO Cookies: Logoff

An alternative is provided by calling a separate logon dialog at the appropriate place and defining the user using the module *SUSR_INTERNET_USERSWITCH*. Possible errors (user blocked, user invalid in time period, logon data invalid, and so on) are checked and caught by this module.

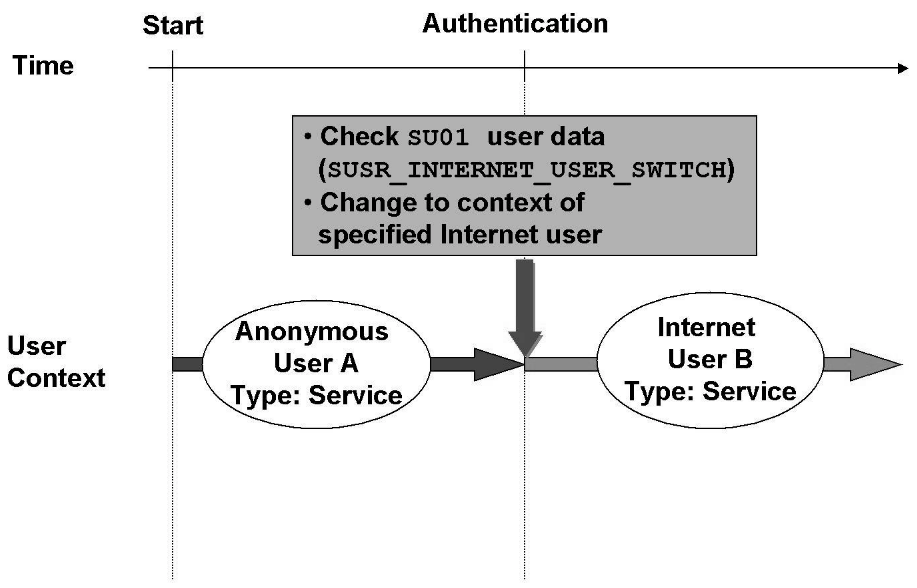

Start
Authentication
Time
• Check SU01 user data (SUSR_INTERNET_USER_SWITCH)
• Change to context of specified Internet user
User Context
Anonymous User A Type: Service
Internet User B Type: Service

Figure 86: Changing the User Context

Hint: Changing the user context can only take place between a user of the type Service (anonymous user) and a user of the type Dialog or Service. Switching the user context between two dialog users not allowed.

In the case of the last two options, it must be ensured on all protected pages that the system checks whether the user change has taken place so that no cross-entry through direct entry of a URL is possible. So that renewed authentication is not necessary for every page of the protected area, the application should be executed stateful after the user switch has taken place.

Creating Internet Users

To get to the protected area of an application, the user must authenticate himself/herself. The corresponding system user, also called Internet user, only exists if it has been created in the Internet beforehand by the system administrator (SU01). If this is not the case, the Internet user must be created at runtime by the external user of the application himself/herself (for example, applicant scenario or online shop). To do this, the function module *SUSR_USER_INTERNET_CREATE* is used. Only few personal data items are entered in this case. The authorizations are assigned through a reference user and are just enough to use the Web application. For more details on this topic, refer to the function module documentation.

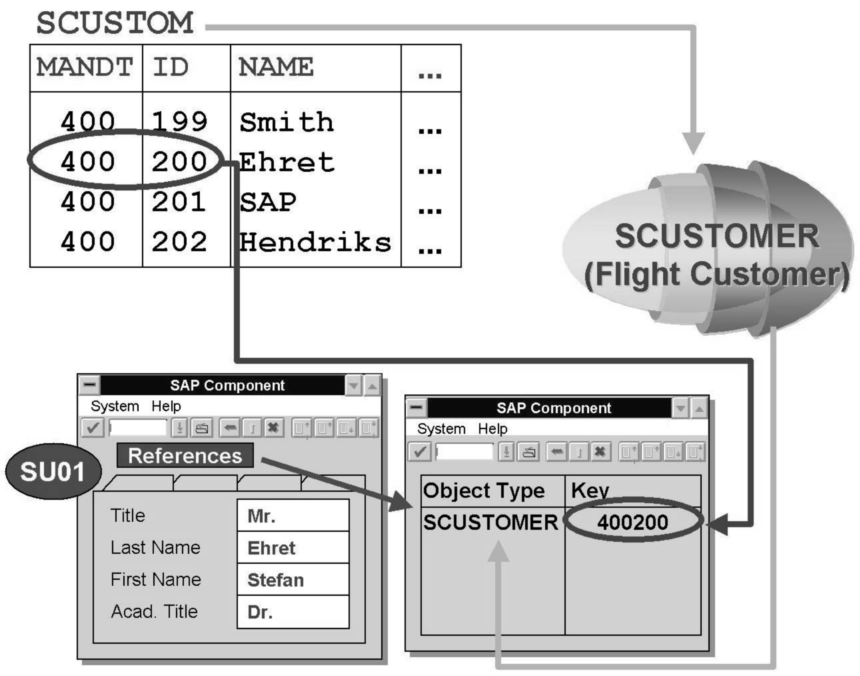
For various classes of Internet users, you may require different reference users with different authorizations respectively (for example, customers versus suppliers).

Links Between Internet Users and Master Data

Internet users can be assigned to their respective master data (applicant, customer, vendor) either by the administrator using the transaction SU01 (Button References) or through supplying the respective interface parameters when the function module *SUSR_USER_INTERNET_CREATE* is called. Generally, the Internet scenario is set up so that after input of personal data in a form, a master record (applicant, customer...) is first created in the corresponding database table (typical name of the function module: *BAPI_<Business-Object>_CREATEFROMDATA*). Afterwards, the Internet user is created - whereby the key information of the master record (for example, client + applicant number / customer number) is passed on.

Several references to business objects can be assigned to an Internet user. This reflects the fact that the user can assume different roles (applicant and customer), but can be managed with a single Internet user.

Additional function modules for changing the password, blocking an Internet user, assigning more business objects, and so on, all begin with the text literal **SUSR_USER** or with the text literal **BAPI_USER**.


SCUSTOM

MANDT	ID	NAME	...
400	199	Smith	...
400	200	Ehret	...
400	201	SAP	...
400	202	Hendriks	...

Title	Mr.
Last Name	Ehret
First Name	Stefan
Acad. Title	Dr.

Object Type	Key
SCUSTOMER	400200

Figure 87: Creating Internet Users

Users in Back-End Systems

In the case of many applications, it is sufficient to have one user preset for the destination for the back-end system. This user must have the appropriate authorization for the BAPIs to be called. However, if you wish to use specific users, then you should consider the use of Single Sign-On (SSO) to avoid the cumbersome logon procedure for each system. SAP systems prior to 4.6D must have a certain minimum Support Package level, and the workplace PlugIn must be installed. For details on this topic, refer to the documentation. The user IDs (not the passwords) must be identical in all SSO systems.

Encrypting the Transferred Data

Through the use of the Secure-Socket-Layer-Protocol (SSL protocol), HTTPS encrypts the data from the client to the server, and vice versa. The prerequisite for this is that the appropriate software is installed and configured. For details, refer to the documentation.

Exercise 9: Security

Exercise Objectives

After completing this exercise, you will be able to:

- Assign pages of a BSP application to protected or public areas
- Create anonymous users for the the public area in SICF
- Use SSO cookies for authentication of pages in the protected area
- Implement explicit logoff when using SSO cookies

Business Example

In the online flight-booking scenario, the user should be able to get information on flights without logging on, but to carry out a booking, he or she will have to go through an authentication process at runtime. SSO cookies should be used to hold the logon information.

Task:

Continue to work with your BSP application or copy the model solution from the last exercise. To do this, copy your BSP application ZNET200_##_13 or the BSP application NET200_S_13, giving it the name ZNET200_##_14. ## stands for your group number. Adhere to the names given and always enter single-digit group numbers with 0#. The model solution for this exercise is NET200_S_14.

1. In your BSP application, delete the pages of the protected area (*protected/...*). Copy the appropriate pages from the template BSP application **NET200_T_14**. The pages are available created by HTML means (*protected/customer.htm, protected/confirm.htm*), and are available created with elements of the BSP extension HTMLB (*protected/customer_htmlb.htm, protected/confirm_htmlb.htm*). Activate the BSPs.
2. Start the transaction SICF. Navigate to the node of your BSP application. Create the subnode for the public area (*public*) and the protected area (*protected*). Assign the Internet user **NET200USER** (password **test**) to the node *public* of your BSP application. For the protected area of the application, set up a check for the use of an SSO cookie. To do this, select the tab *Error Pages -> Logon Error*, and, in the field *Redirect to URL*, enter the text **`/sap/public/bsp/sap/system/login.htm?<%=PATHTRANS%>`**. Activate your application and test it.

Continued on next page

Surprisingly, a dialog box for entering the user data still appears when you call a page of the public area. Why? Remove this problem.

Hint: Not only the BSP called, but also the included subobjects (MIME objects) have to be in a public accessible area. Therefore, the MIME objects in the MIME Repository must also be assigned to a public service.

3. Ensure that the user can again log off explicitly from the protected pages. To do this, create on both pages a transmission button that calls the page *SESSIONEXIT.HTM* of the application *SYSTEM*. Have the URL for calling this page created using the class method **`GET_SESSIONEXIT_URL( )`** of the class **`CL_BSP_LOGIN_APPLICATION`**. Navigation on the page *protected/customer.htm* or *protected/customer_htmlb.htm* should take place dynamically, that is, the navigation to the logoff page should take place in the event handler *OnInputProcessing*. On the page *protected/confirm.htm* or *protected/confirm_htmlb.htm*, however, navigation should be static. The attribute *ACTION* of the *FORM* tag needs to be extended and set with an appropriate value.

Solution 9: Security

Task:

Continue to work with your BSP application or copy the model solution from the last exercise. To do this, copy your BSP application ZNET200_##_13 or the BSP application NET200_S_13, giving it the name ZNET200_##_14. ## stands for your group number. Adhere to the names given and always enter single-digit group numbers with 0#. The model solution for this exercise is NET200_S_14.

1. In your BSP application, delete the pages of the protected area (*protected/...*). Copy the appropriate pages from the template BSP application **NET200_T_14**. The pages are available created by HTML means (*protected/customer.htm, protected/confirm.htm*), and are available created with elements of the BSP extension HTMLB (*protected/customer_htmlb.htm, protected/confirm_htmlb.htm*). Activate the BSPs.
   a) In the navigation area, select the pages *protected/customer.htm, protected/confirm.htm* of your application. With the right mouse key, click *Delete*. Open the BSP application **NET200_T_14**. Copy the BSPs *protected/customer.htm, protected/confirm.htm* or *protected/customer_htmlb.htm, protected/confirm_htmlb.htm* (with the right mouse key, click *Copy*).
2. Start the transaction SICF. Navigate to the node of your BSP application. Create the subnode for the public area (*public*) and the protected area (*protected*). Assign the Internet user **NET200USER** (password **test**) to the node *public* of your BSP application. For the protected area of the application, set up a check for the use of an SSO cookie. To do this, select the tab *Error Pages -> Logon Error*, and, in the field *Redirect to URL*, enter the text **`/sap/public/bsp/sap/system/login.htm?<%=PATHTRANS%>`**. Activate your application and test it.

   Surprisingly, a dialog box for entering the user data still appears when you call a page of the public area. Why? Remove this problem.

   Hint: Not only the BSP called, but also the included subobjects (MIME objects) have to be in a public accessible area. Therefore, the MIME objects in the MIME Repository must also be assigned to a public service.

Continued on next page

a) Transcription SICF. Navigate to your service. Expand the node *default_host* $\rightarrow$ *sap* $\rightarrow$ *bc* $\rightarrow$ *bsp* $\rightarrow$ *sap* $\rightarrow$ *znet200_##_14*. Create two subnodes: With the right mouse key, click *New Subelement*.

   Name: **`Public`** or **`Protected`**

   Type of Service Node: **`Independent Service`**

   Now open the subnode *public*. Maintain the description. Enter the user **`NET200USER`**, password **`test`** in the appropriate fields. Set the checkbox *Logon Data Required*. Leave the fields for client and language empty. Ignore the information message stating the user does not exist.

   Now navigate to the node *protected*. Select the tab *Error Pages -> Logon Error*, and, in the field *Redirect to URL*, enter the literal **`/sap/public/bsp/sap/system/login.htm?<%=PATH-TRANS%>`**. Then select the field *Form Fields (Base64)* so that the form data can be encrypted. Save the data.

   Activate the node of your BSP application with all subnodes. Close all the browser windows. Start your BSP application using the transaction SE80. A user query still appears for the first three pages of the application (classic HTML means). If the user is not entered, you will recognize the reason. The included style sheet (*style.css*) is not assigned to any public service.

   Navigate to the MIME Repository. Open the node *ZNET200_##_14*. Create a subnode called *public*. Copy the MIME object *styles.css* using Drag&Drop into this node and delete it in the node *ZNET200_##_14*.

   Navigate again to SE80. On all pages that were created by classic HTML means, correct the path specification for including the CSS in the tag `<link rel=stylesheet href="...">`. Activate the pages and test the application. Now, no further logon dialog appears for the first three pages of the application. However, when a protected page is displayed for the first time, a query box for the user data and the creation of an SSO cookie will appear.

3. Ensure that the user can again log off explicitly from the protected pages. To do this, create on both pages a transmission button that calls the page *SESSIONEXIT.HTM* of the application *SYSTEM*. Have the URL for calling this page created using the class method **`GET_SESSIONEXIT_URL( )`** of the class **`CL_BSP_LOGIN_APPLICATION`**. Navigation on the page *protected/customer.htm* or *protected/customer_htmlb.htm* should take place dynamically, that is, the navigation to the logoff page should

Continued on next page

take place in the event handler *OnInputProcessing*. On the page *protected/confirm.htm* or *protected/confirm_htmlb.htm*, however, navigation should be static. The attribute *ACTION* of the *FORM* tag needs to be extended and set with an appropriate value.

a) For the source code, see below. Save and test the BSP application. After you have triggered the transmission button, the SSO cookie will be deleted. A dialog box queries whether the current browser window is to be closed as well.

protected/customer.htm - Layout

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

<html>
  ...
  <body>
    ...
<!---------------------------------------------------->
<!-- Submit: Customer  Info                        -->
<!---------------------------------------------------->
      <br>
      <input type="submit"
             name="OnInputProcessing(BOOK)"
             value="<%=otr(NET200/BOOK)%>">
      <input type="submit"
             name="OnInputProcessing(GETCUSTOMER)"
             value="<%=otr(NET200/GET_SAP_CUSTOMER_DATA)%>">
      <input type=submit
             name="OnInputProcessing(EXIT)"
             value="<%= otr(sotr_vocabulary_basic/exit) %>">
    </form>
  </body>
</html>
```

protected/customer.htm - OnInputProcessing

```
* event handler for checking and processing user input and
* for defining navigation

...
```

Continued on next page

```
CASE event_id.

************************ WHEN BOOK ******************
  WHEN 'BOOK'.
    ...
************************ WHEN GETCUSTOMER ***********
  WHEN 'GETCUSTOMER'.

************************ WHEN EXIT ******************
  WHEN 'EXIT'.
    CLASS cl_bsp_login_application DEFINITION LOAD.
    DATA: exit_url TYPE string.
    exit_url = cl_bsp_login_application=>get_sessionexit_url( ).
* destroy SSO-cookie and end session in SAP system
    navigation->goto_page( exit_url ).

ENDCASE.
```

protected/customer_htmlb.htm - Layout

```
<%@extension name="htmlb" prefix="htmlb" %>
<htmlb:content design="DESIGN2002" >
  <htmlb:document>
    ...
      <htmlb:form>
        ...
          <htmlb:gridLayoutCell columnIndex = "1"
                                rowIndex    = "16"
                                colSpan     = "2">
            <htmlb:button
                   id      = "submit1"
                   text    = "<%= otr(net200/Book) %>"
                   onClick = "book"/>
            <htmlb:button
                   id      = "submit2"
                   text = "<%= otr(net200/Get_SAP_customer_data) %>"
                   onClick = "getcustomer"/>
            <htmlb:button
                   id      = "submit3"
                   text = "<%= otr(sotr_vocabulary_basic/exit) %>"
                   onClick = "exit"/>
```

Continued on next page

```
          </htmlb:gridLayoutCell>
        </htmlb:gridLayout>
      </htmlb:form>
    </htmlb:documentBody>
  </htmlb:document>
</htmlb:content>
```

protected/customer.htm - OnInputProcessing

```
* event handler for checking and processing user input and
* for defining navigation

...

IF event_id = cl_htmlb_manager=>event_id.

  DATA: event TYPE REF TO cl_htmlb_event.
  event = cl_htmlb_manager=>get_event( runtime->server->request ).

  CASE event->server_event.


************************* WHEN BOOK *******************
    WHEN 'book'.
      ...

************************* WHEN GETCUSTOMER ************
    WHEN 'getcustomer'.

************************* WHEN EXIT *******************
    WHEN 'exit'.
      CLASS cl_bsp_login_application DEFINITION LOAD.
      DATA: exit_url TYPE string.
      exit_url = cl_bsp_login_application=>get_sessionexit_url( ).
* destroy SSO-cookie and end session in SAP system
      navigation->goto_page( exit_url ).

  ENDCASE.

ENDIF.
```

Continued on next page

protected/confirm.htm - Layout

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

<html>
  ...
  <body>
    ...
<%-- static navigation: OnInputProcessing not needed --%>
    <form action="<%=exit_url%>">
      <input type=submit
             name="exit"
             value="<%= otr(sotr_vocabulary_basic/exit) %>"
    </form>
  </body>

</html>
```

protected/confirm_htmlb.htm - Layout

```
<%@extension name="htmlb" prefix="htmlb" %>
<htmlb:content design="DESIGN2002" >
  <htmlb:document>
    ...
      <htmlb:form action="<%= exit_url %>" >
        ...
          <htmlb:gridLayoutCell columnIndex = "1"
                                rowIndex    = "4"
                                colSpan     = "2" >
            <htmlb:button
                   id      = "exit"
                   text    = "<%= otr(sotr_vocabulary_basic/exit) %>"
                   onClick = "exit" />
          </htmlb:gridLayoutCell>
        </htmlb:gridLayout>
      </htmlb:form>
    </htmlb:documentBody>
  </htmlb:document>
</htmlb:content>
```

Continued on next page

protected/confirm.htm / protected/confirm_htmlb.htm - OnInitialization

```
* event handler for data retrieval

CLASS cl_bsp_login_application DEFINITION LOAD.
exit_url = cl_bsp_login_application=>get_sessionexit_url( ).

it_book_id = cl_net200s_final=>booking_numbers.
```

Lesson Summary

You should now be able to:

- Describe different logon procedures
- Define services in transaction SICF and there set the logon procedure to be used
- Create anonymous users for a service
- Set up applications with public and protected areas
- Create Internet users dynamically

Related Information

- http://service.sap.com/security

Lesson: Connecting to SAP Systems Through RFC

Lesson Overview

You can use the functions of SAP R/3 and other SAP components from within the SAP Web Application Server to create Web applications for these components.

Lesson Objectives

After completing this lesson, you will be able to:

- Give an overview of RFC and BAPIs
- Give an overview of the important tools in the RFC/BAPI environment
- Call a BAPI in an SAP R/3 back-end system

Business Example

For security reasons, the Web application should not be implemented in the system where the business data is stored. The BSP application should be created in an SAP Web Application Server while the business data is stored in the database of a separate SAP component. Therefore, business data must be exchanged between the systems.

System Landscape

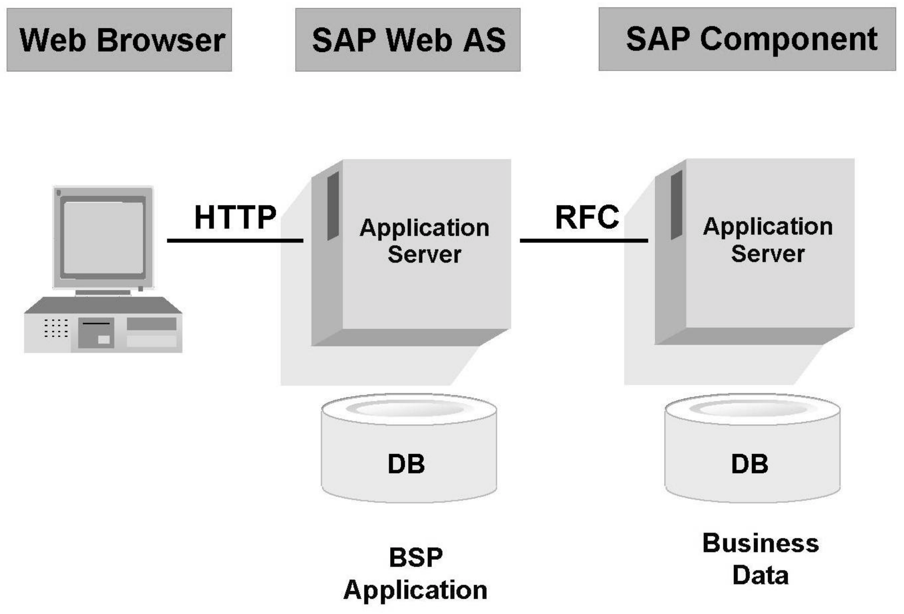


Figure 88: System Landscape

Overview of RFC and Tools

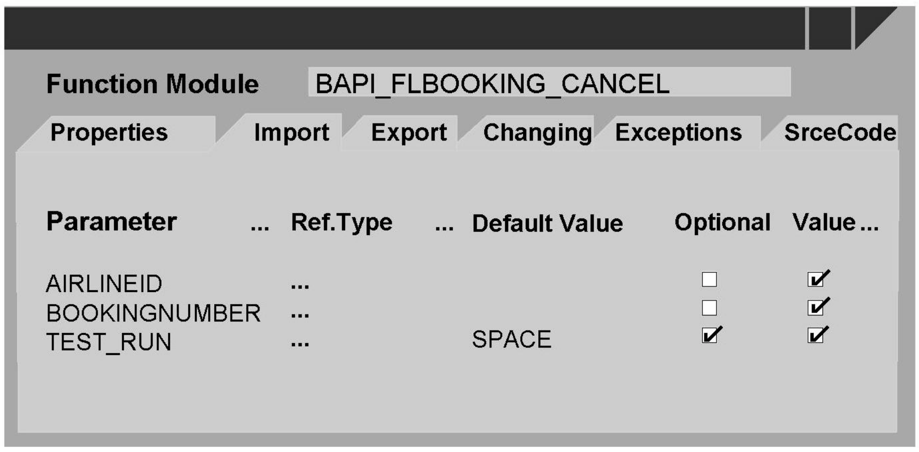
All the ABAP function modules (FMs) are managed in the SAP Function Builder (transaction code SE37). A certain quantity of these FMs are Remote Function Call-enabled (RFC-enabled) and can thus be addressed from outside. These RFC-enabled modules are called **RFM**s (RFC-enabled function modules). RFMs are subject to additional rules - for example, no changing parameters can be used in them.

An RFM interface consists of import, export, and table parameters, as well as defined exceptions. Import and export parameters are usually either simple fields based on a Dictionary definition (for example, KNA1-KUNNR) or structures that consist of fields themselves. All export parameters are optional; import and table parameters can also be mandatory.

Simple fields are also described as scalar parameters. Optional scalar import parameters can contain a default value that is used whenever the caller does not use the parameter.

Within the Function Builder, you can navigate to the respective Dictionary definitions of all parameters by double-clicking the appropriate area.

The most important attributes here are the data type, the length, the conversion routine that may have been used, and the allowed value set.



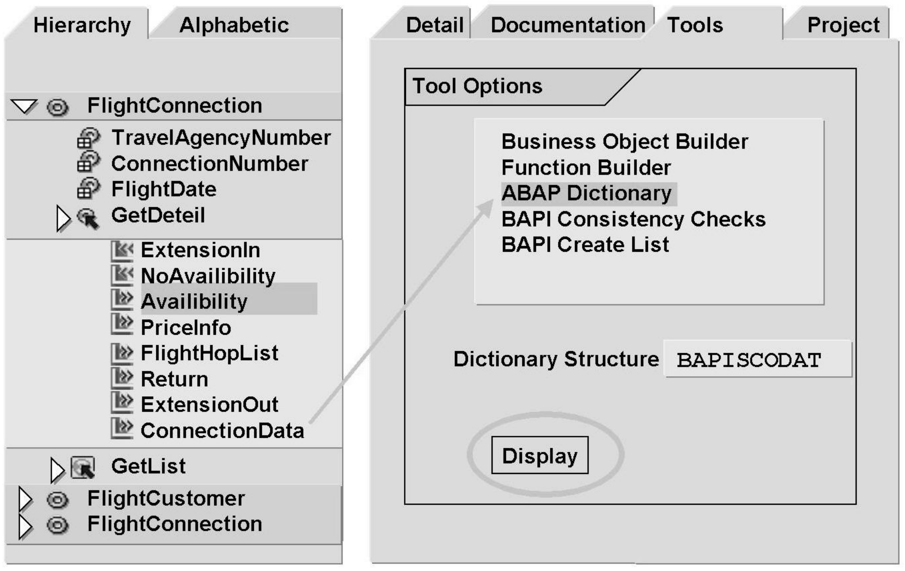
Function Module BAPI_FLBOOKING_CANCEL

Parameter	...	Ref.Type	...	Default Value	Optional	Value ...
AIRLINEID		...			☐	✓
BOOKINGNUMBER		...			☐	✓
TEST_RUN		...		SPACE	✓	✓

Figure 89: Displaying a Function Module Using Transaction SE37

Overview of BAPIs and Tools

BAPIs (Business Application Programming Interfaces) are a subset of RFMs. You must develop BAPIs in accordance with the guidelines defined in the BAPI Programming Guide, and you must define them as methods of object types in the Business Object Repository (BOR).



Dictionary Structure BAPISCODAT

Figure 90: The BAPI Explorer

Here is a list of the most important BAPI specifications:

- Released BAPIs are the standard interfaces for synchronous access to SAP.
- Customers can develop their own BAPIs. Attach new BAPIs to a subclass; do not modify SAP object types.
- BAPIs are usually upwards compatible and well-documented.
- You call BAPIs in ABAP by programming a CALL FUNCTION to the respective RFM.
- BAPIs generally have no exceptions.
- The BOR object types usually have key fields that are used for accessing the relevant tables.
- If you have instance-dependent BAPIs, the key fields appear at the RFC level as import parameters.
- If you have instance-creating BAPIs, the key fields appear at the RFC level as export parameters.
- Instance-independent BAPIs do not use key fields.
- BAPIs can have different parameter names in BOR than in the Function Builder.
- Table parameters can also be marked in the BOR as import, export, or import/export. This has no effect on calling the RFM.
- As of release 4.0, BAPIs that execute database changes should no longer contain any COMMIT WORK statement. The application must implement the external Commit using *BapiService.TransactionCommit*.
- BAPIs mainly use internal data representation. The GUI conversion routines are not automatically called. SAP provides conversion BAPIs so that the application can convert between internal and external formats.
- There are special BAPIs of the help-value object type available for interpreting keys (for example, country keys) and for providing value sets for input fields.

Calling a BAPI in a Back-End System

The SAP Web Application Server does not know the metadata of the BAPIs in the back-end systems. To avoid manual input, the Object Navigator (SE80) provides a mini-BAPI browser that can display the metadata for BAPIs and other RFMs as well as generate an appropriate type definition, and so on.

You start the BAPI browser from within the ABAP Editor using the menu *Goto → BAPI Browser*. You can then copy these definitions into your own application. The mini-BAPI browser works on the basis of RFM names so that it is best to first ascertain these names in the BAPI browser of the backend system.

The backend systems used must be defined as RFC destinations (transaction SM59). Very often, a standard user with sufficient authorizations is defined here so that not every user of the Web application needs to be defined in the back-end system.

The destination that you use in the source text should ideally be a logical one to which you then, as an administrator, assign a corresponding physical destination.

Exercise 10: Connections Using RFC

Exercise Objectives

After completing this exercise, you will be able to:

- Operate the BAPI browser
- Use BAPIs in the back-end system

Business Example

If you wish to develop applications that are to access the application logic of SAP components, you do so using the RFC connection. As a rule, BAPIs are called.

In the self-service flight booking application, the user wishes to book a flight. He or she is already a customer. Using his or her customer number, the customer can read his or her data in the SAP R/3 system. The corresponding input fields, such as name, are automatically filled in the BSP.

Task:

Continue to work with your BSP application or copy the model solution from the last exercise. For this purpose, copy the NET200_S_14 to the name ZNET200_##_15; ## is your group number. Adhere to the names given and always enter single-digit group numbers with 0#. The model solution for this exercise is NET200_S_15.

1. After you have entered the customer number in the appropriate field on the page *protected/customer.htm*or *protected/customer_htmlb.htm* and have clicked the corresponding transmission button, details on this customer should be read from table KNA1 of the back-end system. For this, use an appropriate BAPI. Beforehand, convert the customer number into the internal format (leading zeroes). Proceed as follows:

   In the event handler *OnInputProcessing*, call the function module *BAPI_CONVERSION_EXT2INT1* in the back-end system (destination: **NET200_RFC**). Generate the module call using the BAPI browser. Also copy the required types and data objects from the display in the BAPI browser. Pass an internal table with the values to be converted to the interface parameter *DATA*. Fill a line of this internal table with the customer number. To do this, create a suitable work area (line type identical with the internal table) and set the subfield of the work area.

Continued on next page

OBJTYPE	METHOD	PARAMETER	FIELD	EXT_FOR-MAT
'KNA1'	'Create'	'Customer'	'Customer'	customer_no

After the function module is called, the formatted customer number is in the column *INT_FORMAT* of the internal table. Pass this value to the data field *CUSTOMER_NO*.

2. In the same way, call the function module that belongs to the BAPI *CUSTOMER.GETDETAIL1* in the back-end system. Here, too, copy the types and data object definitions from the display in the BAPI browser.

Hint: Only the interface parameters *CUSTOMERNO*, *PE_PERSONALDATA*, and *RETURN* are required.

If successful (*RETURN-TYPE* = *'S'* or *RETURN-TYPE* = *' '*), pass on the detailed information and the page attributes of the BSP as follows:

PE_PERSONALDATA-	Page attribute name
TITLE_P	FORM
FIRSTNAME	FIRST_NAME
LASTNAME	LAST_NAME
POSTL_COD1	POSTAL_NUM
CITY	CITY
COUNTRY	COUNTRY

If an error occurs while the data is being read, pass on the error message *RETURN-MESSAGE* to the page attribute *ERR_MSG* instead. Activate the BSPs and test the BSP application.

Solution 10: Connections Using RFC

Task:

Continue to work with your BSP application or copy the model solution from the last exercise. For this purpose, copy the NET200_S_14 to the name ZNET200_##_15; ## is your group number. Adhere to the names given and always enter single-digit group numbers with 0#. The model solution for this exercise is NET200_S_15.

1. After you have entered the customer number in the appropriate field on the page *protected/customer.htm*or *protected/customer_htmlb.htm* and have clicked the corresponding transmission button, details on this customer should be read from table KNA1 of the back-end system. For this, use an appropriate BAPI. Beforehand, convert the customer number into the internal format (leading zeroes). Proceed as follows:

   In the event handler *OnInputProcessing*, call the function module *BAPI_CONVERSION_EXT2INT1* in the back-end system (destination: **NET200_RFC**). Generate the module call using the BAPI browser. Also copy the required types and data objects from the display in the BAPI browser. Pass an internal table with the values to be converted to the interface parameter *DATA*. Fill a line of this internal table with the customer number. To do this, create a suitable work area (line type identical with the internal table) and set the subfield of the work area.

OBJTYPE	METHOD	PARAMETER	FIELD	EXT_FOR-MAT
'KNA1'	'Create'	'Customer'	'Customer'	customer_no

   After the function module is called, the formatted customer number is in the column *INT_FORMAT* of the internal table. Pass this value to the data field *CUSTOMER_NO*.

   a) Have the page *protected/customer.htm* or *protected/customer_htmlb.htm* displayed in SE80. Edit the event handler *OnInputProcessing*. Position the cursor in the case distinction after the line `WHEN 'GETCUSTOMER'` or `WHEN 'getcustomer'`. Start a second mode and open any BSP in transaction SE80. Here you can start the BAPI browser through the menu path *Goto → BAPI Browser*. Search for the entry **NET200_RFC** and enter the name of the function module there. You can display types, data objects, and the call by double-clicking them. Select the source code and copy this into the source code of your page that is displayed in the first mode.

Continued on next page

The source code is shown below.

protected/customer.htm - OnInputProcessing

```
* event handler for checking and processing user input and
* for defining navigation

DATA:
   customer_id TYPE s_customer,
   book_id     TYPE s_book_id,
   it_book_id  TYPE TABLE OF s_book_id,
   wa_hop_list LIKE LINE OF it_flight_hop_list,
   counter     TYPE s_countnum,
   cust_data   TYPE bapiscunew,
   wa_return   TYPE bapiret2,
   it_return   TYPE TABLE OF bapiret2.


  CASE event_id.

************************* WHEN BOOK *******************
    WHEN 'BOOK'.
      ...
************************* WHEN GETCUSTOMER ************
    WHEN 'GETCUSTOMER'.
* Fill internal table for data format conversion
      DATA: t_data   TYPE TABLE OF bapiconvrs.
      DATA: wa_data  LIKE LINE  OF t_data.
      DATA: t_return TYPE TABLE OF bapiret2.

      wa_data-objtype    = 'KNA1'.
      wa_data-method     = 'Create'.
      wa_data-parameter  = 'Customer'.
      wa_data-field      = 'Customer'.
      wa_data-ext_format = customer_no.
      APPEND wa_data TO t_data.

* Convert Customer Number to internal format
      CALL FUNCTION 'BAPI_CONVERSION_EXT2INT1'
        DESTINATION 'NET200_RFC'
        TABLES
          data   = t_data
          return = t_return.
```

Continued on next page

```
* Write back converted Customer Number to data field
      READ TABLE t_data INDEX 1 INTO wa_data.
      customer_no = wa_data-int_format.

************************* WHEN EXIT *******************
    WHEN 'EXIT'.
      ...
  ENDCASE.

ENDIF.
```

protected/customer_htmlb.htm - OnInputProcessing

```
* event handler for checking and processing user input and
* for defining navigation

DATA:
   customer_id TYPE s_customer,
   book_id     TYPE s_book_id,
   it_book_id  TYPE TABLE OF s_book_id,
   wa_hop_list LIKE LINE OF it_flight_hop_list,
   counter     TYPE s_countnum,
   cust_data   TYPE bapiscunew,
   wa_return   TYPE bapiret2,
   it_return   TYPE TABLE OF bapiret2.


IF event_id = cl_htmlb_manager=>event_id.

  DATA: event TYPE REF TO cl_htmlb_event.
  event = cl_htmlb_manager=>get_event( runtime->server->request ).

  CASE event->server_event.

************************* WHEN BOOK *******************
    WHEN 'book'.
      ...
************************* WHEN GETCUSTOMER ************
    WHEN 'getcustomer'.
* Fill internal table for data format conversion
      DATA: t_data   TYPE TABLE OF bapiconvrs.
      DATA: wa_data  LIKE LINE  OF t_data.
```

Continued on next page

```
      DATA: t_return TYPE TABLE OF bapiret2.

      wa_data-objtype    = 'KNA1'.
      wa_data-method     = 'Create'.
      wa_data-parameter  = 'Customer'.
      wa_data-field      = 'Customer'.
      wa_data-ext_format = customer_no.
      APPEND wa_data TO t_data.

* Convert Customer Number to internal format
      CALL FUNCTION 'BAPI_CONVERSION_EXT2INT1'
        DESTINATION 'NET200_RFC'
        TABLES
          data   = t_data
          return = t_return.

* Write back converted Customer Number to data field
      READ TABLE t_data INDEX 1 INTO wa_data.
      customer_no = wa_data-int_format.

************************* WHEN EXIT *******************
    WHEN 'exit'.
      ...
  ENDCASE.

ENDIF.
```

2. In the same way, call the function module that belongs to the BAPI *CUSTOMER.GETDETAIL1* in the back-end system. Here, too, copy the types and data object definitions from the display in the BAPI browser.

   Hint: Only the interface parameters *CUSTOMERNO*, *PE_PERSONALDATA*, and *RETURN* are required.

   If successful (*RETURN-TYPE = 'S'* or *RETURN-TYPE = ' '*), pass on the detailed information and the page attributes of the BSP as follows:

PE_PERSONALDATA-	Page attribute name
TITLE_P	FORM
FIRSTNAME	FIRST_NAME

Continued on next page

LASTNAME	LAST_NAME
POSTL_COD1	POSTAL_NUM
CITY	CITY
COUNTRY	COUNTRY

If an error occurs while the data is being read, pass on the error message *RETURN-MESSAGE* to the page attribute *ERR_MSG* instead. Activate the BSPs and test the BSP application.

a)

protected/customer.htm / protected/customer_htmlb.htm - OnInputProcessing

```
* event handler for checking and processing user input and
* for defining navigation

      ...
************************* WHEN GETCUSTOMER ************
      ...

      CALL FUNCTION 'BAPI_CONVERSION_EXT2INT1'
        DESTINATION 'NET200_RFC'
        TABLES
          data   = t_data
          return = t_return.

      READ TABLE t_data INDEX 1 INTO wa_data.
      customer_no = wa_data-int_format.

* Get Details to selected Customer Number
      DATA pe_personaldata TYPE bapikna101_1.
      DATA return          TYPE bapireturn1.

      CALL FUNCTION 'BAPI_CUSTOMER_GETDETAIL1'
        DESTINATION 'NET200_RFC'
        EXPORTING
          customerno      = customer_no
        IMPORTING
          pe_personaldata = pe_personaldata
          return          = return.

* Transferr details to page attributes
```

Continued on next page

```
IF return-type <> ' ' AND return-type <> 'S'.
  err_msg = return-message.
ELSE.
  err_msg    = space.
  form       = pe_personaldata-title_p.
  first_name = pe_personaldata-firstname.
  last_name  = pe_personaldata-lastname.
  postal_num = pe_personaldata-postl_cod1.
  city       = pe_personaldata-city.
  country    = pe_personaldata-country.
ENDIF.
```

Lesson Summary

You should now be able to:

- Give an overview of RFC and BAPIs
- Give an overview of the important tools in the RFC/BAPI environment
- Call a BAPI in an SAP R/3 back-end system

Lesson: Utilities for Creating BSP Applications

Lesson Overview

In this training unit, we will show you how utilities for creating Business Server Pages (for example, external tools) can be used.

Lesson Objectives

After completing this lesson, you will be able to:

- Implement utilities, such as external tools, to efficiently develop BSP applications

Business Example

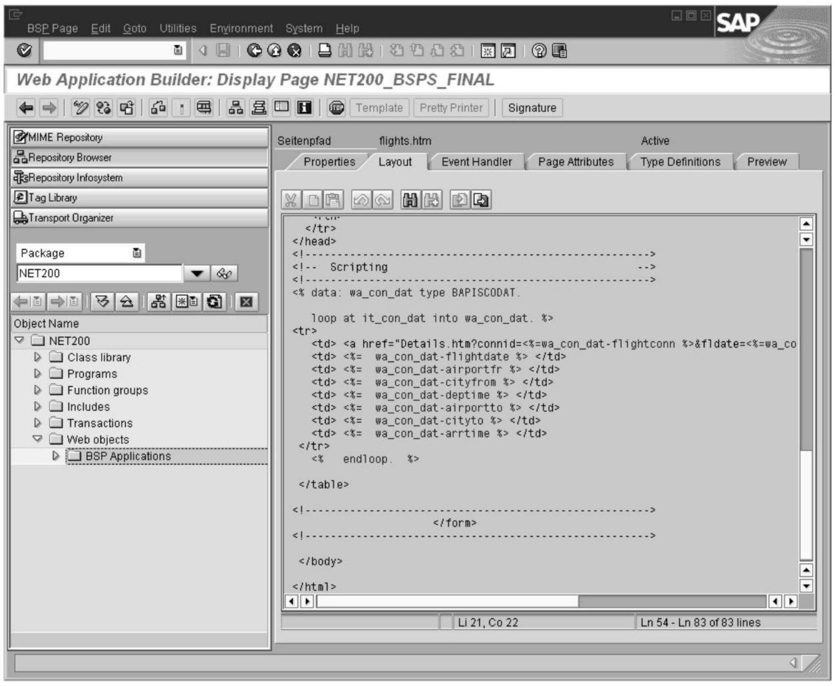
Developing BSPs will be made much more efficient if you use utilities such as Pretty Printer, Tag Library, or the standard publishing tools.

Web Application Builder

In order to be able to process a BSP application and the Business Server Pages contained therein, use the tool **Web Application Builder**. The Web Application Builder is integrated in the Object Navigator (Transaction SE80) and triggered automatically as soon as you commence processing (creating or changing a BSP application or a Business Server Page).

To process the Business Server Pages, you have an built-in editor at your disposal. The event handlers, which contain ABAP source code only, have the full range of ABAP Editor functions - for example, Template or Pretty Printer. The ABAP Editor functions are only available to a limited degree for the layout implementation of a page that may contain HTML, ABAP, as well as JavaScript language elements. For example, there is a syntax check available for ABAP scripting parts of a page, but there is no syntax check for other implemented languages (HTML, JavaScript).

As of SAP Web AS 6.20 with Support Package 3, you also have Pretty Printer available in the layout of pages with flow logic, views, and page fragments. In this way, the individual page components (ABAP scripting, JavaScript scripting, static source code, BSP extensions) can be aligned clearly in relationship to one another.



```
  </tr>
</head>
<!------------------------------------------------------------->
<!--  Scripting                                             -->
<!------------------------------------------------------------->
<% data: wa_con_dat type BAPISCODAT.

   loop at it_con_dat into wa_con_dat. %>
<tr>
   <td> <a href="Details.htm?connid=<%=wa_con_dat-flightconn %>&fldate=<%=wa_co
   <td> <%=  wa_con_dat-flightdate %> </td>
   <td> <%=  wa_con_dat-airportfr %> </td>
   <td> <%=  wa_con_dat-cityfrom %> </td>
   <td> <%=  wa_con_dat-deptime %> </td>
   <td> <%=  wa_con_dat-airportto %> </td>
   <td> <%=  wa_con_dat-cityto %> </td>
   <td> <%=  wa_con_dat-arrtime %> </td>
 </tr>
   <%   endloop.  %>

 </table>

<!------------------------------------------------------------->
                      </form>
<!------------------------------------------------------------->

 </body>

</html>
```

Figure 91: The Web Application Builder

The **Tag Browser** enables you to copy HTML tags, BSP directives, WML tags, or elements from BSP extensions by dragging them to the layout implementation. By double-clicking the respective BSP element, you can branch to the documentation. Using the Tag Browser requires that the user knows the exact meaning of the individual tags and their attributes because there is no access to documentation for HTML tags, WML tags, or BSP directives. The Tag Browser is integrated in the Object Navigator.

Using External HTML Editors

To implement the HTML source code, you can also work with **local external HTML editors** (for example, MS FrontPage™, Macromedia Dreamweaver™ or Adobe GoLive™). For this purpose, you must execute two steps in the Web Application Builder:

1. Choose the menu path *Utilities* → *Setting* and enter the path for the external application in the tab page *Business Server Pages*.
2. Start the external application under the menu path *Edit* → *Start Local HTML Editor*.

As soon as you close the external application, the system asks whether the page is to be reloaded. If you confirm this, the system copies the locally processed HTML source text into the page layout.

You can implement a local HTML editor - that is, one that is installed on your front end - when you are creating the page layout. However, this means that you can always process and save only one page at a time. First, you have to log on to the SAP Web Application Server through the SAP GUI and start the SAP Web Application Builder before the BSP can be processed in the external editor.

In practice, it will generally be the case that specialized Web designers design the layout. These people generally have no access to the SAP Web Application Server, nor should they receive such access.

Using the WebDAV (Distributed Authoring and Versioning) protocol, which is an enhancement of the HTTP protocol, you can access the layout of the BSPs stored in the database of the SAP Web Application Server directly from within a WebDAV client. The SAP Web Application Server in this case plays the role of the WebDAV server. For this purpose, it is equipped with an appropriate WebDAV service (*BSP_DEV*). This is a special HTTP request handler that implements a remote connection through the WebDAV protocol. Here the WebDAV service is implemented through a global ABAP class (*CL_O2_HTTP_WEBDAV*).

Hint: The delivered services can be found in transaction SICF. Using this transaction, you can also create new services for yourself.

The following scenarios are possible:

- The WebDAV client is used as an external editor.
- You create the layout with the help of a WebDAV client in the form of HTML pages.

In the first scenario, the layout of BSPs that have already been created using the Web Application Builder can be exported with the help of the WebDAV client, edited in an easy manner, and then saved again to the database of the SAP Web Application Server.

In the second scenario, the HTML pages are then stored by means of a mass import (together with graphics, CSS, and so on) in the database of the SAP Web Application Server. At first, all the imported objects are interpreted as MIME objects and therefore managed in the MIME Repository. However, the HTML pages can be easily converted into BSPs and afterwards processed in the Web Application Builder.

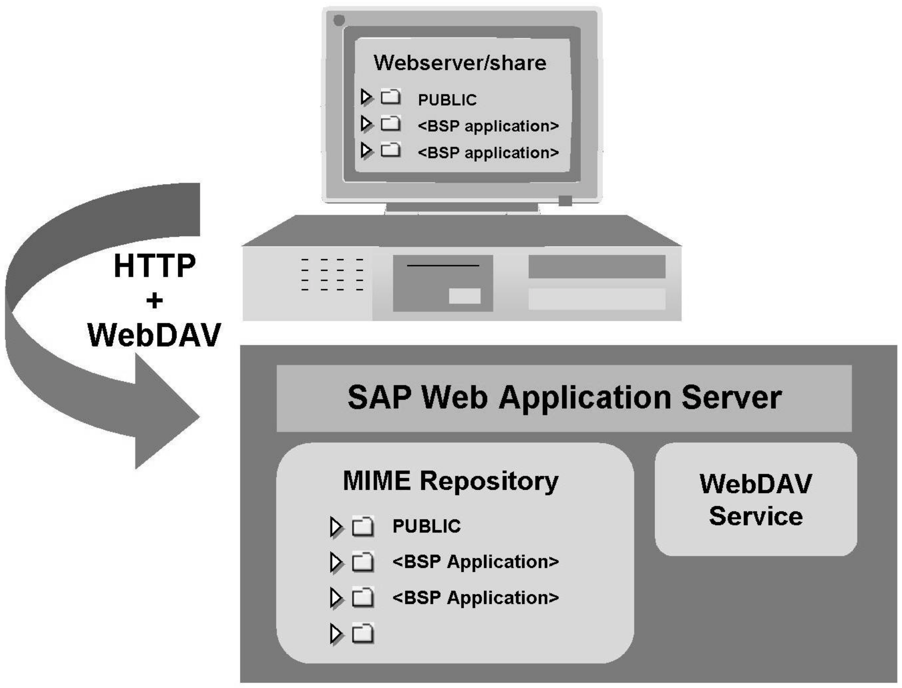


Figure 92: Implementing the WebDAV Service

The WebDAV server is accessed by the WebDAV client in the Windows environment through a so-called Web folder. This first needs to be defined.

1. Creating a Web folder from the Windows Explorer (using *Web Folders* or *My Network Places*).
2. Here you must use the URL of the service in question:

   ```
   http://<server-URL>:<port>/sap/bc/bsp_dev
   ```

   Hint: Make sure that the WebDAV service is active and the client for which the logon is to take place is stored in the service tree.

Lesson Summary

You should now be able to:

- Implement utilities, such as external tools, to efficiently develop BSP applications

Lesson: Other Topics

Lesson Overview

This lesson provides an overview of special topics such as Mobile Business, SAP Smart Forms integration in BSP applications, sending mails from BSP applications, and the use of the SAP Web Application Server as an HTTP client.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe which steps are required to send mails from BSP applications, integrate SAP Smart Form documents into BSP applications, and create Web application for mobile devices.

Business Example

You want to see which other possibilities the BSP programming model offers.

Mobile Business

It is generally predicted that there will be increasing importance attached to the use of mobile terminals, such as mobile phones, laptops, or handheld devices. The current GSM standard (Global Standard for Mobile Communication) will be replaced by UMTS (Universal Mobile Telecommunications System) and will enable a much greater data transfer rate (up to 2 MBits/s). In this way, volume-intensive applications will be possible.

Mobile terminals are already being used for SAP applications, such as mySAP CRM and mySAP PLM.

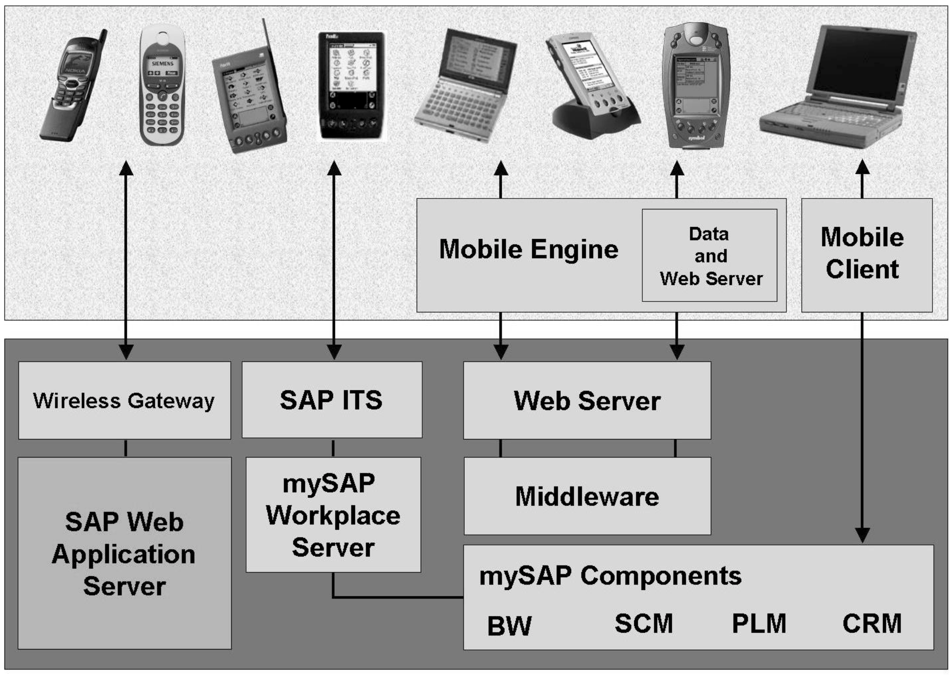


Figure 93: Overview: Architecture for Mobile Terminals

To write business applications for mobile terminals, you must take factors such as display size and color assignment into consideration. Likewise, browser functions that are used on these terminals are standardized. Therefore, the server used must be able to recognize which client has sent the request and it must be possible to format the layout accordingly.

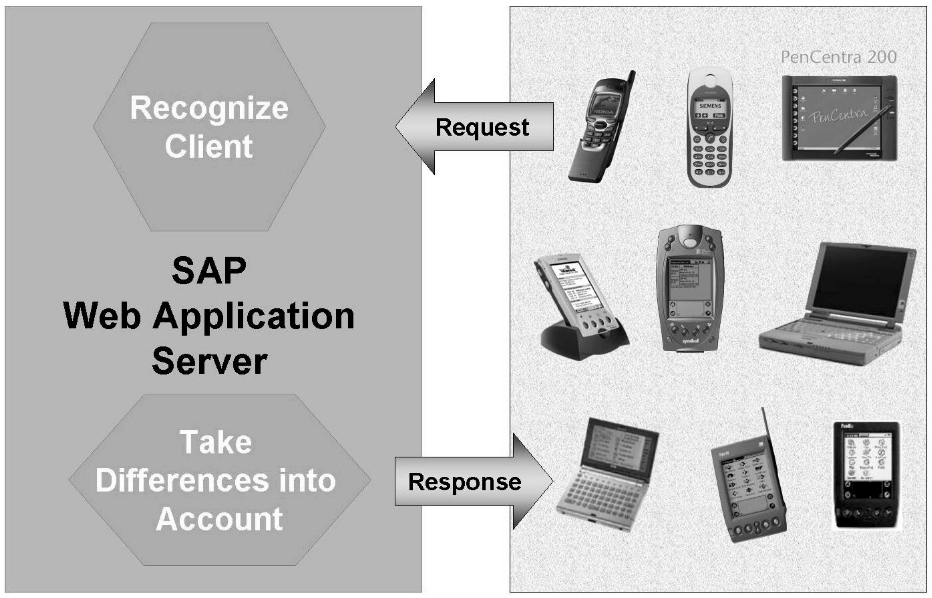
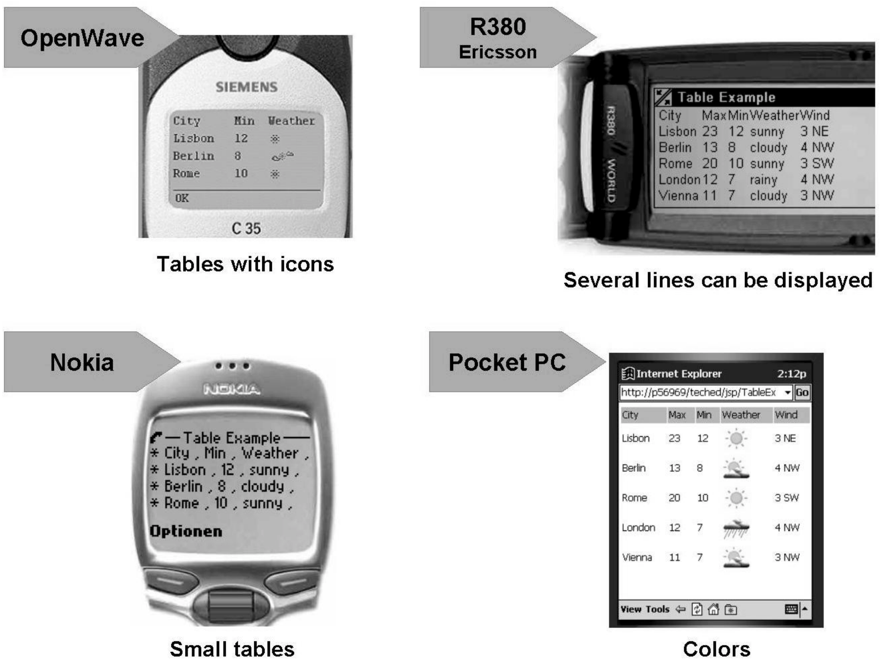


Figure 94: Recognizing the Client

You need to receive information via the client - for example, which language can be used (HTML, WML), what the size and form of the display is, whether special micro-browser properties should be considered. All this information must be available if applications are to function properly on mobile terminals (and be user-friendly).



City	Min	Weather
Lisbon	12	☀
Berlin	8	⛅
Rome	10	☀

OK

Table Example

City	Max	Min	Weather	Wind
Lisbon	23	12	sunny	3 NE
Berlin	13	8	cloudy	4 NW
Rome	20	10	sunny	3 SW
London	12	7	rainy	4 NW
Vienna	11	7	cloudy	3 NW

—Table Example—
* City , Min , Weather ,
* Lisbon , 12 , sunny ,
* Berlin , 8 , cloudy ,
* Rome , 10 , sunny ,

Optionen

Internet Explorer 2:12p
http://p56969/teched/jsp/TableEx Go

City	Max	Min	Weather	Wind
Lisbon	23	12	☀	3 NE
Berlin	13	8	⛅	4 NW
Rome	20	10	☀	3 SW
London	12	7	🌧	4 NW
Vienna	11	7	⛅	3 NW

View Tools

Figure 95: Examples of Table Display

Through the IF_CLIENT_INFO interface, the Internet Communication Framework of the SAP Web Application Server provides over 100 methods that can be used to query all the necessary information on the client at runtime.

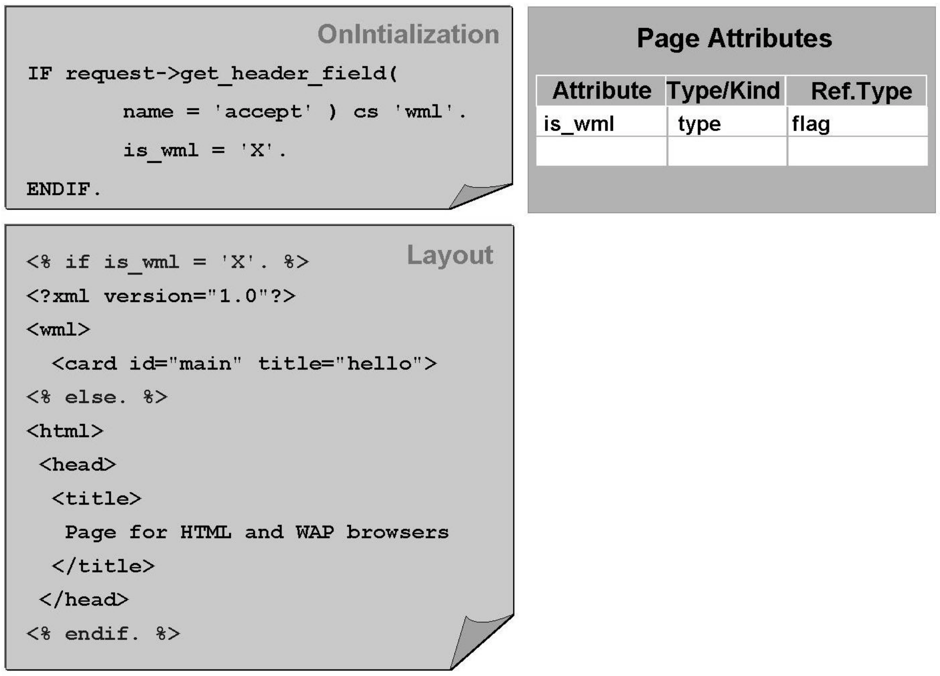
- GET_BROWSER_CATEGORY
- GET_BROWSER_NAME
- GET_BROWSER_OS
- GET_CSS_SUPPORTED
- GET_FORM_FACTOR
- GET_JAVA_SUPPORTED
- GET_PAGE_SIZE_MAX
- GET_SOUND_SUPPORTED
- GET_TITLE_SUPPORTED

To use the interface methods, access is done through the global object RUNTIME.

```
data: client_info type ref to if_client_info,
      b_name      type string.
client_info = runtime->client_info.
client_info->get_browser_name( value = b_name ).
```

All information on a client is kept in the form of an internal table as a private attribute of the RUNTIME object. This table is empty at first. When a method of the interface IF_CLIENT_INFO is called for the first time, the system performs a device recognition query. According to the device determined, this internal table will be supplied with device-specific information. The HTTP header field USER-AGENT is analyzed for device recognition. The value of this field contains the client ID. Possible clients are maintained in the database table MOB_DEVCFG and are identified through this table. The properties of each of these clients are stored in the database table MOB_DEVCAP and are copied from here into the internal table of the RUNTIME object. In this way, new devices can be easily identified through new entries in this table. On the other hand, this means that the database table must first be filled with contents for all clients to be supported. You can maintain over 100 attributes (screen height, screen width, color depth, and so on) for each device type.

In the layout of a Business Server Page, you can use the markup language WML for programming. When you create a WML-based page, choose the appropriate WML suffix (for example, start.wml). If the system determines only at runtime whether a WAP mobile phone or a regular Web browser is serving as a client, do not enter a prefix for the Business Server Page. Instead, find out - using the GET_HEADER_FIELD(NAME = 'ACCEPT') method - which language is to be used in the layout.



```
IF request->get_header_field(
       name = 'accept' ) cs 'wml'.
       is_wml = 'X'.
ENDIF.
```

Page Attributes

Attribute	Type/Kind	Ref.Type
is_wml	type	flag

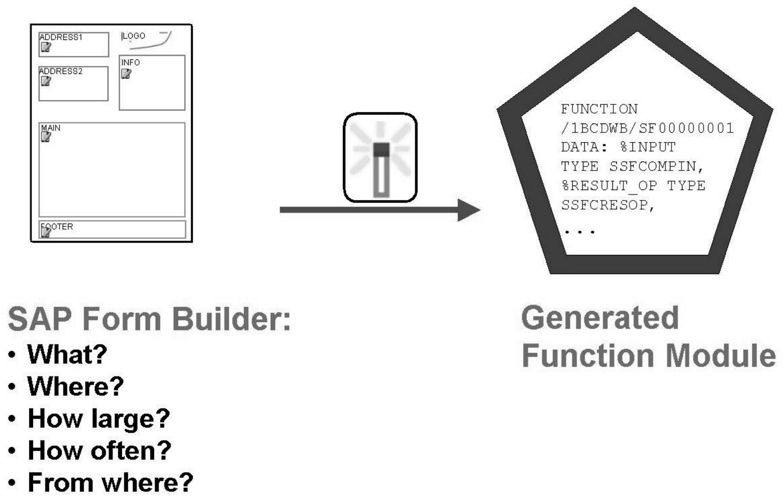
```
<% if is_wml = 'X'. %>
<?xml version="1.0"?>
<wml>
  <card id="main" title="hello">
<% else. %>
<html>
 <head>
  <title>
   Page for HTML and WAP browsers
  </title>
 </head>
<% endif. %>
```

Figure 96: Code Example

For further information on the subject “Using Mobile Devices as Client”, refer to the following links: www.sap.com/mobile and service.sap.com/mobile.

SAP Smart Forms and SAP Web Forms

With **SAP Smart Forms**, you have a tool available as of SAP R/3 Release 4.6C. With this tool you can quickly create and design forms for printing your business documents - for example, invoices or dunning forms. You can print these using standard SAP programs. Using SAP Smart Forms , you can also create your own forms and programs independently of the SAP standard templates.



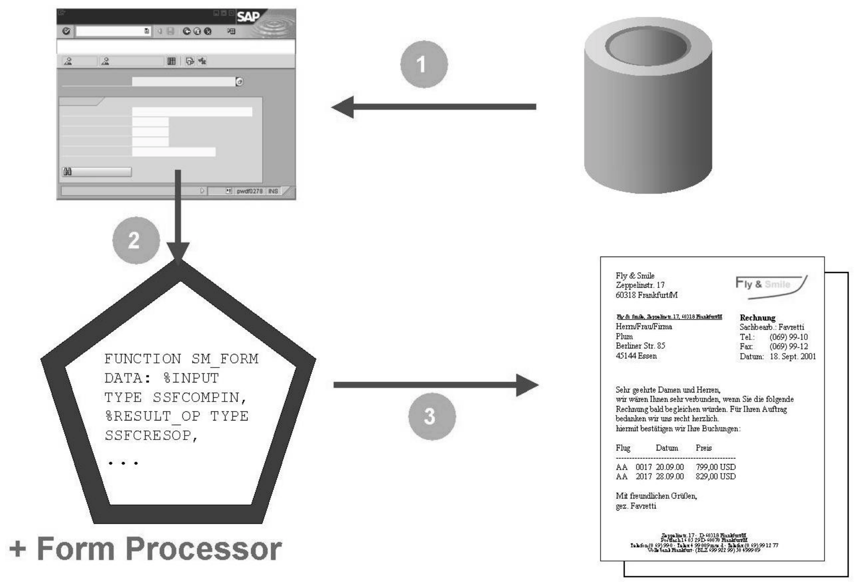
```
FUNCTION
/1BCDWB/SF00000001
DATA: %INPUT
TYPE SSFCOMPIN,
%RESULT_OP TYPE
SSFCRESOP,
...
```

Figure 97: The Maintenance Transaction SMARTFORMS

Using the transaction SMARTFORMS you call the SAP Form Builder for form maintenance. Here you define the following information:

- The form layout - for example, the positions of the text fields or graphics
- The interface - that is, which data and which type of data is to be printed or returned during form processing
- The processing logic - for example, conditions for processing certain form elements

Then you must activate the form. The system automatically generates a function module here.



```
FUNCTION SM_FORM
DATA: %INPUT
TYPE SSFCOMPIN,
%RESULT_OP TYPE
SSFCRESOP,
...
```

Fly & Smile
Zeppelinstr. 17
60318 Frankfurt/M

Fly & Smile, Zeppelinstr. 17, 60318 Frankfurt/M
Herrn/Frau/Firma
Plum
Berliner Str. 85
45144 Essen

Rechnung
Sachbearb.: Favretti
Tel.: (069) 99-10
Fax: (069) 99-12
Datum: 18. Sept. 2001

Sehr geehrte Damen und Herren,
wir wären Ihnen sehr verbunden, wenn Sie die folgende Rechnung bald begleichen würden. Für Ihren Auftrag bedanken wir uns recht herzlich.
hiermit bestätigen wir Ihre Buchungen:

```
Flug        Datum      Preis
------------------------------------
AA   0017  20.09.00   799,00 USD
AA   2017  28.09.00   829,00 USD
```

Mit freundlichen Grüßen,
gez. Favretti

Zeppelinstr. 17 · D-60318 Frankfurt/M
Postfach 1 6 03 29 D-60070 Frankfurt/M
Telefon (0 69) 99-0 · Telex 4 99 009 mus d · Telefax (0 69) 99 12 77
Volksbank Frankfurt · (BLZ 499 922 99) 50 4999 09

Figure 98: Principle of Form Printing Using SAP Smart Forms

The process for calling an SAP Smart Form from within an application is as follows:

1. The application reads all the relevant business data (for example, from database tables).
2. Then it calls the generated form function module and passes the data to it through the latter's interface.
3. A form processor, which is automatically called together with the function module, takes care of the actual processing.

With the SAP Web Application Server, it is possible to include SAP Smart Forms (or to be more precise, the function modules generated from them) in Business Server Pages and to display them in a standard Web browser. There the user can, if required, make his or her entries and return the **SAP Web Form** to the SAP Web Application Server for further processing by pressing the appropriate button. It is not necessary to create new forms for Business Server Pages because you can use the same forms as you use for printing or faxing. Additional elements like submit buttons and checkboxes can be integrated, if necessary. When printing the form, these elements are automatically suppressed.

So that SAP Smart Forms can be integrated into the Business Server Pages, their respective function modules should not return, as standard, the SAP spool format (OTF = Output Text Format), but HTML format. The output is made up of the following components: pure data in XSF format (XML for SAP Smart Forms), information on formatting as CSS (Cascading Style Sheet), and the data in HTML format. The output is transformed into

HTML format on the server side by an XSLT program (Extensible Style Language Transformation). The browser does not need any extensions for display purposes.

There are two ways of outputting forms in HTML: statically or dynamically. In the output options for the form, you can set the output format to "XSF Output + HTML" in the SAP Form Builder or set the respective switch in the exporting parameter *output_options* when you call the generated function module.

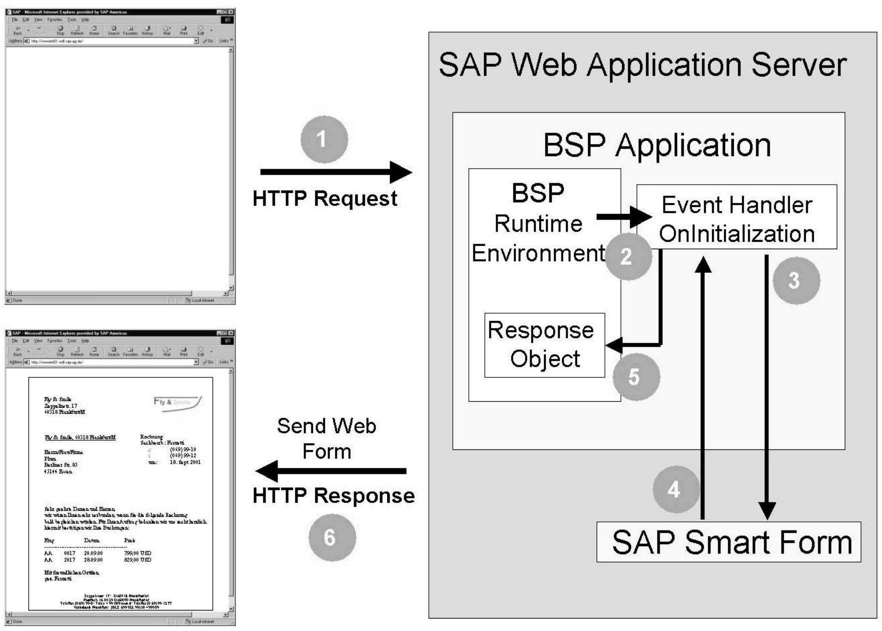


Figure 99: Web Form Principle

You typically call forms in the event handler **OnInitialization** in a Business Server Page. For this purpose, the importing parameter *job_output_info* must have a variable of the type *ssfcrescl* , which, after being called, contains the data and information to be processed. You then evaluate this variable (after conversion) using the standard methods response->set_header_field and response->set_data so that an HTTP response can be created.

Evaluation of Form Input:

You can integrate the following input-ready fields into a SAP Smart Form: checkboxes, radio button groups, submit buttons, list boxes, as well as text fields. Also, you can mark elements as hidden. To make fields ready for input, call up the SAP Form Builder and, on the tab page for Web properties for text elements, mark the required fields with the respective attributes. All the input-ready fields in a form are displayed in a single HTML form. So that this form can be sent, for example, to a Business Server Page, there must be at least one submit pushbutton on the form, and the field *xsfaction* in the function module exporting parameter *output_options* must contain the target URL. The next Business Server Page could then evaluate the inputs using the method *request->get_form_fields*.

For more information, use the online help for SAP Smart Forms (component abbreviation: BC-SRV-SSF), Unit "Web Forms for Internet Applications".

Sending Mails from BSP Applications

From BSP applications, you can also send mails using global classes and methods that are provided by the Business Communication Service (BCS). Here, the following requirements must be fulfilled by the SAP Web Application Server:

- If mails are to be sent using the SMTP protocol, the SMTP plug-in must be available and configured in the profile.

  In this case, the SMTP node must be configured and activated in transaction SCOT.
- If mails are to be sent using the SAP Internet Mail Gateway, the IMGW node must be configured and activated in the transaction SCOT.
- You must have a mail domain assigned to your system.
- In SU01, a mail address of the type INT (communication type e-mail) must be assigned to the sender (system user).

Sending a Mail from a BSP Application

1. You must create an object representing the mail request.

   This requires the global class *CL_BCS* and the static method *create_persistent*.

```
  DATA: send_request TYPE REF TO cl_bcs.
* create persistent send request
  send_request = cl_bcs=>create_persistent( ).
```

2. You must create a mail document.

   This requires the global class *CL_DOCUMENT_BCS* and the static method *create_document*. The mail text is managed in an internal table.

```
DATA: document TYPE REF TO cl_document_bcs.
DATA: mail_itab TYPE soli_tab,
      textlength TYPE so_obj_len.
```

   The internal table consists of a column of the type *character* with length 255. Fill this table with the text you want to send. For example, this can be a standard text that is stored in the database or you can set it up individually using text elements.

```
document = cl_document_bcs=>create_document(
            i_type    = 'RAW'
            i_text    = mail_itab
            i_length  = textlength
            i_subject = 'NET200 - test mail' ).
```

   Now you assign the document you have created to the mail request.

```
* add document to send request
send_request->set_document( document ).
```

3. You must define the sender name.

   This requires the global class *CL_SAPUSER_BCS* and the static method *create*.

```
DATA: sender TYPE REF TO cl_sapuser_bcs.
```

Continued on next page

```
* get sender object
  sender = cl_sapuser_bcs=>create( sy-uname ).
```

Now you assign the sender you have created to the mail request.

```
* add sender
  send_request->set_sender( sender ).
```

4. You must define the receiver name.

   For this you require the global class *CL_CAM_ADDRESS_BCS*, the interface *IF_RECIPIENT_BCS*, and the static method *create_internet_address*.

```
  CLASS: cl_cam_address_bcs DEFINITION LOAD.

  DATA: address TYPE REF TO if_recipient_bcs.
  DATA: c_address  TYPE adr6-smtp_addr.
```

   The recipient is created and passed to the mail request.

```
* create recipient
  MOVE recipient TO c_address.
  address =
  cl_cam_address_bcs=>create_internet_address( c_address ).

* add recipient to send request
  send_request->add_recipient(
     EXPORTING i_recipient  = address
               i_express    = ' '
               i_copy       = ' '
               i_blind_copy = ' ' ).
```

5. In the last step, you release the mail request. The mail is sent asynchronously.

```
* send document
  send_request->send( ).
  COMMIT WORK.
```

Using the SAP Web Application Server as a Client

The SAP Web Application Server can also function as a client: It can send a request to a HTTP server and receive a response. With the help of the methods of the **IF_HTTP_CLIENT** interface, the request is set up in an ABAP program, sent, and the response is received and evaluated. Another SAP Web Application Server or an arbitrary HTTP server can function as an HTTP server.

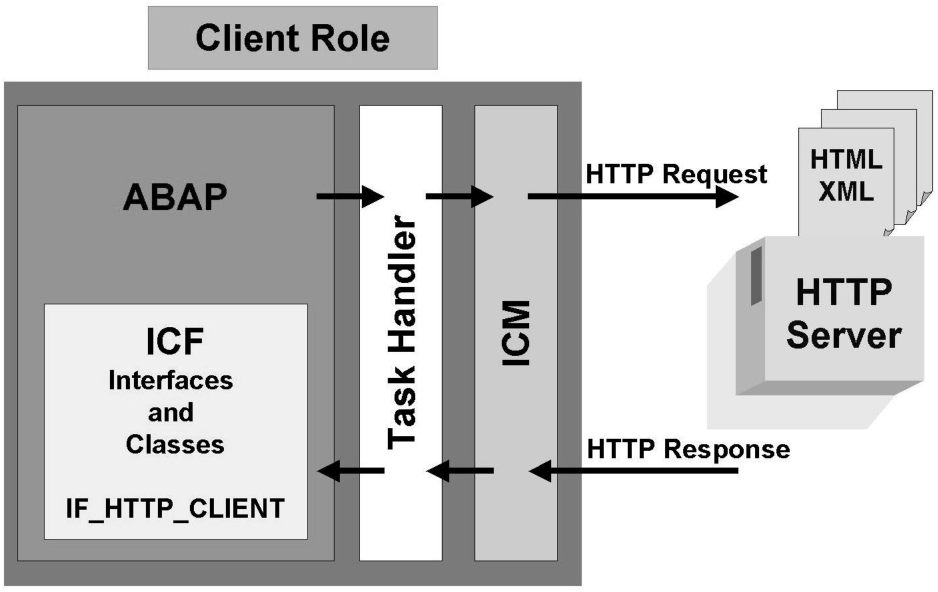


Figure 100: Client Role of the SAP Web Application Server

The client role and server role of the SAP Web AS can also be combined. You can thus request an object of an HTTP server from within a BSP application using HTTP. This object can then be processed or added directly to the HTTP response of the SAP Web AS. As a possible scenario, you could, within a BSP application, request data from a content server in form of an XML document; in the BSP application, you could then transfer this data to ABAP attributes using an XSLT transformation and then display it in the layout of the BSP.

You must depict all the necessary steps in the ABAP program. So, too, the evaluation of the response. For this purpose, suitable methods are available (for example, GET_CDATA in order to get the page content as a string).

For details on this topic, refer to the documentation.

Lesson Summary

You should now be able to:

- Describe which steps are required to send mails from BSP applications, integrate SAP Smart Form documents into BSP applications, and create Web application for mobile devices.

Related Information

- For more information on SAP Smart Forms and SAP Web Forms, refer to the online documentation. (Component abbreviation: BC-SRV-SSF), Unit "Web Forms for Internet Applications".
- For further information on the topic of using mobile devices as a client, refer to www.sap.com/mobile and service.sap.com/mobile.
- For more information on the topic of using the SAP Web Application Server as a client, refer to the online documentation (component BC-MAS). Test programs for this topic are in the package SHTTP.

Unit Summary

You should now be able to:

- Describe different logon procedures
- Define services in transaction SICF and there set the logon procedure to be used
- Create anonymous users for a service
- Set up applications with public and protected areas
- Create Internet users dynamically
- Give an overview of RFC and BAPIs
- Give an overview of the important tools in the RFC/BAPI environment
- Call a BAPI in an SAP R/3 back-end system
- Implement utilities, such as external tools, to efficiently develop BSP applications
- Describe which steps are required to send mails from BSP applications, integrate SAP Smart Form documents into BSP applications, and create Web application for mobile devices.

Course Summary

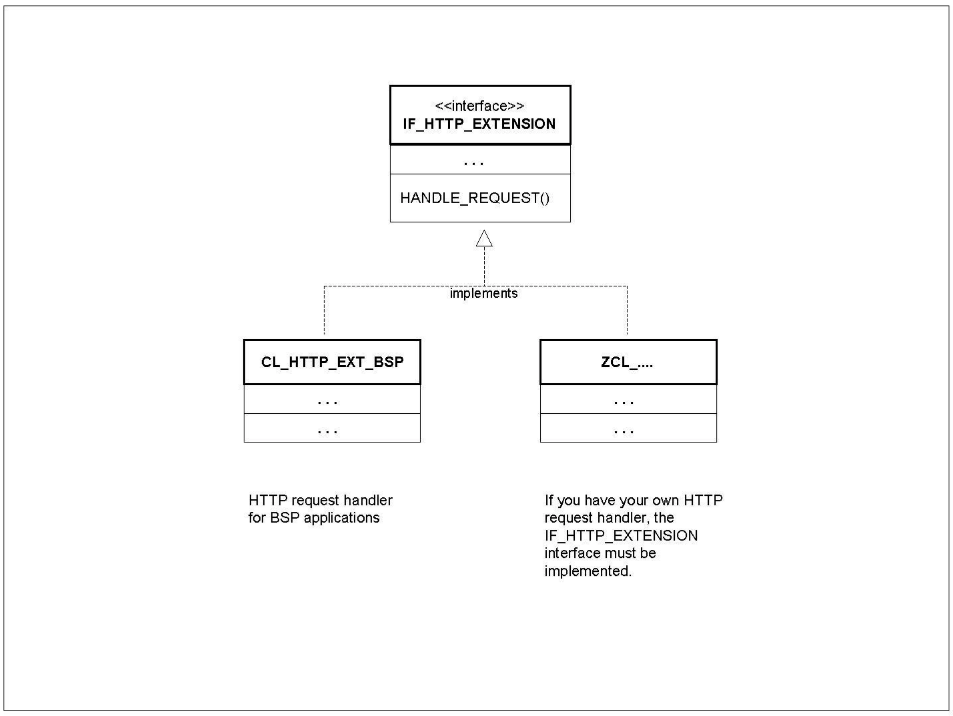
You should now be able to:

- Describe the system architecture of the SAP Web Application Server
- Describe the request/response cycle
- Name the components of a Business Server Page and a BSP application and describe their use
- Develop Web applications based on Business Server Pages
- Implement the layout of Business Server Pages using HTMLB elements
- Implement language-specific BSP applications
- Explain how to assign a desired corporate identity design without modification by assigning a topic
- Use data from other SAP systems by calling BAPIs in your BSP applications

Appendix 1

UML Diagrams

SICF: Request Handler



<<interface>>
IF_HTTP_EXTENSION
...
HANDLE_REQUEST()
implements
CL_HTTP_EXT_BSP
...
...
ZCL_....
...
...
HTTP request handler for BSP applications
If you have your own HTTP request handler, the IF_HTTP_EXTENSION interface must be implemented.

Figure 101: BSP Request Handler and User-Defined Request Handler

The Internet Communication Framework (ICF)

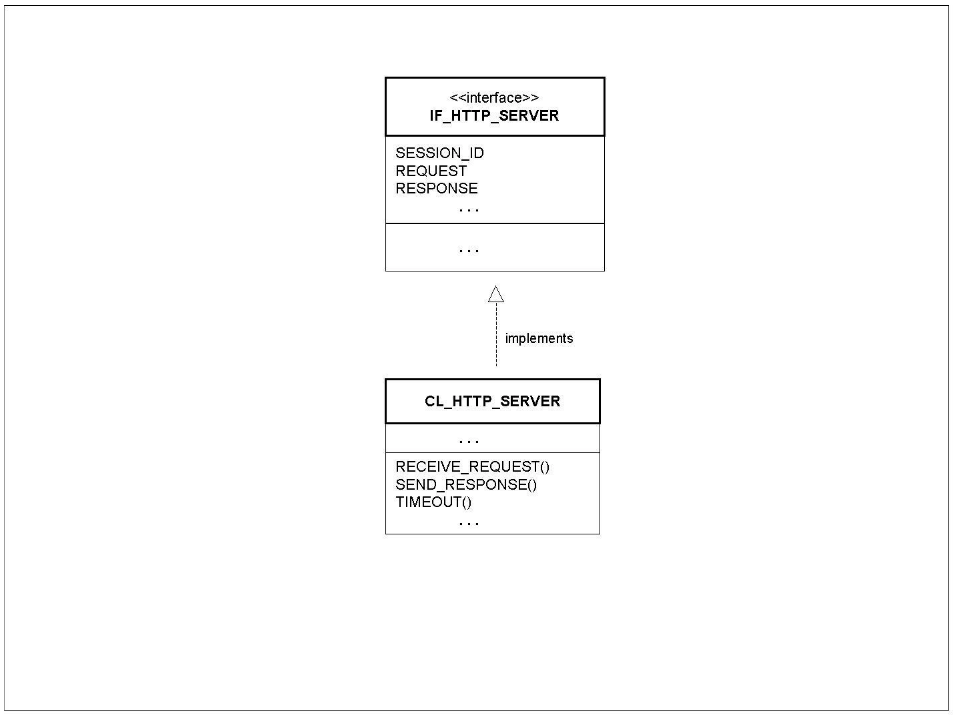


Figure 102: The ICF Manager (CL_HTTP_SERVER)

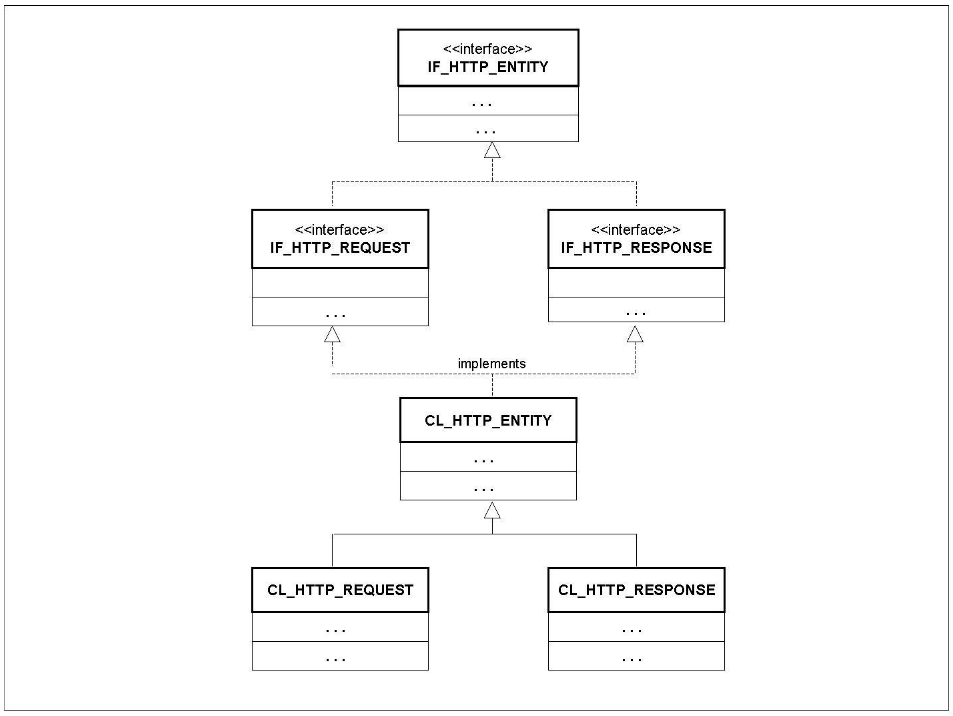


Figure 103: Classes CL_HTTP_REQUEST and CL_HTTP_RESPONSE

Classes and Interfaces in the BSP Programming Model (Examples)

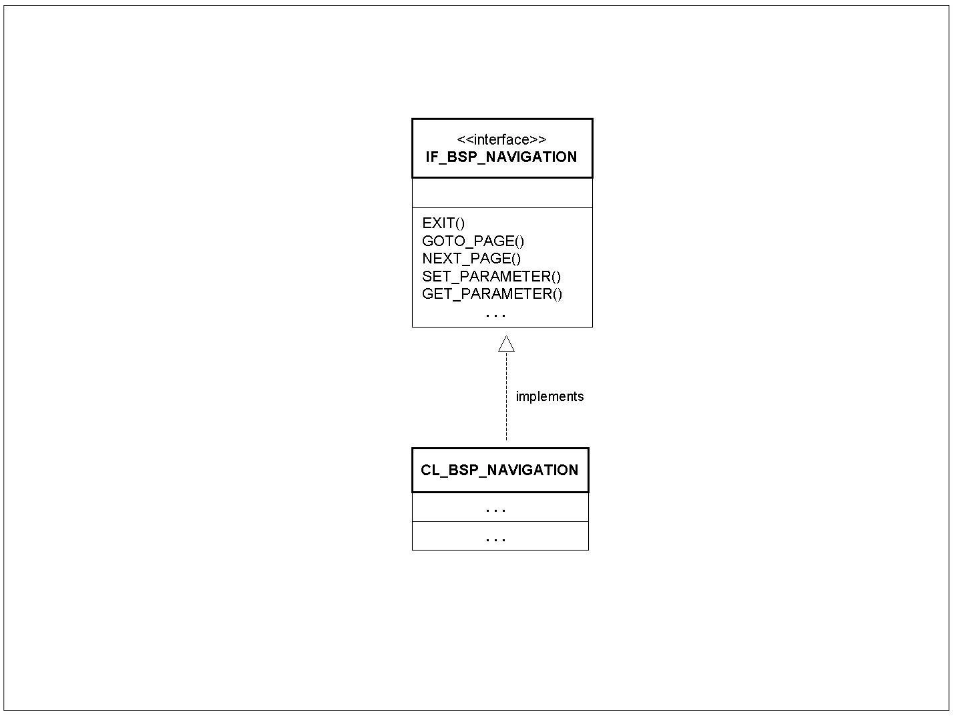


Figure 104: Class CL_BSP_NAVIGATION

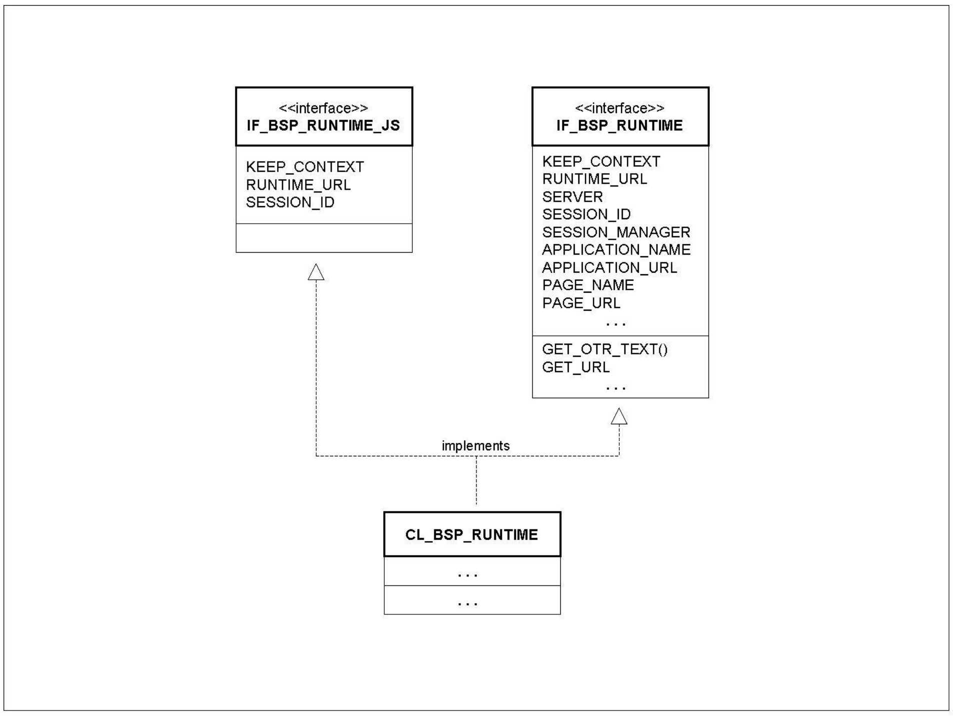


Figure 105: Class CL_BSP_RUNTIME

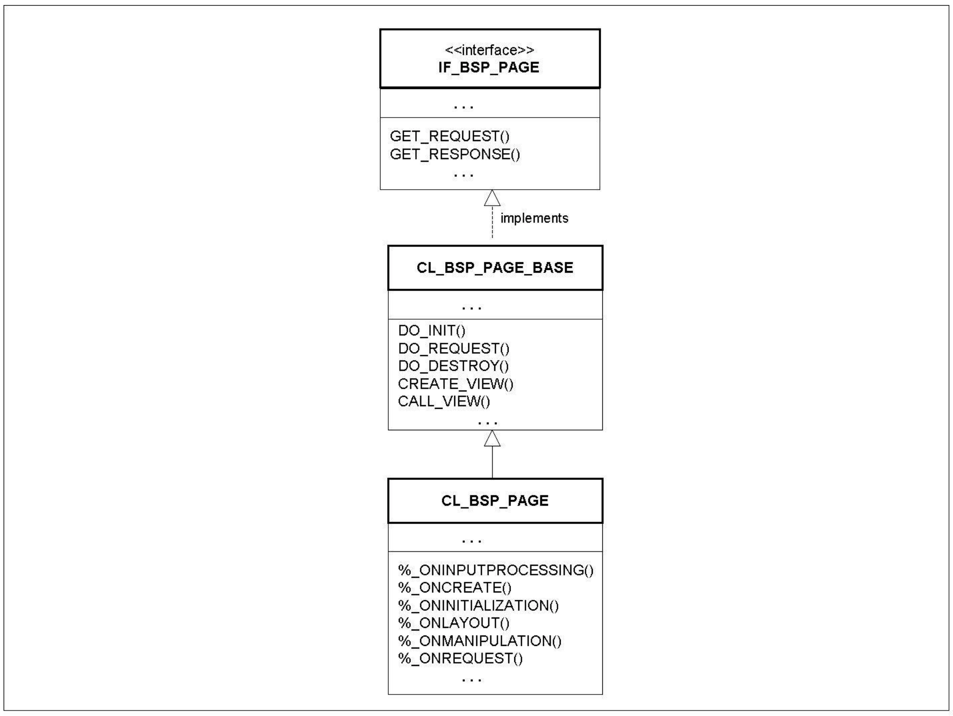


Figure 106: Class CL_BSP_PAGE

The BSP Extension Framework

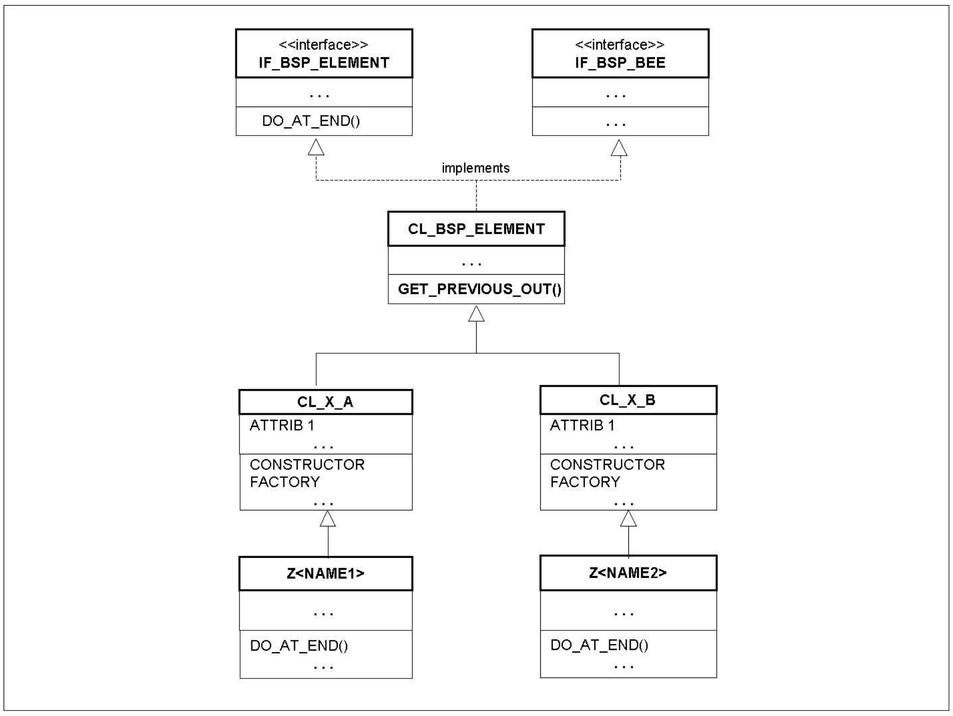


Figure 107: Using the General Basis Class CL_BSP_ELEMENT to Define Elements A and B of a BSP Extension X

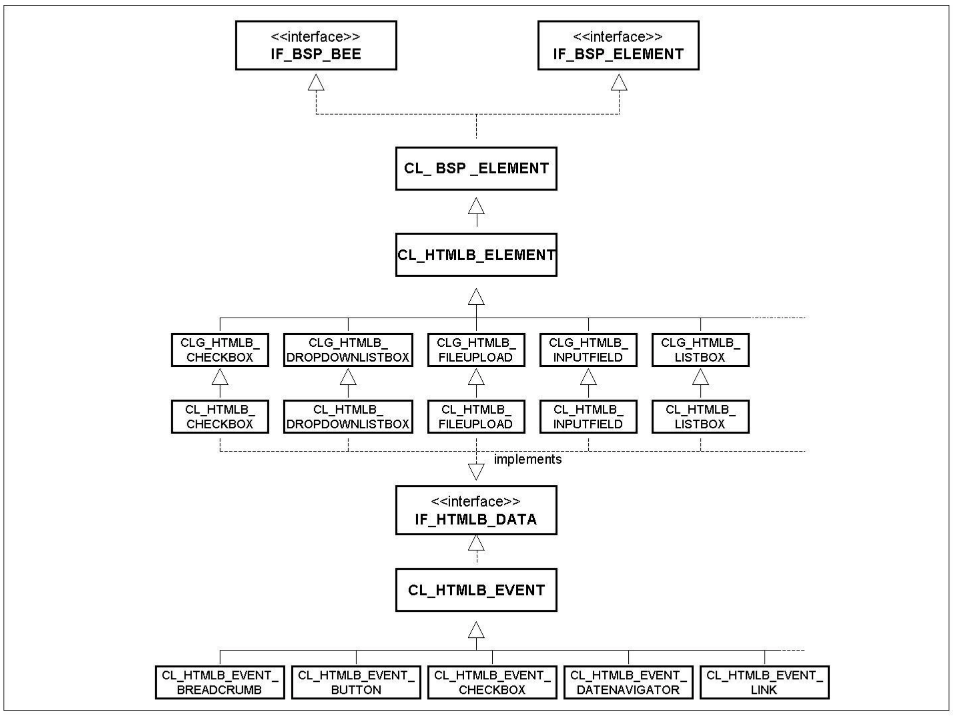


Figure 108: HTMLB: Using the Special Basis Class CL_HTMLB_ELEMENT to Define Elements

MVC Design Pattern

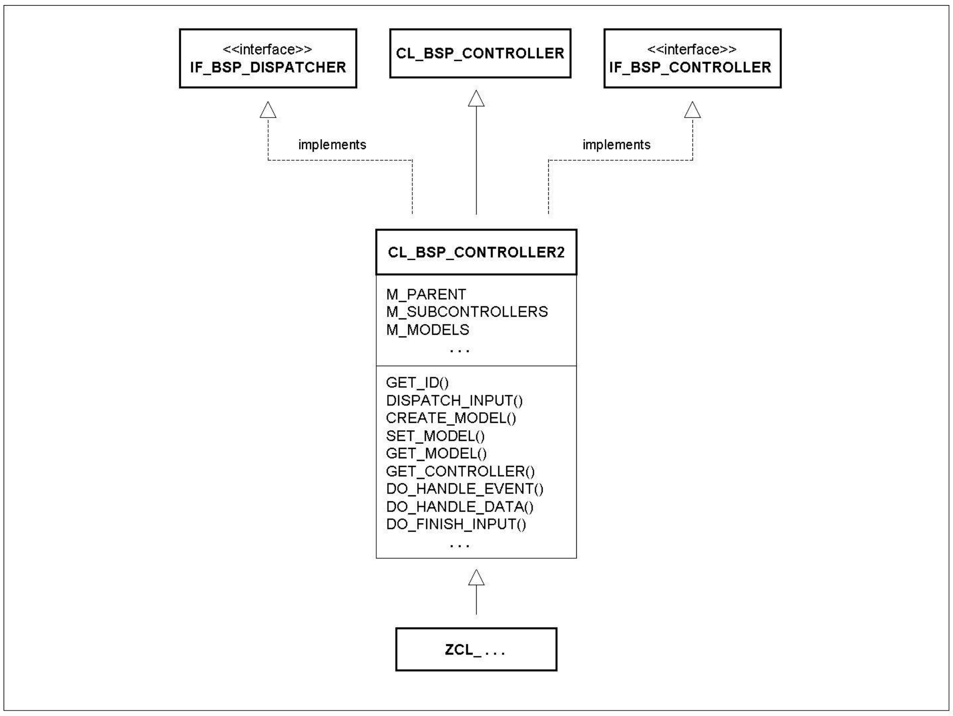
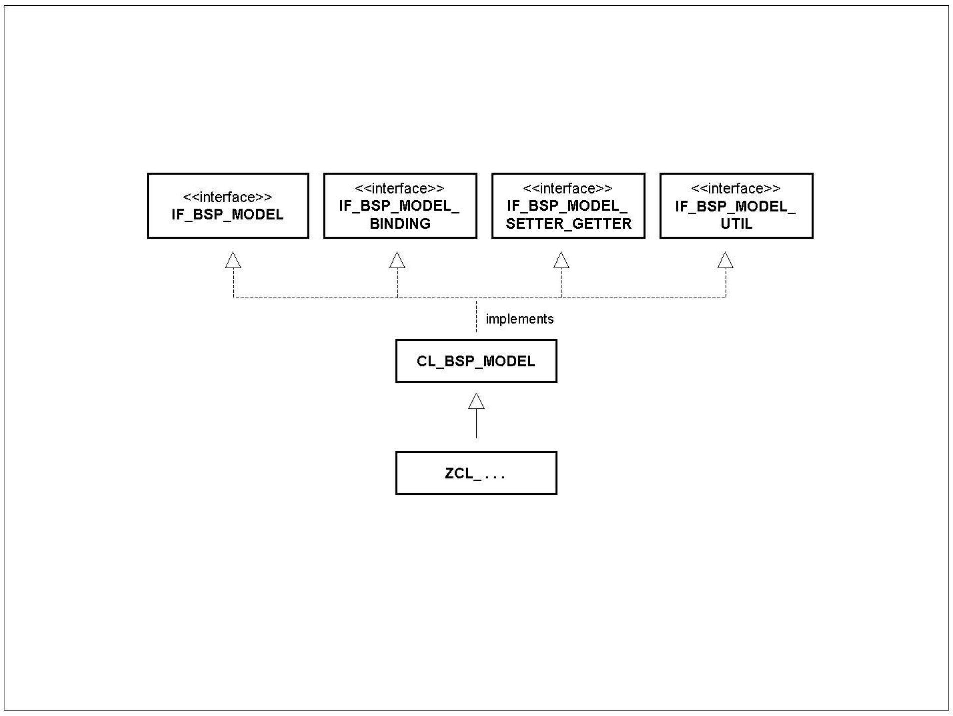


Figure 109: Controller Class



<<interface>> IF_BSP_MODEL
<<interface>> IF_BSP_MODEL_ BINDING
<<interface>> IF_BSP_MODEL_ SETTER_GETTER
<<interface>> IF_BSP_MODEL_ UTIL
implements
CL_BSP_MODEL
ZCL_ . . .

Figure 110: Model Class

Index

Feedback

SAP AG has made every effort in the preparation of this course to ensure the accuracy and completeness of the materials. If you have any corrections or suggestions for improvement, please record them in the appropriate place in the course evaluation.