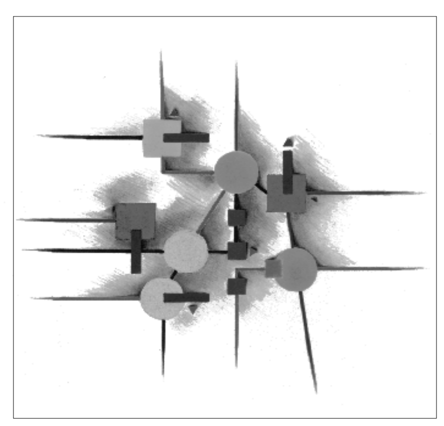
SAP Business Connector
Building Output Templates And DSPs



SAP System

Release 4.8

SAP AG - Dietmar-Hopp-Allee 16 - D69190 Walldorf

Copyright

©Copyright 2007 SAP AG.. All rights reserved.

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

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

Microsoft® , WINDOWS® and EXCEL®, NT® and SQL-Server® are registered trademarks of Microsoft Corporation.

IBM®, OS/2®, DB2/6000®, AIX®, OS/400® and AS/400® are registered trademarks of IBM Corporation.

OSF/Motif® is a registered trademark of Open Software Foundation.

ORACLE®, is a registered trademark of ORACLE Corporation, California, USA.

webMethods® is a registered trademark of webMethods Incorporated, Virginia, USA.

INFORMIX®-OnLine for SAP is a registered trademark of Informix Software Incorporated.

UNIX ® and X/Open® are registered trademarks of SCO Santa Cruz Operation.

SAP®, R/2®, R/3®, RIVA®, ABAP/4®, SAPaccess®, SAPmai®l, SAPoffice®, SAP-EDI®, SAP Business Workflow®, SAP Early Watch®, SAP Archive Link®, R/3 Retail®, ALE/WEB®, SAPTRONIC® are registered trademarks of SAP AG.

All rights reserved.

Contents

Chapter 1. **Introduction** 5

Welcome to *Building Output Templates and DSPs* *6*

Typographical Conventions 6

Program Code Conventions 7

Related Documentation 8

Viewing this Document 9

Printing this Guide 9

Chapter 2. **Creating Output Templates** 11

What Is an Output Template? 12

What Does an Output Template Look Like? 12

How Do You Assign an Output Template to a Service? 13

When Does the Server Use an Output Template? 15

Using Output Templates to Return Output to HTTP Clients 15

Guidelines for Using HTML-based Output Templates with HTTP Clients 15

Guidelines for Using XML-based Output Templates with HTTP Clients 15

Using Output Templates to Return Output to SMTP and FTP Clients 16

Using Output Templates to Return Output to Wireless Devices 16

Guidelines for Creating WML Output Templates 16

Guidelines for Creating HDML Output Templates 17

Applying Output Templates Arbitrarily 17

Creating an Output Template 17

Chapter 3. **Using Dynamic Server Pages (DSPs)** 21

What Is a Dynamic Server Page? 22

What Does a DSP Look Like? 22

When Do You Use DSPs? 23

What Are the Advantages of Using DSPs? 23

Creating DSPs 23

Publishing DSPs 24

Securing a DSP 24

Requesting a DSP 25

Hyperlinks to DSPs 26

Using the DSP Tags 26

Begin...End Constructs 27

Invoking Services Using the %invoke% Tag 28

What Is Scope? 29

Why Does Scope Matter? 30
Ending the Scope of the Invoke Action 30
Referencing Variables In and Out of Scope 32
Passing Parameters with a DSP 33
Passing Parameters Between DSPs 35
Passing Parameters Between Services within a DSP 36
Catching Errors 36
Extracting Results from an Array Variable 37
Extracting Results from a Record 38
Using the %loop% Tag to Examine the Current Scope 38
Conditionally Executing Blocks of Code 39
Building Conditional Blocks With the %ifvar% Tag 39
Testing for a Particular Value 40
Building Conditional Blocks with the %switch% Tag 40
Specifying a Default Case 42
Inserting Text Files In a DSP 42
Arbitrarily Processing DSP Tags 43

Appendix A. Tag Descriptions 45
Overview 45
%comment% 46
%ifvar% 47
%include% 49
%invoke% 50
%loop% 52
%loopsep% 55
%nl% 56
%rename% 56
%scope% 58
%switch% 62
%sysvar% 64
%value% 65

Index 69

1

Introduction

- Welcome to Building Output Templates and DSPs 6
- Typographical Conventions 6
- Related Documentation 8
- Viewing this Document 9
- Printing this Guide 9

Welcome to *Building Output Templates and DSPs*

This guide is for those developers who want to build output templates to format the results of services executed on the SAP BC Server and for developers who want to build browser-based clients using Dynamic Server Pages (DSPs).

Typographical Conventions

This document uses the following typographical conventions:

Convention	Example
Procedures are designated by a blue box in the left column. Procedures are presented as a series of numbered steps.	1 On the **Activity** menu, click **File**.
Terms that identify elements, options, selections, and commands on the screen are shown in bold.	The **Service** field on the **Properties** tab specifies the name of the requested service.
Characters that you must type exactly are shown in a typewriter font.	Type: `setup` and then press ENTER.
Variable information that you must type based on your specific situation or environment is shown in italics.	Type: <*sapbc*>`\setup` and then press ENTER.
Keyboard keys are shown in uppercase.	Press ENTER; then press TAB.
Keys that you must press simultaneously are joined with the “+” symbol.	Press CTRL+ALT+M.
Directory paths are shown with the “\” directory delimiter unless the subject is UNIX-specific. In these cases, the “/” is used. If you are working in a UNIX environment, substitute a “/” for the “\” shown in the procedures in this book.	<*sapbc*>\server\packages\ Default

Convention	Example
Information that you must read before beginning a procedure or that alerts you to negative consequences of certain actions is denoted using this notation.	**Important!** If the folder is not already open in the Service Browser, open it before you start the following procedure.
Notes that provide related, but non-critical, information are denoted using this notation.	**Note**: When you start SAP BC Developer, you are prompted to log on to a SAP BC Server.
Helpful information such as shortcuts and alternatives.	**Tip!** You can also use CTRL+C to copy an object.s

Program Code Conventions

For programming code and command syntax, this document uses the following typographical conventions:

Convention	Example
Keywords and values that you must type exactly as printed are shown in typewriter font.	`%CoSymbol%`
Variable values or parameters that you must supply are shown in italics.	`%`*`VarName`*`%`
Keywords or values that are optional are enclosed in [ ]. Do not type the [ ] symbols in your own code.	`%loop` *`LoopVar`* `[null=`*`NullValue`*`]%`

Related Documentation

The following documents are companions to this guide. Some documents are in PDF format and others are in HTML.

Refer to this book...	For...
SAP BC Administration Guide	Information about using the Server Administrator to configure, monitor, and control the SAP BC Server. This book is for server administrators. You will find this book at: <*sapbc*>\server\doc\SAPBCAdministrationGuide.pdf
SAP BC Developer Guide	Information about creating and testing services and client applications. This book for is for solution developers. You will find this book at: <*sapbc*>\developer\doc\SAPBCDeveloperGuide.pdf
SAP BC Built-In Services Guide	Descriptions of services that are installed on your SAP BC Server. This book for is for solution developers. You will find this book at: <*sapbc*>\developer\doc\SAPBCBuiltInServicesGuide.pdf
SAP BC Java API Reference	Descriptions of the Java classes you use to create services. This reference is for developers who build services using Java. You will find this book at: <*sapbc*>\server\doc\api\Java\index.html
SAP BC Administrator's Online Reference	Information about the controls in the SAP BC Server Administrator screens and step-by-step procedures describing how to perform tasks with the Server Administrator. You can access the online reference by clicking **Help** on any of the screens in the Server Administrator.
Developer Online Reference	Information about the controls in the Developer application windows and step-by-step procedures describing how to perform tasks with the Developer. You can access the online reference by clicking **Help** in an application window or dialog box in Developer.

Viewing this Document

To view this document, which is in PDF format, you must have Acrobat Reader™ 4.0 or later installed on your system. If you have an earlier version of Acrobat Reader, you will receive the following error message when you open this document and Acrobat Reader will not display the images in this document:

```
Could not find the ColorSpace named 'Cs8.'
```

If you do not have this software or you do not have the correct version, you can download a free copy from:

http://www.adobe.com/downloads.

Printing this Guide

To produce a hard copy of this guide, print this document from Acrobat Reader.

2

Creating Output Templates

- What Is an Output Template? 12
- What Does an Output Template Look Like? 12
- How Do You Assign an Output Template to a Service? 13
- When Does the Server Use an Output Template? 15
- Creating an Output Template 17

What Is an Output Template?

Output templates allow you to insert output values from a service into a document that you define. They work much like server-side includes in that they contain special “tags” that the SAP BC Server processes before passing the document back to the client.

Output templates are used most frequently to customize the HTML page that a service returns to a browser-based application. However, you can also use them to generate an other types of documents. For example, if you have a service that retrieves a record from a relational database, you might use an output template to format your results as an XML document or a comma-delimited record before returning it to the requestor.

You can also use output templates to return WML or HDML content to wireless devices, such as an Internet-enabled wireless phone or an Internet-enabled personal digital assistant.

What Does an Output Template Look Like?

A template is simply a String containing text and one or more *tags*. Tags are special commands—enclosed in % symbols—that cause the SAP BC Server to perform a specified action—typically to insert the value of a variable—at a specified point in the String.

For example, if you have the following values in the pipeline:

Output in the Pipeline

Variable Name	Value
CompanyName	Bitterroot Boards
AccountNum	BTB-9590651
ContactName	Lauren Cheung
PhoneNum	406-721-5000

You might create a template that looks as follows to return the values in an HTML document to a browser-based client. The strings in **bold** are template tags instructing the server to insert specified values from the pipeline at run time.

Output Template for an HTML Document

```
<!DOCTYPE HTML PUBLIC >
<HTML><HEAD><TITLE></TITLE></HEAD>
<BODY><P>Contact information for account %value AccountNum% is:</P>
   <TABLE>
      <TR><TD><B>Account Name</B></TD><TD>%value CompanyName%</TD></TR>
      <TR><TD><B>Contact Name</B></TD><TD>%value ContactName%</TD></TR>
      <TR><TD><B>Phone Number</B></TD><TD>%value PhoneNum%</TD></TR>
   </TABLE>
   </BODY></HTML>
```

Output templates are not limited to HTML. You could create an output template that looks like this to return values in an XML document:

Output Template for an XML Document

```
<?xml version="1.0"?>
<ACCOUNT_INFO>
   <ACCOUNT_NAME>%value AccountNum%</ACCOUNT_NAME>
   <COMPANY_NAME>%value CompanyName%</COMPANY_NAME>
   <CONTACT_NAME>%value ContactName%</CONTACT_NAME>
   <PHONE_NUM>%value PhoneNum%</PHONE_NUM>
<ACCOUNT_INFO>
```

Or an output template that looks like this to return values as a comma-separated record:

Output Template for an XML Document

```
%value AccountNum%,%value CompanyName%,%value ContactName%,%value PhoneNum%
```

How Do You Assign an Output Template to a Service?

Use of an output template for a service is optional. You assign an output template to a service using the **Settings** tab in Developer.

When using output templates, keep in mind that:

- You do not have to assign an output template to a service.
- A service can have at most one output template assigned to it at a time (you can dynamically change the output template assignment at run time, however).
- You can assign the same output template to more than one service.
- If you assign an existing output template to a service, the output template must reside in the <*sapbc*>\server\packages*packageName*\templates directory where *packageName* is the package in which the service is located.
- You can reference one output template from within another.

To assign an output template to a service

1. Start Developer and connect to the server on which the service resides.
2. In the Service Browser, select the service to which you want to assign an output template.
3. Click the **Settings** tab.
4. In the **Name** field, do one of the following:
   - If you want to assign a *new* output template to the service, type the name of the new output template. By default, Developer assigns the service an output template with the name *FolderName_ServiceName*.
   - If you want to assign an *existing* output template to the service, type the file name of the existing output template. You do *not* need to include the path information or the file name extension.

Important! If you want to assign an existing output template to the service, make sure the output template resides in the <*sapbc*>\server\packages*packageName*\templates directory where *packageName* is the same package in which the service is located.

5. In the **Type** list, do one of the following to specify the format for the output template:

Select...	To...
html	Assign an HTML output template to the service.
xml	Assign an XML output template to the service.
wml	Assing a WML output template to the service.
hdml	Assign an HDML output template to the service.

Note: The **Type** you select determines the extension for the output template file (*.html, *.xml, *.wml, or *.hdml). In cases where the output is returned to a Web browser, the **Type** also determines the value of the HTTP “Content-Type” header field (e.g., “text/html,” “text/xml,” “text/vnd.wap.wml,” or “text/x-hdml“).

6. Do one of the following:
   - If you assigned a new output template to the service, click **New** to create the output template.
   - If you assigned an existing output template to the service and you want to edit the template, click **Edit**.

7 In the **Template Name** dialog box, create or edit the output template by typing in HTML, XML, WML, or HDML content, and/or by inserting template tags. For more information about creating an output template, see "Creating an Output Template" on page 17. For more information about template tags, see Appendix A, "Tag Descriptions" on page 45.

8 Click **Save**.

When Does the Server Use an Output Template?

The server applies output templates to the results of services that are invoked by HTTP, FTP, or SMTP clients (i.e., requests that come through the HTTP, HTTPS, FTP, or SMTP listeners). You can also arbitrarily apply output templates to the pipeline using the built-in services in the pub.report folder.

Using Output Templates to Return Output to HTTP Clients

If a service has an output template assigned to it, the server automatically applies the template to the results of the service (i.e., the contents of the pipeline) anytime that service is externally invoked by an HTTP client. (If a service does not have an output template, the server simply returns the results of the service in the body of an HTML document, formatted as a two-column table.)

Guidelines for Using HTML-based Output Templates with HTTP Clients

If you want to use an HTML-based output template to return output to an HTTP client, keep the following points in mind:

- Make sure the output template produces a valid HTML document.
- If the client checks the Content-Type value in the HTTP response header, make sure to set the **Type** option (on the **Settings** tab) to "html" when assigning the output template to the service. This option tells the server to set Content-Type to "text/html" when it returns results using this output template.

Guidelines for Using XML-based Output Templates with HTTP Clients

If you want to use an XML-based output template to return output to an HTTP client, keep the following points in mind:

- Make sure the output template generates a valid and well-formed XML document.
- Make sure to set the **Type** option (on the **Settings** tab) to "xml" when assigning the output template to the service. This option tells the server to set Content-Type to "text/xml" when it returns results using this output template.

- If your client is a browser, make sure it can accept and display XML (for example, Microsoft Internet Explorer 5.0 can display XML) or that the client machine has a MIME definition for Content-Type “text/xml”.

Using Output Templates to Return Output to SMTP and FTP Clients

Besides HTTP clients, the server also applies output templates to the results it returns to FTP and email clients. However, be aware that:

- For e-mail clients, the server can apply either XML- or HTML-based output templates.
- For FTP clients, the server will only apply XML-based output templates. If an HTML-based output template is assigned to the service, it is ignored.

Using Output Templates to Return Output to Wireless Devices

The SAP BC Server can use WML (Wireless Markup Language) output templates or HDML (Handheld Device Markup Language) output templates to return output to Internet-enabled wireless devices. You might want to return service output in WML and HDML if you allow the use of wireless devices to invoke services on the SAP BC Server. For example, if you allow clients to submit purchase orders using a wireless device, you would want to be able to send their order confirmation to the wireless device.

For information about how the SAP BC Server communicates with a wireless device, see the *SAP BC Administration Guide*.

Guidelines for Creating WML Output Templates

If you want to use a WML output template to return output to an Internet-enabled wireless device, such as a wireless phone, keep the following points in mind:

- The output template needs to produce a valid and well-formed WML document.
- Make sure to set the **Type** option to “wml” when assigning the output template to a service. This option tells the server to set Content-Type to “text/vnd.wap.wml” when it returns output to the client.
- Wireless devices have small screen dimensions.
- The browser on the wireless device needs to support WML 1.1 or higher to receive a WML output template from the SAP BC Server.
- Different types of Web browsers on wireless devices may display WML pages differently.

- Some Web browsers for wireless devices place a limitation on the length of a URL: name in WML pages. Make sure that you create WML pages that are compliant with the browser requirements.

For more information about WML, see http://www.oasis-open.org/cover/.

Guidelines for Creating HDML Output Templates

If you want to use an HDML output template to return output to an Internet-enabled wireless device, such as a personal digital assistant, keep the following points in mind:

- The output template needs to produce a valid HDML document.
- Make sure to set the **Type** option to "hdml" when assigning the output template to a service. This option tells the server to set Content-Type to "text/x-hdml" when it returns output to the client.
- Wireless devices have small screen dimensions.
- The browser on the wireless device needs to support HDML 3.0 or higher to receive an HDML output template from the SAP BC Server.

For more information about HDML, see http://www.Phone.com.

Applying Output Templates Arbitrarily

You can arbitrarily apply output templates to the contents of the pipeline using the built-in services in the pub.report folder. For information about using these services, see the *SAP BC Built-In Services Guide*.

Creating an Output Template

An output template is an arbitrary string of text that you create and save in a file. You can use Developer to create an output template file or you can create it with an ordinary text editor.

When you create an output template, keep the following points in mind:

- The template file must reside in the template directory of the package where the service resides (i.e., <*sapbc*>\server\packages*packageName*\templates).
- You must give an output template file a name that is unique within the package.
- If you want to work with (i.e., edit or assign) an output template using Developer, you must give the file an HTML, XML, WML, or HDML extension.

To create the contents of an output template file, type all literal text exactly as you want it to appear in the result and then embed any of the following tags where you want the server to execute them at run time. For a complete description of each tag, see Appendix A, "Tag Descriptions" on page 45.

Important! These tags are case-sensitive and must be typed exactly as shown in this document. Additionally, all text between %…% symbols in a tag must appear on one line (i.e. a tag cannot contain line breaks).

Use this tag...	To...
`%value` *VarName*`%`	Insert the value of the specified variable into the output string.

`%scope` *RecordName*`%`
.
.
.
`%end%`
	Limit the scope for the enclosed block of code to those elements in the specified Record.

`%loop` *VarName*`%`
.
.
.
`%end%`
	Repeat the enclosed block of code once for each element in the a specified array variable.

`%ifvar` *VarName*`%`
.
.
.
`%end%`
	Conditionally include the enclosed block of code if a variable exists or matches a specified value.

`%switch` *VarName*`%`
`%case` *Value1*`%`
`%case` *Value2*`%`
`%case` *Value3*`%`
.
.
.
`%end%`
	Conditionally include a block of code depending on the value of a specified variable.

`%switch` *VarName*`%`
`%case` *Value1*`%`
`%case` *Value2*`%`
`%case` *Value3*`%`
.
.
.
`%end%`
	Conditionally include a block of code depending on the value of a specified variable.

`%include` *TemplateName*`%`	Insert and execute a specified output template.
`%nl%`	Insert a new line in the output string.

Use this tag...	To...

`%comment%`
.
.
.
`%end%`
	Omit the enclosed block of text from the output.

`%rename` *VarName NewVarName*`%`	Change the name of a variable for the purpose of referencing it in the output template. This may be necessary if you include predefined output templates that use different variable names than those in the pipeline.

3

Using Dynamic Server Pages (DSPs)

- What Is a Dynamic Server Page? 22
- Creating DSPs 23
- Publishing DSPs 24
- Securing a DSP 24
- Requesting a DSP 25
- Hyperlinks to DSPs 26
- Using the DSP Tags 26
- Arbitrarily Processing DSP Tags 43

What Is a Dynamic Server Page?

A *dynamic server page* (DSP) is a document embedded with special codes (tags) that instruct the SAP BC Server to perform certain actions when an HTTP (or HTTPS) client requests the document. DSPs are used to construct browser-based applications. Because they are HTML-based, they can be used to build complex user interfaces that includes any valid construct (e.g., forms, cascading style sheets, JavaScript) recognized by the client's browser.

DSPs are similar to output templates. In fact, the tags you use to compose output templates are the same ones you use to compose DSPs. However, DSPs have one additional tag—the %invoke% tag—that distinguishes them from output templates. This tag allows a DSP to invoke a service.

Important! When the SAP BC Server returns a DSP, it always sets the value of the HTTP `content-type` header field to `text/html`. Therefore, a DSP should only contain HTML content and should only be used by clients that recognize and accept this content type.

What Does a DSP Look Like?

A DSP looks like an ordinary HTML document that contains additional tags enclosed in % symbols (e.g., %loop%). When a client requests a DSP, the SAP BC Server executes the action specified by the tag and substitutes the result of that action (based on the rules of the tag) in the document it returns to the client.

Note: A DSP tag is never sent to the client; the client receives only the result of the tag.

The following is an example of a very basic DSP (tags are underscored). In this example, the DSP invokes a service and then loops over the list of Records (called *orders*) that the service returns.

```
<HTML>
<HEAD>
<title>Order Tracking System</title>
</HEAD>

<BODY>
<h1>Current Orders</h1>
  %invoke orders:showOrders%
    %loop orders%
    <p>Date:  %value orderDate% PO Number: %value orderNum%</p>
    %endloop%
  %endinvoke%
</BODY>
</HTML>
```

The %invoke% tag invokes a service...

...and the %loop%...%endloop% block inserts the results into the HTML document.

When Do You Use DSPs?

DSPs are used to build browser-based clients (i.e., clients that use a Web browser to retrieve documents). They allow you to construct a more secure and flexible user interface than can be built by directly invoking a service from a browser.

The SAP BC Server Administrator is a good example of the type of user interface you can create with DSPs. The interface for this application is composed entirely of DSPs. You may want to refer to it for design ideas and examples of how to use particular tags. To examine the DSPs that make up the server's user interface, look at the DSP files in the <*sapbc*>\server\packages\WmRoot\pub directory on your SAP BC Server.

What Are the Advantages of Using DSPs?

In the example on page 22, the DSP invokes a service and displays its results. You could accomplish the same thing by invoking the service directly from a browser and applying an output template to the result. However, DSPs have several advantages over directly invoking a service with a URL:

- They conceal the INVOKE mechanism and the name of the service from the user.
- They give you the flexibility to change the name of a service or replace one service with another without changing the way in which the end user invokes the service. (The user always invokes the same DSP, whose contents you can change as needed.)
- They can easily be updated and extended.
- They allow you to execute multiple services via a single request.
- They allow you to conditionally execute a service based on run-time input. For example, you might build a DSP that actually contains several different HTML pages, and use the %switch% tag to select among them.

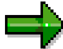
Creating DSPs

To create a DSP, you must compose it with a text editor and then save it on the SAP BC Server (see "Publishing DSPs" on page 24). Unlike output templates, you do not create DSPs with the SAP BC Developer.

When you build a DSP, do the following:

- Type literal text exactly as you want it to appear in the document that you want the SAP BC Server to return to the client.
- Insert DSP tags at the points where you want their results to appear. For a summary of valid DSP tags and how to use them, see "Using the DSP Tags" on page 26.

Important! Make sure the document that you create resolves into a valid HTML document.



Note: While building your DSP, keep in mind that at run time the SAP BC Server will process it once, from top to bottom.

Publishing DSPs

To run a DSP, you must publish it on a SAP BC Server. To do this, take the following general steps.

1 **Save the DSP document in a text file that has a ".dsp" extension.**

Example showorders.dsp

2 **Place the DSP file in the pub directory of the package in which you want the dsp to reside.**

Examples
To publish a DSP in the orders package, you would copy it to:

<*sapbc*>\server\packages\orders\pub

To publish a DSP in the status subdirectory within the orders package, you would copy it to:

<*sapbc*>\server\packages\orders\pub\status

Securing a DSP

When you publish a DSP, you need to configure the server's security mechanisms to protect the DSP from unauthorized access. DSPs have two levels of security protection you need to set.

- **Access to the DSP itself**. Access to a DSP is controlled by the an Access Control List (ACL). An ACL specifies which users have permission to retrieve the DSP. An ACL allows you to make access to the DSP as liberal (e.g., allow access to anyone) or as restrictive (e.g., restrict access to only certain people) as you need.

  To assign an ACL to a DSP, you must update (or create) the .access file in the directory where the DSP resides. For procedures, see "Assigning ACLs to Files that a Server Can Serve" in the *SAP BC Administration Guide*.

Note: Unlike a service, access to a DSP cannot be restricted to a particular port. Thus you do not specify port-level controls for a DSP.

- **Access to services invoked by the DSP**. When a user requests a DSP, the services invoked by the DSP are subject to a port-level check (against the port on which the DSP was requested) and an ACL check (against the user that requested the DSP). To ensure that the services in your DSP execute successfully, you must do the following:

- Make sure that the services it invokes are allowed to execute on the port(s) where the DSP will be requested.
- Make sure that users who are authorized to use the DSP are also authorized to execute the services that the DSP invokes. (For convenience, you might want to assign the same ACL to the DSP and to the services it invokes.)

For information about configuring port-level security and assigning ACLs to services, see the *SAP BC Administration Guide*.

Note: A service that is internally invoked by a service in a DSP is not subject security control unless the **Enforce ACL on Internal Invokes** option is set for the internally invoked service. When this option is set, the server performs an ACL check on the service using the user ID under which the DSP was initially requested

Requesting a DSP

To process a DSP, you request it from a browser using the following URL format:

`http://`*hostName*:*portNum*/*packageName*/*fileName*`.dsp`

where:

hostName	Is the host name or IP address of the SAP BC Server on which the DSP resides.
portNum	Is the port number on which the SAP BC Server listens for HTTP requests.
packageName	Is the name of the package to which the DSP belongs. *packageName* must match the package directory in which the DSP resides within <*sapbc*>\server\packages on the server. If you do not specify a package name, the server looks for the named DSP in the Default package. *This parameter is case-sensitive!*
fileName.dsp	Is the name of the file containing the DSP. This file name must have a ".dsp" extension, and it must reside within the pub directory (or a subdirectory beneath pub) under the package directory named in *packageName*. If the DSP resides in a subdirectory, include the name of that subdirectory in the file name (see example below). *This parameter is case-sensitive!*

Examples

The following URL retrieves showorders.dsp from a package named ORDER_TRAK on a server named rubicon:

```
http://rubicon:5555/ORDER_TRAK/showorders.dsp
```

The following URL retrieves showorders.dsp from the STATUS subdirectory in a package named ORDER_TRAK on a server named rubicon:

```
http://rubicon:5555/ORDER_TRAK/STATUS/showorders.dsp
```

The following URL retrieves showorders.dsp from the Default package on a server named rubicon:

```
http://rubicon:5555/showorders.dsp
```

Hyperlinks to DSPs

Typing the DSP's URL on the address line in your browser is one way to run a DSP. However, when you use DSPs to build a user interface, you will often invoke DSPs from HTML forms and links as shown in the following example.

```
    <HTML>
    <HEAD>
    <title>Order Tracking System</title>
</HEAD>

<BODY>
   <A HREF=/ORDER_TRAK/showorders.dsp>Show Orders</A>
</BODY>
</HTML>
```

Using the DSP Tags

To develop a DSP, you embed DSP tags where you want the results of the tags to appear. The following is a summary of tags that you can use to build DSPs. For a complete description of each tag, see Appendix A, "Tag Descriptions" on page 45.

Important! DSP tags are case-sensitive. In your DSP, you must type them exactly as shown below (e.g., type `%loop%`, not `%LOOP%`).

Use this tag...	To...
%invoke%	Invoke a service within a DSP
%value% %rename% %scope%	Manipulate variables.
%ifvar% %switch%	Conditionally process a block of code within a DSP.
%loop%	Reiterate a block of code within a DSP.

Use this tag...	To...
%include%	Insert the contents of a text file (which may contain additional DSP tags) into the DSP.
%comment%	Denote a comment within a DSP. Comments are neither processed by the DSP Processor nor returned to the requestor.

Note: Tags are automatically resolved by SAP BC Server when a client requests that DSP via an HTTP or HTTPS request. If you want to resolve tags within a document at some arbitrary point in a service, you can explicitly run the DSP Processor against the document using the services in the pub.report folder. For information about using these services, see "Arbitrarily Processing DSP Tags" on page 43.

Begin…End Constructs

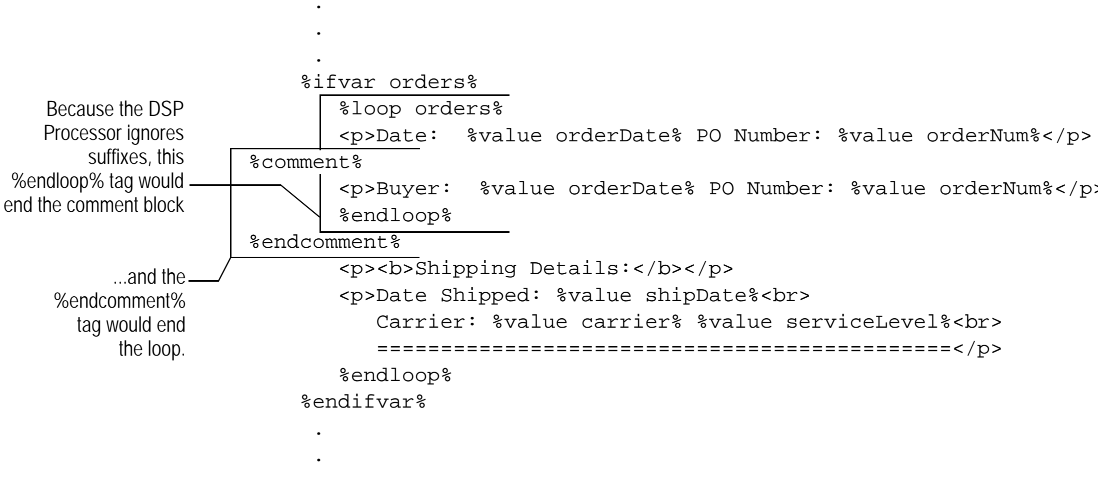
Many DSP tags have both beginning and ending elements. When you use the %loop% tag, for example, you enclose the code over which you want the DSP Processor to iterate, within a %loop%...%end% construct.

To make your DSP easier to read, you can append a suffix to the %end% element of any construct to visually associate it with its beginning element. For example, in the following DSP, the %end% element of the %loop% construct is named %endloop% and the %end% element for the %ifvar% construct is named %endifvar%.

```
    .
    .
    .
  %ifvar orders%
     %loop orders%
     <p>Date:  %value orderDate% PO Number: %value orderNum%</p>
     %endloop%
  %endifvar%
    .
    .
    .
```

Be aware that only the first three characters of an %end% element are significant. The DSP Processor ignores any suffixes that you add and simply associates an %end% element with the most recent beginning element (in other words, the %end% element always ends the *current* construct). This means that you can nest one construct within another (as shown above), but you cannot overlap them.

For example, you cannot use the %comment%...%endcomment% construct to "remark out" an %endloop% element as shown in the following sample.



Invoking Services Using the %invoke% Tag

You use the %invoke% tag to invoke a service in a DSP. When this tag is processed, the SAP BC Server executes the specified service at the point where the tag appears and returns the results of the service to the DSP processor.

The basic format of the %invoke% tag is as follows, where *serviceName* is the fully qualified name of the service that you want to invoke:

```
%invoke serviceName%
    Block of Code
[%onerror%
    Block of Code]
%end%
```

Example

```
  .
  .
  .
%invoke orders:getShipInfo%
    <p>
    Date Shipped:  %value shipDate%<br>
    Carrier:  %value carrier% %value serviceLevel%
    </p>
%onerror%
    %include standarderror.txt%
%endinvoke%
  .
  .
  .
```

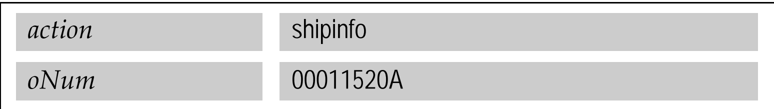
What Is Scope?

When you invoke a service in a DSP, notice that you do not specifically state which parameters you want to pass to it. Instead, the service automatically receives an IData object containing all the variables that are in the DSP's current *scope*.

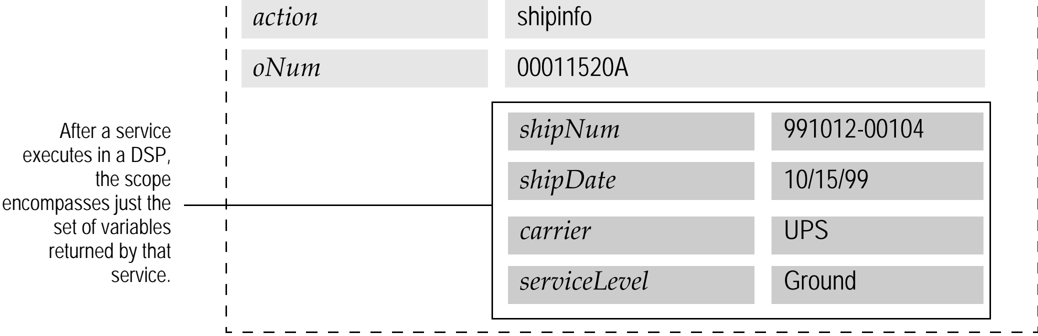
Scope refers to the set of variables upon which a DSP can operate directly. When a DSP is initially invoked, its scope encompasses the set of *name*=*value* pairs it receives (via GET or POST) from the requestor. For example, if you were to request a DSP with the following URL:

```
http://rubicon:5555/ORDER_TRAK/getorderinfo.dsp?action=shipinfo&oNum=00011520
```

Its initial scope would look like this:



After you execute a service with the %invoke% tag, the scope automatically switches to encompass the set of variables returned by that service. For example, if the DSP in the preceding example invokes a service that returns shipping information, that DSP's collection of variables would look like this after the service executes. Note that just the variables returned by the service are within scope.



After a service executes in a DSP, the scope encompasses just the set of variables returned by that service.

Conceptually, you can think of a DSP as maintaining its run-time variables in a set of nested containers. It starts with a container that holds the variables submitted by the requestor. When a service is invoked, it puts the results from the service in a new container inside the initial container—the new container comprises the current scope. If another service is invoked while that container is open, the variables returned by the from the service are put into another container, which is placed inside the previous container, and so forth.

Important! Besides the %invoke% tag, certain other tags (e.g., %loop%) implicitly switch scope. This behavior is noted in the tag descriptions in Appendix A, “Tag Descriptions” on page 45.

Why Does Scope Matter?

Understanding (and controlling) the scope within a DSP is important for two reasons:

- Only those variables that are within the current scope are passed to a service that you invoke in a DSP. So, to ensure that a service receives all the variables it needs at run time, you must make sure that all those variables are all within scope. For more information about passing parameters to and within a DSP, see “Passing Parameters with a DSP” on page 33.
- Only those variables that are within scope can be addressed without qualifiers. (See “Referencing Variables In and Out of Scope” on page 32.) Moreover, except for the initial scope (whose variables exist for the entire life of a DSP), certain tags, such a the %end% tag, cause the current scope to close *and* discard the variables within it.

Ending the Scope of the Invoke Action

The %end% element in the %invoke%...%end% construct ends the scope for that invoke. It marks the point in the DSP where the variables associated with that construct are dropped and scope reverts to the previous level.

The following example shows a DSP that invokes two services in sequence. In this DSP, both services receive the variables from the *initial* scope as input.

```
<HTML>
<HEAD>
<title>Order Tracking System</title>
</HEAD>
<!--User passes in order number in param named <oNum>  -->
<!--The initial scope contains only <oNum>             -->
```

```
%invoke orders:getShipInfo%
<!--Scope switches to results of getShipInfo          -->
<!--You can reference variables in the initial scope  -->
<!--with a relative addressing qualifier              -->
    <p>
    Order:  %value /oNum%<br>
    Date Shipped:  %value shipDate%<br>
    Carrier:  %value carrier% %value serviceLevel%
    </p>
%endinvoke%
```

```
<!--Scope reverts to the initial scope               -->
<!--Results from getShipInfo have been discarded     -->
<!--and cannot be accessed beyond this point         -->
```

```
%invoke orders:getCustInfo%
<!--Scope switches to results of getCustInfo         -->
<!--You can reference variables in the initial scope -->
<!--with a relative addressing qualifier             -->
    <table>
    <tr><td>Company:</td>   <td>%value companyName%</td></tr>
    <tr><td>Phone:</td>     <td>%value phoneNum%</td></tr>
    <tr><td>Address:</td>   <td>%value StreetAddr1%<br>
                                %value StreetAddr2%</td></tr>
    <tr><td></td >          <td>%value city%, %value state%</td></tr>
    <tr><td></td >          <td>%value postalCode%</td></tr>
    </table>
%endinvoke%
```

```
<!--Scope reverts to the initial scope              -->
<!--Results from getCustInfo have been discarded    -->
<!--and cannot be accessed beyond this point        -->
</BODY>
</HTML>
```

The following example shows a DSP that invokes two services. In this example, the second service is invoked within the scope of the first service, so it receives the output from the first service as input.

```
<HTML>
<HEAD>
<title>Order Tracking System</title>
</HEAD>
<!--User passes in order number in param named <oNum>  -->
<!--The initial scope contains only <oNum>             -->
```

```
%invoke orders:getShipInfo%
<!--Scope switches to results of getShipInfo          -->
<!--You can reference variables in the initial scope  -->
<!--with a relative addressing qualifier              -->
    <p>
    Order:  %value /oNum%<br>
    Date Shipped:  %value shipDate%<br>
    Carrier:  %value carrier% %value serviceLevel%
    </p>
%invoke orders:getCustInfo%
<!--Scope switches to results of getCustInfo           -->
<!--You can reference variables in the initial scope   -->
<!—and prior scope with relative addressing qualifiers -->
    <table>
    <tr><td>Company:</td>   <td>%value companyName%</td></tr>
    <tr><td>Phone:</td>     <td>%value phoneNum%</td></tr>
    <tr><td>Address:</td>   <td>%value StreetAddr1%<br>
                                %value StreetAddr2%</td></tr>
    <tr><td></td >          <td>%value city%, %value
                                 state%</td></tr>
    <tr><td></td >          <td>%value postalCode%</td></tr>
    </table>
%endinvoke%
<!--Scope reverts back to results of getShipInfo            -->
<!--Results from getCustInfo have been discarded            -->
<!--and cannot be accessed beyond this point                -->
%endinvoke%
```

```
<!--Scope reverts to the initial scope               -->
<!--Results from getShipInfo have been discarded    -->
<!--and cannot be accessed beyond this point         -->

</BODY>
</HTML>
```

Referencing Variables In and Out of Scope

You can refer to variables that are in the current scope directly—without any qualifiers. To reference a variable that is out of scope, you must use the following directory-like notation to describe its position relative to either the current scope or the initial scope.

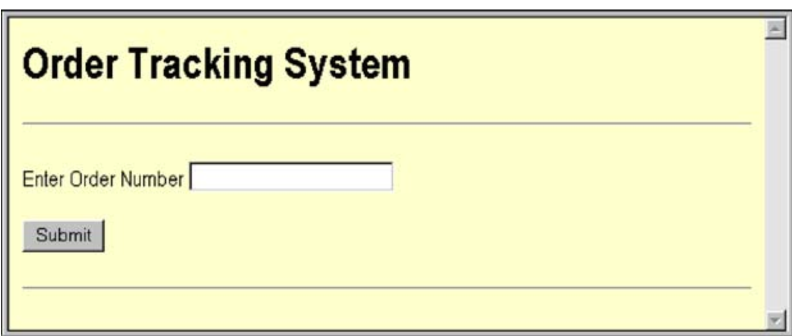
Use this notation...	To...
variableName	Reference a variable in the current scope. **Example:** `shipNum`
../variableName	Reference a variable one or more levels above the current scope. **Example:** `../oNum` One level above **Example:** `../../oNum` Two levels above

Use this notation...	To...	
/variableName	Reference a variable in the initial scope.	
	Example: `/oNum`	
recName/variableName	Reference a variable within a specific Record.	
	Example: `buyerInfo/state`	Selects the *state* element from the Record *buyerInfo* in the current scope
	Example: `../buyerInfo/oNum`	Selects the *state* element from the Record *buyerInfo* one level above current scope

Passing Parameters with a DSP

You use the standard HTTP "GET" and "POST" methods to pass input parameters to a DSP. In a browser-based client, you usually do this with an HTML form. For example, if you were creating an order-tracking application, you might create a form that prompts the user for an order number and invokes a DSP that returns the shipping status of that order.

You can use an HTML form to pass input to a DSP



```
<HTML>
<HEAD>
<TITLE>Order Tracking System</TITLE>
</HEAD>
<BODY BGCOLOR="#FFFFCC">
<H1>Shipping Information</H1>
<HR>

<FORM ACTION="/ORDER_TRAK/getorderinfo.dsp" METHOD="GET">
  <P>Enter Order Number         <INPUT TYPE="TEXT" NAME="oNum"> <BR>
  <INPUT TYPE="HIDDEN" NAME="action" VALUE="shipinfo">          <BR>
  <INPUT TYPE="SUBMIT" VALUE="Submit">
</FORM>

<HR>
</BODY>
</HTML>
```

Use an HTML form to invoke a DSP...

...and pass input parameters to it.

When the DSP is invoked, its initial scope encompasses two parameters: *oNum* and *action*. Services that you invoke within this scope receive an IData object containing these two elements.

The following code shows the contents of the DSP (getorderinfo.dsp) invoked by the previous example. It uses the value in *action* to conditionally execute a section of the DSP that invokes the service. (For more information about using the %switch% tag to conditionally execute a section of code, see "Building Conditional Blocks with the %switch% Tag" on page 40.)

```
<HTML>
<HEAD>
<title>Order Tracking System</title>
</HEAD>
<BODY BGCOLOR="#FFFFCC">

<!--User passes in order number in param named <oNum> -->
<!--and requested action in <action>                   -->

<H1>Order Tracking System</H1>
%switch action%
  %case 'shipinfo'%
     %invoke orders:getShipInfo%
     <p>
     Order:  %value /oNum%<br>
     Date Shipped:  %value shipDate%<br>
     Carrier:  %value carrier% %value serviceLevel%
     </p>
     %endinvoke%
  %case 'orderinfo'%
     %invoke orders:getOrderInfo%
  .
  .
  .
```

Passing Parameters Between DSPs

Because HTTP does not preserve variables from one request to another, to pass data from one DSP to another, you must explicitly set those values in the documents that you return to the requestor. For example, let's say you want to allow your user to view or edit the shipment displayed by the order-tracking DSP above. To do this, you must return a document containing links to the DSPs that perform these tasks, and these DSPs will need the order number (*oNum*) that the user submitted on the original page. To pass the order number to these DSPs, you must put *oNum* in the page you return to the client.

The following example shows two ways in which you can build a DSP that will pass a variable to another DSP.

```
<HTML>
<HEAD>
<title>Order Tracking System</title>
</HEAD>
<BODY BGCOLOR="#FFFFCC">

<H1>Order Tracking System</H1>
<!--User passes in order number in param named <oNum> -->
<!--and requested action in <action>                  -->
%switch action%
  %case 'shipinfo'%
  %invoke orders:getShipInfo%
    <H2>Shipping Information for Order %value /oNum%</H2>
    <P>Date Shipped:  %value shipDate%<BR>
    Carrier:  %value carrier% %value serviceLevel%
    </P>
    <HR>
    %ifvar shipDate -isnotempty%
      <FORM ACTION="/ORDER_TRAK/editshipinfo.dsp" METHOD="get">
      <P><B>Change this Shipment:</B> </P>
      <P><INPUT TYPE="RADIO" NAME="action" VALUE="edit">
         Edit Shipment Details</P>
      <P><INPUT TYPE="RADIO" NAME="action" VALUE="cancel">
         Cancel this shipment</P>
         <INPUT TYPE="SUBMIT" VALUE="Submit">
      <INPUT TYPE="HIDDEN" NAME="oNum" VALUE="%value /oNum">
      </FORM>
      <HR>
    %endifvar%
    <P><A HREF="/ORDER_TRAK/getorderinfo.dsp
        ?action=orderinfo&oNum=%value /oNum%">View Entire Order</A></P>
  %endinvoke%
  %case 'getorder'%
    %invoke orders:getOrderInfo%
      .
      .
      .
%endswitch%
</BODY>
</HTML>
```

You can pass a parameter in a hidden input element...

...or as a name-value pair in a link to a DSP.

Passing Parameters Between Services within a DSP

To pass data between services within a DSP, you can use any of the following techniques:

- Invoke the service that needs the parameter within the scope of the service that produces the parameter. When a service is invoked in a DSP, it receives all of the variables that are "in scope" at the point where it is invoked. For an example of this, see the sample code on page 31.
- Use the %rename% tag to copy a variable that is out of scope into the current scope. For details and examples, see the %rename% tag description on page 56.
- Use the %scope% tag to add a variable to the current scope and specify its value. For details and examples, see the %scope% tag description on page 58.

Catching Errors

If you want your DSP to react in a specified way when the invoked service fails, include an %onerror% tag within your %invoke%…%end% construct. The code in the %onerror% block executes only if an exception occurs while the service executes or the service returns an error.

When the %onerror% block executes, the scope contains the following values:

Key...	Description
error	A String containing the Java class name of the exception that was thrown (e.g., com.wm.app.b2b.server.AccessException).
errorMessage	A String containing the exception message.
errorInput	The IData object that was passed to the invoked service.
errorOutput	The IData object returned by the invoked service. If the service returned an error, *errorOutput* will contain $error and any other variables that were in the pipeline when the service ended. **Important!** If the service experiences an exception (i.e., the server is not able to execute it successfully), *errorOutput* will not be exist. This variable is only produced when the service returns an error.
errorService	The name of the invoked service.

The following example shows how you might use an %onerror% clause to return an error message to the user.

```
   .
   .
   .
%invoke orders:getShipInfo%
  <H2>Shipping Details for Order %value /oNum%</H2>
  <P>Date Shipped:  %value shipDate%<BR>
    .
    .
    .
%onerror%
  <HR>
  <P><FONT COLOR="#FF0000">The Server could not process your request
  because the following error occurred. Contact your server
  administrator.</FONT></P>
  <TABLE WIDTH="50%" BORDER="1">
  <TR><TD><B>Service</B></TD><TD>%value errorService%</TD></TR>
  <TR><TD><B>Error</B></TD><TD>%value error%  
                              %value errorMessage%</TD></TR>
  </TABLE>
  <HR>
%endinvoke%
    .
    .
    .
```

Extracting Results from an Array Variable

If a service returns an array variable—such as a String List, a String Table, or a Record List—to a DSP, you use the %loop% tag with a *variableName* to extract values from the elements in the array.

When you use %loop% on a Record variable, the scope within the loop block automatically changes to encompass just those elements within the specified Record.

The following example shows a loop that extracts values from a Record List (called *items*) that is returned by the service named orders:getOrderInfo.

```
%invoke orders:getOrderInfo%
  <P>This shipment contains the following items</P>
  <TABLE WIDTH="90%" BORDER="1">
  <TR><TD>Number</TD><TD>Qty</TD><TD>Description</TD><TD>Status</TD></TR>
```

```
   %loop items%
      <TR>
      <TD>%value stockNum%</TD>
      <TD>%value qty%</TD>
      <TD>%value description%</TD>
      <TD>%value status%</TD>
      </TR>
   %endloop%
   </TABLE>
 %endinvoke%
     .
     .
     .
```

For additional information about the %loop% tag, see the %loop% tag description on page 52.

Extracting Results from a Record

To extract results from a Record (i.e., an IData object), you use the %loop% tag with the -`struct` option to execute a block of code once for each key in the structure.

The following example shows how you would use to extract values from each key in a Record named *buyerInfo*.

```
 %invoke orders:getOrderInfo%
   <P>Buyer:</P><P>
   %loop -struct buyerInfo%
      %value%<BR>
   %endloop%
   </TABLE>
     .
     .
     .
```

Using the %loop% Tag to Examine the Current Scope

If you use the –`struct` option without specifying the name of a Record, the loop executes once for each element in the current scope. During testing and debugging, you may want to use this technique to examine the variables and their values at a particular point in the DSP. The following example shows the code you would use to display the name of each key and its contents in the current scope.

```
     .
     .
     .
<P>
  %loop -struct%
     %value $key% %value%<BR>
  %endloop%
</P>
     .
     .
     .
```

Conditionally Executing Blocks of Code

There are two tags you can use to conditionally execute code in a DSP: %ifvar% and %switch%. Both tags selectively execute a block of code based on the existence or value of a variable at run time.

Building Conditional Blocks With the %ifvar% Tag

The %ifvar% tag is similar to an "if…then…else" expression in other programming languages. You use it to denote a block of code that is to be executed only when a specified variable exists or contains a value that you specify.

The basic format of the %ifvar% tag is as follows, where *variableName* specifies the name of the variable that will be evaluated at run time:

```
%ifvar variableName%
    Block of Code
[%else%
    Block of Code]
%end%
```

Example

```
      .
      .
      .
<!--Check for presence of backordered items in the order -->
<!—and display if they exist                              -->
   %ifvar backItems%
       <p>Backordered Items
       %loop backItems%
           %value%<BR>
       %endloop%
   %endifvar%
      .
      .
      .
```

Testing for a Particular Value

In the preceding example, the enclosed block of code executes if a variable named *backItems* exists in the current scope. To test for the content of a variable, you can apply the following options to the %ifvar% tag.

Use this option...	To...
`-isnull`	Test whether the specified variable exists and is null.
`-notempty`	Test whether the specified variable contains a value (i.e., the value is not null or empty).
`equals('`*anyString*`')`	Test whether the specified variable contains a specific value. (*variableName* must be a String variable to use this option.)
`vequals(`*refVariable*`)`	Test whether the value of the specified variable matches the contents of another variable in the pipeline.

The following example shows an %ifvar%...%else%...%end% construct that executes one of two blocks, depending on the contents of a variable named *clubMember*.

```
   .
   .
   .
%invoke orders:getShipInfo%
  <H2>Shipping Details for Order %value /oNum%</H2>
  <P>Date Shipped:  %value shipDate%<BR>
    .
    .
    .
  %ifvar clubMember equals('Y')%
    %include membershipterms.txt%
  %else%
    %include nonmembershipterms.txt%
  %endifvar%
%endinvoke%
    .
    .
    .
```

For additional information about the %ifvar% tag, see the %ifvar% description on page 47.

Building Conditional Blocks with the %switch% Tag

You can use the %switch% tag to construct a conditional expression based on the value of a specified variable. The %switch% tag allows you to define a separate block of code (a case) for each value that a variable can take at run time. You use this tag instead of %ifvar% when you have more than two possible paths of execution. (You can also handle multiple paths of execution by building multiple %ifvar% expressions, however, the %switch% tag is much easier to use and maintain for this purpose.)

The basic format of the %switch % tag is as follows, where *variableName* specifies the name of the variable you want to evaluate at run time and *switchValue* is the value that will cause a case to execute:

```
%switch variableName%
  %case 'switchValue'%
    Block of Code
  %case 'switchValue'%
    Block of Code
     .
     .
     .
  [%case%
    Default Block of Code]
%end%
```

At run time, the DSP Processor evaluates each %case% block in order, and executes the first block whose *switchValue* matches the value in *variableName.* After processing the code within that block, the DSP processor skips the remaining cases in the %switch% construct.

The following example shows an %switch% construct that executes one of two cases, depending on the contents of a variable named *action*.

```
     .
     .
     .
%switch action%
  %case 'shipinfo'%
     %invoke orders:getShipInfo%
        .
        .
        .
     %endinvoke%
  %case 'vieworder'%
     %invoke orders:getOrderInfo%
        .
        .
        .
     %endinvoke%
%endswitch%
```

Specifying a Default Case

If you want to specify a block of code that executes when all other cases are not true, include a case without a *switchValue* as shown in the following example:

```
      .
      .
      .
%switch acctType%
  %case 'Platinum'%
     %include platorderform.html%
  %case 'Gold'%
     %include goldform.html%
  %case%
     %include regorderform.html%
%endswitch%
```

To create a default case, omit *switchValue* and put this case at the end of the %switch% construct.

If you include a default case, you must put it at the end of the switch construct. For additional information about using the %switch% tag, see the %switch% tag description on page 62.

Inserting Text Files In a DSP

The %include% tag allows you to insert a text file in a DSP. When you use the %include% tag, the DSP Processor inserts the specified file and evaluates its contents (and processes any tags that it contains) from top to bottom at run time.

The basic format of the %include% tag is as follows, where *fileName* specifies the name of the file that you want to insert into the DSP. (If *fileName* is not in the same directory as the DSP, you must specify its path relative to the DSP as shown by the example.)

```
%include fileName%
```

Example

```
      .
      .
      .
%switch acctType%
  %case 'Platinum'%
     %include forms\platorderform.txt%
  %case 'Gold'%
     %include forms\goldform.txt%
  %case%
     %include forms\regorderform.txt%
%endswitch%
```

When you insert a file, it inherits the scope that is current at the point where you call it. For additional information about the %include% tag, see the %include% tag description on page 49.

Arbitrarily Processing DSP Tags

The SAP BC Server automatically processes DSP tags when a DSP is requested via HTTP. You can also process the tags in a string of text at any arbitrary point during a service using the services in the pub.report folder. When you use these services, tags are processed against the variables that are in the pipeline at run time.

For information about using the services in the pub.report folder, see the *SAP BC Built-In Services Guide*.

Tag Descriptions

- Overview . . . 45
- %comment% . . . 46
- %ifvar% . . . 47
- %include% . . . 49
- %invoke% . . . 50
- %loop% . . . 52
- %loopsep% . . . 55
- %nl% . . . 56
- %rename% . . . 56
- %scope% . . . 58
- %switch% . . . 62
- %sysvar% . . . 64
- %value% . . . 65

Overview

This appendix describes the tags you use to construct output templates and Dynamic Server Pages (DSPs).

Important! Tags are case-sensitive and must be typed into a template or DSP exactly as shown in this appendix (e.g., %loop%, not %LOOP% or %Loop%).

Important! All text between %…% symbols in a tag must appear on one line (i.e., no line breaks).

The examples shown in this appendix assume a pipeline that looks as follows:

Contents of the Pipeline

Key	Value	
submittor	Mark Asante	
shipNum	991015-00104	
shipDate	10/15/99	
carrier	UPS	
serviceLevel	Ground	
arrivalDate	10/18/99	
items	Key	Value
	qty	10
	stockNum	BK-XS160
	description	Extreme Spline 160 Snowboard- Black
	orderNum	GSG-99401088
	status	Partial Order
	qty	15
	stockNum	WT-XS160
	description	Extreme Spline 160 Snowboard- White
	orderNum	GSG-99401088
	status	Complete
supplierInfo	Key	Value
	companyName	Bitterroot Boards, LLC
	streetAddr1	1290 Antelope Drive
	streeAddr2	
	city	Missoula
	state	MT
	postalCode	59801
	supplierID	BRB-950817-001
	phoneNum	406-721-5000
	faxNum	406-721-5001
	email	Shipping@BitterrootBoards.com
buyerInfo	Key	Value
	companyName	Global Sporting Goods, Inc.
	accountNum	
	phoneNum	(216) 741-7566
	faxNum	(216) 741-7533
	streetAddr1	10211 Brookpark Road
	streeAddr2	
	city	Cleveland
	state	OH
	postalCode	22130
	email	Receiving@GSG.com
backItems	SL-XS170 Extreme Spline 170 Snowboard- Silver	
	BL-KZ111 Kazoo 111 Junior Board- Blue	
	BL-KZ121 Kazoo 121 Junior Board- Blue	

%comment%

You use the %comment% tag to include remarks in your code. At run time, the server ignores all text (and tags) between the %comment% and %end% tag.

Syntax

```
%comment%
    Block of Code
%end%
```

Effect on Scope

None

Examples

The following example contains a section of explanatory information.

```
%comment%
Use this template to generate an order list from any Record containing a
purchased item number, quantity, description, and PO number
%end%
<tr>
<td>%value stockNum%</td>
<td>%value qty%</td>
<td>%value description%</td>
<td>%value orderNum%</td>
```

</tr>

%ifvar%

You use the %ifvar% tag to conditionally include or exclude a block of code based on the existence or value of a specified variable.

Syntax

```
%ifvar Variable [option option option…]%
    Block of Code
[%else%
    Block of Code]
%end%
```

Arguments

Variable specifies the name of the variable that determines whether or not the enclosed block of code is processed.

Options

You can use any of the following options with this tag. To specify multiple options, separate them with spaces.

Options	Description
`-isnull`	Includes the enclosed block of code only if *Variable* is null. **Example** `%ifvar backItems -isnull%`
`-notempty`	Includes the enclosed block of code only if *Variable* contains one or more characters (for string variables only). **Example** `%ifvar supplierInfo/email -notempty%`
`equals('`*AnyString*`')`	Includes the enclosed block of code only if the value of *Variable* matches the string you specify in *AnyString*. (*AnyString* is case-sensitive. 'FedEx' does not match 'Fedex' or 'FEDEX'). **Example** `%ifvar carrier equals('FedEx')%`
`vequals(`*RefVar*)	Includes the enclosed block of code only if the value of *Variable* matches the value of the variable you specify in *RefVar*. **Example** `%ifvar supplierInfo/state vequals(buyerInfo/state)%`
matches (’*AnyPattern*”)	Includes the enclosed block of code only if the value of *Variable* matches the value of the variable you specify in *AnyPattern* (*AnyPattern* is case sensitive: ’F*Ex’ does not match ’Fedex’ or ’FEDEX’. **Example** `%ifvar carrier matches('DH*')%`

Effect on Scope

None

Notes

For readability, you can optionally use %endif% or %endifvar% instead of %end% to denote the end of the %ifvar% block.

Examples

The following example inserts a paragraph if a variable named *AuthCode* exists.

```
.
.
.
%ifvar AuthCode%
   <p>Authorization Code Received %value $date%: %value AuthCode%</p>
%endif%
.
.
.
```

The following example generates a line for each element in the *backItems* String List only if *backItems* exists; otherwise, it prints a standard message.

```
.
```

```
.
.
%ifvar backitems%
   <p>The following items are backordered</p>
   <p>
   %loop backItems%
      %value%<BR>
   %endloop%
</p>
%else%
   <p>There are no backordered items pending for your account</p>
%end%
.
.
.
```

%include%

You use the %include% tag to reference a text file. When you %include% a text file, the server inserts the contents of the specified file (processing any tags it contains) at run time. If you use template and/or DSPs extensively, you may want to build a library of standard “code fragments” that you reference using %include% tags as needed.

Syntax

`%include` *FileName*`%`

Arguments

FileName specifies the name of the text file that you want to insert. If the text file is not in the same directory as the template or DSP that references it, *FileName* must specify its path relative to the template or DSP file.

Effect on Scope

None. If the inserted file contains tags, they inherit the scope that is in effect at the point where the %include% tag appears.

Examples

The following example inserts a file called “TMPL_ShipToBlock.html” into the code. Because path information is not provided, the server expects to find this file in the same directory as the file containing the template or DSP.

```
.
.
.
%scope buyerInfo%
   <p>Shipped To:<br>
      %include TMPL_ShipToBlock.html%</p>
%end%
.
.
.
```

The following example inserts a file called “TMPL_ShipToBlock.html” into the code. At run time, the server expects to find this file in a subdirectory called “StandardDSPs” in the directory where the template or DSP resides.

```
.
.
.
%scope buyerInfo%
   <p>Shipped To:<br>
      %include StandardDSPs/TMPL_ShipToBlock.html%</p>
%end%
.
.
.
```

The following example inserts a file called “TMPL_ShipToBlock.html” into the code. At run time, the server expects to find this file in the template or DSPs parent directory.

```
.
.
.
%scope buyerInfo%
   <p>Shipped To:<br>
      %include ../TMPL_ShipToBlock.html%</p>
%end%
.
.
.
```

%invoke%

You use the %invoke% tag to execute a service from a DSP. When you use this tag, the server executes the specified service at run time and returns the results of the service to the DSP. You may optionally include the %onerror% tag within the %invoke% block to define a block of code that executes if an exception occurs while the service executes or the service returns an error.

This tag can only be used in DSPs. It cannot be used in an output template.

Syntax

```
%invoke serviceName%
   Block of Code
[%onerror%

   Block of Code]
%end%
```

Arguments

serviceName specifies the fully-qualified name of the service that you want to invoke.

Effect on Scope

Within an %invoke% block, the scope switches to the results of the service. If the service fails, the scope within the %onerror% block contains the following:

Key	Description
error	A String containing the Java class name of the exception that was thrown (e.g., com.wm.app.b2b.server.AccessException).
errorMessage	A String containing the exception message.
errorInput	The IData object that was passed to the invoked service.
errorOutput	The IData object containing the output returned by the invoked service. If the service returned an error, *errorOutput* will contain *$error*. If the service experienced an exception, *errorOutput* will not exist.
errorService	The name of the invoked service.

Examples

The following example invokes a service that returns shipping information and a form allowing the user to optionally edit or cancel an order. This example also includes a %onerror% block that displays error information if the service fails.

```
%invoke orders:getShipInfo%
  <H2>Shipping Details for Order %value /oNum%</H2>
  <P>Date Shipped:  %value shipDate%<BR>
  Carrier:  %value carrier% %value serviceLevel%
  </P>
  <HR>
  %ifvar shipDate -isnotempty%
    <FORM ACTION="http://rubicon:5555/orders/editShipInfo.dsp" METHOD="get">
    <P><B>Change this Shipment:</B> </P>
    <P><INPUT TYPE="RADIO" NAME="action" VALUE="edit">
       Edit Shipment Details</P>
    <P><INPUT TYPE="RADIO" NAME="action" VALUE="cancel">
       Cancel this shipment</P>
       <INPUT TYPE="SUBMIT" VALUE="Submit">
    <INPUT TYPE="HIDDEN" NAME="oNum" VALUE="%value /oNum">
    </FORM>
    <HR>
  %endifvar%
  <P><A HREF="http://rubicon:5555/orders/getorder.dsp
      ?action=showorder&oNum=%value /oNum%">View Entire Order</A></P>
%onerror%
  <HR>
  <P><FONT COLOR="#FF0000">The Server could not process your request
  because the following error occurred. Contact your server
  administrator.</FONT></P>
  <TABLE WIDTH="50%" BORDER="1">
  <TR><TD><B>Service</B></TD><TD>%value errorService%</TD></TR>
  <TR><TD><B>Error</B></TD><TD>%value error%  
                             %value errorMessage%</TD></TR>
  </TABLE>
  <HR>
%endinvoke%
    .
    .
    .
```

%loop%

You use the %loop% tag to repeat a block of code once for each element in a specified array (String List or Record List) or for each key in a Record.

Syntax

`%loop` [*Variable*] [*option option option…*] `%`
  Block of Code
`%end%`

Arguments

Variable specifies the name of the array variable over which you want the enclosed section of code to iterate.

- You may optionally omit *Variable* and specify the `–struct` option to loop over each element in the current scope.
- When looping against a set of complex elements (e.g., Records, Record Lists, String Lists) in a Record, you can optionally use the #$key keyword to specify *Variable* instead of explicitly specifying a key name. When you use #$key in place of an explicit key name, it indicates that you want to apply the body of the loop to (i.e., switch the scope to) the current key. This allows you to process the contents of a Record whose key names are not known. It is most often used with the `–struct` option to process a set of Records contained within another Record. For an example of how this option is used, see the **Examples** section, below.

If *Variable* is a...	The loop is applied to...
String List	Each String in the list.
Record List	Each Record in the Record List.
Record	Each key in the Record. When you use %loop% to process the elements of a Record, you must also use the –struct option in the %loop% tag.

Options

You can use any of the following options with this tag. To specify multiple options, separate the options with spaces.

Option	Description
`-struct`	Specifies that *Variable* is a Record and instructs the server to apply the loop once to each key in that Record. When you use the –struct option, you can use the *$key* variable to retrieve the name of each element in the Record. See examples, below.
`-eol`	Ends the body of the loop at the next end-of-line (EOL) character in the code. When you use –eol, you can omit the %end% tag.

Effect on Scope

If *Variable* is a Record List, scope switches to the Record on which the loop is executing.

Notes

Omit the variable name from the %value% tag if it is used in the body of a loop for a String table or a Record. When variable name is omitted, the server inserts the value of the current element at run time.

For readability, you can optionally use %endloop% instead of %end% to denote the end of a %loop% block.

Examples

The following example generates a paragraph for each record in the *items* Record List.

```
.
.
.
<p>This shipment contains the following items:</p>
%loop items%
   <p>%value qty% %value stockNum% %value description% %value status%</p>
%end%
.
.
.
```

The following example generates a line for each element in the *backItems* String List.

```
.
.
.
<p>The following items are backordered</p>
<p>
%loop backItems%
   %value%<BR>
%end%
</p>
.
.
.
```

The following example shows how you can nest %loop% tags to process the individual Record elements in a Record list.

```
.
.
.
<This shipment contains the following items:</p>
<table>
%loop items%
<tr>
   %loop -struct%
   <td>%value%</td>
   %end%
</tr>
%end%
.
.
.
```

The following example shows how you can use the %loop% tag to dump the contents (key names and values) of the current scope.

```
.
.
.
%loop -struct%
```

```
   %value $key% %value%<BR>
%end%
.
.
.
```

The following example shows how you use the #$key option to loop over the individual elements in a collection of Records contained within another Record.

```
This example assumes that the service returns a Record named MatchingAddresses
that holds a set of Records (of unknown name and quantity) containing address
information.
.
.
.
<p>The following addresses were returned:</p>
%loop -struct MatchingAddresses%
   %loop -struct #$key%
      <p>
      %value streetAddr1%
      %value streetAddr2%
      %value city%, %value state% %value postalCode%
      </p>
   %end%
%end%
.
.
.
```

%loopsep%

You use the %loopsep% tag to insert a specified character sequence between the results from a %loop% block.

Syntax

`%loopsep 'sepString'|[option]%`

Arguments

sepString is a string that you want to insert between each result.

Options

-nl specifies that you want to insert the platform specific newline sequence between the results.

Effect on Scope

None

Notes

%loopsep% does not insert *sepString* (nor the newline sequence) after the result produced by the last iteration of the loop.

Examples

The following example inserts a comma between each value produced by the loop.

```
.
.
.
%loop items%
   %loop -struct%
     %value%
     %loopsep ','%
   %endloop%
%endloop%
.
.
.
```

%nl%

You use the %nl% tag to generate a new line character in the code. The tag is useful when you want to preserve the ending of a line that ends in a tag. If you do not explicitly insert a %nl% tag on such lines, the server drops the new line character following that tag. (Note that this tag does not insert the HTML line break
code. It merely inserts a line break character, which is treated as white space.) The main reason you use this tag is to preserve the format of the underlying code in a DSP, which can make it easier to read during debugging.

Syntax

%nl%

ArgEffect on Scope

None

Examples

The following example shows how the %nl% tag is used to preserve the line endings on lines occupied by the three %value% tags. If the %nl% tag did not appear in this code, the three lines would be concatenated in the HTML document generated by the server.

```
.
.
.
<hr>
<p>Shipping Info:
%value carrier%%nl%
%value serviceLevel%%nl%
%value arrivalDate%%nl%</p>
<hr>
.
.
.
```

%rename%

You use the %rename% tag to move or copy a variable in the pipeline.

Syntax

`%rename` *SourceVar TargetVar* [*option option option…*]`%`

Arguments

SourceVar is the name of the variable that you want to move or copy. *SourceVar* can reside in any existing scope or Record.

TargetVar specifies the name of the variable to which you want *SourceVar* moved or copied. *TargetVar* must be in the current scope. If *TargetVar* does not exist, it will be created. If *TargetVar* already exists, it will be overwritten.

Options

You can use the following option with this tag.

Option	Description
`-copy`	Copies *SourceVar* to *TargetVar* instead of moving it to *TargetVar*. If you do not use –copy, *SourceVar* is deleted after its contents are copied to *TargetVar*.

Effect on Scope

Does not cause scope to switch to a different level, but, depending on how you use this tag, it may alter the contents of the current scope.

Examples

The following example renames the *state* variable in the current scope.

```
.
.
.
%scope buyerInfo%
%rename state ST%
      %invoke TMPL_ShipToBlock.html%</p>
%end%
.
.
.
```

The following example copies the variable named *oNum* from the previous scope into the current scope.

```
.
.
.
%invoke orders:getCustInfo%</p>
  %rename ../oNum oNum -copy%
  %invoke orders:getOrderDetails%</p>
.
.
.
```

%scope%

You use the %scope% tag to restrict a specified block of code to a particular Record in the pipeline. You can also use the %scope% tag to define a completely new Record and switch the scope to that record.

Syntax

%scope [*RecordName*] [*option option option…*]%
    Block of Code
%end%

Arguments

RecordName specifies the name of a Record within the current scope. If you do not specify *RecordName*, the param and rparam options will extend the current scope. If you specify a new *RecordName*, the scope switches to that Record.

Options

You can use any of the following options to add elements to the scope specified by *RecordName.* When you specify multiple options, separate them with spaces.

Important! If you set the value of an existing variable with these options, the value you specify will replace the variable's current value.

Note: For space reasons, the %scope% tag is shown on multiple lines in some of the examples below. Be aware that when you use the %scope% tag in a template or DSP, *the entire tag must appear on one line*.

Option	Description
`param(`*Name*`='`*Value*`')`	Defines a new String or String array with the name you specify in *Name* and assigns to it the string you specify in *Value*. If *Name* is a String, specify one *Value* and enclose it in single quotes. **Example** `%scope param(buyerClass='Gold')%` If *Name* is a String array: ■ Include a set of empty brackets at the end of the name to indicate that you are defining an array. ■ Enclose each element *Value* in single quotes. ■ Separate elements with commas. **Example** `%scope param(shipPoints[]='BWI','LAX',` `'ORD','MSP','DFW')%`

`rparam`(*Name*={*Key*='*Value*'; *Key*='*Value*';*Key*='*Value*'})	Defines a new Record or Record List with the name you specify in *Name,* and assigns to it, values that you provide in a list of *Key*=*Value* pairs.
	
	If *Name* is a Record:
	
	■ Enclose its list of elements in braces { }.
	■ Separate the elements with semi-colons.
	■ Enclose *Value* strings in single quotes.
	
	Example `%scope rparam(custServiceInfo=`
	`{csClass='Gold';csPhone='800-444-2300';`
	`csRep='Lauren Cheung'})%`
	
	If *Name* is a Record List:
	
	■ Enclose each Record in the list with braces { }
	■ Separate Records with vertical bars `|`.
	■ Separate elements within each Record with semi-colons.
	■ Enclose *Value* strings in single quotes.
	
	Example `%scope rparam(custServiceCtrs[]=`
	`{csName='Memphis';csPhone='800-444-2300';}|`
	`{csName='Troy';csPhone='800-444-3300';}|`
	`{csName='Austin';csPhone='800-444-4300';})%`

Effect on Scope Switches scope to the specified Record.

Notes The specified scope remains in effect until the next %scope% tag is encountered (which declares a new scope) or the next, non-nested %end% tag is encountered (which reverts to the prior scope).

For readability, you may use %endscope% instead of %end% to denote the end of the %scope% structure.

Examples

The following example sets the scope to the record named *buyerInfo*.

```
.
.
.
%scope buyerInfo%
   <p>Shipped To:<br>
      %value companyName%<br>
      %value streetAddr1%<br>
      %value streetAddr2%<br>
      %value city%, %value state%  %value postalCode%
%end%
.
.
.
```

The following example sets the scope to Record *buyerInfo* and then uses the `param` option to add variables called *buyerClass* and *shipPoint* to that Record.

```
.
.
.
%scope buyerInfo param(buyerClass='Gold') param(shipPoint='BWI Hub')%
   <p>Shipped To:<br>
      %value companyName%<br>
      %value streetAddr1%<br>
      %value streetAddr2%<br>
      %value city%, %value state%  %value postalCode%</p>
<hr>
      <p>Point of Departure: %value shipPoint%<br>
      Customer Class: %value buyerClass%</p>
%end%
.
.
.
```

The following example sets the scope to Record *buyerInfo* and then uses the `rparam` option to add a String variable named *req* to that scope before invoking a service.

```
.
.
.
<p>Open Orders:</p>
%scope buyerInfo rparam(req=openorders)%
%invoke orders:getOrderInfo%<br>
   %loop orders%
      Date: %value oDate%
      Number: <A HREF="showOrder.dsp&oNum=%value oNum%">%value oNum%</A>
      <br>
   %endloop%
<p>Click Order Number to View Details:</p>
%endscope%
.
.
.
```

%switch%

You use the %switch% tag to process one block from a series of predefined alternatives based on the value of a specified variable at run time.

Syntax

```
%switch Variable%
  %case 'SwitchValue'%
    Block of Code
  %case 'SwitchValue'%
    Block of Code
      .
      .
      .
 [%case%
    Default Block of Code]
%end%
```

Arguments

Variable specifies the name of the variable whose value will determine which case is processed at run time.

SwitchValue is a string that specifies the value that will cause the associated case to be processed at run time.

Effect on Scope

None

Notes

To select a case, *SwitchValue* must match the value of *Variable* **exactly**. *SwitchValue* is case-sensitive—'FedEx' does not match 'Fedex' or 'FEDEX'.

The server evaluates %case% tags in the order that they appear in your code. When it finds a “true” case, it processes the associated block of code and then exits the %switch%…%end% structure.

You can specify a default case using the %case% tag without a *SwitchValue*. This case will be processed if *Variable* does not exist or if none of the other cases are true. The default case must appear as the last %case% in the %switch%…%end% structure.

For readability, you can optionally use %endswitch% instead of %end% to denote the end of the %switch% structure.

Examples

The following example inserts a specified paragraph, depending on the value in *carrier*.

```
.
.
.
%switch carrier%
  %case 'FedEx'%
     <p>Shipped via Federal Express %value serviceLevel% %value trackNum%
  %case 'UPS'%
     <p>Shipped via UPS %value serviceLevel%
  %case 'Freight'%
     <p>Shipped via %transCompany%<br>
        FOB:  %value buyerInfo/streetAddr1%<br>
        %value buyerInfo/streetAddr2%<br>
        %value city%, %value state%  %postalCode%
%end%
</p>
.
.
.
```

The following example invokes different services based on the value of a variable named *action*.

```
<HTML>
<HEAD>
<title>Order Tracking System</title>
</HEAD>
<BODY BGCOLOR="#FFFFCC">
<H1>Order Tracking System</H1>
<HR>
%switch action%
  %case 'shipinfo'%
     %invoke orders:getShipInfo%
        .
        .
        .
  %case 'showorder'%
     %invoke orders:getOrderInfo%
        .
        .
        .
  %case 'showinvoice'%
     %invoke orders:getInvoices%
        .
        .
        .
%end%
<HR>
%include stdFooter.txt%
</BODY>
</HTML>
```

%sysvar%

You use the %sysvar% tag to insert the value of a special variable or server property into the document.

Syntax

`%sysvar` *SystemVariable*`%`

Arguments

SystemVariable is one of the following values, indicating which system variable or property you want to insert.

Value	Description
`host`	Inserts the name of the server that processed the DSP or template.
`date`	Inserts the current date. The date will appear in "Weekday Month Day HH:MM:SS Locale Year" format—e.g., `Fri Aug 12 04:15:30 Pacific 1999`.
`lastmod`	Inserts the date and time that this file was last modified. The date will appear in "Weekday Month Day HH:MM:SS Locale Year" format—e.g., `Fri Aug 12 04:15:30 Pacific 1999`.
`property(`*propertyName*`)`	Inserts the current value of the server property specified by *propertyName* (e.g., watt.server.port). See Appendix B, "Server Configuration Parameters" in the *SAP BC SOAP Programming Guide* for a list of server properties.

Effect on Scope

None

Examples

The following example inserts the name of the server processing the DSP or template and the date on which it was processed:

```
.
.
.
Response generated on %sysvar date% by host %sysvar host%
.
.
.
```

The following example includes the value of the “watt.server.port” property, which identifies the server’s main HTTP listening port:

```
.
.
.
<p>
%sysvar host% was listening on %sysvar property(watt.server.port)%
<p>
.
.
.
```

%value%

You use the %value% tag to insert the value of a specified variable in a document.

Syntax

`%value [`*Variable*`] [`*option option option…*`]%`

Arguments

Variable specifies the name of the variable whose value you want to insert into the code. You may specify the name of a variable that exists in the pipeline or the following keyword.

Variable	Description
`$key`	Inserts the name of the current key. You can use this keyword when looping over a Record (i.e., using the `%loop -struct%` tag) to retrieve the name of each key in the current scope or a specified Record.

When you specify *Variable*, the server retrieves the variable from the current scope. To select a variable outside the current scope, you use the following symbols to address it.

You would use...	To...
`/`*Variable*	Insert Variable from the initial scope.
`../`*Variable*	Insert Variable from the parent of the current scope.
RecName/Variable	Insert Variable from a Record named RecName.

If you don’t specify *Variable*, the current element within the current scope is assumed. (See example of this usage, below.)

Options

You can use any of the following options with this tag. To specify multiple options, separate the options with spaces.

Option	Description
`null='`*AnyString*`'`	Specifies the string that you want the server to insert when *Variable* is null. You specify the string in *AnyString*. **Example** `%value carrier null='No Carrier Assigned'%`
`empty='`*AnyString*`'`	Specifies the string that you want the server to insert when *Variable* contains an empty string. You specify the string in *AnyString*. **Example** `%value description empty='Description Not Found'%`
`index=`*IndexNum*	Specifies the index of an element that you want to insert. You can use this option to extract an element from an array variable. Specify an integer that represents the element's position in the array. (Arrays are zero-based.) **Example** `%value backItems index=1%`
`encode(`*Code*`)`	Encodes the contents of *Variable* prior to inserting it, where *Code* specifies the encoding system you want the server to apply to the string. Some of them should be used to prevent cross site scripting attacks (XSS).

Set Code to...	To encode the value using...
`xml`	XML encoding
`xmlattr`	XML encoding in attributes
`b64`	Base-64 encoding
`html`	HTML encoding (XSS)
`htmlattr`	HTML encoding in tag attributes (XSS)
`script`	Script encoding in JavaScript sections of HTML documents (XSS)
`url`	URL encoding for URL parts, mainly URL parameters (XSS)
`location`	URL encoding for complete URLs in attributes (XSS)

Option	Description
`decode(`*Code*`)`	Decodes the contents of *Variable* prior to inserting it, where *Code* specifies the way in which *Variable* is currently encoded.

Set Code to...	To decode the value using...
`b64`	Base-64 encoding

Option	Description	
	url	URL encoding

Effect on Scope None

Examples The following example invokes a service and then uses the %value% tag to insert the results of the service into the DSP.

```
.
.
.
%invoke orders:getOrderInfo%<br>
<p>%value buyerInfo/companyName%<br>
   %value buyerInfo/acctNum%
  <P>This shipment contains the following items</P>
  <TABLE WIDTH="90%" BORDER="1">
  <TR><TD>Number</TD><TD>Qty</TD><TD>Description</TD><TD>Status</TD></TR>
   %loop items%
     <TD>%value stockNum%</TD>
     <TD>%value qty%</TD>
     <TD>%value description%</TD>
     <TD>%value status%</TD>
     </TR>
   %endLOOP%
  </TABLE>
%endinvoke%
.
.
.
```

The following example dumps the contents of the current scope.

```
.
.
.
<p>
   %loop -struct%
      %value $key% %value%<br>
   %endloop%
<p>
.
.
.
```

The following example inserts the contents of *carrier*. If carrier is null, the string “UPS” is inserted. If carrier is empty, the string “UPS” is also inserted.

```
%value carrier null=UPS empty=UPS%
```

The following example inserts the contents of several variables into several places within HTML code and shows how to prevent cross site scripting attacks for the different cases.

```
.
.
.
<script>
%loop results%
addResult ("%value name encode (script)%", "%value result encode (script)%");
%endloop%
</script>
Find <A HREF="%value url encode (location)%">here</A> a central overview.
Find <A HREF="%http://www.starfleet.org/getShipDescription?name=%value name
encode (url)%">here</A> a local overview.
<select>
<option value="%value label encode(htmattr)%">%value artist encode
(html)%</option>
</select>
.
.
.
```

Index

Symbols

#$key variable 53, 55
$key variable 53, 54, 65
%case% tag 40, 62
%comment% tag 19, 27, 46
%else% tag 39, 47
%ifvar% tag 18, 26, 47
  usage and examples 39
%include% tag 18, 27, 49
  usage and examples 42
%invoke% tag 22, 26, 50
  usage and examples 28
%loop% tag 18, 26, 38, 52
  struct option 38
  usage and examples 37
%loopsep% tag 55
%nl% tag 18, 56
%onerror% tag 36, 50
%rename% tag 19, 26, 36, 56
%scope% tag 18, 26, 36, 58
%switch% tag 18, 26, 62
  default case specification 42
  usage and examples 40
%sysvar% tag 64
%value% tag 18, 26, 65

Numerics

1 6

A

ACLs and DSPs 24
advantages of using DSPs 23
API
  related documentation 8
Applying Output Templates Arbitrarily 17
Arbitrarily Processing DSP Tags 43
array values, looping over 37
assigning a template to a service 13

B

begin…end constructs 27
Building Conditional Blocks With the %ifvar% Tag 39
Building Conditional Blocks with the %switch% Tag 40

C

case tag 40, 62
Catching Errors 36
closing the scope 30
comment tag 19, 27, 46
Conditionally Executing Blocks of Code 39
Content-Type header
  for DSPs 22
  text/html 15
  text/vnd.wap.wml 16
  text/x-hdml 17
  text/xml 15
conventions used in this document 6, 7
creating
  DSPs 23
  HDML-based output templates 17
  HTML-based output templates 15
  output templates 17
  WML-based output templates 16
  XML-based output templates 15
current scope, dumping 38

D

directory location of DSPs 24
directory location of output templates 13
documentation
  conventions used 6
  printing 9
  related manuals 8
  viewing 9
DSPs
  #$key variable 53
  $key variable 54, 65

ACL settings 24
advantages of using 23
begin…end constructs 27
catching errors 36
comment tag 27, 46
conditional code 39
creating 23
definition of 22
dumping the current scope 38
end tag usage 27
hyperlinks to 26
if…then…else construct 39
ifvar tag 26, 39, 47
include tag 27, 42, 49
invoke tag 26, 28, 50
location of 24
loop tag 26, 37, 38, 52
looping over a Record 38
looping over an array 37
loopsep tag 55
nl tag 56
onerror tag 36
overview of 22
passing parameters between DSPs 35
passing parameters between services in a DSP 36
passing parameters to a service 33
publishing 24
rename tag 26, 56
requesting from a browser 25
scope tag 26, 58
scope, definition of 29
security checking 24
simple example of 22
summary of tags 26
switch tag 26, 40, 62
sysvar tag 64
URL format 25
value tag 26, 65
when to use 23
dumping the current scope 38
dynamic server pages. *See* DSPs

E

else tag 39, 47
email clients and output templates 15, 16
end tag, usage 27
ending the scope 30
error handling in DSPs 36
Extracting Results from a Record 38
Extracting Results from an Array Variable 37

F

file date, inserting 64
files, inserting in a DSP 42
FTP clients and output templates 15, 16

G

Guidelines for Creating HDML Output Templates 17
Guidelines for Creating WML Output Templates 16
Guidelines for Using HTML-based Output Templates with HTTP Clients 15
Guidelines for Using XML-based Output Templates with HTTP Clients 15

H

Handheld Device Markup Language 16
HDML 16
HDML-based templates 17
host name, inserting 64
How Do You Assign an Output Template to a Service? 13
HTML-based templates 15
HTTP clients and output templates 15
hyperlinks to DSPs 26

I

ifvar tag 18, 26, 47
usage and examples 39
include tag 18, 27, 49
usage and examples 42
input, passing to a service 33, 35, 36
inserting file date 64
inserting files in a DSP 42
inserting host name 64
inserting server property value 64

inserting timestamp 64
invoke tag 26, 50
 usage and examples 28
invoking services from a DSP 28

L

location of DSP files 24
location of template files 13
loop tag 18, 26, 52
 dumping the current scope 38
 struct option 38
 usage and examples 37
looping over a Record 38
looping over an array 37
looping over elements in current scope 38
loopsep tag 55

N

new line, preserving in code 56
nl tag 18, 56

O

onerror tag 36, 50
output templates 12
 applying arbitrarily 17
 assigning to a service 13
 comment tag 19, 46
 creating 17
 definition of 12
 for browser-based clients 15
 for e-mail clients 16
 for FTP clients 16
 for HTTP clients 15
 for non-browser clients 16
 for wireless devices 16
 guidelines for HDML templates 17
 guidelines for HTML templates 15
 guidelines for WML templates 16
 guidelines for XML templates 15
 ifvar tag 18, 47
 include tag 18, 49
 location of 13
 loop tag 18, 52
 loopsep tag 55
 nl tag 18, 56
 rename tag 19, 56
 scope tag 18, 58
 summary of tags 17
 switch tag 18, 62
 sysvar tag 64
 value tag 18, 65
 when used 15
overview of DSPs 22
overview of output templates 12

P

parameters, passing to a DSP 33, 35
passing parameters between DSPs 35
passing parameters between services in a DSP 36
passing parameters to a service in a DSP 33
PDF, viewing 9
port-level security and DSPs 24
printing
 this guide 9
publishing DSPs 24

R

Record elements, looping over 38
Referencing Variables In and Out of Scope 32
rename tag 19, 26, 36, 56
report services 43
requesting a DSP 25

S

scope
 closing 30
 definition of 29
 effect on pipeline 30
 examining current values of 38
 referencing variables outside of 32
scope tag 18, 26, 36, 58
Securing a DSP 24
server property value, inserting 64
services, invoking in a DSP 22, 28

services, passing input to 33, 36
services, security issues 24
Settings tab 13
SMTP clients and output templates 15, 16
Specifying a Default Case 42
struct option 38, 53
summary of tags
  for DSPs 26
  for output templates 17
switch tag 18, 26, 62
  default case specification 42
  usage and examples 40
sysvar tag 64

T

tags
  begin…end constructs 27
  for DSPs 26
  for output templates 17
  syntax 45
templates. *See* output templates
templates directory 13
Testing for a Particular Value 40
timestamp, inserting 64
typographical conventions 6, 7

U

URL format for a DSP 25
Using Output Templates to Return Output to HTTP Clients 15
Using Output Templates to Return Output to SMTP and FTP Clients 16
Using Output Templates to Return Output to Wireless Devices 16
Using the %loop% Tag to Examine the Current Scope 38
Using the DSP Tags 26

V

value tag 18, 26, 65
variables, referencing 32
viewing
  documentation in PDF format 9
viewing this document in PDF format 9

W

What Are the Advantages of Using DSPs? 23
What Does a DSP Look Like? 22
What Does an Output Template Look Like? 12
What Is a Dynamic Server Page? 22
What Is an Output Template? 12
What Is Scope? 29
When Do You Use DSPs? 23
When Does the Server Use an Output Template? 15
Why Does Scope Matter? 30
wireless devices and output templates 16
Wireless Markup Language 16
WML 16
WML-based templates 16

X

XML-based templates 15