

Internet Application Development With Flow Files: Reference



HELP.BCFESITFLOW

Release 4.6C



Copyright

© Copyright 2001 SAP AG. All rights reserved.

No part of this publication 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[®], NT[®], EXCEL[®], Word[®], PowerPoint[®] and SQL Server[®] are registered trademarks of Microsoft Corporation.

IBM[®], DB2[®], OS/2[®], DB2/6000[®], Parallel Sysplex[®], MVS/ESA[®], RS/6000[®], AIX[®], S/390[®], AS/400[®], OS/390[®], and OS/400[®] are registered trademarks of IBM Corporation.

ORACLE[®] is a registered trademark of ORACLE Corporation.

INFORMIX[®]-OnLine for SAP and Informix[®] Dynamic Server[™] are registered trademarks of Informix Software Incorporated.

UNIX[®], X/Open[®], OSF/1[®], and Motif[®] are registered trademarks of the Open Group.






HTML, DHTML, XML, XHTML are trademarks or registered trademarks of W3C[®], World Wide Web Consortium, Massachusetts Institute of Technology.

JAVA[®] is a registered trademark of Sun Microsystems, Inc.

JAVASCRIPT[®] is a registered trademark of Sun Microsystems, Inc., used under license for technology invented and implemented by Netscape.

SAP, SAP Logo, R/2, RIVA, R/3, ABAP, SAP ArchiveLink, SAP Business Workflow, WebFlow, SAP EarlyWatch, BAPI, SAPHIRE, Management Cockpit, mySAP.com Logo and mySAP.com are trademarks or registered trademarks of SAP AG in Germany and in several other countries all over the world. All other products mentioned are trademarks or registered trademarks of their respective companies.

Icons

Icon	Meaning
	Caution
	Example
	Note
	Recommendation
	Syntax

Contents

Internet Application Development With Flow Files: Reference	5
Request/Response Cycle	8
Module Provider Interface	10
Flow Logic.....	13
Flow Logic Syntax.....	16
FLOW Element.....	17
STATE Element.....	18
MODULE Element.....	19
PERSISTENT Element.....	21
CONVERTER Element.....	22
INPUTMAPPING Element.....	23
OUTPUTMAPPING Element.....	24
FILEMAPPING Element.....	25
RESULT Element	26
EXPR Element.....	27
EXCEPTION Element.....	28
DEFAULT Element.....	30
EVENT Element	31
Module Provider Connection Types	33
Module Call Result Caching	36
Flow File Application Components	38
Flow File Applications: Example Scenarios	40
Flow File Example 1: Display Development Classes.....	41
Flow File Example 2: Online Store	43

Internet Application Development With Flow Files: Reference

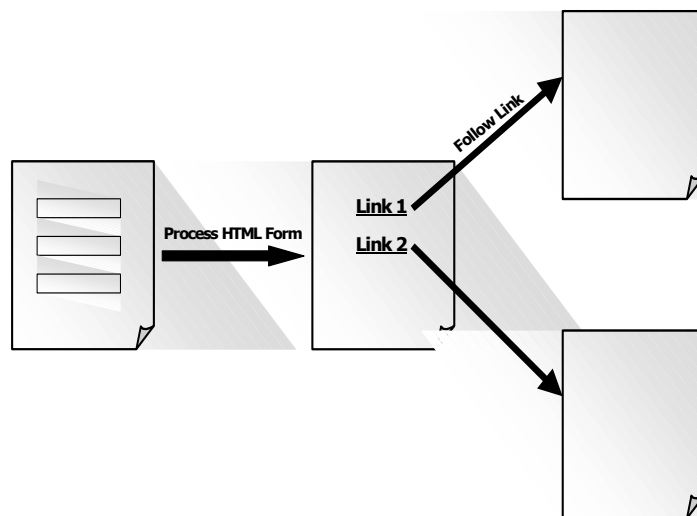
Purpose

This documentation describes an implementation model for developing Internet applications driven by the Internet Transaction Server (ITS).

This model allows you to develop Internet applications that consist of linked HTML pages, which you can populate with data retrieved from the R/3 System (or any other external system). The pages can offer a range of application functions, and are generated by following hyperlinks or processing HTML forms. The dialog flow is determined on the client side by the user, who can navigate freely between pages.

Since the dialog flow is not fixed in advance, much depends on what the user decides to do. This contrasts with the dialog flow in other business scenarios, where the business application can put restrictions on how users can navigate.

The following graphic illustrates the basic concept:



This documentation describes how to develop applications that use flow files where the business logic is implemented in modules called from the R/3 System.

In future releases, it will be possible to define module calls from any external system.

Internet Application Development With Flow Files: Reference

Implementation Considerations

You should consider using this implementation model for applications that offer many application functions on one page, and the dialog flow is not fixed in advance.

Such applications have simple point-and-click user interfaces, limited manual data input, and reduced data formatting requirements. They are often used in e-commerce scenarios.

Integration

To develop applications that use flow files where the business logic is implemented in module calls from the R/3 System, you need to install the following components:

- The ITS
 - The ITS forms the interface between the R/3 System and the Internet.
- The SAP@Web Studio
 - The SAP@Web Studio is a PC tool for implementing services, which include all the files required by the ITS to drive applications.
- The R/3 System

Features

Like all other implementation models for developing Internet applications driven by the ITS, this model allows you to develop applications that send documents back to the Web browser client in HTML format, since this format can be handled by all major Web browsers.

Like all other implementation models, there is a clear separation between business logic and presentation aspects. In this case, defining the dialog flow is also a separate task.

- You implement a set of modules that comprise the business logic in the R/3 System (or other external system).
 - If you are implementing the business logic in the R/3 System, you create Business APIs (BAPIs) or standard remote-enabled function modules (RFCs) with the Function Builder in the ABAP Workbench.
- You implement the presentation and the dialog flow in the SAP@Web Studio.
 - To do this, you need to create an ITS service, which contains all the files required to implement and run the application.
 - The presentation determines the look and feel of each application.
 - You design the presentation by creating a set of HTML^{Business} templates.
 - The dialog flow determines which template is displayed when, depending on what the user decides to do next.
 - You implement the dialog flow by defining flow logic in flow files. Flow logic operates like a state machine, because it defines the sequence in which modules are called based on events and exceptions.
 - There is one flow file for each HTML^{Business} template that requires a dialog flow definition.

Using flow files gives you more flexibility when developing Internet applications, because you can define both the presentation and the dialog flow independently of the business logic.

The flow file implementation model is suitable for developing applications that offer at least some of the following features:

- Several application functions on one page
- Point-and-click interfaces
- Limited data entry checking and formatting
- User-defined dialog flow

This documentation is a reference manual, which includes a list of flow logic syntax elements and their usage.

For a step-by-step introduction to developing Internet applications with flow files, with the help of an example application, see [Internet Application Development With Flow Files: Tutorial \[Ext.\]](#).

Constraints

To implement Internet applications with flow files, you should have:

- ITS Release 4.6C
- SAP@Web Studio Release 4.6C
- R/3 Release 4.6A or higher



Some Business APIs (BAPIs) are not available in R/3 releases prior to 4.6B. For this reason, SAP recommends that you use R/3 Release 4.6B or higher, if you intend to develop flow file applications that use BAPIs to define the business logic.

In future, it will be possible to develop applications that use flow files in earlier R/3 releases.

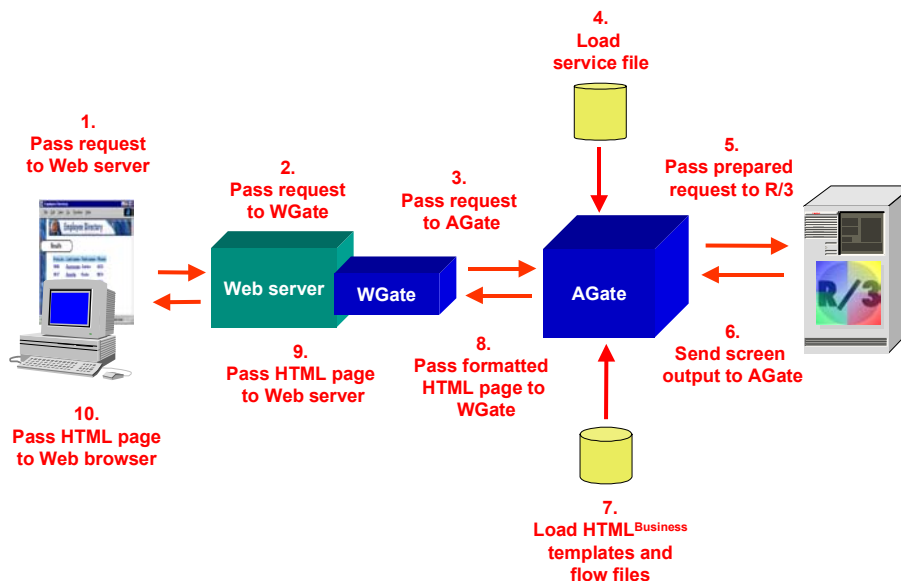
Request/Response Cycle

Request/Response Cycle

The process used by the Internet Transaction Server (ITS) to drive applications is similar in all implementation models.

The graphic below shows a single request/response cycle from the Web browser client to the R/3 application server for an application that is driven by the ITS and uses flow files.

Applications Using Flow Files: Single Request/Response Cycle



When the user starts an application that uses flow files and performs an action in the Web browser by clicking on a hyperlink or entering data in an HTML form for processing, a single request/response cycle includes the following steps:

1. The Web browser passes the request as a set of name/value pairs, either specified in the hyperlink or entered in the HTML form, to the Web server via HTTP.
This set of name/value pairs is known as the request context.
2. The Web server passes the request to WGate (the Web gateway).
WGate is the Web server extension that links the Web server to the ITS.
3. WGate passes the request to AGate (the application gateway) via TCP/IP.
AGate is the core processing component of the ITS, and contains configuration data about how a request can be fulfilled.
4. AGate loads the appropriate service file for the application and uses the information stored there to establish a connection to the R/3 System.
The AGate itself does not execute any business logic - this is done in the R/3 System.

Request/Response Cycle

5. AGate passes the request as a set of parameters to one or more modules in the R/3 System via the Module Provider Interface. Parameters are passed to the modules by finding fields in the request context with identical names.

The modules used can be Business APIs (BAPIs) or standard remote-enabled function modules.

6. The R/3 System executes the called modules and returns the results to AGate.
7. AGate merges the set of name/value pairs into the request context and passes this to an HTML^{Business} template, which is the HTML page that displays the result of the request in the user's Web browser.

An HTML^{Business} template is an HTML page that allows you to merge data retrieved from the R/3 System.

HTML^{Business} templates allow you to design the presentation separately from the business logic and the dialog flow.

- The business logic is implemented in the R/3 System with BAPIs or standard remote-enabled function modules.
 - The presentation is implemented in the SAP@Web Studio as an HTML^{Business} template, with the dialog flow defined as flow logic in attached flow files.
8. AGate passes the formatted HTML page to WGate.
 9. WGate passes the HTML page to the Web server.
 10. The Web server passes the HTML page to the Web browser, which displays it to the user.



After each request/response cycle, the ITS does not retain any data.

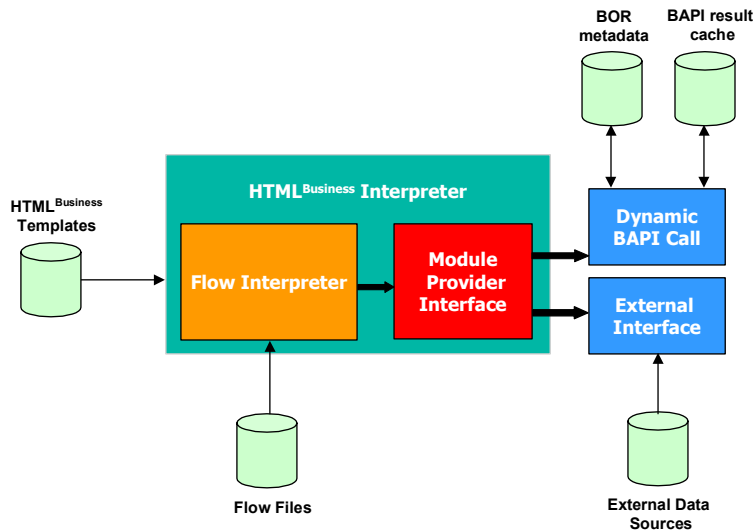
AGate communicates with the backend systems via a pluggable interface called the XGateway, which is implemented as a DLL.

The above graphic shows how the AGate component of the ITS interacts with the R/3 System to satisfy a user request from a Web browser, but the XGateway also supports modules called from systems other than R/3. This means that you can process Web browser requests by calling modules from R/3, any other external system, or a combination of modules from more than one system.

Module Provider Interface

Module Provider Interface

To support new types of module calls, a module provider has to be implemented for each type of module. In the case of the R/3 System, supported module types include Business APIs (BAPIs) and standard remote-enabled function modules.



Example Application

A simple application that searches a database of employees in the company could consist of the following components:

- An HTML^{Business} template for the presentation.
- A flow file to define the dialog flow, depending on what action the user takes.
- A remote-enabled function module to execute the business logic.

HTML^{Business} Template

The HTML^{Business} template `SearchEmployee.html` defines the application's look and feel:

```

<HTML>
<HEAD>
  <TITLE> Employee Search </TITLE>
</HEAD>
<BODY>
<FORM method=post action=`wgateURL()` `>
<TABLE>
  <td>Name <input type=text name="empname "value="`empname`"></td>
</TABLE>
  
```

```

<TABLE>
  `if ( results-name.dim > 0 )`
  `repeat with j from 1 to results-name.dim`
    <tr>
      <td> `results-name[j]` </td>
      <td> `results-address[j]` </td>
      <td> `results-phonenum[j]` </td>
    </tr>
  `end`
</TABLE>
<input type = "submit" name = "~event" value = "search">
</FORM>
</BODY>
</HTML>

```

Flow File

The flow file `SearchEmployee.flow` contains the flow logic that defines the dialog flow:

```

<FLOW>
  <STATE NAME="present" >
    <MODULE NAME="EMPLOYEE_GET" type= "RFC" STATEFUL="0" >
      <EXCEPTION next_template="add_record"
name="NO_RECORDS_FOUND">
        </EXCEPTION>
      </MODULE>
    </STATE>
    <EVENT name = "search" next_state = "present">
    </EVENT>
  </FLOW>

```

Remote-Enabled Function Module

The remote-enabled function module `EMPLOYEE_GET` in R/3 defines the business logic and has the following interface:

Parameter	Parameter Name
Import	<code>empname</code>
Export	-
Tables	<code>results</code> <code>results-name</code> <code>results-address</code> <code>results-phonenum</code>
Exceptions	<code>NO_RECORDS_FOUND</code>

Processing Steps

When the user starts this application in a Web browser, enters an employee name, and clicks *Search* on the `SearchEmployee.html` form, the following sequence of events occurs:

1. The WGate component of the Web server sends the request to AGate.

Module Provider Interface

- AGate processes the request and stores it as a set of name/value pairs in the request context data structure.

The request context data structure is the main interface between the AGate and the module provider. The context object provides methods for retrieving and merging values into the context.

In this example, the request context contains the following name/value pairs:

- **empname** “John“
- **~event** “search“

- The flow interpreter executes the flow file **SearchEmployee.flow**.

Since the parameter **~event** is set to “search“, it calls the Module Provider Interface, and passes the name of the module, the module type, and the request context.

- The Module Provider Interface determines which module provider to call based on the module type.

At present, the module provider is implemented for the types “RFC“ and “BAPI“ in the R/3 System. Depending on the module type, the appropriate module provider DLL is loaded. Execution of the module and other related operations are performed through the Module Provider Interface. In the above example, the module type is “RFC“.

- Before calling the module, the module provider determines the module’s interface and populates the necessary parameters.

In the above example, the module provider retrieves the value of **empname** from the request context to populate the import parameter **empname**.

- The results of the call (that is, the export parameters and the tables) are merged into the request context.

If the employee is found and the result is merged into the request context, the name/value pairs may look like these:

Parameter Name	Example Value
empname	“John“
~event	“search“
results-name	“John“
results-address	“San Francisco“
results-phonenum	“415-111-9111“

In the above example, only one module is called, but you can call several modules one after the other. These modules can also be of different types. The output from one module call then becomes the input for the next module call.

- If an exception is raised, the exception **NO_RECORDS_FOUND** has to be merged into the request context under the name **~ModuleException**.
- The request context values are merged into the HTML^{Business} template and sent to the Web browser.

Flow Logic

Flow logic defines the dialog flow of an application by specifying logical transitions between application states in flow files, which are associated with HTML^{Business} templates.

An HTML^{Business} template may or may not have an associated flow file. This depends whether a dialog flow definition is required to react to different actions taken by the user. If a flow file exists, the naming convention is:

`<template name>.flow`

where `template name` is the name of the HTML^{Business} template and `.flow` is the file extension.



The flow file `MyTemplate.flow` is associated with the HTML^{Business} template `MyTemplate.html`.

A typical flow file consists of a set of events and states. Events define the entry points to different logical state(s), or how to load a different template directly.

- States

A state can contain one or more operations (module calls). Depending on the result(s) of the operation(s), a transition to another state or template occurs. Transitions are ways to leave a current state. A flow file can consist of one or more states.

A state is defined by the **STATE** element. The **STATE** element contains one or more **MODULE** declarations, and one or more **PERSISTENT** elements:

- **MODULE**

The **MODULE** element describes a module. Each module must have a name and a type that corresponds to the name of an API associated with the Module Provider Interface.

In the case of the Module Provider Interface to the Business Object Repository (BOR) in the R/3 System, the module name is the name of a Business API (BAPI).

- **PERSISTENT**

The **PERSISTENT** element describes a parameter whose value has to be persistent (retained) throughout a user session.

Due to the nature of remote-enabled function modules, parameter values are refreshed during every request/response cycle, but this element helps keep track of certain key variables and retains their values during an entire user session. The ITS server allocates a small (35K) session context to hold these persistent variables.

- Events

An event triggers state processing. You raise an event by specifying the parameter `~event` either in the URL of an hypertext link or as part of the HTML form data.

An event is defined by the **EVENT** element.

For full details, see [Flow Logic Syntax \[Page 16\]](#).

Flow Logic

The modules called in the flow logic operate like a state machine with several nodes. Each node can contain one or more modules that are called sequentially. After each module call, the results are merged into the request context. This means that module calls can be chained in the sense that the result of one module can serve as an input parameter for the next module call.

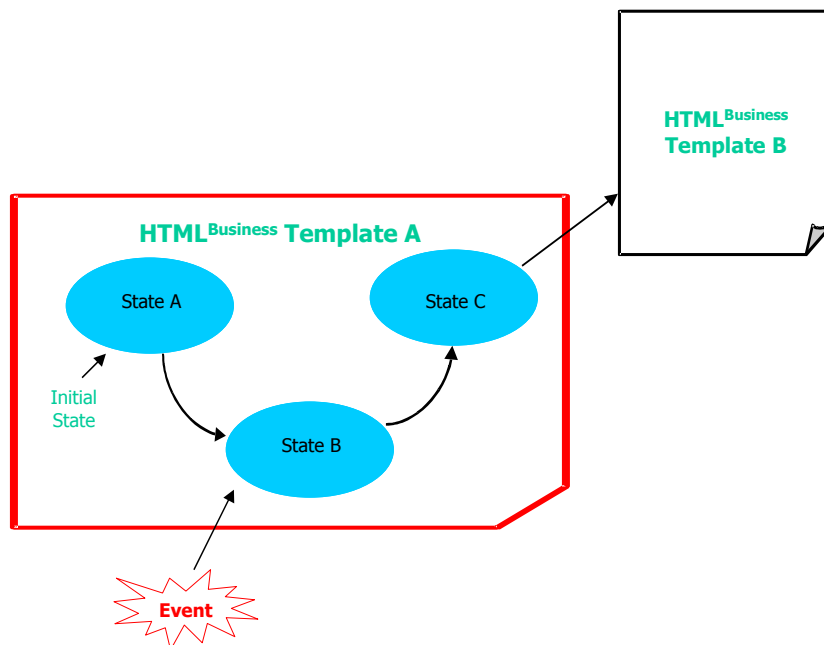
Calling several modules sequentially allows for complex chaining, but it makes sense to keep the flow logic of flow file applications simple. If, for example, you are implementing business logic with Business APIs (BAPIs) or remote-enabled function modules in the R/3 System, you should implement chains that are more complex than a simple transfer of values there, because ABAP is more suitable for implementing complex chaining logic.

The start of a state machine – the flow logic – is determined by an event, which is raised by specifying a parameter `~event` in the URL of a hypertext link or as part of an HTML form.

After each module call, you can apply a set of tests. For example, you can test whether a module raises an exception or returns a particular value. If the test is positive, either a state transition occurs, or you can load a different template. In the latter case, the flow logic of the target template starts either with an `onload` or `ontouch` event type, which specifies the initial state.

The following graphic illustrates the concept:

Flow Logic: HTML^{Business} Template Flow



This graphic shows the basic dialog flow for HTML^{Business} template **A**, which has three states: **A**, **B**, and **C**. State **A** is the initial state.

When template **A** is loaded, execution of the dialog flow logic always starts at state **A**. When all the module calls associated with this state have been executed, template **A** is populated with the data in the request context, and the resulting HTML page is displayed in the user's Web browser.

When the user clicks on a hyperlink, this raises an event. In this case, the flow logic determines that execution should start at state **B**. However, due to an exception in one of the modules, the

flow logic actually transitions to state **C** where the next template **B** is finally loaded. The ITS then loads the flow logic of template **B** and starts the execution at the initial state of template **B**.

Flow Logic Syntax

Flow Logic Syntax

The flow logic that defines the dialog flow of flow file applications uses a subset of Extensible Markup Language (XML) elements.

This subset includes the following elements:

XML Element	Description
FLOW [Page 17]	Defines the start of the flow logic.
STATE [Page 18]	Defines a state.
MODULE [Page 19]	Defines a module.
PERSISTENT [Page 21]	Defines a persistent parameter.
CONVERTER [Page 22]	Defines a data download from the R/3 System (or other external system) and a conversion to a specified format.
INPUTMAPPING [Page 23]	Defines the mapping of input parameters from the application to the R/3 System (or other external system).
OUTPUTMAPPING [Page 24]	Defines the mapping of output parameters from the R/3 System (or other external system) to the application.
FILEMAPPING [Page 25]	Defines a data upload from the application to the R/3 System (or other external system).
RESULT [Page 26]	Defines a module test result.
EXPR [Page 27]	Defines a valid HTML ^{Business} expression.
EXCEPTION [Page 28]	Defines a module test exception.
DEFAULT [Page 30]	Defines the default behavior.
EVENT [Page 31]	Defines a test for a specified event.



The number of XML elements used in flow logic is intentionally restricted, because complex flow logic should be implemented elsewhere. The Internet Transaction Server (ITS) merely triggers the modules called.

In the case of applications that call modules from the R/3 System, complex chaining logic should be implemented in ABAP.

FLOW Element

Description

The **FLOW** element defines the start of the flow logic.

The **FLOW** element can contain one or more **STATE** declarations followed by one or more **EVENT** declarations.

The **FLOW** element is a required element.

Syntax

```
<!ELEMENT FLOW ( STATE+ , EVENT+ )>
```



```
<flow>  
...  
</flow>
```

STATE Element

STATE Element

Description

The **STATE** element defines a logical state.

A flow file can consist of one or more states, but each state must have a unique name.

A state can contain one or more **MODULE** declarations, and one or more **PERSISTENT** elements.

The result(s) of the operation(s) is a transition to another state or template.

Syntax

```
<!ELEMENT      STATE ( PERSISTENT?, MODULE+ )>
<!ATTLIST     STATE
              NAME          CDATA          #REQUIRED>
```



```
<state name="mystate">
  <module name= "mymodule1" type="RFC">
  </module>
  <module name= "mymodule2" type="RFC">
  </module>
</state>
```

Attributes of the **STATE** element:

Attribute	Description	Required
NAME	State name.	Yes

MODULE Element

Description

The **MODULE** element defines a module.

Each module must have a name and a type that corresponds to the name of an API associated with the Module Provider Interface.

In the case of the Module Provider Interface to the Business Object Repository (BOR) in the R/3 System, the module name is the name of a Business API (BAPI).

A **MODULE** element can contain:

- One or more **RESULT** or **EXCEPTION** elements
- Zero or one **DEFAULT** elements
- Zero or more **INPUTMAPPING** and/or **OUTPUTMAPPING** elements
- Zero or more **CONVERTER** elements
- Zero or more **PERSISTENT** elements

Syntax

```
<!ATTLIST    MODULE ( (RESULT|EXCEPTION)* ,DEFAULT?
              INPUTMAPPING?, OUTPUTMAPPING?,
              PERSISTENT?) >
<!ATTLIST    MODULE
              NAME          CDATA          #REQUIRED
              STATEFUL     (1|0) "0"      #IMPLIED>
```



```
<module name= "mymodule" type="RFC">
</module>
```

Attributes of the **MODULE** element:

Attribute	Description	Required
NAME	Module name.	Yes
TYPE	Module type.	Yes

MODULE Element

STATEFUL	Specifies whether a module call is stateful or stateless. <ul style="list-style-type: none">• If the value of STATEFUL is 1, the call is stateful. RFC connection is maintained throughout the entire user session.• If the value of STATEFUL is 0 (default), the call is stateless. RFC connection is closed after each module call.	No
POOLED	Specifies a pooled connection.	No
CACHE	Specifies a cached connection.	No

PERSISTENT Element

Description

The **PERSISTENT** element defines a parameter value that needs to be retained during an entire user session.

Due to the nature of remote function calls (RFCs), parameter values are refreshed during each request/response cycle, so the **PERSISTENT** element helps keep track of the values of certain key variables during a user session. A small session context is allocated to hold persistent variables.

Syntax

```
<!ELEMENT      PERSISTENT      EMPTY>
<!ATTLIST     PERSISTENT
              NAME              CDATA      #REQUIRED>
```



```
<module name="mymodule" type="RFC">
...
<persistent name = "bk_full"/>
...
</module>
>
```

The syntax of the **PERSISTENT** element defined within a **MODULE** element is similar to its description within a **STATE**. In this example, the persistent parameters have module level scope.

Attributes of the **PERSISTENT** element:

Attribute	Description	Required
NAME	Persistent name	Yes



Do not use the **PERSISTENT** as a convenient way to retain all parameter values.

CONVERTER Element

CONVERTER Element

Description

The **CONVERTER** element defines a data download from the R/3 System (or other external system) and a conversion to a specified format.

The **CONVERTER** element requires a type attribute and a list of attribute/value pairs that provide additional information for the conversion.

Syntax



```
<converter type ="load" input = "data-rawdata" format = "GIF">
```

Here, the input type **data-rawdata** specifies an R/3 table called **data**, which has a column called **rawdata** containing the data to be converted to GIF format.

At present, support for the conversion type is confined to **LOAD**, which specifies a direct download of data from the R/3 System for display in the Web browser, without conversion to another format.

The second parameter of the element is a list of attribute pairs that provides additional information for the **CONVERTER** element.

For type **LOAD**, the required attribute-value pairs are: **INPUT** and **FORMAT**.

- The **INPUT** attribute specifies a table of data to be retrieved from the R/3 System.
 - This attribute value may or may not include a suffix appended after a hyphen.
 - If a suffix is provided, the **CONVERTER** element traverses the specified table column(s).
 - If no suffix is provided, the **CONVERTER** element traverses all table columns.
- The **FORMAT** attribute specifies the MIME type of the data contents.

The **FORMAT** attribute supports the formats **.ppt**, **.gif** and **.jpeg**.



In future releases, the formats **.xls** and **.doc** will also be supported.

INPUTMAPPING Element

Description

The **INPUTMAPPING** element defines the mapping of input parameters from the application to the R/3 System (or other external system).

Syntax

```
<!ELEMENT INPUTMAPPING EMPTY >
<!ATTLIST INPUTMAPPING
          SOURCE CDATA
          #REQUIRED
          TARGET CDATA
          #REQUIRED>
```



```
<module name="mymodule" type="RFC">
...
<inputmapping source="sourceID" target="targetID">
</inputmapping>
...
</module>
```

Attributes of the **INPUTMAPPING** element:

Attribute	Description	Required
SOURCE	Parameter name defined in the application.	Yes
TARGET	Parameter name of the called module.	Yes

The value in the **SOURCE** parameter is copied to the **TARGET** parameter.

OUTPUTMAPPING Element

OUTPUTMAPPING Element

Description

The **OUTPUTMAPPING** element defines the mapping of output parameters from the R/3 System (or other external system) to the application.

Syntax

```
<!ELEMENT      OUTPUTMAPPING EMPTY >
<!ATTLIST     OUTPUTMAPPING
              SOURCE          CDATA
              #REQUIRED
              TARGET          CDATA
              #REQUIRED>
```



```
<module name="mymodule" type="RFC">
...
<outputmapping source="sourceID" target="targetID">
</outputmapping>
...
</module>
```

Attributes of the **OUTPUTMAPPING** element:

Attribute	Description	Required
SOURCE	Parameter name of the called module.	Yes
TARGET	Parameter name defined in the application.	Yes

The value in the **SOURCE** parameter is copied to the **TARGET** parameter.

FILEMAPPING Element

Description

The **FILEMAPPING** element defines a data upload from the application to the R/3 System (or other external system).

Syntax



```
<module name="mymodule" type="RFC">
...
<filemapping source="myfile" target="table1">
</filemapping>
...
</module>
```

Attributes of the **FILEMAPPING** element:

Attribute	Description	Required
SOURCE	Name of file to be loaded.	Yes
TARGET	Name of R/3 internal table where file data is to be placed. It is important that this internal table has only one column representing the raw data.	Yes

The value in the **SOURCE** parameter is copied to the **TARGET** parameter.

RESULT Element

RESULT Element

Description

The **RESULT** element defines a module test result.

Syntax

```
<!ELEMENT      RESULT ( EXPR ) >
<!ATTLIST     RESULT
              NEXT_STATE      CDATA
              #IMPLIED
              NEXT_TEMPLATE    CDATA
              #IMPLIED>
```



```
<module name="mymodule" type="RFC">
...
<result next_state ="read">
<expr> item-no == "1" </expr>
<exception name = "InputDataFormatError"
next_template = "checkinput"/>
...
</module>
```

If the test passes, either the next state or the next HTML^{Business} template is processed.

The [EXPR \[Page 27\]](#) element defines a valid HTML^{Business} expression that evaluates the return parameter of a module call.

The [EXCEPTION \[Page 28\]](#) element defines a module test exception.

Attributes of the **RESULT** element:

Attribute	Description	Required
NEXT_STATE	Specifies the next state.	If NEXT_TEMPLATE is not specified.
NEXT_TEMPLATE	Specifies the next HTML ^{Business} template without the <code>.html</code> extension. You can also specify the service and theme. For example: NEXT_TEMPLATE = " <code><service>/<theme>/<template></code> "	If NEXT_STATE is not specified.

Either **NEXT_STATE** or **NEXT_TEMPLATE** is sufficient.

EXPR Element

Description

The **EXPR** element defines a valid HTML^{Business} expression that evaluates the return parameter of a module.

Syntax

```
<!ELEMENT      EXPR EMPTY >
```



```
<module name="mymodule" type="RFC">
...
<result next_state ="read">
<expr> item-no == "1" </expr>
<exception name = "InputDataFormatError"
next_template = "checkinput"/>
...
</module>
```

EXCEPTION Element

EXCEPTION Element

Description

The **EXCEPTION** element defines a module test exception.

Syntax

```
<!ELEMENT      EXCEPTION      EMPTY >
<!ATTLIST     EXCEPTION
              NAME              CDATA
              #REQUIRED
              NEXT_STATE        CDATA #IMPLIED
              NEXT_TEMPLATE     CDATA
              #IMPLIED>
```



```
<module name="mymodule" type="RFC">
...
<result next_state = "read">
<expr> item-no == "1" </expr>
<exception name = "InputDataFormatError"
next_template = "checkinput"/>
...
</module>
```

If the test passes, either the next state or the next HTML^{Business} template is processed.



If an input data format error occurs, the ITS generates an exception called **InputdataFormatError** instead of stopping the template processing.

You must catch this exception yourself. You can either generate a pop-up window or use a Javascript function to catch the error message in the context **borErrorMsg**. If the exception is not caught, the dialog flow processing terminates and an error message is displayed in the Web browser.

Attributes of the **EXCEPTION** element:

Attribute	Description	Required
NAME	Name of a module test exception.	Yes.
NEXT_STATE	Specifies the next state.	If NEXT_TEMPLATE is not specified.

EXCEPTION Element

<p>NEXT_TEMPLATE</p>	<p>Specifies the next HTML^{Business} template without the .html extension.</p> <p>You can also specify the service and theme.</p> <p>For example:</p> <pre>NEXT_TEMPLATE = "<service>/<theme>/<template></pre>	<p>If NEXT_STATE is not specified.</p>
-----------------------------	---	---

Either **NEXT_STATE** or **NEXT_TEMPLATE** is sufficient.

DEFAULT Element

DEFAULT Element

Description

The **DEFAULT** element defines the default behavior.

Syntax

```
<!ELEMENT      DEFAULT      EMPTY >
<!ATTLIST     DEFAULT
              NEXT_STATE     CDATA      #IMPLIED
              NEXT_TEMPLATE  CDATA
```



```
<module name="mymodule" type="RFC"
...
default next state = "<next state>"
...
</module>
```

Attributes of the **DEFAULT** element:

Attribute	Description	Required
NEXT_STATE	Specifies the next state.	If NEXT_TEMPLATE is not specified.
NEXT_TEMPLATE	Specifies the next HTML ^{Business} template without the <code>.html</code> extension. You can also specify the service and theme. For example: NEXT_TEMPLATE = " <code><service>/<theme>/<template></code> "	If NEXT_STATE is not specified.

Either **NEXT_STATE** or **NEXT_TEMPLATE** is sufficient.

If a **DEFAULT** element is present, either a state transition takes place or another HTML^{Business} template is processed.

If no **DEFAULT** element is present, and no other method to leave the current state is specified, the current template becomes the next template to be processed.

The **DEFAULT** element is processed **only** if no other test using the **RESULT** or **EXCEPTION** element has passed.

EVENT Element

Description

The **EVENT** element defines a test for a specified event.

Syntax

```

<!ELEMENT      EVENT EMPTY >
<!ATTLIST     EVENT
              NAME          CDATA          #REQUIRED
              NEXT_STATE    CDATA          #IMPLIED
              NEXT_TEMPLATE CDATA
              #IMPLIED>
    
```

If the event specified in the **NAME** attribute is raised, either the next state or the next HTML^{Business} template is processed.

You raise an event by specifying the parameter `~event` either in the URL of an hypertext link or as part of the HTML form data.

Two event types deserve some attention. Either or both can be present in a flow file.

Event Type	Description
onload	This event type is processed when an HTML ^{Business} template is displayed for the first time. It is not triggered for subsequent refreshing of the same page.
ontouch	This event type is processed whenever an HTML ^{Business} template is loaded or refreshed.

You can have multiple event definitions, and you can declare an event called `onload`, which is automatically processed during execution.

Attributes of the **EVENT** element:

Attribute	Description	Required
NAME	Name of a module test exception.	Yes.
NEXT_STATE	Specifies the next state.	If NEXT_TEMPLATE is not specified.
NEXT_TEMPLATE	Specifies the next HTML ^{Business} template without the <code>.html</code> extension. You can also specify the service and theme. For example: <code>NEXT_TEMPLATE = "<service>/<theme>/<template></code>	If NEXT_STATE is not specified.

Either **NEXT_STATE** or **NEXT_TEMPLATE** is sufficient.

EVENT Element

Module Provider Connection Types

The Internet Transaction Server (ITS) recognizes the following connection types in the flow logic:

- Stateful
- Stateless
- Pooled
- Cached

Connection Type: Stateful

This is a dedicated RFC connection established in a specific user context.

Stateful RFC connections are maintained during the lifetime of a user's Web browser session and closed when the user terminates the session, or when a session expires after session timeout.

Stateful calls use an existing connection with the specific user logon information. In this case, subsequent calls depend on the outcome of the previous state, and on the user making the call.

After the call, the RFC connection to the external system is kept open and used for subsequent stateful calls.

Unless a stateful connection is terminated explicitly (with the event `~Logout`), timeout will close the connection.



An example of a stateless call would be:

```
<module name= "MyModule" type="RFC" stateful="1">
```

Connection Type: Stateless

This is a reusable RFC connection established in a non-specific user context.

Stateless RFC connections are used for a single call (a single request/reponse cycle) and then closed.

Stateless calls are used where subsequent calls do not depend on the outcome of any previous state, or which user is making the call. You could use stateless calls for operations such as catalog searches.



An example of a stateless call would be:

```
<module name= "mymodule" type="RFC">
```

Calls are stateless by default, so you do not need to define a stateless call explicitly.

Connection Type: Pooled

The ITS maintains a (fixed size) pool of RFC connections. If a connection type is pooled, an existing RFC connection can be used. A pooled user must be an anonymous R/3 user.

Pooled calls are stateless calls by default, but there is a slight difference:

Module Provider Connection Types

- With stateless connections, the connection is always closed after a call.
- With pooled connections, the R/3 context is cleared, but the connection is returned to connection pool. This reduces the connection overhead.

You can use a free connection from the pool of connections where:

```
conn.login = user.login
conn.language = user.language
conn.client = user.client
```

If no connection is available, the least recently used connection is closed and a new one opened with appropriate user, language, and client. The anonymous user and password is stored in the service file.

You define pooled connections in the service file with the following parameters

- `~poollogin`
- `~poolpassword`
- `~poolclient`

If these parameters are not defined in the service file, the ITS uses the value defined for the parameters `~login`, `~password`, and `~client`.



An example of a pooled call would be:

```
<module name= "MyModule" type="RFC" pooled="1">
```

Connection Type: Cached

Cached calls are pooled connections by default, because the user has to be anonymous, so only static (read only) results of calls can be cached.

With this connection type, the ITS looks in the cache for the data. If nothing is found, the results of the call are written to the cache.



An example of a cached call would be:

```
<module name= "MyModule" type="RFC" cache="1">
```

Summary of Connection Types

You can use a combination of stateful, stateless, and pooled connections.

At present, specification of the connection type in the flow logic is achieved using three 3 boolean attributes:

- `stateful`
- `pooled`
- `cache`

If no stateful attribute is defined, 0 (stateless) is the default.

Properties of the different connection types:

Module Provider Connection Types

Connection Type	Flow Logic Syntax	State	Logon	Cached
Stateful	<code>stateful="1"</code>	Yes	User	No
Stateless	Not required (default)	-	User	No
Pooled	<code>pooled="1"</code>	-	Anonymous	No
Cached	<code>cache="1"</code>	-	Anonymous	Yes

Module Call Result Caching

Module Call Result Caching

The Internet Transaction Server (ITS) caching scheme is designed to reduce the load on the R/3 System (or other external system) when resolving identical requests.

When a user starts an Internet application, a typical request could involve searching a catalog based on search criteria. By caching the result data in the ITS, we can eliminate the overhead of accessing R/3 every time, and thus considerably improve system performance.

Cache Administration

There are several variables available for managing the cache. You can:

- Specify the cache size
 - The cache size is defined during ITS setup, but you can specify a different size by modifying the variable `CacheSize` in ITS Administration.
- Clear the cache
 - You can clear the cache at any time with the relevant utility function in ITS Administration.
 - You can clear the cache at a specified time every day by setting the following variables in ITS Administration.
 - `CacheInvalidateHour`
 - `CacheInvalidateMinute`

If you do not set these variables, the ITS uses a default time.

Enabling the Cache in the Flow Logic

You can enable caching at module call level by setting the parameter `cache` to 1.



```
<module name="mymodule" type="RFC" cache = "1">
```

By default, the `cache` parameter is set to 0.

The cache should be used only if the results of module calls are expected to be static for a reasonable amount of time.

Accessing Cache Statistics

If you have access to privileged commands, you can display the cache statistics in your Web browser by entering the following URL:

```
http://<myserver:myport>/scripts/wgate/<service>?~command=CacheStats
```

The resulting table displays cache statistics for the current AGate process and specifies values for the following:

Cache Statistic	Variable	Description
Cache Memory	<i>Free</i>	Current amount of unused cache (in bytes).
	<i>Used</i>	Current amount of used cache (in bytes).

Module Call Result Caching

	<i>Maximum</i>	Maximum cache size allowed (in bytes).
Cache Elements	<i>In memory</i>	Amount of context data currently stored in memory.
	<i>On disk</i>	Amount of context data currently stored on disk.
	<i>Total</i>	Total amount of context data currently stored in memory and on disk.
Cache Events	<i>Total hits</i>	Total number of cache hits since initialization.
	<i>Total misses</i>	Total number of cache misses since initialization.
	<i>Hits (memory)</i>	Number of cache hits where data was in memory
	<i>Hits (disk)</i>	Number of cache hits where data was on disk

Cached data is identified by:

- Module name
 - In the case of modules implemented in the R/3 System, this would be the name of the Business API (BAPI) or remote-enabled function module.
- Parameter values
 - This includes input field parameters, but not table parameters. Tables as results are cached, but they are not used to identify the cached data. Therefore, if function modules use tables as input parameters, this would lead to cached results being independent of these parameters.
- Client
- Language

If the cache exceeds its maximum size, the contents are written to a cache file. This file is shared between AGates and different machines.

Flow File Application Components

Flow File Application Components

Each flow file application driven by the Internet Transaction Server (ITS) consists of several components:

- Components created in the R/3 System (or other external system):
 - This is a set of modules to implement the business logic.

If you define module calls in the R/3 System, you can implement the business logic with Business APIs (BAPIs) or standard remote-enabled function modules.
- Components created in the SAP@Web Studio:
 - Service file (required)

The service file contains the service description, which is the the set of parameters that determines how an application runs.

Service file names have the format `<service>.srvc`.

Each service can be divided into one or more themes. Themes are instances of services that differ only in look and feel, but not in functionality.
 - HTML^{Business} templates (required)

HTML^{Business} templates are the means used by the ITS to display application screens in a Web browser when running a service.

For each screen, there must be one HTML^{Business} template. Each template contains standard HTML code, and HTML^{Business} statements.

HTML^{Business} is an SAP-specific macro language, which allows you to merging R/3 data dynamically into HTML templates.

HTML^{Business} template names have the format `<template>.html`.
 - Flow files (required)

Flow files contain the flow logic that defines logical transitions between application states depending on what the user decides to do when running an application.

When you are developing applications with flow files, you need to generate one flow file for each HTML^{Business} template that requires a dialog flow definition.

Flow file names have the format `<template>.flow`.
 - Language resource files (optional)

Language resources are language-independent texts used by the ITS to run a service in a particular language.

Language resource file names have the format `<service>_<language>.htrc`.
 - Multipurpose Internet Mail Extension (MIME) files (optional)

MIME files contain the image, sound, and video elements you may want to include in services to enhance the visual appearance and effectiveness of your application.

All ITS files (except MIME files) are stored on the ITS server under:

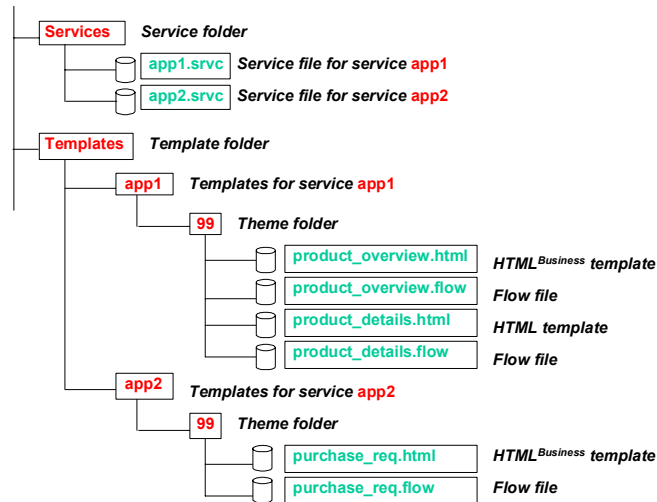
```
c:\program files\SAP\ITS\2.0\<virtual ITS>\
```

Flow File Application Components

MIME files are stored on the Web server under:

```
c:\Inetpub\wwwroot-<virtual ITS>\SAP\ITS\mimes\<service>\
```

The following graphic shows the ITS directory structure for a typical flow file application:



For full details about ITS file types created in the SAP@Web Studio, see [ITS File Types \[Ext.\]](#).

Flow File Applications: Example Scenarios

Flow File Applications: Example Scenarios

The following example scenarios demonstrate how to implement two different flow file applications:

- [Flow File Example 1: Display Development Classes \[Page 41\]](#)
- [Flow File Example 2: Online Store \[Page 43\]](#)

Flow File Example 1: Display Development Classes

This example demonstrates how to implement a function that searches for development classes matching a certain pattern.

This application requires two HTML^{Business} templates:

- `devcsearch.html`
- `devcdisplay.html`

Each of these templates requires a matching flow file that defines the flow logic.

`devcsearch.html`

Since `devcsearch.html` does not display any data, it does not require a module call when the template is loaded. Therefore, the flow logic simply points to the next template.

`devsearch.flow`

```
<FLOW>

  <EVENT NAME="Search" NEXT_TEMPLATE="devcdisplay">
  </EVENT>

</FLOW>
```

`devsearch.html`

```
<HTML>
  <BODY>

    <FORM ACTION="`wgateURL()`" METHOD="POST">

      <INPUT TYPE="TEXT" NAME="PATTERN">
      <INPUT TYPE="SUBMIT" NAME="~Event"
VALUE="Search">

    </FORM>

  </BODY>
</HTML>
```

When the user chooses *Search* on the HTML page, the event `search` is raised. This prompts the Internet Transaction Server (ITS) to process the HTML^{Business} template `devcdisplay.html`.

Flow File Example 1: Display Development Classes

devcdisplay.html

The flow logic of `devcdisplay.html` triggers the module `Devclass.GetList`, which retrieves a list of development classes that match the entered pattern.

Suppose the module `DevClass.GetList` is implemented by the `BAPI_GET_DEVCLASSES` function, which has a parameter `PATTERN` that is automatically passed from the request context. The BAPI returns an internal table `T_TDEVC` which has the same structure as the database table `TDEVC`.

Function BAPI_GET_DEVCLASSES

```
FUNCTION BAPI_GET_DEVCLASSES.  
  
    REFRESH T_TDEVC.  
  
    SELECT * FROM TDEVC WHERE DEVCLASS LIKE PATTERN  
           INTO TABLE T_TDEVC.  
  
ENDFUNCTION.
```

devdisplay.flow

```
<FLOW>  
  
    <STATE NAME="GetList">  
        <MODULE NAME="Devclass.GetList" type="BAPI">  
        </MODULE>  
    </STATE>  
  
    <EVENT NAME="onLoad" NEXT_STATE="GetList">  
    </EVENT>  
  
</FLOW>
```

devdisplay.html

```
<HTML>  
  
    <BODY>  
  
        <TABLE>  
            `repeat with i from 1 to T_TDEVC.dim`  
            <TR><TD>`T_TDEVC-DEVCLASS[i]`</TD></TR>  
            `end`  
        </TABLE>  
  
    </BODY>  
</HTML>
```

Flow File Example 2: Online Store

This example demonstrates how to implement a function that prompts users to log on in one of the following situations:

- When they place an item in the shopping basket for the first time
- When they proceed to checkout and place the order

In both cases, the logon procedure is the same, but the position in the dialog flow is different. For this reason, you can implement the logon procedure with an HTML^{Business} template called `login.html`, which allows users to enter a user name and password.

login.html

Here is the HTML^{Business} template, followed by the flow file:

login.html

```
<!-- login.html -->

<HTML>

  <BODY>

    <FORM ACTION="`wgateURL()`" METHOD="POST">
      <INPUT TYPE="TEXT" NAME="USERNAME">
      <INPUT TYPE="PASSWORD" NAME="PASSWORD">
      <INPUT TYPE="SUBMIT" NAME="~Event"
VALUE="Login">
    </FORM>

  </BODY>

</HTML>
```

login.flow

```
<FLOW>

  <STATE NAME="Login">
    <MODULE NAME="User.Login" TYPE="BAPI"
STATEFUL="1">
      <EXCEPTION NAME="LOGIN_FAILED">
      </EXCEPTION>
      <DEFAULT NEXT_TEMPLATE="proinfo">
      </DEFAULT>
    </MODULE>
  </STATE>

  <EVENT NAME="Login" NEXT_STATE="Login">
```

Flow File Example 2: Online Store

```
</EVENT>
```

```
</FLOW>
```

The module `User.Login` could be implemented by the function module `BAPI_USER_LOGIN`:

Function `BAPI_USER_LOGIN`

```
FUNCTION BAPI_USER_LOGIN.

    DATA: UNAME (40) .
    DATA: PASSWORD (40) .

    * Some password verification code ...

    IF PASSWORD NE MYPASSWORD.
        RAISE LOGIN_FAILED.
    ELSE.
        CUSTOMER = UNAME.
    ENDIF.

ENDFUNCTION.
```

`BAPI_USER_LOGIN` checks the password entered by the user:

- If the password is not correct, an exception is raised
- If the password is correct, the customer number is stored in the variable `CUSTOMER`.

Since `BAPI_USER_LOGIN` is stateful (as defined in the flow logic), the customer number is retained throughout the user session.

If the login procedure is successful, the next template is `prodinfo.html`.

`prodinfo.html`

The product information HTML^{Business} template `prodinfo.html` contains an *Add* button and an *Order* button. Both buttons can call the stateful module `BAPI_IS_USER_LOGGED_ON`, which checks whether the customer is already logged on.



Since `BAPI_USER_LOGIN` and `BAPI_IS_USED_LOGGED_ON` belong to the same function group, the variable `CUSTOMER` is shared between them.

`prodinfo.flow`

```
<FLOW>

    <STATE NAME="CheckLogin">
        <MODULE NAME="User.IsLoggedIn" TYPE="BAPI"
STATEFUL="1">
            <EXCEPTION NAME="LOGIN_REQUIRED"
                NEXT_TEMPLATE="login.html">
            </EXCEPTION>
            <DEFAULT NEXT_TEMPLATE="shopping_basket"
            </DEFAULT>
```

Flow File Example 2: Online Store

```
</MODULE>
</STATE>

<EVENT NAME="Add"    NEXT_STATE="CheckLogin">
</EVENT>
<EVENT NAME="Order" NEXT_STATE="CheckLogin">
</EVENT>

</FLOW>
```

Flow File Example 2: Online Store

The module `User.IsLoggedIn` is implemented by the function `BAPI_IS_USER_LOGGED_ON`:

Function `BAPI_USER_IS_LOGGED_ON`

```
FUNCTION BAPI_USER_IS_LOGGED_ON.  
  
    IF CUSTOMER IS INITIAL.  
        RAISE LOGIN_REQUIRED.  
    ENDIF.  
  
ENDFUNCTION.
```

Implementing Portal Pages

You can implement portal pages either as HTML^{Business} templates that are part of an HTML frameset or as a single page.

On the container page, you can use the HTML^{Business} macro `include` to place an HTML^{Business} template on the container page. The flow logic is processed for each included template:

```
<HTML>  
<BODY>  
    <TABLE>  
  
<TR><TD>`include (~name="product_overview")`</TD></TR>  
  
<TR><TD>`include (~name="recommendations")`</TD></TR>  
    </TABLE>  
</BODY>  
</HTML>
```