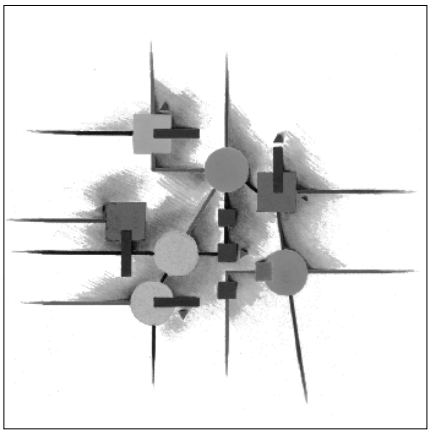
SAP Business Connector SOAP Programming Guide



SAP SYSTEM

Release 4.8

SAP® AG • Dietmar-Hopp-Allee 16 • D-69190 Walldorf

Copyright

Icons

Icon	Meaning
	Caution
	Example
	Note
	Important

CONTENTS

Chapter 1: Introduction

Welcome!

Welcome to the *SAP BC SOAP Programming Guide*. This guide describes how to use the SAP BC SAP BC server to exchange SOAP messages over the Internet. It is for solution developers who need to understand:

- How to receive and process SOAP messages and SOAP remote procedure calls with the SAP BC SAP BC server.
- How to build customized processors that operate on SOAP messages that the SAP BC SAP BC server receives.
- How to submit SOAP messages and SOAP remote procedure calls to other servers.

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 Server Administrator's Guide	Information about using the SAP BC 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 is for solution developers. You will find this book at: <*sapbc*>\Developer\doc\SAPBCDeveloperGuide.pdf
SAP BC Server Built-In Services Reference Guide	Descriptions of services that are installed on your SAP BC Server. This book is for solution developers. You will find this book at: <*sapbc*>\Developer\doc\SAPBCBuiltInServicesGuide.pdf
SAP BC Server Java API Reference	Descriptions of the Java classes you can use to create Java clients and services. This reference is for solution developers who build services using Java. You will find this book at: <*sapbc*>\Server\doc\api\Java\index.html and <*sapbc*>\Developer\doc\api\Java\index.html

Viewing this Document

To view this document, you must have Acrobat Reader™ Version 4.0 or later installed on your system. If you have an earlier version of Acrobat Reader, you will receive the following message when you open the document:

```
Could not find the Color Space named 'Cs8.'
```

To download a free copy of Acrobat Reader, go to:

http://www.adobe.com

Printing this Document

To produce a hard copy of this document, print it from Acrobat Reader. You will find the document's title page and table of contents at the *end* of the printed copy. To create a traditional, paper-based manual, simply move these pages to the front of the document after it is printed.

Other References

The following is a list of additional resources that provide information about SOAP.

- *Simple Object Access Protocol (SOAP) 1.1 - W3C Note 08 May 2000* at http://www.w3.org/TR/SOAP/.
- *SOAPRPC.com* resources at http://www.soaprpc.com/resources/.

Chapter 2: Overview of SOAP

What is SOAP?

The *Simple Object Access Protocol* (SOAP) is a standard, lightweight protocol for exchanging messages across the Internet. It uses XML to define a basic message packet that can be used to convey an arbitrary XML document or a remote procedure call (RPC). This description is based on the *Simple Object Access Protocol (SOAP) 1.1*. Differences to the newer Version 1.2 can be found on the website of the *World Wide Web Consortium* (http://www.w3.org/TR/2007/REC-soap12-part0-20070427#L4697).

What Does a SOAP Message Look Like?

A SOAP message uses XML to form a simple message packet. The packet consists of an *envelope* that encloses two elements: an optional *header* and a mandatory *body*.

A simple SOAP message packet

Note: Although not shown in the example above, SOAP 1.1 also allows additional, implementation-specific elements to follow the body of a SOAP message. In this book, such elements are referred to as *trailers*. For more information about trailers, see “Trailers” on page 16.

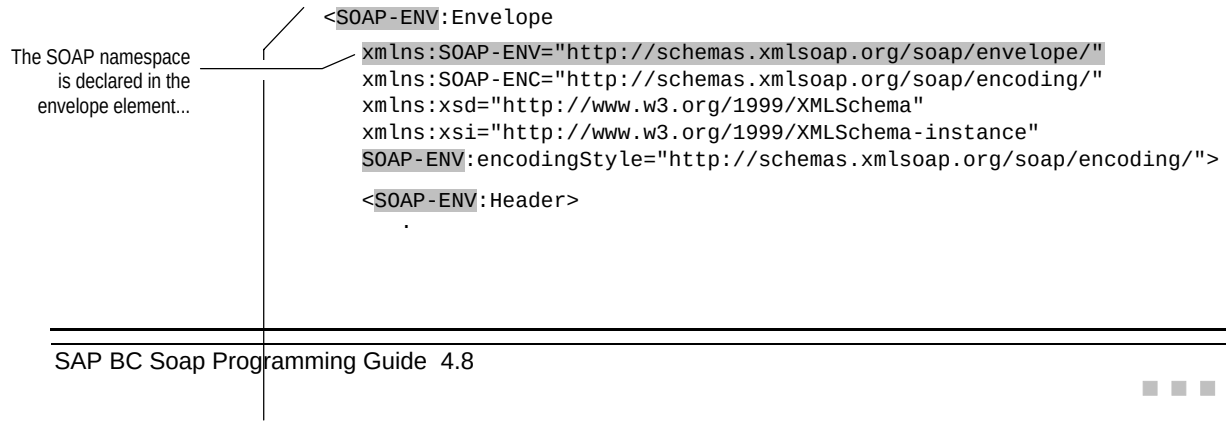
The Envelope

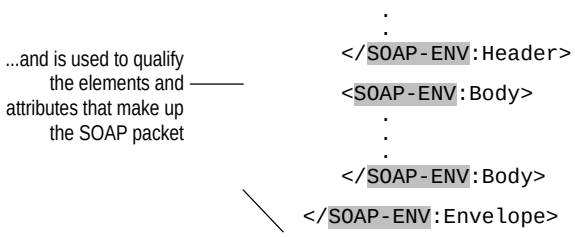
The envelope is the top-level element in a SOAP message—it is the "container" that holds the entire message. The envelope must be the first (that is, the outermost) element in a SOAP message. It has the name Envelope.

- The envelope *may* contain a header element. When a SOAP message contains a header, the header element must be the first child within the envelope. For additional information about the header element, see "The Header" on page 12.
- The envelope *must* contain a body element. The body carries the content of the message. For more information about the body element see "The Body" on page 14.
- The envelope *must* be associated with a SOAP namespace. The SOAP namespace serves as the qualifier for the message's SOAP-related elements and attributes. It also specifies the XML schema to which the message conforms. For additional information about the SOAP namespace, see "The SOAP Namespace Declaration," below.
- The envelope may include other implementation-specific namespace declarations.
- The envelope may contain the encodingStyle attribute, which specifies the way in which the elements within the envelope are serialized and deserialized.
- The envelope may contain additional implementation-specific attributes, but if it does, these attributes must be namespace qualified.
- The envelope may contain additional implementation-specific children besides the header and body elements; however, if it does, the additional elements must be namespace qualified and must follow the body element.

The SOAP Namespace Declaration

A SOAP 1.1 message uses the namespace http://schemas.xmlsoap.org/soap/envelope/ to qualify the elements and attributes that make up the SOAP message packet. A SOAP 1.1 message *must* declare this namespace in the SOAP envelope. By convention, the prefix SOAP-ENV is given to the SOAP namespace as shown in the following example; however, a message may use any prefix to represent the SOAP namespace.





The primary purpose of the SOAP namespace is to distinguish SOAP-related elements and attributes from the application-specific elements and attributes conveyed in the message. The SOAP namespace also serves another purpose—it specifies the schema to which the SOAP message conforms.

Messages that conform to the *Simple Object Access Protocol (SOAP) 1.1 - W3C Note 08 May 2000* must use the following namespace:

http://schemas.xmlsoap.org/soap/envelope/

Messages whose envelopes declare a different namespace, or do not declare a namespace at all, are considered invalid and are rejected by the SAP BC Server.

Note: Unless otherwise noted, when the namespace prefix `SOAP-ENV` appears in this document, it represent the namespace http://schemas.xmlsoap.org/soap/envelope/.

The Header

A SOAP message can optionally include a header to convey information peripheral to the document in the body of the message. The header element provides a place where a sender can pass auxiliary, implementation-specific information such as authorization codes, routing information, version numbers, or message IDs. For example, if your solution routes invoices through one or more approval steps before passing it to an accounts-payable processor, the header could hold the document's routing information.

When a header is included in a SOAP message, it must appear as the first child element within the envelope and must have the name Header. A header may contain one or more child elements. Each child is referred to as a *header entry*. All header entries must be namespace qualified.

Important! The inclusion of a header, and the significance of the entries within it, are completely driven by the implementation—the SOAP specification does not define any standard header entries. It simply provides the header as a container that implementers can use as needed. The parties exchanging SOAP messages are completely responsible for defining any header entries that their solution requires and for processing them correctly.

The following example shows a SOAP envelope containing one header entry called <MSG:priority>. Note that the entry is namespace qualified, which is a requirement of all header entries in a SOAP message.

Soap Message with one header entry

This message has a header...

...containing one entry

```
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:xsd="http://www.w3.org/1999/XMLSchema"
   xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
   SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Header>
      <MSG:priority xmlns:MSG="http://www.gsx.com/">9</MSG:priority>
   </SOAP-ENV:Header>
   <SOAP-ENV:Body>
      .
      .
      .
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SOAP Header Attributes

The SOAP specification defines two optional attributes for a header entry. These attributes allow an entry to specify its intended recipient and to indicate whether the recipient is required to process the entry.

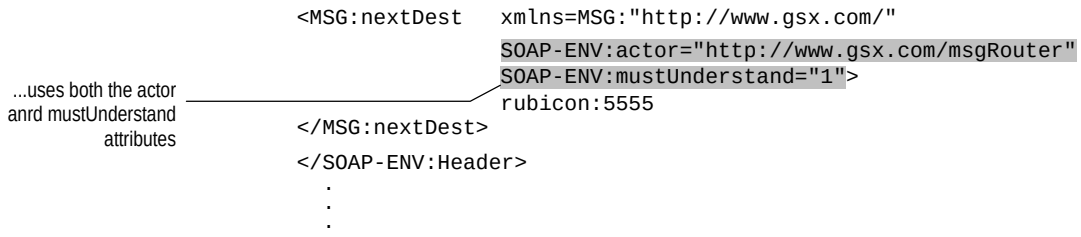
Attribute	Description
actor	Identifies the intended recipient of the header entry. This attribute allows a sender to direct a header entry to a specific process.
mustUnderstand	Indicates whether the processing of the header entry is mandatory. Recipients that do not recognize an entry whose mustUnderstand attribute is set *must* reject the message and return a fault to the client.

The following example shows a SOAP header entry that uses both the actor and mustUnderstand attributes. Note that the attributes are qualified with the SOAP namespace.

Message that uses the 'actor' and 'mustUnderstand' attributes

This header entry...

```
      .
      .
      .
<SOAP-ENV:Header>
```



The actor and mustUnderstand attributes may be omitted, used alone, or used together. For more information about using the actor and mustUnderstand attributes, see the *Simple Object Access Protocol (SOAP) 1.1 - W3C Note 08 May 2000* at http://www.w3.org/TR/2000/NOTE-SOAP-20000508/#_Toc478383499.

The Body

When a SOAP message conveys an arbitrary XML document, (sometimes referred to as *application data* or the *business payload*) the document is carried in the body of the message. When SOAP is used as an RPC protocol, the body specifies the method name that the client is calling and carries the method's input values.

A SOAP message *must* contain a body element. This element must be named Body. If a SOAP message contains a header, the Body element must appear immediately *after* the header. Otherwise, the body must be the first element in the SOAP envelope.

Each immediate child of the Body element is referred to as a *body entry*.

A body containing one entry

```
<SOAP-ENV:Body>
   <GL:JournalEntry xmlns:GL="http://www.gsx.com/gl/">
      <transaction>
         <entry>
            <Id>2398</Id>
            <amt>237.50</amt>
            <acct>Cash</acct>
         </entry>
         <entry>
            <Id>2398</Id>
            <amt>-237.50</amt>
            <acct>AR</acct>
         </entry>
      </transaction>
   </GL:JournalEntry>
</SOAP-ENV:Body>
```

A body containing two body entries

```
   .
   .
   .
<SOAP-ENV:Body>
   <RQ:customer xmlns:RQ="http://www.gsx.com/rfq/">
      <acctNo>AGT-432398</acctNo>
      <name>GSX Sporting Goods</name>
      <phone>218-376-2500</phone>
   </RQ:customer>

   <RQ:addr xmlns:RQ="http://www.gsx.com/rfq/">
      <street1>1501 Bridger Hwy<street1>
      <street2></street2>
      <city>Laurel</city>
      <state>MN</state>
   </RQ:addr>
</SOAP-ENV:Body>
   .
   .
   .
```

Note: Although a SOAP message must contain a body, the body does not have to contain data. A message that has an empty Body element is a valid SOAP message.

SOAP Fault Elements

The SOAP specification defines one body element, whose name is Fault. A recipient must return the Fault element if it cannot process a SOAP message successfully.

The Fault element contains the following children:

Element	Value
`<faultcode>`	A qualified name indicating the type of error that occurred. The SOAP specification defines several standard error codes (e.g., `SOAP-ENV:Server`, `SOAP-ENV:Client`). See http://www.w3.org/TR/SOAP/#_Toc478383510 for details.
`<faultstring>`	A string describing the fault that occurred.
`<faultactor>`	Optional. A URI indicating which process or application produced the fault.
`<detail>`	Optional. An element containing implementation-specific details about the error. This element *must* be present if the error occurs while processing the body of the message. For a description of the detail element returned by the SAP BC Server, see Appendix A *SOAP Faults Returned by the SAP BC*

server.

The following shows an example of the SOAP fault that the SAP BC server returns when a sender submits a message to a non-existent SOAP processor.

A SOAP message returning a fault

A fault is returned in the body of a message

```
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsd="http://www.w3.org/1999/XMLSchema"
   xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance">
   <SOAP-ENV:Body>
      <SOAP-ENV:Fault>
         <faultcode>
            SOAP-ENV:Server
         </faultcode>
         <faultstring>
            [B2BPCKG.0088.9123] Requested SOAP processor
             mySoapProc is not registered on this server
         </faultstring>
         <faultactor>
            http://localhost:5555/soap/mySoapProc
         </faultactor>
      </SOAP-ENV:Fault>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

When you write clients that submit SOAP messages to the SAP BC server, your client code should test for the presence of a fault code and process the response appropriately. For information about when and how the SAP BC server returns a SOAP fault, see Appendix A *SOAP Faults Returned by the SAP BC server.*

Trailers

The SOAP 1.1 specification permits additional implementation-specific elements (elements besides a header and a body) to reside in a SOAP envelope. The SAP BC server refers to these elements as *trailers.* If a SOAP envelope carries trailers, they must appear *after* the body and they must be namespace qualified.

Important! It appears likely that trailers will not be permitted in future versions of SOAP (versions 1.2 and later). *If you are designing a completely new solution, we recommend that you avoid using trailers*. However, if you exchange SOAP messages with older systems that make use of trailers, SAP BC provides services that allow you to work with them.

A SOAP message containing two trailers

```
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:xsd="http://www.w3.org/1999/XMLSchema"
   xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
   SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">

   <SOAP-ENV:Header xmlns:MSG="http://www.gsx.com/">
      <MSG:priority>9</MSG:priority>
   </SOAP-ENV:Header>

   <SOAP-ENV:Body>
      <GL:JournalEntry xmlns:GL="http://www.gsg.com/gl/">
         <transaction>
            <entry>
                <amt>237.50</amt>
                <acct>Cash</acct>
            </entry>
         </transaction>
      </GL:JournalEntry>
   </SOAP-ENV:Body>
```

trailer

```
   <AUDIT:TransInfo xmlns:AUDIT="http://www.etrans.com/monitor/>
      <server>14.226.151.96</server>
      <port>5540</port>
      <time>2001-06-13 16:00:00 5</time>
      <node>http://www.gsx.com/clearInv</node>
   </AUDIT:TransInfo>
```

trailer

```
   <AUDIT:TransInfo xmlns:AUDIT="http://www.etrans.com/monitor>
      <server>20.117.70.33</server>
      <port>8081</port>
      <time>2001-06-13 16:00:04 5</time>
      <node>http://www.gsx.com/updateCustAcct</node>
   </AUDIT:TransInfo>

<SOAP-ENV:Envelope
```

Trailers allow you to transmit information in a SOAP message without placing it in the body or the header of the message. Although used infrequently, they are permitted by the SOAP 1.1 specification and they provide a mechanism that you can use to deliver information to a pre-processor, a message handler or some other intermediate process without trespassing on the message's header or body.

Chapter 3: SOAP Support on the SAP BC Server

Overview

Support for SOAP is delivered by a SOAP message handler and a set of built-in services. These facilities allow you to:

- Receive and process SOAP messages via HTTP or HTTPS.
- Submit SOAP messages to other servers via HTTP or HTTPS.
- Compose and decompose SOAP messages using a set of built-in services.
- Make remote procedure calls via SOAP RPC.
- Make SAP BC services available to clients via SOAP remote procedure calls.

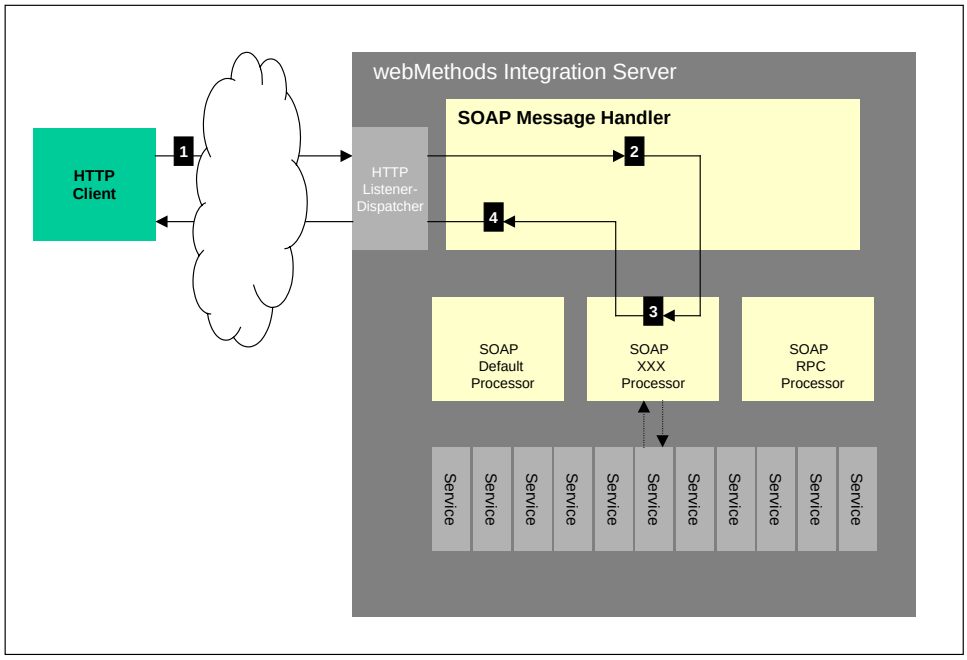
SOAP Versions Supported by webMethods

The SAP BC Server supports SOAP 1.1 as described in *Simple Object Access Protocol (SOAP) 1.1 - W3C Note 08 May 2000* at http://www.w3.org/TR/2000/NOTE-SOAP-20000508.

Receiving SOAP Messages with the SAP BC server

The following diagram illustrates the way in which the SAP BC server receives and processes SOAP messages.

Processing and receiving a SOAP request



Stage 1

The HTTP Client Posts a SOAP document to the SAP BC server

An HTTP client submits a SOAP message by posting it to the URL for a SOAP processor on the SAP BC server. A SOAP processor is a special service that operates on SOAP messages.

The URL for a SOAP processor has the following format:

`http://`*`hostName`*`:`*`portNum`*`/soap/[`*`processDirective`*`]`

Where...	Is...
hostName	The numeric IP address or the domain name of the SAP BC Server.
portNum	The number of a port on which *hostName* accepts HTTP or HTTPS requests.
processDirective	The process directive associated with the requested SOAP processor. The process directive is a unique name that you assign to a SOAP processor when you register it on the SAP BC server. If *processDirective* is omitted or set to “default,” the message is passed to the default SOAP processor.

Stage 2

The SOAP Message Handler Invokes the Appropriate SOAP Processor

When the SAP BC server receives a message for a SOAP processor, it passes the message to the SOAP message handler, which does the following:

- Verifies that the requested SOAP processor is registered on the server. If the SOAP processor does not exist or is not available, the message handler returns a SOAP fault containing the `B2BSERV.0088.9123` or `B2BSERV.0088.9111` error. (For a description of these error messages, see Appendix A *SOAP Faults Returned by the SAP BC server*.)

- Verifies that the message declares the appropriate SOAP namespace. If the message does not declare a namespace or declares a namespace other than http://schemas.xmlsoap.org/soap/envelope/, the message handler returns a SOAP fault containing the `B2BSERV.0088.9128` error.

- Validates the structure of the message against the SOAP schema. If the message violates the SOAP schema, the message handler returns a SOAP fault containing the `B2BSERV.0088.91125` error.

  Be aware that during the validation step, the message handler validates only the structure of the SOAP envelope. For example, it ensures the message has at least one body element and there is at most one header element. *Validating the application data that is carried inside the SOAP envelope is the responsibility of the processor or application that consumes the SOAP message.*

After the message handler establishes that the SOAP processor is available and it is in receipt of a valid SOAP message, it does the following:

- Transforms the message into an object called *soapRequestData*. This object contains the entire SOAP envelope in addition to other operational values that the SAP BC server uses to manage the message internally.

- Creates an empty object called the *soapResponseData*. This object is used to compose the message to be returned to the client.

- Invokes the requested SOAP processor and passes the *soapRequestData* and *soapResponseData* objects to it.

Stage 3 The SOAP Processor Performs Work and Generates a Response

The selected SOAP processor handles the message in *soapRequestData* (usually by calling other services on the SAP BC Server) and composes a response message in *soapResponseData*.

The SAP BC server is installed with two SOAP processors: the *default* processor and the *RPC* processor. You can also build and register custom SOAP processors to suit your needs.

Stage 4 The Message Handler Returns the Response to the Client

When the SOAP processor ends or exits, the message handler generates an HTTP response from the message contained in *soapResponseData* and returns it to the client.

Note: If the SOAP processor or one of the services it calls throws an exception, the message handler automatically returns a SOAP fault to the client.

Sending SOAP Messages with the SAP BC server

Besides receiving and processing SOAP messages, the SAP BC server can send SOAP messages to remote servers via HTTP.

To send a SOAP message, you execute pub.client:soapHTTP with the following input parameters:

Input Parameter...	Description...
Address	The URL where the SOAP message is to be sent.
soapRequestData	The *soapRequestData* object containing the message that you want to send. You construct and populate this object using the server's message composition services (e.g., createSoapData, addHeaderEntry, addBodyEntry).
Auth	The user name and password that the SAP BC server must supply when it connects to the target server.

This service returns a *soapResponseData* object that contains the response document returned by the target server. You use the server's data-retrieval services (e.g., getHeaderEntries, getBody) to retrieve information from the message.

For more information about sending SOAP messages from the SAP BC server, see "Composing and Sending SOAP Messages" on page 68.

Sending SOAP RPC Messages with the SAP BC server

The SAP BC server supports SOAP RPC, which allows you to make remote procedure calls to other servers with SOAP.

To submit a remote procedure call, you execute pub.client:soapRPC with the following basic set of parameters:

Input Parameter...	Description...
Address	The URL to which the remote procedure call is to be sent.
Method	The name of the remote procedure that you want to execute.
reqParms	The input parameters that are to be submitted to the remote procedure.
Auth	The user name and password that the SAP BC server must supply when it connects to the target server.

This service returns a Record called *respParms*, which contains the results from the remote procedure.

For information about submitting SOAP remote procedure calls from the SAP BC server, see "Using the SOAP RPC Client" on page 77.

Chapter 4: Building Solutions with SOAP

Building Solutions that Receive SOAP Messages

If you are building a solution that receives and processes SOAP messages, you will need to do the following:

- Understand the structure of the SOAP message that clients will submit to your SAP BC server.
- Define the work that you want the SAP BC server to perform when it receives the SOAP message.
- Determine whether one of the SOAP processors provided by the SAP BC server will satisfy the needs of your solution or whether you will need to build and register a “custom” SOAP processor.

What is a SOAP Processor?

A SOAP processor is a service that acts upon SOAP messages that the SAP BC server receives. When the SOAP message handler receives a SOAP message, it invokes the SOAP processor based on the *process directive* specified in the URL requested by the client.

The process directive is the last segment of the URL for the SOAP message handler on a SAP BC Server. For example, if a client submits a SOAP request to the following URL, the SOAP message handler would invoke the SOAP processor registered as “genLedger.”

SOAP Processors provided by the SAP BC server

The SAP BC server is installed with the following SOAP processors:

SOAP Processor	Description
default	This is a basic processor that invokes a service based on the name of the first element in the body of the SOAP message. This processor passes the entire envelope to the services that it invokes. For information about the default processor, see “Using the Default SOAP Processor” on page 33.
RPC	This processor processes SOAP remote procedure calls (messages that conform to the remote procedure call (RPC) section of the SOAP specification). For information about the RPC processor, see “Using the SOAP RPC Processor” on page 42.

The SAP BC server also allows you to create your own customized processors and register them with the SOAP message handler. You might do this, for example, if your

SOAP messages do not conform to the structure expected by the supplied processors or if you want to create a processor that is optimized for specific message content.

For information about building and installing customized SOAP processors, see “Creating Custom SOAP Processors” on page 57.

Universal Names

Both the default and RPC processors route messages to services based on a qualified name (QName) that appears in the body of a message. To facilitate routing by QName, every service on a SAP BC Server has a *universal name* in addition to its regular SAP name. A universal name is a unique public identifier that external protocols use to reference a service on a SAP BC Server.

The structure of a universal name is the same as the structure of a QName in an XML namespace—it has two parts: a *namespace name* and a *local name.*

- The namespace name is a qualifier that distinguishes a SAP BC Service from other resources on the Internet that might have the same name. For example, there might be many resources with the name AcctInfo. A namespace name distinguishes one AcctInfo resource from another by specifying the name of the collection to which it belongs (similar to the way in which a state or province name serves to distinguish cities with the same name—for example, Springfield, *Illinois*, versus Springfield, *Ontario*).

  Like namespaces in XML, the namespace portion of a universal name is usually specified as a URI. This convention assures uniqueness, because URIs are based on globally unique domain names.

  A namespace name can be composed of any sequence of characters except leading and trailing white-space characters. For example, the following are all valid namespace names:

  http://www.gsx.com

  myNamespaceName

  gl.journals.cashTransactions

- The local name uniquely identifies a service within a particular namespace. Most sites use the service’s unqualified name as its local name. Under this scheme a service named gl.journals:closeGL would have a local name of closeGL.

  The SAP BC server does not require you to use the service name as a local name. A local name can be composed of any combination of ASCII characters except white-space and the characters identified in the `watt.server.illegalNSChars` parameter on your SAP BC server. By default, these characters are:

  ```
  ? - # & @ ^ ! % * : $ . / \ ` ; , ~ + = ) ( | } { ] [ > <
  ```

Additionally, a local name *must not* start with a number. Any of the following would be a valid local name for a service called orders:postOrder:

postOrder

PO

orders_add_PO

Implicit and Explicit Universal Names

Every service that exists on a SAP BC Server has an *implicit* universal name. In addition, you may optionally assign an *explicit* universal name to a service.

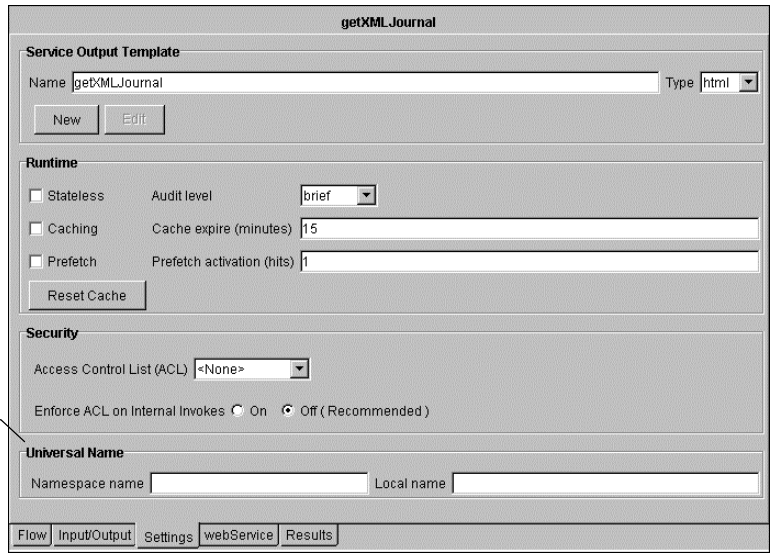
- An implicit universal name is automatically derived from the name of the service itself, where:
  - The *namespace name* is the fully qualified name of the folder in which the service resides.
  - The *local name* is the unqualified name of the service.

  The following shows examples of the implicit names that would be derived from various service names:

Fully qualified service name	Namespace Name	Local Name
gl.journals:jrnlEntry	gl.journals	jrnlEntry
gl.journals.query:viewJournals	gl.journals.query	viewJournals
orders:postPO	orders	postPO

- An explicit universal name is a universal name that you assign to a service with Developer. When you assign an explicit universal name, you specify both the namespace name and the local name.

Settings Tab in Developer 4.x



You assign an explicit universal name on the Settings tab in Developer

Note: It is possible to assign an explicit universal name that is the same as the implicit name of another existing service. When this condition exists, the explicit name takes precedence. That is, when a universal name is requested, the SAP BC server searches its registry of explicit names first. If it does not find the requested name there, it looks for a matching implicit name.

To assign, edit, or view an explicit universal name

1. Start Developer and connect to the server on which the service resides.
2. In the Service Browser, select the service whose universal name you want to assign, edit, or view.
3. Click the **Settings** tab.
4. If you want to assign or edit the service's universal name, specify the following in the **Universal Name** group box:

In this field...	Specify...
Namespace name	The name that will qualify the local name of this service. The name can be composed of any sequence of characters except leading and trailing white-space characters. **Note:** By convention, a URI is generally used as the namespace name (e.g., http://www.gsx.com/gl). This assures that the universal name is globally unique.
Local name	A name that uniquely identifies the service within the collection encompassed by **Namespace name**. The name can be composed of any sequence of ASCII

characters except white-space characters and the characters in `watt.server.illegalNSChars` on your SAP BC server. By default, these characters are:

```
-?#&@^!%*:$./\`;,~+=)(|}{][><
```

Additionally, and the first character of the local name must not be a number.

Note: Most sites use the unqualified portion of the service name as the local name.

5 Click  to save the new settings.

To delete an explicit universal name

1 Start Developer and connect to the server on which the service resides.

2 In the Service Browser, select the service whose universal name you want to delete.

3 Click the Settings tab.

4 In the Universal Name group box, remove the current settings from the Namespace name and Local name fields.

5 Click  to save the new settings.

The Universal Name Registry

The SAP BC Server maintains a registry, called the *Universal Name Registry*, which maps explicit universal names to the services that they represent. The registry is generated each time the SAP BC server is started and is maintained in memory while the server is running.

When you use the Developer to assign, modify, or delete a service's universal name, you update the Universal Name Registry. To view the contents of the registry, you can execute the service pub.universalName:list in Developer and view the contents of the *names* variable on the Results tab. (This service resides in the WmPublic package.)

Services You Use to Interact with the Registry

The following describes services that you can use to display the Universal Name Registry or locate the name of a service associated with an explicit universal name. For more information about these services, see the *SAP BC Server Built-In Services Reference Guide*.

Service	Description
pub.universalName:list	Returns a list of the entries in the current registry. Each Record in the list represents an entry in the registry and contains a service's fully qualified SAP BCname and both parts of its explicit universal name.

Service	Description
pub.universalName:find	Returns the fully qualified service name for a specified explicit universal name.

Building Solutions that Send SOAP Messages

To build a solution that sends a SOAP message to a SOAP-compliant server, you need to do the following:

- Define the structure of the SOAP message that you want to send.
- Determine where the SOAP message will be submitted for processing (get the URL of the server that will process the message).
- Understand the structure of the SOAP message that the server will return in response to the SOAP message that you send.
- Build a service that composes the SOAP message, submits it via HTTP to the appropriate server, and processes the response.

For information about building services that compose and send SOAP messages, see “Composing and Sending SOAP Messages” on page 68.

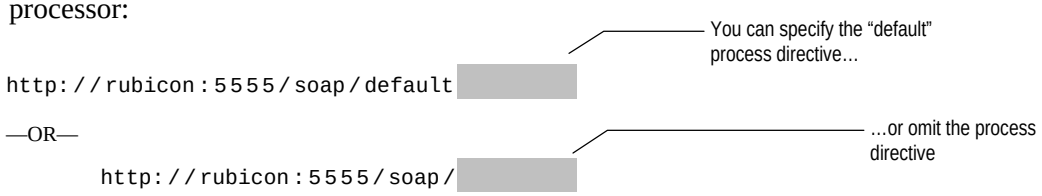
For information about building a services that submit SOAP RPC messages to remote servers, see “Using the SOAP RPC Client” on page 77.

Chapter 5: Using the Default SOAP Processor

Accessing the Default Processor

The SAP BC server provides a default SOAP processor registered under the name "default." The SOAP message handler invokes this SOAP processor when it receives a URL whose process directive is set to "default" or is omitted entirely.

The following examples illustrate the two types of URLs that invoke the default processor:



Behavior of the Default SOAP Processor

The default SOAP processor acts as a dispatcher—it delegates messages to other services on the SAP BC server. It does this by invoking the service whose universal name matches the qualified name (QName) of the message body's first element. This service is referred to as the *target service*.

How the Processor Selects the Target Service

The default SOAP processor selects a target service by matching the fully expanded QName of message body's first element to a universal name (implicit or explicit) of a service on the SAP BC Server.

For example, if the default processor were to receive a SOAP message with the body shown below, it would invoke the service whose universal name is made up of the namespace name http://www/exprint.com/GL/ and the local name JournalEntry.

```
   .
   .
   .
<SOAP-ENV:Body>
   <GL:JournalEntry xmlns:GL="http://www.exprint.com/GL/">
      <Id>2398</Id>
      <date>03/15/2000</date>
      <amt>237.50</amt>
      <acct>Cash</acct>
   </GL:JournalEntry>
</SOAP-ENV:Body>
   .
   .
   .
```

If the default processor received a message with the body shown below, it would invoke the service whose universal name is made up of the namespace name GL.journals.queries and the local name viewJournal.

```
   .
   .
   .
<SOAP-ENV:Body>
   <GL:viewJournal xmlns:GL="GL.journals.queries">
      <acct>Cash</acct>
      <fromDate>01/01/2000</fromDate>
      <toDate>03/31/2000</toDate>
   </GL:viewJournal>
</SOAP-ENV:Body>
   .
   .
   .
```

How the Default Processor Controls Access to Target Services

Access to services invoked via the default processor is controlled by the server's Access Control Lists (ACLs). When a client submits a request to the default processor, its credentials are checked against the ACL for the target service that it requests. If the client is not authorized to execute the target service, the request is rejected.

For information about creating and assigning ACLs to services, see the *SAP BC Server Administrator's Guide*.

What if requested Service Does Not Exist?

If the default processor cannot locate the service whose universal name matches the QName of the body's first element, it returns a SOAP fault to the client with the following error: `[B2BSERV.0088.9122] Service` *namespaceName*`:`*localName* `does not exist.` See Appendix A *SOAP Faults Returned by the SAP BC server* for information about this error.

Processor Inputs and Outputs

When the default SOAP processor calls a target service, it passes two input parameters to the service:

- *soapRequestData*, an object containing the entire SOAP message
- *soapResponseData*, an object containing an empty SOAP message.

When the target service exits, the default SOAP processor returns *soapResponseData* to the SOAP message handler. The message handler extracts the SOAP message from *soapResponseData* and returns the message to the client.

Be aware that neither the default processor nor the SOAP message handler adds any content to the response message in *soapResponseData*. It is the responsibility of the target service to populate *soapResponseData* with message content.

Building Target Services for the Default Processor

To use the default SOAP processor, you must build target services that process incoming SOAP messages. A target service can be any type of service—a flow service, a Java service, a C/C++ service—however, it must accept a *soapRequestData* object and a *soapResponseData* object as input. Additionally, the target service must produce a *soapResponseData* object that is populated with the data that is to be returned to the client.

A target service can contain any sort of logic. A typical target service usually performs three basic tasks:

- It extracts the pertinent information from the incoming SOAP message.
- It processes the information (usually by passing it to one or more services that perform some type of business logic).
- It composes the response SOAP message to be returned to the client.

How to Create a Target Service for the Default Processor

The following describes the general steps you take to create a target service for the default processor.

1 Create a new service that has the following signature:

Inputs:	*soapRequestData*	(of type Object)
	soapResponseData	(of type Object)
Outputs:	*soapResponseData*	(of type Object)

You can use pub.soap.utils:requestResponseSpec to specify the inputs and outputs for the service.

2 Use the SOAP data-retrieval utilities to extract information from the message. The data-retrieval utilities are services such as getBody and getHeaderEntries, which you use to fetch elements from SOAP message objects. These services return the requested element as a *node* (or an array of *nodes*). To extract data from the returned *node*, you query it using the pub.web:queryDocument service.

Important! Be aware that you cannot query a *soapRequestData* object directly. To extract information from *soapRequestData* (or similar SOAP objects, such as *soapData* and *soapResponseData*), you must use one of the data-retrieval services to extract an element of the message (e.g., the header, the body, or the entire envelope) and query the resulting node.

3 Invoke services to perform work on the extracted data. After extracting the data with which you want to work, you can pass it to services that contain your business logic. (To ensure that the data you have extracted is in the correct format, you might want to validate it with pub.schema:validate or make sure that the service to which you pass the data performs data validation on its input parameters.)

4 Use the SOAP message-composition utilities to populate *soapResponseData*. The message-composition utilities are services such as addHeaderEntry and addBodyEntry that you use to add content to the empty message in *soapResponseData*.

The message-composition services require a *node* representation of the header entry, body entry, or trailer that you want to add to the message. You can generate a *node* using services such as pub.web:recordToDocument and pub.web:stringToDocument. For an example of how to do this, see Step 3.1 in the sample code shown on page 38.

5 Assign the appropriate universal name to the service. When you finish building a target service, you must ensure that its universal name matches the QName that clients will use to direct SOAP messages to it. In other words, the service's universal name must match the QName of the first element in the body of the client's SOAP message. If clients will use a QName that does not match the service's implicit universal name, you must set the service's universal name explicitly. For more information about explicitly setting a universal name, see "To assign, edit, or view an explicit universal name" on page 29.

Error Handling

If a target service throws an exception while it is processing, the message handler automatically returns a SOAP fault to the client. Depending on the type of error that occurs, the SOAP fault may include a "detail" element, which provides specific information about the exception. This element will include a stackTrace element if the client is a member of the Developers or Administrators user group.

When the target service throws an exception, the message handler returns a SOAP fault...

..containing an error message...

...and details about the error

```
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsd="http://www.w3.org/1999/XMLSchema"
   xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance">
   <SOAP-ENV:Body>
      <SOAP-ENV:Fault>
         <faultcode>
         SOAP-ENV:Client
         </faultcode>
         <faultstring>
         [B2BSERV.0088.9134] Exception occurred while processing the body of the message
         </faultstring>
         <faultactor>
         http://hostName:portNum/soap/processorName
         </faultactor>
         <detail xmlns="http://www.webmethods.com/2001/10/soap/">
           <exception>
              <className>nameOfClassThatWasThrown</className>
              <message xml:lang="en-US">detailedMessageIssuedbyException</message>
              <stackTrace xml:lang="en-US">textFromStackTrace</stackTrace>
           </exception>
         </detail>
      </SOAP-ENV:Fault>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Note: The SOAP message returned to the client when an exception occurs contains only the SOAP fault. It does not include any message content (e.g., header entries, body entries) that the target service may have inserted into *soapResponseData* before the exception occurred.

For more information about SOAP faults, see Appendix A *SOAP Faults Returned by the SAP BC server*.

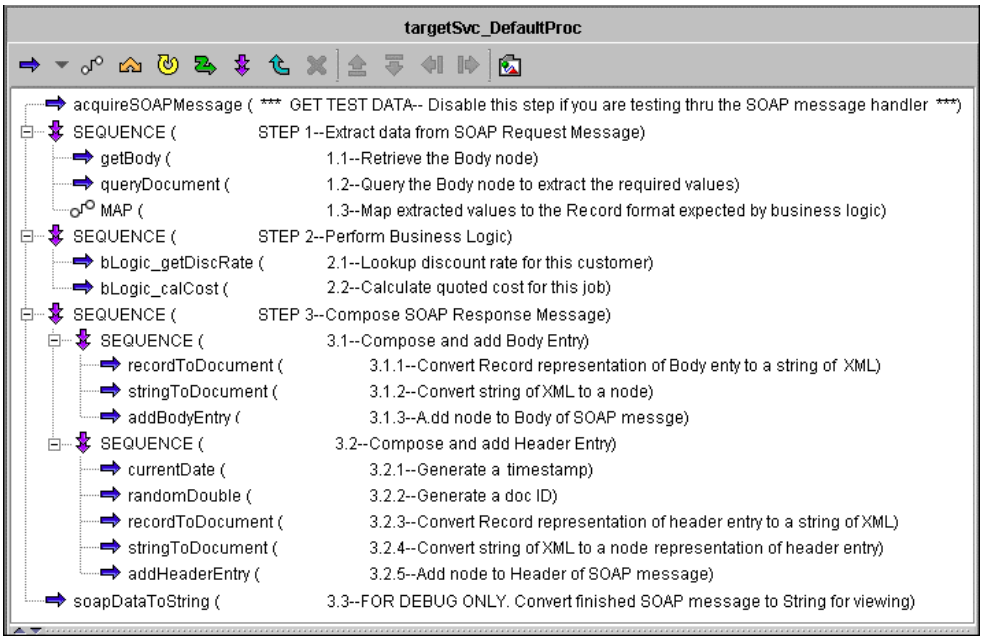
Example—Target Service for Default Processor

The following is an example of a target service that takes a SOAP message, extracts information from the body of the message, passes the information to a set of business services, and composes a SOAP response containing the results of the services.

This example is located in sample.soap:targetSvc_defaultProc in the WmSample package. You may want to open this example with Developer to see how the pipeline is mapped between steps.

Note: If you want to execute this service from Developer, enable the acquireSOAPMessage step at the top of the flow. This service generates a test *soapRequestData* and *soapResponseData* object, which simulates the pipeline that this service would receive from the default SOAP processor. If you want to execute this service as a target of the default processor, disable acquireSOAPMessage.

Target service that extracts data from a SOAP message and composes a response



STEP 1 Extract data from SOAP Request Message. This sequence retrieves several specific pieces of business data by extracting the body of the message from *soapRequestData* and querying the

result. This example expects a SOAP message structured as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xsd="http://www.w3.org/1999/XMLSchema"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">

<SOAP-ENV:Header>

</SOAP-ENV:Header>

<SOAP-ENV:Body>

    <RFQ:quoteReq xmlns:RFQ="http://www.exprint.com/orderSys">

      <acct>1417-A199-0404-5POLY</acct>

      <jobSpecs>

        <copies>5000</copies>

        <stock>30F-SIL</stock>

        <ink>P440</ink>

      </jobSpecs>

    </RFQ:quoteReq>

</SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

To extract the business data from the message, this service executes the following steps:

Step	Description
1.1	getBody—This step retrieves the body of the message from *soapRequestData*. It returns a node that represents the *entire* Body element.
1.2	queryDocument—This step extracts the business data by executing the following XQL queries against the node returned in Step 1.1.

Var Name	XQL Query
acct	`/RFQ:quoteReq/acct/text()`
stock	`/RFQ:quoteReq/jobSpecs/stock/text()`
copies	`/RFQ:quoteReq/jobSpecs/copies/text()`
ink	`/RFQ:quoteReq/jobSpecs/ink/text()`

If you examine the queryDocuement step with Developer, you will see that it also executes the following query, which extracts the entire Body node to a String:

Var Name	XQL Query
wholeNode	`/source()`

This query is included for debugging purposes. It allows you to examine the raw XML associated with the Body node. If you were to put this service into production, you would omit this query.

1.3	MAP—This step maps the results from Step 1.2 to a Record, which the business services will process in the next step. It also cleans up the pipeline by dropping unneeded variables.

STEP 2 **Perform business logic.** This sequence invokes business services that process the data extracted by Step 1. In this example, the business services use the data to calculate the cost of a printing job. They return the cost in a variable named *qCost*.

STEP 3 **Compose SOAP response message.** This sequence generates the response message that carries the results back to the client. It produces a SOAP message structured as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xsd="http://www.w3.org/1999/XMLSchema"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance">
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"

<SOAP-ENV:Header>

    <AUDIT:msgInfo xmlns:AUDIT="http://www.accumon.com/msgTracker">

      <msgType>quoteResp</msgType>

      <sender>http:www.exprint.com/RFQ</sender>

      <docID>RFQ-0.41</docID>

      <tStamp>20010731.155453.454</tStamp>

    </AUDIT:msgInfo>

</SOAP-ENV:Header>

<SOAP-ENV:Body>
```

The service generates a header entry called 'msgInfo'...

```
      <RFQ:quoteResp xmlns:RFQ="http://www.exprint.com/orderSys">

      <acct>1417-A199-0404-5POLY</acct>

      <jobSpecs>

        <stock>30F-SIL</stock>

        <ink>P440</ink>
```

...and a body entry called 'quoteResp'

```
        <copies>5000</copies>

      </jobSpecs>

      <qCost>2850</qCost>

    </RFQ:quoteResp>

</SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

To produce this response message, the service executes the following steps:

Step	Description
3.1	This sequence executes the following services to add a body entry to *soapResponseData*.

Step	Description
3.1.1	**recordToDocument**—This service generates a Record called *RFQ:quoteResp*, which describes the body entry to be inserted into *soapResponseData* and converts the Record to XML. Note that the Record includes a field called *@xmlns:RFQ*, which sets the namespace attribute in the resulting XML.
3.1.2	**stringToDocument**—This step converts the XML to a node object. (Recall that to add a body entry to *soapResponseData*, you must place the entry in the pipeline as a node.)

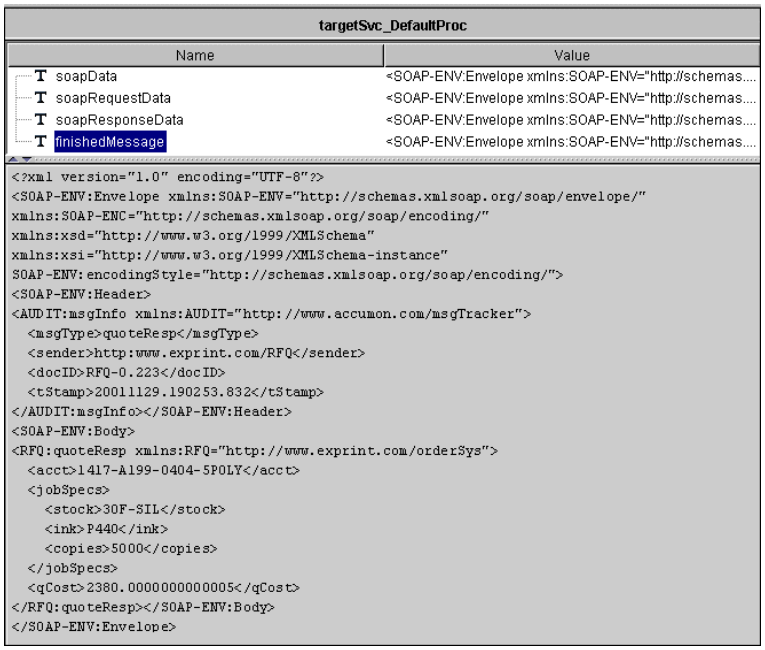
3.1.3 addBodyEntry—This step adds the body entry to *soapResponseData*.

3.2 This sequence executes the following services to add a header entry to *soapResponseData*.

Step	Description
3.2.1	recordToDocument—This step creates a Record called *AUDIT:msgInfo*, which describes the header entry to be inserted into *soapResponseData* and converts the Record to XML. Note that the Record contains a field called *@xmlns:AUDIT*, which sets the namespace attribute in the resulting XML.
3.2.2	stringToDocument—This step converts the XML to a node. (Recall that to add a header entry to *soapResponseData*, you must place the entry in the pipeline as a node.)
3.2.3	addHeaderEntry—This step adds the header entry to *soapResponseData*.

3.3 FOR DEBUG ONLY. This step converts the contents of *soapResponseData* to a String using the soapDataToString service. This allows you to examine the finished message with Developer, which is useful during testing and debugging. You would not include this step in a production service.

If you examine the contents of *finishedMessage* on the Results tab, you will see a SOAP message similar to the one below. If this service were invoked through the default processor, the message handler would send this message to the client.



Chapter 6: Using the SOAP RPC Processor

What is the RPC Processor?

The RPC processor is the subsystem that the SAP BC server uses to receive and process SOAP remote procedure calls (RPCs). The RPC processor is invoked when the message handler receives a URL with a process directive set to “rpc,” as shown below:

```
http://rubicon:5555/soap/rpc
```

To submit a SOAP RPC message to the SAP BC server, specify the “rpc” directive in the URL

What Does a SOAP RPC Message Look Like?

A SOAP RPC message is an ordinary SOAP message whose body conveys a remote procedure call. The body of an RPC message has a standard structure that specifies the name of the requested procedure and its input parameters.

- The qualified name (QName) of the first element within the body identifies the procedure that the client is calling. On the SAP BC server, the QName specifies the universal name of the service to be executed.

- The children within the first element represent the input parameters for the requested procedure.

For example, the following message body contains a call to a procedure named `GL:journalEntry`. It also conveys four input parameters to this procedure: *Id*, *date*, *amt*, and *acct*.

Basic structure of a SOAP remote procedure call

The QName of the first element specifies the requested procedure...

...and its children supply the input parameters

```
.
.
.
<SOAP-ENV:Body>
   <GL:journalEntry xmlns:GL="http://www.expt.com/GL">
      <Id>2398</Id>
      <date>03/15/2000</date>
      <amt>237.50</amt>
      <acct>Cash</acct>
   </GL:journalEntry>
</SOAP-ENV:Body>
.
.
.
```

For more information about the structure of a SOAP RPC message, see the *Simple Object Access Protocol (SOAP) 1.1 - W3C Note 08 May 2000* at http://www.w3.org/TR/SOAP/.

QNames and Input Parameters

When an RPC processor receives a SOAP remote procedure call, it must:

- Resolve the QName in the body in the message with the name of a procedure in the local environment, and…
- Render the input parameters in a form that the target procedure can use.

On the SAP BC server, the RPC processor performs this work by:

- Matching the QName to the universal name of a service, and…
- Engaging the message coder to convert the XML-encoded input parameters to a set of Java objects that the service can consume.

What Does a Results Message Look Like?

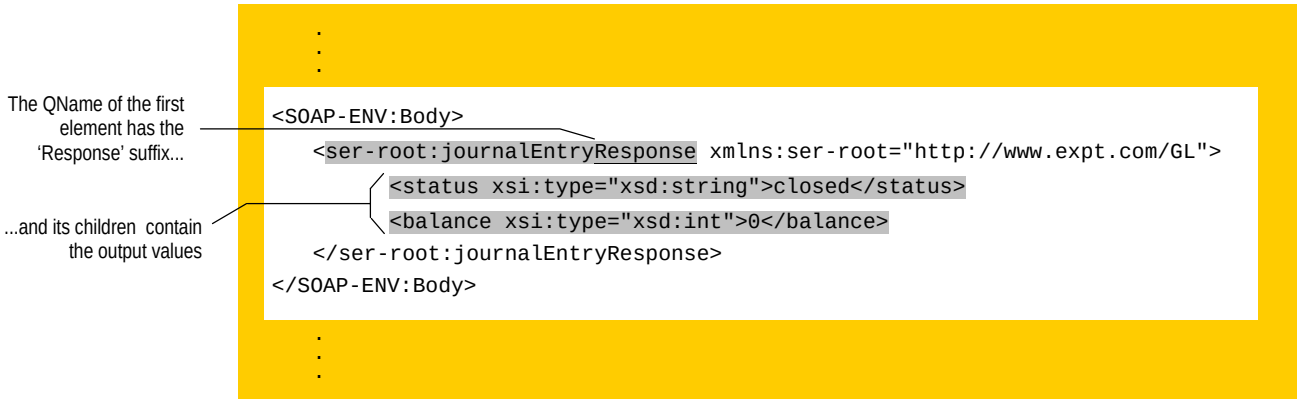
Results are also carried in the body of the message and are formatted in a standard way.

– The qualified name (QName) of the first element within the body of the message is the same as the QName of the first in the original request, except that the suffix “Response” is appended to it.

– The children of this element contain the output values from the requested procedure.

The following is an example of a response message that contains two output parameters: *status* and *balance*.

Basic structure of a SOAP RPC Results message



The results use the same namespace as the original request. By default, the SAP BC Server uses the prefix “ser-root” for this namespace.

For more information about the structure of a SOAP RPC response, see the *Simple Object Access Protocol (SOAP) 1.1 - W3C Note 08 May 2000* at http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

Behavior of the RPC Processor

When the RPC processor receives a SOAP message, the following occurs:

1 The processor searches for the target service—that is, it locates the service whose universal name (explicit or implicit) matches the QName of the first element in the body of the message.

For example, if the RPC processor received the following request, it would fetch the service whose universal name matches the namespace name http://www/expt.com/GL/ and the local name journalEntry.

```
   .
   .
   .
<SOAP-ENV:Body>
   <GL:journalEntry xmlns:GL="http://www.expt.com/GL/">
      <date>03/15/2000</date>
      <amt>237.50</amt>
      <acct>Cash</acct>
   </GL:journalEntry>
</SOAP-ENV:Body>
   .
   .
   .
```

If the processor cannot locate the target service, it returns a SOAP fault that contains the following error message:

```
[B2BPCKG.0088.9122] Service namespaceName:localName does not exist
```

For more information about this error, see Appendix A *SOAP Faults Returned by the SAP BC server*.

2 If the processor finds the requested service, the message coder decodes the input parameters—that is, it extracts the XML-encoded input parameters from the body of the message and converts them into Java objects. For more information about the decoding process, see “Decoding the Input Parameters” on page 50.

3 The processor invokes the target service and passes the decoded parameters to the service via the pipeline.

4 When the service ends, the message coder encodes the results—that is, it converts the parameters named in the output signature to a set of XML-encoded values. For more information about the encoding process, see “Encoding the Output Parameters" on page 54.

5 The processor assembles the SOAP response message and passes it back to the message handler.

How the RPC Processor Controls Access to Target Services

Access to services invoked via the SOAP RPC is controlled by the server's Access Control Lists (ACLs). When a client submits a request to the RPC processor, its credentials are checked against the ACL for the target service it requests. If the client is not authorized to execute the requested service, the request is rejected.

For information about creating and assigning ACLs to services, see the *SAP BC Server Administrator's Guide*.

Building Target Services for the RPC Processor

Any service on the SAP BC server can function as a target service of a SOAP remote procedure call, providing it satisfies the following criteria:

- The service's implicit or explicit universal name matches the QName that the client will submit in the remote procedure call. For information about assigning universal names to services, see "To assign, edit, or view an explicit universal name" on page 29.
- The names of the parameters in the target service's input signature match the names of the parameters that the client will pass in the remote procedure call. (Keep in mind that these names are case sensitive.)
- The service's output signature defines *all* of the values that are to be returned to the client.
- The service's ACL specifies the groups of users who are authorized to execute it.

To ensure that parameters are properly encoded, decoded, and validated, you should also ensure that:

- The Content Type property for each String-based parameter in the output signature specifies the data type that is to be assigned when they are XML-encoded. (See "The Message Coder and the RPC Processor" on page 50 for information about how the Content Type property is used to encode and decode parameter values.)
- The service's Validate Input and/or Validate Output settings are enabled if you want the input and/or output values to be validated at run time.

Error Handling

If a target service throws an exception while it is processing, the message handler automatically returns a SOAP fault to the client. Depending on the type of error that occurs, the SOAP fault may include a "detail" element, which provides specific

information about the exception. This element will include a stackTrace element if the client is a member of the Developers or Administrators user group.

For more information about SOAP faults, see Appendix A *SOAP Faults Returned by the SAP BC server*.

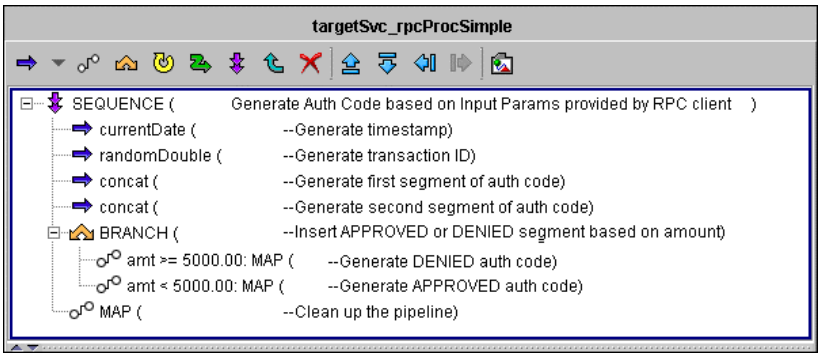
Example—Target Service for the RPC Processor

The following is an example of a very simple target service that is suitable for calling with SOAP RPC. This service takes three String parameters (*acct*, *amt*, and *loc*) and returns one String parameter (*authCode*).

This service is located in sample.soap:targetSvc_rpcProcSimple. You can call it via SOAP RPC by executing the client service, sample.soap:buildRPC_SendHTTPSimple, from Developer. When you run the client service in Developer, it will prompt you for the following values:

For this input parameter...	Enter...
acct	Any string of characters.
amt	A decimal value, such as 150.75 or 15.00. (Omit commas from large values; otherwise, the value will fail validation.)
loc	Any string of characters.
userName	A user name that belongs to the Developers ACL.
password	The password for the user name that you entered in *userName*.

A simple target service

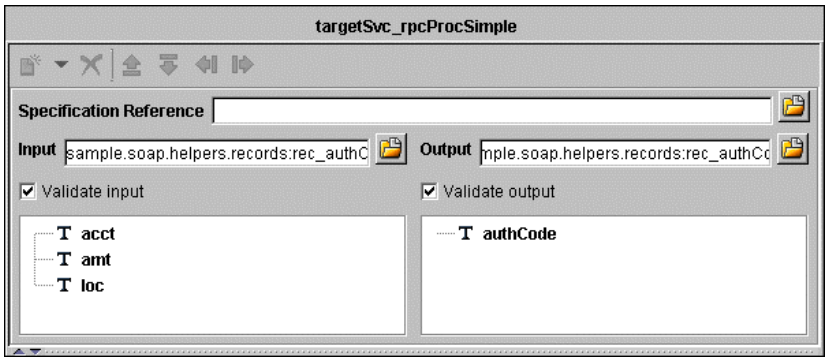


This flow simulates a service that issues an authorization code based on the value in the input parameter *amt*. Because a SOAP RPC target does not require any special logic, the service does not perform any work that is SOAP-specific. You could call this service from any type of client. It contains nothing that restricts its use to SOAP remote procedure calls.

The characteristics that are significant if this service is to be used as a SOAP RPC target are its *signature* and its *universal name*.

- The Signature. The signature for this service defines three input variables (*acct*, *amt*, and *loc*) and one output variable (*authCode*). Note that the Validate input and Validate output settings are enabled in this example to ensure that the input and output values are validated at run time.

  Signature for the sample target service

  The signature is critical because it determines how variables are decoded and encoded at run time. In this example, the input variables are declared as Strings, which means that the message coder will render them as Strings regardless of the data types specified in the XML.

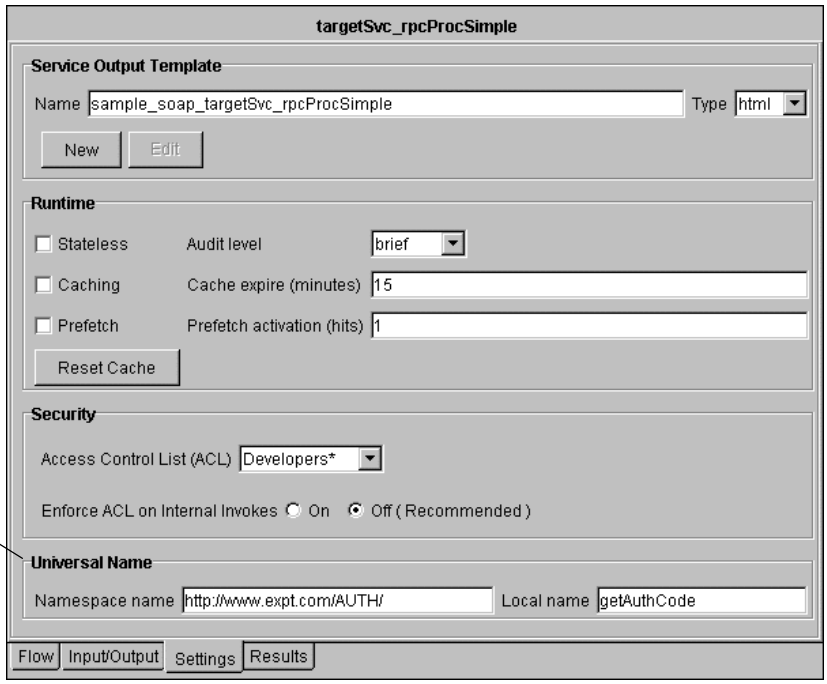
  If you examine the properties for these parameters with the Developer, you will see that the Content Type and Field Must Exist at Runtime properties are specified. These will be used to validate the values that the client provides.

The signature declares one output variable called *authCode*. Because this is the only variable declared in the output signature, it will be the only parameter returned to the client.

If you examine the properties for *authCode*, you will see that its Content Type is set to `string {http://www.w3.org/2001/XMLSchema}`. This tells the message coder to set the type attribute to "xsd:string" when it encodes this parameter.

- The Universal Name. An explicit universal name has been specified for this service on the Settings tab. The universal name determines the QName will cause this service to execute. In this example, the service will be triggered when a client submits a SOAP remote procedure call whose QName is composed of the namespace name `http://www.expt.com/AUTH` and the local name `getAuthCode`.

Settings tab for the sample target service



The universal name is on the Settings tab

The Message Coder and the RPC Processor

The message coder is the subsystem that decodes and encodes the input and output parameters associated with a SOAP remote procedure call. Its role is to turn a set of XML-encoded values into a pipeline of the appropriate data types (the *decoding* process) and, conversely, to turn objects in the pipeline to a set of XML-encoded values (the *encoding* process).

Note: The encoding and decoding process is sometimes referred to as “serializing and deserializing” or “marshalling and unmarshalling.”

Encoding/Decoding Rules

To encode and decode the parameters in a SOAP RPC message, the message coder follows the SOAP 1.1. encoding style, which is specified by the following SOAP encodingStyle attribute:

```
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
```

This attributes specifies the set of rules that govern how the input and output parameters of a SOAP RPC message are to be represented in XML. These rules are defined in the “SOAP Encoding” section of the *Simple Object Access Protocol (SOAP) 1.1 W3C Note* at http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

Note: The SAP BC server supports only the SOAP 1.1. encoding style.

Decoding the Input Parameters

The *decoding process* is the process of converting a set of XML-encoded values to an IData object (a pipeline).

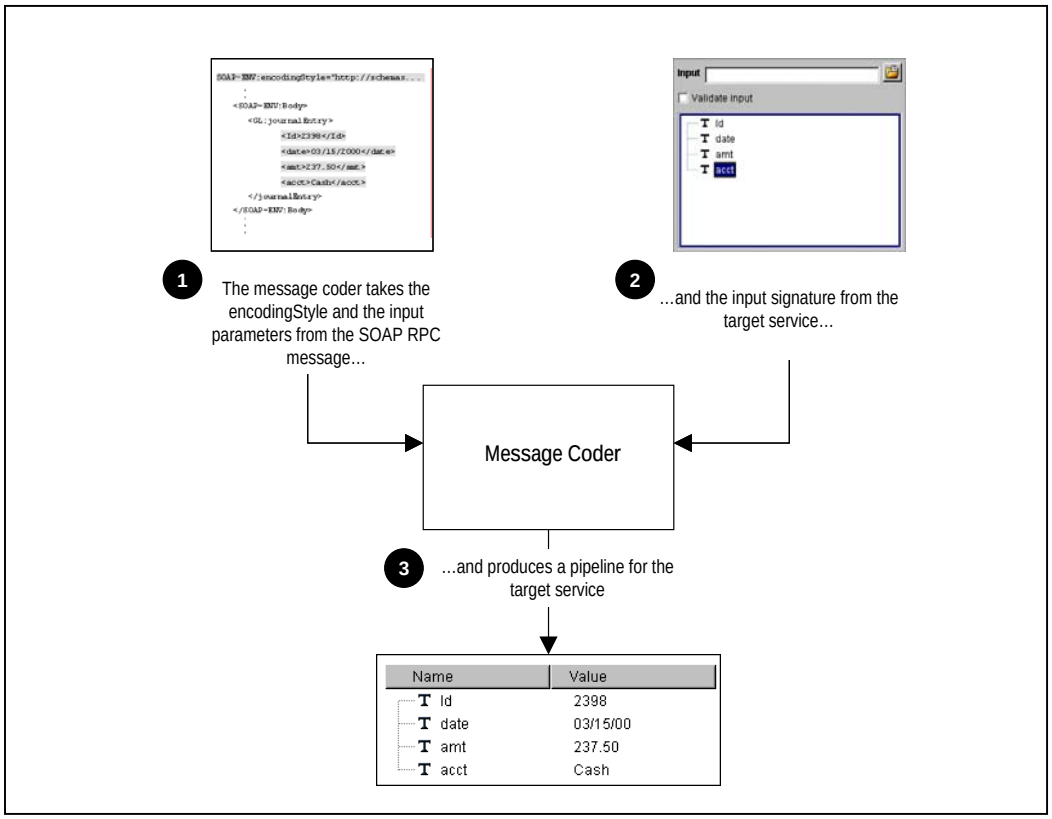
When the RPC processor receives a valid procedure call, it engages the message coder, which decodes the input parameters and produces the pipeline that is passed to the target service.

Note: The message coder is also engaged by the SOAP RPC client to decode the output parameters returned by a SOAP remote procedure call. For information about this use of the message coder, see “The Message Coder and the RPC Client” on page 83.

Transforming Input Parameters into a Pipeline

To produce a pipeline for the target service, the message coder uses the input parameters from the body of the message and the input signature from the target service.

Producing a pipeline from the input parameters in a remote procedure call



During the decoding process, the message coder matches the names of the parameters passed by the client with the names of the parameters defined in the input signature. It uses the input signature to determine how to render the parameters in the pipeline.

- If the input signature declares a parameter as a String, the parameter is rendered as a Java String object, regardless of the data type declared in the XML.

- If the input signature declares a parameter as an Object, the value is rendered according to the data type declared in the XML (for a list of XML data types and the Java classes to which they are converted, see Appendix B *Encoding/Decoding Data-Type Mapping*). If an XML element does not declare its type, the parameter is rendered as a String.

The following table shows how the message coder would decode an input parameter named *amt* given various data-type declarations and input signatures. Note that when the input signature defines the parameter as a String, the message coder *always* produces a String object, regardless of the data type declared in the XML.

When XML data type is...	and Input Signature is..	The Message Coder produces...	
`<amt xsi:type="xsd:decimal">500.00</amt>`	T amt	T amt	
`<amt xsi:type="xsd:decimal">500.00</amt>`	* amt	* amt	of class java.math.BigDecimal
`<amt xsi:type="xsd:float">500.00</amt>`	T amt	T amt	
`<amt xsi:type="xsd:float">500.00</amt>`	* amt	* amt	of class java.lang.Float
`<amt>500.00</amt>`	T amt	T amt	
`<amt>500.00</amt>`	* amt	T amt	

Parameters that are not in the Input Signature

The message coder decodes every input parameter that a client submits—even those that do not appear in the input signature. When a client submits parameters that are not declared in the input signature, the message coder renders them as Strings.

Decoding Complex Structures and Arrays

For complex data types (XML elements that contain child elements), the message coder produces Records (IData objects) in the pipeline.

The message coder creates arrays of elements (i.e., String Lists, Record Lists, Object lists) for elements that are properly encoded as SOAP arrays in the SOAP message. For more information about the how arrays are encoded in a SOAP message, see the "Arrays" section in the *Simple Object Access Protocol (SOAP) 1.1 W3C Note* at http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

Decoding Multi-Referenced Parameters

The message coder can decode parameters that are referenced via the href attribute, providing the reference meets the following conditions:

- The value referenced by the href attribute appears within the body of the SOAP message. The message coder cannot resolve external references (references that point to resources outside of the message) or references that point to elements outside of the Body element.
- The element containing the href attribute appears *after* the value that it references. For example, the message coder can successfully decode the following reference:

This reference can be decoded, because the value of *id1* is assigned before it is referenced.

```
.
.
.
<cDate id="id1">03/15/2000</cDate>
<amt id="id2">300</amt>
<oDate href="#id1"></oDate>
.
```

```
          .
          .
```

But it cannot decode the following:

```
          .
          .
          .
<oDate href="#id1"></oDate>
<amt id="id2">300</amt?
<cDate id="id1">03/15/2000</cDate>
          .
          .
```

This reference cannot be decoded, because the value of *id1* is assigned after it is referenced.

Note: If the message coder cannot resolve a reference, it generates a null object for that parameter. It also reports the problem in server.log if the server is running at debug level 5 or higher.

For referenced elements, the message coder puts a single copy of the source data in the pipeline and generates references to it.

Decoding and Validation

Be aware that the message coder does not validate the input parameters that it decodes. For example, it does not verify that an element of type `xsi:type=decimal` actually contains a numeric value. It simply renders the element as a java.lang.String object or a java.math.BigDecimal object, depending on the data type specified in the signature of the target service.

Additionally, the message coder does not verify that the set of parameters it receives from the client matches the parameters in the input signature in name or number. If a client includes parameters that are not in the input signature, the message coder includes them in the pipeline anyway (as Strings). If a parameter is declared in the input signature, but is not supplied by the client, that parameter is omitted from the pipeline.

Validating Input Parameters

If you want to validate the input parameters that a client submits via SOAP RPC, you use the server’s normal validation mechanisms to impose data validation on the target service. For example, to ensure that clients provide a non-negative integer for a parameter named *hours*, you would:

– Constrain the *hours* parameter by setting its Content Type property to:

```
nonNegativeInteger {http://www.w3.org/2001/XMLSchema}
```

– Enable the parameter's Field Must Exist at Runtime property, and

– Set the Validate Input option on the target service’s Input/Output tab.

These settings would cause the server to throw an exception if the client omitted the *hours* parameter or submitted a value such as ".5" or "twenty" rather than a non-negative integer. For more information about validating data at run time, see the *SAP BCDeveloper Guide*.

Encoding the Output Parameters

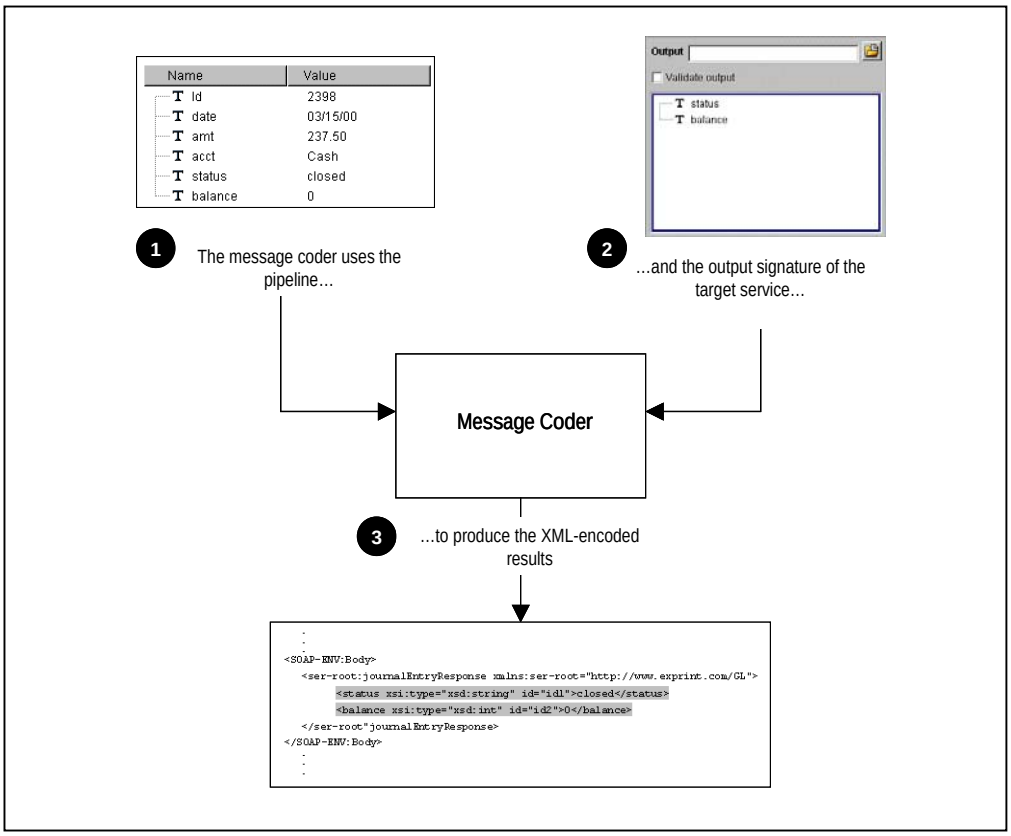
The *encoding process* is the process of converting a Java object to an XML-encoded value. The SOAP RPC processor engages the message coder to encode the results of services that are invoked via a SOAP remote procedure call.

Note: The message coder is also engaged by the SOAP RPC client to encode the input parameters for an outbound remote procedure call. For information about this use of the message coder, see "The Message Coder and the RPC Client" on page 83.

Transforming Output Parameters to XML

When the message coder encodes the results of a remote procedure call, it uses both the values in the pipeline and the output signature of the target service.

Encoding the output parameters from a remote procedure call



During the encoding process, the message coder converts the parameters defined in the output signature to XML-encoded values.

- If the output signature declares a parameter as a String, the value of the parameter is encoded according to its Content Type property. For example, if Content Type were `nonNegativeInteger {http://www.w3.org/2001/XMLSchema}`, the value would be encoded as a `xsi:type="xsd:nonNegativeInteger"`. If Content Type is not specified, the message coder encodes the value as a string (as if the parameter were Content Type `string {http://www.w3.org/2001/XMLSchema}`).

- If the output signature declares a parameter as an Object, the value of the parameter is encoded according to its underlying Java class. For example, if the object were a java.lang.Boolean, the message coder would declare it as `xsi:type="xsd:boolean"` in the resulting XML. For a list of recognized Java classes and the XML Schema data types to which they are converted, see Appendix B *Encoding/Decoding Data-Type Mapping*. If the message coder does not recognize the underlying class, it encodes the parameter as a string (it uses the object's toString( ) method to produce the parameter's value).

Important! It is important to understand that the message coder encodes *only* those parameters defined in the service's output signature. *It does not encode the entire pipeline.* Therefore, if a service has no output signature, no values are returned to the client. Additionally, this means that if any of the original input values need to be returned to the client, those values must be included in the output signature.

Encoding Complex Structures and Arrays

The message coder encodes Records (IData objects) as complex data types.

The message coder encodes Record Lists, String Lists, and Object Lists as SOAP arrays. For more information about the how arrays are encoded in SOAP messages, see the "Arrays" section in the *Simple Object Access Protocol (SOAP) 1.1 W3C Note* at http://www.w3.org/TR/2000/NOTE-SOAP-20000508.

Encoding Multi-Referenced Elements

By default, the message coder encodes output parameters as independent elements, regardless of whether they reference the same underlying objects in the pipeline. However, this behavior can be modified with the `watt.server.SOAP.useMultiReference` parameter.

When `watt.server.SOAP.useMultiReference` is true, the message coder will generate the appropriate `id` and href attributes for parameters that reference the same underlying data. For example, if the parameters *cDate* and *oDate* refer to the same object, the message coder encodes them like this if `watt.server.SOAP.useMultiReference=true`:

```
.
.
.
<cDate id="id1">03/15/2000</cDate>
<amt id="id2">300</amt>
```

The *oDate* parameter is encoded as a reference

```
<oDate href="#id1"></oDate>
<acct id=id4>cash</acct>
 .
 .
 .
```

And encodes them like this if `watt.server.SOAP.useMultiReference=false` or is not set at all (which is the server's installed behavior):

```
 .
 .
 .
<cDate>03/15/2000</cDate>
<amt>300</amt>
<oDate>03/15/2000</oDate>
<acct>cash</acct>
 .
 .
 .
```

The *oDate* parameter is encoded as an independent element

For more information about `watt.server.SOAP.useMultiReference` and other SOAP-related server parameters, see Appendix C *SOAP-Related Server Parameters*. For more information about setting server parameters, see the *SAP BC Server Administrator's Guide*.

Chapter 7: Creating Custom SOAP Processors

What is a Custom SOAP Processor?

If the SOAP processors provided by SAP do not suit your needs, you can create your own customized SOAP processor and register it on the SAP BC server. For example, you might create a custom processor that delegates messages based on the value of a particular header entry, or you might create a customer processor that drops messages into queues based on the type of documents they carry.

Implementing a custom SOAP processor on the SAP BC server involves two basic steps:

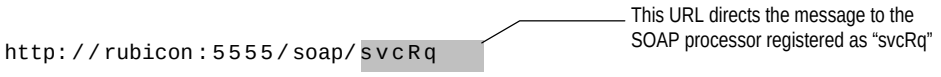
1 First, build a service that acts as a SOAP processor. You can code this service to operate on SOAP messages in whatever manner your solution requires. However, it must accept a *soapRequestData* object and a *soapResponseData* object as input, and must return a *soapResponseData* object as output. For information about this step, see "Building a Custom SOAP Processor" on page 59.

2 Second, register the service on the SAP BC server as a SOAP processor. You do this to associate the service with a specific process directive (a name that is unique among all SOAP processors on the server) and make it accessible to the SOAP message handler. For information about registering a service as a SOAP processor, see "Registering a SOAP Processor" on page 65.

Accessing a Custom SOAP Processor

To direct a message to a custom processor, a client specifies the processor's "process directive" in the URL. The process directive is the name under which the custom processor is registered with the SOAP message handler. For example, the following URL would invoke the custom processor registered as "inbox."

The following URL would invoke the custom processor registered as "svcRq."



Important! Process directives are case sensitive. The directive specified in the URL must match exactly the process directive as it is defined on the SAP BC Server.

If the URL specifies a process directive that is not registered on the server, the SOAP message handler returns a SOAP fault containing the following error message:

```
[B2BPCKG.0088.9123] Requested SOAP processor directiveName is not
                    registered on this server.
```

For information about SOAP faults, see Appendix A *SOAP Faults Returned by the SAP BC server*.

Building a Custom SOAP Processor

A SOAP processor is a service that operates on SOAP messages. It can be any type of service and can contain any kind of logic that your solution requires.

Inputs and Outputs

When a the SOAP message handler invokes a custom processor, it passes two input parameters to that processor:

- *soapRequestData*, an object containing the entire SOAP message
- *soapResponseData*, an object containing an empty SOAP message

When the custom processor ends or exits, the SOAP message handler returns the contents of *soapResponseData* to the client (unless the processor throws an exception, in which case the SOAP message handler generates a SOAP fault and returns it to the client).

How to Create a Custom SOAP Processor

The following describes the general steps you take to create a custom processor.

1 Create a new service that has the following signature:

Inputs:	*soapRequestData* (of type Object) *soapResponseData* (of type Object)
Outputs:	*soapResponseData* (of type Object)

You can use pub.soap.utils:requestResponseSpec to specify the inputs and outputs for the service.

2 Code the service to process SOAP messages in the way you need. The following describes several tasks that a custom processor typically executes. However, the behavior of your processor will depend entirely on the needs of your solution. It might include all, some, or none of the following:

- Extracting data from the SOAP request message. To extract information from a SOAP message, you use the data-retrieval utilities such as getBody and getHeader. (For a complete list of the data-retrieval services, see page 73). These services retrieve a specified element from the *soapRequestData* and return the requested element as a *node* (or an array of *nodes*). To extract data from the returned *node*, you query it with the pub.web:queryDocument service.

Important! Be aware that you cannot query the *soapRequestData* object directly. To extract information from *soapRequestData* (or similar SOAP objects, such as *soapData* and *soapResponseData*), you must use one of the data-retrieval services to extract an element of the message (e.g., the header, the body, or the entire envelope) and query the resulting node.

- Invoking services to process the data extracted from the message. After extracting the data with which you want to work, you can map it to the appropriate variables (if necessary) and pass it to services that process it in some way. (To ensure that the data you have extracted is in the correct format, you might want to validate it against a schema using pub.schema:validate or make sure that the service to which you pass the data performs data validation on its input parameters.)

- Invoking services based on a particular QName in the SOAP request message. If your SOAP messages use a qualified name (QName) to specify a target service on the SAP BC Server, you can use pub.soap.utils:getQName (which resides in the WmPublic package) to extract the element's QName from the message and then use pub.universalName:find (which resides in the WmPublic package) to locate the service associated with that name.

Note: To invoke the service associated with a QName, you will need to create a Java service that performs a doInvoke of the service returned by pub.universalName:find. See the com.wm.app.b2b.server package in the *SAP BC Server Java API Reference* for information about the doInvoke method. This method is a member of the Service class.

3 Use the SOAP message-composition services to populate *soapResponseData*. The message-composition services are services (such as addHeaderEntry and addBodyEntry) that you use to add content to the empty message in *soapResponseData*. (For a complete list of message-composition services, see page 69.)

The message-composition services require a *node* representation of the header entry or body entry that you want to add to the message. You can generate a *node*, using services such as pub.web:recordToDocument and pub.web:stringToDocument. For an example of how to do this, see Step 3 in the sample code shown on page 62.

4 When you finish coding the service, register it as a SOAP processor. This step associates the service with a specific process directive and makes it accessible to the SOAP message handler. For information about registering a service as a SOAP processor, see "Registering a SOAP Processor" on page 65.

Error Handling

If your SOAP processor (or any service that it calls) throws an exception, the SOAP message handler automatically returns the following SOAP fault to the client. The fault includes a "detail" element that provides specific information about the exception. This element will include a stackTrace element if the client is a member of the Developers or Administrators user group.

```
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsd="http://www.w3.org/1999/XMLSchema"
   xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance">
```

Note: The SOAP message returned to the client when an exception occurs contains only the SOAP fault. It does not include any message content (e.g., header entries, body entries) that the target service may have inserted into *soapResponseData* before the exception occurred.

Returning Your Own SOAP Faults

Although the SOAP message handler automatically returns a SOAP fault when a custom processor throws an exception, there may be times when you want your custom processor to return a SOAP fault. For example, you might want to return a fault if the client omits a required piece of data or if a requested resource is not available. In these situations, you can choose to do either of the following:

- Code your processor to detect the error condition and throw an exception. This will cause the message handler to return the standard SOAP fault code shown above. If you include a detailed error message when you throw the exception, that message will be included in the detail element within the SOAP fault.

—OR—

- Code the processor to detect the error, but instead of throwing an exception, compose your own SOAP fault in *soapResponseData* and then exit normally. This will cause the message handler to simply return the contents of *soapResponseData* (which contains the SOAP fault that your processor generated) to the client. If you choose this approach, you must be thoroughly familiar with the SOAP specification regarding fault codes.

Important! The SOAP specification states explicitly that the recipient of a SOAP message *must* return a SOAP fault anytime it fails to process the submitted message successfully.

For more information about SOAP faults, see Appendix A of this document (*SOAP Faults Returned by the SAP BC server*) and the *Simple Object Access Protocol (SOAP) 1.1 - W3C Note 08 May 2000* at http://www.w3.org/TR/2000/NOTE-SOAP-20000508/#_Toc478383507.

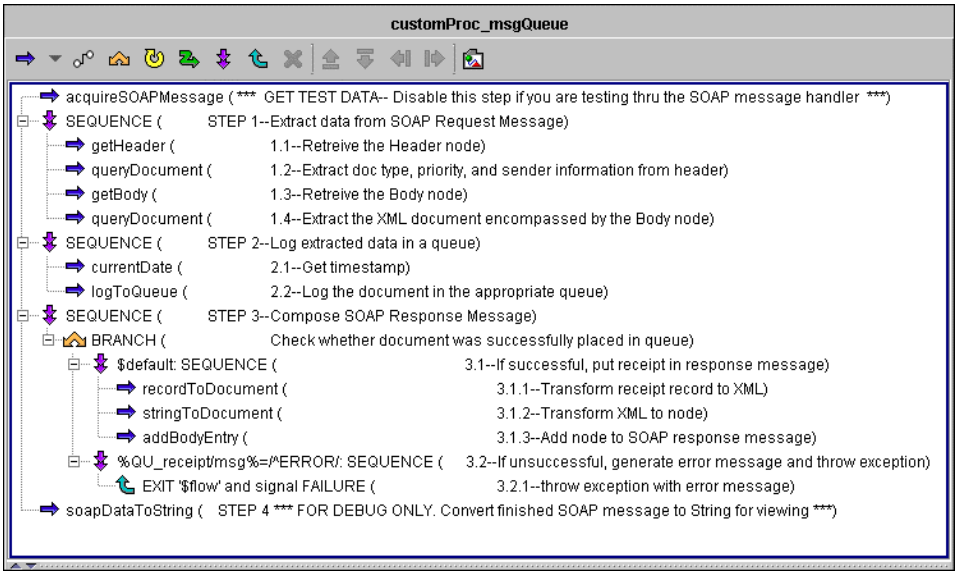
Example—Custom Processor

The following is an example of a SOAP processor that drops the body of the message (along with several other pieces of control information that it extracts from the header) into a queue based on the value of a certain header entry. If the document is successfully logged into the queue, the processor returns an acknowledgment. Otherwise, it generates an error message and throws an exception so that the client receives a SOAP fault.

This example is located in sample.soap:customProc_msgQueue in the WmSample package. You might want to open this example with Developer to see how the pipeline is mapped between steps.

Note: If you want to execute this service from Developer, enable the acquireSOAPMessage step at the top of the flow. This step generates *soapRequestData* and *soapResponseData* objects, which simulate the pipeline that this service would receive from the SOAP message handler. If you want to execute this service via the SOAP message handler (that is, from a client), disable acquireSOAPMessage.

Custom processor that drops messages into queues



#	Description
STEP 1	Extract data from SOAP request message. This sequence retrieves required pieces of information from the message by extracting the header and body elements from *soapRequestData* and querying the results. This example expects a SOAP message that is structured as follows:

This service pulls certain control information from the header...

...and extracts the document carried in the body of the message and drops it in a queue

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xsd="http://www.w3.org/1999/XMLSchema"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">

<SOAP-ENV:Header>
   <EXP:postmark xmlns:EXP="http://www.exprint.com/inbox">
     <docType>RFQ</docType>
     <priority>05</priority>
     <sender>http://www.polymfg.com/PURCH/A30155G</sender>
     <tStamp>20011129.161434206</tStamp>
   </EXP:postmark>
</SOAP-ENV:Header>

<SOAP-ENV:Body>
   <RFQ:quoteReq xmlns:RFQ="http://www.exprint.com/orderSys">
     <acct>1417-A199-0404-5POLY</acct>
     <jobSpecs>
       <copies>5000</copies>
       <stock>30F-SIL</stock>
       <ink>P440</ink>
     </jobSpecs>
   </RFQ:quoteReq>
</SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

To extract information from the message, this service executes the following steps:

Step	Description
1.1	getHeader—This step retrieves the header from *soapRequestData*. It returns a node that represents the *entire* header element (that is, from <Header> to </Header>).
1.2	queryDocument—This step extracts specific pieces of control information from the header by executing the following XQL queries against the node returned by Step 1.1.

Var Name	XQL Query
QU_docType	`/PMK:postmark/docType/text()`
QU_priority	`/PMK:postmark/priority/text()`
QU_sender	`/PMK:postmark/sender/text()`
QU_timeSent	`/PMK:postmark/tStamp/text()`

If you examine the queryDocuement step with Developer, you will see that it also executes the following query, which extracts the entire Header node as a String:

Var Name	XQL Query
wholeNode	`/source()`

This query is included for debugging purposes. It allows you to examine the raw XML associated with the Header node. If you were to put this service into production, you would omit this query.

1.3 getBody—This step retrieves the body from *soapRequestData*. It returns a node that represents the *entire* body element (that is, from <Body> to </Body>).

1.4 queryDocument—This step extracts the contents of the body element by executing the following XQL query against the node returned in Step 1.3.

Var Name	XQL Query
QU_doc	`//source()`

STEP 2 Log extracted data in a queue. This sequence generates a timestamp and drops the information extracted by Step 1 into a queue (in this example, this action is simulated by a MAP step). It produces a "receipt" that reports the name of the queue into which the message was dropped. If the message is rejected by the queue, the receipt will contain an error message.

STEP 3 Compose SOAP Response Message. This sequence checks the result generated by Step 2. If Step 2 returns an error, this step generates an error message and throws an exception (which causes a SOAP fault to be returned to the client). Otherwise, it generates a response message that is structured as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xsd="http://www.w3.org/1999/XMLSchema"

xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">

<SOAP-ENV:Header>

</SOAP-ENV:Header>

<SOAP-ENV:Body>

   <QU_receipt>

     <msg>Document RFQ.20013513.163535.0895.A8F7 successfully placed in
          queue RFQ</msg>

     <entryTime>20013513.163535.0895</entryTime>

     <qID>RFQ.20013513.163535.0895.A8F7</qID>

   </QU_receipt>

</SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

The receipt from the queue is returned in the body of the SOAP response message

To produce the response message, this service executes the following steps:

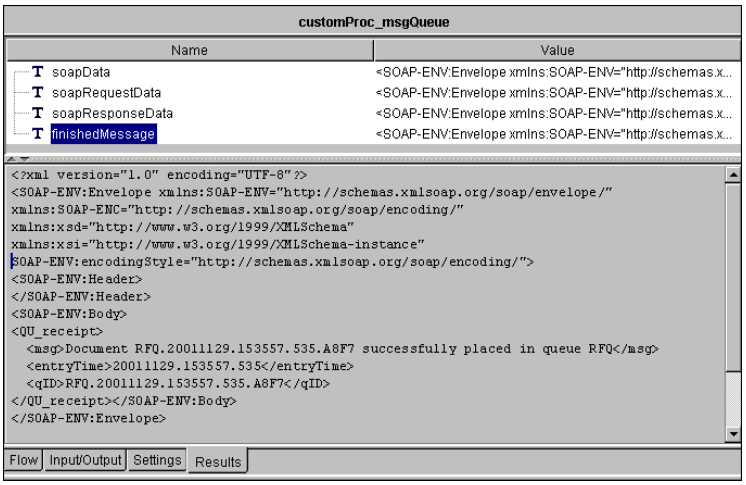
Step	Description
3.1.1	recordToDocument—This service converts the receipt record to XML.
3.1.2	stringToDocument—This step converts the XML to a *node* object. (Recall that to add a body entry to *soapResponseData*, you must place the entry in the pipeline as a node.)

3.1.3 addBodyEntry—This step adds the body entry to *soapResponseData*.

STEP 4

FOR DEBUG ONLY. This step converts the contents of *soapResponseData* to a String using the soapDataToString service. This allows you to examine the finished message with Developer, which is useful during testing and debugging. You would not include this step in a production service.

If you execute this service and examine the contents of *finishedMessage* on the Results tab, you will see a SOAP message similar to the following:



Registering a SOAP Processor

To make a custom SOAP processor available for use, you must register it on the SAP BC server. This action associates your processor with a particular directive and makes it known to the SOAP message handler.

To register a SOAP processor, you execute the service, pub.soap.processor:registerProcessor. To ensure that your SOAP processor is registered whenever the server is restarted, we recommend that you include pub.soap.processor:registerProcessor in a startup service.

Note: If you register your SOAP processor using a start-up service, make sure that the package in which the start-up service resides designates the WmPublic package as a dependency. This will instruct the server to load WmPublic before it loads your package and ensure that pub.soap.processor:registerProcessor is accessible when your start-up service executes.

How to Register a SOAP Processor

The following describes the general steps you take to register a SOAP processor.

1 Create a flow service that invokes pub.soap.processor:registerProcessor. Set the following input variables for this service:

Set...	To...
directive	A String specifying the process directive that you want to assign to the processor. Note: Use only letters, digits, or the characters `-_.!~*'( )` in the name you specify in *directive*.
svcName	A String specifying the fully qualified name of the service that you are registering as a SOAP processor.
descriptiveName	Optional. A String that contains a descriptive comment for this SOAP processor. This comment is shown when you run the utility service pub.soap.processor:list to get a list of the registered SOAP processors.

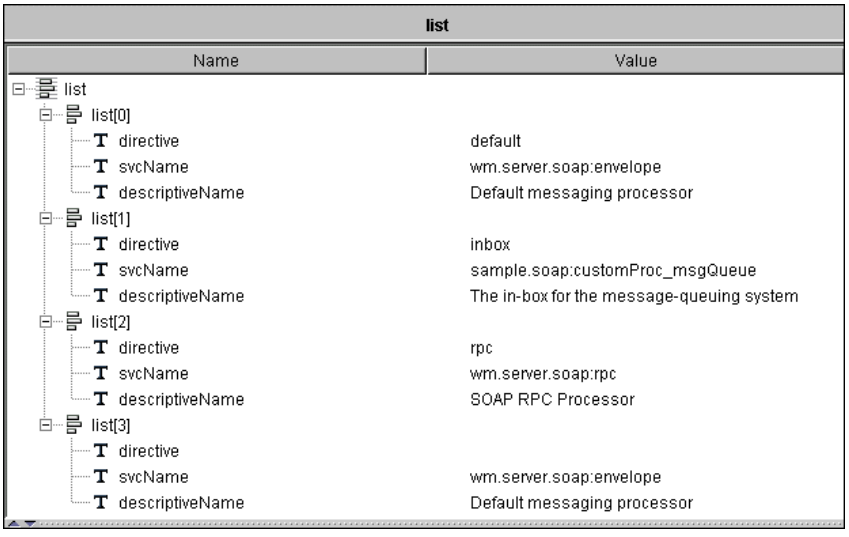
2 Run the flow service that you just created and verify that it executed successfully. If a processor is already registered under the name you specified in *directive*, the service will throw an exception. In this case, you must either register the new processor under a different name, or, if you want the new processor to replace the current one, you must unregister the current processor (with pub.soap.processor:unregisterProcessor) and then register the new one.

3 Add the flow service to the start-up list. To ensure that your processor is automatically registered with the SOAP message handler whenever the server is started, we suggest that you make it a start up service.

Viewing the List of Registered SOAP Processors

If you want to view the list of currently registered SOAP processors, execute pub.soap.processor:list from Developer. This service returns a Record List called *list* that contains a Record for each SOAP processor registered on the server.

List of registered SOAP processors



Each Record in *list* will contain the following keys:

Key	Value
directive	A String containing the process directive that is assigned to the SOAP processor.
svcName	A String containing the fully qualified name of the service that functions as the SOAP processor.
descriptiveName	A String containing the descriptive comment that was given to the SOAP processor when it was registered. This element will be empty if the processor was not registered with a descriptive comment.

Deactivating a Registered SOAP Processor

To deactivate a registered SOAP processor, execute pub.soap.processor:unregisterProcessor and specify the process directive of the processor that you want to deactivate. This action will remove the processor from the list of registered SOAP processors and make it inaccessible to the message handler.

You can also use this service when you want to change the registration information for a SOAP processor. For example, if you wanted to change a processor's directive, service, or descriptive name, you would unregister the processor and then re-register it with the new information.

Chapter 8: Composing and Sending SOAP Messages

Overview

The SAP BC server provides a set of services that allow you to generate SOAP messages and send them across the network via HTTP.

Important! If you want to use SOAP to submit remote procedure calls, you compose those messages with SOAP RPC client. For information about sending RPC messages, see "Using the SOAP RPC Client" on page 77.

Composing a SOAP Message

To compose a SOAP message, you first create an "empty" SOAP object with the createSoapData service. Then you use the message-composition services (e.g., addHeaderEntry and addBodyEntry) to add content to it.

How to Compose a SOAP Message

The following describes the general steps you take to code a service that composes a SOAP message.

1 Create an empty SOAP object using pub.soap.utils:createSoapData. You do not need to pass any input to this service.

This service returns an empty SOAP object named *soapData*.

2 Add content to *soapData* using any of the following message-composition services. You can execute these services in any order.

Use this service...	To...
pub.soap.utils:addHeaderEntry	Add a single header entry to the message. If your message includes multiple header entries, execute addHeaderEntry once for each entry that you want to add.
pub.soap.utils:addBodyEntry	Add a single body entry to the message. If your message includes multiple body entries, execute addBodyEntry once for each entry that you want to add.
pub.soap.utils:addTrailer	Add a trailer to the message. If your message includes multiple trailers, execute addTrailer once for each trailer that you want to add.

The message-composition services require two inputs: the *soapData* object to which you are adding content and a *node* representation of the element that you want to add. A *node* is an object that contains a parsed representation of an XML element. You can generate a *node*, using the pub.web:recordToDocument and pub.web:stringToDocument services. For an example of how to create a node, see Steps 2 and 3 in the sample shown on page 70.

Note: The SOAP specification states that header entries and trailers must be namespace qualified, so remember to use namespace-qualified elements in the nodes that you pass to the addHeaderEntry and addTrailer services.

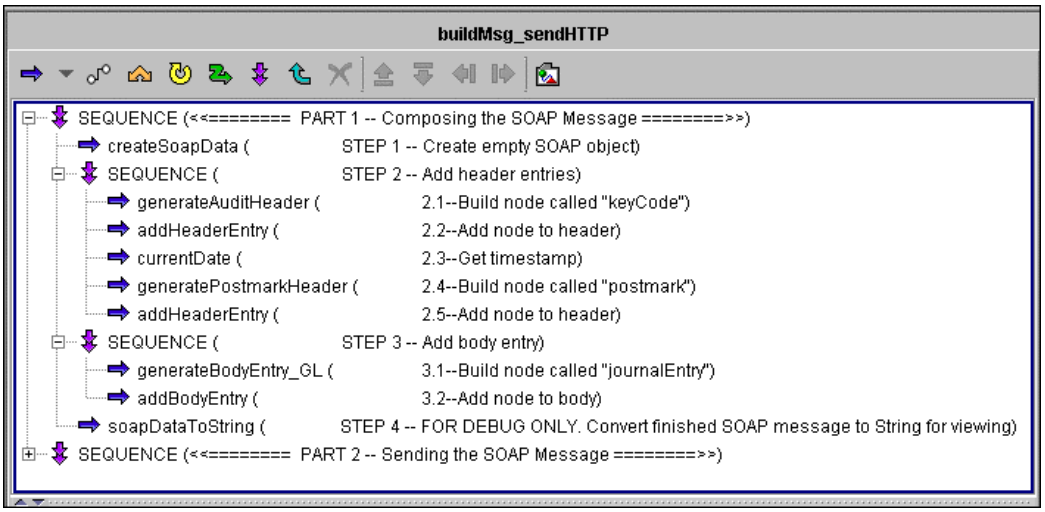
When you finish populating the *soapData* object, you use pub.client:soapHTTP to send the SOAP message to a server via HTTP. For a detailed procedure, see “Sending a SOAP Message” on page 72.

Example—Composing a SOAP Message

The following flow service generates a SOAP message that contains two header entries and one body entry. This example is located in sample.soap:buildMsg_sendHTTP in the WmSample package. You may want to open this example with Developer to see how the pipeline is mapped between steps.

Note: The following is a two-part example. The first part illustrates how to compose a SOAP message. This part is explained below. The second part illustrates how to send the SOAP message. It is explained in “Sending a SOAP Message” on page 72.

Composing a SOAP Message



#	Description
STEP 1	**Create empty SOAP object.** This step creates an empty SOAP object. It does not take any inputs. It puts an empty SOAP object named *soapData* in the pipeline.
STEP 2	**Add header entries.** This sequence builds the following header entries and adds them to *soapData*:

```
<AUDIT:keyCode xmlns:AUDIT="http://www.accumon.com/tracker">3</AUDIT:keyCode>

<EXP:postmark xmlns:EXP="http://www.exprint.com/inbox">
 <docType>GL</docType>
 <priority>09</priority>
 <sender>http://www.exprint.com/acct/AP</sender>
 <tStamp>20010716.0307340941</tStamp>
</EXP:postmark>
```

#	Description

Note that both entries are namespace qualified, as required by the SOAP specification.

To accomplish this, the service executes the following steps:

Step	Description
2.1	generateAuditHeader—This helper service converts a Record representation of the first header entry to a node. (Recall that to add a header entry to a *soapData* object, you must place the entry in the pipeline as a node.) You can examine the helper service to understand how this is accomplished.
2..2	addHeaderEntry—This step adds the first header entry to *soapData*.
2..3	currentDate—This service generates a timestamp, which will be inserted into the second header entry in the next step.
2..4	generatePostmarkHeader—This helper service converts a Record representation of the second header entry to a node. You can examine the helper service to understand how this is accomplished.)
2..5	addHeaderEntry—This step adds the second header entry to *soapData*.

STEP 3 Add body entry. This step generates the following body entry and adds it to *soapData*:

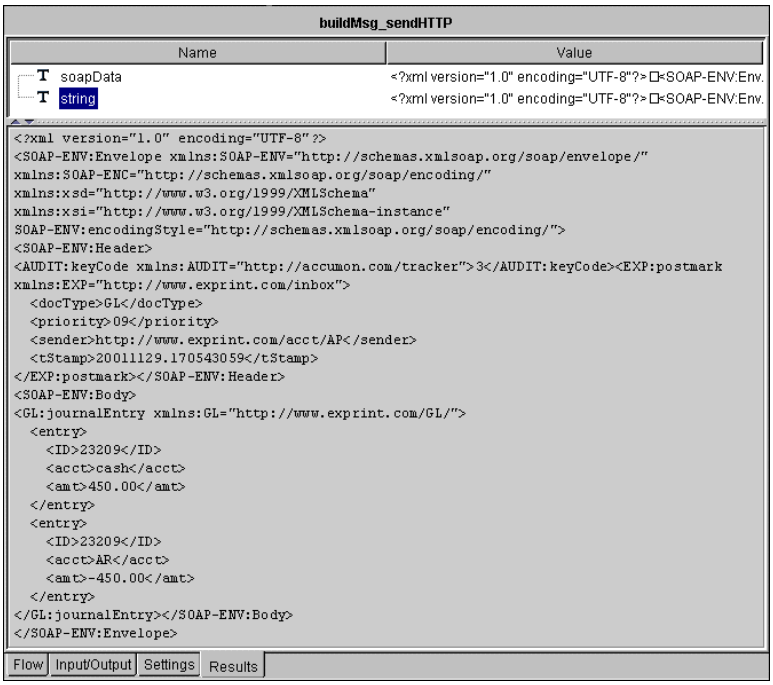
```
<GL:journalEntry xmlns:GL="http://www.exprint.com/GL/">
    <entry>
       <ID>23209</ID>
       <acct>cash</acct>
       <amt>450.00</amt>
    </entry>
    <entry>
      <ID>23209</ID>
      <acct>AR</acct>
      <amt>-450.00</amt>
    </entry>
</GL:journalEntry>
```

To accomplish this, the service executes the following steps:

Step	Description
3.1	generateBodyEntry_GL—This helper service converts a Record representation of the body entry to a node. (Recall that to add a body entry to a *soapData* object, you must place the entry in the pipeline as a node.) You can examine the helper service to understand how this is accomplished.
3.2	addBodyEntry—This step adds the body entry to *soapData*.

STEP 4 FOR DEBUG ONLY. This step converts the contents of *soapData* to a String using the soapDataToString service. This allows you to examine the finished SOAP message with Developer, which is useful during testing and debugging. You would not include this step in a production service.

If you examine the contents of *string* on the Results tab, you will see a SOAP message similar to the following:



Sending a SOAP Message

To use the SAP BC server to send a SOAP message to a remote server, you compose the SOAP message in a *soapData* object as described in the previous section, and then you pass that object to pub.client:soapHTTP, which submits it to a server that you specify.

SOAP messages that you send via soapHTTP elicit a response, which, if the target server is SOAP compliant, will be a SOAP response document. Therefore, your service must also include logic to process the response that the target server returns.

How to Send a SOAP Message via HTTP

The following describes the general steps you take to code a service that sends a SOAP message to a remote server via HTTP.

1 Compose a SOAP message in a *soapData* object. For information about composing SOAP messages, see “Composing a SOAP Message” on page 69.

2 Send the SOAP message to the server using pub.client:soapHTTP. This service takes the SOAP message from the *soapData* object produced by Step 1 and posts it to the URL that you specify.

> Note that the soapHTTP service expects to find the SOAP message in an object named *soapRequestData*. If you are building a flow service, you will need to use the Pipeline Editor to map the *soapData* produced in Step 1 to *soapRequestData*.)
>
> Besides the *soapData* object from Step 1, you must provide the following parameters to soapHTTP:

Name	Description
address	A String specifying the HTTP address to which you want to message posted. Example `http://rubicon:5555/soap/default`
auth	A Record (an IData object) specifying the user name and password that the service will submit to the target server:

Name	Description
type	A String specifying the type of authentication that the service will perform. Set *type* to `basic`.
user	A String specifying the user name that this service will use to access a protected resource.
password	A String specifying the password that this service will use to access a protected resource.

You may also provide the following optional parameters:

Name	Description
validateSoap	A String indicating whether or not you want the request and response envelope to be validated against the SOAP schema.

Set to...	To...
`true`	Validate the SOAP envelope generated by `soapHTTP` and the one received by `soapHTTP`. When *validateSoap* is true, the service will throw an exception if the request or response does not conform to the SOAP schema.
`false`	Bypass the validation process.

SOAPAction	A String specifying the value of the `SOAPAction` HTTP header. Note: The `SOAPAction` header was required by the initial SOAP specification, but has since been deprecated. The SAP BC server does not use the `SOAPAction` header and accepts SOAP messages that omit it. If you are designing a completely new solution, we recommend that you avoid using the `SOAPAction` header. However, if you exchange SOAP messages with systems that require a `SOAPAction` header, this parameter allows you to set it.

3 Use the data-retrieval services to fetch information from the response message. If the server returns a SOAP message, `soapHTTP` returns the message in a SOAP object called *soapResponseData*. To retrieve the data in *soapResponseData*, you use any of the following data-retrieval services.

Use this service...	To...
pub.soap.utils:getBody	Retrieve the body as a single node.
pub.soap.utils:getBodyEntries	Retrieve the contents of the body as an array of nodes, where each element in the array represents a single body entry.
pub.soap.utils:getDocument	Retrieve the SOAP envelope as a node.
pub.soap.utils:getHeader	Retrieve the header as a single node.

Use this service...	To...
pub.soap.utils:getHeaderEntries	Retrieve the contents of the header as an array of nodes, where each element in the array represents a single header entry.
pub.soap.utils:getTrailers	Retrieve the trailers as an array of nodes, where each element in the array represents a single trailer.

The data-retrieval services return a *node* object (or an array of *node* objects) as output. To extract information from a *node*, you can query it with the pub.web:queryDocument service. See Step 2 in the sample flow shown on page 75 for an example of how to do this.

The soapHTTP service also returns a status parameter, which you can use to test the results before processing them.

Output Parameter	Description
soapStatus	A String indicating whether the SOAP request message was processed successfully, where:

A value of...	Indicates that...
`0`	The remote server successfully processed the SOAP request message and returned a SOAP response message.
`1`	The remote server returned a SOAP fault, indicating that the SOAP request message was received, but was not processed successfully.
`2`	The server returned an error that was not a SOAP fault. This indicates that some type of HTTP error occurred (often, an HTTP 404). You can check the *status* element in *header* to determine what type of HTTP error occurred.

For an example of how to test these values, see Step 2 in the following example.

Example—Sending a SOAP Message

The following flow service composes a SOAP message and then sends it to a SOAP processor at http://localhost:5555/soap/inbox.

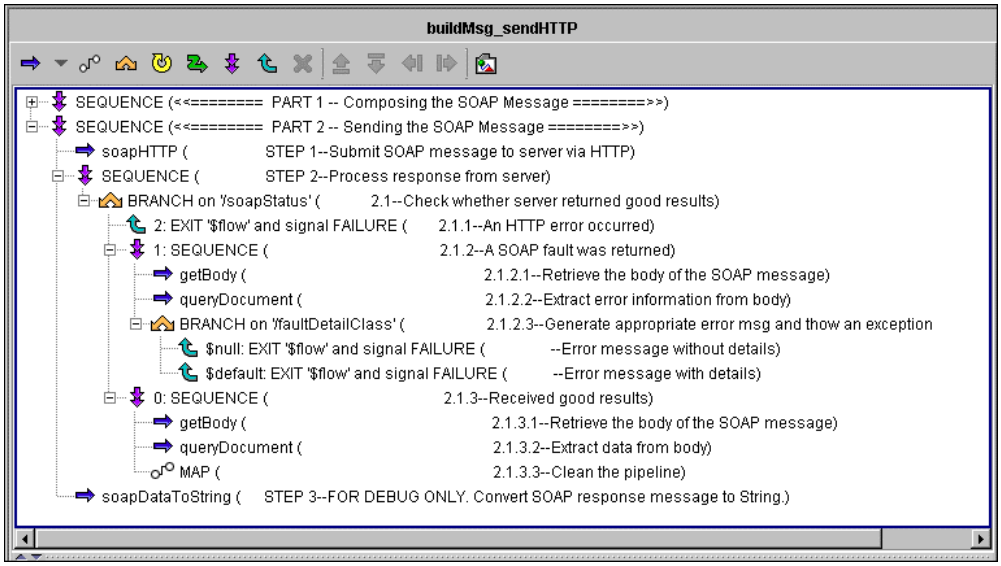
If you want to execute this example from Developer, you must first execute sample.soap:registerProcessor to register the customProc_MsgQueue service with the SOAP message handler (it will be registered under the "inbox" process directive). If you are not running the SAP BC server on your local machine, modify the URL in the *address* parameter in Step 1 to point to the machine where your server is running.

This example will prompt you for a user name and password when you execute it. To run this sample successfully, you must provide a user name and password that belongs to the Developers ACL on your SAP BC server.

Note: The following example is broken into two parts. The first part illustrates how to compose a SOAP message. This part is explained in "Composing a SOAP Message" on page 69. The second part illustrates how to send the

SOAP message. This part is explained below.

Sending a SOAP message and processing the response



STEP 1 **Submit SOAP message to server via HTTP.** This step sends the SOAP message to the SOAP processor registered under the name "inbox." Note that the *soapData* object created in Part 1 is mapped to the *soapRequestData* that this step takes as input. This step also sets the following input values:

Name	Description
address	This parameter is set to: `http://localhost:5555/soap/inbox` This URL assumes that the SAP BC server is running on your local machine and is listening for HTTP requests on port 5555. If your server is running on a different machine or port, modify the host name and port portions of this URL.
auth	This Record specifies the user name and password that this service will use to connect to the SAP BC server. These values are mapped from the *userName* and *password* parameters that this service takes as input. (When you execute this service from Developer, you are prompted for these values.) This example submits its message to the customProc_MsgQueue processor, which is controlled by the Developers ACL. To submit a message to this processor, you must provide a user name and password that belongs to the Developers ACL.

STEP 2 **Process response from server.** This sequence processes the results from Step 1. To determine whether the request was processed successfully, it checks the state of the *soapStatus* variable:

If *soapStatus* is...	The service...
2	Composes an error message and throws an exception. A value of 2 indicates that an HTTP failure occurred.

`1` Extracts error information from the returned message and throws an exception. A value of 1 indicates that the remote server returned a SOAP fault.

`0` Processes the contents of *soapResponseData*. A value of 0 indicates that the remote server received and successfully processed the SOAP message.

STEP 3 FOR DEBUG ONLY. This step converts the contents of *soapResponseData* to a String using soapDataToString. This allows you to examine the SOAP response message with Developer, which is useful during testing and debugging. You would not include it in a production service.

If you examine the contents of *string* on the Results tab, you will see a SOAP message similar to the following:

Chapter 9: Using the SOAP RPC Client

Overview

The SOAP RPC client—pub.client:soapRPC—allows you to call procedures on remote, SOAP-enabled servers.

To use the SOAP RPC client successfully, you need to have the following information about the procedure that you are calling:

- The HTTP address of the server on which it resides.
- The credentials (user name and password), if any, that are required to access it.
- The qualified name (QName) associated with the procedure.
- The names and data types of the parameters that the procedure expects as input and returns as output.

Using pub.client:soapRPC

The following describes the general steps you take to submit a SOAP remote procedure call from the SAP BC Server.

1 Invoke pub.client:soapRPC with the following input parameters:

Input Parameter	Description
address	A String specifying the HTTP address of the server on which the remote procedure resides. (If you are submitting the request to a SAP BC Server, remember to direct it to the RPC processor as shown in the following example.) Example: `http://rubicon:5555/soap/rpc`
auth	An IData object (a Record) specifying the user name and password that are to be submitted to the server specified in *address*, where:

Key	Description
type	A String specifying the type of authentication that the server uses. Set *type* to `basic`.
user	A String specifying the user name that is to be presented to the server.
pass	A String specifying the password for the user name specified in *user*.

Input Parameter	Description
method	An IData object (a Record) specifying the QName of the requested procedure, where:

Key	Description
namespaceName	A String specifying the namespace portion of the procedure's QName.

Input Parameter	Description
	localName A String specifying the local portion of the procedure's QName.
reqParms	An IData object (a Record) whose elements represent the input parameters that are to be passed to the remote procedure. For example, if you were to pass three String parameters, *acct*, *amt*, and *org*, containing the values, `Cash`, `150.00` and `Sales`, *reqParms* would contain the following: Key: *acct* Value: `Cash` Key: *amt* Value: `150.00` Key: *org* Value: `Sales` At run time, the values in *reqParms* are XML-encoded by the message coder. For a description of this process, see "Encoding the Input Parameters for the Remote Procedure Call" on page 83.
targetInputSignature	Optional. A String specifying the fully qualified name of the stored Record that is to be used to validate and encode the contents of *reqParms*. For a description of how the message coder uses *targetInputSignature*, see "Encoding the Input Parameters for the Remote Procedure Call" on page 83.
targetOutputSignature	Optional. A String specifying the fully qualified name of the stored Record that is to be used to validate and decode the output value returned by the remote procedure. For a description of how the message coder uses *targetInputSignature*, see "Decoding the Output Parameters from a Remote Procedure Call" on page 85.
SOAPAction	Optional. A String specifying the value to which you want the `SOAPAction` HTTP header set. Note: The `SOAPAction` header was required by the initial SOAP specification, but has since been deprecated. The SAP BC server does not use the `SOAPAction` header and accepts SOAP messages that omit it. If you are designing a completely new solution, we recommend that you avoid using the `SOAPAction` header. However, if you exchange SOAP messages with systems that require a `SOAPAction` header, this parameter allows you to set it.

2 Process the results returned by pub.client:soapRPC. If the remote procedure returns output values, those values are decoded by the message coder and placed in an IData object called *respParms*. For example, if the procedure were to return two string parameters, *status* and *balance*, whose values were `closed` and `-4.95`, *respParms* would contain the following:

Key	Value
status	`closed`
balance	`-4.95`

When decoding the results, the message coder uses the Record or Specification named in *targetOutputSignature* (if one was specified) to validate the values in *respParms*. For a description of the decoding process, see “Decoding the Output Parameters from a Remote Procedure Call” on page 85.

Besides *respParms*, pub.client:soapRPC returns the following values, which you can use to determine whether the request was processed successfully:

<table>
<tr><th>Output Parameter</th><th>Description</th></tr>
<tr><td>soapStatus</td><td>A String indicating whether the remote procedure call was processed successfully, where:
<table>
<tr><th>A value of...</th><th>Indicates that...</th></tr>
<tr><td><code>0</code></td><td>The remote server successfully returned the results of the remote procedure call.</td></tr>
<tr><td><code>1</code></td><td>The remote server returned a SOAP fault, indicating that the remote procedure call was received, but was not processed successfully.</td></tr>
<tr><td><code>2</code></td><td>The server returned an error that was not a SOAP fault. This indicates that some type of HTTP error occurred (often, an HTTP 404). You can check the status element in header (below) to determine what type of HTTP error occurred.</td></tr>
</table></td></tr>
<tr><td>header</td><td>An IData object (Record) containing information from the HTTP header returned by the remote server. header contains the following elements:
<table>
<tr><th>Key</th><th>Value</th></tr>
<tr><td>lines</td><td>An IData object in which each entry represents a field (line) of the response header. Key names represent the names of the header fields and key values are Strings containing the values of the header fields.</td></tr>
<tr><td>status</td><td>A String containing the status code from the HTTP response. For example, if the server returns a HTTP 404 error, status will contain <code>404</code>.</td></tr>
<tr><td>statusMessage</td><td>A String containing the status message from the HTTP response.</td></tr>
</table></td></tr>
<tr><td>soapResponseData</td><td>A SOAP object containing the entire SOAP response message. You can extract data from this object using the data-retrieval services such as getBody and getHeaderEntries. (For a complete list of data-retrieval services, see page 73).</td></tr>
</table>

Example—Submitting a Remote Procedure Call

The following flow service calls a remote procedure that takes three String parameters (*acct*, *amt*, and *loc*) and returns one String parameter (*authCode*).

This example is located in sample.soap:buildRPC_SendHTTPSimple. You may want to examine it with SAP BCDeveloper to understand how its parameters are set.

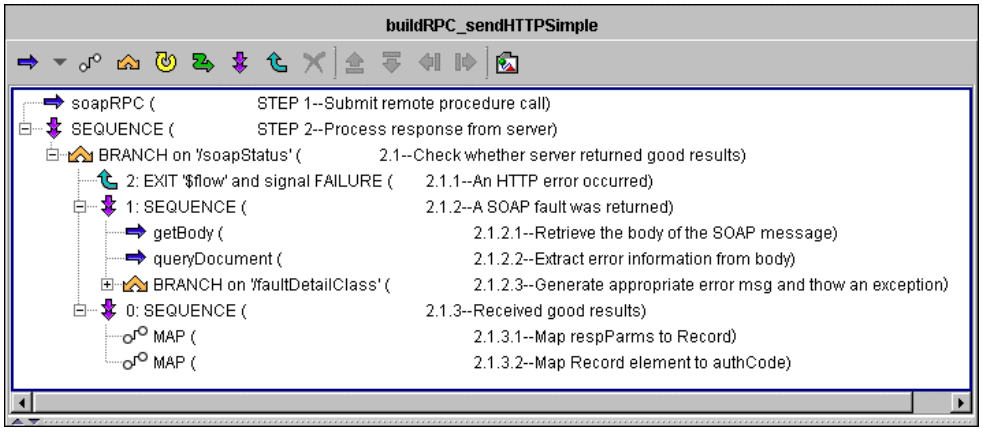
The service calls a remote procedure whose QName is made up of the namespace name, http://www.expt.com/AUTH/, and local name, getAuthCode. On a SAP BC Server (version 4.6 or later) this name is registered to sample.soap:targetSvc_rpcProcSimple. (This service resides in the WmSamples package and is described in the “Example—Target Service for the RPC Processor” on page 47.)

To execute this example from Developer, modify the URL in the *address* parameter in Step 1 to point to the machine where your SAP BC server is running.

When you execute this example, you will be prompted for the following values:

For this input parameter...	Enter...
acct	Any string of characters.
amt	A decimal value, such as 150.75 or 15.00. (Omit commas from large values; otherwise, the value will fail validation.)
loc	Any string of characters.
userName	A user name that belongs to the Developers ACL.
password	The password for the user name that you entered in *userName*.

Example of a service that submits a remote procedure call and processes the response



STEP 1 **Submit remote procedure call.** This step composes the remote procedure call and submits it to the server specified in *address*. If you examine the pipeline, you will see that its input parameters are set as follows:

Name	Description
address	This parameter is set to: `http://localhost:5555/soap/rpc` This directs the request to the RPC processor on a SAP BC Server. This URL assumes that the SAP BC server is running on your local machine and is listening for HTTP requests on port 5555. If your server is running on a different machine or port, modify the host name and port portions of this URL.
auth	This Record specifies the user name and password that this service will use to connect to the SAP BC Server. These values are mapped from the *userName* and *password* parameters that this service takes as input. (When you execute this service from SAP BCDeveloper, you are prompted for these values.) Note: When you submit a SOAP remote procedure call to an SAP BC server, your credentials are verified against the ACL for the target service.
method	This Record specifies the QName of the remote procedure. In this example, *method* is set as follows: Key: *namespaceName* — Value: `http://www.expt.com/AUTH/` Key: *localName* — Value: `getAuthCode` On the SAP BC Server, this QName represents the universal name assigned to the service sample.soap:targetSvc_rpcProcSimple.
reqParms	This Record contains the input parameters that are to be passed to the remote procedure. Note that *authCodeReq*, a Record that is part of the input signature for this example, is mapped to *reqParms*.
targetInputSignature	This parameter is set to: `sample.soap.helpers.records:rec_authCodeReq` which is the name of an SAP BC Record definition. This parameter tells the message coder to validate the contents of *reqParms* (the input parameters for the remote procedure) against this Record definition at run time. It also instructs the message coder to encode the contents of *reqParms* according to the data types specified in the Record definition.
targetOutputSignature	This parameter is set to: `sample.soap.helpers.records:rec_authCodeResp` which is the name of an SAP BC Record definition. This parameter tells the message coder to validate the contents of *respParms* (the results from the remote procedure) against this Record definition at run time. It also instructs the message coder to decode the contents of *respParms* according to the data types specified in the Record definition.

STEP 2 **Process response from server.** This sequence processes the results from Step 1.

To determine whether the request was processed successfully, this step first checks the state of the *soapStatus* parameter:

If *soapStatus* is...	The service...
2	Composes an error message and throws an exception. A value of 2 indicates that an HTTP failure occurred.
1	Extracts error information from the returned message and throws an exception. A value of 1 indicates that the remote server returned a SOAP fault.
0	Processes the contents of *respParms*. A value of 0 indicates that the remote server returned the results of the requested procedure.

The Message Coder and the RPC Client

The SOAP RPC client engages the message coder to convert the input and output parameters to and from XML.

Encoding the Input Parameters for the Remote Procedure Call

At run time, the parameters in *reqParms* are converted to XML by the message coder. This process is known as *encoding*. The way in which the message coder encodes the parameters in *reqParms* depends on whether *targetInputSignature* is set.

- If *targetInputSignature* is not set, the message coder encodes the elements in *reqParms* according to their underlying Java classes. For example, if a parameter were of class java.lang.Boolean, the message coder would encode it as `xsi:type="xsd:boolean"` in the resulting XML. If the message coder does not recognize the underlying class, it encodes the parameter as a string (it uses the object's toString( ) method to produce the parameter's value).

  The following table shows how the message coder would encode a parameter *amt* (with a value of "500.00") depending on its Java class.

If the Java Class were...	The Message Coder would produce...
java.math.BigDecimal	`<amt xsi:type="xsd:decimal">500.00</amt>`
java.lang.Float	`<amt xsi:type="xsd:float">500.00</amt>`
myJavaClass	`<amt xsi:type="xsd:string">`*results of toStringMethod*`</amt>`

  For a list of recognized Java classes and the XML data types to which they are converted, see Appendix B *Encoding/Decoding Data-Type Mapping*.

- If *targetInputSignature* is set, the message coder first validates the contents of *reqParms* against the Record specified in *targetInputSignature*. If the parameters in *reqParms* violate the Record specified in *targetInputSignature*, (for example, if a required parameter is missing or a value is not of the correct type), the message coder throws an exception.

  If the parameters are valid, the message coder encodes them as follows:

  – If the parameter is a String, the value of the parameter is encoded according to its Content Type property. For example, if its Content Type is `nonNegativeInteger {http://www.w3.org/2001/XMLSchema}`, the

value is encoded as a `xsi:type="xsd:nonNegativeInteger"`. If the Content Type property is not specified, the message coder encodes the value as a string (as though its Content Type were `string {http://www.w3.org/2001/XMLSchema}`).

The following table shows how the message coder would encode a String parameter *amt* (with a value of "500.00") for different Content Type values.

If Content Type were…	The Message Coder would produce…
`decimal {http://www.w3.org/2001/XMLSchema}`	`<amt xsi:type="xsd:decimal">500.00</amt>`
`float{http://www.w3.org/2001/XMLSchema}`	`<amt xsi:type="xsd:float">500.00</amt>`
`not specified`	`<amt xsi:type="xsd:string">500.00</amt>`

– If the parameter is an Object, the value of the parameter is encoded according to its underlying Java class.

Encoding Complex Structures and Arrays

The message coder encodes Records (IData objects) as complex data types.

The message coder encodes Record Lists, String Lists, and Object Lists as SOAP arrays. For more information about the how arrays are encoded in SOAP messages, see the "Arrays" section in the *Simple Object Access Protocol (SOAP) 1.1 W3C Note* at http://www.w3.org/TR/SOAP/#_Toc478383522.

Encoding Multi-Referenced Elements

By default, the message coder encodes parameters as independent elements, regardless of whether they reference the same underlying objects in the pipeline. However, this behavior can be modified with the `watt.server.SOAP.useMultiReference` parameter.

When `watt.server.SOAP.useMultiReference` is true, the message coder will generate the appropriate `id` and href attributes for parameters that reference the same underlying data. For example, if the parameters *cDate* and *oDate* refer to the same object, the message coder encodes them like this if `watt.server.SOAP.useMultiReference=true`:

```
.
.
.
<cDate id="id1">03/15/2000</cDate>
<amt id="id2">300</amt>
<oDate href="#id1"></oDate>
<acct id=id4>cash</acct>
.
.
.
```

The *oDate* parameter is encoded as a reference

And encodes them like this if `watt.server.SOAP.useMultiReference=false` or is not set at all (which is the server's installed behavior):

```
.
.
.
<cDate>03/15/2000</cDate>
<amt>300</amt>
<oDate>03/15/2000</oDate>
<acct>cash</acct>
.
```

The *oDate* parameter is encoded as an independent element

:

For more information about `watt.server.SOAP.useMultiReference` and other SOAP-related server parameters, see Appendix C *SOAP-Related Server Parameters*. For more information about setting server parameters, see the *SAP BC Server Administrator's Guide*.

Decoding the Output Parameters from a Remote Procedure Call

When the results of a remote procedure call are returned to the SOAP RPC client, the message coder converts them from XML and places them, as Java Objects, in *respParms*. This process is known as *decoding*.

The way in which the message coder decodes the results depends on whether *targetOutputSignature* is set.

- If *targetOutputSignature* is not set, the output parameters are rendered according to the data types declared in the XML. For example, it would convert a parameter whose data type is `xsi:type=xsd:decimal` to a Java object of class java.math.BigDecimal. (For a list of XML data types and the Java classes to which they are converted, see Appendix B *Encoding/Decoding Data-Type Mapping*). If an XML element does not declare its type, the parameter is rendered as a String.
- If *targetOutputSignature* is set, the message coder converts the output parameters according to the Record specified in *targetOutputSignature*.
  - If targetOutputSignature declares a parameter as a String, the value is rendered as a String object, regardless of the data type declared in the XML.
  - If targetOutputSignature declares a parameter as an Object, the value is rendered according to the data type declared in the XML.

When *targetOutputSignature* is specified, the message coder also validates the contents of *respParms*. If the parameters returned in *respParms* violate the Record defined in *targetOutputSignature*, (for example, if a required parameter is missing or a value is not of the correct type), the message coder throws an exception.

Decoding Complex Structures and Arrays

For complex data types (XML elements that contain child elements), the message coder produces Records in the pipeline.

The message coder will create arrays of elements (i.e., String Lists, Record Lists, Object lists) for elements that are properly encoded as SOAP arrays. For more information about the how arrays are encoded in a SOAP message, see the "Arrays" section in the *Simple Object Access Protocol (SOAP) 1.1 W3C Note* at http://www.w3.org/TR/2000/NOTE-SOAP-20000508/#_Toc478383522.

Decoding Multi-Referenced Parameters

The message coder can decode parameters that are referenced via the href attribute, providing the parameter containing the href attribute appears *after* the value that it references.

For example, the message coder can successfully decode the following reference:

This reference can be decoded, because the value of *id1* is assigned before it is referenced.

```
.
.
.
<cDate id="id1">03/15/2000</cDate>
<amt id="id2">300</amt>
<oDate href="#id1"></oDate>
.
.
.
```

But cannot decode the following:

This reference cannot be decoded, because the value of *id1* is assigned after it is referenced.

```
.
.
.
<oDate href="#id1"></oDate>
<amt id="id2">300</amt?
<cDate id="id1">03/15/2000</cDate>
.
.
.
```

Note: If the message coder cannot resolve a reference, it generates a null object for that parameter. It also reports the problem in server.log if the server is running at debug level 5 or higher.

For referenced elements, the message coder produces a single copy of the source data and generates references to it.

Appendix A: SOAP Faults Returned by the SAP BC server

Basic Structure of a SOAP Fault

If the SAP BC server is not able to process a SOAP message, it returns a SOAP fault. A SOAP fault is a SOAP message with the following structure.

The body contains the Fault element, which contains information about the failure

```
<SOAP-ENV:Envelope
     xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
     xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
     xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
  <SOAP-ENV:Fault>
    <faultcode>…</faultcode>
    <faultstring>…</faultstring>
    <faultactor>…</faultactor>
    <detail>…</detail>
  </SOAP-ENV:Fault>
</SOAP-ENV:Body>
   </SOAP-ENV:Envelope>
```

Elements of a SOAP Fault

The body of a SOAP fault contains the Fault element. When the Integration Serve returns a SOAP fault, it contains the following children:

Element	Value
`<faultcode>`	A QName indicating the class of error that occurred. The namespace for the QName will be the same as that of the SOAP envelope and the local name will be one of the following:

Local Name	Indicates that...
`Server`	The SAP BC server was not able to process the message because of a problem on the server.
`Client`	The SAP BC server was not able to process the message because it was formatted improperly. —OR— SAP BC server was not able to process the message because the SOAP processor (or one of the services it called) threw an exception.
`MustUnderstand`	The SOAP processor could not obey a header element whose mustUnderstand attribute was set to 1.
`VersionMismatch`	The SAP BC server does not support the SOAP namespace specified in the SOAP envelope.

`<faultstring>`	A message describing the error. For a list of messages that might appear

in faultstring, see “SAP BC SOAP Faults” on page 93.

`<faultactor>` A URI indicating which SOAP processor returned the error.

Example: `http://rubicon:5555/soap/rpc`

`<detail>` An element containing detailed information about the error. The SAP BC server returns this element when an exception occurs while it is processing the message.

The `<detail>` element returned by the SAP BC server is from the following namespace:

```
http://www.webmethods.com/2001/10/soap
```

It contains an element called `<exception>`, which has the following children and provides specific information about the exception:

Element	Description
`<classname>`	The name of the Java class that was thrown.
`<message>`	The detailed error message from the exception.
`<stackTrace>`	A Java stack trace. Important! This element is included only if the client is a member of the Developers or Administrators group on the SAP BC Server.
`<serviceStackTrace>`	A service stack trace. This element is returned if a service stack is available to the SOAP message handler. It generally appears only when an exception is thrown by one of the SAP BCSOAP processors or utilities. Important! This element is included only if the client is a member of the Developers or Administrators group on the SAP BC Server.

Example—Unknown SOAP Processor

The following is an example of the SOAP fault that the SAP BC server returns when a client directs a message to processor that does not exist on the server.

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSchema">

<SOAP-ENV:Body>
```

```
 <SOAP-ENV:Fault>
  <faultcode>SOAP-ENV:Server</faultcode>
  <faultstring>
    [B2BPCKG.0088.9123] Requested SOAP processor inbox is not registered
    on this server
  </faultstring>
  <faultactor>http://localhost:5555/soap/inbox</faultactor>
 </SOAP-ENV:Fault>
</SOAP-ENV:Body>
```

Note that this fault does not include the `<detail>` element.

Example—Exception While Processing Message

The following is an example of the SOAP fault that the SAP BC server returns if an exception is thrown while it is processing a SOAP message. Note that this fault includes the detail element, which provides information about the exception.

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSchema">

 <SOAP-ENV:Body>

   <SOAP-ENV:Fault>

    <faultcode>SOAP-ENV:Server</faultcode>

    <faultstring>

        [B2BPCKG.0088.9112]  An Exception was thrown in the server

    </faultstring>

    <faultactor>http://localhost:5555/soap/inbox</faultactor>

    <detail xmlns="http://www.webmethods.com/2001/10/soap">

        <exception>

            <className>com.wm.lang.flow.FlowException</className>

            <message xml:lang="en_US">

             [B2BCORE.0049.9010] Failure invoking unknown service at

             'unlabeledINVOKE'. The service may have been renamed, moved

             or disabled.

            </message>

            <stackTrace xml:lang="en_US">

com.wm.lang.flow.FlowException: [B2BCORE.0049.9010] Failure invoking unknown
service at 'unlabeledINVOKE'. The service may have been renamed, moved or disabled.
   at java.lang.Throwable.&lt;init&gt;(Throwable.java:84)
   at java.lang.Exception.&lt;init&gt;(Exception.java:35)
   at com.wm.util.LocalizedException.&lt;init&gt;(LocalizedException.java:83)
   at com.wm.lang.flow.FlowException.&lt;init&gt;(FlowException.java:42)
   at com.wm.lang.flow.FlowExit.getFailure(FlowExit.java:199)
   at com.wm.lang.flow.FlowState.willExit(FlowState.java:195)
   at com.wm.lang.flow.FlowSequence.invoke(FlowSequence.java:150)
   at com.wm.lang.flow.FlowRoot.invoke(FlowRoot.java:199)
   at com.wm.lang.flow.FlowState.invokeNode(FlowState.java:459)
   at com.wm.lang.flow.FlowState.step(FlowState.java:341)
   at com.wm.lang.flow.FlowState.invoke(FlowState.java:309)
   at com.wm.app.b2b.server.FlowSvcImpl.baseInvoke(FlowSvcImpl.java:1334)
   at com.wm.app.b2b.server.ServiceManager.invoke(ServiceManager.java:692)
   at com.wm.app.b2b.server.ServiceManager.invoke(ServiceManager.java:478)
   at com.wm.app.b2b.server.HTTPSOAPHandler.process(HTTPSOAPHandler.java:234)
```

```
   at com.wm.app.b2b.server.HTTPDispatch.run(HTTPDispatch.java:255)
   at com.wm.util.pool.PooledThread.run(PooledThread.java:103)
   at java.lang.Thread.run(Thread.java:498)
            </stackTrace>

        </exception>

     </detail>

    </SOAP-ENV:Fault>

  </SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

SAP BC SOAP Faults

The following are SOAP-related errors that are issued by the SAP BC Server.

[B2BCORE.0076.0013] SOAP Encoding Registrar cannot register Schema type ***typeName***; Schema type already exists

The message coder encountered a serious internal error.

Contact SAP BCTechnical Support

[B2BCORE.0076.0014] SOAP Message Coder failure; a Runtime Exception was thrown in the SOAP Coder

The message coder encountered a serious internal error.

Contact SAP BCTechnical Support

[B2BCORE.0076.0015] SOAP Message Coder cannot encode variable ***varName***; variable does not have a valid XML name

The message coder could not XML-encode the SOAP RPC request or response because the pipeline contained the variable *varName*, whose name is not a valid name for an XML element.

This error often occurs because the variable name contains a colon. Change the name of *varName* to one that is also valid as an XML element name (or map *varName* to a new variable, and use that variable as your SOAP RPC input or output parameter).

[B2BCORE.0076.0016] Soap Message Coder warning; unregistered coder for variable ***varName***, using String

The message coder was not able to recognize the data type for *varName* so it treated the variable as a String.

Note: This message is a warning that is written to server.log at debug level 5 and higher.

Check the incoming XML and verify that it declares the correct data type.

[B2BCORE.0076.0017] SOAP Message Coder error; no coder is registered for encoding style ***styleName***

The message coder encountered a serious internal error.

Contact SAP BCTechnical Support

[B2BCORE.0076.0018] Soap Message Coder warning; unregistered coder for variable ***varName***, using BasicData

The message coder was not able to recognize the complex data type for *varName* so it treated the variable as a Record (an IData object).

Note: This message is a warning that is written to server.log at debug level 5 and higher.

Check the incoming XML and verify that it declares the correct data type.

[B2BCORE.0076.0019] SOAP Message Coder cannot decode message; encoding style ***styleName*** is not a registered style

The message coder could not decode the submitted XML because it does not support the specified encoding style.

Check the incoming XML and verify that the encodingStyle attribute is specified correctly and that it specifies a style that the SAP BC server supports.

[B2BCORE.0076.0020] Soap Message Coder warning; invalid HREF found ***reference***

The message coder could not resolve the href pointing to *reference*. A null value was produced instead.

Note: This message is a warning that is written to server.log at debug level 5 and higher.

Examine the incoming XML and make sure that the specified reference points to an element that exists in the body of the message and that the element to which it points appears *before* the element containing the href.

[B2BCORE.0076.0021] SOAP Message Coder cannot decode message; arrayType attribute value ***attValue*** is malformed

The message coder could not decode an XML-encoded array because it cannot understand the value of the arrayType attribute.

Check the incoming XML and make sure that the arrayType attribute is specified correctly. For example, make sure that it is not missing a [symbol.

[B2BCORE.0076.0022] SOAP Message Coder cannot decode message; cannot determine type of element ***elementName***

The message coder could not decode the XML-encoded parameters it has received, because the simple element *elementName* has an unknown data type (for example, `xsi:type="myDataType"`).

Check the incoming XML and make sure the data type has been specified correctly. Also make sure that it is using the correct namespace.

[B2BCORE.0076.0023] SOAP Message Coder cannot decode message; unable to create array for arrayType attribute value ***attValue***

The message coder could not decode an XML-encoded array because it could not allocate an array of the specified dimensions.

Check the incoming XML and make sure that the arrayType attribute is specified correctly. For example, make sure that it does not specify a negative dimension or an invalid array size (e.g., too many dimensions or dimensions that are unrealistically large). If the array is unusually large or the server is extremely very low on memory, the possibility also exists that this error was the result of an out-of-memory condition.

[B2BCORE.0076.9101] SOAP Encoding Registrar cannot register encoding style ***styleName***; encoding style already exists

The encoding registrar encountered a serious internal error.

Contact SAP BCTechnical Support

[B2BCORE.0076.9102] SOAP Encoding Registrar cannot register Java type ***typeName***; Java type already exists

The encoding registrar encountered a serious internal error.

Contact SAP BCTechnical Support

[B2BCORE.0076.9103] SOAP Encoding Registrar cannot register Schema type ***typeName***; Schema type already exists

The encoding registrar encountered a serious internal error.

Contact SAP BCTechnical Support

[B2BCORE.0076.9201] SOAP Message Coder cannot decode message; no encoding style is specified

The message coder could not decode the parameters in the SOAP message because the message does not specify an encoding style.

This error usually occurs because the SOAP encodingStyle attribute is missing, misplaced or misspelled. Examine the RPC request that was submitted to the SAP BC server and ensure that the SOAP Envelope includes the encodingStyle attribute. Also check that the encodingStyle attribute has the proper syntax and namespace.

[B2BCORE.0076.9202] SOAP Message Coder failure; a Runtime Exception was thrown in the SOAP Coder

The message coder encountered a serious internal error.

Contact SAP BCTechnical Support

[B2BCORE.0076.9203] SOAP Message Coder cannot decode message; Invalid number ***value***

The message coder could not decode an XML-encoded array because one of its dimensions is set to a nonnumeric value.

Check the incoming XML and make sure that the arrayType attribute specifies an positive integer. For example, make sure it has not been inadvertently set to a nonnumeric value such as 'A'.

[B2BCORE.0076.9204] SOAP Message Coder cannot encode output data; pipeline does not match output signature, duplicate variable ***varName*** in pipeline

The message coder could not XML-encode the results of a SOAP remote procedure call because the variable *varName* appeared in the pipeline more than once.

On the SAP BC server, examine the target service to determine why it is not producing output values that match the output signature for *varName*. Correct the service's signature and/or logic after identifying the source of the problem.

[B2BCORE.0076.9205] SOAP Message Coder cannot encode variable ***varName***; variable does not have a valid XML name

The message coder could not XML-encode the SOAP RPC request or response because the pipeline contained the variable *varName*, whose name is not a valid name for an XML element.

This error often occurs because the variable name contains a colon. Change the name of *varName* to one that is also valid as an XML element name (or map *varName* to a new variable, and use that variable as your SOAP RPC input or output parameter).

[B2BCORE.0076.9206] SOAP Message Coder error; no coder is registered for encoding style ***styleName***

The message coder encountered a serious internal error.

Contact SAP BCTechnical Support

[B2BCORE.0076.9208] SOAP Message Coder cannot decode message; cannot determine type of element ***elementName***

The message coder could not decode the XML-encoded parameters it has received, because the simple element *elementName* has an unknown data type (for example, `xsi:type;"myDataType"`).

Check the incoming XML and make sure the data type has been specified correctly. Also make sure that it is using the correct namespace.

[B2BCORE.0076.9209] SOAP Message Coder cannot decode message; unable to create array for arrayType attribute value ***attValue***

The message coder could not decode an XML-encoded array because it could not allocate an array of the specified dimensions.

Check the incoming XML and make sure that the arrayType attribute is specified correctly. For example, make sure that it does not specify a negative dimension, a zero dimension, or an invalid array size (e.g., too many dimensions or dimensions that are unrealistically large). If the array is unusually large or the server is extremely very low on memory, the possibility also exists that this error was the result of an out-of-memory condition.

[B2BCORE.0076.9210] SOAP Message Coder cannot decode message; arrayType attribute value ***attValue*** is malformed

The message coder could not decode an XML-encoded array because it cannot understand the value of the arrayType attribute.

Check the incoming XML and make sure that the arrayType attribute is using the correct syntax. For example, make sure that it is not missing a [symbol.

[B2BCORE.0076.9211] SOAP Message Coder cannot encode output data; pipeline does not match output signature, required variable ***varName*** is missing

The message coder could not transform the results of a SOAP remote procedure call because the required variable *varName* was not in the pipeline.

On the SAP BC server, examine the target service to determine why it is not producing the required output variables. Correct the service's signature and/or logic after identifying the source of the problem.

[B2BCORE.0076.9212] SOAP Message Coder cannot encode output data; pipeline does not match output signature, extra variable ***varName*** in pipeline

The message coder encountered a serious internal error.

Contact SAP BCTechnical Support

[B2BSERV.0088.9102] ASSERTION: The directive ***directiveName*** in the URL ***urlName*** should be ***directiveName***

The SOAP message handler encountered a serious internal error.

Contact SAP BCTechnical Support

[B2BSERV.0088.9103] Error reading SOAP Message from HTTP stream -- appears to be empty

The SOAP message handler encountered a serious internal error.

Contact SAP BCTechnical Support

[B2BSERV.0088.9105] Failed to register SOAP Handler ***directiveName***

The SAP BC server was not able to successfully add the specified SOAP processor to the processor registry.

Verify that the SOAP processor directive and fully qualified service name have been specified correctly. This error usually occurs because the directive name or the service name has not been specified.

[B2BSERV.0088.9107] Failed to unregister SOAP Handler ***directiveName***

The SAP BC server was not able to successfully remove the specified processor from the SOAP Processor registry.

This error generally occurs because the registry does not contain a processor registered to the specified directive. Verify that the process directive was specified correctly (Remember that directive names are case-sensitive.) Also verify that the specified directive is currently registered in the SOAP Processor registry. It may have already been removed.

[B2BSERV.0088.9108] SOAP Handler ***directiveName*** already registered

The SAP BC server was not able to successfully register the SOAP processor, because the registry already has a processor registered under the specified directive.

Register the SOAP processor under a different directive. Or unregister the processor that is currently registered to *directiveName*, and then register this SOAP Processor.

[B2BSERV.0088.9109] SOAP is only supported using HTTP POST protocol

The SOAP message handler was not able to accept the request because the SOAP message was submitted to server using the HTTP GET method. The SOAP message handler only accepts requests via the POST method (per W3C specification).

Modify the client to use HTTP POST to submit SOAP messages to the SAP BC server.

[B2BSERV.0088.9110] Invalid BOOLEAN value ***value*** for variable ***varName***; must be 0 or 1

The addHeaderEntry utility failed because the mustUnderstand attribute was not set correctly.

Check the logic in the process that failed to make sure that it sets the value of *varName* to either 0 or 1. No other values are permitted when setting the mustUnderstand attribute.

[B2BSERV.0088.9111] Cannot load or execute the SOAP processor ***directiveName***

The SOAP message handler cannot process the SOAP message because the requested SOAP processor is non-operational.

On the server, check the status of the service that is registered to *directiveName*. Make sure this service is loaded and operating successfully. (Frequently this error occurs because the processor was modified and then did not recompile successfully.)

[B2BSERV.0088.9112] An Exception was thrown in the server

The SAP BC server, the SOAP processor, or one of the services processing the message has thrown an exception.

See the detail element in the SOAP fault to determine the exception that was thrown and which process it was thrown by.

[B2BSERV.0088.9113] The SOAP Envelope does not have a Body block

The SOAP client was given a *soapRequestData* that did not contain a body element.

Examine the logic in the client to see whether it uses the stringToSoapData or streamToSoapData service to generate their SOAP objects. If so, determine whether the String or stream from which the SOAP object was generated represented a valid SOAP message.

Also check whether the SOAP processor or client used any of the data removal utilities (such as removeHeaderEntry or removeBodyEntry) and might have inadvertently removed the entire Body element.

[B2BSERV.0088.9114] Missing SOAP directive in the URL ***url***

The SOAP message handler encountered a serious internal error.

Contact SAP BCTechnical Support

[B2BSERV.0088.9116] Missing required parameter ***paramName***

The requested SOAP utility failed to execute because a required parameter was not supplied.

Check the logic in the SOAP processor or client to make sure that it passes the correct parameters to each SOAP utility service that it invokes.

[B2BSERV.0088.9117] One or more header entries were not understood by the SOAP processor

The SOAP processor has rejected the SOAP message because it cannot obey a mandatory header entry (a header entry whose mustUnderstand attribute is enabled).

[B2BSERV.0088.9118] Parameter ***paramName*** must be a valid soapData

The requested SOAP utility failed to execute because parameter *paramName* is not a SOAP object.

Examine the logic in the SOAP processor or client and make sure that it passes a SOAP object in parameter *paramName*.

[B2BSERV.0088.9119] SOAP Processor did NOT return a valid SOAP Response

The message handler did not receive a valid SOAP message from the SOAP processor.

Examine the logic in the SOAP processor to make sure that it composes the SOAP response message correctly. For example, check to see whether it the processor uses the stringToSoapData or streamToSoapData services to generate *soapResponseData*. If so, determine whether the original String or stream represents a valid SOAP message. If the SOAP processor logic seems correct, contact SAP BCTechnical Support. This would indicate a serious internal error.

[B2BSERV.0088.9120] Parameter ***paramName*** is not one of the valid data types: ***dataType***

The requested SOAP utility failed to execute because parameter *paramName* did not have the correct data type. The type in *dataType* is the data type that is expected.

Examine the logic in the SOAP processor or client and make sure that it passes to *paramName* an object of the data type specified in *dataType*.

[B2BSERV.0088.9122] Service ***serviceName*** does not exist

The SOAP facility could not complete the requested operation because it could not find the specified service. This error is issued by various SOAP processes (e.g., registering a SOAP processor, invoking a SOAP processor, invoking a target service).

Examine the process that failed and make sure that it specifies the name of the requested service correctly. For example, if you receive this error when registering a SOAP processor, check that you have specified the *svcName* parameter correctly. Remember that service names are case-sensitive.

[B2BSERV.0088.9123] Requested SOAP processor ***directiveName*** is not registered on this server

The SAP BC server could not process the SOAP message because it does not have the SOAP processor requested by the client.

On the client side, verify that the correct process directive is specified in the URL (remember that the directive is case-sensitive). If the client has specified the process directive correctly, then use pub.soap.processor:list to verify that the requested processor is registered on the SAP BC server.

[B2BSERV.0088.9124] SOAP Message does not conform to the SOAP message model

The SOAP message passed to the validateSoapData utility failed validation (for example, it is missing the Body element or the Header element follows the body).

Examine the logic that produced the message and see whether it uses the stringToSoapData or streamToSoapData service to manually generate a SOAP object instead of using the message composition services (e.g., createSoapData, addHeaderEntry, addBodyEntry). If so, determine whether the String or stream from which the SOAP object is generated represents a valid SOAP message.

Also check whether the SOAP processor or client used any of the data removal utilities (such as removeHeaderEntry or removeBodyEntry) and might have inadvertently removed required portions of the SOAP object (for example, the entire Body element).

[B2BSERV.0088.9125] SOAP request does not conform to the SOAP message model

The SOAP message handler could not process the SOAP request because it violates the SOAP message schema (for example, it is missing the Body element or the Header element follows the body).

On the client side, correct the logic that builds the message to ensure that it produces a valid SOAP message.

[B2BSERV.0088.9126] SOAP response does not conform to the SOAP message model

The SOAP message handler could not return the SOAP response generated by a SOAP processor on the SAP BC server because the message violates the SOAP message schema.

Examine the logic in the SOAP processor and see whether it uses the stringToSoapData or streamToSoapData service to generate a SOAP object. If so, determine whether the String or stream from which the SOAP object is generated represents a valid SOAP message.

Examine the logic that produced the message and see whether it uses the stringToSoapData or streamToSoapData service to manually generate a SOAP object instead of using the message composition services (e.g., createSoapData, addHeaderEntry, addBodyEntry). If so, determine whether the String or stream from which the SOAP object is generated represents a valid SOAP message.

Also check whether the SOAP processor or client used any of the data removal utilities (such as removeHeaderEntry or removeBodyEntry) and might have inadvertently removed required portions of the SOAP object (for example, the entire Body element).

[B2BSERV.0088.9127] The server could not load the SOAP XML Validator

The SOAP message handler encountered a serious internal error.

Contact SAP BCTechnical Support.

[B2BSERV.0088.9128] Request is from namespace ***'msgNamespace'***, server requires namespace ***'svrNamespace''***

The SOAP message handler could not process the message because the message is not from a version of SOAP that the SAP BC server supports. The SAP BC server supports the version indicated in *svrNamespace*.

From the client side, resubmit the message using a version of SOAP that the SAP BC server supports.

[B2BSERV.0088.9129] Invalid node: XML blocks must be in a container

The requested SOAP utility (e.g., addHeaderEntry, addBodyEntry) failed to execute because it received a node that does not represent an enclosed XML element.

This error can occur if the node is missing opening and closing tags (e.g., the node contains "`hello`" instead of "`<a>hello</a>`"). Examine the process that called the failing utility and make sure that it passes a valid node object to the SOAP utility.

[B2BSERV.0088.9130] Invalid node: XML blocks can only have a single root container

The requested SOAP utility (e.g., addHeaderEntry, addBodyEntry) failed to execute because it received a node that does not have a root element.

This error can occur if the node contains an XML fragment that represents uncontained elements (e.g., the node contains "`<b>hello</b> <c>world</c>`" instead of "`<a> <b>hello</b> <c>world</c> </a>`"). Examine the logic that called the failing utility and make sure that it passes a node containing only one top-level XML element.

[B2BSERV.0088.9131] Invalid node: must be well-formed XML

The requested SOAP utility (e.g., addHeaderEntry, addBodyEntry) failed to execute because it received a node that does not represent a well-formed XML document.

This error can occur if the node contains overlapping or missing tags (e.g., a node that contains "`<b>hello`" or "`<a> <b>hello <c>world</b></c> </a>`"). Examine the logic that called the failing utility and make sure that it passes a node containing well-formed XML.

[B2BSERV.0088.9132] Failed to register SOAP Handler ***directiveName***, illegal characters in directive name

The SAP BC server was not able to successfully register the SOAP processor, because *directiveName* contains characters that are not allowed in a directive name.

Register the SOAP processor under a name that is composed only of letters, digits, or the characters `-_.!~*'( )`.

[B2BSERV.0088.9133] Error while encoding RPC output

The message coder could not XML-encode the parameters associated with a SOAP remote procedure call.

See the detail element in the SOAP fault for information about the specific failure that occurred.

[B2BSERV.0088.9134] Exception occurred while processing the body of the message

The SAP BC server, the SOAP processor, or one of the services processing the SOAP message has thrown an exception.

See the detail element in the SOAP fault to determine the exception that was thrown and which process it was thrown by.

[B2BSERV.0088.9135] A WMDocument Exception was thrown in the server, usually because an XML block was not well-formed

The SOAP message handler, the soapHTTP service, or the soapRPC service has received a SOAP message that contains invalid XML.

Check the process that submitted the SOAP message to the SAP BC server and make sure that it is producing valid XML. This error usually occurs because of a basic errors in the XML document, such as missing tags or overlapping elements.

[B2BSERV.0088.9136] A WattEvaluationException was thrown by the XQL Query engine

The XQL query processor encountered a serious internal error.

Contact SAP BCTechnical Support.

[B2BSERV.0088.9137] Invalid input parameter; ***paramName*** must be a Record (IData)

The requested SOAP utility failed to execute because the object passed in *paramName* is not an IData object.

Check the logic in the SOAP processor or client to make sure that it passes an IData in *paramName*.

[B2BSERV.0088.9138] Input parameters do not conform to targetInputSignature: ***validationDetails***

The pub.client:soapRPC service could not submit the remote procedure call because the input parameters did not pass the data-validation process.

This error is thrown when a client supplies an invalid set of input parameters for a SOAP remote procedure call. Specifically, it indicates that the parameters submitted to pub.client:soapRPC in *reqParms* do not match the structure and constraints of the Record specified in *targetInputSignature*. Generally, you will want to code your client to detect this kind of error and take some type of corrective action when it occurs.

[B2BSERV.0088.9139] Output parameters do not conform to targetOutputSignature: ***validationDetails***

The SOAP RPC client could not submit the remote procedure call because the output parameters returned by the remote server did not pass the data-validation process.

This error is thrown when a client receives an invalid set of output parameters (results) from a SOAP remote procedure call. Specifically, it indicates that the parameters returned to pub.client:soapRPC in *respParms* do not match the structure and constraints of the Record specified in *targetOutputSignature*. Generally, you will want to code your client to detect this kind of error and take some type of corrective action when it occurs.

[B2BSERV.0088.9140] Invalid target signature ***recordName***, must be a Record

The SOAP RPC client could not process the remote procedure call because it was not able to validate the input or output parameters associated with the request. The object it was given for validation is not a Record.

Examine the logic in the SOAP RPC client and make sure that the object it specifies in *targetInputSignature* and/or *targetOutputSignature* is a Record. That is, these parameters, if used, specify the fully qualified name of a Record or a specification that exists on the SAP BC server.

[B2BSERV.0088.9141] Invalid target signature ***recordName***, record does not exist

The SOAP RPC client could not process the remote procedure call because it was not able to validate the input or output parameters associated with the request. The specified Record does not exist on the SAP BC server.

Examine the logic in the SOAP RPC client and make sure that *targetOutputSignature* and/or *targetOutputSignature* are correctly specified. (Remember that Record names are case-sensitive.). Also verify that the specified Record exists on the server and the package in which it resides is loaded and enabled.

Appendix B: Encoding/Decoding Data-Type Mapping

XML-to-Java Mappings (Decoding)

The following tables describe the Java objects that the message coder creates for the various XML Schema data types.

Note: The message coder generates the objects listed below only when a parameter is declared as an Object in the accompanying signature. This list does not apply to parameters that the signature declares as Strings. If the signature defines a parameter as a String or if a parameter is not defined in a signature, the message coder always renders it as a String.

Data types from http://schemas.xmlsoap.org/soap/encoding/

The following describes the Java objects to which *Simple Object Access Protocol (SOAP) 1.1 - W3C Note 08 May 2000* types are converted (i.e., types from namespace http://schemas.xmlsoap.org/soap/encoding/):

When the XML parameter is of type...	The message coder renders it as a...
ENTITIES	String
ENTITY	String
ID	String
IDREF	String
IDREFS	String
NCName	String
NMTOKEN	String
NMTOKENS	String
NOTATION	String
Name	String
QName	String
base64	byte[ ]
binary	String
boolean	Boolean
byte	Byte
century	String
date	GregorianCalendar
decimal	BigDecimal
double	Double
float	Float
int	Integer
integer	BigInteger
language	String

long	Long
month	String
negativeInteger	BigInteger
nonNegativeInteger	BigInteger
nonPositiveInteger	BigInteger
positiveInteger	BigInteger
recurringDate	String
recurringDay	String
recurringDuration	String
short	Short
string	String
time	GregorianCalendar
timeDuration	String
timeInstant	String
timePeriod	String
unsignedByte	Short
unsignedInt	Long
unsignedLong	BigInteger
unsignedShort	Integer
uriReference	String
year	String

Data types from http://www.w3.org/1999/XMLSchema

The following describes the Java objects to which the types defined in *XML Schema Datatypes - W3C Last Call Draft April 7 2000* are converted (i.e., types from namespace http://www.w3.org/1999/XMLSchema).

When the XML parameter is of type...	The message coder renders it as a...
ENTITIES	String
ENTITY	String
ID	String
IDREF	String
IDREFS	String
NCName	String
NMTOKEN	String
NMTOKENS	String
NOTATION	String
Name	String
QName	String
binary	String
boolean	Boolean

byte	Byte
century	String
date	GregorianCalendar
decimal	BigDecimal
double	Double
float	Float
int	Integer
integer	BigInteger
language	String
long	Long
month	String
negativeInteger	BigInteger
nonNegativeInteger	BigInteger
nonPositiveInteger	BigInteger
positiveInteger	BigInteger
recurringDate	String
recurringDay	String
recurringDuration	String
short	Short
string	String
time	GregorianCalendar
timeDuration	String
timeInstant	String
timePeriod	String
unsignedByte	Short
unsignedInt	Long
unsignedLong	BigInteger
unsignedShort	Integer
uriReference	String
year	String

Data types from http://www.w3.org/2000/10/XMLSchema

The following describes the Java objects to which *XML Schema Datatypes - W3C Candidate Recommendation Oct 24 2000* types are converted (i.e., types from namespace http://www.w3.org/2000/10/XMLSchema):

When the XML parameter is of type...	The message coder renders it as a...
CDATA	String
ENTITIES	String

When the XML parameter is of type…	The message coder renders it as a…
ENTITY	String
ID	String
IDREF	String
IDREFS	String
NCName	String
NMTOKEN	String
NMTOKENS	String
NOTATION	String
Name	String
QName	String
binary	String
boolean	Boolean
byte	Byte
century	String
date	GregorianCalendar
decimal	BigDecimal
double	Double
float	Float
int	Integer
integer	BigInteger
language	String
long	Long
month	String
negativeInteger	BigInteger
nonNegativeInteger	BigInteger
nonPositiveInteger	BigInteger
positiveInteger	BigInteger
recurringDate	String
recurringDay	String
recurringDuration	String
short	Short
string	String
time	GregorianCalendar
timeDuration	String
timeInstant	String
timePeriod	String
token	String
unsignedByte	Short
unsignedInt	Long

When the XML parameter is of type...	The message coder renders it as a...
unsignedLong	BigInteger
unsignedShort	Integer
uriReference	String
year	String

Data types from http://www.w3.org/2001/XMLSchema

The following describes the Java objects to which *XML Schema Datatypes - W3C Recommendation May 2 2001* types are converted (i.e., types from namespace http://www.w3.org/2001/XMLSchema):

When the XML parameter is of type...	The message coder renders it as a...
ENTITIES	String
ENTITY	String
ID	String
IDREF	String
IDREFS	String
NCName	String
NMTOKEN	String
NMTOKENS	String
NOTATION	String
Name	String
QName	QName
anyURI	String
base64Binary	String
boolean	Boolean
byte	Byte
date	GregorianCalendar
dateTime	GregorianCalendar
datetime	GregorianCalendar
decimal	BigDecimal
double	Double
duration	String
float	Float
gDay	String
gMonth	String
gMonthDay	String
gYear	String
gYearMonth	String

When the XML parameter is of type…	The message coder renders it as a…
int	Integer
integer	BigInteger
language	String
long	Long
negativeInteger	BigInteger
nonNegativeInteger	BigInteger
nonPositiveInteger	BigInteger
normalizedString	String
positiveInteger	BigInteger
short	Short
string	String
time	GregorianCalendar
token	String
unsignedByte	Short
unsignedInt	Long
unsignedLong	BigInteger
unsignedShort	Integer

Java-to-XML Mappings (Encoding)

The following describes how XML data types to which Java objects are encoded.

Note: The following data types are used only when message coder is encoding a parameter that is defined as an Object in the accompanying signature or is encoding a parameter that is not declared in a signature. If a parameter is defined as a String in the accompanying signature, it is always encoded according to its Content Type constraint.

Object of type…	Is encoded as type…	From namespace…
BigDecimal	decimal	http://www.w3.org/2001/XMLSchema
BigInteger	integer	http://www.w3.org/2001/XMLSchema
Boolean	boolean	http://www.w3.org/2001/XMLSchema
Byte	byte	http://www.w3.org/2001/XMLSchema
byte[ ]	base64	http://schemas.xmlsoap.org/soap/encoding/
Date	date	http://www.w3.org/2001/XMLSchema
Document	string	http://www.w3.org/2001/XMLSchema
Double	double	http://www.w3.org/2001/XMLSchema
ElementNode	string	http://www.w3.org/2001/XMLSchema
Float	float	http://www.w3.org/2001/XMLSchema
GregorianCalendar	datetime	http://www.w3.org/2001/XMLSchema
Hashtable	Hashtable	http://www.webmethods.com/2001/10/soap/encoding

Object of type...	Is encoded as type...	From namespace...
IData*	data	http://www.webmethods.com/2001/10/soap/encoding
Integer	int	http://www.w3.org/2001/XMLSchema
Long	long	http://www.w3.org/2001/XMLSchema
MBoolean	boolean	http://www.w3.org/2001/XMLSchema
MByte	byte	http://www.w3.org/2001/XMLSchema
MDouble	double	http://www.w3.org/2001/XMLSchema
MFloat	float	http://www.w3.org/2001/XMLSchema
MInteger	int	http://www.w3.org/2001/XMLSchema
MLong	long	http://www.w3.org/2001/XMLSchema
MShort	short	http://www.w3.org/2001/XMLSchema
QName	QName	http://www.w3.org/2001/XMLSchema
Short	short	http://www.w3.org/2001/XMLSchema
String	string	http://www.w3.org/2001/XMLSchema
Vector	Vector	http://www.webmethods.com/2001/10/soap/encoding

* Or any class that implements the IData interface (e.g. BasicData, Values).

SAP BC to XML Mappings (Encoding & Decoding)

Data types from http://www.webmethods.com/2001/10/soap/encoding

The following Java objects are encoded with webMethods-specific types (i.e., types from namespace http://www.webmethods.com/2001/10/soap/encoding).

Java Object	XML type
Hashtable	Hashtable
Vector	Vector
IData*	data

* Or any class that implements the IData interface (e.g. BasicData, Values).

Appendix C: SOAP-Related Server Parameters

SOAP Parameters

The following describes the server parameters that you can use to configure the way in which SOAP operates on the SAP BC server. For information about how to set server parameters, see the *SAP BC Server Administrator's Guide*.

watt.server.SOAP.directive

Specifies the directive that represents the SOAP message handler. The default is `soap`. This parameter determines the URL that is used to access SOAP processors on the SAP BC server.

```
http://hostName:portNum/[processDirective]
```

This parameter specifies this segment of the SOAP processor URL

watt.server.SOAP.validateSOAPMessage

Specifies whether the SOAP message handler will validate messages that it receives and sends. Set to true or false. Default is true.

When `watt.server.SOAP.validateSOAPMessage` is true, the message handler validates messages against the SOAP schema. Be aware that this validation process checks only that the message envelope is structured correctly—for example, it checks that the message has at least one body element and there is at most one header element. It does not validate the data carried by the message.

If you are operating in a production environment where the validity of the messages submitted to the server is assured, you might set `watt.server.SOAP.validateSOAPMessage` to false to optimize performance.

watt.server.SOAP.useMultiReference

Specifies how the message coder is to encode parameters that point to the same underlying data object in the pipeline. Set to true or false. Default is false.

For example, if two output parameters *cDate* and *oDate* point to the same underlying data in the pipeline, the message coder encodes the parameters like this if `watt.server.SOAP.useMultiReference` is true:

```
.
.
.
<cDate id="id1">03/15/2000</cDate>
<amt id="id2">300</amt>
<oDate href="#id1"></oDate>
<acct id=id4>cash</acct>
.
.
.
```

The *oDate* parameter is encoded as a reference

And like this if `watt.server.SOAP.useMultiReference` is false:

```
.
.
.
<cDate>03/15/2000</cDate>
<amt>300</amt>
<oDate>03/15/2000</oDate>
<acct>cash</acct>
.
.
.
```

The *oDate* parameter is encoded as an independent element

Note: This parameter affects only the way in which multi-referenced parameters are *encoded*. It does not affect the way in which they are *decoded*. The message coder always recognizes and decodes multi-referenced elements regardless of how `watt.server.SOAP.useMultiReference` is set.

INDEX