Oracle® Java CAPS HTTP Binding Component User's Guide

ORACLE®

Contents

Using the HTTP Binding Component

This guide provides an overview of the HTTP Binding Component, and includes details that are necessary to configure and deploy the binding component in a JBI project. The HTTP Binding Component provides connectivity for SOAP over HTTP in a JBI 1.0 compliant environment. The HTTP Binding Component is used as both a provider proxy to support connectivity to services in the JBI environment, and as a consumer proxy to invoke services. The HTTP Binding Component is a JSR 208-compliant JBI runtime component that implements the Java Business Integration component interfaces.

What You Need to Know

These links take you to what you need to know before you use the HTTP Binding Component.

Reference Information

These links take you to additional reference information about configuring and using the HTTP Binding Component.

About the HTTP Binding Component

The HTTP Binding Component provides connectivity for SOAP over HTTP in a JBI 1.0 compliant environment. The HTTP Binding Component is used as both a provider proxy to support connectivity to services in the JBI environment, and as a consumer proxy to invoke services.

The JBI platform enables software vendors to provide services that can be invoked by external components using different protocols. These services are represented as JBI service engines and implement the business logic of the service. JBI binding components implement the protocol transformations between abstract messages handled by the JBI Service Engines and concrete messages of the protocol.

The HTTP Binding Component enables external components to invoke services, hosted by the JBI platform, using SOAP messages over the HTTP/HTTPS protocol. It also allows JBI components to invoke external web services using the same SOAP over the HTTP/HTTPS protocol.

The transformation of abstract messages to SOAP messages occurs in the HTTP Binding Component. WSDL extensibility elements, as defined in the WSDL 1.1, SOAP 1.1, SOAP 1.2 specifications, and Basic Profile 1.1, are used to properly configure this transformation. These WSDL extensibility elements are part of the `binding` and `service` sections of WSDL documents. These WSDL documents are the major artifacts included in a Service Unit, the JBI deployable unit into a Binding Component. Both the `binding` and `service` sections of a WSDL document must be properly filled out to determine how the message is transformed and the destination of that message.

HTTP/SOAP Binding Architecture

The HTTP Binding Component's SOAP binding enables external components to invoke web services hosted by the JBI platform using SOAP messages over the HTTP protocol. It also enables JBI engines to exposes their services. The web service can be unsecured or secured, using HTTP over SSL, TLS, or some other technology.

The SOAP binding resides in the JBI framework. It leverages the services offered by the web container to receive incoming web service requests. The Binding components are based on standards and do not use any proprietary API to promote reusability across different J2EE based JBI platforms. The binding does not use any web container services for outbound web services requests.

The HTTP Binding Component is B.P.1.0 compliant. The binding component addresses Basic Profile 1.1 requirements for messaging, such as:

- XML representation of SOAP messages, including fault messages
- Transport level security
- SOAP processing model
- Use of SOAP in HTTP

The following diagram illustrates relationship between the HTTP Binding Component and the other components within the runtime environment.
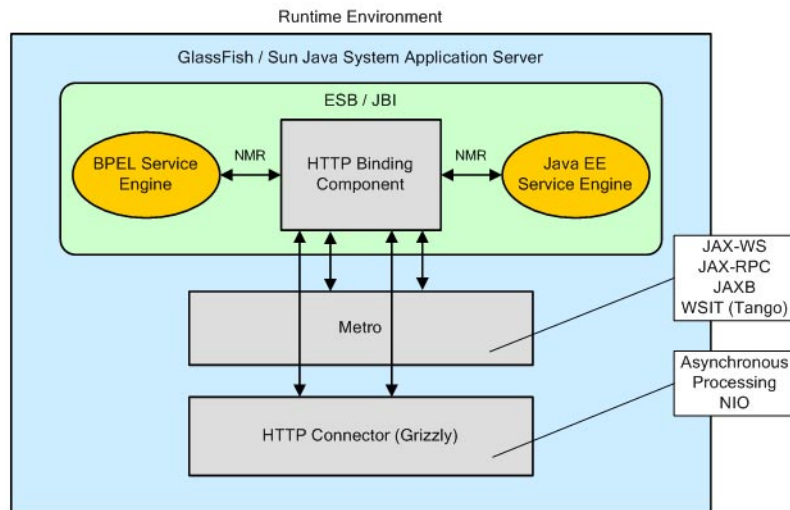
The Application Server contains the ESB/JBI container, as well as the Metro container that incorporates JAX-WS, JAX-RPC, JAXB, and Tango (WSIT), and the HTTP (Grizzly) container that adds Asynchronous Processing and NIO. The ESB/JBI container incorporates the HTTP Binding Component, the BPEL Service Engine, and the Java EE Service Engine, as well as other JBI Components. The HTTP/SOAP Binding Component provides interaction between the various containers and JBI components within the Application Server.

HTTP Binding Component Features

Features of the HTTP/SOAP Binding Component include the following:

- Supports the WSDL 1.1, SOAP 1.1, SOAP 1.2 specifications and Basic Profile 1.1. Message exchanges to and from this binding component make use of the JBI WSDL 1.1 wrapper for the normalized message
- Implements HTTP and SOAP binding as defined by the WSDL 1.1 specification

- Follows WS-I 1.0 conventions and adds additional support for nonconforming components
- Supports Document and Remote Procedure Call (RPC) style web services
- Supports literal and encoded use
- Supports the common convention of WSDL retrieval, such as `<service uri>?wsdl`
- Uses XML Catalogs following the OASIS Committee Specification, which enable the component to resolve schemas locally without resorting to network access
- Supports JBI service unit deployments to define the web services to provide or consume
- Makes use of the WSDL extensibility (standard HTTP and SOAP extensions) to define external communication details for the web services to provision or consume
- Supports Binding Component to Binding Component Connection, which allows direct HTTP Binding Component to HTTP Binding Component connections within a composite application
- Provides enhanced Application Variable Support
- Supports HTTP and HTTPS connections
- Provides Metro (JAX-WS, Tango/WSIT) integration to support WS-* features (such as WS-Security, WS-Addressing, WS-Reliable Messaging, WS-Transaction, and so forth) for incoming and outgoing requests
- Provides full JAX-WS integration for incoming and outgoing requests
- Supports GlassFish cluster mode

Service Provider Features

An HTTP Binding Component acting as a service provider supports sending HTTP or SOAP messages to an external web service.

Service provider features include:

- Outbound requests handled through the Java API for XML Web Services (JAX-WS)
- Support for proxy and proxy server authentication

Service Consumer Features

The HTTP Binding Component, acting as a service consumer, services HTTP requests or SOAP requests from HTTP clients, transforms (normalizes) them, and sends them to the normalized message router.

Service consumer features include:

- Uses asynchronous I/O (NIO) in the server to service thousands of concurrent incoming requests
- Packaged with an embedded HTTP server (Grizzly)
- Supports clustering

Security Features

Security features include support for transport level security through:

- Basic authentication
- SSL support

HTTP Binding Component Example Scenario

The following purchase order example illustrates how an HTTP Binding Component can be used in a composite application. In this example scenario, a single HTTP Binding Component acts as both a service provider and service consumer.

Purchase Order Example

A medical supply company provides a web site that contains a line of products for pre-approved customers only. One of these customers, a clinic, logs onto the web site and orders 1000 surgical masks and 2000 pairs of latex gloves. The purchase order is received and stored by the medical supply company's server, and a response is sent back to the clinic to confirm that the order has been received.

FIGURE 1   HTTP Binding Component Acting as Service Provider and Service Consumer
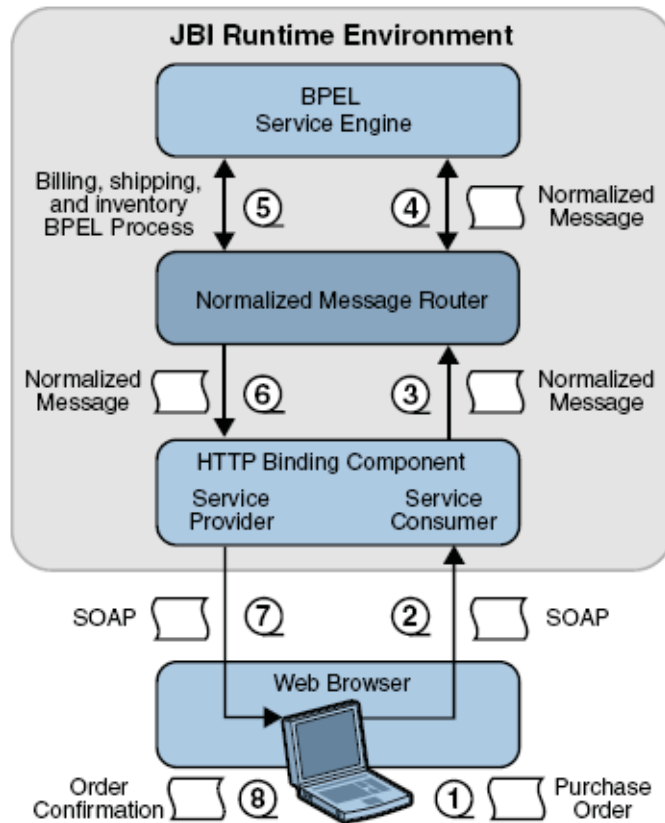
The purchase order handling system in this scenario is represented by a web service implemented using the GlassFish Application Server with the JBI framework.

The actors in this situation are as follows:

- **Web Client** — Packages up the purchase order as a SOAP request and sends it to the server.

- **HTTP Binding Component** — Sends and receives HTTP and SOAP messages.

- **BPEL Service Engine** — Responsible for implementing the core business logic of fulfilling the purchase order.

- **Normalized Message Router (NMR)** — Routes normalized messages between JBI components. In this scenario, it routes normalized messages to and from the HTTP BC and the BPEL SE.

Scenario Message Flow

The steps of the purchase order scenario message flow follow the numbers in figure 1.

1. The web client, using a client-side scripting language like the JavaScript, takes the purchase order information entered into the web form and packages it into a SOAP message. The format of the SOAP message is defined using a WSDL.

2. The SOAP message is sent to a web service endpoint hosted by the HTTP Binding Component.

3. The HTTP Binding Component transforms the SOAP message into a normalized message. The normalized message is sent to the Normalized Message Router.

4. The Normalized Message Router routes the normalized message to the BPEL Service Engine.

5. The BPEL Service Engine interprets the purchase order information and properly invokes other BPEL processes to fulfill the request.

6. The BPEL Service Engine creates a response message in the form of a Normalized Message. The normalized message is sent to the Normalized Message Router.

7. The HTTP Binding Component receives the response message and converts it to a SOAP message. The SOAP message is sent back to the web client as a proper response as defined by the WSDL.

8. The web client takes the response and creates a human-readable HTML page to inform the customer whether the purchase order was accepted or rejected.

SOAP Processing

The HTTP Binding Component provides external connectivity for SOAP over HTTP in a JBI 1.0 compliant environment. The HTTP Binding Component supports the SOAP 1.1 and SOAP 1.2 specifications and implements SOAP binding from the WSDL 1.1 specification. The HTTP Binding Component also supports Basic Profile 1.1, and SOAP Attachments.

This section includes the following topics:

SOAP 1.1 WSDL Extensibility Elements

The SOAP 1.1 WSDL elements enable you to configure SOAP Connectivity and SOAP Binding information for the HTTP Binding Component.

- The only "SOAP 1.1 Connectivity Element" on page 14 is "SOAP 1.1 address Element" on page 14
- "SOAP 1.1 Binding Elements" on page 14 elements include the following:
  - "SOAP 1.1 binding Element" on page 15
  - "SOAP 1.1 operation Element" on page 15
  - "SOAP 1.1 body Element" on page 16
  - "SOAP 1.1 fault Element" on page 18
  - "SOAP 1.1 header and headerfault Elements" on page 19

SOAP 1.1 Connectivity Element

The only SOAP 1.1 Connectivity element is the address element.

SOAP 1.1 address Element

The SOAP 1.1 address extensibility element specifies the address used to connect to the SOAP server.

TABLE 1   SOAP 1.1 address Element Attributes

Property	Description	Required or Optional	Example
location	A URL which indicates the address used to connect to the SOAP server	Required	http://myhost:7676/mars/kb423

The following example illustrates the use of the SOAP 1.1 address element.

```
<port binding="y:binding" name="soapEndpoint">
   <soap:address location="http://myhost:7676/some/additional/context" />
</port>
```

SOAP 1.1 Binding Elements

The SOAP 1.1 extensibility elements for binding abstract WSDL messages to SOAP messages fall into several sections or levels.

Each level signifies how the binding should occur:

- binding level — the configuration applies to the entire port type
- operation level — the configuration applies only to the operation
- message level — the configuration applies to that particular message, whether the message is input or output

SOAP 1.1 `binding` Element

The SOAP 1.1 `binding` element indicates that the binding is bound to the SOAP 1.1 protocol format: Envelope, Header and Body. This element does not indicate the encoding or format of the message, for example, that it necessarily follows section 5 of the SOAP 1.1 specification.

TABLE 2 SOAP 1.1 `binding` Element Attributes

Property	Description	Required or Optional	Example
transport	Indicates to which transport of SOAP this binding corresponds	Optional	http://schemas.xmlsoap.org/soap/http
style	Indicates the default style of this particular SOAP binding	Optional	rpc

The SOAP 1.1 `binding` element must be present when using the SOAP binding. The following example illustrates the use of the SOAP 1.1 `binding` element.

```
<definitions .... >
    <binding .... >
        <soap:binding transport="uri"? style="rpc|document"?>
    </binding>
</definitions>
```

The `style` attribute value is the default style attribute for each contained operation. If the style attribute is omitted, the value is assumed to be "document".

The value of the required `transport` attribute indicates the transport to use to deliver SOAP messages. The URI value `http://schemas.xmlsoap.org/soap/http` corresponds to the HTTP binding in the SOAP specification. Other URIs may be used here to indicate other transports such as SMTP, FTP, and so forth.

SOAP 1.1 `operation` Element

The SOAP 1.1 `operation` element provides binding information from the abstract operation to the concrete SOAP operation.

TABLE 3   SOAP 1.1 operation Element Attributes

Property	Description	Required or Optional	Example
soapAction	Indicates the soapAction that should be put into the HTTP header	Optional	urn:someSoapAction
style	Indicates the default style of this particular SOAP operation	Optional	rpc

The following example illustrates the use of the SOAP operation element.

```
<definitions .... >
    <binding .... >
        <operation .... >
            <soap:operation soapAction="uri"? style="rpc|document"?>?
        </operation>
    </binding>
</definitions>
```

The style attribute indicates whether the operation is RPC-oriented, with messages containing parameters and return values, or document-oriented, with messages containing documents. This information is used to select an appropriate programming model. The value of this attribute also affects the way in which the body of the SOAP message is constructed. If the attribute is not specified, it defaults to the value specified in the soap:binding element. If the soap:binding element does not specify a style, it is assumed to be "document".

The soapAction attribute specifies the value of the SOAPAction header for this operation. Use this URI value directly as the value for the SOAPAction header. do not attempt to make a relative URI value absolute when making the request. For the HTTP protocol binding of SOAP, this value is required and has no default value. For other SOAP protocol bindings, this value should not be specified, and the soap:operation element can be omitted.

SOAP 1.1 body Element

The SOAP 1.1 body element provides binding information from the abstract operation to the concrete SOAP operation.

TABLE 4   SOAP 1.1 body Element Attributes

Property	Description	Required or Optional	Example
parts	Indicates the parts from the WSDL message that will be included in the body element	Optional	part1

TABLE 4 SOAP 1.1 body Element Attributes   *(Continued)*

Property	Description	Required or Optional	Example
use	Indicates how message parts are encoded in the SOAP body	Optional	literal
encodingStyle	Indicates a particular encoding style to use	Optional	http://someEncodingStyle
namespace	Indicates the namespace of the wrapper element for RPC style messages	Optional	urn:someNamespace

The following example illustrates the SOAP 1.1 body element.

```
<definitions .... >
    <binding .... >
        <operation .... >
            <input>
                <soap:body parts="nmtokens"? use="literal|encoded"?
                            encodingStyle="uri-list"? namespace="uri"?>
            </input>
            <output>
                <soap:body parts="nmtokens"? use="literal|encoded"?
                            encodingStyle="uri-list"? namespace="uri"?>
            </output>
        </operation>
    </binding>
</definitions>
```

The optional parts attribute of type nmtokens indicates which parts appear somewhere within the SOAP body portion of the message. Other parts of a message may appear in other portions of the message, such as when SOAP is used in conjunction with the multipart/related MIME binding. If the parts attribute is omitted, then all parts defined by the message are assumed to be included in the SOAP Body portion.

The use attribute indicates whether the message parts are encoded using some encoding rules, or whether the parts define the concrete schema of the message.

If use is encoded, then each message part references an abstract type using the type attribute. These abstract types are used to produce a concrete message by applying an encoding that is specified by the encodingStyle attribute. The part names, types and value of the namespace attribute are all inputs to the encoding, although the namespace attribute only applies to content that is not explicitly defined by the abstract types. If the referenced encoding style allows variations in its format, as does the SOAP encoding, then all variations must be supported ("reader makes right").

If use is literal, then each part references a concrete schema definition using either the element or type attribute. In the first case, the element referenced by the part will appear directly under the body element for document style bindings, or under an accessor element named after the

message part in RPC style. In the second case, the type referenced by the part becomes the schema type of the enclosing element: body for document style or part accessor element for RPC style.

You can use the value of the encodingStyle attribute when the use is literal to indicate that the concrete format was derived using a particular encoding such as the SOAP encoding, but that only the specified variation is supported ("writer makes right").

The value of the encodingStyle attribute is a list of URIs, each separated by a single space. The URIs represent encodings used within the message, in order of most restrictive to least restrictive, like the encodingStyle attribute defined in the SOAP specification.

SOAP 1.1 `fault` Element

The fault element specifies the contents of SOAP Fault Details element. It is patterned after the body element.

TABLE 5 SOAP 1.1 fault Element Attributes

Property	Description	Required or Optional	Example
name	Indicates the name of the part from the WSDL message that will be included in the fault element	Required	part1
use	Indicates how message parts will be encoded in the SOAP fault	Required	literal
encodingStyle	Indicates a particular encoding style to use	Optional	http://someEncodingStyle
namespace	Indicates the namespace of the wrapper element for RPC style messages	Optional	urn:someNamespace

The following example illustrates the SOAP fault element.

```
<definitions .... >
    <binding .... >
        <operation .... >
            <fault>*
                <soap:fault name="nmtoken" use="literal|encoded"
                            encodingStyle="uri-list"? namespace="uri"?>
            </fault>
        </operation>
    </binding>
</definitions>
```

The name attribute relates the soap:fault to the wsdl:fault defined for the operation. The fault message must have a single part.

The use, encodingStyle, and namespace attributes are all used in the same way as those used with the body element, except that style="document" is assumed, because faults do not contain parameters.

SOAP 1.1 `header` and `headerfault` Elements

The header and headerfault elements enable you to define headers that are transmitted inside the header element of the SOAP Envelope. You do not have to exhaustively list all headers that appear in the SOAP Envelope using header. For example, extensions to WSDL may imply specific headers should be added to the actual payload and you do not have to list those headers here.

TABLE 6   SOAP 1.1 header Element Attributes

Property	Description	Required or Optional	Example
message	Indicates the WSDL message that will be used in binding to the header element	Required	part1
part	Indicates the parts from the WSDL message that will be included in the header element	Required	part1
use	Indicates how message parts will be encoded in the SOAP header	Required	literal
encodingStyle	Indicates a particular encoding style to use	Optional	http://someEncodingStyle
namespace	Indicates the namespace of the wrapper element for RPC style messages	Optional	urn:someNamespace

TABLE 7   SOAP 1.1 headerfault Element Attributes

Property	Description	Required or Optional	Example
name	Indicates the WSDL message that will be used in binding to the headerfault element	Required	part1
part	Indicates the parts from the WSDL message that will be included in the headerfault element	Required	part1

TABLE 7 SOAP 1.1 `headerfault` Element Attributes *(Continued)*

Property	Description	Required or Optional	Example
use	Indicates how message parts will be encoded in the SOAP headerfault	Required	literal
encodingStyle	Indicates a particular encoding style to use	Optional	http://someEncodingStyle
namespace	Indicates the namespace of the wrapper element for RPC style messages	Optional	urn:someNamespace

The following example illustrates the SOAP `header` and `headerfault` elements.

```
<definitions .... >
    <binding .... >
        <operation .... >
          <input>
            <soap:header message="qname" part="nmtoken" use="literal|encoded"
                        encodingStyle="uri-list"? namespace="uri"?>*
              <soap:headerfault message="qname" part="nmtoken" use="literal|encoded"
                            encodingStyle="uri-list"? namespace="uri"?/>*
            <soap:header>
          </input>
          <output>
            <soap:header message="qname" part="nmtoken" use="literal|encoded"
                        encodingStyle="uri-list"? namespace="uri"?>*
              <soap:headerfault message="qname" part="nmtoken" use="literal|encoded"
                            encodingStyle="uri-list"? namespace="uri"?/>*
            <soap:header>
          </output>
        </operation>
    </binding>
</definitions>
```

The `use`, `encodingStyle`, and `namespace` attributes are all used in the same way as those used with the body element, except that `style="document"` is assumed because headers do not contain parameters.

Together, the `message` attribute (of type `QName`) and the `part` attribute (of type `nmtoken`) reference the message part that defines the `header` type.

The optional `headerfault` elements that appear inside the header and have the same syntax as the header, enable you to specify the header types used to transmit error information pertaining to the header, and defined by the header. The SOAP specification states that errors pertaining to headers must be returned in the headers. This mechanism enables you to specify the format of such headers.

SOAP 1.2 WSDL Extensibility Elements

The SOAP 1.2 WSDL extensibility elements enable you to configure two sets of information for the HTTP Binding Component: SOAP 1.2 connectivity information, and binding information to convert WSDL messages to and from SOAP messages.

- The only "SOAP 1.2 Connectivity Element" on page 21 is "SOAP 1.2 address Element" on page 21
- "SOAP 1.2 Binding Elements" on page 22 elements include the following:
    - "SOAP 1.2 binding Element" on page 22
    - "SOAP 1.2 operation Element" on page 23
    - "SOAP 1.2 body Element" on page 24
    - "SOAP 1.2 fault Element" on page 25
    - "SOAP 1.2 header and headerfault Elements" on page 27

SOAP 1.2 Connectivity Element

The only SOAP 1.2 Connectivity element is the address element.

SOAP 1.2 address Element

The SOAP 1.2 address extensibility element specifies the URL address used to connect to the SOAP server.

TABLE 8   SOAP 1.2 address Element Attributes

Property	Description	Required or Optional	Example
location	A URL address used to connect to the SOAP server	Required	http://myhost:7676/mars/kb423

The following example illustrates the use of the SOAP 1.2 address element.

```
<service name="echoService">
      <port name="echoPort" binding="tns:echoBinding">
          <soap12:address location="http://localhost:9080/echoService/echoPort"/>
      </port>
   </service>
```

> **Note** – The required `location` attribute (of type `xs:anyURI`) is a URI at which the endpoint can be accessed. The value of the `location` attribute cannot be a relative URI. The URI scheme specified must correspond to the transport or transfer protocol specified by the `soap12:binding/@transport` attribute of the corresponding `wsdl:binding` of the containing `wsdl:port`.

SOAP 1.2 Binding Elements

The SOAP 1.2 extensibility elements for binding abstract WSDL messages to SOAP 1.2 messages fall into several sections or levels.

Each level signifies how the binding should occur:

- binding level — the configuration applies to the entire port type
- operation level — the configuration applies only to the operation
- message level — the configuration applies to that particular message, whether the message is input or output

SOAP 1.2 `binding` Element

The SOAP 1.2 extensibility elements, for binding abstract WSDL messages to SOAP 1.2 messages, fall into different sections or levels.

Each level signifies how the binding should occur:

- binding level — the configuration applies to the entire port type
- operation level — the configuration applies only to the operation
- message level — the configuration applies to that particular message, whether the message is input or output

TABLE 9   SOAP 1.2 `binding` Element Attributes

Property	Description	Required or Optional	Example
transport	Indicates to which transport of SOAP this binding corresponds	Optional	http://schemas.xmlsoap.org/soap/http
style	Indicates the default style of this particular SOAP binding	Optional	rpc

The SOAP 1.2 `binding` element must be present when using the SOAP binding. The following example illustrates the use of the SOAP 1.2 `binding` element.

```
<wsdl:definitions ...>
...
    <wsdl:binding ...>
      <soap12:binding style="rpc|document" ? transport="xs:anyURI"wsdl:
required="xs:boolean" ? />
```

Note – The code sample above was wrapped for display purposes.

The `style` attribute value is the default style attribute for each contained operation. If the style attribute is omitted, the value is assumed to be "document".

The value of the required `transport` attribute indicates the transport to use to deliver SOAP messages. The URI value `http://schemas.xmlsoap.org/soap/http` corresponds to the HTTP binding in the SOAP specification. Other URIs may be used here to indicate other transports such as SMTP, FTP, and so forth.

SOAP 1.2 `operation` Element

The SOAP 1.2 `operation` element provides binding information from the abstract operation to the concrete SOAP operation.

TABLE 10   SOAP 1.2 `operation` Element Attributes

Property	Description	Required or Optional	Example
soapAction	Indicates the action parameter carried in the application/soap+xml Content-Type header field	Optional	urn:someSoapAction
style	Indicates the default style of this particular SOAP operation	Optional	rpc
soapActionRequired	Indicates whether the value of the soapAction attribute is or is not required to be part of request message	Optional	true

The following example illustrates the use of the SOAP `operation` element.

```
<definitions ....>;
    <binding .... >;
        <operation .... >;
            <soap12:operation soapAction="xs:anyURI" ?
                              soapActionRequired="xs:boolean" ?
                              style="rpc|document" ?
                              wsdl:required="xs:boolean" ? /> ?
        </soap12:operation>
```

```
        </binding>;
</definitions>
```

The `style` attribute value, if present, is a string that specifies the style for the operation. The `style` attribute indicates whether the operation is RPC-oriented (a messages containing parameters and return values) or document-oriented (a message containing documents). If the `style` attribute is omitted from the `soap12:operation` element, then the operation inherits the style specified or implied by the `soap12:binding` element in the containing `wsdl:binding` element.

The `soapAction` attribute (of type `xs:anyURI`) specifies the value of the action parameter, carried in the `application/soap+xml` Content-Type header field, for this operation. The value of this attribute must be an absolute URI.

The `soapActionRequired` attribute (of type `xs:Boolean`), if present, indicates whether the value of the `soapAction` attribute is or is not required to be conveyed in the request message. If the `soapActionRequired` attribute is omitted, its value defaults to `true`. When the value of `soapActionRequired` is `true`, the `soapAction` attribute must be present

SOAP 1.2 `body` Element

The SOAP 1.2 body element specifies how the message parts appear within the SOAP body element.

TABLE 11   SOAP 1.2 body Element Attributes

Property	Description	Required or Optional	Example
parts	Indicates the parts from the WSDL message that will be included in the body element	Optional	part1
use	Indicates how message parts are encoded in the SOAP body	Optional	literal
encodingStyle	Indicates a particular encoding style to use	Optional	http://someEncodingStyle
namespace	Indicates the namespace of the wrapper element for RPC style messages	Optional	urn:someNamespace

The following example illustrates the SOAP 1.2 body element.

```
<wsdl:definitions ... >

  <wsdl:binding ... >
    <wsdl:operation ... >
        <wsdl:input>
            <soap12:body parts="soap12:tParts" ?
```

```
                              namespace="xs:anyURI" ?
                              use="literal|encoded" ?
                              encodingStyle="xs:anyURI" ? ... />
          ...
        </wsdl:input>
        <wsdl:output>
            <soap12:body parts="soap12:tParts ?
                    namespace="xs:anyURI" ?
                    use="literal|encoded" ?
                    encodingStyle="xs:anyURI" ? ... />
          ...
        </wsdl:output>
    </wsdl:operation>
    ...
  </wsdl:binding>
...
</wsdl:definitions>
```

The optional `parts` attribute (of type `soap12:tParts`, which is a list of xs:NMTOKENs) indicates which message parts are bound to the SOAP 1.2 body element of the message. Other message parts may be bound to other portions of the message, such as when SOAP is used in conjunction with the multipart/related MIME binding, or when bound as SOAP header blocks. If the `parts` attribute is omitted, then all of the parts defined by the associated wsdl:message are assumed to be included in the SOAP body.

The `use` attribute, if present, indicates whether the message parts are encoded using some encoding rules, or the parts define the concrete schema of the message. If the value is "encoded" the message parts are encoded using encoding rules that are specified by the value, actual or implied, of the `encodingStyle` attribute. If the value is "literal" then the message parts are literally defined by the schema types referenced.

The `encodingStyle` attribute (of type `xs:anyURI`), if present, identifies the set of encoding rules used to construct the message. This attribute must not be present unless the `style` attribute of the `soap12:binding` element of the containing wsdl:binding has a value of "rpc" and the `use` attribute on the containing `soap12:body` element has a value of "encoded". The value of the `encodingStyle` attribute, if present, must not be a relative URI.

SOAP 1.2 `fault` Element

The `fault` element specifies the contents of SOAP 1.2 Fault Details element. It is patterned after the `body` element.

TABLE 12 SOAP 1.2 `fault` Element Attributes

Property	Description	Required or Optional	Example
name	Indicates the name of the part from the WSDL message that will be included in the fault element	Required	part1

TABLE 12 SOAP 1.2 `fault` Element Attributes *(Continued)*

Property	Description	Required or Optional	Example
use	Indicates how message parts will be encoded in the SOAP 1.2 fault	Required	literal
encodingStyle	Indicates a particular encoding style to use	Optional	http://someEncodingStyle
namespace	Indicates the namespace of the wrapper element for RPC style messages	Optional	urn:someNamespace

The following example illustrates the SOAP `fault` element.

```
<wsdl:definitions ... >
...
    <wsdl:binding ... >
    ...
        <wsdl:operation ... >
        ...
            <wsdl:fault ... >*
                <soap12:fault name="xs:NMTOKEN"
                        namespace="xs:anyURI" ?
                        use="literal|encoded" ?
                        encodingStyle="xs:anyURI" ? ... />
            ...
        </wsdl:fault>
    </wsdl:operation>
    ...
    </wsdl:binding>
...
</wsdl:definitions>
```

The name attribute (of type `xs:NMTOKEN`) associates the corresponding `wsdl:fault` defined in the `wsdl:portType` for the containing `wsdl:operation`.

The use attribute, if present, indicates whether the message parts are encoded using some encoding rules, or whether the parts define the concrete schema of the message. If the value is "encoded" the message parts are encoded using some encoding rules as specified by the value, actual or implied, of the `encodingStyle` attribute. If the value is "literal" then the message parts are literally defined by the schema types referenced.

The namespace attribute (of type `xs:anyURI`), if present, defines the namespace to be assigned to the `wrapper` element for the fault. This attribute is ignored if the `style` attribute of either the `soap12:binding` element of the containing `wsdl:binding` or of the `soap12:operation` element of the containing `wsdl:operation` is either omitted or has a value of "document". This attribute must be present if the value of the `style` attribute of the `soap12:binding` element of the containing `wsdl:binding` is "rpc". The value of the `namespace` attribute must not be a relative URI.

The encodingStyle attribute (of type xs:anyURI), if present, identifies the set of encoding rules used to construct the fault message. This attribute must not be present unless the style attribute of the soap12:binding element of the containing wsdl:binding has a value of "rpc" and the use attribute on the containing soap12:body element has a value of "encoded". The value of the encodingStyle attribute must not be a relative URI.

SOAP 1.2 `header` and `headerfault` Elements

The header and headerfault elements enable you to define headers that are transmitted inside the header element of the SOAP Envelope. You do not have to exhaustively list all headers that appear in the SOAP 1.2 Envelope using header. For example, extensions to WSDL may imply specific headers should be added to the actual payload and you do not have to list those headers here.

TABLE 13 SOAP 1.2 header Element Attributes

Property	Description	Required or Optional	Example
message	Indicates the WSDL message that will be used in binding to the header element	Required	part1
part	Indicates the parts from the WSDL message that will be included in the header element	Required	part1
use	Indicates how message parts will be encoded in the SOAP header	Required	literal
encodingStyle	Indicates a particular encoding style to use	Optional	http://someEncodingStyle
namespace	Indicates the namespace of the wrapper element for RPC style messages	Optional	urn:someNamespace

The following example illustrates the SOAP 1.2 header element.

```
<wsdl:definitions ... >
...
    <wsdl:binding ... >
...
        <wsdl:operation ... >
...
            <wsdl:input ... >*
                <soap12:header message="xs:QName"
                        part="xs:NMTOKEN"
                        use="literal|encoded"
                        namespace="xs:anyURI" ?
```

```
                            encodingStyle="xs:anyURI" ? ... /> *
...
            </wsdl:input>
            <wsdl:output ... >*
                <soap12:header message="xs:QName"
                        part="xs:NMTOKEN"
                        use="literal|encoded"
                        namespace="xs:anyURI" ?
                        encodingStyle="xs:anyURI" ? ... /> *
...
            </wsdl:output>
        </wsdl:operation>
...
    </wsdl:binding>
...
</wsdl:definitions>
```

The `message` attribute (of type `xs:QName`), together with the `parts` attribute, indicates which message parts are bound as children of the SOAP 1.2 `header` element of the message. The referenced message does not need to be the same as the message that defines the SOAP Body.

The `parts` attribute (of type `xs:NMTOKEN`), together with the `message` attribute, indicates which message part is bound as a child of the SOAP 1.2 `header` element of the message.

The `namespace` attribute (of type `xs:anyURI`), if present, defines the namespace to be assigned to the header element serialized with `use="encoded"`. In all cases, the header is constructed as if the `style` attribute of the `wsoap12:binding` element, of the containing `wsdl:binding`, has a value of "document". The value of the `namespace` attribute, if present, must not be a relative URI.

The `use` attribute indicates whether the message parts are encoded using some encoding rules, or whether the parts define the concrete schema of the message. If the value is "encoded" the message parts are encoded using some encoding rules as specified by the value, actual or implied, of the `encodingStyle` attribute. If the value is "literal" then the message parts are literally defined by the schema types referenced.

The `encodingStyle` attribute (of type `xs:anyURI`), if present, identifies the set of encoding rules used to construct the message. This attribute must not be present unless the `style` attribute, of the `soap12:binding` element, of the containing `wsdl:binding`, has a value of "rpc" and the `use` attribute, on the containing `soap12:body` element, has a value of "encoded". The value of the `encodingStyle` attribute, if present, must not be a relative URI.

`/soap12:header/@{any}}}` is an extensibility mechanism that enables additional attributes, that are defined in a foreign namespace, to be added to the element.

Optional `soap12:headerfault` elements, which appear inside `wsoap12:header` elements, specify the header types used to transmit error information pertaining to the header, defined by the `soap12:header`.

TABLE 14 SOAP 1.2 `headerfault` Element Attributes

Property	Description	Required or Optional	Example
message	Indicates the WSDL message that will be used in binding to the headerfault element	Required	part1
part	Indicates the parts from the WSDL message that will be included in the headerfault element	Required	part1
use	Indicates how message parts will be encoded in the SOAP headerfault	Required	literal
encodingStyle	Indicates a particular encoding style to use	Optional	http://someEncodingStyle
namespace	Indicates the namespace of the wrapper element for RPC style messages	Optional	urn:someNamespace

The following example illustrates the SOAP 1.2 `headerfault` element.

```
<wsdl:definitions ... >
...
    <wsdl:binding ... >
...
        <wsdl:operation ... >
...
            <wsdl:input ... >

                <soap12:header ... >
                    <soap12:headerfault message="xs:QName"
                                part="xs:NMTOKEN"
                                use="literal|encoded"
                                namespace="xs:anyURI" ?
                                encodingStyle="xs:anyURI" ?
                                        ... /> *
                </soap12:header> *
...
            </wsdl:input> *
            <wsdl:output ... >
                <soap12:header ... >
                    <soap12:headerfault message="xs:QName"
                                part="xs:NMTOKEN"
                                use="literal|encoded"
                                namespace="xs:anyURI" ?
                                encodingStyle="xs:anyURI" ?
                                        ... /> *
                </soap12:header> *
...
            </wsdl:output> *
```

```
        </wsdl:operation>
...
    </wsdl:binding>
...
</wsdl:definitions>
```

The `headerfault` elements, which appear inside header and have the same syntax as header, enable you to specify the header types that are used to transmit error information pertaining to the header, defined by the header. The SOAP specification states that errors pertaining to headers must be returned in headers, and this mechanism enables you to specify the format of such headers.

The `message` attribute (of type `xs:QName`), together with the `parts` attribute, indicates which message part is to be bound as a child of the SOAP 1.2 `header` element of the message, for returning faults pertaining to the enclosing `soap12:header`. The referenced message does not need to be the same as the message that defines the SOAP Body.

The `parts` attribute (of type `xs:NMTOKEN`), together with the `message` attribute, indicates which message part is to be bound as a child of the SOAP 1.2 `header` element of the message for returning faults pertaining to the enclosing `soap12:header`.

The `namespace` attribute (of type `xs:anyURI`), if present, defines the namespace to be assigned to the wrapper element for an `rpc-style` operation. This attribute is ignored if the `style` attribute of either the `soap12:binding` element, of the containing `wsdl:binding` or of the `soap12:operation` element of the containing `wsdl:operation`, is either omitted or has a value of "document". This attribute must be present if the value of the `style` attribute of the `soap12:binding` element of the containing `wsdl:binding` is "rpc". The value of the `namespace` attribute, must not be a relative URI.

The `use` attribute indicates whether the message parts are encoded using some encoding rules, or whether the parts define the concrete schema of the message. If the value is "encoded" the message parts are encoded using some encoding rules, as specified by the value, actual or implied, of the `encodingStyle` attribute. If the value is "literal" then the message parts are literally defined by the schema types referenced.

The `encodingStyle` attribute (of type `xs:anyURI`), if present, identifies the set of encoding rules used to construct the message. This attribute must not be present unless the `style` attribute, of the `soap12:binding` element, of the containing `wsdl:binding`, has a value of "rpc" and the `use` attribute on the containing `soap12:body` element has a value of "encoded". The value of the `encodingStyle` attribute must not be a relative URI.

`/soap12:headerfault/@{any}}}` is an extensibility mechanism that enables additional attributes, defined in a foreign namespace, to be added to the element.

WS-I Basic Profile 1.1

The HTTP Binding Component conforms to the Basic Profile 1.1 Web service specification.

HTTP Processing

The HTTP extensibility elements describe interactions between a web browser and a web site through the HTTP 1.1 GET and POST verbs. These elements enable applications other than web browsers to interact with the site.

This section includes the following topics:

- "HTTP WSDL Extensibility Elements" on page 31
- "HTTP Connectivity Element" on page 31
- "HTTP Binding Elements" on page 32

HTTP WSDL Extensibility Elements

The HTTP WSDL elements enable you to configure HTTP Connectivity and HTTP Binding information for the HTTP Binding Component:

- The only "HTTP Connectivity Element" on page 31 is the "HTTP address Element" on page 31
- "HTTP Binding Elements" on page 32 elements include the following:
    - "HTTP binding Element" on page 32
    - "HTTP operation Element" on page 33
    - "HTTP urlEncoded Element" on page 33
    - "HTTP urlReplacement Element" on page 34

You can specify the following protocol-specific information:

- An indication that a binding uses HTTP GET or POST

- An address for the port

- A relative address for each operation (relative to the base address defined by the port)

HTTP Connectivity Element

The only HTTP Connectivity element is the address element.

HTTP address Element

The HTTP address extensibility element enables you to specify connectivity information to the HTTP server.

TABLE 15   HTTP `address` Element Attributes

Property	Description	Required or Optional	Example
location	A URL that specifies the connectivity information to connect to the HTTP server.	Required	`http://myhost:7676/some/additional/context`

The following example illustrates the use of the HTTP `address` extensibility element defined for a service port.

```
<port binding="y:binding" name="soapEndpoint">
   <http:address location="http://myhost:7676/some/additional/context" />
</port>
```

HTTP Binding Elements

The HTTP extensibility elements for binding abstract WSDL messages to HTTP messages fall into several sections.

Each section signifies how the binding should occur.

- binding level — applies to the entire port type
- operation level — applies only to the operation
- message level — applies to a particular message, whether it is input or output.

HTTP `binding` Element

The HTTP `binding` element specifies that the binding is bound to the HTTP protocol.

TABLE 16   HTTP `binding` Element Attributes

Property	Description	Required or Optional	Example
verb	Indicates to which transport of HTTP this binding corresponds.	Required	GET

The HTTP `binding` element must be present when using the HTTP binding. The following example illustrates the HTTP `binding` element.

```
<definitions .... >
    <binding .... >
        <http:binding verb="nmtoken" />
    </binding>
</definitions>
```

The value of the required verb attribute indicates the HTTP verb. Common values are GET or POST, but others may be used. Note that HTTP verbs are case sensitive.

HTTP `operation` Element

The HTTP operation element provides binding information from the abstract operation to the concrete HTTP operation.

TABLE 17 HTTP operation Element Attributes

Property	Description	Required or Optional	Example
location	Indicates the relative URI. Combined with the address location attribute.	Required	o1

The following example illustrates the WSDL operation element.

```
<definitions .... >
    <binding .... >
        <operation .... >
            <soap:operation location="uri" />
        </operation>
    </binding>
</definitions>
```

The location attribute specifies a relative URI for the operation. This URI is combined with the URI specified in the http:address element to form the full URI for the HTTP request. The URI value must be a relative URI.

HTTP `urlEncoded` Element

The urlEncoded element indicates that all of the message parts are encoded into the HTTP request URI using the standard URI-encoding rules (name1=value&name2=value...). The names of the parameters correspond to the names of the message parts. Each value contributed by the part is encoded using a name=value pair. You can use this value with GET to specify URL encoding, or with POST to specify a FORM-POST. For GET, the "?" character is automatically appended as necessary.

Example:

```
<definitions .... >
    <binding .... >
        <operation .... >
            <input .... >
                <http:urlEncoded/>
            </input>
            <output .... >
                <-- mime elements -->
```

```
            </output>
        </operation>
    </binding>
</definitions>
```

HTTP `urlReplacement` Element

The `urlReplacement` element indicates that all the message parts are encoded into the HTTP request URI using a replacement algorithm:

- The relative URI value of `http:operation` is searched for a set of search patterns.
- The search occurs before the value of the `http:operation` is combined with the value of the location attribute from `http:address`.
- There is one search pattern for each message part. The search pattern string is the name of the message part surrounded with parenthesis "(" and ")".
- For each match, the value of the corresponding message part is substituted for the match at the location of the match
- Matches are performed before any values are replaced. Replaced values do not trigger additional matches.

Message parts must not have repeating values.

Example:

```
<definitions .... >
    <binding .... >
        <operation .... >
            <input .... >
                <http:urlReplacement/>
            </input>
            <output .... >
                <-- mime elements -->
            </output>
        </operation>
    </binding>
</definitions>
```

HTTP GET and POST Processing

The HTTP Binding Component supports HTTP Binding as defined by the WSDL 1.1 specification. For more information on the WSDL 1.1 specification for the HTTP/SOAP Binding, refer to WSDL 1.1 Specification .

The binding component supports the following message processing:

- "XML/HTTP GET Processing" on page 35
- " Using the HTTP Binding Component with the HTTP POST Method" on page 39

XML/HTTP GET Processing

The HTTP Binding Component is used as a provider proxy to provide connectivity to services in the JBI environment, or as consumer proxy to invoke services. The binding component implements the HTTP 1.1 GET binding defined in the WSDL 1.1 Specification, enabling applications to consume or provide services from the JBI environment using a web browser-like HTTP GET interaction.

Configuring the HTTP Binding Component for HTTP Get Interactions

To configure the HTTP Binding Component to function for HTTP GET interactions, the WSDL file of the service to which the binding component is acting as proxy, needs to use the following HTTP binding language elements defined in the WSDL 1.1 Specification:

- An `<http:binding>` element indicating that a WSDL binding uses HTTP GET.
- An `<http:address>` element representing the address of the port.
- An `<http:operation>` element representing a relative address for each operation, that is relative to the `<http:address>` defined by the port.
- An `<http:urlEncoded>` and `<http:urlReplacement>` element to indicate how all of the message parts of a request are encoded and made a part of the HTTP request URI.

Examples that demonstrate how to configure the HTTP Binding Component as a provider proxy or consumer proxy are available at Using the HTTP Binding Component with the HTTP GET method (`http://wiki.open-esb.java.net/Wiki.jsp?page=HttpBcGetInteraction`) and Using the HTTP Binding Component with the HTTP POST Method (`http://wiki.open-esb.java.net/Wiki.jsp?page=HttpBcPostInteraction`).

Binding Details

The following sections describe how the HTTP Binding Component supports and implements the HTTP Binding language elements. Unless indicated otherwise, each of the WSDL elements described below are defined in the namespace `http://schemas.xmlsoap.org/wsdl/http/`.

http:binding Element

The `<http:binding>` element:

- Indicates that the binding uses the HTTP protocol.
- Must be specified as a subordinate element of a `<wsdl:binding>`

Note – Currently the HTTP Binding Component only supports GET and POST values (please note that HTTP methods are case-sensitive).

Example:

```
<http:binding verb="POST"/>
```

http:address Element

The `<http:address>` element:

- Represents the address of the port.
- Must be specified as a subordinate element of a `<wsdl:port>`
- Requires a location attribute that specifies the base URI for the port.

Example:

```
<http::address location="http://localhost/MyService/MyPort"/>
```

http:operation Element

The `<http:operation>` element:

- Represents an address for an operation.

- Must be specified as a subordinate element of a `<wsdl:operation/>`

- Requires a location attribute that specifies the base URI for the operation.

- The full URI for a request to this port and to this specific operation. The `<http:address>` element addresses the port, and since a port can have multiple operations, this element represents a relative address to a particular operation; the value of this element's location attribute is appended to the value of the`<http:address>` location attribute to form the full URI for a request to this port and to this specific operation.

The HTTP Binding Component supports a blank location attribute value for this element.

Example:

```
Given:
    <http:operation location="Submit">
    <http:address location="http://localhost/MyService/MyPort">

The full HTTP request URI is:
http://localhost/MyService/MyPort/Submit
```

http:urlEncoded Element

The <http:urlEncoded> element:

- Indicates that all message parts that make up the input message are encoded into the request URI using query-string encoding, as follows:
  - Name-value pair formatted as name=value
  - Ampersand-delimited name-value pairs:name1=value1&name2=value2&...
  - Message part names comprise the names in each pair, and message part values comprise the values in each pair.
- Must be specified as a subordinate element of a <wsdl:input/>
- Requires a location attribute that specifies the base URI for the operation.
- Is only supported by the HTTP Binding Component for the GET method. <http:urlEncoded> is ignored for the POST method. For more information, see POST URL Processing of the HTTP WSDL Binding Implementation (http://wiki.open-esb.java.net/Wiki.jsp?page=HttpPostUrlProcessing).

Example:

```
Given this description:
<wsd:message name="MyMessage">
    <wsdl:part name="partA" type="xsd:string"/>
    <wsdl:part name="partB" type="xsd:string"/>
</wsdl:message>
...
<wsdl:portType name="MyPortType">
    <wsdl:operation name="MyOperation">
        <wsdl:input message="MyMessage"/>
    </wsdl:operation>
</wsdl:portType>
...
<wsdl:binding name="MyBinding" type="MyPortType">
    <http:binding verb="GET"/>
    <wsdl:operation name="MyOperation">
        <wsdl:input>
            <http:urlEncoded/>
        </wsdl:input>
    </wsdl:operation>
</wsdl:binding>
...
<wsdl:service name="MyService">
    <wsdl:port name="Port1" binding="MyBinding">
        <http:address location="http://localhost/MyService/MyPort"/>
    </wsdl:port>
</wsdl:service>

Given these values mapped to the input parts:
"valueY" -> "partA"
"valueZ" -> "partB"

The full HTTP request URI is:
http://localhost/MyService/MyPort/MyOperation/partA=valueY&partB=valueZ
```

http:urlReplacement

The `<http:urlReplacement/>` element:

- Indicates that all message parts that make up the input message are encoded into the request URI using the replacement algorithm detailed in the WSDL 1.1 Specification.

- Must be specified as a subordinate element of a `<wsdl:input/>`

- Requires a location attribute that specifies the base URI for the operation.

- Is only supported by the HTTP Binding Component for the GET method. `<http:urlEncoded>` is ignored for the POST method. For more information, see POST URL Processing of the HTTP WSDL Binding Implementation (`http://wiki.open-esb.java.net/Wiki.jsp?page=HttpPostUrlProcessing`).

Example:

```
Given this description:
<wsd:message name="MyMessage">
    <wsdl:part name="partA" type="xsd:string"/>
    <wsdl:part name="partB" type="xsd:string"/>
</wsdl:message>
...
<wsdl:portType name="MyPortType">
    <wsdl:operation name="MyOperation">
        <wsdl:input message="MyMessage"/>
    </wsdl:operation>
</wsdl:portType>
...
<wsdl:binding name="MyBinding" type="MyPortType">
    <http:binding verb="GET"/>
    <wsdl:operation name="MyOperation/(partA)/subcategory/(partB)">
        <wsdl:input>
            <http:urlReplacement/>
        </wsdl:input>
    </wsdl:operation>
</wsdl:binding>
...
<wsdl:service name="MyService">
    <wsdl:port name="Port1" binding="MyBinding">
        <http:address location="http://localhost/MyService/MyPort"/>
    </wsdl:port>
</wsdl:service>

Given these values mapped to the input parts:
"valueY" -> "partA"
"valueZ" -> "partB"

The full HTTP request URI is:
http://localhost/MyService/MyPort/MyOperation/valueY/subcategory/valueZ
```

Using the HTTP Binding Component with the HTTP POST Method

The HTTP BC implements the HTTP 1.1 POST binding defined in the WSDL 1.1 specification, enabling applications to consume or provide services from the JBI environment using a web browser-like HTTP GET interaction.

Configuring the HTTP Binding Component for HTTP Get Interactions

To configure the HTTP Binding Component to function for HTTP POST interactions, the WSDL file of the service to which the binding component is acting as proxy, needs to use the following HTTP binding language elements defined in the WSDL 1.1 Specification:

- An <http:binding> element indicating that a WSDL binding uses HTTP POST.
- An <http:address> element representing the address of the port.
- An <http:operation> element representing a relative address for each operation, that is relative to the <http:address> defined by the port.
- An <http:urlEncoded> and <http:urlReplacement> element to indicate how all of the message parts of a request are encoded and made a part of the HTTP request URI.

Examples that demonstrate how to configure the HTTP Binding Component as a provider proxy or consumer proxy are available at Using the HTTP Binding Component with the HTTP GET method and Using the HTTP Binding Component with the HTTP POST method.

Note – Currently the HTTP Binding Component only supports the use of <http:urlReplacement> and <http:urlEncoded> with HTTP GET.

Binding Details

For information on the Binding details for these elements, see "Binding Details" on page 35.

HTTP POST Treatment of http:urlEncoded and http:urlReplacement

The HTTP Binding Component does not use the WSDL HTTP Binding consistently across GET and POST-style interactions, due to request structure differences between GET and POST requests.

The differences are:

- GET requests do not carry additional data aside from what is included in the URL (and in the HTTP headers).
- POST requests can send additional data in the request entity body. For example, when a web browser is used to submit a form (or upload a file through a form) by POST, the form data, or the contents of the file, is sent as the body of the request. The data is not made part of the request URL.

Because of these differences, the current HTTP Binding Component implementation considers `http:urlEncoded` and `http:urlReplacement` to be meaningful only when used in conjunction with HTTP GET, because these binding elements refer to URL encoding styles that apply only to GET requests.

For HTTP POST, the current implementation ignores both http:urlEncoded and http:urlReplacement binding elements.

HTTP Binding Component Runtime Properties

The HTTP Binding Component's runtime properties can be configured from the NetBeans IDE, or from a command prompt (command line interface) during installation.

The HTTP Binding Component properties apply to the binding component as a whole, including all provider and consumer endpoints.

To display or edit the properties in the NetBeans IDE, do the following:

1. From the Services tab of the NetBeans IDE, expand the Servers node.

2. Start your application server. To do this, right-click your application server and select Start from the shortcut menu.

3. Under the application server, expand the JBI → Binding Components nodes and select the HTTP Binding Component (com.sun.httpsoapbc). The current HTTP Binding Component properties are displayed at the right side of the NetBeans IDE. You can also double-click the HTTP Binding Component to open a properties window.

4. Edit the properties as needed. To apply any changes you make to the runtime HTTP Binding Component properties, stop and restart the HTTP Binding Component.

FIGURE 2 HTTP Binding Component Runtime Properties
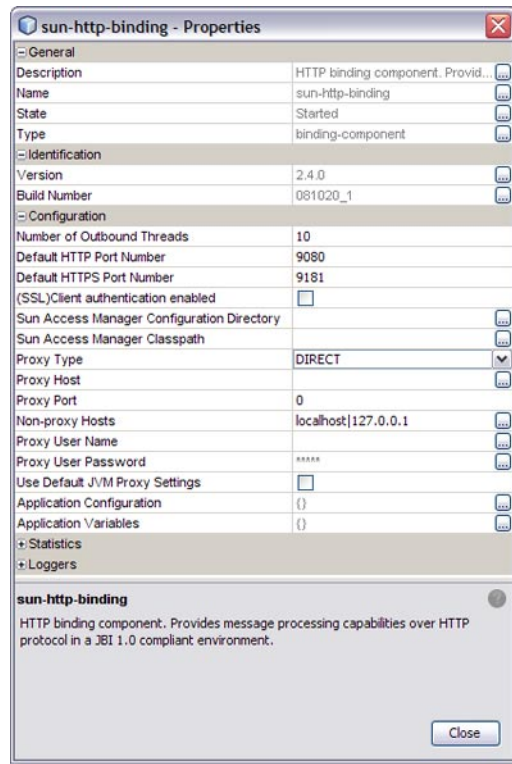
The HTTP Binding Component properties specify clustering and proxy settings, and reference the Binding Component's description, name, type, and state.

TABLE 18 HTTP Binding Component Runtime Properties

Property	Description	Required or Optional	Example
General Properties			
Description	Indicates the purpose of the HTTP Binding Component. This property is displayed for reference purposes.	Automatic	HTTP Soap Binding to send SOAP messages, for example, to and from BPEL service engine.
Name	Indicates the name of the HTTP Binding Component. This property is displayed for reference purposes.	Automatic	com.sun.httpsoapbc-1.0-2

TABLE 18 HTTP Binding Component Runtime Properties   *(Continued)*

Property	Description	Required or Optional	Example
State	Indicates the state of the HTTP Binding Component as "Started" or "Stopped." This property is displayed for reference purposes.	Automatic	Started
Type	Indicates the type of component. This property is displayed for reference purposes.	Automatic	binding-component

Identification Properties

Property	Description	Required or Optional	Example
Version	Indicates the component specification version.	Static	1.0
Build Number	Indicates the component build number.	Static	080311_4

Configuration Properties

Property	Description	Required or Optional	Example
Number of Outbound Threads	Specifies the maximum number of threads to process outbound HTTP/SOAP invocations concurrently. The value can be any integer from 1 to 2147483647.	Required	10
Default HTTP Port Number	Specifies the default HTTP port number for the HTTP Binding Component instance. This property is required for clustering and allows each HTTP Binding Component to be differentiated by its unique default port number. A default port number is calculated and preassigned when the binding component is initially installed in the application server instance. A file containing the persisted configuration is stored for each component. This is used to assign a unique default port number for each HTTP Binding Component instance on a computer.	Required	8180

TABLE 18   HTTP Binding Component Runtime Properties   *(Continued)*

Property	Description	Required or Optional	Example
Default HTTPS Port Number	Specifies the default HTTP Secure port number for the HTTP Binding Component instance. This property is required for clustering and allows each HTTP Binding Component to be differentiated by its unique default port number. A default port number is calculated and preassigned when the binding component is initially installed in the application server instance. A file containing the persisted configuration is stored for each component. This is used to assign a unique default port number for each HTTP Binding Component instance on a computer.	Required	8280
(SSL) Client authentication enabled	Specifies if client authentication 2-way (mutual) SSL on the default HTTPS port is enabled. Restart the binding component to effect changes for this property.	Optional	Select the checkbox to enable
Sun Access Manager Configuration Directory	Specifies the location of the Sun Access Manager configuration directory, which contains the Access Manager properties file. If you are using the OpenSSO Web Service Security Agent (WSS), use this property to specify the directory that contains the AMConfig.properties file. For more information see "Authentication Mechanisms for Consumer Endpoints" on page 83	Optional	C:\JavaCAPS\appserver\ addons\accessmanager
Sun Access Manager Classpath	Specifies the client SDK JAR and WS provider JAR files to be added to the classpath. The files you specify depend on whether you are using Access Manager or OpenSSO Web Service Security Agent (WSS) For more information see "Authentication Mechanisms for Consumer Endpoints" on page 83	Optional	C:\JavaCAPS\appserver\ addons\accessmanager\ amclientsdk.jar, C:\GlassFishESBv21\ glassfish\addons\ accessmanager\ amWebServicesProvider.jar Note: Files are comma separated.

TABLE 18   HTTP Binding Component Runtime Properties   *(Continued)*

Property	Description	Required or Optional	Example
Proxy Type	Specifies the proxy type as SOCKS, HTTP, or DIRECT. Enter one of the following String values: SOCKS The proxy server is a SOCKS (version 4 or version 5) server. HTTP The proxy is an HTTP proxy server. DIRECT The connection does not go through any proxy.	Required	SOCKS
Proxy Host	Specifies the proxy host name or IP address.	Optional	polaris.sun.com
Proxy Port	Specifies the proxy port number.	Required	2080
Non-proxy Hosts	Specifies the list of hosts that you do not want to go through the proxy. Each host is separated with a pipe "|".	Optional	localhost|127.0.0.4
Proxy User Name	Specifies the user name used to the proxy server. For SOCKS-v4, no authentication is required. For SOCKS-v5, the binding component supports no authentication, and Username/Password authentication. For HTTP Proxy, the binding component supports Basic Authentication, Digest Access, and NTLM. Basic Authentication requires a specified username and password. Digest Access and NTLM require a dedicated proxy server for support.	Required in some cases	
Proxy User Password	Specifies the password used in conjunction with the ProxyUserName to access the proxy server. For SOCKS-v4, no authentication is required. For SOCKS-v5, the binding component supports no authentication, and Username/Password authentication. For HTTP Proxy, the binding component supports Basic Authentication, Digest Access, and NTLM. Basic Authentication requires a specified username and password. Digest Access and NTLM require a dedicated proxy server for support.	Required in some cases	

TABLE 18 HTTP Binding Component Runtime Properties *(Continued)*

Property	Description	Required or Optional	Example
Use Default JVM Proxy Settings	Indicates whether the HTTP Binding Component's proxy settings are specified by the existing JVM settings or by the HTTP Binding Component properties. The options indicate the following: true The proxy is controlled by the existing JVM system settings. The settings are outside of this binding component, so all additional proxy settings are ignored. false The proxy is controlled by the binding component proxy settings.	Required	false
Allowed hostname aliases for localhost	Specifies a comma-separated list of alias names that the HTTP BC uses to validate the host name in a service URL when it deploys an application. The HTTP BC does not perform any general validation of the aliases in the list, so they must only be valid as resolvable host names on the host system. Use this property when a given host supports multiple interfaces that have different FQDNs (fully qualified domain names) for the host.	Optional	None

TABLE 18 HTTP Binding Component Runtime Properties *(Continued)*

Property	Description	Required or Optional	Example
Application Configuration	Specifies the values for a Composite Application's endpoint connectivity parameters (normally defined in the WSDL service extensibility elements), and apply these values to a user-named endpoint Config Extension Property. The Application Configuration property editor includes fields for all of the connectivity parameters that apply to that component's binding protocol. When you enter the name of a saved Config Extension and define the connectivity parameters in the Application Configuration editor, these values override the WSDL defined connectivity attributes when your project is deployed. To change these connectivity parameters again, you simply change the values in the Application Configuration editor, then shutdown and start your Service Assembly to apply the new values.	Optional	The user-defined name of the Config Extension you want and define, and the values for the connection parameters.
Application Variables	Specifies a list of name:value pairs for a given stated type. The application variable name can be used as a token for a WSDL extensibility element attribute in a corresponding binding. The Application Variables configuration property offers four variable types: ■ String: Specifies a string value, such as a path or directory. ■ Number: Specifies a number value. ■ Boolean: Specifies a Boolean value. ■ Password: Specifies a password value.	Optional	Enter the name value, such as PASSWORD, and enter the variable Value, such as SECRET. For Boolean values, the Value field provides a checkbox (checked = true). For Password values, the Value entered is masked as asterisks.

Statistics Properties

Includes 19 different component activities including exchanges, errors, requests, replies, and so forth.	Lists component statistics that are collected for actions such as endpoints activated, average response time, completed exchanges, and so forth. Running statistics are automatically collected and displayed.	Automatic	240

Loggers Properties

TABLE 18   HTTP Binding Component Runtime Properties   *(Continued)*

Property	Description	Required or Optional	Example
Includes over 30 different component activities that can be recorded by the server.log.	Specifies the level of logging for each event. There are eight levels of logging, FINEST (most detailed), FINER, FINE, CONFIG, INFO, WARNING, SEVERE (failure messages only), and OFF.	Optional	WARNING

HTTP Binding Component Client Endpoint Properties

The HTTP Binding Component's Client Endpoint Configuration Properties allow you to define the outbound properties for the specific WSDL port. The properties editor is accessed from the Composite Application Service Assembly (CASA) Editor.

▼ Accessing the HTTP Binding Component Client Endpoint Properties

1   Open your project in the CASA Editor and click the Build icon in the CASA Editor toolbar.

All of the project's WSDL ports (endpoints) are visible.

2   Right-click the outbound client endpoint (green arrow) that you want to configure, and choose Properties from the pop-up menu.
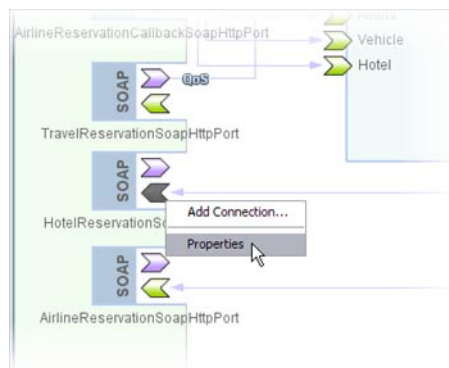
The Properties Editor Appears.
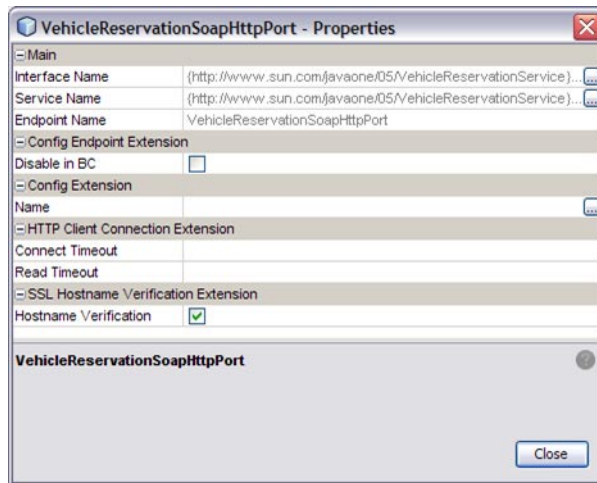
HTTP BC Client Endpoint Configuration Properties

The properties editor contains the following configuration properties:

Main Properties

Interface Name   Provides the existing name of the interface.

Service Name   Provides the existing name of the service.

Endpoint Name   Provides the existing endpoint name.

JAX-WS Extension Properties

Handlers   Specifies a list of JAX-WS handlers that act as interceptors to inject application logic before and/or after service invocations. These interceptors can alter the protocol or payload messages to extend message processing capabilities. For more information about configuring JAX-WS handlers, see Using JAX-WS Handlers With the HTTP Binding Component, and for more information about JAX-WS handlers in general, see A Little Bit About Handlers in JAX-WS.

Config Endpoint Extension Properties

Disable in BC   Specifies whether the endpoint is enabled or disabled. To disable and endpoint select the checkbox. The endpoint will not be activated the next time the composite application is deployed.

Config Extension Properties

Name   Specifies the name of the Application Configuration used by this endpoint.

Application Configurations which define endpoint configuration extension properties, and be defined and named in the Application Configuration property, an HTTP Binding Component Runtime Property. Various application configurations can be configured and named. These are then available for use by the endpoint by specifying the application configuration name in the endpoint's Config Extension Property. For more information, see "Using Application Configuration to Configure Connectivity Parameters" on page 119

HTTP Client Connection Extension Properties

Connect Timeout   Specifies the connect timeout value in milliseconds, used to open an HTTP connection to an external service. A value of 0 (zero) indicates an infinite timeout.

Read Timeout   Specifies the read timeout in milliseconds, indicating the configured length of time to read from the input stream when an HTTP connection to an external service is established. A value of 0 (zero) indicates an infinite timeout.

SSL Hostname Verification Extension Properties

Hostname Verification   Specifies whether Hostname Verification is enabled. Select the checkbox to enable Hostname Verification and when a custom host name verifier is used. Clear the checkbox to turn Hostname Verification off.

Hostname Verification ensures that the host name in the digital certificate matches the host name in the URL to which client connects. In some cases it is useful to turn verification off, such as for test purposes or when security is not an issue.

Note – To enable any changes to your project configuration, redeploy your project.

Using Normalized Message Properties to Propagate Binding Context Information

Normalized Message properties are commonly used to specify metadata that is associated with message content. `javax.jbi.security.subject` and `javax.jbi.message.protocol.type` are two examples of standard normalized Message properties defined in the JBI Specification.

Normalized Message properties are used to provide additional capabilities in Open ESB, such as:

- Getting and Setting transport context properties. For example, HTTP headers in the incoming HTTP request, or file names read by the File Binding Component

- Getting and Setting protocol specific headers or context properties (SOAP headers)

- Getting and Setting additional message metadata. For example. a unique message identifier, or an endpoint name associated with a message

- Dynamic configurations. For example, to dynamically overwrite the statically configured destination file name at runtime

Some of the use cases mentioned above require protocol/binding specific properties, typically used by a particular binding component. Other properties are considered common or general purpose properties that all participating JBI components make use of, for example, the message ID property, which can be utilized to uniquely identify or track a given message in the integration.

Using Normalized Message Properties in a BPEL Process

The Normalized Message properties are accessed from the BPEL Designer Mapper view. When you expand a variable's Properties folder it exposes the variable's predefined NM properties, as well as the regular BPEL specific WSDL properties used in correlation sets, assigns, and to build expressions . If the specific NM property you need is not currently listed, additional NM properties can be added.

Using Predefined Normalized Message Properties in a BPEL Process

Predefined Normalized Message properties are ready for use, from a variable's Properties file.

▼ To use predefined normalized message properties in a BPEL process

1   From the Design View diagram, select the activity with the process you want to edit.

2   Click `Mapper` to switch to the Mapper view of the BPEL process.

3   From the Output pane, expand the Variable you want to edit and its Properties file.

The Properties file contains the predefined Normalized Message (NM) properties.
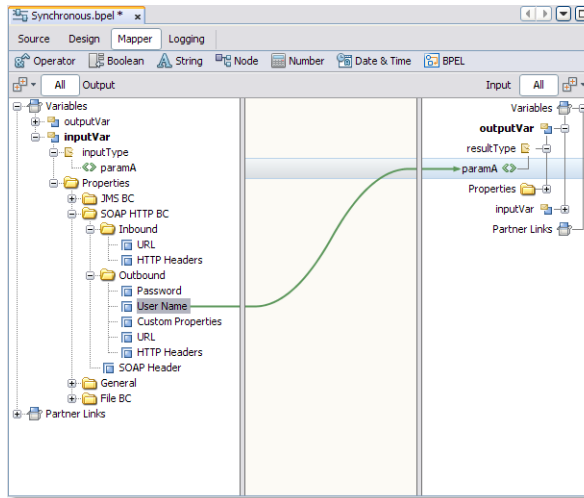
4   To use a predefined NM Property, select the property and use it to build an expression or an assign.

Adding Additional Normalized Message Properties to a BPEL Process

If the specific NM Property you want is not listed, you can add additional NM properties.

There are two options available when adding NM Properties:

- **Add NM Property Shortcut:** This option typically supports *simple type* properties, in that it does not grant access to some data within the NM Property.

- **Add NM Property:** This option provides access to data within the NM property used to build expressions.

▼ **To add a Normalized Message Property Shortcut to a BPEL process**

1 **From the Output or Input panes of the BPEL Mapper, expand the node for the variable to which you want to add an NM property. Right-click that variables `Properties` directory node and select `Add NM Property Shortcut` from the pop-up menu.**

The Add NM Property Shortcut dialog box appears.

2 **Enter the information for the new NM property into the Add NM Property Shortcut dialog box, as follows:**

a. **Property Name: The NM property name (see each binding component's documentation for available NM properties).**

b. **Display Name: The display name for the NM property. This is a user-defined name that appears in the Mapper tree. The display name is optional.**

3 **Click OK.**

The new NM property is added to the Mapper tree under the variables Properties directory. The property can now be used in assigns and to build expressions.

▼ **To edit an NM Property Shortcut**

1 **To edit an existing NM property shortcut, right-click the NM property shortcut in the BPEL Mapper tree and choose `Edit NM Property Shortcut` in the pop-up menu.**

The Add NM Property Shortcut dialog box appears.

2 **Edit the NM Property Name or Display Name, and click `OK`.**

▼ **To delete an NM Property Shortcut**

1 **To delete an NM property shortcut, right-click the property in the Mapper tree.**

2 **Choose `Delete NM Property Shortcut` in the pop-up menu.**

The NM Property Shortcut is deleted.

▼ To add a Normalized Message Property to a BPEL process

1   **From the Output or Input panes of the BPEL Mapper, expand the node for the variable to which you want to add an NM property. Right-click that variables `Properties` directory node and select `Add NM Property` from the pop-up menu.**

The Add NM Property dialog box appears.

2   **Enter the information for the new NM property in the Add NM Property dialog box, as follows:**

a.   **Property Name: User-defined property name. This name is displayed in mapper tree and stored in WSDL file.**

b.   **Type or Element:Displays the property type or element associated with the selected node in the `Map Property To` tree.**

c.   **Associate property with message: Specifies with which message type the property is associated.**

- A check mark indicates that the new NM property is associated with all variables of the specified message type. For example, in the image below, the new NM property will be associated with the *requestMessage* type.

- Unchecked indicates that the new NM property is associated with all variables of any message type.

d.   **Map Property To: The `Map Property To` tree displays all of the predefined NM properties. This is used to build a query or choose a property type.**

When you select a node within the property tree the `Type or Element` and `Query` fields are populated automatically. Valid endpoint nodes are displayed in bold.

e.   **New NM Property: Select the `New NM Property` checkbox to add a specific NM property, and enter the name of the property in the `New NM Property` field. The new NM property is added to the `Map Property To` tree.**

f.   **Sync with tree:When this checkbox is selected, the Query field is automatically synchronized with the selected node in the `Map Property To` tree.**

g. **Query: Displays the query type associated with the selected node in the `Map Property To` tree.**

3   **Click OK. The new NM property name is added to the tree in the BPEL Mapper, and the NM property is stored in nmPropertiesDefinitions.wsdl as a pair of elements: `<vprop:property>` and `<vprop:propertyAlias>`**
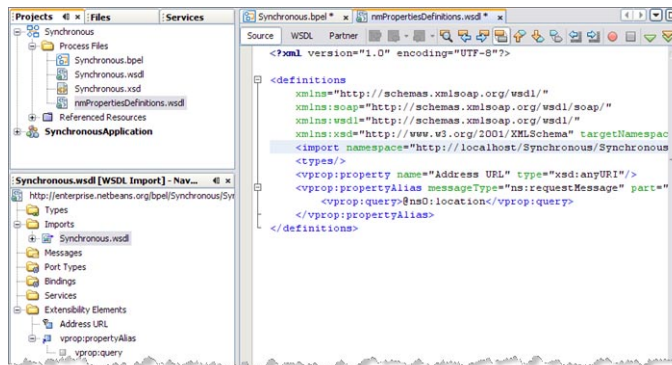
The new NM property can now be used in assigns and to build expressions.

▼ To delete an NM Property

1   To delete a new NM property, right-click the property in the Mapper tree.

2   Choose `Delete NM Property` in the pop-up menu.

The property is deleted.

BPEL Code Generation Using NM Properties

Data copied from an NM property or an NM property shortcut generates code that is similar to the following:

```
<from variable="inputVar" sxnmp:nmProperty="org.glassfish.openesb.file
.outbound.dcom.username"/>
```

Data copied from WSDL properties based on NM property generates code that is similar to the following:

```
<from variable="inputVar" property="ns3:DemoNMProperty"/>
```

When properties and NM properties are used to build an expression, code similar to the following code is generated:

```
<from>concat(bpws:getVariableProperty('inputVar', 'ns3:DemoNMProperty'),
sxnmp:getVariableNMProperty('inputVar','org.glassfish.openesb.file.outbound.dcom.
username'))</from>
```

An NM property used in a condition generates code that is similar to the following:

```
<condition>sxnmp:getVariableNMProperty('inputVar', 'my.nmProperty.boolean')</condition>
```

Normalized Message Properties

Normalized Message properties are either *General*, available to all participating JBI components, or protocol/binding specific, used by a particular binding component.

TABLE 19 General Normalized Message Properties

Property Name	Type	Description and Use
org.glassfish.openesb. messaging.groupid	java.lang.String	Uniquely identifies a message with the group to which a message belongs. For example, it applies the RM sequence group number for SOAP messages, or a time stamped file name (where the file record message comes from). This property is optional.
org.glassfish.openesb. messaging.messageid	java.lang.String	Uniquely identifies a message. For batch processing this might be a record number (for example, a particular record in a file), or a GUID. This property is mandatory.
org.glassfish.openesb. messaging.lastrecord	java.lang.String	The value is a string representation of boolean ("true" or "false"). This property can be used to signal the last record in a group, e.g. the last record in a RM sequence for SOAP messages, or the last record in a file when multiple record processing is turned on for File BC. This property is mandatory.
org.glassfish.openesb. exchange.endpointname	java.lang.String	The value a string representation of the endpoint name set on the exchange. This represents the endpoint name of the "owner" of the message, and could be made available by JBI runtime.

SOAP HTTP Binding Component Specific Normalized Message Properties

The following properties are specific to the HTTP (SOAP) Binding Component.

TABLE 20 SOAP HTTP Binding Component NM Properties

Property Name	Type	Description and Use
org.glassfish.openesb. headers.soap	java.util.Map	The map contains a list of SOAP header elements. The key is the QName of the SOAP header. The value is a DocumentFragment object. The DocumentFragment has one node in it, the header element itself.

TABLE 20   SOAP HTTP Binding Component NM Properties   *(Continued)*

Property Name	Type	Description and Use
org.glassfish.openesb.inbound.http.headers	java.util.Map	The map contains a list of HTTP headers. The key is the HTTP header name.
		The value is the string representation of the HTTP header value.
		This property provides all of the HTTP headers that the HTTP BC receives in the incoming message. The map also includes two additional properties that the HTTP Binding Component populates based on the transport context: `ClientHostName` and `ClientPortNumber`, which provide the information about the client's host IP address and port number
org.glassfish.openesb.outbound.http.headers	java.util.Map	The map contains a list of HTTP headers. The key being the HTTP header name.
		The value is the string representation of the HTTP header value.
		This property is used to allow any custom HTTP headers to be propagated to the outgoing service invocations.
org.glassfish.openesb.outbound.custom.properties	java.util.Map	The map contains a list of custom properties*.
		The map key is a string.
		The map value can be any Object.
org.glassfish.openesb.inbound.address.url	java.lang.String	On the receiving (server) side, this property is populated by the HTTP Binding Component with the server address URL (for example, address URL on soap:address)
org.glassfish.openesb.outbound.address.url	java.lang.String	On the sending (client) side, this property is used to dynamically overwrite the default address defined in the SOAP or HTTP binding WSDL.
		The HTTP Binding Component does a basic URL validation on the address set on the property before using it to invoke an external service. If it is an invalid URL, the HTTP BC proceeds with the service invocation using the statically configured address URL.
org.glassfish.openesb.outbound.basicauth.username	java.lang.String	This is a sender (client) side property only.
		When set, the user name will be set on the HTTP basic authentication header.

TABLE 20   SOAP HTTP Binding Component NM Properties   *(Continued)*

Property Name	Type	Description and Use
org.glassfish.openesb. outbound.basicauth.password	java.lang.String	This is a sender (client) side property only.
		When set, the user name will be set on the HTTP basic authentication header.

Note – The `org.glassfish.openesb.custom.properties` property is designed to allow custom data to be set on the HTTP/SOAP binding message context. The custom properties on the binding message context can then be made available in the security CallbackHandlers. For example, you can allow custom SAML assertion headers to be set in the SAML CallbackHandler based on the user credentials (application data) set on the binding message context.

Quality of Service (QOS) Features

Quality of Service features are configured from the CASA Editor, and include properties used to configure Retry (Redelivery) and Throttling.

This section contains the following topics:

Configuring the Quality of Service Properties

The QOS attributes are configured from the Config QoS Properties Editor, accessed from the Composite Application Service Assembly (CASA) Editor. For an example of how to access the Config QOS Properties Editor, see

Attribute	Description	Value/Example
Consumer Settings		
Service Name	Specifies the consumer service name. Click the ellipses button to open the QName Editor. Select a pre-existing Namespace URL or enter a new Namespace URL and prefix.	{http://j2ee.netbeans.org /wsdl/SoapBasicAuth} SoapBasicAuthServices
Endpoint Name	Specifies the consumer endpoint name. Click the ellipses button to open an edit window.	SoapBasicAuthPort WssToken
Provider Settings		

Attribute	Description	Value/Example
Service Name	Specifies the provider service name. Click the ellipses button to open the QName Editor. Select a pre-existing Namespace URL or enter a new Namespace URL and prefix.	{http://enterprise. netbeans.org/bpel/ BasicAuthBP/ SoapBasicAuthAM} AMPartnerLink
Endpoint Name	Specifies the Provider endpoint name. Click the ellipses button to open an edit window.	SoapBasicAuthAM PortTypeRole_myRole

RedeliveryExtension Settings

maxAttempts	Specifies the number of retries to attempt before using the on-failure option.	20
waitTime	Specifies time (in milliseconds) to wait between redelivery attempts.	300

Attribute	Description	Value/Example
on-failure	Specifies the type of action to be taken when message exchange (ME) redelivery attempts have been exhausted. The on-failure options are ■ delete: When the final defined delivery attempt has failed, the QoS utility abandons the message exchanges (ME) and returns a Done status to the JBI component, which proceeds to its next process instance. This option is only supported for *In-Only* message exchanges. ■ error: When the final defined delivery attempt has failed, the QoS utility returns an Error status to the JBI component, and the JBI component throws an Exception. This is the default option, and is supported for both *In-Only* and *In-Out* message exchanges. ■ redirect: Similar to the *delete* option, except that the QoS utility re-routes the ME to the configured redirect endpoint when the maxAttempts count has been exhausted. If the QoS utility is successful in routing the message to the redirect endpoint, a Done status is returned to the JBI component; otherwise, an Error status is returned. This option is supported for *In-Only* message exchanges only. ■ suspend: The QoS utility returns an Error status to the JBI component if it is not able to deliver the ME to the actual provisioning endpoint. After the redelivery attempts have been exhausted, the JBI Component suspends the process instance. This option is only supported if monitoring is enabled in the JBI Component, since the user must use the monitoring tool to resume a suspended instance. This option is supported for both *In-Only* and *In-Out* message exchanges.	delete

ThrottlingExtension Settings

| maximum-Concurrency Limit | Specifies the maximum number of concurrent messages that can be processed on a specific connection. This number is used to set up the maximum number of concurrent messages that the internal endpoint sends to the provider endpoint. | 10 |

Message Throttling: Configuring and Using

Throttling allows you to set the maximum number of concurrent messages that are processed by a particular endpoint. Increased message load and large message payloads can cause memory usage spikes that can decrease performance. Throttling limits resource consumption so that consistent performance is maintained.

The HTTP Binding Component, using functionality provided by the Grizzly HTTP Web Server, manages the flow of messages by evaluating endpoints to determine when it is necessary to suspend requests and when to resume processing as usual.

For more information in regard to HTTP BC and Throttling, see HTTP BC Throttling (`http://wiki.open-esb.java.net/Wiki.jsp?page=HTTPBCThrottling`).

Configuring the HTTP Binding Component Endpoint for Throttling

For the HTTP Binding Component, throttling is a QOS feature configured from the CASA Editor.

▼ To configure Throttling for an HTTP/SOAP WSDL port

1   From the NetBeans IDE Projects window, right-click the Service Assembly node under your composite application, and select Edit from the popup menu.

The CASA Editor opens containing your composite application.
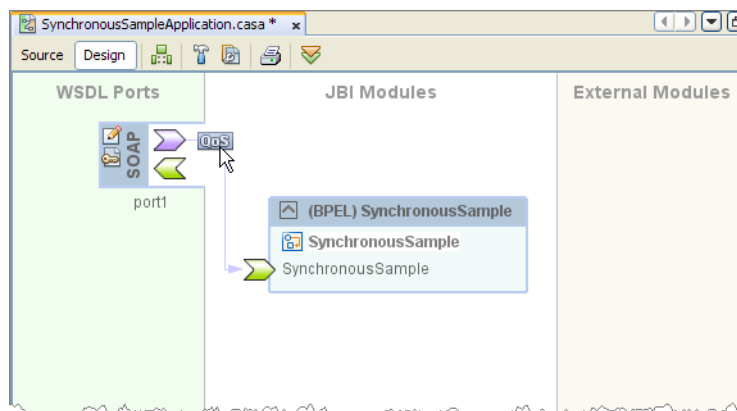
2   In the CASA Editor, click the QOS icon located on the link between your JBI Module and the WSDL port you want to configure.

The QOS Properties Editor appears.

3   In the QOS Properties Editor, click the property field for maximumConcurrencyLimit under ThrottlingExtension, and enter an integer for the maximum number of concurrent messages allowed for this endpoint.
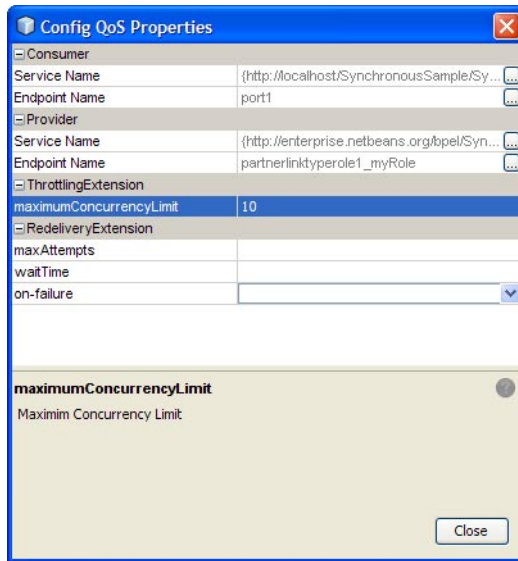
4   Click Close.

The appropriate throttling configuration for the connection is generated in the project's jbi.xml file, when the service assembly is built.

Redelivery: Configuring and Using

Redelivery is a Quality of Service mechanism that handles message delivery when first-time delivery fails. Redelivery allows you to define the number of attempts that the system makes to deliver a message, the time between attempts, and the final result for an undeliverable message or nonresponsive endpoint.

Redelivery is configured for a specific connection from the Composite Application Service Assembly (CASA) Editor, by clicking the QoS icon for that connection. This opens the Config QoS Properties for that connection. From the RedeliveryExtension section of the editor, configure the Redelivery properties.

The Redelivery configuration parameters are:

- **maxAttempts:** Specifies the number of times that the project attempts to re-deliver a message. An error status is returned to the JBI component for each failed attempt.
- **waitTime:** Specifies the time, in milliseconds, that the project waits between redelivery attempts.

- **on-failure:** Specifies the actions taken and the message destination when the specified redelivery attempts have been exhausted. This parameter has four options: delete, redirect, suspend, and error.

    The on-failure parameter has four options: delete, redirect, suspend, and error.

    - **delete:** The delete option specifies that when the final attempt to re-deliver the message has failed, the QoS utility deletes the message and returns a Done status to the JBI component, at which time the component proceeds to its next process. The delete option only supports *In-Only* message exchanges.

    - **redirect:** The redirect option specifies that after the final attempt to re-deliver the message has failed, the QoS utility redirects the message to a user-defined endpoint, such as a "dead-message" folder. Upon successful delivery to the redirect endpoint, the QoS utility returns a Done status to the JBI component, at which time the component proceeds to its next process. The redirect option only supports *In-Only* message exchanges.

    - **suspend:** The suspend option specifies that when the final attempt to re-deliver the message has failed, the JBI component suspends the process instance . This option is only supported if monitoring is enabled in the JBI Component, since the user must use the monitoring tool to resume a suspended instance. This option is supported for both *In-Only* and *In-Out* message exchanges.

    - **error:** The error option specifies that when the final attempt to re-deliver the message is exhausted, the JBI component throws an exception. This option is only supported if monitoring is enabled in the JBI Component, since the user must use the monitoring tool to resume a suspended instance. This option is supported for both *In-Only* and *In-Out* message exchanges.

Note: The on-failure options: delete and redirect, cannot be applied to In-Out message exchanges because In-Out message exchanges require a specific response from the process instance to proceed further, and as such, the return value for these options does not suffice.

For more information regarding Redelivery, see Redelivery (http://wiki.open-esb.java.net/Wiki.jsp?page=Redelivery).

Using the Tango Web Service Features with the HTTP Binding Component

Tango is a key component of the Metro Project. Tango (also known as WSIT) is an implementation of the key enterprise web services, commonly known as WS-*services*, such as WS-Security, WS-Reliable Messaging, WS-Transactions, and so forth. Tango leverages the existing JAX-WS and EJB programming models and allows you to define Security, Reliability, and Transactional capability for application endpoints by bundling an additional configuration file with your application.

The HTTP Binding Component exposes several Tango features that can be applied to your composite application projects.

- **Messaging Optimization:** Modifies web service messages for optimal processing and bandwidth efficiency. Message Optimization is recommended if your client endpoint will be processing web documents larger than 1KB.

  MTOM `Message Transmission Optimization Mechanism` optimizes web service messages so that they are efficiently transmitted over the internet by encoding the XML code for better processing time and minimal bandwidth requirements.

- **WS-Addressing:** Enables re-routing of requests and responses. WS-Addressing supports normalized web service addresses, enabling multiple transports to be used (other than HTTP).

- **Reliable Messaging:** Ensures that application messages are delivered once only, and optionally in the correct order, to web service endpoints.

  - **WS Reliable Messaging:** Defines a standard for identifying, tracking and managing message delivery between two parties reliably, ensuring recovery from failures that may be caused by messages that are lost or received in the wrong order. See for more information see "Configuring Reliable Message Delivery" on page 64.

  - **WS Atomic Transactions:** Supports a two phase commit protocol to ensure that either all of the operations invoked within a transaction succeed, or they are all rolled back.

- **Security:** Works in addition to existing transport-level security to provide interoperable message content integrity and confidentiality.

  - **WS Security:** Defines a standard set of SOAP extensions used when building secure web services to implement message content integrity and confidentiality. Supports various security token formats, trust domains, signature formats, and encryption technologies.

  - **WS Secure Conversation:** Allows a consumer and provider to establish a shared security context for multiple-message-exchanges. The Secure Conversation authentication specification defines a standardized way to authenticate a series of messages, thereby addressing the short comings of web services security. With the WS Security Conversation model, the security context is defined as a new web services security token type, obtained using a binding of Web Services Trust.

  - **WS Trust:** Defines extensions to Web Services Security that provide methods for issuing, renewing, and validating security tokens. Supports the management of Trust relationships.

Configuring Reliable Message Delivery

The following example demonstrates how to configure Reliable Message Delivery for a project, and uses the Synchronous BPEL Process sample included with NetBeans.

▼ Installing the Synchronous BPEL Process sample

1 **In the NetBeans IDE, select the Projects tab to display the Projects window.**

2 **From the File menu, select New Project.**

The New Project dialog box appears.

3 **In the Categories list of the New Project dialog box, select Samples → SOA → Synchronous BPEL Process, and click Next.**

4 **Accept the default project name and location, and click Finish.**

Your new project appears in the Projects window.

▼ Configuring Web Services for a Project from the CASA Editor

1 **In the NetBeans IDE, expand the SynchronousSampleApplication node in the Projects window. Right-click Service Assembly and select Edit.**

The CASA Editor opens in the NetBeans IDE, displaying the Design View of the Synchronous Sample application. The CASA Editor creates and modifies a .casa file, which contains the configuration information for the composite application. For this sample, the CASA Editor created the SynchronousSampleApplication.casa file.

2 **In the CASA Editor, click the Build Project icon to build the composite application.**

When the build successfully completes, the Design View displays a WSDL port endpoint, a JBI module, and the connection between the endpoint and the JBI module.

3 **Right-click the SOAP Binding and select `Clone WSDL Port to edit...` from the popup menu.**

The Clone WSDL port to CompApp dialog appears. Click OK to continue. The SOAP Binding icon now contains icons to access the Properties Editor and Server/Client Configuration.

4 **Click the Server/Client Configuration icon on the SOAP Binding, and select Server Configuration.**

The WS-Policy Attachment dialog box for the SOAP Binding port appears.

5 **From the WS-Policy Attachment dialog box, click the check box for `Reliable Message Delivery` to enable reliable messaging, and click the check box for `Deliver Messages In Exact Order` to enable message order. For even more options click the `Advanced` button.**

6 **Click OK.**

Configuring the Tango Web Services Attributes exposed by the HTTP Binding Component

For composite applications, the Web Services attributes are configured for a WSDL port using in the WS Policy Attachment Editor associated with the specific endpoint. The WS Policy Attachment Editor for a WSDL port is accessed in the CASA Editor.

This section contains the following topics:

Accessing the Tango (WSIT) Web Service Attribute Configuration

The Web Services attributes are configured for each WSDL port using in the WS Policy Attachment Editor associated with the specific endpoint. The WS Policy Attachments Editor is accessed from the CASA Editor by right-clicking a WSDL port and selecting Edit Web Service Attributes → Server Configuration, or Client Configuration.

The following directions assume that you have already created a Composite Application. This option is available after a WSDL port has been cloned or created in CASA.

▼ Accessing the WS-Policy Attachment Editor for a Specific Endpoint

1   From the NetBeans IDE's Projects window, expand your composite application. Right-click the Service Assembly node and select `Edit` from the popup menu.

The Composite Application Service Assembly (CASA) Editor appears, containing the composite application.

2   In the CASA Editor, select the Build Project icon to build the composite application.

When the build successfully completes, the Design view displays the WSDL port endpoints, JBI modules, and the connections between each.

3 **If you did not build or clone the WSDL port, the WS Policy Attachment for that port is not available for configuration. To clone a port, right-click the WSDL port, and select `Clone WSDL Port to edit` in the popup menu.**
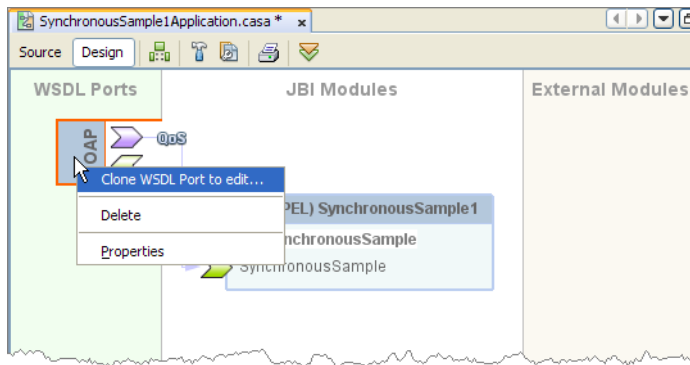
After the port has been cloned, the Port Properties and Web Service Attributes icons are added to the port.

4 **Click the ports Web Service Attributes icon (the bottom icon on the port) and select Server Configuration or Client Configuration to open the appropriate WS Policy Attachment Configuration Editor.**

Server Configuration—Web Service Attributes

The Server Configuration web service attributes exposed by the HTTP Binding Component are configured from the WS Policy Attachment Configuration Editor.
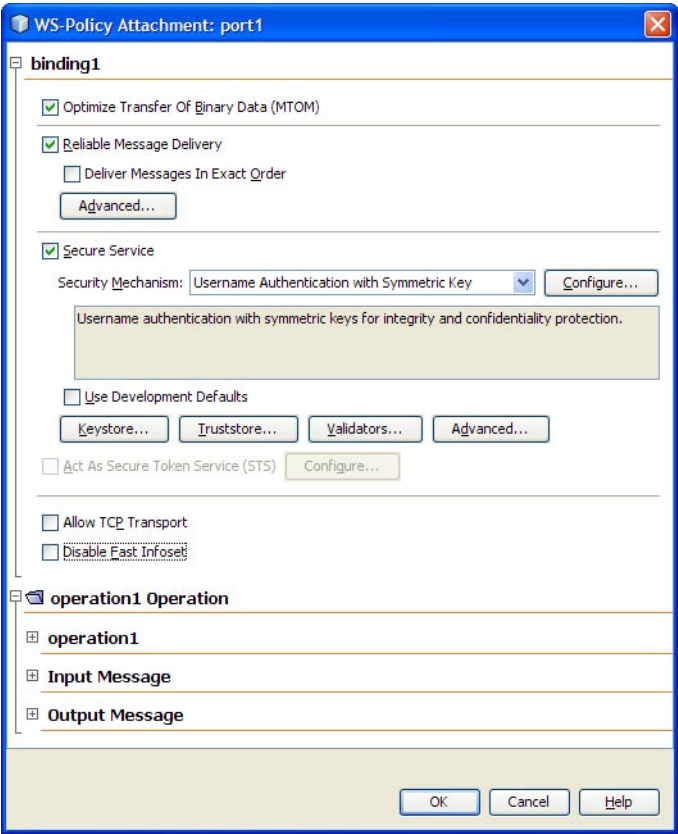
The Server Configuration web service attributes include the following:

Attribute	Description	Value
Binding Settings		
Optimize Transfer of Binary Data (MTOM)	Specifies whether the web service is configured to optimize messages that it transmits and decodes optimized messages that it receives.	Select the checkbox to enable.

Attribute	Description	Value
Reliable Message Delivery	Specifies whether the service sends an acknowledgement to the clients for each message that is delivered, thus enabling clients to recognize message delivery failures and retransmit the message.	Select the checkbox to enable.
	Deliver Messages in Exact Order: Specifies whether the Reliable Messaging protocol ensures that the application messages for a given message sequence are delivered to the endpoint application in the order indicated by the message numbers. This option increases the time to process application message sequences and may result in slower of web service performance. Only enable this option when ordered delivery is required by the web service	Select the checkbox to enable.
	Flow Control: Specifies whether the Flow Control feature is enabled. It may be necessary to withhold messages from the application if ordered delivery is required and some preceding messages have not yet arrived. If the number of stored messages reaches the threshold specified in the Max Buffer Size setting, incoming messages belonging to the sequence are ignored.	Select the checkbox to enable.
	Maximum Flow Control Buffer Size: Specifies the number of messages that are buffered for a message sequence. 32 is the default setting.	32 is the default value.
	Inactivity Timeout (ms): Specifies in milliseconds, the time interval at which source or destination can terminate a message sequence due to inactivity. A web service endpoint will always terminate a sequence whose timeout has expired. To keep a sequence active, an inactive client sends a stand-alone message with an AckRequested header to act as a heartbeat when the end of the inactivity timeout interval approaches.	600,000 (milliseconds) is the default value.
Secure Service	Specifies whether web service security options are enabled for all of the operations of a web service.	Select the checkbox to enable.

Attribute	Description	Value
Security Mechanism	Specifies the security mechanism used by the web service operation. The available security mechanisms are: ■ "Username Authentication with Symmetric Key " on page 90 ■ "Mutual Certificates Security" on page 93 ■ "Transport Security (SSL)" on page 95 ■ "Message Authentication over SSL" on page 96 ■ "SAML Authorization over SSL" on page 98 ■ "Endorsing Certificate" on page 100 ■ "SAML Sender Vouches with Certificates" on page 101 ■ "SAML Holder of Key" on page 104 ■ "STS Issued Token" on page 106 ■ "STS Issued Token with Service Certificate" on page 109 See the "Configuring Security Mechanisms" on page 90 section for more information.	Select the security mechanism to be used by your application. Information about your selected mechanism and its additional requirements is displayed in the message box below your selection.
	Configure: The configuration button opens a configuration editor for the selected security mechanism.	See the Security Mechanisms section for more information about configuration properties.
	Use Development Defaults: Specifies whether to import certificates into the GlassFish keystore and truststore to be used immediately for development. The default certificates are imported in the correct format and a default user is created in the *file* realm, with username "wsitUser". For your project you will most likely choose to use your own certificates and user settings, but in a development environment you may find the defaults useful.	Check box Selected indicates that you are using the default certificates.

Attribute	Description	Value
Keystore	Click the Keystore button to open the Keystore Configuration Editor. The editor specifies the following information: ■ Location: Use the Browse button to specify the location and name of the keystore. ■ Keystore Password: Specifies the password for the keystore file. If you are running under GlassFish, GlassFish's password is already entered. If you have changed the keystore's password from the default, you must specify the correct value in this field. ■ Alias: Specifies the alias of the certificate in the specified keystore to be used for authentication. The Keystore alias for non-STS applications is `xws-security-client` for client-side, and `xws-security-server` for server-side configuration. The Keystore alias for STS applications is `xws-security-client` for both client-side and STS Configuration. ■ Key Password: Specifies the password of the key within the keystore. By default, the key password uses the store password. Only specify a password in this field when the key password is different. ■ Alias Selector Class: Specifies the selector class for aliases.	Configure the Keystore from the Keystore Configuration Editor.
Truststore	Click the Truststore button to open the Truststore Configuration Editor. The editor specifies the following information: ■ Location: Use the Browse button to specify the location and file name of the truststore that stores the public key certificates of the CA and the client's public key certificate. ■ Truststore Password: Specifies the password for the Truststore. If you are running under GlassFish, GlassFish's password is *changeit*. If you have changed the truststore's password from the default, you must specify the correct value in this field. ■ Load Aliases: Clicking the Load Aliases button populates the Alias field with the aliases contained in the truststore file. The Location and Truststore Password fields must be specified correctly for this option to work. ■ Certificate Selector: Specifies a String which specifies the identities of zero or more certificates. The specifiers can conform to X.509 naming conventions. A certificate selector can also use various shortcuts to match either subject alternative names, the filename, or even the issuer.	Configure the Truststore from the Truststore Configuration Editor.

Attribute	Description	Value
Validators	Click the Validators button to open the Validator Configuration Editor. The editor specifies the following information: ■ Username Validator: Specifies the validator class used to validate username and password on the server side. This option is only used by a web service. Note: When using the default Username Validator, make sure that the username and password of the client are registered with GlassFish (using Admin Console) if using GlassFish, or is included in the tomcat-users.xml file if using Tomcat. ■ Timestamp Validator: Specifies the validator class to be used to check the token timestamp to determine whether the token has expired or is still valid. ■ Certificate Validator: Specifies the validator class to be used to validate the certificate supplied by the client or the web service. ■ SAML Validator: Specifies the validator class to be used to validate SAML token supplied by the client or the web service.	Configure the Validators from the Validator Configuration Editor.
Advanced (Advanced Security Options)	Click the Advanced button to open the Advanced Security Options Editor. The editor specifies the following information: ■ Maximum Clock Skew (ms): Specifies the maximum difference allowed between the system clocks of the sender and recipient in milliseconds. ■ Timestamp Freshness Limit (ms): Specifies the Timestamp Freshness Limit in milliseconds. Timestamps received with a creation time older than the Timestamp Freshness Limit period are rejected by the receiver. ■ Use Default Certificate Revocation Mechanism: If this option is selected, the default revocation checking mechanism of the underlying PKIX service provider is used.	Configure the Advanced Security Options from the Advanced Security Options Editor.

Attribute	Description	Value

Act as a Secure Token Service (STS)	Select the Act as a Secure Token Service checkbox and click the Configure button to open the STS Configuration Editor. The editor specifies the following information:
	■ Issuer: Specifies an identifier for the issuer for the issued token. This value can be any String that uniquely identifies the STS.
	■ Contract Implementation Class: Specifies the actual implementation class for the WSTrustContract interface that handles token issuance, validation, and so forth. Default value is `com.sun.xml.ws.trust.impl.IssueSamlTokenContractImpl` for issuing SAML assertions, or click Browse to select another contract implementation class.
	■ Lifetime Issued Tokens (ms): Specifies the life span of the token issued by the STS. The default value is 36000 ms.
	■ Encrypt Issued Key: Specifies whether the issued key is encrypted using the service certificate. Selected indicates yes.
	■ Encrypt Issued Token: Specifies whether the issued token is encrypted using the service certificate. Selected indicates yes.
	■ Service Providers: Specifies the Service Providers that have a trust relationship with the STS. Click Add to specify a a new provider. Providers can be listed using the following protocols:
	  ■ Provider Endpoint URI: Specifies the endpoint URI of the service provider.
	  ■ Certificate Alias: Specifies the alias of the certificate of the service provider in the keystore.
	  ■ Token Type: Specifies the type of token the service provider requires.
	  ■ Key Type: Specifies the type of key the service provider requires: public key or symmetric key.
		Configure the STS Configuration Options from the STS Configuration Editor.

Allow TCP Transport	Specifies whether the service supports TCP and HTTP message transport. TCP enhances performance for smaller messages by eliminating the overhead of sending messages over HTTP protocol.	Select the checkbox to enable.
Disable Fast Infoset	Specifies whether Fast Infoset is enables for faster parsing, faster serializing, and creating smaller document sizes, compared with equivalent XML Documents. When this option is selected, the Web service will not process incoming messages or produce outgoing messages encoded using Fast Infoset.	Select the checkbox to enable.
Operation Settings		

Attribute	Description	Value
Transactions	Specifies the level at which transactions are secured.	
Input Message Settings		
Authentication Token	Specifies which supporting token will be used to sign and/or encrypt the specified message parts. Options include Username, X509, SAML, Issued, or *None*.	Username
	Signed: Specifies that the authentication token must be a signed, supporting token. A signed supporting token is also signed by the primary message signature.	Select the checkbox to enable.
	Endorsed: Specifies that the authentication token must be endorsed. With an endorsing supporting token, the key represented by the token is used to endorse/sign the primary message signature.	Select the checkbox to enable.
	Message Parts: Specifies the message parts that must be signed and/or encrypted. Click the Message Parts button to open the Message Parts dialog box. From the Message Parts dialog box you can specify the following options for message parts or elements: ■ Sign: Specifies that the message part requires a digital signature for integrity protection. ■ Encrypt: Specifies that the message part requires encryption for confidentiality. ■ Require: Specifies that the message part is required for a message. The Message Parts dialog box also includes the following buttons: ■ Add Body: Adds a row for the message body (this is only necessary if a row has been removed). ■ Add Header: adds a row for either a specific SOAP header part or for all SOAP header parts (this is only necessary if the SOAP header row in question has been deleted). ■ Add XPath: adds rows that enable you to specify signature and/or encryption for an XPath expression or a URI which indicates the version of XPath to use. The Required field is selected by default. Only one XPath element is allowed. ■ Remove: removes a selected row.	Sign
Output Message Settings		

Attribute	Description	Value
Message Parts	Specifies the message parts that must be signed and/or encrypted. Click the Message Parts button to open the Message Parts dialog box. See *Message Parts* under Input Message above for more information.	

Client Configuration — Web Service Attributes

The Client Configuration web service attributes exposed in the WS Policy Attachment Editor are dependent on the project and the server configuration.

The attributes exposed by the HTTP Binding Component are described in the following table.

Attribute	Description	Value
Transport Settings		
Automatically Select Optimal Encoding (XML/Fast Infoset)	Specifies whether to use XML or Fast Infoset encoding. Fast Infoset is a more efficient alternative to XML that uses a binary encoding. If the service is configured to allow Fast Infoset, select this option to use Fast Infoset for faster parsing, faster serializing, and smaller document sizes when compared with equivalent XML documents.	Select the checkbox to enable.

Attribute	Description	Value
Automatically Select Optimal Transport (XML/Fast Infoset)	Specifies whether client runtime checks to see if the service supports TCP. If it does, the client uses TCP transport automatically for service-client communication. TCP provides better performance when sending smaller messages. The performance enhancement is visible mostly in smaller messages because the overhead of sending messages over the HTTP protocol is eliminated. If the service does not support TCP, or if this option is not selected for the client, HTTP is used for transport.	Select the checkbox to enable.
Security Settings		
Use development defaults	Specifies whether to import certificates into the GlassFish Keystore and Truststore so that they can be used immediately for development. The security mechanisms require the use of v3 certificates. The default GlassFish Keystore and Truststore do not contain v3 certificates at this time. In order to use message security mechanisms with GlassFish, it is necessary to obtain Keystore and Truststore files that contain v3 certificates and import the appropriate certificates into the default GlassFish stores. In addition to importing certificates, when this option is selected a default user is created in the file realm with username wsitUser. For a production environment, provide your own certificates and user settings.	Select the checkbox to enable.

Attribute	Description	Value
Keystore	Click the Keystore button to open the Keystore Configuration Editor. The editor specifies the following information: ■ Location: Specifies the directory and file name containing the certificate key to be used to authenticate the client. Use the Browse button to specify the location and name. ■ Keystore Password: Specifies the password for the keystore used by the client. The default GlassFish password is `changeit`. ■ Alias: Specifies the alias of the certificate in the specified keystore to be used for authentication. ■ Load Aliases: Click this button to populate the Alias list with all of the certificates available in the selected keystore. This option will only work if the keystore location and password are correct. ■ Key Password: Specifies the password of the key within the keystore. By default, the key password uses the store password. Only specify a password in this field when the key password is different. ■ Alias Selector Class: Specifies the selector class for aliases.	Configure the Keystore from the Keystore Configuration Editor.

Attribute	Description	Value
Truststore	Click the Truststore button to open the Truststore Configuration Editor. The editor specifies the following information: ■ Location: Specifies the directory and file name of the client truststore containing the certificate of the server. Use the Browse button to select the location and file name. ■ Truststore Password: Specifies the password for the Truststore used by the client. If you are running under GlassFish, GlassFish's password is `changeit`. ■ Alias: Specifies the peer alias of the certificate in the truststore that is to be used when the client needs to send encrypted data. ■ Load Aliases: Clicking the Load Aliases button populates the Alias field with the aliases contained in the truststore file. The Location and Truststore Password fields must be specified correctly for this option to work. ■ Certificate Selector: Specifies a String which specifies the identities of zero or more certificates. The specifiers can conform to X.509 naming conventions. A certificate selector can also use various shortcuts to match either subject alternative names, the filename, or even the issuer.	Configure the Truststore from the Truststore Configuration Editor.
Authentication Credentials	Specifies whether the Authentication Credentials are Dynamic or Static. The two proceeding property fields that are associated with Authentication Credentials change, depending on the Authentication Credentials property value. When the value is set as `Static`, specify the default username and password. Note: The Static option has a risk of exposing the password as a plain text String stored in the WSIT client side configuration. However, when used in the context of GlassFish, this static option has a special utility for embedded web service clients (Example: A servlet or an EJB acting as a web service Client). The Password in this case can be specified as a placeholder by starting the password String start with a "$" character. The WSIT security runtime then makes a SecretKeyCallback passing the password placeholder (minus the "$" character). The actual password is then obtained as a result of the SecretKeyCallback. For more information see WSIT Security Configuration Demystified (`http://xwss.dev.java.net/articles/security_config.html`)	Dynamic

Attribute	Description	Value
Username Callback Handler *or* Username	Specifies the Username Callback Handler (when the Authentication Credentials value is set as Dynamic). A CallbackHandler is a class that implements a javax.security.auth.callback. For the Username Callback Handler (javax.security.auth.callback.NameCallback), the NameCallback is used to retrieve the Username. This is necessary when the Security Mechanism requires the client to supply a Username and a Password. The CallbackHandler invocation only applies to a Plain J2SE web service client. For more information see WSIT Security Configuration Demystified (http://xwss.dev.java.net/articles/security_config.html)	Username Callback Handler
	Specifies the name of an authorized user (when the Authentication Credentials value is set as Static). This option is best used only in the development environment. When the Default Username and Default Password are specified, the username and password are stored in the wsit-client.xml file in clear text, which presents a security risk. Do not use this option for production.	Username
Password Callback Handler *or* Password	Specifies the Username Callback Handler (when the Authentication Credentials value is set as Dynamic). For the Password Callback Handler (javax.security.auth.callback.PasswordCallback), the PasswordCallback is used to retrieve the Password. This is necessary when the Security Mechanism requires the client to supply a Username and a Password. The CallbackHandler invocation only applies to a Plain J2SE web service Client. For more information see WSIT Security Configuration Demystified (http://xwss.dev.java.net/articles/security_config.html)	Password Callback Handler
	Specifies the password for the authorized user (when the Authentication Credentials value is set as Static). This option is best used only in the development environment. When the Default Username and Default Password are specified, the username and password are stored in the wsit-client.xml file in clear text, which presents a security risk. Do not use this option for production.	Password

Attribute	Description	Value
SAML Callback Handler	Specifies the SAML Callback Handler. To use a SAML Callback Handler, you need to create one, as there is no default. A CallbackHandler is a class that implements a javax.security.auth.callback. The SAML Callback Handler (com.sun.xml.wss.impl.callback.SAMLCallback), is necessary when using a Security Mechanism that requires the client to supply a SAMLAssertion, such as a Sender-Vouches or a Holder-of-Key assertion. For more information see WSIT Security Configuration Demystified (http://xwss.dev.java.net/articles/security_config.html)	SAML Callback Handler

Advanced Configuration Settings

Attribute	Description	Value
RM Resend Interval (ms)	Specifies the time interval (in milliseconds) at which the sender resends unacknowledged messages to the receiver. By default, the resend happens every 2000ms.	2000
RM Close Timeout (ms)	Specifies the interval (in milliseconds) at which the client waits for a close() call to return. If unacknowledged messages are received after this interval is reached, and the call to close has returned, an error is logged regarding the lost messages.	0
RM Ack Request Interval (ms)	Specifies the suggested minimum interval (in milliseconds) that the sender should allow to elapse between Acknowledgement requests to the receiver.	200
Secure Session Token Lifetime (ms)	Specifies the life span of the security session (the interval at which the security session expires).	36000
Renew Expired Secure Session Tokens	Specifies whether expired secure session tokens are renewed.	Select the checkbox to enable.
Require Cancel of Secure Session	Specifies whether cancel of secure session is enabled.	Select the checkbox to enable.
Maximum Clock Skew (ms)	Specifies the maximum difference allowed between the system clocks of the sender and recipient in milliseconds.	300000
Timestamp Freshness Limit (ms)	Specifies the Timestamp Freshness Limit in milliseconds. Timestamps received with a creation time older than the Timestamp Freshness Limit period are rejected by the receiver.	300000
Use Default Certificate Revocation Mechanism	If this option is selected, the default revocation checking mechanism of the underlying PKIX service provider is used.	Select the checkbox to enable.

HTTP Binding Component Security

The HTTP Binding Component provides external connectivity between the JBI environment and external environments. To ensure that transactions are secure, the HTTP Binding Component employs both transport and message security.

- **Transport Layer Security** provides mechanisms to secure network transactions between clients and servers, such as the username/password used in HTTP Basic authentication and authorization.
- **Message Layer Security** uses security information contained within the message or message attachment, such as the UsernameToken used in SOAP Message Security.

This section contains the following topics:

Using Basic Authentication with the HTTP Binding Component

Basic authentication enables you to require credentials, in the form of a username and password, to make a transaction. These credentials are transmitted as plain text. The username and password are encoded as a sequence of base-64 characters before transmission to ensure privacy. So, for example, the user name "Fred" and password "Dinosaur" are combined as "Fred:Dinosaur." When encoded in base-64, these characters are equivalent to "RnJlZDpEaW5vc2F1cg0K".

For a Provider web service, a request message from a client contains the user name and password fields in the request header.

For a Consumer web service invoking a web service with basic authentication enabled, the user name and password are appended to the request headers for authentication.

For more information on basic authentication protocol see RFC 1945 (Hypertext Transfer Protocol HTTP/1.0), RFC 2616 (Hypertext Transfer Protocol HTTP/1.1), and RFC 2617 (HTTP Authentication: Basic and Digest Access Authentication).

Basic Authentication Supported Features

Basic authentication is supported by specifying a policy in the WSDL. A basic authentication policy can be added to the WSDL either manually or by using the WS-Policy Attachment window accessed from CASA and provided through Tango (WSIT). A basic authentication policy is specified at the root level of the WSDL and a reference to the policy is made in the WSDL Port type section, binding the policy to the endpoint.

To support basic authentication, the HTTP Binding Component defines the following WSDL elements:

- **MustSupportBasicAuthentication**: This element has an attribute called *on* which can be used to turn authentication on or off. This attribute accepts the values true or false. The MustSupportBasicAuthentication element within a policy is required to enable basic authentication in the endpoint.

- **UsernameToken**: This element specifies the user name and password fields for one of the following actions:

    - Authenticate the request when the endpoint is a provider

    - Invoke a web service with basic authentication enabled when the configured endpoint is a consumer

        The user name and password fields can be specified either as plain text in the WSDL, or as tokens in the WSDL and configured at runtime.

Authentication Mechanisms for Consumer Endpoints

Four types of authentication mechanisms are supported for web service consumer endpoints.

A consumer endpoint can be configured to use one of these mechanisms by adding it as a child element to the MustSupportBasicAuthentication element of the endpoints Policy.

- "*WssTokenCompare* Username/Password Authentication" on page 83: Compares the username and password extracted from the HTTP Authorization request header with the username and password specified in the Policy's WssUsernameToken10 and WssPassword elements.

- "Using the Access Manager for Authentication and Authorization" on page 84: Configures the consuming endpoint to use the Access Manager to authenticate the HTTP client's credentials.

- "Using the OpenSSO Web Services Security (WSS) Agent for Authentication and Authorization" on page 87: Configures the consuming endpoint to use the OpenSSO Web Services Security Agent to authenticate the HTTP client's credentials.

- "Using the GlassFish Realm Security to Authenticate the HTTP Client Credentials" on page 89: Configures the consuming endpoint to use the Realm security to authenticate the HTTP client's credentials.

The following sections describe these mechanisms in more detail.

WssTokenCompare **Username/Password Authentication**

To use the WssTokenCompare feature, the Policy element must be present, and specify the username and password that are used for authentication. The username and password extracted from the HTTP Authorization request header are compared with the username and password specified in the Policy's WssUsernameToken10 and WssPassword elements.

The following sample WSDL contains the policy and its reference to use `WssTokenCompare`. Note that an application variable token is used for the password so that the password is not exposed in the WSDL. The value of the password can be specified in the component's `Application Variable` property in NetBeans.

```
<wsdl:service name="echoService">        <wsdl:port name="echoPort"
binding="tns:echoBinding">           <soap:address location="http://pponnala-tecra-xp.stc.com:18181/
        echoService/echoPort"/>          <wsp:PolicyReference URI="#HttpBasicAuthBindingBindingPolicy"/>
    </wsdl:port> </wsdl:service>  <wsp:Policy wsu:Id="HttpBasicAuthBindingBindingPolicy">
    <mysp:MustSupportBasicAuthentication on="true">            <mysp:BasicAuthenticationDetail>
          <mysp:WssTokenCompare/>             </mysp:BasicAuthenticationDetail>
    </mysp:MustSupportBasicAuthentication>       <mysp:UsernameToken
mysp:IncludeToken="http://schemas.xmlsoap.org/ws/     2005/07/securitypolicy/IncludeToken/AlwaysToRecipient">
       <wsp:Policy>              <sp:WssUsernameToken10>wilma</sp:WssUsernameToken10>
          <sp:WssPassword>${pass_token}</sp:WssPassword>
     </wsp:Policy>    </mysp:UsernameToken> </wsp:Policy>
```

Note – The code displayed above is wrapped for display purposes.

Using the Access Manager for Authentication and Authorization

To use Access Manager to configure access-level authorization, you configure the consuming endpoint to use the Access Manager to authenticate the client's credentials. The HTTP Binding Component SOAP binding integrates seamlessly with Access Manager to authenticate the HTTP client's credentials (the username and password extracted from the HTTP Authorization header) against the user's credentials in the Access Manager database.

Installing the Access Manager Add-on

Access Manager is installed as an GlassFish add-on which includes the Access Manager Server and JAR files. To install Access Manager, do the following:

1. Download the standalone (15 MB) Sun Java System Access Manager 7.1 Patch 1. You can locate the installation file through Oracle support at http://www.oracle.com/technetwork/indexes/downloads/sdlc-decommission-333274.html.

2. Extract the access_manager-7_1–p01–rr.zip file to the following directory: /GlassFishESBv21/glassfish/bin/accessmanager

3. Install the Access Manager add-on to GlassFish using the following asadmin command from your CLI: /GlassFishESBv21/glassfish/bin/asadmin install-addon /accessmanager/am_installer.jar

   Access Manager is extracted into /GlassFishESBv21/glassfish/addons/accessmanager with the necessary JAR files and AMConfig.properties.

4. Restart the GlassFish server. Upon restart the post-configuration is done automatically for Access Manager

Installing Access Manager with Java Application Platform SDK

You can also download Access Manager as part of the Java Application Platform SDK installation, following the SDK Update 7 Installation Instructions. Upon installation, the Access Manager is available in the SDK install directory in the `addons/accessmanager` directory.

To configure and deploy the Access Manager instance that is installed with the SDK package from GlassFish ESB, modify the `server.policy` file of GlassFish from Java CAPS as follows:

From the Command Line:

1. Copy (cp) `/JavaCAPS/appserver/domains/domain1/config/server.policy` to `/JavaCAPS/appserver/domains/domain1/config/server.policy.Orig`

2. Cat `/~<SDK_location>/addons/accessmanager/as9.0_serverpolicy` to `/JavaCAPS/appserver/domains/domain1/config/server.policy`.

3. Autodeploy amserver.war.

   Copy (cp) `/~<SDK_location>/addons/accessmanager/amserver.war` to `/JavaCAPS/appserver/domains/domain1/autodeploy`

4. Restart GlassFish

   `/JavaCAPS/stop_glassfish_domain1`

   `/JavacAPS/start_glassfish_domain1`

Configure the HTTP Binding Component to use Access Manager

To configure the Sun Access Manager Configuration Directory, do the following:

1. Access the HTTP Binding Component Properties from the NetBeans Services window. Right-click `sun-http-binding` under `Servers → GlassFish → JBI → Binding Components`, and choose `Properties` from the pop-up menu.

2. Configure the Sun Access Manager Configuration Directory property to specify the location of the Access Manager's `AMConfig.properties` file. For example: `C:/JavaCAPS/appserver/addons/accessmanager`.

   Configure the Sun Access Manager Configuration Directory property to point to the directory that contains the AMConfig.properties file. For example: `C:/GlassFishESBv21/glassfish/addons/accessmanager`

3. Configure the Sun Access Manager Classpath property to point to the following JAR files extracted to the `/JavacAPS/appserver/addons/accessmanager` directory:

   - amclientsdk.jar
   - amWebServicesProvider.jar

   Note that the two files must be separated by a comma.
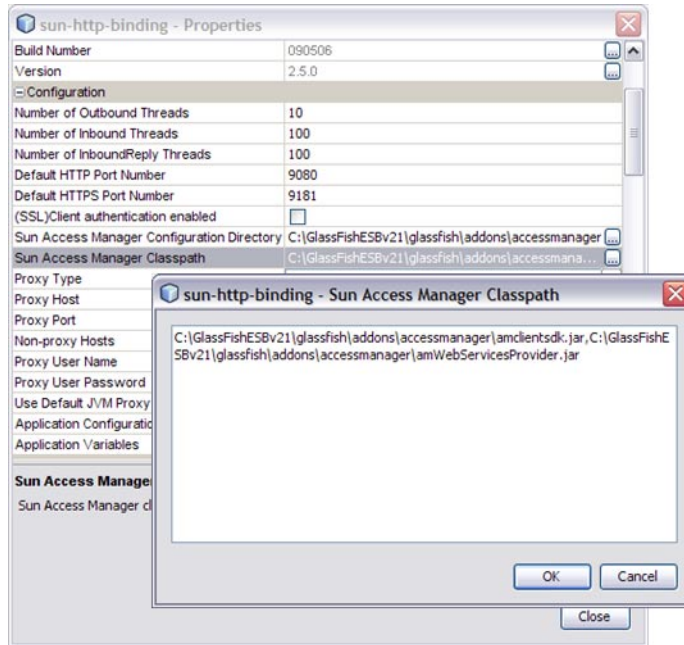
4. Modify the AMConfig.properties file as needed to connect to Access Manager. At a minimum, the following properties must be configured:

```
com.iplanet.am.naming.url=@PROTOCOL@:      //@SERVER_HOST@:@SERVER_PORT@/@DEPLOY_URI@/namingservice
com.sun.identity.agents.app.username=@APPLICATION_USER@ com.iplanet.am.service.password=@APPLICATION_PASSWD@
am.encryption.pwd=@ENCRYPTION_KEY@ com.iplanet.am.server.protocol=@SERVER_PROTOCOL@
com.iplanet.am.server.host=@SERVER_HOST@ com.iplanet.am.server.port=@SERVER_PORT@
com.iplanet.am.services.deploymentDescriptor=@DEPLOY_URI@ com.sun.identity.loginurl=@SERVER_PROTOCOL@://@SERVER_HOST@:
    @SERVER_PORT@/@DEPLOY_URI@/UI/Login com.sun.identity.liberty.authnsvc.url=@SERVER_PROTOCOL@://
    @SERVER_HOST@:@SERVER_PORT@/@DEPLOY_URI@/Liberty/authnsvc
```

5. Configure the policy in the WSDL to enable Authorization by changing the Access Manager authorization attribute to true (note the attribute authorization="true" in the example below). This attribute is optional and the default value is false.

The following sample WSDL contains the policy and its reference to use Access Manager.

```
<service name="AuthAMService">        <port name="AuthAMPort"
binding="tns:AuthAMBinding">               <soap:address location="http://localhost:${HttpDefaultPort}/AuthAMService
/AuthAMPort"/>          <wsp:PolicyReference URI="#HttpAuthorizationBindingAMPolicy"/>
    </port> </service> <wsp:Policy wsu:Id="HttpAuthorizationBindingAMPolicy">
    <mysp:MustSupportBasicAuthentication on="true">          <!--
authenticationType is one of simple, am, or realm -->          <mysp:BasicAuthenticationDetail>
          <mysp:AccessManager authorization="true"/>          </mysp:BasicAuthenticationDetail>
    </mysp:MustSupportBasicAuthentication> </wsp:Policy>
```

For more information on HTTP Binding Component authorization using Sun Access Manager, and Access Manager Classpath configuration, see: HTTP BC Access Manager Authorization.

> **Note –** When OpenSSO Enterprise Server is running on an HTTPS port, the certificates on the OpenSSO Enterprise server must be installed on the client side in order for the HTTP Binding Component to access the server. Certificates should be installed in the GlassFish domain config directory. For example: `\GlassFishESBv21\glassfish\domains\domain1\config`.

Using the OpenSSO Web Services Security (WSS) Agent for Authentication and Authorization

To configure access-level authorization using OpenSSO Web Services Security Agent, you configure the consuming endpoint to use OpenSSO WSS Agent to authenticate the client's credentials (the username and password extracted from the HTTP Authorization header) against the user's credentials in the WSS Agent database. OpenSSO Web Services Security Agent allows the HTTP Binding Component to talk to OpenSSO Enterprise Server 8 installed on a remote or local computer.

Install OpenSSO Enterprise Server

To use the OpenSSO Web Services Security Agent, first download and install OpenSSO Enterprise Server following the directions and requirements presented in the Installing and Configuring a Single OpenSSO Enterprise Instance document.

Configure the HTTP Binding Component to use OpenSSO Web Service Security

With OpenSSO Enterprise Server installed, you can now configure the HTTP Binding Component to use the OpenSSO WSS Agent. The OpenSSO WSS Agent file contains the client configuration AMConfig.properties, and OpenSSO ClientSDK, that allow web service providers and clients to easily integrate, to validate and secure web service communications.

1. Download openssowssproviders.zip.
2. Create a directory, such as /JavaCAPS/WSSAgent_OpenSSO/, and extract the contents of the openssowssproviders.zip file into it.
3. From the NetBeans Services window, make sure that the GlassFish server is started. If not, right-click GlassFish and choose Start from the pop-up menu.
4. Open the HTTP Binding Component Properties. To do this, expand Servers → GlassFish → JBI → Binding Components in the Services window, right-click `sun-http-binding` and select Properties from the pop-up menu.

   The HTTP Binding Component Properties Editor appears.
5. Configure the Sun Access Manager Classpath property to point to the following JAR files that you extracted to the WSSAgent_OpenSSO directory:

   - openssoclientsdk.jar
   - openssowssproviders.jar

   Note that the two files must be separated by a comma.
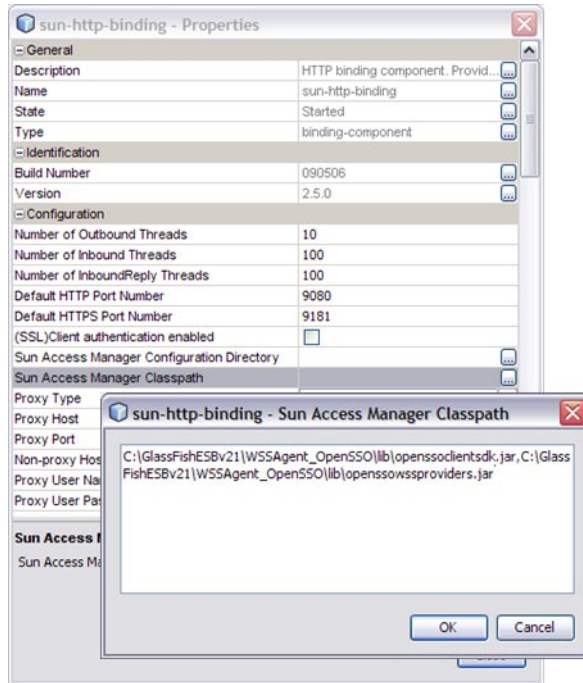
6. Configure the Sun Access Manager Configuration Directory property to point to the directory that contains the AMConfig.properties file. This is located in the ./resources directory extracted to the WSSAgent_OpenSSO directory.
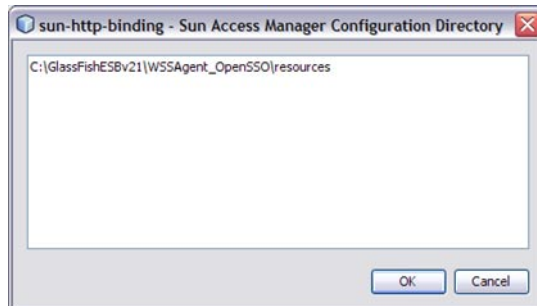
7. Modify the AMConfig.properties file as needed to connect to OpenSSO Enterprise Server. At a minimum, the following properties must be configured:

```
com.iplanet.am.naming.url=@PROTOCOL@:    //@SERVER_HOST@:@SERVER_PORT@/@DEPLOY_URI@/namingservice
com.sun.identity.agents.app.username=@APPLICATION_USER@ com.iplanet.am.service.password=@APPLICATION_PASSWD@
am.encryption.pwd=@ENCRYPTION_KEY@ com.iplanet.am.server.protocol=@SERVER_PROTOCOL@
com.iplanet.am.server.host=@SERVER_HOST@ com.iplanet.am.server.port=@SERVER_PORT@
com.iplanet.am.services.deploymentDescriptor=@DEPLOY_URI@ com.sun.identity.loginurl=@SERVER_PROTOCOL@://@SERVER_H
    @SERVER_PORT@/@DEPLOY_URI@/UI/Login com.sun.identity.liberty.authnsvc.url=@SERVER_PROTOCOL@://
    @SERVER_HOST@:@SERVER_PORT@/@DEPLOY_URI@/Liberty/authnsvc
```

8. Restart GlassFish and HTTP BC and test the modifications.

Using the GlassFish Realm Security to Authenticate the HTTP Client Credentials

The HTTP Binding Component can integrate with GlassFish Application Server, out of the box, to provide authentication of requesting clients by authenticating the client against the credentials in a "realm". To take advantage of this security feature, the HTTP/SOAP Binding Component's consuming endpoint needs to be properly configured in the WSDL.

To configure an HTTP/SOAP endpoint to use Realm security configure the `PolicyReference` element which belongs in the namespace, `http://schemas.xmlsoap.org/ws/2004/09/policy`. The `PolicyReference` identifies the Policy, which also belongs in the namespace, `http://schemas.xmlsoap.org/ws/2004/09/policy`, that provides the details for configuring Realm security.

This is an example of an endpoint with an associated PolicyReference element.

```
<port name="SoapBasicAuthPortRealm" binding="tns:SoapBasicAuthRealmBinding">
    <soap:address location="http://localhost:12081/SoapBasicAuthService
/SoapBasicAuthRealmPort"/>       <wsp:PolicyReference URI="#HttpBasicAuthBindingBindingRealmPolicy"/> </port>
```

The `PolicyReference` element contains an attribute called `URI`. The value of the URI consists of a '#' character followed by the name of the policy defined somewhere else in the WSDL. Taking this example further, the example below defines the Policy that the `PolicyReference` references. In the following example, ignore the `UsernameToken`. This is used by the "outbound" endpoint for sending the username/password credential when it sends a request. You don't need to have this element for "inbound" (consuming) endpoints, but it's included here to illustrate the bi-directionality of an endpoint.

```
<wsp:Policy wsu:Id="HttpBasicAuthBindingBindingRealmPolicy">
    <mysp:MustSupportBasicAuthentication on="true">          <mysp:BasicAuthenticationDetail>
        <mysp:Realm realmName="file" />            </mysp:BasicAuthenticationDetail>
    </mysp:MustSupportBasicAuthentication>      <mysp:UsernameToken
mysp:IncludeToken="http://schemas.xmlsoap.org/ws/2005 /07/securitypolicy/IncludeToken/AlwaysToRecipient">
        <wsp:Policy>              <sp:WssUsernameToken10>wilma</sp:WssUsernameToken10>
            <sp:WssPassword>pebbles</sp:WssPassword>           </wsp:Policy>
    </mysp:UsernameToken>  </wsp:Policy>
```

> **Note –** The code above has been wrapped for display purposes

The PolicyReference and Policy elements are used above simply to ensure that we adhere to the standard for SOAP binding. There are no Tango WS-Policy Attachments involved and the WS-Policy Attachment "runtime" will ignore the child element MustSupportBasicAuthentication which is specific to the HTTP Soap BC. MustSupportBasicAuthentication is in the namespace, http://sun.com/ws/httpbc/security/BasicauthSecurityPolicy.

For example, your GlassFish installation comes with a preconfigured file realm which is essentially a file-based user database. See the GlassFish documentation on Realm security, or for a demonstration of how Realm security is configured for a SOAP endpoint see Securing Communication using GlassFish Realm Security.

Configuring Security Mechanisms

This section discusses the following Security Mechanisms available through Tango, and the server configuration options for each selection. For more information on any of these security mechanisms, see Security Mechanisms. (http://docs.oracle.com/cd/E17802_01/webservices/webservices/reference/tutorials/wsit/doc/WSIT_Security4.html)

The available security mechanisms are:

- "Username Authentication with Symmetric Key " on page 90
- "Mutual Certificates Security" on page 93
- "Transport Security (SSL)" on page 95
- "Message Authentication over SSL" on page 96
- "SAML Authorization over SSL" on page 98
- "Endorsing Certificate" on page 100
- "SAML Sender Vouches with Certificates" on page 101
- "SAML Holder of Key" on page 104
- "STS Issued Token" on page 106
- "STS Issued Token with Service Certificate" on page 109
- "STS Issued Endorsing Token" on page 112

Username Authentication with Symmetric Key

The Username Authentication with Symmetric Keys mechanism protects your application for integrity and confidentiality. Symmetric key cryptography relies on a single, shared, secret key that is used to both sign and encrypt a message, and is usually faster than public key cryptography.

Server-Side Requirements

The following server-side options need to be configured for this security mechanisms:

- **Keystore:** Configure the Keystore to specify the alias identifying the service certificate and private key. For the GlassFish Keystores, the file is `keystore.jks` and the alias is `xws-security-server`, assuming that you have updated the GlassFish default certificate stores.

- **User in GlassFish:** Add a user to the file realm of GlassFish to use a mechanism that requires a user database for authentication.

Client-Side Requirements

The following client-side options need to be configured for this security mechanisms:

- **Truststore:** Configure the Truststore that contains the certificate and trusted roots of the server. For the GlassFish truststores, the file is `cacerts.jks` and the alias is `xws-security-server`, assuming that you have updated the GlassFish default certificate stores.
  When using an STS mechanism, the client specifies the Truststore and certificate alias for the STS, not the service. For the GlassFish stores, the file is `cacerts.jks` and the alias is `wssip`.

- **Default User:** Configure either a default username and password, a UsernameCallbackHandler, or leave these options blank and specify a user at runtime.

- **User in GlassFish:** Add a user to the file realm of GlassFish to use a mechanism that requires a user database for authentication.

TABLE 21 Username Authentication with Symmetric Key Configuration Properties

Property	Description	Value
Authentication Token	Specifies which supporting token will be used to sign and/or encrypt the specified message parts. Options include Username, X509, SAML, Issued, or None	Username

TABLE 21 Username Authentication with Symmetric Key Configuration Properties *(Continued)*

Property	Description	Value
Algorithm Suite	Specifies the algorithm suite required to perform cryptographic operations with symmetric or asymmetric key-based security tokens. An algorithm suite specifies actual algorithms and allowed key lengths. A mechanism alternative will define what algorithms are used and how they are used. The value of this attribute is typically referenced by a security binding and is used to specify the algorithms used for all cryptographic operations performed under the security binding. The default value is Basic 128 bit. Some of the algorithm suite settings require that Unlimited StrengthEncryption be configured in the Java Runtime Environment (JRE), particularly the algorithm suites that use 256 bit encryption. For instructions on downloading and configuring unlimited strength encryption, see: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136007.html	Basic 128bit
Security Header Layout	Specifies the layout rule to apply when adding items to the security header. The options are: ■ **Strict:** Items are added to the security header following the general principle of ?declare before use? ■ **Lax:** Items are added to the security header in any order that conforms to WSS: SOAP Message Security. However, WSIT follows Strict even when Lax is selected. ■ **Lax (Timestamp First):** The same as Lax, except that the first item in the security header must be a wsse:Timestamp. ■ **Lax (Timestamp Last):**The same as for Lax, except that the last item in the security header must be a wsse:Timestamp.	Strict
Require Derived Keys	Specifies that a derived key is required. A derived key is a cryptographic key created from a password or other user data. Derived keys allow applications to create session keys as needed, eliminating the need to store a particular key. The use of the same session key (for example, when using Secure Conversation) for repeated message exchanges is sometimes considered a risk. To reduce that risk, enable Require Derived Keys.	Select the checkbox to enable.

TABLE 21 Username Authentication with Symmetric Key Configuration Properties *(Continued)*

Property	Description	Value
Establish Secure Session (Secure Conversation)	Secure Session enables establishes a shared security context between the consumer and provider when a multiple-message-exchange sequence is first initiated. Subsequent messages use (possibly derived) session keys that increase the overall security while reducing the security processing overhead for each message. When this option and Require Derived Keys are both enabled, a derived key will be used. If not, the original session key will be used. Note on Secure Session and Reliable Message Delivery: Reliable Messaging can be used independently of the security mechanisms; however, when used with a security mechanism, Reliable Messaging requires the use of Secure Session, which will be automatically configured for a security mechanism when Reliable Messaging is selected before the security mechanism is selected. If Secure Session is selected for a security mechanism and the Reliable Messaging option is not selected before the security mechanism is specified, Reliable Messaging will need to be manually selected in order for Secure Session to work.	Select the checkbox to enable.
Require Derived Keys for Secure Session	Specifies that a derived key is required for Secure Session. See Require Derived Key above for more information.	Select the checkbox to enable.
Require Signature Confirmation	Specifies that the responder process the signature in the request. Select this option to reduce the risk of attacks when the WSS Version is 1.1 .	Select the checkbox to enable.
Encrypt Signature	Specifies whether the primary signature and signature confirmation elements must be encrypted.	Select the checkbox to enable.
Encrypt before Signing	Specifies that the order of message protection is to encrypt the SOAP content, then sign the entire SOAP body. The encryption key and signing key must be derived from the same source key. If not selected, the default behavior is Sign Before Encrypt.	Check box Selected indicates disabled.

Mutual Certificates Security

The Mutual Certificates Security mechanism uses security through authentication and message protection to ensure integrity and confidentiality. This mechanism requires a keystore and truststore file for both the client and server sides of the application.

For an example of configuring WS Security for Mutual Certificates Security see Using the WSIT Mutual Certificates Security Mechanism with the HTTP BC (`http://wiki.open-esb.java.net/Wiki.jsp?page=HTTPBCWSITMutualCerts`)

Server-Side Requirements

The following server-side options need to be configured for this security mechanisms:

- **Keystore:** Configure the Keystore to specify the alias identifying the service certificate and private key. For the GlassFish Keystores, the file is `keystore.jks` and the alias is `xws-security-server`, assuming that you have updated the GlassFish default certificate stores.

- **Truststore (no alias):** Configure the Truststore to specify the alias that contains the certificate and trusted roots of the client. For the GlassFish Truststores, the file is `cacerts.jks` and the alias is `xws-security-client`, assuming that you have updated the GlassFish default certificate stores.

Client-Side Requirements

The following client-side options need to be configured for this security mechanisms:

- **Keystore:** Configure the keystore to point to the alias for the client certificate. For the GlassFish Keystores, the file is `keystore.jks` and the alias is `xws-security-client`, assuming that you have updated the GlassFish default certificate stores.

- **Truststore:** Configure the Truststore that contains the certificate and trusted roots of the server. For the GlassFish truststores, the file is `cacerts.jks` and the alias is `xws-security-server`, assuming that you have updated the GlassFish default certificate stores.
  When using an STS mechanism, the client specifies the Truststore and certificate alias for the STS, not the service. For the GlassFish stores, the file is `cacerts.jks` and the alias is `wssip`.

TABLE 22   Mutual Certificates Security Configuration Properties

Property	Description	Value
Algorithm Suite	Specifies the algorithm suite required to perform cryptographic operations with symmetric or asymmetric key-based security tokens. See Algorithm Suite under Table 21 for more information.	Basic 128bit
Security Header Layout	Specifies the layout rule to apply when adding items to the security header. Options are Strict, Lax, Lax (Timestamp First), and Lax (Timestamp Last). See Security Header Layout under Table 21 for more information.	Strict

TABLE 22   Mutual Certificates Security Configuration Properties   *(Continued)*

Property	Description	Value
Require Derived Keys	Specifies that a derived key is required. A derived key is a cryptographic key created from a password or other user data. Derived keys allow applications to create session keys as needed, eliminating the need to store a particular key. The use of the same session key (for example, when using Secure Session) for repeated message exchanges is sometimes considered a risk. To reduce that risk, enable Require Derived Keys.	Select the checkbox to enable.
Establish Secure Session (Secure Conversation)	Secure Session enables establishes a shared security context between the consumer and provider when a multiple-message-exchange sequence is first initiated. Subsequent messages use (possibly derived) session keys that increase the overall security while reducing the security processing overhead for each message. For more information see Establish Secure Session under Table 21.	Select the checkbox to enable.
Require Derived Keys for Secure Session	Specifies that a derived key is required for Secure Session. See Require Derived Keys above for more information.	Select the checkbox to enable.
Encrypt Signature	Specifies whether the primary signature and signature confirmation elements must be encrypted.	Select the checkbox to enable.
Encrypt before Signing	Specifies that the order of message protection is to encrypt the SOAP content, then sign the entire SOAP body. The encryption key and signing key must be derived from the same source key. If not selected, the default behavior is Sign Before Encrypt.	Check box Selected indicates disabled.

Transport Security (SSL)

The Transport Security mechanism uses SSL for authentication and confidentiality during message transport. Transport-layer security relies on secure HTTP transport (HTTPS) using Secure Sockets Layer (SSL). This point-to-point security mechanism that can be used for authentication, message integrity, and confidentiality.

Server-Side Requirements

The following server-side options need to be configured for this security mechanisms:

- **SSL:** Configure the system to point to the client and server Keystore and Truststore files.
- **User in GlassFish:** Add a user to the file realm of GlassFish to use a mechanism that requires a user database for authentication.

Client-Side Requirements

The following client-side options need to be configured for this security mechanisms:

- **SSL:** Configure the system to point to the client and server Keystore and Truststore files.
- **User in GlassFish:** Add a user to the file realm of GlassFish to use a mechanism that requires a user database for authentication.

TABLE 23   Transport Security (SSL) Configuration Properties

Property	Description	Value
Algorithm Suite	Specifies the algorithm suite required to perform cryptographic operations with symmetric or asymmetric key-based security tokens. See Algorithm Suite under Table 21 for more information.	Basic 128bit
Security Header Layout	Specifies the layout rule to apply when adding items to the security header. Options are Strict, Lax, Lax (Timestamp First), and Lax (Timestamp Last). See Security Header Layout under Table 21 for more information.	Strict
Require Client Certificate	Specifies that a client certificate must be provided to the server for verification. If you are using a security mechanism with SSL, a client certificate will be required by the server both during its initial handshake and again during verification.	Check box Selected indicates disabled.

Message Authentication over SSL

The Message Authentication over SSL mechanism attaches a cryptographically secured identity or authentication token with the message and use SSL for confidentiality protection. Authentication is specified through a Username Supporting Token or an X.509 Supporting Token.

Server-Side Requirements

The following server-side options need to be configured for this security mechanisms:

- **SSL:** Configure the system to point to the client and server Keystore and Truststore files.
- **User in GlassFish:** Add a user to the file realm of GlassFish to use a mechanism that requires a user database for authentication.

Client-Side Requirements

The following client-side options need to be configured for this security mechanisms:

- **Keystore:** Configure the keystore to point to the alias for the client certificate. For the GlassFish Keystores, the file is `keystore.jks` and the alias is `xws-security-client`, assuming that you have updated the GlassFish default certificate stores.

- **SSL:** Configure the system to point to the client and server Keystore and Truststore files.

TABLE 24 Message Authentication over SSL Configuration Properties

Property	Description	Value
Authentication Token	Specifies which supporting token will be used to sign and/or encrypt the specified message parts. Options include Username, X509, SAML, Issued, or None	Username
WSS Version	Specifies which version of the Web Services Security specification is followed. Options are 1.0 and 1.1. Enabling WSS 1.1 enables the Server to reuse an encrypted key already generated by the client. This saves the time otherwise required to create a Symmetric Key during the course of response, encrypt it with the client public key (which is also an expensive RSA operation), and transmit the encrypted key in the message (it occupies markup and requires Base64 operations). Enabling WSS 1.1 also enables encrypted headers.	1.1
Algorithm Suite	Specifies the algorithm suite required to perform cryptographic operations with symmetric or asymmetric key-based security tokens. See Algorithm Suite under Table 21 for more information.	Basic 128bit
Security Header Layout	Specifies the layout rule to apply when adding items to the security header. Options are Strict, Lax, Lax (Timestamp First), and Lax (Timestamp Last). See Security Header Layout under Table 21 for more information.	Strict
Establish Secure Session (Secure Conversation)	Secure Session enables establishes a shared security context between the consumer and provider when a multiple-message-exchange sequence is first initiated. Subsequent messages use (possibly derived) session keys that increase the overall security while reducing the security processing overhead for each message. For more information see Establish Secure Session under Table 21.	Select the checkbox to enable.

TABLE 24   Message Authentication over SSL Configuration Properties   *(Continued)*

Property	Description	Value
Require Derived Keys for Secure Session	Specifies that a derived key is required for Secure Session. A derived key is a cryptographic key created from a password or other user data. Derived keys allow applications to create session keys as needed, eliminating the need to store a particular key. The use of the same session key for repeated message exchanges is sometimes considered a risk. To reduce that risk, enable Require Derived Keys for Secure Session.	Select the checkbox to enable.
Require Signature Confirmation	Specifies that the responder process the signature in the request. Select this option to reduce the risk of attacks when the WSS Version is 1.1 .	Select the checkbox to enable.

SAML Authorization over SSL

The SAML Authorization over SSL mechanism attaches an authorization token to the message. SSL is used for confidentiality protection. In this mechanism, the SAML token is expected to carry some authorization information about an end user. The sender of the token is actually vouching for the credentials in the SAML token.

Server-Side Requirements

The following server-side options need to be configured for this security mechanisms:

- **Keystore:** Configure the Keystore to specify the alias identifying the service certificate and private key. For the GlassFish Keystores, the file is keystore.jks and the alias is xws-security-server, assuming that you have updated the GlassFish default certificate stores.

- **Truststore (no alias):** Configure the Truststore to specify the alias that contains the certificate and trusted roots of the client. For the GlassFish Truststores, the file is cacerts.jks and the alias is xws-security-client, assuming that you have updated the GlassFish default certificate stores.

- **SSL:** Configure the system to point to the client and server Keystore and Truststore files.

Client-Side Requirements

The following client-side options need to be configured for this security mechanisms:

- **Keystore:** Configure the keystore to point to the alias for the client certificate. For the GlassFish Keystores, the file is keystore.jks and the alias is xws-security-client, assuming that you have updated the GlassFish default certificate stores.

- **Truststore:** Configure the Truststore that contains the certificate and trusted roots of the server. For the GlassFish truststores, the file is cacerts.jks and the alias is xws-security-server, assuming that you have updated the GlassFish default certificate stores.

   When using an STS mechanism, the client specifies the Truststore and certificate alias for the STS, not the service. For the GlassFish stores, the file is cacerts.jks and the alias is wssip.

- **SAML Callback Handler:** Specify a SAML Callback Handler. To use a SAML Callback Handler, you need to create one, as there is no default.

- **SSL:** Configure the system to point to the client and server Keystore and Truststore files.

TABLE 25   SAML Authorization over SSL Configuration Properties

Property	Description	Value
SAML Version	Specifies which version of the SAML token should be used. The SAML Version is something the CallbackHandler has to verify, not the security runtime. SAML tokens are defined in WSS: SAML Token Profile documents, available from http://www.oasis-open.org/specs/index.php.	1.1 (Profile 1.0)
WSS Version	Specifies which version of the Web Services Security specification is followed. Options are 1.0 and 1.1. Enabling WSS 1.1 enables the Server to reuse an encrypted key already generated by the client. This saves the time otherwise required to create a Symmetric Key during the course of response, encrypt it with the client public key (which is also an expensive RSA operation), and transmit the encrypted key in the message (it occupies markup and requires Base64 operations). Enabling WSS 1.1 also enables encrypted headers.	1.1
Algorithm Suite	Specifies the algorithm suite required to perform cryptographic operations with symmetric or asymmetric key-based security tokens. See Algorithm Suite under Table 21 for more information.	Basic 128bit
Security Header Layout	Specifies the layout rule to apply when adding items to the security header. Options are Strict, Lax, Lax (Timestamp First), and Lax (Timestamp Last). See Security Header Layout under Table 21 for more information.	Strict

TABLE 25 SAML Authorization over SSL Configuration Properties   *(Continued)*

Property	Description	Value
Require Client Certificate	Specifies that a client certificate must be provided to the server for verification. If you are using a security mechanism with SSL, a client certificate will be required by the server both during its initial handshake and again during verification.	Check box Selected indicates disabled.
Require Signature Confirmation	Specifies that the responder process the signature in the request. Select this option to reduce the risk of attacks when the WSS Version is 1.1 .	Select the checkbox to enable.

Endorsing Certificate

The Endorsing Certificate mechanism uses secure messages that use symmetric key for integrity and confidentiality, and an endorsing client certificate to augment the claims provided by the token associated with the message signature. The client knows the service's certificate, and requests need to be endorsed or authorized by a special identity.

Server-Side Requirements

The following server-side options need to be configured for this security mechanisms:

- **Keystore:** Configure the Keystore to specify the alias identifying the service certificate and private key. For the GlassFish Keystores, the file is keystore.jks and the alias is xws-security-server, assuming that you have updated the GlassFish default certificate stores.

- **Truststore:** Configure the Truststore to specify the alias that contains the certificate and trusted roots of the client. For the GlassFish Truststores, the file is cacerts.jks and the alias is xws-security-client, assuming that you have updated the GlassFish default certificate stores.

Client-Side Requirements

The following client-side options need to be configured for this security mechanisms:

- **Keystore:** Configure the keystore to point to the alias for the client certificate. For the GlassFish Keystores, the file is keystore.jks and the alias is xws-security-client, assuming that you have updated the GlassFish default certificate stores.

- **Truststore:** Configure the Truststore that contains the certificate and trusted roots of the server. For the GlassFish truststores, the file is cacerts.jks and the alias is xws-security-server, assuming that you have updated the GlassFish default certificate stores.
  When using an STS mechanism, the client specifies the Truststore and certificate alias for the STS, not the service. For the GlassFish stores, the file is cacerts.jks and the alias is wssip.

TABLE 26 Endorsing Certificate Configuration Properties

Property	Description	Value
Algorithm Suite	Specifies the algorithm suite required to perform cryptographic operations with symmetric or asymmetric key-based security tokens. See Algorithm Suite under Table 21 for more information.	Basic 128bit
Security Header Layout	Specifies the layout rule to apply when adding items to the security header. Options are Strict, Lax, Lax (Timestamp First), and Lax (Timestamp Last). See Security Header Layout under Table 21 for more information.	Strict
Establish Secure Session (Secure Conversation)	Secure Session enables establishes a shared security context between the consumer and provider when a multiple-message-exchange sequence is first initiated. Subsequent messages use (possibly derived) session keys that increase the overall security while reducing the security processing overhead for each message. For more information see Establish Secure Session under Table 21.	Select the checkbox to enable.
Require Derived Keys for Secure Session	Specifies that a derived key is required for Secure Session. See Require Derived Key above for more information. A derived key is a cryptographic key created from a password or other user data. Derived keys allow applications to create session keys as needed, eliminating the need to store a particular key. The use of the same session key for repeated message exchanges is sometimes considered a risk. To reduce that risk, enable Require Derived Keys for Secure Session.	Select the checkbox to enable.
Require Signature Confirmation	Specifies that the responder process the signature in the request. Select this option to reduce the risk of attacks when the WSS Version is 1.1 .	Select the checkbox to enable.
Encrypt Signature	Specifies whether the primary signature and signature confirmation elements must be encrypted.	Select the checkbox to enable.
Encrypt before Signing	Specifies that the order of message protection is to encrypt the SOAP content, then sign the entire SOAP body. The encryption key and signing key must be derived from the same source key. If not selected, the default behavior is Sign Before Encrypt.	Check box Selected indicates disabled.

SAML Sender Vouches with Certificates

This mechanism uses mutual certificates to provide integrity and confidentiality for messages, and uses a Sender Vouches SAML token to provide authorization. The Sender Vouches method establishes correspondence between a SOAP message and the SAML assertions added to the

SOAP message. Confirmation evidence, used to establish correspondence between the subject of the SAML subject statements (in SAML assertions) and SOAP message content, is provided by the attesting entity.

The message payload needs to be signed and encrypted. The requestor is vouching for the credentials (present in the SAML assertion) of the entity on behalf of which the requestor is acting. The initiator token, which is an X.509 token, is used for signature. The recipient token, which is also an X.509 token, is used for encryption. For the server, this is reversed, the recipient token is the signature token and the initiator token is the encryption token. A SAML token is used for authorization.

For an example of configuring WS Security for SAML Sender Vouches with Certificates see Using the SAML Sender Vouches with Certificates Security Mechanism with the HTTP BC (http://wiki.open-esb.java.net/Wiki.jsp?page=HTTPBCWSITSAMLSV)

Server-Side Requirements

The following server-side options need to be configured for this security mechanisms:

- **Keystore:** Configure the Keystore to specify the alias identifying the service certificate and private key. For the GlassFish Keystores, the file is keystore.jks and the alias is xws-security-server, assuming that you have updated the GlassFish default certificate stores.

- **Truststore (no alias):** Configure the Truststore to specify the alias that contains the certificate and trusted roots of the client. For the GlassFish Truststores, the file is cacerts.jks and the alias is xws-security-client, assuming that you have updated the GlassFish default certificate stores.

Client-Side Requirements

The following client-side options need to be configured for this security mechanisms:

- **Keystore:** Configure the keystore to point to the alias for the client certificate. For the GlassFish Keystores, the file is keystore.jks and the alias is xws-security-client, assuming that you have updated the GlassFish default certificate stores.

- **Truststore:** Configure the Truststore that contains the certificate and trusted roots of the server. For the GlassFish truststores, the file is cacerts.jks and the alias is xws-security-server, assuming that you have updated the GlassFish default certificate stores.
  When using an STS mechanism, the client specifies the Truststore and certificate alias for the STS, not the service. For the GlassFish stores, the file is cacerts.jks and the alias is wssip.

- **SAML Callback Handler:** Specify a SAML Callback Handler. To use a SAML Callback Handler, you need to create one, as there is no default.

TABLE 27 SAML Sender Vouches with Certificates Configuration Properties

Property	Description	Value
SAML Version	Specifies which version of the SAML token should be used. The SAML Version is something the CallbackHandler has to verify, not the security runtime. SAML tokens are defined in WSS: SAML Token Profile documents, available from http://www.oasis-open.org/specs/index.php.	1.1 (Profile 1.0)
Algorithm Suite	Specifies the algorithm suite required to perform cryptographic operations with symmetric or asymmetric key-based security tokens. See Algorithm Suite under Table 21 for more information.	Basic 128bit
Security Header Layout	Specifies the layout rule to apply when adding items to the security header. Options are Strict, Lax, Lax (Timestamp First), and Lax (Timestamp Last). See Security Header Layout under Table 21 for more information.	Strict
Require Derived Keys	Specifies that a derived key is required. A derived key is a cryptographic key created from a password or other user data. Derived keys allow applications to create session keys as needed, eliminating the need to store a particular key. The use of the same session key for repeated message exchanges is sometimes considered a risk. To reduce that risk, enable Require Derived Keys.	Select the checkbox to enable.
Establish Secure Session (Secure Conversation)	Secure Session enables establishes a shared security context between the consumer and provider when a multiple-message-exchange sequence is first initiated. Subsequent messages use (possibly derived) session keys that increase the overall security while reducing the security processing overhead for each message. For more information see Establish Secure Session under Table 21.	Select the checkbox to enable.
Require Derived Keys for Secure Session	Specifies that a derived key is required for Secure Session. See Require Derived Keys above for more information.	Select the checkbox to enable.
Encrypt Signature	Specifies whether the primary signature and signature confirmation elements must be encrypted.	Select the checkbox to enable.
Encrypt before Signing	Specifies that the order of message protection is to encrypt the SOAP content, then sign the entire SOAP body. The encryption key and signing key must be derived from the same source key. If not selected, the default behavior is Sign Before Encrypt.	Check box Selected indicates disabled.

SAML Holder of Key

This mechanism protects messages with a signed SAML assertion (issued by a trusted authority) carrying client public key and authorization information with integrity and confidentiality protection using mutual certificates. The Holder-of-Key (HOK) method establishes the correspondence between a SOAP message and the SAML assertions added to the SOAP message. For more information see the SAML Token Profile document at http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.0.pdf.

Server-Side Requirements

The following server-side options need to be configured for this security mechanisms:

- **Keystore:** Configure the Keystore to specify the alias identifying the service certificate and private key. For the GlassFish Keystores, the file is keystore.jks and the alias is xws-security-server, assuming that you have updated the GlassFish default certificate stores.

- **Truststore (no alias):** Configure the Truststore to specify the alias that contains the certificate and trusted roots of the client. For the GlassFish Truststores, the file is cacerts.jks and the alias is xws-security-client, assuming that you have updated the GlassFish default certificate stores.

Client-Side Requirements

The following client-side options need to be configured for this security mechanisms:

- **Keystore:** Configure the keystore to point to the alias for the client certificate. For the GlassFish Keystores, the file is keystore.jks and the alias is xws-security-client, assuming that you have updated the GlassFish default certificate stores.

- **Truststore:** Configure the Truststore that contains the certificate and trusted roots of the server. For the GlassFish truststores, the file is cacerts.jks and the alias is xws-security-server, assuming that you have updated the GlassFish default certificate stores.
  When using an STS mechanism, the client specifies the Truststore and certificate alias for the STS, not the service. For the GlassFish stores, the file is cacerts.jks and the alias is wssip.

- **SAML Callback Handler:** Specify a SAML Callback Handler. To use a SAML Callback Handler, you need to create one, as there is no default.

TABLE 28   SAML Holder of Key Configuration Properties

Property	Description	Value
SAML Version	Specifies which version of the SAML token should be used. The SAML Version is something the CallbackHandler has to verify, not the security runtime. SAML tokens are defined in WSS: SAML Token Profile documents, available from http://www.oasis-open.org/specs/index.php.	1.1 (Profile 1.0)
Algorithm Suite	Specifies the algorithm suite required to perform cryptographic operations with symmetric or asymmetric key-based security tokens. See Algorithm Suite under Table 21 for more information.	Basic 128bit
Security Header Layout	Specifies the layout rule to apply when adding items to the security header. Options are Strict, Lax, Lax (Timestamp First), and Lax (Timestamp Last). See Security Header Layout under Table 21 for more information.	Strict
Require Derived Keys	Specifies that a derived key is required A derived key is a cryptographic key created from a password or other user data. Derived keys allow applications to create session keys as needed, eliminating the need to store a particular key. The use of the same session key for repeated message exchanges is sometimes considered a risk. To reduce that risk, enable Require Derived Keys.	Select the checkbox to enable.
Establish Secure Session (Secure Conversation)	Secure Session enables establishes a shared security context between the consumer and provider when a multiple-message-exchange sequence is first initiated. Subsequent messages use (possibly derived) session keys that increase the overall security while reducing the security processing overhead for each message. For more information see Establish Secure Session under Table 21.	Select the checkbox to enable.
Require Derived Keys for Secure Session	Specifies that a derived key is required for Secure Session. See Require Derived Keys above for more information.	Select the checkbox to enable.
Encrypt Signature	Specifies whether the primary signature and signature confirmation elements must be encrypted.	Select the checkbox to enable.
Encrypt before Signing	Specifies that the order of message protection is to encrypt the SOAP content, then sign the entire SOAP body. The encryption key and signing key must be derived from the same source key. If not selected, the default behavior is Sign Before Encrypt.	Check box Selected indicates disabled.

STS Issued Token

Protects messages using a token issued by a trusted Secure Token Service (STS) for message integrity and confidentiality protection.

To use this mechanism for the web service, select this option as your security mechanism. You must have a Security Token Service that can be referenced by the service. The security configuration for the client-side of this application is dependent upon the security mechanism selected for the STS, and not on the security mechanism selected for the application. The client Truststore must contain the certificate of the STS, which has the alias of wssip if you are using the updated GlassFish certificates.

Server-Side Requirements

The following server-side options need to be configured for this security mechanisms:

- **Keystore:** Configure the Keystore to specify the alias identifying the service certificate and private key. For the GlassFish Keystores, the file is keystore.jks and the alias is xws-security-server, assuming that you have updated the GlassFish default certificate stores.

- **Truststore:** Configure the Truststore to specify the alias that contains the certificate and trusted roots of the client. For the GlassFish Truststores, the file is cacerts.jks and the alias is xws-security-client, assuming that you have updated the GlassFish default certificate stores.

- **STS:** You must have a Security Token Service that can be referenced by the service. The STS is secured using a separate (non-STS) security mechanism.

Client-Side Requirements

The following client-side options need to be configured for this security mechanisms:

- **Keystore:** Configure the keystore to point to the alias for the client certificate. For the GlassFish Keystores, the file is keystore.jks and the alias is xws-security-client, assuming that you have updated the GlassFish default certificate stores.

- **Truststore:** Configure the Truststore that contains the certificate and trusted roots of the server. For the GlassFish truststores, the file is cacerts.jks and the alias is xws-security-server, assuming that you have updated the GlassFish default certificate stores.
  When using an STS mechanism, the client specifies the Truststore and certificate alias for the STS, not the service. For the GlassFish stores, the file is cacerts.jks and the alias is wssip.

- **STS:** You must have a Security Token Service that can be referenced by the service. The STS is secured using a separate (non-STS) security mechanism. The security configuration for the client-side of this application is dependent upon the security mechanism selected for the STS, and not on the security mechanism selected for the application.

TABLE 29 STS Issued Token Configuration Properties

Property	Description	Value
Issuer Address	Specifies the address of the issuer (STS) that will accept the security token presented in the message. The element type is an endpoint reference. An STS contains a set of interfaces used to issue, exchange, and validate security tokens. For example, for JAX-WS services, the Issuer Address is: `http://localhost:8080/jaxws-sts/sts`	http://localhost:8080/jaxws-sts/sts
Issuer Metadata Address	Specifies the address from which to retrieve the issuer metadata. This should just be the URLs. For example, for JAX-WS services, the Issuer Metadata Address is as follows: `http://localhost:8080/jaxws-sts/sts`	http://localhost:8080/jaxws-sts/sts
Token Type	Specifies the type of SAML token required by the service provider. For example: `urn:oasis:names:tc:SAML1.0:assertion`. The options are 1.0, 1.1, or 2.0.	1.1
Key Type	Specifies the type of key preferred by the service provider. The choices are public key or symmetric key: ■ Symmetric Key cryptography relies on a shared secret and is usually faster than Public Key cryptography ■ Public Key cryptography relies on a key that is made public to all and is primarily used for encryption but can be used for verifying signatures. Applies to Issued Token mechanisms only.	Symmetric Key
Algorithm Suite	Specifies the algorithm suite required to perform cryptographic operations with symmetric or asymmetric key-based security tokens. See Algorithm Suite under Table 21 for more information.	Basic 128bit

TABLE 29 STS Issued Token Configuration Properties   *(Continued)*

Property	Description	Value
Security Header Layout	Specifies the layout rule to apply when adding items to the security header. Options are Strict, Lax, Lax (Timestamp First), and Lax (Timestamp Last). See Security Header Layout under Table 21 for more information.	Strict
Require Derived Keys for Issued Token	Specifies that a derived key is required for Secure Session. A derived key is a cryptographic key created from a password or other user data. Derived keys allow applications to create session keys as needed, eliminating the need to store a particular key. The use of the same session key for repeated message exchanges is sometimes considered a risk. To reduce that risk, enable Require Derived Keys for Issued Token.	Select the checkbox to enable.
Establish Secure Session (Secure Conversation)	Secure Session enables establishes a shared security context between the consumer and provider when a multiple-message-exchange sequence is first initiated. Subsequent messages use (possibly derived) session keys that increase the overall security while reducing the security processing overhead for each message. For more information see Establish Secure Session under Table 21.	Select the checkbox to enable.
Require Derived Keys for Secure Session	Specifies that a derived key is required for Secure Session. See Require Derived Keys for Issue Token above for more information.	Select the checkbox to enable.
Require Signature Confirmation	Specifies that the responder process the signature in the request. Select this option to reduce the risk of attacks when the WSS Version is 1.1 .	Select the checkbox to enable.
Encrypt Signature	Specifies whether the primary signature and signature confirmation elements must be encrypted.	Select the checkbox to enable.
Encrypt before Signing	Specifies that the order of message protection is to encrypt the SOAP content, then sign the entire SOAP body. The encryption key and signing key must be derived from the same source key. If not selected, the default behavior is Sign Before Encrypt.	Check box Selected indicates disabled.

STS Issued Token with Service Certificate

Similar to STS Issued Token, except that in addition to the service requiring the client to authenticate using a SAML token issued by a designated STS, confidentiality protection is achieved using a service certificate. A service certificate is used by a client to authenticate the service and provide message protection. For GlassFish, a default certificate of s1as is included.

To use this mechanism for the web service, select this option as your security mechanism. You must have a Security Token Service that can be referenced by the service. The security configuration for the client-side of this application is dependent upon the security mechanism selected for the STS, and not on the security mechanism selected for the application. The client Truststore must contain the certificate of the STS, which has the alias of wssip if you are using the updated GlassFish certificates.

Server-Side Requirements

The following server-side options need to be configured for this security mechanisms:

- **Keystore:** Configure the Keystore to specify the alias identifying the service certificate and private key. For the GlassFish Keystores, the file is keystore.jks and the alias is xws-security-server, assuming that you have updated the GlassFish default certificate stores.

- **Truststore:** Configure the Truststore to specify the alias that contains the certificate and trusted roots of the client. For the GlassFish Truststores, the file is cacerts.jks and the alias is xws-security-client, assuming that you have updated the GlassFish default certificate stores.

- **STS:** You must have a Security Token Service that can be referenced by the service. The STS is secured using a separate (non-STS) security mechanism.

Client-Side Requirements

The following client-side options need to be configured for this security mechanisms:

- **Keystore:** Configure the keystore to point to the alias for the client certificate. For the GlassFish Keystores, the file is keystore.jks and the alias is xws-security-client, assuming that you have updated the GlassFish default certificate stores.

- **Truststore:** Configure the Truststore that contains the certificate and trusted roots of the server. For the GlassFish truststores, the file is cacerts.jks and the alias is xws-security-server, assuming that you have updated the GlassFish default certificate stores.
  When using an STS mechanism, the client specifies the Truststore and certificate alias for the STS, not the service. For the GlassFish stores, the file is cacerts.jks and the alias is wssip.

- **STS:** You must have a Security Token Service that can be referenced by the service. The STS is secured using a separate (non-STS) security mechanism. The security configuration for the client-side of this application is dependent upon the security mechanism selected for the STS, and not on the security mechanism selected for the application.

TABLE 30   STS Issued Token with Service Certificate Configuration Properties

Property	Description	Value
Issuer Address	Specifies the address of the issuer (STS) that will accept the security token presented in the message. The element type is an endpoint reference. An STS contains a set of interfaces used to issue, exchange, and validate security tokens. For example, for JAX-WS services, the Issuer Address is: `http://localhost:8080/jaxws-sts/sts`	http://localhost:8080/jaxws-sts/sts
Issuer Metadata Address	Specifies the address from which to retrieve the issuer metadata. This should just be the URLs. For example, for JAX-WS services, the Issuer Metadata Address is as follows: `http://localhost:8080/jaxws-sts/sts`	http://localhost:8080/jaxws-sts/sts
Token Type	Specifies the type of SAML token required by the service provider. For example: `urn:oasis:names:tc:SAML1.0:assertion`. The options are 1.0, 1.1, or 2.0.	1.1
Key Type	Specifies the type of key preferred by the service provider. The choices are public key or symmetric key: - Symmetric Key cryptography relies on a shared secret and is usually faster than Public Key cryptography - Public Key cryptography relies on a key that is made public to all and is primarily used for encryption but can be used for verifying signatures. Applies to Issued Token mechanisms only.	Symmetric Key

TABLE 30   STS Issued Token with Service Certificate Configuration Properties   *(Continued)*

Property	Description	Value
Key Size	Specifies the size of the symmetric key requested in number of bits. This information is provided as an indication of the desired strength of the security. Valid choices include 128, 192, and 256. The security token is not obligated to use the requested key size, nor is the STS obligated to issue a token with the same key size. That said, the recipient should try to use a key at least as strong as the specified value if possible. Applies to Issued Token mechanisms only.	128
Algorithm Suite	Specifies the algorithm suite required to perform cryptographic operations with symmetric or asymmetric key-based security tokens. See Algorithm Suite under Table 21 for more information.	Basic 128bit
Security Header Layout	Specifies the layout rule to apply when adding items to the security header. Options are Strict, Lax, Lax (Timestamp First), and Lax (Timestamp Last). See Security Header Layout under Table 21 for more information.	Strict
Require Derived Keys	Specifies that a derived key is required A derived key is a cryptographic key created from a password or other user data. Derived keys allow applications to create session keys as needed, eliminating the need to store a particular key. The use of the same session key for repeated message exchanges is sometimes considered a risk. To reduce that risk, enable Require Derived Keys.	Select the checkbox to enable.
Establish Secure Session (Secure Conversation)	Secure Session enables establishes a shared security context between the consumer and provider when a multiple-message-exchange sequence is first initiated. Subsequent messages use (possibly derived) session keys that increase the overall security while reducing the security processing overhead for each message. For more information see Establish Secure Session under Table 21.	Select the checkbox to enable.
Require Derived Keys for Secure Session	Specifies that a derived key is required for Secure Session. See Require Derived Keys above for more information.	Select the checkbox to enable.

TABLE 30 STS Issued Token with Service Certificate Configuration Properties *(Continued)*

Property	Description	Value
Require Signature Confirmation	Specifies that the responder process the signature in the request. Select this option to reduce the risk of attacks when the WSS Version is 1.1 .	Select the checkbox to enable.
Encrypt Signature	Specifies whether the primary signature and signature confirmation elements must be encrypted.	Select the checkbox to enable.
Encrypt before Signing	Specifies that the order of message protection is to encrypt the SOAP content, then sign the entire SOAP body. The encryption key and signing key must be derived from the same source key. If not selected, the default behavior is Sign Before Encrypt.	Check box Selected indicates disabled.

STS Issued Endorsing Token

Similar to STS Issued Token, except that the client authenticates using a SAML token that is issued by a designated STS. An endorsing token is used to sign the message signature.

Message integrity and confidentiality are protected using ephemeral keys encrypted for the service. Ephemeral keys use an algorithm where the exchange key value is purged from the cryptographic service provider (CSP) when the key handle is destroyed. The service requires messages to be endorsed by a SAML token issued by a designated STS.

For this mechanism, the service requires that secure communications be endorsed by a trusted STS. The service does not trust the client directly, but instead trusts tokens issued by a designated STS. In other words, the STS is taking on the role of a second service with which the client has to securely authenticate.

To use this mechanism for the web service, select this option as your security mechanism. You must have a Security Token Service that can be referenced by the service. The security configuration for the client-side of this application is dependent upon the security mechanism selected for the STS, and not on the security mechanism selected for the application. The client Truststore must contain the certificate of the STS, which has the alias of wssip if you are using the updated GlassFish certificates.

Server-Side Requirements

The following server-side options need to be configured for this security mechanisms:

- **Keystore:** Configure the Keystore to specify the alias identifying the service certificate and private key. For the GlassFish Keystores, the file is keystore.jks and the alias is xws-security-server, assuming that you have updated the GlassFish default certificate stores.

- **Truststore:** Configure the Truststore to specify the alias that contains the certificate and trusted roots of the client. For the GlassFish Truststores, the file is cacerts.jks and the alias is xws-security-client, assuming that you have updated the GlassFish default certificate stores.

- **STS:** You must have a Security Token Service that can be referenced by the service. The STS is secured using a separate (non-STS) security mechanism.

Client-Side Requirements

The following client-side options need to be configured for this security mechanisms:

- **Keystore:** Configure the keystore to point to the alias for the client certificate. For the GlassFish Keystores, the file is keystore.jks and the alias is xws-security-client, assuming that you have updated the GlassFish default certificate stores.

- **Truststore:** Configure the Truststore that contains the certificate and trusted roots of the server. For the GlassFish truststores, the file is cacerts.jks and the alias is xws-security-server, assuming that you have updated the GlassFish default certificate stores.
  When using an STS mechanism, the client specifies the Truststore and certificate alias for the STS, not the service. For the GlassFish stores, the file is cacerts.jks and the alias is wssip.

- **STS:** You must have a Security Token Service that can be referenced by the service. The STS is secured using a separate (non-STS) security mechanism. The security configuration for the client-side of this application is dependent upon the security mechanism selected for the STS, and not on the security mechanism selected for the application.

TABLE 31 STS Issued Endorsing Token Configuration Properties

Property	Description	Value
Issuer Address	Specifies the address of the issuer (STS) that will accept the security token presented in the message. The element type is an endpoint reference. An STS contains a set of interfaces used to issue, exchange, and validate security tokens. For example, for JAX-WS services, the Issuer Address is: http://localhost:8080/jaxws-sts/sts	http://localhost:8080/jaxws-sts/sts

**TABLE 31** STS Issued Endorsing Token Configuration Properties   *(Continued)*

Property	Description	Value
Issuer Metadata Address	Specifies the address from which to retrieve the issuer metadata. This should just be the URLs. For example, for JAX-WS services, the Issuer Metadata Address is as follows: `http://localhost:8080/jaxws-sts/sts`	http://localhost:8080/jaxws-sts/sts
Token Type	Specifies the type of SAML token required by the service provider. For example: `urn:oasis:names:tc:SAML1.0:assertion`. The options are 1.0, 1.1, or 2.0.	1.1
Key Type	Specifies the type of key preferred by the service provider. The choices are public key or symmetric key: ■ Symmetric Key cryptography relies on a shared secret and is usually faster than Public Key cryptography ■ Public Key cryptography relies on a key that is made public to all and is primarily used for encryption but can be used for verifying signatures. Applies to Issued Token mechanisms only.	Symmetric Key
Key Size	Specifies the size of the symmetric key requested in number of bits. This information is provided as an indication of the desired strength of the security. Valid choices include 128, 192, and 256. The security token is not obligated to use the requested key size, nor is the STS obligated to issue a token with the same key size. That said, the recipient should try to use a key at least as strong as the specified value if possible. Applies to Issued Token mechanisms only.	128
Algorithm Suite	Specifies the algorithm suite required to perform cryptographic operations with symmetric or asymmetric key-based security tokens. See Algorithm Suite under Table 21 for more information.	Basic 128bit

TABLE 31 STS Issued Endorsing Token Configuration Properties   *(Continued)*

Property	Description	Value
Security Header Layout	Specifies the layout rule to apply when adding items to the security header. Options are Strict, Lax, Lax (Timestamp First), and Lax (Timestamp Last). See Security Header Layout under Table 21 for more information.	Strict
Require Derived Keys for X509 Token	Specifies that a derived key is required for X509 Token. See Require Derived Key above for more information. A derived key is a cryptographic key created from a password or other user data. Derived keys allow applications to create session keys as needed, eliminating the need to store a particular key. The use of the same session key for repeated message exchanges is sometimes considered a risk. To reduce that risk, enable Require Derived Keys for X509 Token.	Select the checkbox to enable.
Require Derived Keys for Issued Token	Specifies that a derived key is required for Issued Token. See Require Derived Keys for X509 Token above for more information.	Select the checkbox to enable.
Establish Secure Session (Secure Conversation)	Secure Session enables establishes a shared security context between the consumer and provider when a multiple-message-exchange sequence is first initiated. Subsequent messages use (possibly derived) session keys that increase the overall security while reducing the security processing overhead for each message. For more information see Establish Secure Session under Table 21.	Select the checkbox to enable.
Require Derived Keys for Secure Session	Specifies that a derived key is required for Secure Session. See Require Derived Keys for X509 Token above for more information.	Select the checkbox to enable.
Require Signature Confirmation	Specifies that the responder process the signature in the request. Select this option to reduce the risk of attacks when the WSS Version is 1.1 .	Select the checkbox to enable.
Encrypt Signature	Specifies whether the primary signature and signature confirmation elements must be encrypted.	Select the checkbox to enable.
Encrypt before Signing	Specifies that the order of message protection is to encrypt the SOAP content, then sign the entire SOAP body. The encryption key and signing key must be derived from the same source key. If not selected, the default behavior is Sign Before Encrypt.	Check box Selected indicates disabled.

Using Application Variables to Define Name/Value Pairs

The binding component Application Variables property allows you to define a list of name:value pairs for a given stated type. The application variable name can be used as a token for a WSDL extensibility element attribute in a corresponding binding. For example, if you were defining an application variable for the hostname as FOO, then the WSDL attribute would be ${FOO}. In the Application Variables property you would enter a String value of FOO for the name, and the desired attribute as the value. When you deploy an application that uses application variables, any variable that is referenced in the application's WSDL is loaded automatically.
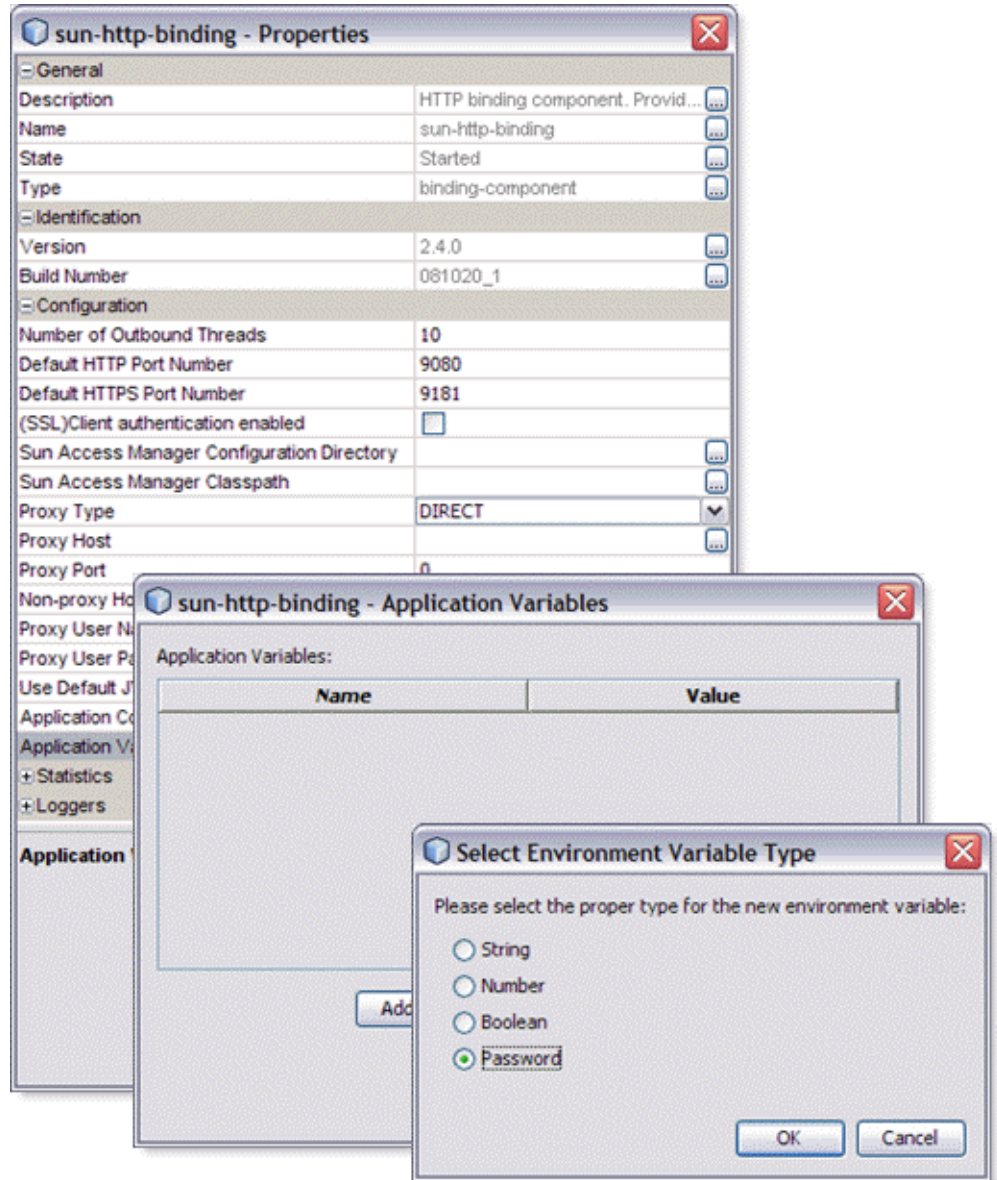
The Application Variables configuration property offers four variable types:

- **String:** Specifies a string value, such as a path or directory.
- **Number:** Specifies a number value.
- **Boolean:** Specifies a Boolean value. The VALUE field provides a checkbox (checked = true).

■ **Password:** Specifies a password value. The password is masked and displays only asterisks.

Variables also allow greater flexibility for your WSDL files. For example, you can use the same WSDL for different runtime environments by using application variables to specify system specific information. These values can then be changed from the binding component runtime properties as needed, for any specific environment.

When you deploy an application that uses Application Variables, all of the Application Variables that are referenced in the application's WSDL files are loaded automatically. If you attempt to start an application and an Application Variables value is not defined (no value is specified for the Application Variable) an exception is thrown.

To change a property when the application is running, change your Application Variable property value, then right-click your application in the Services window under Servers → GlassFish → JBI → Service Assemblies, and click Stop in the popup menu. When you restart your project, your new settings will take effect.

Using Application Variables for password protection

To protect passwords that would otherwise appear as clear text in your WSDL file, you can enter a Password application variable as a token. In the following example, a password application variable is created that uses the name SECRET and the password PROTECT.

▼ Creating a password Application Variable

1 **From the Binding Components directory, under `Servers` → `GlassFish` → `JBI` in the Servers window, select the sun-http-binding.**

The sun-http-binding Properties appear in the Properties window.

2 **Click-on the Application Variables property ellipsis (...) button.**

The Application Variables editor appears.

3 **Click `Add`, select `Password` as your variable type, and click `OK`.**

A new row is added to the Application Variables editor.

4 **Enter `SECRET` as the name, and enter `PROTECT` as the value.**

Because this is a password type, the characters of your password are displayed as asterisks.

5 **Use the application variable name `${SECRET}` as your WSDL password attribute, using the dollar sign and curly braces as shown.**

Using Application Configuration to Configure Connectivity Parameters

The Application Configuration property allows you to configure the external connectivity parameters for an application that you have created, such as a service assembly, and without changing or rebuilding the application, deploy the same application into a different system. For example, you could take an application that is running in a test environment, and deploy it to a production environment without rebuilding the application.

From the Application Configuration property, you can specify values for a Composite Application's external connectivity parameters, which are normally defined in the WSDL service extensibility elements. You can then apply these values to a user-named endpoint Config Extension Property. The Application Configuration property editor includes fields for all of the connectivity parameters that apply to that component's binding protocol. When you enter the name of a saved Config Extension and define the connectivity parameters in the Application Configuration editor, these values override the WSDL defined connectivity attributes when your project is deployed. To change these connectivity parameters again, you simply change the values in the Application Configuration editor, then shutdown and start your Service Assembly to apply the new values.

The Application Configuration property editor allows you to create several application configurations referenced by their own user-defined names. Note that different binding component protocols will have different attributes. The HTTP binding attributes are not the same as the JMS binding attributes, and therefore, the Application Configuration property editors for each of these binding components will contain different attributes.

To change a property when the application is running, change your Application Configuration property value, then right-click your application in the Services window under Servers → GlassFish → JBI Service Assemblies, and click Stop in the popup menu. When you restart your project, your new settings will take effect.

The HTTP Binding Component's Application Configuration property contains one parameter only: HTTP URL Location.

▼ To apply a named Config Extension to the Application Configuration

1 In the CASA editor, specify a name for your endpoint Configuration Extension , from the endpoint's Configuration Extension property.

2 In the Services window, go to the Binding Components directory under the JBI node, right-click the binding component used by your application, and select Properties.

The Properties window appears.

3. **In the binding component's properties, click on the Application Configuration property ellipsis (...) button.**

   The Application Configuration property editor appears.

4. **Enter the user-defined name of the Config Extension you want and define the values for the connection parameters.**

5. **Once the Application Configuration values have been defined, deploy your application.**

Enhanced Logging

The HTTP Binding Component runtime Logger properties include over 30 different component activities that can be monitored and recorded at user-designated levels. Logging levels are set separately for each of these activities from the HTTP Binding Component Properties Editor.

Each logger can be set to record information at any of the following levels:

- **FINEST:** messages provide highly detailed tracing
- **FINER:** messages provide more detailed tracing
- **FINE:** messages provide basic tracing
- **CONFIG:** provides static configuration messages
- **INFO:** provides informative messages
- **WARNING:** messages indicate a warning
- **SEVERE:** messages indicate a severe failure
- **OFF:** no logging messages

Exception messages start with a unique identifier. The HTTP Binding Component exception messages starts with HTTPBC. For example: `HTTPBC-E00101.Start_failed=HTTPBC-E00101: {0} failed to start. {1}`

Statistics Monitoring

The HTTP Binding Component records and maintains statistics for 19 different component activities including exchanges, errors, requests, replies, and so forth. These statistics are recorded during the lifecycle of an endpoint, and accessed from the HTTP Binding Component Properties Editor. For example: statistics for the number of times that a send request has been completed are available in the application's HTTP Binding Component properties as the current value for `Statistics → Sent Requests`.

Using WS-Transaction

The HTTP Binding Component is integrated with WS-Transaction, an implementation of WS-Atomic Transaction available through Tango (WSIT). WS-AtomicTransaction is a specification that defines a two phase protocol to ensure that transactions are fully completed or fully rolled back, also known as "all or nothing." Depending on the transactions success, the registered transaction participants arrive at a commit or abort decision, and both participants are informed of the final result.

The SoapWSATCompositeApp Sample Composite Application

For a sample composite application that demonstrates how to use WS-Transaction with the HTTP Binding Component and the BPEL Service Engine, see HTTP BC AtomicTransactions (`http://wiki.open-esb.java.net/Wiki.jsp?page=HTTPBCWSAtomicTransaction`).

The sample demonstrates the following:

- Using existing Java Transaction APIs (JTA) to initiate and complete the transaction.
- Invocations of transacted web service operations flow transactional context from client to web service.
- Transactional context import from web service client to HTTP/SOAP Binding Component.
- Transaction propagation from the HTTP/SOAP Binding Component to the BPEL Service Engine and from the BPEL Service Engine to the HTTP/SOAP Binding Component .
- Transactional context flow resuming from the HTTP/SOAP Binding Component to the web service.
- Persistent resources updated with client-created transactions are all committed or rolled back as a single atomic transaction.
- After the client-side code commits or aborts the JTA transaction, the client confirms that all operations in the transaction succeeded or failed by using calls to verify methods on the transacted web service.

Clustering Support for the HTTP Binding Component

A cluster is a logical entity encompassing zero or more server instances. Simply speaking, a cluster is a collection of application server instances that can distribute a workload throughout the clustered application instances for optimal performance. These server instances share the same set of applications, resources, and configuration information. A clustered server instance belongs to exactly one cluster, and inherits everything from that parent cluster. Instances in a cluster can extend over any number of computers.

The GlassFish Application Server supports clustering of homogenous application server instances (containing the same set of JBI components, applications, and configuration

information) installed on a single host or on multiple hosts. Applications that run on each application server instance are independent, but are also manageable by an administration infrastructure, either through web browser based (DAS) or command line clients.

HTTP Load Balancer

The HTTP Load Balancer is a web server plug-in that accepts HTTP and HTTPS requests and distributes them to application server instances in a cluster. This allows the HTTP Binding Component to be scaled horizontally, running on multiple instances in a GlassFish Application Server cluster.

The advantages of clustering are many:

- Increases overall system throughput by distributing workload among multiple physical machines
- Servers with different hardware capacities can have the work load distributed in favor of more powerful hosts
- In the event that one particular application server instance is overloaded or becomes unavailable, requests can be re-routed to the least loaded application server instances
- Clustering is invisible to the client. As far as the client is concerned, all HTTP requests are directed to the web server instance where the load balancer is configured

The HTTP Load Balancer includes the following features:

- Sticky Round Robin load balancing algorithm
- Support for multiple clusters
- Configurable health failover capability (less than 30ms)
- Checks and reloads for dynamic changes made to the load balancer configuration
- Support for quiescence - enabling rolling web service upgrades
- Automatic retry of failed requests for impotent URLs
- Configurable error pages

Configuring the HTTP Binding Component for Clustering

For the most part, configuring the HTTP Binding Component for clustering is handled by GlassFish Application Server. The HTTP Binding Component is a pre-installed component in the application server. Default HTTP and HTTPS port numbers are calculated and preassigned when the binding components are installed in the server instances. A web service, serviced by an HTTP Binding Component, is identified by a unique URL identifier with the structure: `"http://<hostname>:<port>/<context>"`.

Each component instance in the cluster must have exclusive access to the resource, therefore a unique port number is assigned to each component instance. A predefined token name is used in the WSDL artifact to resolve the actual port value when the component is deployed into each instance.

Predefined HTTP Port Tokens

Predefined token names:

- "${HttpDefaultPort}" for the HTTP port
- "${HttpsDefaultPort}" for the HTTPS port

These token names are used in lieu of a real port number in the endpoint URL (soap:address) to allow the application client to direct HTTP requests to the default port. The value of the token is then resolved by the HTTP Binding Component, based on the configured default values when an application is deployed.

Note – If you reinstall an HTTP Binding Component, you must reconfigure the default ports properly for each component instance.

Understanding the ${HttpDefaultPort} Token

The following section provides a little background on the ${HttpDefaultPort} token and how it's resolved when an application is deployed.

Just like the GlassFish web services, which are always deployed to a designated HTTP port (8080 is the configured default), the HTTP Binding Component also has a default HTTP port to which web services are deployed. Since the HTTP Binding Component comes with GlassFish as a pre-installed component, a default HTTP port is always assigned to it. The default port is configured in the JBI Runtime module during the installation of GlassFish, at which time it allocates an available port for each HTTP Binding Component instance in the GlassFish domain(s).

Originally, this default port setting and the ${HttpDefaultPort} token were placed in the WSDL URL to support clustering, where multiple HTTP BC instances could be running on the same machine. As such, when an application is deployed, the port token is used to resolve the actual port value to the assigned port in each instance, with no chance of port collisions.

Since then, the use of the port has evolved such that the HTTP Binding Component (the web service container in JBI) acts in a fashion that is similar to the GlassFish web service container. When an application "arrives" in the binding component, it looks up its default HTTP port setting, and replace the token in the URL with the actual port number. If the default port number is not configured, an *Initialization failed* exception is thrown.

Common User Scenarios

The following common user scenarios convey how components interact with external systems to achieve specific business goals. The first five scenarios apply to *design-time* operations, and the remaining scenarios apply to *runtime* operations:

1. "Validating HTTP Extensibility Elements from the WSDL Editor" on page 124
2. "Adding a SOAP Template to a WSDL Document" on page 124
3. "Adding an HTTP Template to a WSDL Document" on page 125
4. "Web Service Client Calling an Operation Using HTTP Basic Authentication" on page 126
5. "Web Service Implementing an Operation Protected by HTTP Basic Authentication" on page 126
6. "Web Service Client Calling an Operation Using SSL Authentication" on page 127
7. "Web Service Implements an Operation Protected by SSL Authentication" on page 128

Validating HTTP Extensibility Elements from the WSDL Editor

In this example, validation of HTTP Extensibility Elements is invoked from the WSDL Editor. This example assumes that you are working with an existing WSDL document containing HTTP extensibility elements.

Results

The WSDL Validation window appears at the bottom of the editor. In a normal flow case, there is a statement saying no errors were found. In the exception flow case, there is a dialog displaying all of the current errors.

Main Scenario

This scenario is the same for both normal flow and exception flow.

1. Double-click the WSDL to open the WSDL Editor.
2. From the WSDL Editor toolbar, click the "Validate XML" button. The Output pane appears at the bottom of the NetBeans IDE.
3. From the Project Explorer, right-click the WSDL file and select "Validate XML" from the pop-up menu. Validation results are displayed in the Output pane.

Adding a SOAP Template to a WSDL Document

In this example, you use the New WSDL Document wizard to generate SOAP Extensibility elements.

Results

The generated WSDL contains SOAP extensibility elements at the binding level, the binding operation level, the binding operation input level, and the port level. The binding level subtype is set to the binding subtype selected in step 4 of the New WSDL Document wizard.

Main Scenario

1. A new WSDL document is created by right-clicking the project in the Project Explorer and selecting "New → WSDL Document" from the pop-up menu. The New WSDL Document wizard appears.

2. Follow steps 1-3 of the wizard to generate a new WSDL document.

3. From step 4 of the wizard, select "SOAP" as the Binding Type. The available binding subtype options appear in the Binding Subtype field.

   Select an appropriate option:
   - RPC Literal
   - Document Literal
   - RPC Encoded

4. Click "Finish" to generate the WSDL document.

Adding an HTTP Template to a WSDL Document

In this example, you use the New WSDL Document wizard to generate HTTP extensibility elements.

Results

The generated WSDL contains HTTP extensibility elements at the binding level, the binding operation level, the binding operation input level, and the port level. The binding level subtype is set to the binding subtype selected in step 4 of the New WSDL Document wizard.

Main Scenario

1. A new WSDL document is created by right-clicking the project in the Project Explorer and selecting "New → WSDL Document" from the pop-up menu. The New WSDL Document wizard appears.

2. Follow steps 1-3 of the wizard to generate a new WSDL document.

3. From step 4 of the wizard, select "HTTP" as the Binding Type. The available binding subtype options appear in the Binding Subtype field.

   Select an appropriate option:
   - Post Operation UrlEncoded
   - Post Operation UrlReplacement
   - Get Operation UrlEncoded
   - Get Operation UrlReplacement

4. Click "Finish" to generate the WSDL document.

Web Service Client Calling an Operation Using HTTP Basic Authentication

In this example, a client invokes a service that requires HTTP Basic Authentication. This example assumes that you are running a deployed BPEL project with a WSDL configured to handle HTTP Basic Authentication. This BPEL project invokes a service protected using HTTP Basic Authentication.

Results

The service processes the expected SOAP Message through HTTP after verifying the security credentials.

Main Scenario

1. A web service client invokes an in-only abstract operation that is implemented by a BPEL process. The abstract operation has a concrete HTTP SOAP binding, so the client must use SOAP over HTTP protocol to properly invoke the operation.
2. The BPEL Process, acting as the client, receives the message for the abstract operation and invokes a different in-only abstract operation. This operation has a concrete HTTP SOAP binding that requires HTTP Basic Authentication.
3. The binding component picks up the normalized message and converts it to a SOAP message.
4. The binding component pulls the appropriate username and password from the Access Manager or from the WSDL.
5. The binding component forwards the message and proper security credentials to the service.

Web Service Implementing an Operation Protected by HTTP Basic Authentication

In this example, a user creates a BPEL project in JBI that is protected by HTTP Basic Authentication. This example assumes that you are running a deployed BPEL project with a BPEL process which implements a service that requires HTTP Basic Authentication.

Results

The JBI process receives the expected SOAP Message through HTTP after verifying the security credentials.

Main Scenario

1. A BPEL Service Engine requires basic authentication for the operation that it implements.

2. The HTTP Binding Component receives the HTTP message and parses out the HTTP Basic Authentication security information.

3. The binding component verifies the security information using a known database of user names and passwords from the Access Manager or from the WSDL.

4. The binding component creates a normalized message and sends it to the Normalized Message Router.

5. A BPEL process, belonging to a BPEL Service Engine, processes the abstract message and returns a status message of either Done or ERROR.

Web Service Client Calling an Operation Using SSL Authentication

In this example, a client invokes a service that requires SSL Authentication. This example assumes that you are running a deployed BPEL project with a WSDL configured for SSL Authentication. This BPEL project invokes a service that is protected by SSL Authentication.

Results

The service receives the expected SOAP Message through HTTP after verifying the security credentials.

Main Scenario

1. The BPEL process acts as the client to the service implementation. The abstract operation has a concrete HTTP SOAP binding, so the client must use SOAP over HTTP protocol to properly invoke the operation.

2. The HTTP Binding Component receives the SOAP message, converts it to a normalized message, and forwards the message to the Normalized Message Router to the awaiting BPEL process.

3. The BPEL Process, acting as the client, receives the abstract operation message and invokes a different in-only abstract operation. This operation has a concrete HTTP SOAP binding that requires SSL Authentication.

4. When the client BPEL process invokes the abstract operation, a normalized message is generated and sent to the Normalized Message Router.

5. The binding component picks up the normalized message and converts it to a SOAP message.

6. The binding component establishes secure communication with the service provider and forwards the request to them.

Web Service Implements an Operation Protected by SSL Authentication

In this example, a server implements a service that requires SSL Authentication. This example assumes that you have deployed a BPEL project with a BPEL process which implements a service that requires SSL authentication.

Results

The service receives the expected SOAP Message through HTTP after verifying the security credentials.

Main Scenario

1. A web service client invokes an In-Only abstract operation that is implemented by a BPEL process. This operation has a concrete HTTP SOAP binding, so the client must use HTTP protocol to properly invoke the operation.

2. The binding component institutes the SSL hand shake and establishes secure communication with the client.

3. The binding component receives the HTTP message and parses out the SSL Authentication security information.

4. The binding component verifies the security information using known SSL certificates.

5. The binding component creates a normalized message and sends it to the Normalized Message Router.

6. A BPEL process processes the abstract message and returns a status message of either `Done` or `ERROR`.