IBM Systems - iSeries

e-business and Web serving
WebSphere Application Server - Express Version 5
Web services

Version 5 Release 4

IBM Systems - iSeries

e-business and Web serving
WebSphere Application Server - Express Version 5
Web services

Version 5 Release 4

Note

Before using this information and the product it supports, be sure to read the information in "Notices," on page 175.

Third Edition (February 2006)

This edition applies to version 5 of IBM WebSphere Application Server - Express for iSeries (5722-IWE) and to all subsequent releases and modifications until otherwise indicated in new editions. This version does not run on all reduced instruction set computer (RISC) models nor does it run on CISC models.

Contents

Web services

Web services are self-contained, modular applications that you can describe, publish, locate, and invoke over a network. Web services reflect a new, service-oriented approach to programming. This approach is based on the idea of building applications by discovering and implementing network services or by invoking available applications to accomplish some task. This approach is independent of specific programming languages or operating systems. Web services delivers interoperability; the ability for components created in different programming languages to work together as if they were created using the same language. Web services rely on existing transport technologies, such as HTTP, and standard data encoding techniques, such as Extensible Markup Language (XML), for invoking the implementation.

See these topics for more information about Web services:

“Web services overview”
This topic discusses the Web services environment, including the roles involved in the Web services life cycle.

“Migrate Web services” on page 3
If you have Web services applications that were developed for WebSphere$^{(R)}$ Application Server Version 4 or Version 5, see this topic for migration instructions.

“Develop Web services” on page 6
See this topic for information about developing Web services and Web services clients.

“Assemble Web services” on page 62
See this topic for information about packaging your Web services applications for deployment.

“Deploy Web services” on page 65
This topic describes how to install your Web services application into the application server run time.

“Configure Web services” on page 66
See this topic for information about Web services configuration, including configuration scripts and security.

“Web services resources” on page 170
See this topic for additional information about Web services.

Web services overview

A typical Web services scenario is a business application requesting a service from a given URL using Simple Object Access Protocol (SOAP) over a HyperText Transport Protocol (HTTP) transport. The service receives the request, processes it, and returns a response. Examples of a simple Web service include weather reports and stock quotes. The method call is synchronous, that is, it waits until the result is available. Transaction Web services, supporting quotes, business-to-business (B2B) or business-to-client (B2C) operations include airline reservations or purchase orders.

The key components of a Web service are:

- Simple Object Access Protocol (SOAP)
- Web Services Description Language (WSDL)

You can review the Web services client programming model in the Web services for J2EE specification available in the “Web services resources” on page 170 topic. The programming model is similar to the EJB client programming model. There is a remote interface that the client uses to interact with the service. A Java Naming and Directory Interface (JNDI) lookup method can locate the service for a client running in a Web container or client container. The client obtains a stub that implements the remote interface and makes calls to invoke operations on the remote service.

A WebSphere Application Server - Express Java Web service client can exist as one of the following entities:

- As an unmanaged stand-alone Java application.
- As a Java bean or a servlet running in a Web container that is acting as a client.

These topics describe further concepts of Web services:

“Web services architecture”
This topic discusses how Web service providers, brokers, and requesters interact to provide and run Web services.

“Web services operations” on page 3
This topic discusses the life cycle of a Web service, and the roles played by providers, brokers, and requesters in that cycle.

IBM’s AlphaWorks provides tools for creating Web Services Description Language (WSDL) files and Simple Object Access Protocol (SOAP) clients and describes working examples. For more information, see http://www.alphaworks.ibm.com/tech/webservicestoolkit

.

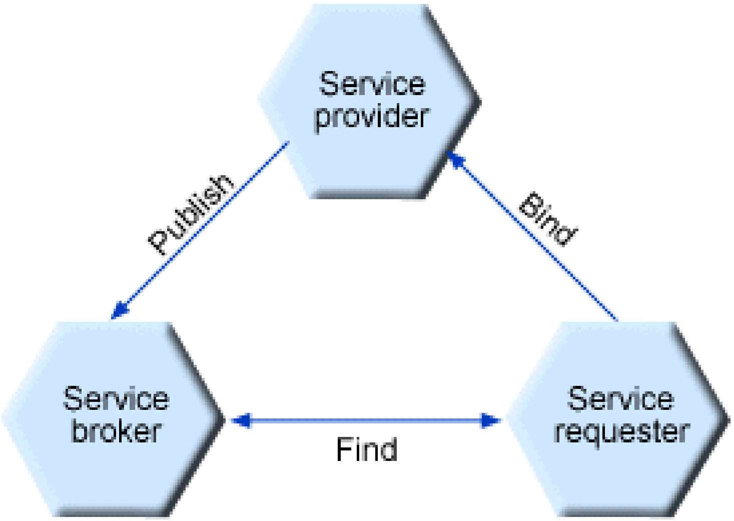
Web services architecture

The Web services architecture includes three roles:

- **Service provider**
  Web service providers create components, then publish them to a repository. On the WebSphere Application Server - Express platform, these components include:
  - Java beans
  - DB2 Universal Database stored procedures
  - Server-side scripts that implement the Bean Scripting Framework (BSF)

  Web service providers can also unpublish components (remove them from the repository) when they are no longer needed.
- **Service broker**
  Web service brokers categorize Web services as they are published, and search for them as service requests are received. Web brokers are roughly analagous to Internet search engines, except that they locate components instead of Web pages.
- **Service requester**
  Web service requesters look up, or locate and invoke components as services. They act as the client for published Web services.

This diagram illustrates how client and server roles can interact to provide Web services.



Web services operations

Web services reduce programming complexity because application designers do not have to worry about implementing the services they invoke. Interactions in Web services are bound dynamically at runtime. A service requester describes the features of the required service and uses the service broker to find an appropriate service.

A Web services component has this life cycle:

1. **Creation**
   The Web services provider creates the service component by defining its interfaces and invocation methods. WebSphere Application Server - Express supports Java beans, DB2 Universal Database stored procedures, and Bean Scripting Framework (BSF) scripts.
2. **Publication**
   The Web services provider publishes the service component to a repository. The Web services broker categorizes the new Web service within its listing.
3. **Location**
   The Web services requester looks up, or locates, a Web service component through the service broker.
4. **Invocation**
   Once the service requester locates the service component, it invokes and implements it.
5. **Unpublication**
   When the Web service provider decides that a Web service should no longer be available, it removes, or unpublishes, the Web service from the repository. The service broker likewise removes the service component from its listing.

Migrate Web services

If you have used the Apache SOAP support to create Web services client applications in WebSphere Application Server - Express Version 5.0, you may need to migrate your applications files for your applications. The following table summarizes the Web services supported by WebSphere Application Server.

WebSphere Application Server - Express Version	Web services supported
5.0 and 5.0.1	SOAP 2.3
5.0.2 or later	J2EE (JSR 109)

For more information on migrating your Web services, see these topics:

"Migrate Apache SOAP Web services to Web services for J2EE" (Version 5.0.2 or later)
This topic describes how to migrate your Version 5.0 and 5.0.1 Web services to the J2EE standards supported in Version 5.0.2.

"Migrate applications developed with Version 5 Web services technology preview" on page 6 (Version 5.0.2)
This topic describes how to migrate your WSDL application developed with the WebSphere Application Server Version 5 Web services technology preview to WebSphere Application Server Version 5.0.2

Notes:

- It is recommended that new Web services be developed using the J2EE standards. For more information, see "Develop Web services" on page 6.
- Security cannot be directly migrated from SOAP 2.3 to the J2EE standards. After you have migrated your Web services to the J2EE standard, see "Configure Web services security (Version 5.0.2 or later)" on page 77 to secure your Web service.

Migrate Apache SOAP Web services to Web services for J2EE

Note: This page applies to WebSphere Application Server - Express Version 5.0.2 and later only.

If you have used the Apache SOAP support to create Web services client applications in WebSphere Application Server - Express Version 5.0 and want use Web services for J2EE, also known as JSR 109, you need to migrate your Version 5.0 client applications.

To migrate these client applications to the JSR 109 Web services standards:

1. **Plan your migration strategy.**
   There are two ways you can port an Apache SOAP client to Java API for XML-based RPC (JAX-RPC) Web services client:
   - If you have, or can create, a Web Services Description Language (WSDL) document for the service, consider using the **WSDL2Java** command tool to generate bindings for the Web service. It is more work to adapt an Apache SOAP client to use the generated JAX-RPC bindings, but the resulting client code is more robust and easier to maintain. To follow this path, see "Develop a Web services client" on page 11.
   - If you do not have a WSDL document for the service, do not expect the service to change, and you want to port the Apache SOAP client with a minimal work, you can convert the code to use the JAX-RPC dynamic invocation interface (DII), which is similar to the Apache SOAP APIs. The DII APIs do not use WSDL or generated bindings.

   You should be aware that since JAX-RPC does not specify a framework for user-written serializers, the JAX-RPC does not support the use of custom serializers. If your application cannot conform to the default mapping between Java, WSDL, and XML supported by WebSphere Application Server - Express, you should not attempt to migrate the application.

   The remainder of this topic assumes that you have decided to use the JAX-RPC DII APIs.
2. **Convert the client application from Apache SOAP to JAX-RPC DII**
   The Apache SOAP API and JAX-RPC DII API structures are similar. You can instantiate and configure a call object, set up the parameters, invoke the operation, and process the result in both.

   In JAX-RPC, you can create a generic instance of a Service object with `javax.xml.rpc.Service service = ServiceFactory.newInstance().createService(new QName(""));`

a. **Create the call object.**
In Apache SOAP, an instance of the call object is created by `org.apache.soap.rpc.Call call = new org.apache.soap.rpc.Call()`

In JAX-RPC, an instance of the call object is created by `java.xml.rpc.Call call = service.createCall();`

b. **Set the endpoint URI.**
In Apache SOAP, the target URI for the operation is passed as a parameter to `call.invoke`: `call.invoke("http://...", "");`

In JAX-RPC, the setTargetEndpointAddress() method is used as a parameter to `parametcall.setTargetEndpointAddress("http://...");`

Apache SOAP has a setTargetObjectURI() method on the call object that contains routing information for the request. JAX-RPC has no equivalent method. The information in the targetObjectURI is included in the targetEndpoint URI for JAX-RPC.

c. **Set the operation name.**
In Apache SOAP, the operation name is configured on the call object by `call.setMethodName("opName");`

The setOperationName method, which accepts a `QName` instead of a `String` parameter, is used in JAX-RPC as follows:

```
call.setOperationName(new javax.xml.namespace.Qname("namespace",
                      "opName"));
```

d. **Set the encoding style.**
In Apache SOAP, the encoding style is configured on the call object by `call.setEncodingStyleURI(org.apache.soap.Constants.NS_URI_SOAP_ENC);`

In JAX-RPC, the encoding style is set by a property of the call object `call.setProperty(javax.xml.rpc.Call.ENCODINGSTYLE_URI_PROPERTY, "http://schemas.xmlsoap.org/soap/encoding/");`

e. **Declare the parameters and set the parameter values.**
Apache SOAP parameter types and values are described by parameter instances, which are collected into a Vector and set on the call object before the call, for example:

```
Vector params = new Vector ();
params.addElement (new org.apache.soap.rpc.Parameter(name,
                 type, value, encodingURI));
// repeat for additional parameters... call.setParams (params);
```

For JAX-RPC, the call object is configured with parameter names and types without providing their values, for example:

```
call.addParameter(name, xmlType, mode);
// repeat for additional parameters call.setReturnType(type);
```

where

- *name* (type `java.lang.String`) is the name of the parameter
- *xmlType* (type `javax.xml.namespace.QName`) is the XML type of the parameter
- *mode* (type `javax.xml.rpc.ParameterMode`) the mode of the parameter, for example: IN, OUT, or INOUT

f. **Make the call.**
In Apache SOAP, the operation is invoked on the call object by `org.apache.soap.Response resp = call.invoke(endpointURI, "");`

In JAX-RPC, the parameter values are collected into an array and passed to `call.invoke` as follows:

`Object resp = call.invoke(new Object[] {parm1, parm2,...});`

g. **Check for faults.**
In Apache SOAP, you can check for a SOAP fault on the invocation by checking the Response:

```
if resp.generatedFault then {
  org.apache.soap.Fault f = resp.getFault;
  f.getFaultCode();
  f.getFaultString(); }
```

A `java.rmi.RemoteException` is thrown in JAX-RPC if a SOAP fault occurs on the invocation.

```
try
  ... call.invoke(...)
catch (java.rmi.RemoteException) ...
```

h. **Retrieve the result.**
In Apache SOAP, if the invocation was successful and returns a result, it can be retrieved from the Response object:

```
Parameter result = resp.getReturnValue(); return result.getValue();
```

In JAX-RPC, the result of the invocation is the returned object when no exception is thrown:

```
Object result = call.invoke(...);
  ...
return result;
```

Note: Examples may be wrapped for display purposes.

Migrate applications developed with Version 5 Web services technology preview

This section describes how to migrate from the Web services technology preview to WebSphere Application Server - Express Version 5.0.2.

You can normally reuse the Web Services Description Language (WSDL) file. If you created the WSDL file using the **Java2WSDL** command in the Web services technology preview and you encounter problems, regenerate the file. You must regenerate all Java classes generated by the WSDL2Java command in the technology preview. These classes fall into two categories, the development classes and the deployment classes. The development classes include the enterprise beans and exception classes generated by the **WSDL2Java** command. These classes are regenerated by running the **wsdl2java -role develop-client** (or **-role develop-server**) command. The deployment classes include `*Stub.java, *Locator.java, *_Ser.java, *_Deser.java, and *_Helper.java` classes. After you verify that no deployment class source code is included in the application EAR file, you can regenerate the deployment classes either by using the **wsdeploy** command, or by selecting **Deploy Web Services** on the application installation panels of the administrative console. Classes generated by the **WSDL2Java** command are identified by a comment at the top of the source file. You can reuse the original Web service client code and server implementation code. If you have any JAX-RPC handlers, rewrite the code to conform with SAAJ 1.1. For more information, see http://java.sun.com/xml/saaj/

. You can reuse the original webservices.xml and webservicesclient.xml deployment descriptors. Any information added to the original ibm-webservices-bnd.xml and ibm-webservicesclient-bnd.xml files need to be migrated to the new ibm-webservices-bnd.xmi and ibm-webservicesclient-bnd.xmi files.

In the Web services technology preview, all application exceptions extended IBM WebServicesFault. Now all application exceptions extend java.lang.Exception to comply with the J2EE specification.

Develop Web services

WebSphere Application Server uses Web services standards developed for the Java language under the Java Community Process (JCP). These standards include the Java API for XML-based remote procedure call (JAX-RPC (JSR-101)) and Web services for J2EE (JSR-109). These specifications are new in WebSphere Application Server Version 5.0.2. If you have a previous version of WebSphere Application Server, see "Develop and manage Simple Object Access Protocol (SOAP)" on page 35 for information about developing Web services.

The JAX-RPC standard covers the programming model and bindings for using Web Services Description Language (WSDL) for Web services in the Java language. The Web services standard for J2EE covers the use of JAX-RPC in a J2EE environment, as well as the deployment of Web services implementations in a J2EE server. Both standards are part of the J2EE 1.4 release.

For more information on JAX-RPC, JSR-109, tutorials and other Web services and J2EE information, see "Web services resources" on page 170.

You can also use the WebSphere Studio Application Developer Version 5.1 graphical user interface development tools to develop Web services that integrate with WebSphere Application Server.

"Develop a J2EE Web service based on an existing application" (Version 5.0.2 or later)
See this topic for information about converting your existing application into a Web service.

"Develop a J2EE Web service based on an existing WSDL file" on page 10 (Version 5.0.2 or later)
See this topic for information about developing a new Web services application.

"Develop a Web services client" on page 11 (Version 5.0.2 or later)
See this topic for information about developing a client for a Web service.

"Web services development artifacts" on page 12
This topic describes the configuration files and interfaces that are part of a Web services application.

"Map between Java, WSDL, and XML" on page 13
This topic contains reference information about how WebSphere Application Server maps between Java and XML technologies such as XML schema, WSDL, and SOAP.

"UDDI4J" on page 50
This topic describes how to use the UDDI4J to generate and parse messages sent to and received from a UDDI server.

"Enable Web services to use the Web Services Invocation Framework (WSIF)" on page 51
WSIF is a WSDL-oriented Java API that allows you to invoke Web services dynamically, regardless of what format the service is implemented in, or what mechanism is used to access it. This topic describes how to enable your Web services to use WSIF.

"Develop and manage Simple Object Access Protocol (SOAP)" on page 35 (Versions 5.0 and 5.0.1 only)
See this topic for information about developing Web services for WebSphere Application Server Version 5.0 and 5.0.1. Note that the Apache SOAP Web services are deprecated in Versions 5.0.2 and later.

Develop a J2EE Web service based on an existing application

1. Access an existing Java bean Web archive (WAR)file that you want to use as a Web service.
2. "Develop a service endpoint interface."
   The service endpoint interface defines which methods should be made available as a Web service.
3. "Develop a Web Services Description Language (WSDL) file" on page 8.
4. "Develop Web service deployment descriptor templates from the WSDL file" on page 8.
5. "Configure the webservices.xml deployment descriptor" on page 9.
6. "Configure the ibm-webservices-bnd.xmi deployment descriptor" on page 10.

Develop a service endpoint interface

The service endpoint interface defines the Web services methods. The Web service implementation must implement methods that have the same signature as the methods on the service endpoint interface. There are a number of restrictions on which types to use as parameters and results of service endpoint interface methods. These restrictions are documented in the Java API for XML remote procedure call (JAX-RPC) specification.

If the Web service implementation is a Java bean, develop the service endpoint interface from the bean or an interface the bean implements.

To develop a service endpoint interface, follow these steps:

1. Create a Java interface that contains the methods that you want to include in the service endpoint interface.
   The interface should extend the java.rmi.Remote interface. Each method throws the java.rmi.RemoteException exception. If you start with an existing Java interface, remove any methods that do not conform to JAX-RPC.
2. Compile the interface.
   You need /QIBM/ProdData/WebASE/ASE5/lib/j2ee.jar in your CLASSPATH to compile the interface.

Develop a Web Services Description Language (WSDL) file

You need a Web Services Description Language (WSDL) file to use Web services. You can develop your own WSDL file or get one from a Web service provider. This documentation assumes you are creating your own.

To develop a WSDL file, follow these steps:

1. Run the Java2WSDL seiInterface command.
   - Move the WSDL file to the WEB-INF/wsdl subdirectory if you are using a Java bean.

   A WSDL file named seiInterface.wsdl is created.
2. Edit the generated WSDL file and inspect the part names.
   The WSDL parts have names like arg_0_0. Modify the WSDL file to use the actual names of the Java parameters.
3. (Optional) Use the Java2WSDL command tool to generate the correct part names of WSDL file.
   You can automatically generate and set the correct part names by using the Java2WSDL command tool. Generating and setting the part names is done by providing additional information to the Java2WSDL command in the form of a Java implementation class that implements the same methods as the Service Endpoint Interface and is compiled with debug information on (javac -g). Parameter names are stored in the .class file with the debug information. If your implementation class was compiled with debug on, you can use the Java2WSDL -implClass seiImpl seiInterface command to generate a WSDL file with the proper part names.

Develop Web service deployment descriptor templates from the WSDL file

To develop the deployment descriptor templates from a Web Services Description Language (WSDL) file, you must obtain the Uniform Resource Locator (URL) of the WSDL file to use. If it is a local file, the URL has this format:

```
file:/path/file_name.wsdl
```

where *path* is the directory path that contains the file, and *file_name* is the name of the wsdl file. You can also specify local files using the absolute or relative file system path.

To develop deployment descriptor templates, run the WSDL2Java command at a command prompt.

- For a J2EE Web services application, run this command:

  ```
  WSDL2Java -verbose -role develop-server -container type -genJava No wsdlURL
  ```

  This command generates the server deployment descriptor files in the WEB-INF subdirectory:

  - webservices.xml
  - ibm-webservices-bnd.xmi
- For a Web services client, run this command:

  ```
  WSDL2Java -verbose -role develop-client -container type -genJava No wsdlURL
  ```

  This command generates the client deploment descriptor files in the WEB-INF subdirectory:
  - webservicesclient.xml
  - ibm-webservicesclient-bnd.xmi

In these commands, *type* is `Web` for a Java bean-based implementation, and *wsdlURL* is the URL of the WSDL file. The value that you specify for the -container parameter determines to which subdirectory the templates are generated. When -container Web parameter is specified, all deployment descriptors and the JAX-RPC mapping file are generated into the WEB-INF subdirectory of the output directory.

The Java API for XML-based remote procedure call (JAX-RPC) mapping file is needed for both server and client use, and is generated by default when you run the WSDL2Java command. To generate the deployment descriptors only, and not any Java classes, specify the -genJava No parameter with the WSDL2Java command tool.

If the -verbose option is specified, the command displays a list of all generated files.

Examples

The following example uses a WSDL file named AddressBookJ2WB.wsdl:

Generate the template files:

```
WSDL2Java -verbose -role develop-client -container Web -genJava No META-INF\AddressBookJ2WB.wsdl
```

The deployment descriptor templates are generated into the WEB-INF for client subdirectories as follows:

```
Parsing XML file: META-INF/AddressBookJ2WB.wsdl
Generating: WEB-INF\webservicesclient.xml
Generating: WEB-INF\ibm-webservicesclient-bnd.xmi
Generating: WEB-INF\AddressBookJ2WB_mapping.xml
Generating: META-INF\webservices.xml
Generating: META-INF\ibm-webservices-bnd.xmi
Generating: META-INF\AddressBookJ2WB_mapping.xml
```

Configure the webservices.xml deployment descriptor

To configure the webservices.xml deployment descriptor, follow these steps:

1. Open the webservices.xml deployment descriptor in a text editor.
2. Specify appropriate values for the file elements.
   Most of the file elements contain default values. Elements that do not have default values are flagged with “??” in the XML file. Set values for these elements:
   - The <webservice-description-name> element
     This name must be unique within the XML file if there are multiple webservice-description elements. This name is used as part of the directory structure for naming published WSDL files when publishing files during deployment.
   - The <wsdl-file> element
     Change the tag element from the full path within the Web archive (WAR) file to the WSDL file defining the Web service being implemented. By convention, the WSDL file is placed in the WEB-INF/wsdl subdirectory for Web modules.
   - The <jaxrpc-mapping-file> element
     Set this element to the full path within the WAR file to the generated Java API for XML-based remote procedure call (JAX-RPC) mapping file. By convention, the mapping file is placed in the WEB-INF subdirectory for Web modules.

- The <port-component-name> element
  Change the element to a string that uniquely identifies the port within the module.
- The <wsdl-port> element
  This identifies the port in the WSDL file that corresponds to this deployment descriptor port. The <namespaceURI> subelement indicates the namespace portion of the WSDL port name. Set this element to the namespace of the matching port in the WSDL file identified by the <wsdl-file> element. This namespace is typically the target namespace in the WSDL file. The <localpart> subelement indicates the local portion of the WSDL port name. Set this element to the name of the matching WSDL port.
- The <service-endpoint-interface> element
  Set this element to the class name, including package, of the Java interface that is the Service Endpoint Interface for the Web service.
- The <service-impl-bean> tag (if the Web service implementation is a Java bean in a Web module)
  Set the <servlet-link> element to reference the bean that implements the methods of the Service Endpoint Interface.

Configure the ibm-webservices-bnd.xmi deployment descriptor

To configure the ibm-webservices-bnd.xmi deployment descriptor, follow these steps:

1. Open the ibm-webservices-bnd.xmi deployment descriptor in a text editor.
2. For the <wsdescBindings> element, specify these attributes:
   - Set the wsDescNameLink attribute to the value of the <webservice-description-name> element in the webservices.xml deployment descriptor.
   - Set the pcNameLink attribute to the value of the <port-component-name> element of the corresponding port in the webservices.xml deployment descriptor.

Develop a J2EE Web service based on an existing WSDL file

1. “Develop implementation templates, deployment descriptor templates, and bindings from a WSDL file.”
2. “Complete the Java bean or enterprise bean implementation.”
3. “Configure the webservices.xml deployment descriptor” on page 9.
4. “Configure the ibm-webservices-bnd.xmi deployment descriptor.”

Develop implementation templates, deployment descriptor templates, and bindings from a WSDL file

To develop the deployment descriptor templates from a Web Services Description Language (WSDL) file, you must obtain the Uniform Resource Locator (URL) of the WSDL file to use. If it is a local file, the URL has this format:

```
file:/path/file_name.wsdl
```

where *path* is the directory path that contains the file, and *file_name* is the name of the wsdl file. You can also specify local files using the absolute or relative file system path.

To generate implementation templates, bindings, and deployment descriptors, specify the `-role develop-server` parameter and the `-container` parameter when you run the WSDL2Java command:

```
WSDL2Java -verbose -role develop-server -container type wsdlURL
```

where *type* is `Web` for a Java bean-based implementation, and *wsdlURL* is the URL of the WSDL file.

If you specify the verbose parameter, the command displays a list of all generated files

Complete the Java bean or enterprise bean implementation

To complete the implementation of a Java bean, follow these steps:

1. Inspect the remote interface template, *portType*_RI.java, where *portType* is the name of the <wsdl:portType> element in the WSDL file. If necessary, modify the template.
2. Inspect the home interface template, portTypeHome.java. If necessary, modify the template.
3. Edit the implementation template, *binding*Impl.java, where *binding* is the name of the <wsdl:binding> element in the WSDL file.
   a. Complete the implementation of the methods in the template.
   b. (Optional) Make changes if necessary.
   c. (Optional) Change the class name if the binding name is not acceptable.
4. Compile the Java classes.
5. Assemble a Web archive (WAR) file. See Assemble your application for instructions on assembling applications. Include all of the classes that the WSDL2Java command generates.

Develop a Web services client

To set up a development environment for Web services clients, see "Set up a Web services client development environment."

To create the client code and artifacts that enable the application client to access a Web service, follow these steps:

1. Locate the Web Services Description Language (WSDL) file that defines the Web service to access.
2. "Develop implementation templates, deployment descriptor templates, and bindings from a WSDL file" on page 10.
   Follow the steps in this task, but use the -role client -container client option with the WSDL2Java command tool. The client-side bindings and artifacts are generated.
3. Complete the client implementation.
4. (Optional) "Configure the webservicesclient.xml deployment descriptor."
   Complete this step if you are developing a managed client that runs in the J2EE client container.
5. (Optional) "Configure the ibm-webservices-bnd.xmi deployment descriptor" on page 10.
   Complete this step if you are deploying a managed client that runs in the J2EE client container and you want to override the default client settings. See "Web services assembly properties" on page 62 for more information about the ibm-webservicesclient-bnd.xmi deployment descriptor.

If you are developing a managed client that runs in the J2EE client container, "Assemble a Web services client" on page 65.

Set up a Web services client development environment

WebSphere Application Server - Express provides several scripts that you can use to develop Web services clients and implementations. To use the scripts, you must set up the Java environment that Web services J2SE clients use and set the classpath variable for Web services clients.

To set up a development environment for Web services client applications, follow these steps:

1. On the CL command line, run the STRQSH (Start Qshell) command.
2. On the Qshell command line, use the cd command to change to the directory that contains the script. For example, if you are using WebSphere Application Server, run this command:
   ```
   cd /QIBM/ProdData/WebASE/ASE5/bin
   ```
3. Run the setupWebServiceClientEnv script. There are no parameters to specify for this script. For additional information on the script, see "The setupWebServiceClientEnv script" on page 73.

Configure the webservicesclient.xml deployment descriptor

After you generate the webservicesclient.xml file, you may want to edit some of the values in the file. If the default values that the WSDL2Java generates are acceptable, you do not need to change any of them.

To configure the webservicesclient.xml deployment descriptor, follow these steps:

1. Open the webservicesclient.xml deployment descriptor in a text editor.
2. For the <description> element in the <service-ref> element, specify a descriptive name for the service that the client accesses.

   Note: If your client uses more than one Web service, you must specify values in the <service-ref> element for each Web service.
3. For the <service-ref-name> element in the <service-ref> element, specify the name that the Java Naming Directory Interface (JNDI) uses to locate the service. The JNDI lookup string for this service has this format:

   ```
   java:comp/env/service-ref-name
   ```

   where *service-ref-name* is . By convention, the value that you specify for *service-ref-name* should begin with `service/`.
4. For the <service-interface> element in the <service-ref> element, specify the class name, including package, of the generated Java interface that is the service interface for this Web service.
5. For the <wsdl-file> element in the <service-ref> element, specify the WSDL file name used by this client relative to the root of the module.
6. For the <jaxrpc-mapping-file> element in the <service-ref> element, specify the file name of the generated Java mapping file relative to the root of the module.
7. For the <service-endpoint-interface> element in the <port-component-ref> element, specify the class name, including package, of the Java interface that is the service endpoint interface for this Web service.

Web services development artifacts

Development artifacts enable a Java bean module to be a Web service. To create a Web service from a Java bean module, these files are added to the Web archive (WAR) modules when you assemble the application:

- **Web Services Description Language (WSDL) XML**
  The WSDL XML file describes the Web service.
- **Service Endpoint Interface**
  A Service Endpoint Interface is the Java interface that corresponds to the Web service port type that is implemented. The Service Endpoint Interface is defined by the WSDL 1.1 W3C Note.
- **webservices.xml**
  The webservices.xml file contains the J2EE Web service deployment descriptor that specifies how the Web service is implemented. The webservices.xml file is defined in the Web services for J2EE specification.

  .
- **ibm-webservices-bnd.xmi**
  This file contains WebSphere product-specific deployment information and is defined in "Web services assembly properties" on page 62.
- **Java API for XML-based remote procedure call (JAX-RPC) mapping file**
  The JAX-RPC mapping deployment descriptor specifies how Java elements are mapped to and from WSDL file elements. The JAX-RPC mapping file is defined in "Web services assembly properties" on page 62.

The following files are added to the application client module at assembly, allowing a J2EE application client to access Web services:

- **WSDL**
  The WSDL file is provided by the Web service implementer.

- **Java interfaces for the Web service**
  The Java interfaces are generated from the WSDL file as specified by the JAX-RPC mapping file. These bindings are the Service Endpoint Interface based on the WSDL port type, or the service interface, which is based on the WSDL service.
- **webservicesclient.xml**
  The webservicesclient.xml file is the client-side deployment descriptor. This file describes the services that are accessed. The file is defined in the Web services for J2EE specification.

- **ibm-webservicesclient-bnd.xmi**
  This file contains WebSphere product-specific deployment information such as security information. The ibm-webservicesclient-bnd.xmi file is defined in “Web services assembly properties” on page 62.
- **Other JAX-RPC binding files**
  The WSDL2Java command-line tool generates additional JAX-RPC binding files based on the WSDL file. These additional files support the client application in mapping Simple Object Access Protocol (SOAP) to Java.

Map between Java, WSDL, and XML

This topic contains the mappings between Java and XML technologies, including XML Schema, Web Services Description Language (WSDL) and Simple Object Access Protocol (SOAP), supported by WebSphere Application Server - Express. Most of these mappings are specified by Java API for XML-based remote procedure call (JAX-RPC). Some mappings optional or unspecified in JAX-RPC are also supported.

Notational conventions

The following table specifies the namespace prefixes and corresponding namespaces used.

Namespace prefix	Namespace
xsd	http://www.w3.org/2001/XMLSchema
xsi	http://www.w3.org/2001/XMLSchema-instance
soapenc	http://schemas.xmlsoap.org/soap/encoding/
wsdl	http://schemas.xmlsoap.org/wsdl/
wsdlsoap	http://schemas.xmlsoap.org/wsdl/soap/
ns	user defined namespace
apache	http://xml.apache.org/xml-soap
wasws	http://websphere.ibm.com/webservices/

Detailed mapping information

See these sections for information on the supported mappings:
- Java-to-WSDL mapping (page 13)
- WSDL-to-Java mapping (page 21)
- Mapping between WSDL and SOAP messages (page 32)

Java-to-WSDL mapping

This section summarizes the Java-to-WSDL mapping rules. The Java-to-WSDL mapping rules are used by the Java2WSDL command tool for bottom-up processing. In bottom-up processing, an existing Java service implementation is used to create a WSDL file defining the Web service. The generated WSDL file can require additional manual editing for the following reasons:

- Not all Java classes and constructs have mappings to WSDL. For example, Java classes that do not comply with the Java bean specification rules might not map to a WSDL construct.
- Some Java classes and constructs have multiple mappings to WSDL. For example, a java.lang.String class can be mapped to either an xsd:string or soapenc:string. The Java2WSDL command chooses one of these mappings, but the WSDL file must be edited if a different mapping is desired.
- There are multiple ways to generate WSDL constructs. For example, the part element in the wsdl:message can be generated with a type or element attribute. The Java2WSDL command makes an informed choice based on the settings of the -style and -use options.
- The WSDL file describes the instance data elements sent in the SOAP message. If you want to modify the names or format used in the message, the WSDL file must be edited.
- The WSDL file requires editing if header or attachment support is desired.
- The WSDL file requires editing if a multipart WSDL, one using wsdl:import, is desired.

For simple services, the generated WSDL file is sufficient. For complicated services, the generated WSDL file is a good starting point.

General issues

- **Package to namespace mapping**
  The JAX-RPC does not specify the default mapping of Java package names to XML namespaces. The JAX-RPC does specify that each Java package must map to a single XML namespace, and likewise. A default mapping algorithm is provided that constructs the namespace by reversing the names of the Java package and adding an http:// prefix. For example, a package named, com.ibm.webservice, is mapped to the namespace http://webservice.ibm.com.

  The default mapping between XML namespaces and Java package names can be overridden using the -NStoPkg and -PkgtoNS options of the WSDL2Java and Java2WSDL commands.
- **Identifier mapping**
  Java identifiers are mapped directly to WSDL file and XML identifiers.

  Java bean property names are mapped to the WSDL file and XML identifiers. For example, a Java bean, with getInfo and setInfo methods, maps to an XML construct with the name, info.

  The Service Endpoint Interface method parameter names, if available, are mapped directly to the XML identifiers. See the WSDL2Java command -implClass option for more details.
- **WSDL construction summary**
  The following table summarizes the mapping from a Java construct to the related WSDL and XML construct.

Java construct	WSDL and XML construct
Service Endpoint Interface	wsdl:portType
Method	wsdl:operation
Parameters	wsdl:input, wsdl:message, wsdl:part (1)
Return	wsdl:output, wsdl:message, wsdl:part (1)
Throws	wsdl:fault, wsdl:message, wsdl:part (1)
Primitive types	xsd and soapenc simple types
Java beans	xsd:complexType
Java bean properties	Nested xsd:elements of xsd:complexType
Arrays	JAX-RPC defined array xsd:complexType
User defined exceptions	xsd:complexType

Note: The generated WSDL file is affected by the -style and -use options. A wsdl:binding that conforms to the generated wsdl:portType is generated. The style and use constructs of the wsdl:binding are determined from the -style and -use options. A wsdl:service containing a port that references the generated wsdl:binding is generated. The names and values of the wsdl:service are controlled by the Java2WSDL command options.

- **Style and use**
  Use the -style and -use options to generate different kinds of WSDL files. The four supported combinations are:
  - -style RPC -use ENCODED
  - -style DOCUMENT -use LITERAL
  - -style RPC -use LITERAL
  - -style DOCUMENT -use LITERAL -wrapped false

  The -use LITERAL option affects the generated WSDL file in the following ways:
  - The soap:body elements in the wsdl:binding are specified as use="literal".
  - The soap:fault elements in the wsdl:binding are specified as use="literal".
  - The soap encoded types are not used.
  - The soap encoded array style is not used. The maxOccurs attribute is used to indicate arrays.

  The -use ENCODED option affects the generated WSDL file in the following ways:
  - The soap:body elements in the wsdl:binding are specified as use="encoded" and the encodingStyle is set.
  - The soap:fault elements in the wsdl:binding are specified as use="encoded" and the encodingStyle is set.
  - The -style RPC option affects the generated WSDL file in the following ways:
  - The wsdl:part elements use the type attribute to reference XML types.
  - The wsdl:binding is specified as style="rpc".

  The -style DOCUMENT -wrapped false option affects the generated WSDL file in the following ways:
  - The wsdl:part elements use the type attribute to reference simple types. The element attribute is used to reference the root xsd:elements for everything that is not a simple type.
  - The wsdl:binding is specified as style="document".

  The -style DOCUMENT -wrapped true option generates a WSDL file that conforms to the .NET WSDL file format:
  - A request xsd:element is generated for each method in the Service Endpoint Interface as follows:
    - The name of the xsd:element is the same as the name of the wsdl:operation.
    - The xsd:element contains an xsd:sequence that contains xsd:elements defining each parameter.
    - The request wsdl:message references the wrapper xsd:element using a single part:
    - The name of the part is parameters.
    - The element attribute is used to reference the wrapper xsd:element.
  - A response xsd:element is generated for each method in the Service Endpoint Interface as follows:
    - The name of the xsd:element is the same as the name of the wsdl:operation appended with Response
    - The xsd:element contains an xsd:sequence that contains xsd:elements defining the return value.
    - The request wsdl:message references this wrapper xsd:element using a single part.
    - The element attribute is used to reference the wrapper xsd:element.
    - The wsdl:binding is specified as style="document".

Mapping of standard XML types from Java types

Some Java types map directly to standard XML types. These types do not require additional XML definitions in the wsdl:types section.

- **JAX-RPC Java primitive type mapping**
  The following table describes the mapping from the Java primitive and standard types to XML standard types. For more information see the JAX-RPC specification.

Java type	**XML type**
int	xsd:int
long	xsd:long
short	xsd:short
float	xsd:float
double	xsd:double
boolean	xsd:boolean
byte	xsd:byte
byte[]	xsd:base64Binary **Note:** The default mapping for byte[] is xsd:base64Binary. The data in byte[] is passed over the wire as a text string encoded in the base64 format. An alternative format is xsd:hexBinary. To use the xsd:hexBinary format: • Edit the WSDL file and change xsd:base64Binary to xsd:hexBinary, or • Change your implementation to use the specialized com.ibm.ws.webservices.engine.types.HexBinary class.
java.lang.String	xsd:string
java.math.BigInteger	xsd:integer
java.math.BigDecimal	xsd:decimal
java.util.Calendar	xsd:dateTime
java.util.Date **Note:** This mapping is not covered by the JAX-RPC.	xsd:date
java.lang.Boolean	xsd:boolean xsi:nillable=true
java.lang.Float	xsd:float xsi:nillable=true
java.lang.Double	xsd:double xsi:nillable=true
java.lang.Integer	xsd:int xsi:nillable=true
java.lang.Short	xsd:short xsi:nillable=true
java.lang.Byte	xsd:byte xsi:nillable=true

Note: The java.lang wrapper classes in the last six lines of the table map to the same XML construct as the corresponding Java primitive type. In addition, the xsi:nillable attribute is generated to indicate that such elements can be null.

- **Additional Java class mappings**
  In addition to the standard JAX-RPC mapping, the following classes are mapped directly to XML types:

Java type	**XML type**
com.ibm.ws.webservices.engine.types.HexBinary	xsd:hexBinary
javax.xml.namespace.QName	xsd:qname
com.ibm.ws.webservices.engine.types.Token	xsd:token

Java type	**XML type**
com.ibm.ws.webservices.engine.types.NormalizedString	xsd:normalizedString
com.ibm.ws.webservices.engine.types.Name	xsd:Name
com.ibm.ws.webservices.engine.types.NCName	xsd:NCName
com.ibm.ws.webservices.engine.types.NMToken	xsd:NMTOKEN
com.ibm.ws.webservices.engine.types.URI	xsd:anyURI
com.ibm.ws.webservices.engine.types.UnsignedLong	xsd:unsignedLong
com.ibm.ws.webservices.engine.types.UnsignedInt	xsd:unsignedInt
com.ibm.ws.webservices.engine.types.UnsignedByte	xsd:unsignedByte
com.ibm.ws.webservices.engine.types.NonNegativeInteger	xsd:nonNegativeInteger
com.ibm.ws.webservices.engine.types.PositiveInteger	xsd:positiveInteger
com.ibm.ws.webservices.engine.types.NonPositiveInteger	xsd:nonPositiveInteger
com.ibm.ws.webservices.engine.types.Time	xsd:time
com.ibm.ws.webservices.engine.types.YearMonth	xsd:gYearMonth
com.ibm.ws.webservices.engine.types.Year	xsd:gYear
com.ibm.ws.webservices.engine.types.Month	xsd:gMonth
com.ibm.ws.webservices.engine.types.Day	xsd:gDay
com.ibm.ws.webservices.engine.types.MonthDay	xsd:gMonthDay
com.ibm.ws.webservices.engine.types.Duration	xsd:duration
java.util.Map	apache:Map **Note:** Any classes that implement java.util.Map are also mapped to apache:Map.
java.util.Collection	soapenc:Array **Note:** Each Java array, except byte[], and every class that implements java.util.Collection is mapped to a JAX-RPC defined soapenc:Array type.
org.w3c.dom.Element	apache:Element
java.util.Vector	apache:Vector
java.awt.Image **Note:** Used for attachment support.	apache:Image
javax.mail.internet.MimeMultiPart **Note:** Used for attachment support.	apache:Multipart
javax.xml.transform.Source **Note:** Used for attachment support.	apache:Source
javax.activation.DataHandler **Note:** Used for attachment support.	apache:DataHandler

Generation of wsdl:types

Java types that cannot be mapped directly to standard XML types are generated in the wsdl:types section.

- **Java arrays**
  Java arrays for the -use ENCODED option, with the exception of byte[], are generated using the following format. See the JAX-RPC specification for more details. Alternative mappings can be found in Table 18.1 of the JAX-RPC specification.

  Java:

  ```
  Item[]
  ```

  Mapped to:

```
<xsd:complexType name="ArrayOfItem">
  <xsd:complexContent>
    <xsd:restriction base="soapenc:Array">
      <xsd:attribute ref="soapenc:arrayType" wsdl:arrayType="ns:Item" />
    </xsd:restriction>
  </xsd:complexContent
</xsd:complexType>
```

- **JAX-RPC value type and bean mapping**
  A Java class that matches the Java value type or bean pattern is mapped to an xsd:complexType. In order for a Java class to be mapped to XML, follow these conditions:
  - The class must have a public default constructor.
  - The class must not implement, directly or indirectly, java.rmi.Remote.
  - Public, nonstatic, nonfinal, nontransient fields are mapped. The class can contain other fields and methods, but they are not mapped.
  - If the class follows the Java bean pattern and has public getter and setter methods, the property is mapped.

  Additional mapping rules can be found in the JAX-RPC specification. This example indicates the mapping for sample base and derived Java classes:

  Java:

```
public abstract class Base {
     public Base() {}
     public int a;                        // mapped
     private int b;                       // mapped via setter/getter
     private int c;                       // not mapped
     private int[] d;                     // mapped via indexed setter/getter

     public int getB() { return b;}       // map property b
     public void setB(int b) {this.b = b;}

     public int[] getD() { return d;}     // map indexed property d
     public void setD(int[] d) {this.d = d;}
     public int getD(int index) { return d[index];}
     public void setB(int index, int value) {this.d[index] = value;}

     public void someMethod() {...}       // not mapped
  }

  public class Derived extends Base {
     public int x;                        // mapped
     private int y;                       // not mapped
  }
```

  Mapped to:

```
<xsd:complexType name="Base" abstract="true">
  <xsd:sequence>
    <xsd:element name="a" type="xsd:int" />
    <xsd:element name="b" type="xsd:int" />
    <xsd:element name="d" minOccurs="0" maxOccurs="unbounded" type="xsd:int"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="Derived">
  <xsd:complexContent>
```

```
    <xsd:extension base="ns:Base">
      <xsd:sequence>
        <xsd:element name="x" type="xsd:int" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Inheritance and abstract classes

The example contains two optional JAX-RPC features that are supported by WebSphere Application Server:

- An abstract class is mapped to an xsd:complexType with abstract="true".
- An indexed bean property (see the methods for d in Base) are mapped to a nested element specified with maxOccurs="unbounded". This format is similar to an XML array, but the SOAP message is different. An XML array defines an element for the array and nested elements for each item in the array. An element defined with maxOccurs indicates a series of items without the surrounding array wrapper element. Both formats are popular.

- **JAX-RPC enumeration class mapping**
  Section 4.2.4 of the JAX-RPC specification defines the mapping from an XML enumeration to a Java class. Though not specifically required by the JAX-RPC, the Java2WSDL command performs the reverse mapping. If you have a class that has the same format as a JAX-RPC enumeration class, it is mapped to an XML enumeration.
- **Holder classes**
  The JAX-RPC specification defines Holder classes in section 4.3.5. A Holder class is used to support in and out parameter passing. Every Holder class implements thejavax.xml.rpc.holders.Holder interface. The Java2WSDL command maps Holder classes to the same XML type as the held type. In addition, references to Holder classes affect the generation of wsdl:messages.
- **Exception classes**
  If a class extends the exception, java.lang.Exception, it is mapped to an xsd:complexType similar to the Java bean mapping. The getter methods of the exception are mapped as nested xsd:elements of the xsd:complexType. See section 5.5.5 of the JAX-RPC specification for more details.

  Note: You need to generate implementation specific exception classes by invoking the WSDL2Java command on the resulting WSDL file.
- **Unsupported classes**
  If a class cannot be mapped to an XML type, the Java2WSDL command issues a message and an xsd:anyType reference is generated in the WSDL file. In these situations, modify the Web service implementation to use the JAX-RPC compliant classes.
- **Generation of root elements**
  If the Java2WSDL command generates an xsd:complexType or xsd:simpleType for a parameter reference, a corresponding xsd:element is also generated. The xsd:element has the same name as the xsd:complexType/xsd:simpleType and uses the type attribute to reference the xsd:complexType/xsd:simpleType. The wsdl:message part can use the element attribute or the type attribute to reference the xsd:element or type. This choice is determined by the -style and -use options.

Generation from the interface or implementation class

The class passed to the Java2WSDL command represents the interface of the wsdl:service. The wsdl:portType and wsdl:message elements generate from this interface or implementation class.

- **Generation of the wsdl:portType**
  The name of the wsdl:portType is the name of the class unless overridden by the -portTypeName option.
- **Generation of wsdl:operation**
  A wsdl:operation generates for each public method in the interface that throws the exception, java.rmi.RemoteException.
  - The name of the wsdl:operation is the name of the method.

- The wsdl:operation has a parameterOrder attribute, which defines the order of the parameters in the signature. Specifically, the parameterOrder lists the order of the parts of the request or response wsdl:messages.
- The wsdl:operation has a nested wsdl:input element that references the request wsdl:message using the message attribute.
- The wsdl:operation has a nested wsdl:output element that references the response wsdl:message using the message attribute.
- The wsdl:operation has a nested wsdl:fault element that references the default wsdl:message using the message attribute.

See sections 5.5.4 and 5.5.5 of the JAX-RPC specification for more information.

- **Generation of wsdl:message**
  Generating the wsdl:message is directly related to the -style and -use options. The following is the default mapping (-style RPC -use ENCODED):
  - A wsdl:message is created to represent the request. A wsdl:part representing each parameter is added to the wsdl:message.
    - The wsdl:part has the same name as the parameter.
    - The wsdl:part uses the type attribute to locate the XML type of the parameter.
  - A wsdl:message is created to represent the response. A wsdl:part representing the method return is created.
    - The wsdl:part has the same name as the method with Return appended.

      Note: The name of the part is not specified by the JAX-RPC and is typically not checked by SOAP engines. The wsdl:part has the same name as the parameter.
    - The wsdl:part uses the type attribute to locate the XML type of the parameter.
    - A wsdl:part is created for each parameter that is a Holder.
    - The wsdl:part has the same name as the parameter.
    - A wsdl:message is created to represent the fault if the operation has a wsdl:fault.
    - A wsdl:part representing the fault is created.
    - The wsdl:part has the same name as the exception.
    - The wsdl:part uses the type attribute to locate the complexType representing the exception.

  The same mapping is used as described if you use the -style RPC and -use LITERAL options. It is also valid to use the wsdl:part element attribute instead of the type attribute to reference the XML schema. If you use the -style DOCUMENT -wrapped false and -use LITERAL options, the same mapping is used as described except the wsdl:part element attribute is used to reference the XML schema. If the XML schema is a primitive type, like xsd:string, the type attribute is used to reference the XML type. The -style DOCUMENT, -wrapped true and -use LITERAL options result in completely different mappings for the request and response messages. For example:
  - A request xsd:element is generated for each method in the Service Endpoint Interface.
    - The name of the xsd:element is the same as the name of the wsdl:operation.
    - The xsd:element contains an xsd:sequence that contains xsd:elements defining each parameter.
    - The request wsdl:message references the wrapper xsd:element using a single part.
      - The name of the part is parameters.
      - The element attribute is used to reference the wrapper xsd:element.
  - A response xsd:element is generated for each method in the Service Endpoint Interface.
    - The name of the xsd:element is the same as the name of the wsdl:operation appended with Response.
    - The xsd:element contains an xsd:sequence that contains xsd:elements defining the return value.
    - The request wsdl:message references this wrapper xsd:element using a single part.
    - The element attribute is used to reference the wrapper xsd:element.

- **Generation of wsdl:binding**
  Generate a wsdl:binding with a name defined by the Java2WSDL -bindingName command.
  - The wsdlsoap:binding style attribute is set to rpc if you use the -style RPC option; otherwise it is set to document.
  - A wsdl:operation generates for each wsdl:operation defined in the wsdl:portType.
  - Each wsdl:operation has corresponding wsdl:input, wsdl:output and wsdl:fault elements.
  - The wsdl:input, wsdl:output and wsdl:fault elements each contain a wsdlsoap:body element.
  - The wsdlsoap:body use attribute is set to literal or encoded according to the -use argument. Set the encodingStyle attribute to http://schemas.xmlsoap.org/soap/encoding/ when use is encoded.
- **Generation of the wsdl:service**
  Generate a wsdl:service with a name defined by the Java2WSDL -serviceElement command. For example:
  - The wsdl:service contains a port with a name defined by the Java2WSL -servicePortName command.
  - The port references the generated wsdl:binding with the binding attribute.
  - The port contains a wsdlsoap:address element with a
  - The location attribute is set to the value of the Java2WSDL -location command.

WSDL-to-Java mapping

The WSDL2Java command tool uses the following rules to generate Java classes when developing your Web services client and server. In addition, implementation specific Java classes are generated that assist in the serialization and deserialization, and invocation of the Web service.

General issues

- **Mapping of namespace to package**
  The JAX-RPC does not specify the mapping of XML namespaces to Java package names. The JAX-RPC does specify that each Java package map to a single XML namespace, and likewise. A default mapping algorithm omits any protocol from the XML namespace and reverses the names. For example, an XML namespace http://websphere.ibm.com becomes a Java package with the name com.ibm.websphere.

  The default mapping of XML namespace to Java package disregards the context-root. If two namespaces are the same up until the first slash, they map to the same Java package. For example, the XML namespaces http://websphere.ibm.com/foo and http://websphere.ibm.com/bar map to the Java package com.ibm.websphere. The default mapping between XML namespaces and Java package names can be overridden using the -NStoPkg and -PkgtoNS options of WSDL2Java and Java2WSDL commands.
- **Identifier mapping**
  XML names are much richer than Java identifiers. They can include characters that are not permitted in Java identifiers. See section 20 of the JAX-RPC specification for the rules to map an XML name to a Java identifier.

  The mapping rules attempt to follow accepted Java coding conventions. Class names always begin with an uppercase letter. Method names begin with a lowercase letter. The WSDL2Java command generates metadata in the _Helper class so that the values are serialized or deserialized using the XML names specified in the WSDL file.
- **Java construction summary**

WSDL and XML	Java
xsd:complexType (struct) **Note:** The xsd:complexType can also represent a Java exception if referenced by a wsdl:message for a wsdl:fault.	Java Bean Class **Note:** The classes, _Helper, _Ser, and _Deser, generate for each Java bean class. These implementation classes aid serialization and deserialization.
nested xsd:element/xsd:attribute	Java bean property

WSDL and XML	Java
xsd:complexType (array)	Java array
xsd:simpleType (enumeration)	JAX-RPC enumeration class
xsd:complexType (wrapper) The method parameter signature typically is determined by the wsdl:message. However, if the WSDL file is a .NET wrapped style, the method parameter signature is determined by the wrapper xsd:element	Service Endpoint Interface method parameter signature **Note:** If a parameter is out or inout, a Holder class generates.
wsdl:message The method parameter signature typically is determined by the wsdl:message. However, if the WSDL file is a .NET wrapped style, the method parameter signature is determined by the wrapper xsd:element	Service Endpoint Interface method signature **Note:** If a parameter is out or inout, a Holder class generates.
wsdl:portType	Service Endpoint Interface
wsdl:operation	Service Endpoint Interface method
wsdl:binding	Stub **Note:** The Stub and ServiceLocator classes are implementation specific.
wsdl:service	Service Interface and ServiceLocator **Note:** The Stub and ServiceLocator classes are implementation specific.
wsdl:port	Port accessor method in Service Interface

Mapping standard XML types

- **JAX-RPC simple XML types mapping**
  The following mappings are XML types to Java types. For more information about these mappings, see section 4.2.1 of the JAX-RPC specification.

XML type	Java type
xsd:string	java.lang.String
xsd:integer	java.math.BigInteger
xsd:int **Note:** If an element with this type has the xsi:nillable attribute set to true, it is mapped to the Java wrapper class of the primitive type.	int
xsd:long **Note:** If an element with this type has the xsi:nillable attribute set to true, it is mapped to the Java wrapper class of the primitive type.	long
xsd:short **Note:** If an element with this type has the xsi:nillable attribute set to true, it is mapped to the Java wrapper class of the primitive type.	short
xsd:decimal	java.math.BigDecimal

XML type	Java type
xsd:float **Note:** If an element with this type has the xsi:nillable attribute set to true, it is mapped to the Java wrapper class of the primitive type.	float
xsd:double **Note:** If an element with this type has the xsi:nillable attribute set to true, it is mapped to the Java wrapper class of the primitive type.	double
xsd:boolean **Note:** If an element with this type has the xsi:nillable attribute set to true, it is mapped to the Java wrapper class of the primitive type.	boolean
xsd:byte **Note:** If an element with this type has the xsi:nillable attribute set to true, it is mapped to the Java wrapper class of the primitive type.	byte
xsd:dateTime	java.util.Calendar
xsd:date **Note:** This mapping is not supported by the JAX-RPC.	java.util.Date
xsd:base64Binary	byte[]
xsd:hexBinary	byte[]
soapenc:base64	byte[]
soapenc:base64Binary	byte[]
soapenc:string	java.lang.String
soapenc:boolean	java.lang.Boolean
soapenc:float	java.lang.Float
soapenc:double	java.lang.Double
soapenc:decimal	java.math.BigDecimal
soapenc:int	java.lang.Integer
soapenc:integer **Note:** This mapping is not supported by the JAX-RPC.	java.math.BigInteger
soapenc:short	java.lang.Short
soapenc:long **Note:** This mapping is not supported by the JAX-RPC.	java.lang.Long
soapenc:byte	java.lang.Byte

- **JAX-RPC optional simple XML type mapping**
  The WSDL2Java command supports the following JAX-RPC optional simple XML types.

XML type	Java type
xsd:qname	javax.xml.namespace.QName
xsd:time	com.ibm.ws.webservices.engine.types.Time
xsd:gYearMonth	com.ibm.ws.webservices.engine.types.YearMonth

XML type	Java type
xsd:gYear	com.ibm.ws.webservices.engine.types.Year
xsd:gMonth	com.ibm.ws.webservices.engine.types.Month
xsd:gDay	com.ibm.ws.webservices.engine.types.Day
xsd:gMonthDay	com.ibm.ws.webservices.engine.types.MonthDay
xsd:token	com.ibm.ws.webservices.engine.types.Token
xsd:normalizedString	com.ibm.ws.webservices.engine.types.NormalizedString
xsd:unsignedLong	com.ibm.ws.webservices.engine.types.UnsignedLong
xsd:unsignedInt	com.ibm.ws.webservices.engine.types.UnsignedInt
xsd:unsignedShort	com.ibm.ws.webservices.engine.types.UnsignedShort
xsd:unsignedByte	com.ibm.ws.webservices.engine.types.UnsignedByte
xsd:nonNegativeInteger	com.ibm.ws.webservices.engine.types.NonNegativeInteger
xsd:negativeInteger	com.ibm.ws.webservices.engine.types.NegativeInteger
xsd:positiveInteger	com.ibm.ws.webservices.engine.types.PositiveInteger
xsd:nonPositiveInteger	com.ibm.ws.webservices.engine.types.NonPositiveInteger
xsd:Name	com.ibm.ws.webservices.engine.types.Name
xsd:NCName	com.ibm.ws.webservices.engine.types.NCName
xsd:NMTOKEN	com.ibm.ws.webservices.engine.types.NMTOKEN
xsd:duration	com.ibm.ws.webservices.engine.types.Duration
xsd:anyURI	com.ibm.ws.webservices.engine.types.URI

- **JAX-RPC xsd:anyType mapping**
  The WSDL2Java command maps an xsd:anyType to a java.lang.Object. This is an optional feature of the JAX-RPC specification. The xsd:anyType can be used to store any XML type other than the XML primitive type. An xsd:anyType is always serialized along with an xsi:type that specifies the actual type.
- **Additional supported mappings**
  The following mappings are also supported by the WSDL2Java command. These mappings are not defined by the JAX-RPC specification.

XML type	Java type
apache:PlainText **Note:** For MIME attachments.	java.lang.String
apache:Map	java.util.Map
apache:Element	org.w3c.dom.Element
wasws:SOAPElement	com.ibm.ws.webservices.xmlsoap.SOAPElement
apache:Vector	java.util.Vector
apache:Image **Note:** For MIME attachments.	java.awt.Image
apache:Multipart **Note:** For MIME attachments.	javax.mail.internet.MimeMultipart
apache:Source **Note:** For MIME attachments.	javax.xml.transform.Source

XML type	Java type
apache:octetStream **Note:** For MIME attachments.	javax.activation.DataHandler
apache:DataHandler **Note:** For MIME attachments.	javax.activation.DataHandler

Mapping XML defined in the wsdl:types section

The WSDL2Java command generates Java types for the XML schema constructs defined in the wsdl:types section. The XML schema language is broader than the required or optional subset defined by the JAX-RPC specification. The WSDL2Java command supports all required mappings and most optional mappings. In addition, the command supports some XML schema mappings that are outside the JAX-RPC specification. In general, the WSDL2Java command ignores constructs that it does not support. For example, the WSDL2Java command does not support the default attribute. If an xsd:element is defined with the default attribute, the default attribute is ignored. In some cases it maps unsupported constructs to wasws:SOAPElement.

- **Mapping of xsd:complexType to Java bean**
  The most common mapping is from an xsd:complexType to a Java bean class.
  - **Standard Java bean mapping**
    The standard Java bean mapping is defined in section 4.2.3 of the JAX-RPC specification The xsd:complexType defines the type. The nested xsd:elements within the xsd:sequence or xsd:all groups are mapped to Java bean properties. For example:

    XML:
    ```
    <xsd:complexType name="Sample">
      <xsd:sequence>
        <xsd:element name="a" type="xsd:string"/>
        <xsd:element name="b" maxOccurs="unbounded" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>
    ```
    Java:
    ```
    public class Sample {
         // ..
         public Sample() {}

         // Bean Property a
         public String getA()            {...}
         public void   setA(String value) {...}

         // Indexed Bean Property b
         public String[] getB()          {...}
         public String   getB(int index)  {...}
         public void     setB(String[] values) {...}
         public void     setB(int index, String value) {...}

       }
    ```
  - **Methods equals() and hashCode()**
    The generated Java bean classes contain an implementation of the equals() method. The generation of this method is outside the JAX-RPC specification. The equals() method returns true if equals() is true for each contained bean property. The implementation accounts for self-referencing loops. This version of the equals() method is typically more useful than the “identity” equals provided by java.lang.Object. A corresponding hashCode() method is also generated in the Java bean class.

- **Properties and indexed properties**
  In the standard Java bean mapping example, the nested xsd:element for property a is mapped to a Java bean property. In addition, the WSDL2Java command maps a nested xsd:element with maxOccurs > 1 to a Java bean indexed property.
- **Attributes**
  The WSDL2Java command also supports the xsd:attribute element, as shown in the following example.

  Attribute a is mapped as a Java bean property, which is exactly the same mapping as a nested xsd:element. Implementation specific metadata is generated in the Sample2_Helper class to ensure that property a is serialized and deserialized as an attribute, and not as a nested element. For example:

  XML:

```
<xsd:complexType name="Sample2">
  <xsd:sequence>
    <xsd:attribute name="a" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
```

  Java:

```
public class Sample2 {
     // ..
     public Sample2() {}

     // Bean Property a
     public String getA()            {...}
     public void   setA(String value) {...}
  }
```

- **Qualified versus unqualified names**

  The WSDL2Java command supports the elementForm and attributeForm schema attributes. This support is not specified in the JAX-RPC specification. These attributes are used to indicate whether an element or attribute is serialized and deserialized with a qualified or unqualified name. The default setting for elementForm is qualified and the default setting for attributeForm is unqualified. These settings do not affect the Java bean class that is generated, but the information is captured in the _Helper class metadata.
- **Extension and the abstract attribute**
  The WSDL2Java command supports extension of an xsd:complexType through the xsd:extension element. This support is required by the JAX-RPC specification.

  The WSDL2Java command supports the abstract attribute. This feature is listed as optional by the JAX-RPC specification.

  The following example shows the accepted use of the extension and abstract constructs. WebSphere Application Server uses the extension and abstract constructs to support polymorphism.

  XML:

```
<xsd:complexType name="Base" abstract="true">
  <xsd:sequence>
    <xsd:element name="a" type="xsd:int" />
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="Derived">
  <xsd:complexContent>
    <xsd:extension base="ns:Base">
      <xsd:sequence>
        <xsd:element name="b" type="xsd:int" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

  Java:

```
public abstract class Base {
   // ...
   public Base() {}

   public int getA() {...}
   public void setA(int a) {...}
}

public class Derived extends Base {
   // ...
   public Derived() {}

   public int getB() {...}
   public void setB(int b) {...}
}
```

- **Support for xsd:any**
  The WSDL2Java command supports xsd:anyelement, which is different than xsd:anyType. This feature is not defined within the JAX-RPC specification and is subject to change.

  If an <xsd:any/> element is defined within xsd:sequence or xsd:all group, SOAP values that do match one of the xsd:elements are stored in the Java bean as com.ibm.ws.webservices.engine.xmlsoap.SOAPElement objects. Values can be accessed from the Java bean using the get_any() and set_any() methods.

- **Mapping of xsd:element**

  An xsd:element is a construct that has a name or name attribute, and a type defined by a complexType or primitive type. There are two different kinds of xsd:elements:

  - Root: Defined directly underneath the schema elements and referenced by other constructs.
  - Nested: Nested underneath group elements and are not referenced by other constructs.

  Root elements are referenced by the WSDL file constructs, especially if the WSDL file is used to describe a literal service. Typically, root elements and types have the same names, which is allowed in the schema language. Under most circumstances the WSDL2Java command can produce Java artifacts without name collisions.

  - **Four ways to reference a type**
    There are four ways that a nested or root xsd:element can reference a type:
    - Use the type attribute:
      This is the most common way to reference a type, for example:

      ```
      <xsd:element name="one" type="ns:myType" />
      ```

      The WSDL2Java command recognizes the type attribute as a reference to a complexType or simpleType named, myType. The WSDL2Java command generates a Java type based on the characteristics of myType. Support for the type attribute is required by the JAX-RPC specification.
    - Use the ref attribute:
      For example:

      ```
      <xsd:element ref="ns:myElement" />
      ```

      The WSDL2Java command recognizes the ref attribute as a reference to another root element named myElement. The name of the element is obtained from the referenced element, such as myElement. The type of the element is the type of the referenced element. The WSDL2Java command generates a Java type based on the characteristics of the referenced type. The ref attribute is an optional feature of the JAX-RPC specification.
    - Use no attribute:
      For example:

      ```
      <xsd:element name="three" />
      ```

      When you do not use an attribute, the WSDL2Java command recognizes a reference to the xsd:anyType as defined by the XML schema specification. The xsd:anyType is an optional type of the JAX-RPC specification.

- Use an anonymous type:
  For example:

  ```
  <xsd:element name="four">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="foo" type="xsd:string" />
      </xsd:sequence>
    </xsd:complexType>
  </ xsd:element>
  ```

  When you use an anonymous type, the WSDL2Java command recognizes a reference to the type defined within the element.

  Note: The complexType does not have a name.

  The WSDL2Java command generates a Java type based on the characteristics of this type. Since the anonymous type does not have a name, the WSDL2Java command uses the name of the container element, which can result in collisions with defined types and other anonymous types. The WSDL2Java command automatically detects and renames classes to avoid collisions. Support for anonymous types is not defined by the JAX-RPC specification, however using anonymous types is common. Note: An xsd:attribute is like an xsd:element; it contains a name and refers to a type. An xsd:attribute can refer to its type with the type attribute or using an anonymous type.

- **Element specific attributes**
  Some attributes can be applied to xsd:elements and not to XML types.

  The maxOccurs attribute indicates the maximum number of occurrences of the element in the SOAP message. The default value is 1. If the value is greater than 1, or unbounded, the WSDL2Java command maps the construct to a Java array or bean indexed property. Metadata is also generated to properly serialize and deserialize a series of elements versus a normal XML array. The maxOccurs attribute is an optional feature of the JAX-RPC specification.

  The minOccurs attribute indicates the minimum number of occurrences of the element in the SOAP message. The default value is 1. The xsi:nillable attribute indicates whether the element can have a nil value. The minOccurs and xsi:nillable settings affect how a null value is serialized in a SOAP message. If minOccurs=0, the null value is not serialized. If xsi:nillable=true, the value is serialized with the xsi:nil=true attribute.

- **Mapping of xsd:complexType to Java array**
  The WSDL2Java command maps the following three kinds of XML formats to Java arrays:

  XML:

  ```
  <xsd:element name="array1" type="soapenc:Array" />
  ```

  Java:

  ```
  Object[] array1;
  ```

  XML:

  ```
  <xsd:complexType name="arrayOfInt">
    <xsd:complexContent>
      <xsd:restriction base:"soapenc:Array">
        <xsd:attribute ref:"soapenc:arrayType" wsdl:arrayType="xsd:int[]" />
      </xsd:restriction>
    </xsd:complexContext>
  </xsd:complexType>
  <xsd:element name="array2" type="ns:arrayOfInt" />
  ```

  Java:

  ```
    int[] array2;
  ```

  XML:

  ```
  <xsd:complexType name="arrayOfInt">
    <xsd:complexContent>
      <xsd:restriction base:"soapenc:Array">
        <xsd:sequence>
          <xsd:element name="item" type="xsd:int" maxOccurs="unbounded" />
        </xsd:sequence>
  ```

```
    </xsd:restriction>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="array3" type="ns:arrayOfInt" />
```

Java:

```
  int[] array3;
```

- **Mapping of xsd:simpleType enumeration**
  The WSDL2Java command maps the following XML enumeration to a JAX-RPC specified enumeration class. See section 4.2.4 of the JAX-RPC specification for more details.

  ```
  <xsd:simpleType name="EyeColorType" >
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="brown"/>
      <xsd:enumeration value="green"/>
      <xsd:enumeration value="blue"/>
    </xsd:restriction>
  </xsd:simpleType>
  ```
- **Mapping of xsd:complexType to exception class**
  If a complexType is referenced in a wsdl:message for a wsdl:fault, the complexType is mapped to a class that extends the exception, java.lang.Exception. This mapping is similar to the mapping of a complexType to a Java bean class, except a full constructor is generated, and only getter methods are generated. See section 4.3.6 of the JAX-RPC specification for more details.
- **Other mappings**
  The WSDL2Java command supports the mapping of xsd:simpleType and xsd:complexTypes that extend xsd:simpleTypes. These constructs are mapped to Java bean classes. The simple value is mapped to a Java bean property named, value. This is an optional feature of the JAX-RPC specification.

Mapping of wsdl:portType

The wsdl:portType construct is mapped to the Service Endpoint Interface. The name of the wsdl:portType is mapped to the name of the class of the Service Endpoint Interface.

Mapping of wsdl:operation

A wsdl:operation within a wsdl:portType is mapped to a method of the Service Endpoint Interface. The name of the wsdl:operation is mapped to the name of the method. The wsdl:operation contains wsdl:input and wsdl:output elements that reference the request and response wsdl:message constructs using the message attribute. The wsdl:operation can contain a wsdl:fault element that references a wsdl:message describing the fault. These faults are mapped to Java classes that extend the exception, java.lang.Exception as discussed in section 4.3.6 of the JAX-RPC specification.

- **Effect of document literal wrapped format**
  If the WSDL file uses the .NET document and literal wrapped format, the method parameters are mapped from the wrapper xsd:element. The .NET document and literal format is automatically detected by the WSDL2Java command. The following criteria must be met:
  - The WSDL file must have style="document" in its wsdl:binding constructs.
  - The WSDL file must have use="literal" in its wsdl:binding constructs.
  - The wsdl:message referenced by the wsdl:operation input construct must have a single part.
  - The part must use the element attribute to reference an xsd:element.
  - The referenced xsd:element, or wrapper element, must have the same name as the wsdl:operation.
  - The wrapper element must not contain any xsd:attributes.

  In such cases, each parameter name is mapped from a nested xsd:element contained within wrapper element. The type of the parameter is mapped from the type of the nested xsd:element. For example:

  XML:

```
<xsd:element name="myMethod" >
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="param1" type="xsd:string" />
      <xsd:element name="param2" type="xsd:int" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
...
<wsdl:message name="response" />
  <part name="parameters" element="ns:myMethod" />
</wsdl:message name="response" />

<wsdl:message name="response" />
...
<wsdl:operation name="myMethod">
  <input name="input" message="request" />
  <output name="output" message="response" />
</wsdl:operation>
```

Java:

```
void myMethod(String param1, int param2) ...
```

- **Parameter mapping**

  If the document and literal wrapped format is not detected, the parameter mapping follows the normal JAX-RPC mapping rules set in section 4.3.4 of the JAX-RPC specification.

  Each parameter is defined by a wsdl:message part referenced from the input and output elements.

  – A wsdl:part in the request wsdl:message is mapped to an input parameter.

  – A wsdl:part in the response wsdl:message is mapped to the return value. If there are multiple wsdl:parts in the response message, they are mapped to output parameters.

    - A Holder class is generated for each output parameter as discussed in section 4.3.5 of the JAX-RPC specification.

  – A wsdl:part that is both the request and response wsdl:message is mapped to an inout parameter.

    - A Holder class is generated for each inout parameter as discussed in section 4.3.5 of the JAX-RPC specification.

    - The wsdl:operation parameterOrder attribute defines the order of the parameters.

  The WSDL2Java command supports overloaded methods, but confirm that the part names of the overloaded methods are unique. For example:

  XML:

```
<wsdl:message name="request" >
  <part name="param1" type="xsd:string" />
  <part name="param2" type="xsd:int" />
</wsdl:message name="response" />

<wsdl:message name="response" />
...
<wsdl:operation name="myMethod" parameterOrder="param1, param2">
  <input name="input" message="request" />
  <output name="output" message="response" />
</wsdl:operation>
```

  Java:

```
void myMethod(String param1, int param2) ...
```

Mapping of wsdl:binding

The WSDL2Java command uses the wsdl:binding information to generate an implementation specific client side stub. WebSphere Application Server uses the wsdl:binding information on the server side to properly deserialize the request, invoke the Web service, and serialize the response. The information in the wsdl:binding should not affect the generation of the Service Endpoint Interface, but it can when the document and literal wrapped format is used or when there are MIME attachments.

- **MIME attachments**

  For a WSDL 1.1 compliant WSDL file, a part of an operation message, which is defined in the binding to be a MIME attachment, becomes a parameter of the type of the attachment regardless of the part declared. For example:

  XML:

```
<wsdl:types>
  <schema ...>
    <complexType name="ArrayOfBinary">
      <restriction base="soapenc:Array">
        <attribute ref="soapenc:arrayType" wsdl:arrayType="xsd:binary[]" />
      </restriction>
    </complexType>
  </schema>
</wsdl:types>

<wsdl:message name="request">
  <part name="param1" type="ns:ArrayOfBinary" />
<wsdl:message name="response" />

<wsdl:message name="response" />
...

  <wsdl:operation name="myMethod">
    <input name="input" message="request" />
    <output name="output" message="response" />
  </wsdl:operation>
  ...

<binding ...
  <wsdl:operation name="myMethod">
    <input>
      <mime:multipartRelated>
        <mime:part>
          <mime:content part="param1" type="image/jpeg"/>
        </mime:part>
      </mime:multipartRelated>
    </input>
   ...
  </wsdl:operation>
```

  Java:

```
void myMethod(java.awt.Image param1) ...
```

  The JAX-RPC requires support for the following MIME types:

MIME type	**Java type**
image/gif	java.awt.Image
image/jpeg	java.awt.Image
text/plain	java.lang.String
multipart/*	javax.mail.internet.MimeMultipart
text/xml	javax.xml.transform.Source
application/xml	javax.xml.transform.Source

  There are a number of problems with MIME attachments as they are defined in WSDL 1.1, including:

  - The semantics of the mime:multipartRelated clause are not fully defined.
  - The semantics do not allow for arrays of MIME attachments.

Because of these problems, several types are not specified by the JAX-RPC for MIME attachments. These types are defined in the supported mappings previously discussed.

- **Headers**

  A wsdl:binding can also define SOAP headers, for example:

  XML:

```
<wsdl:message name="request">
    <part name="param1" type="xsd:string" />
  </wsdl:message/>

  <wsdl:message name="response" />

  <wsdl:operation name="myMethod">
    <input name="input" message="request" />
    <output name="output" message="response" />
  </wsdl:operation>


<binding ...
  <wsdl:operation name="myMethod">
    <input>
      <soap:header message="request" part="param1" use="literal" />
    </input>

  </wsdl:operation>
```

  Java:

```
void myMethod(String param1) ...
```

  This is an example of an explicit header or a header with a value determined from a method parameter. Instead of appearing in the soap:body SOAP message, the value of param1 now appears in the soap:header SOAP message. The WSDL2Java command supports explicit headers and does not support implicit headers. Implicit headers have a value not determined by a parameter. For example, you could replace the soap:header clause in the example with:

```
<soap:header message="someOtherMsgNotAppearingInthePortType"
  part="someOtherPart" use="literal"/>
```

  Note: The WSDL2Java command supports explicit headers, but it is not considered good programming practice to use them. Headers are typically used for middleware logic, not business logic. Explicit headers place parameters used in business logic into the header.

Mapping of wsdl:service

The wsdl:service element is mapped to a Generated Service interface. The Generated Service interface contains methods to access each of the ports in the wsdl:service. The Generated Service interface is discussed in sections 4.3.9, 4.3.10, and 4.3.11 of the JAX-RPC specification.

In addition, the wsdl:service element is mapped to the implementation-specific ServiceLocator class, which is an implementation of the Generated Service interface.

Mapping between WSDL and SOAP messages

The WSDL file defines the format of the SOAP message that is sent over the wire. The WSDL2Java command and the WebSphere Application Server run time use the information in the WSDL file to confirm that the SOAP message is properly serialized and deserialized.

Document versus RPC, literal versus encoded

If a wsdl:binding indicates a message is sent using an RPC format, the SOAP message contains an element defining the operation. If a wsdl:binding indicates the message is sent using a document format, the SOAP message does not contain the operation element.

If the wsdl:part is defined using the type attribute, the name and type of the part are used in the message. If the wsdl:part is defined using the element attribute, the name and type of the element are used in the message. The element attribute is not allowed by the JAX-RPC specification when use="encoded".

If a wsdl:binding indicates a message is encoded, the values in the message are sent with xsi:type information. If a wsdl:binding indicates that a message is literal, the values in the message are typically not sent with xsi:type information. For example:

WSDL:

```
<xsd:element name="c" type="xsd:int" />
  ...
  <wsdl:message name="request">
    <part name="a" type="xsd:string" />
    <part name="b" element="ns:c" />
  </wsdl:message>
 ...
  <wsdl:operation name="method" >
    <input message="request" />
 ...
```

RPC/ENCODED:

```
<soap:body>
    <ns:method>
      <a xsi:type="xsd:string">ABC</a>
      <element attribute is not allowed in rpc/encoded mode>
    </ns:method>
  </soap:body>
```

DOCUMENT/LITERAL:

```
<soap:body>
  <a>ABC</a>
  <c>123</a>
</soap:body>
```

DOCUMENT/LITERAL wrapped:

```
<soap:body>
  <ns:method_wrapper>
    <a>ABC</a>
    <c>123</a>
  <ns:method_wrapper>
</soap:body>
```

The document and literal wrapped mode is the same as the document and literal mode. However, in the document and literal wrapped mode, there is only a single element within the body, and the element has the same name as the operation.

Multi-ref processing

If use=encoded, XML types that are not simpleTypes are passed in the SOAP message using the multi-ref attributes, href and id. The following example assumes that parameters one and two reference the same Java bean named, info containing fields a and b:

Note: Deserialization produces a single instance of the info class for the encoded case, and two instances for the literal case.

RPC/ENCODED:

```
<soap:body>
  <ns:method>
    <param1 href="#id1" />
    <param2 href="#id2" />
  <ns:method>
  <multiref id="id1" xsi:type="ns:info">
    <a xsi:type="xsi:string">hello<a>
    <b xsi:type="xsi:string">world</b>
  </multiref>
</soap:body>
```

RPC/LITERAL:

```
<soap:body>
  <ns:method>
    <param1>
      <a>hello</a>
      <b>world</b>
    </param1>
    <param2>
      <a>hello</a>
      <b>world</b>
    </param2>
  <ns:method>
</soap:body>
```

XML arrays and the maxOccurs attribute

A SOAP message is affected by whether the element is defined by an XML array or using the maxOccurs attribute.

WSDL:

```
<element name="foo" type="ns:ArrayOfString" />
```

Literal Instance:

```
  <foo>
    <item>A</item>
    <item>B</item>
    <item>C</item>
  </foo>
```

WSDL:

```
<element name="foo" maxOccurs="unbounded" type="xsd:string"/>
```

Literal Instance:

```
  <foo>A</foo>
  <foo>B</foo>
  <foo>C</foo>
```

minOccurs and nillable attributes

An element specified with minOccurs=0 that has a null value is not serialized in the SOAP message. An element specifying nillable="true" has a null value and is serialized into a SOAP message with the xsi:nil=true attribute. For example:

```
<a xsi:nil="true" />
```

Qualified versus unqualified

The XML schema attributeForm and elementForm attributes indicate whether the attributes and nested elements are serialized with qualified or unqualified names. If a part name is serialized, it is always serialized as an unqualified name.

Develop and manage Simple Object Access Protocol (SOAP)

This topic and the topics linking to it describe how to use Apache SOAP 2.3 support for WebSphere Application Server - Express Version 5.0 and 5.0.1. This support is deprecated. If you are planning a new Web services project, it is recommended that you use Web services for J2EE. For more information on developing a Web service with J2EE, see "Develop a J2EE Web service based on an existing application" on page 7.

Version 2.2 of the Apache SOAP implementation is integrated into WebSphere Application Server - Express. Apache SOAP Version 2.2 is a Java$^{(TM)}$-based implementation of the Simple Object Access Protocol (SOAP) 1.1 specification with support for SOAP with attachments.

WebSphere Application Server - Express allows you to publish the Java-based and other components as SOAP services:

- Java beans
- DB2 Universal Database stored procedures
- Server-side scripts that implement the Bean Scripting Framework (BSF)

See these topics for more information about developing SOAP services:

"Build a SOAP client"
This topic describes the steps necessary to create a client for a SOAP application. It also includes information about securing SOAP services.

"Deploy a programming component as a SOAP accessible Web service" on page 38
This topic describes the steps necessary to deploy a programming component such as an enterprise bean, DB2 stored procedure, or Bean Scripting Framework scripts.

"Soap examples" on page 39
This topic describes the SOAP response example and the SOAP request example.

"SOAP tools" on page 40
This topic describes how to use the XML-SOAP Admin tool and the SOAPEarEnabler tool to develop and manage your SOAP services.

"Apache SOAP deployment descriptors" on page 42
This topic discusses on the Apache Foundation's implementation of SOAP, which is used to publish and and use SOAP services under WebSphere Application Server - Express.

"Secure SOAP services" on page 43
This topic discusses the security implications of making SOAP services available on the Internet.

Build a SOAP client

The Apache SOAP implementation, integrated with WebSphere Application Server - Express, contains a client API to assist in SOAP client application development. Because the SOAP API is a standard for Web services, any clients that you create to access the WebSphere Application Server - Express SOAP services can also run in different implementations.

The steps for creating a client that interacts with a SOAP Remote Procedure Call (RPC) service include:

1. **Obtain the interface description of the SOAP service**
   This provides you with the signatures of the methods that you want to invoke. You can either look at a WSDL file for the service, or view the service itself to see its implementation.

2. **Create the Call object**
   The SOAP Call object is the main interface to the underlying SOAP RPC code.
3. **Set the target URI (Uniform Resource Identifier)**
   You do this in the Call object using the setTargetObjectURI() method. Pass the URN (Uniform Resource Name, a type of URI), that the service uses for its identifier, in the deployment descriptor.
4. **Set the method name that you want to invoke**
   You do this in the Call object using the setMethodName() method. This method must be one of the methods published by the service located at the URN from the previous step.
5. **Create the necessary Parameter objects for the RPC call**
   Once you have created the Parameter objects, set them in the Call object using the setParams() method. Ensure you have the same number and same type of parameters as those required by the service.
6. **Run the Call object's invoke() method and retrieve the Response object**

   Note: The RPC call is synchronous, so it might take some time to complete.
7. **Validate Response object**
   Check the response for a fault using the getFault() method, and then extract any results or returned parameters.

   Note: Although most of the providers only return a result, the DB2 stored procedure provider can also return output parameters.

Interacting with a document oriented SOAP service requires you to use lower level Apache SOAP API calls. You must first construct an Envelope object that contains the contents of the message (including the body and any headers) that you want to send. Then create a Message object in which you invoke the send() method to perform the actual transmission.

Creating a secure SOAP service

To create a secure SOAP service, perform these steps:

1. Create a simple object.
2. Define an envelope editor.
3. Specify a pluggable envelope editor.
4. Define the transports.

For more information about the envelope editor, see "Envelope Editor."

Your code might look like this example:

Note: For legal information about this code example, see the "Code license and disclaimer information" on page 178.

```
EnvelopeEditor editor= new PluggableEnvelopeEditor(new InputSource(conf), home);
SOAPTransport transport = new FilterTransport(editor, new SOAPHTTPConnection());
call.setSOAPTransport(transport);
```

The characteristics of the secure session are specified by the configuration file, conf.

Envelope Editor: The Envelope Editor is a component that can be plugged into the Apache SOAP transports. On the server side, it is embedded into the RPC and MessageRouterServlets. On the client side, it is embedded in the FilterTransport, which implements the SOAPTransport interface. WebSphere Application Server - Express provides a PluggableEnvelopeEditor, which you can use to plug in some editing components such as signature and verification.

Description of the factory class to instantiate Envelope Editors

A factory class creates Envelope Editors at run time. The factory class is called DSigFactory. The DSigFactory class uses an editor configuration file, and creates an instance of Envelope Editor. The factory class and the configuration file are specified in /QIBM/UserData/WebASE/ASE5/*instance_name*/installedApps/*node_name*/*earfile_name*/soapsec.war/WEB-INF/web.xml, where *instance_name* is the root directory of your WebSphere Application Server - Express instance.

The factory class is described under the <servlet id=“Servlet_17”> and <servlet id=“Servlet_2”> elements:

```
<display-name>Apache-SOAP-SEC</display-name>
<description>SOAP Security Enablement WAR</description>
<servlet id="Servlet_1">
   <servlet-name>rpcrouter</servlet-name>
   <display-name>Apache-SOAP Secure RPC Router</display-name>
   <description>no description</description>
   <servlet-class>com.ibm.soap.server.http.WASRPCRouterServlet</servlet-class>
   <init-param id="InitParam_1">
      <param-name>faultListener</param-name>
      <param-value>org.apache.soap.server.DOMFaultListener</param-value>
   </init-param>
   <init-param id="InitParam_2">
      <param-name>EnvelopeEditorFactory</param-name>
      <param-value>com.ibm.soap.dsig.dsigfactory.DSigFactory</param-value>
         </init-param>
         <init-param id="InitParam_3">
            <param-name>SOAPEvnelopeEditorConfigFilePath</param-name>
            <param-value>conf/sv-editor-config.xml</param-value>
         </init-param>
</servlet>
<servlet id="Servlet_2">
   <servlet-name>messagerouter</servlet-name>
   <display-name>Apache-SOAP Secure Message Router</display-name>
   <servlet-class>com.ibm.soap.server.http.WASMessageRouterServlet</servlet-class>
   <init-param id="InitParam_5">
      <param-name>faultListener</param-name>
      <param-value>org.apache.soap.server.DOMFaultListener</param-value>
   </init-param>
   <init-param id="InitParam_6">
      <param-name>EnvelopeEditorFactory</param-name>
      <param-value>com.ibm.soap.dsig.dsigfactory.DSigFactory</param-value>
   </init-param>
   <init-param id="InitParam_7">
      <param-name>SOAPEvnelopeEditorConfigFilePath</param-name>
      <param-value>conf/sv-editor-config.xml</param-value>
  </init-param>
</servlet>
```

EnvelopeEditorFactory is a factory class. SOAPEnvelopeEditorConfigFilePath is a configuration file for Envelope Editor.

Enabling Envelope Editor

On the client side, the configuration of the Envelope Editor is explicitly programmed. On the server side, the transport hook is enabled automatically in the soapsec.war file when you add the init parameter to the RPC and Message router servlets for the EnvelopeEditorFactory. This entry in the web.xml for the soapsec.war file is added automatically when you enable an application for SOAP and indicate the service is secure.

Configuration file of Envelope Editor

The configuration file, sv-editor-config.xml is located in /QIBM/UserData/WebASE/ASE5/instance_name/installedApps/*node_name*/*ear_file_name*/soapsec.war/conf/sv-editor-config.xml, where *earfile_name* is the name of your Enterprise

Archive (EAR) file. Under the SOAPEnvelopeEditorConfig element, there are two optional elements: incoming and outgoing. The incoming and outgoing element definitions look like this example:

```
<incoming class="com.ibm.xml.soap.security.dsig.SOAPVerifier">
   <init-param>
      <param-name>filename</param-name>
      <param-value>conf/sv-ver-config.xml</param-value>
   </init-param>
</incoming>
<outgoing class="com.ibm.xml.soap.security.dsig.SOAPSigner">
   <init-param>
      <param-name>filename</param-name>
      <param-value>conf/sv-sig-config.xml</param-value>
   </init-param>
</outgoing>
```

The incoming element specifies a class that edits incoming messages and a configuration file for the editing class. The outgoing element specifies a class for outgoing message and a configuration file.

Changing the configuration

You do not have a digital signature for response messages if you remove the outgoing element from /QIBM/UserData/WebASE/ASE5/*instance_name*/installedApps/*node_name*/*earfile_name*/soapsec.war/conf/sv-editor-config.xml and remove the incoming element from /QIBM/UserData/WebASE/ASE5/*instance_name*/installedApps/*node_name*/*ear_file_name*/soapsec.war/conf/cl-editor- config.xml, where *ear_file_name* is the name of your Enterprise Archive (EAR) file.

Note: Examples may be wrapped for display purposes.

Deploy a programming component as a SOAP accessible Web service

Complete these steps to deploy a SOAP-accessible Web service on WebSphere Application Server - Express:

1. **Create or locate the software resource to be published as a service**
   To deploy a service, create one or more of the supported programming components (beans, DB2 stored procedure, or BSF script), or locate an existing piece of code of the supported type.
2. **Assemble an Enterprise Archive (EAR) file**
   Package the component into an Enterprise Archive (EAR) file. This step is a deployment packaging requirement of WebSphere Application Server - Express. Create the EAR file.
3. **Create the SOAP deployment descriptor for the desired service**
   In order to deploy a component as a SOAP service, create an Apache SOAP deployment descriptor that describes the service you are creating. This step designates the programming component as a **service**. The descriptor describes and defines the parts of the code that are invoked through SOAP calls.

   The information contained in the deployment descriptor varies, depending on the type of artifact you are publishing. For example, this deployment descriptor could be used with the StockQuoteSample:

   ```
   <isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
     id="urn:service-urn" [type="message"]>
      <isd:provider type="java" scope="Request | Session | Application" methods="exposed-methods">
        <isd:java class="implementing-class" [static="true|false"]/>
     </isd:provider>
     <isd:faultListener>org.apache.soap.server.DOMFaultListener</isd:faultListener>
   </isd:service>
   ```
4. **Run the SoapEarEnabler tool to enable your Web service**
   You must first package your component into an Enterprise Archive (EAR) file. Next, using the deployment descriptor as input data, add the necessary pieces to the EAR file to enable the component as a Web service. To facilitate this process, use the Java(TM)-based tool, SoapEarEnabler. Depending on whether you secure the Web service, this tool adds two Web modules: soap.war and

soap-sec.war to the EAR file. These Web modules include the SOAP deployment descriptors plus the necessary parts to deploy the service into the WebSphere Application Server - Express run time.

Note: The service does not become available until you install the SOAP enabled EAR file and restart the application server.

5. **Install the modified EAR file into your WebSphere Application Server - Express instance**
   Install the modified EAR file into your WebSphere Application Server - Express instance.
6. **Restart your WebSphere Application Server - Express instance.**

Soap examples

SOAP request example

The SOAP request that follows indicates that the OrderItem() method, from the Some-URI namespace, should be invoked from http://www.somesupplier.com/Supplier. Upon receiving this request, the supplier application at www.somesupplier.com runs the business logic that corresponds to OrderItem.

Note: For legal information about this code example, see the Code example disclaimer.

```
Sample SOAP Request
POST /Supplier HTTP/1.1
Host: www.somesupplier.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"

<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
      <m:OrderItem xmlns:m="Some-URI">
      <RetailerID>557010</RetailerID>
      <ItemNumber>1050420459</ItemNumber>
      <ItemName>AMF Night Hawk Pearl M2</ItemName>
      <ItemDesc>Bowling Ball</ItemDesc>
      <OrderQuantity>100</OrderQuantity>
      <WholesalePrice>130.95</WholeSalePrice>
      <OrderDateTime>2000-06-19 10:09:56</OrderDateTime>
      </m:OrderItem>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The SOAP protocol does not specify how to process the order. The supplier could run a CGI script, invoke a servlet, or perform any other process that generates the response.

SOAP response example

The response to a SOAP Request is an XML document that contains the results of the processing. In this example, this is the order number for the order placed by the retailer.

Note: For legal information about this code example, see the Code example disclaimer.

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
   <SOAP-ENV:Body>
      <m:OrderItemResponse xmlns:m="Some-URI">
      <OrderNumber>561381</OrderNumber>
      </m:OrderItemResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The response does not include a SOAP-specified header. The results are placed in an element whose name matches the method name (OrderItem) with the suffix (Response), such as `OrderItemResponse`.

SOAP tools

WebShere Application Server - Express provides two tools to help you develop and manage your SOAP services. The SoapEarEnabler tool is a Java application that enables a set of SOAP services within an Enterprise Application Archive (EAR) file. Also provided is a modified version of the Apache SOAP XML Admin interface for each SOAP enabled EAR file.

For more information about these tools, see these topics:

"SoapEarEnabler tool"
The SoapEarEnabler tool is a Java application that enables a set of SOAP services within an Enterprise Application Archive (EAR) file. See this topic for more information.

"XML-SOAP Admin tool" on page 41
This interface allows you to list configured services, showing active and stopped services, start and stop a service, and view the Apache SOAP deployment descriptor for a service. See this topic for more information.

SoapEarEnabler tool: The SoapEarEnabler tool is a Java application that enables a set of SOAP services within an Enterprise Application Archive (EAR) file. The SoapEarEnabler guides you through the required steps to enable one or more services within an application. It makes a backup copy of your original EAR file in case you later need to remove or add services.

Note: Start with an existing EAR file. The SoapEarEnabler tool does not accept a SOAP enabled EAR file as input.

You are prompted as to whether you want to add the administration client to the EAR file. This is a Web-based client that allows you to list all active services for a specific context from a browser window. With this interface, you can stop and start existing services. You can choose to not add this interface for security reasons, or you can secure the interface before making a service available.

Perform these steps to invoke and use the SOAPEarEnabler tool:

1. Create an Apache SOAP deployment descriptor for each service to be enabled.
2. You can invoke the SOAPEarEnabler tool from QShell on iSeries.
3. If running in interactive mode, enter the required information when prompted by the tool.

Silent mode

If you use the SOAPEarEnabler in silent mode, you must supply all the required parameters as command line arguments. These include:

- ***ear-file-name***
  This is the name of the EAR file to which you want to add SOAP services. Note that you can type a path relative to the bin directory.
- ***number-of-services***
  This is the number of services you want to enable for the EAR file. If you want to enable more than one service, you must specify that number of each of these parameters:
  - *deployment-descriptor-file-name*
    This is the name of the deployment descriptor file you are using to describe the SOAP service you are enabling for this EAR file.
  - Specify whether the service is an enterprise bean.
    Type `n` to indicate that the service you are adding is not an enterprise bean. WebSphere Application Server - Express does not support enterprise beans.

– Specify whether this service should be secured.
  Type `y` to secure this SOAP service; type `n` if you do not want to enable security.
  - If you typed `y`, type the context root for the secured service. Example: `/soapsec`.
  - If you typed `n`, type the context root for the nonsecured service. Example: `/soap`.

Silent mode examples

When invoking the SOAPEarEnabler tool in silent mode, type the command and its parameters as a single, continuous line.

Note: The text for the following commands have been wrapped for better readability. If you are using the command from iSeries, use '/' in the pathnames instead of '\'.

This is an example of deploying 2 java classes as nonsecured services:

```
 soapearenabler soap.ear 2 xml-soap\java\samples\stockquote\deploymentdescriptor.xml
n 1 samples.jar n xml-soap\java\samples\addressbook\deploymentdescriptor.xml
n 1 samples.jar n /soap
```

Interactive mode

In interactive mode, the SoapEarEnabler tool prompts you for all required information. For example:

```
Please enter the name of your ear file:
..\work\stockquote.ear
How many services would you like your application to contain (1...n)?
1

Now prompting for info for service #1:

   Please enter the file name of the deployment descriptor xml file:
   ..\work\Stockading.xml
   Is this service an EJB (y/n)?
   n
   How many jar files are required for this service (0...n)?
   1
   Classpath requirement #1: Please choose a file ([1] samples.jar, [2] stockquote.war):
   1
   Should this service be secured (y/n)?
   n

   Please enter a context root for your nonsecured services (Example: /soap):
   /soapsamples
   Do you wish to install the administration client?

   Warning! You should not install this client in a production ear
   unless you intend to secure the URI to it.

   Install the administration client (y= yes/n= no)?
   y
```

Note: Examples may be wrapped for display purposes.

XML-SOAP Admin tool: The WebSphere administrative console is available for administering enterprise applications. However, individual Web services are not visible in the administrative console.

The SOAPEarEnabler tool gives you the opportunity to add administrative interfaces to an EAR file. For this purpose, WebSphere Application Server - Express provides a modified version of the Apache SOAP XML Admin interface for each SOAP enabled EAR file. This interface allows you to do the following for each context root:

- List configured services, showing active and stopped services

- Start a service
- Stop a service
- View the Apache SOAP deployment descriptor for a service

Accessing the XML-SOAP Admin tool Access the XML-SOAP Admin tool through a Web browser. For example, to use this interface with the soap samples, type:

```
http://your_node_name:port/soapsamples/admin/index.html
```

where *your_node_name* is the host name of your WebSphere Application Server - Express instance and *port* is the port number that corresponds to that instance.

Note: You must start your WebSphere Application Server - Express instance before you attempt to access the URL.

You cannot use the XML-SOAP Admin tool to add or remove a service. Use the SOAPEarEnabler tool to add or remove services.

Note: To remove a service, you must first start with the original EAR file and then add only the services you want. The SoapEarEnabler tool does not accept a SOAP enabled EAR file as input.

A stopped service is persisted across starts and stops of the application server. Therefore, if you stop a service, it remains stopped until the next time you use the XML-SOAP Admin tool to start it.

You can add the XML-SOAP Admin tool interface to an enterprise application when you SOAP-enable the EAR file using the SOAPEarEnabler Tool. In interactive mode, you are asked whether you want to add the XML-SOAP Admin tool interface. When you reply, yes,the necessary JSP files and bindings are added that allow you to access the XML-SOAP Admin tool interface for the application. The interface is an optional addition because you might not want to expose it in a production environment.

Apache SOAP deployment descriptors

Apache SOAP utilizes XML documents called deployment descriptors to provide the SOAP run time with information on client services. The contents of the deployment descriptor vary, depending on the type of programming component that is published using SOAP. Deployment descriptors provide an array of information, such as:

- The Web service’s Uniform Resource Name (URN), which is used to route the request when it arrives
- Method and class details, if the service is being provided by a Java$^{(TM)}$ class
- User ID and password information, if the service provider must connect to a database

Standard Java class deployment descriptor

A deployment descriptor that publishes a service that is implemented with a standard Java class or bean can look like this example:

```
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment" id="urn:service-urn" [type="message"]>
   <isd:provider type="java" scope="Request | Session | Application" methods="exposed-methods">
      <isd:java class="implementing-class" [static"true|false"]/>
   </isd:provider>
   <isd:faultListener>org.apache.soap.server.DOMFaultListener</isd:faultListener>
</isd:service>
```

In the example,

- Variables:
  - *service-urn* is the URN that you give to a service. (All services deployed within a single EAR file must have URNs that are unique within that EAR file.)
  - *exposed-methods* is a list of methods, separated by spaces, which are being published.

- *implementing-class* is a fully qualified class name (that is, a packagename.classname) that provides the methods that you are publishing.

- The <service> element has an optional attribute called type which is set to the value `message` if the service is document oriented instead of invoked with a remote procedure call (RPC).
- The <java> element has an optional attribute called static, which can be set to either true or false, depending on whether the methods are available, or **exposed**, to service requesters. If exposed, this attribute indicates whether the method is static or not.
- The <provider> element a scope attribute that indicates the lifetime of the instantiation of the implementing class.
  - **Request** indicates the object is removed after the request completes.
  - **Session** indicates the object lasts for the lifetime of the HTTP session.
  - **Application** indicates the object lasts until the servlet that is servicing the requests is terminated.

Bean Scripting Framework (BSF) script deployment descriptor

A deployment descriptor that publishes a service that is implemented with a BSF script can look like the following example:

```
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment" id="urn:service-urn">
   <isd:provider type="script" scope="Request | Session | Application" methods="exposed-methods">
   <isd:script language="language-name" [source="source-filename"]>[script-body]
   </isd:script>
   </isd:provider>
   <isd:faultListener>org.apache.soap.server.DOMFaultListener</isd:faultListener>
</isd:service>
```

where:

- service-urn, exposed-methods, and scope have the same meaning as in the standard Java class deployment descriptor.
- language-name is the name of the BSF-supported language that you use to write the script.

The deployment descriptor must also have a source attribute on the <script> element, or a script-body attribute. The script-body attribute contains the script that is used to provide the service. If the deployment descriptor has the source attribute, then source-filename refers to the file that contains the service implementation.

Secure SOAP services

Originally, the SOAP specification left security issues open; thus, several proposals evolved to bridge the security gaps. The SOAP security extension, included with WebSphere Application Server - Express, is a security architecture based on the SOAP security specification, and widely accepted security technologies such as Secure Sockets Layer (SSL).

See these subtopics for more information about securing Web services that you publish using the SOAP protocol:

"HTTP authentication" on page 44
This topic discusses how to use secure SOAP service with basic HTTP authentication, SSL authentication, and with SOAP signatures.

"SOAP signature components" on page 45
This topic discusses the Envelope Editor, the Signature Header Handler, and the Verification Handler, which are used to implement security in SOAP enabled applications.

"SOAP security files reference" on page 48
This page provides a reference listing of the files related to implementing SOAP security using the Secure Sockets Layer (SSL) protocol.

HTTP authentication: There are three options for security when using HTTP as the transport protocol:

- HTTP basic authentication (page 44)
- SOAP on Secure Sockets Layer (SSL) with HTTP basic authentication (page 44)
- SOAP on SSL with SOAP signature (page 44)

HTTP authentication

Many applications require users to provide identifying information. You can restrict access to data based on the authorization level assigned to users.

The Apache implementation of SOAP has an API to set the user ID/password for HTTP basic authentication.

SOAP on Secure Sockets Layer (SSL) with HTTP basic authentication

To make a request over HTTPS, using the SSL support of Apache SOAP, you need a separate Java Secure Socket Extension (JSSE)

provider. WebSphere Application Server includes the ibmjsse.jar JAR file, which includes these APIs.

The SOAP on SSL scenario is useful for many business-to-business applications because:

- The data in transit is protected from eavesdropping or forgery by SSL
- The server identity is guaranteed by SSL server authentication
- The client identity is authenticated through user ID and password, which are encrypted by the SSL transport

For example, if an inventory application is configured as a web service, the service provider has these two SOAP service entries:

- https://a_company.com/inventory/inquiry
- https://a_company.com/inventory/update

Each SOAP service entry should be deployed as a separate enterprise application (EAR) because each service has a different access control policy, which is that anyone can inquire about the inventory but only the inventory clerks can update the contents.

Perform these steps to enable the SOAP on SSL scenario:

1. Configure the web server (httpd.conf) so that it only allows SSL access to these servlets.
2. Configure the security role for the RPCRouterServlet in the inquiry services EAR so that the RPCRouterServlet for the inquiry service is accessible by everyone, while the RPCRouterServlet for the 'update' service requires authentication based on the HTTP basic authentication (user ID/password).

In this case, the 'update' application does not know the identity of the requester; it only knows that access is granted.

SOAP on SSL with SOAP signature

Applications might need nonrepudiable proof of exchanged messages. One example is a web service that accepts part orders. The business partners establish a form of trust relationship based on public keys. This can be done using the public key infrastructure (PKI) through a third party certificate authority (CA), or by exchanging public keys through a secure channel. The following service is deployed with a signature verification function: `https://a_company.com/partorder`.

Configure signature verification with this information:

- Scope of signature (indicates the portion of the SOAP envelope that must be authenticated. The default is the content of SOAP-ENV:Body).
- Trusted keys or trusted root keys.
- Default key to verify signature if no KeyInfo is specified.
- Other policies regarding signature validation.
- Behavior when signature verification fails.
- Additional requirements on signature (as for example, specific requirements on hash/C14N algorithms to be used, timestamp validity, and so forth).

If the signature is missing or if signature verification fails, the signature verification function can be configured so that the servlet returns a SOAP fault.

To send part orders to the https://a_company.com/partorder service, the service requester should sign her or his SOAP messages with a signature component. The signature component is initialized using two templates:

1. <ds:SignedInfo> template
2. <ds:KeyInfo> template

The <ds:SignedInfo> template controls:

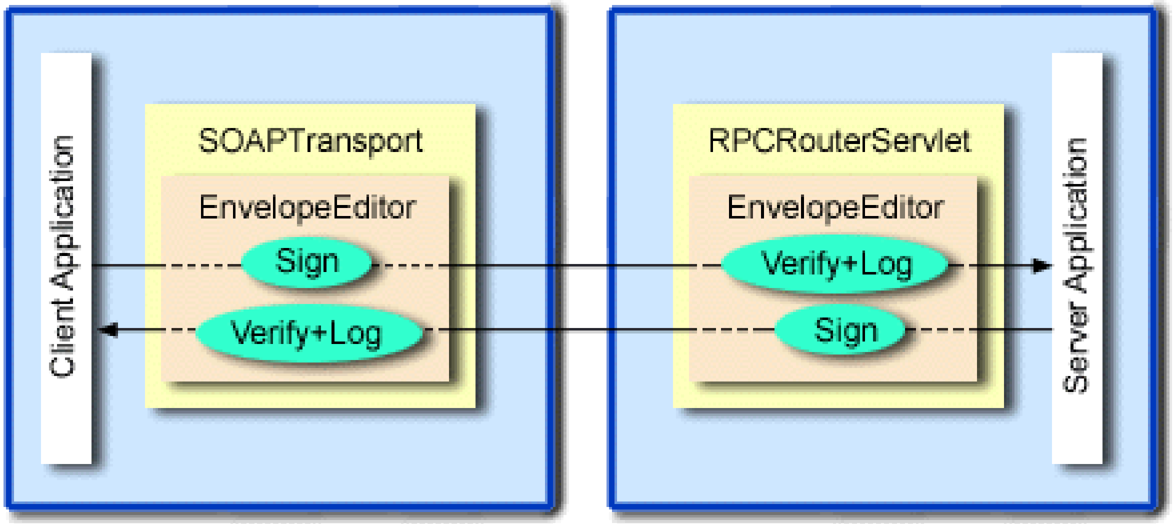
- What parts of the SOAP envelope must be signed
- What algorithms (canonicalization, transformation, digest, sign) should be used

The <ds:KeyInfo> template controls:

- Whether or not to include the entire certificate chain in <ds:KeyInfo>
- Decision to include only certificate and serial number
- Public key value
- Decision to provide no key information (so that the default key must be used for verification)

You can combine the service request with HTTP basic authentication, if necessary.

SOAP signature components: This graphic illustrates the concept of a SOAP signature:



Using the SOAP transport hook, you can plug in security components, namely a **signer** and a **verifier** that has logging capability. The transport hook is called the EnvelopeEditor. A PluggableEnvelopeEditor is also provided, which allows you to plug in your security components. As illustrated, the EnvelopeEditor is encapsulated in the SOAPTransport on the client side. On the server side, EnvelopeEditor is encapsulated in RPC/MessageRouterServlet. This means the same components can be used on either the client or server.

See Envelope Editor for instructions on enabling and using this pluggable component.

When a client application sends a request, the request is signed and transmitted to the server. At the server, the request is verified and delivered to a server application or, in the case of a RPC, to a Java(TM) object. The response is processed in the same manner. The verifier component also has a logging function to log the verified messages in a file. Signatures and verifier components are configurable. You can specify encryption, digest message algorithm, certificate path policy, and other security technologies.

You can control and customize how the SOAP envelope performs the signature and verification processes through these components:

- **"Signature Header Handler"**
  - The Signature Header Handler is a XML-based configuration file.
  - It enables a template for <SignedInfo> (for customizing references, sign/hash algorithms, C14N algorithms, optional timestamp).
  - It also enables a template for <KeyInfo> (for customizing the public key such as X.509 certificate)
- **"Verification Header Handler" on page 47**
  - The Verification Header Handler is a XML-based configuration file.
  - It enables configurable policy (required scope of signature, trusted root, certstore, certpathchecker).
  - It enables exit for Logging (additional application specific verification) A reference implementation of logging component is also provided.

Signature Header Handler: The Signature Header Handler (SHH) inserts a digital signature header into a SOAP envelope. You can customize the SHH configuration with a configuration file. For example, you can specify a signing policy and the key store file.

There are two signature configuration files:

- /QIBM/UserData/WebASE/ASE5/*instance_name*/installedApps/*node_name*/ *ear_file_name*/soapsec.war/conf/sv-sign-config.xml
- /QIBM/UserData/WebASE/ASE5/*instance_name*/installedApps/*node_name*/ *ear_file_name*/soapsec.war/conf/cl-sign-config.xml

where *ear_file_name* is the name of the Enterprise Archive (EAR) file that contains your SOAP application. The soapsamples.ear file contains samples of these configuration files.

Here is an explanation of each configuration element in the Signature Header:

- **KeyStore**
  The KeyStore element specifies a keystore file that holds the signing key. In this example, the type attribute indicates a key store type, and the jks attribute indicates Java(TM) Key Store. The path attribute denotes a keystore file, and the storepass attribute is its store password.

  ```
  <KeyStore
     type="jks"
     path="key/SOAPserver"
     storepass="server"
  />
  ```

  You can use the Key Management tool (iKeyman) to create a keystore file.
- **Policy**
  The PublicKey element specifies the information that should be included in the <ds:KeyInfo> element. With the current implementation, you must either include the complete certificate chain, or omit <ds:KeyInfo>. When you omit <ds:KeyInfo>, the recipient must know the default key to verify the signature.
- **Template**
  The contents of the Template element specify all the details related to XML Signature, including signature algorithms, digest algorithms, canonicalization algorithms, transform algorithms, the portion of the SOAP envelope to be signed, and so forth.
- **Object**
  The template can also have one or more Object elements for additional authentication information, such as a timestamp.
- **ValueOfTimestamp**
  This Signature Header Handler recognizes one special element type, ValueOfTimestamp, which is replaced with a current time and date before being inserted into the signature.

Note: Examples may be wrapped for display purposes.

Verification Header Handler: The Verification Header Handler (VHH) validates a digital signature header in a SOAP envelope. You can customize its configuration by using a configuration file where you specify:

- A verification policy
- The certificate path
- Logging files to record verified messages

There are two signature configuration files:

- /QIBM/UserData/WebASE/ASE5/*instance_name*/installedApps/*node_name*/ *ear_file_name*/soapsec.war/conf/sv-ver-config.xml
- /QIBM/UserData/WebASE/ASE5/*instance_name*/installedApps/*node_name*/ *ear_file_name*/soapsec.war/conf/cl-ver-config.xml

where *instance_name* is the name of the root directory for your instance of WebSphere Application Server.

Here is an explanation of each configuration element in the Verification Header:

- **AllowedAlgorithms**
  All the algorithms supported by this Verification Header Handler must be listed in this element. Algorithms other than these cannot be used in SOAP-SEC:Signature header. The current implementation supports all required algorithms in the XML Signature specification, except for SHA1-MAC.
- **RequiredAuthenticatedParts**
  This section specifies what parts of SOAP message need to be authenticated through the SOAP-SEC:Signature header. Currently two values are supported for the part attribute:
  - When part=“root”, the whole envelope must be signed through the enveloped-signature transform.
  - When part=“body”, the SOAP-ENV:Body element in the SOAP envelope must be referenced by one of the reference elements in the signature.

  In the future, part=“” allows an attachment to be specified. If the specified parts are not authenticated through the signature header entry, verification fails.
- **DefaultVerificationKeys**
  When KeyInfo is missing in the signature, the content of this element is used as a part of the signature. When communicating parties know the identity of each other, the default KeyInfo can be used to reduce the communication data volume.
- **Log**
  Specifies the logging behavior. These versions of logging exist:
  - When target=“all”, all verification attempts are logged.
  - When target=“success”, only successful verification are logged.
  - When target=“fail”, only unsuccessful verification are logged.

  Note: You can specify multiple LogFile elements.

  This example illustrates how to specify logging:

  ```
  <Log>
     <SOAPDSigLogger class=<com.ibm.xml.soap.security.dsig.SOAPDSigLoggerImpl">
        <LogFile target="all" path="SOAPVHH-all.log" append="yes"/>
     </SOAPDSigLogger>
     <SOAPDSigLogger class="com.ibm.xml.soap.security.dsig.SOAPDSigLoggerImpl">
        <LogFile target="fail" path="SOAPVHH-fail.log" append="yes"/>
     </SOAPDSigLogger>
  </Log>
  ```
- **PKIXParameters**
  Currently, the Verification Header Handler supports X.509/PKIX certificates only (no HMAC, no PGP, and so forth). The policies for PKIX certificate verification are specified in this element. This is a straightforward mapping of the Java$^{(TM)}$ CertPath API. Not all of the entries are meaningful in this initial release. The current implementation only allows the use of keystore as the means of specifying trusted root.

Note: Examples may be wrapped for display purposes.

SOAP security files reference: This page provides a reference listing of the files related to implementing SOAP security using the Secure Sockets Layer (SSL) protocol.

SOAP Security related files

This table provides a quick reference for SOAP security topics.

Note: All path listings are relative to the installation directory, /QIBM/UserData/WebASE/ASE5/*instance_name*, where *instance_name* is the root directory of your instance of WebSphere Application Server - Express.

Path	Contents	Description
/installedApps/soapsamples.ear/soapsec.war	Web-INF, conf, key, log, etc.	Home of the soap security servlets
/installedApps/soapsamples.ear/soapsec.war/WEB-INF	web.xml	Modified servlet configuration file for digital signature
/installedApps/soapsamples.ear/soapsec.war/conf	.config files	Configuration files for envelope editors and signature components
/installedApps/soapsamples.ear/soapsec.war/key	SOAPclient, SOAPserver	Keystore files
/installedApps/soapsamples.ear/soapsec.war/logs	Log files	Logs generated during security exchange
/installedApps/soapsamples.ear/ServerSamplesCode/src/*service_name*	server side samples	Source for both the nonsecure and secure samples
/installedApps/soapsamples.ear/ClientCode/nt_bat	scripts to run client samples	Batch files for invoking the client side samples to interact with the server-side services
/installedApps/soapsamples.ear/ClientCode/unix_scripts	scripts to run client samples	Batch files for invoking the client side samples to interact with the server-side services
/installedApps/soapsamples.ear/ClientCode/data	data files used by samples	Windows NT systems only
/installedApps/soapsamples.ear/ClientCode/src	client side samples source	UNIX systems only
/lib	soap.jar, soap-sec.jar, ws-soap-ext.jar	Location of all SOAP related JAR files

SOAP keystore files

SOAP certificates are stored in two keystore files, which are described in this table:

File name	Store password	Description
SOAPserver	server	This keystore is used by a service provider.
SOAPclient	client	This keystore is used by a service requester.

The certificates stored in both the SOAPserver and SOAPclient keystore files are described in this table:

Alias	Issuer	Description
soapca	soapca itself	The certificate of the root Certificate Authority (CA) used for testing purposes
intca1	soapca	The certificate of the CA to issue SSL related certificates

Alias	Issuer	Description
intca2	soapca	The certificate of the CA to issue SOAP-DSIG-related certificates

These two certificates are stored in the SOAPserver keystore:

Alias	Issuer	Description
sslserver	intca1	This is the certificate of the SSL server. This is also stored in the SOAPclient keystore as a trusted certificate. The PKCS12 file including the corresponding private key for this certificate is sslserver.p12.
soapprovider	intca2	This certificate might be used by a service provider to digitally sign its response message. The key password is “server”.

These three certificates are stored in the SOAPclient keystore:

Alias	Issuer	Description
sslclient	intca1	This certificate might be used for the SSL client authentication. The key password is “client”.
sslserver	intca1	This is the certificate of the trusted SSL server and the same as the one stored in the SOAPserver keystore. The PKCS12 file, including the corresponding private key for this certificate, is sslserver.p12.
soaprequester	intca2	This certificate might be used by a service requester to digitally sign its request message. The key password is “client”.

UDDI4J

UDDI4J is a Java$^{(TM)}$ class library that provides an API that is used to interact with a UDDI registry. This class library generates and parses messages sent to and received from a UDDI server. The central class in this set of APIs is com.ibm.uddi.client.UDDIProxy.

This class is a proxy for the UDDI server that is accessed from the client code. Its methods map to the UDDI Specification

.

Note: this document is in PDF format. You must have the Adobe Acrobat Reader installed as a plugin to your browser to view this document.

The classes within com.ibm.uddi.datatype represent data objects that send or receive UDDI information. In the business and service model, the data objects are also known as subpackages.

- **com.ibm.uddi.request**
  The subpackage com.ibm.uddi.request contains messages sent to the server. Generally, these classes are not used directly; rather, they are invoked by the UDDIProxy class.
- **com.ibm.uddi.response**
  The subpackage com.ibm.uddi.response represents response messages from a UDDI server.

UDDI4J error handling

The com.ibm.uddi.client.UDDIProxy package contains the following Java exceptions:

- **UDDIException**
  UDDIException is thrown when errors are received from the UDDI proxy when invoking UDDIProxy inquiry methods. UDDIException can contain a DispositionReport with information regarding the error. APIs that do not return a data object provide the disposition report.
- **SOAPException**
  SOAPException is thrown if a communication error occurs or if the resulting data cannot be parsed as a valid SOAP message.

For more information, visit the IBM DeveloperWorks uddi4j Project site

.

Enable Web services to use the Web Services Invocation Framework (WSIF)

The Web Services Invocation Framework (WSIF) is a WSDL-oriented Java API that allows you to invoke Web services dynamically, regardless of what format the service is implemented in, or what mechanism is used to access it.

WSIF enables you, as a Web services developer, to move away from the usual Web services programming model of working directly with the SOAP APIs, towards a model where you interact with representations of the services. You can work with the same programming model regardless of how the service is implemented and accessed.

For more information about WSIF, see these topics:

“Goals of WSIF”
This topic describes the goals of WSIF.

“An overview of WSIF” on page 53
This topic provides an overview of WSIF including a description of the chitecture and usage scenarios.

“Use WSIF to invoke Web services” on page 56
This topic describes how to use WSIF to invoke Web services.

“WSIF system management and administration” on page 60
This topic describes how to enable security for WSIF, and how to maintain the WSIF properties file.

WSIF API
This topic describes the WSIF APIs.

Goals of WSIF

SOAP bindings for Web services are part of the WSDL specification. When most developers think of using a Web service, they immediately think of assembling a SOAP message and sending it across the network to the service endpoint, using some SOAP client API. For example: with Apache SOAP the client creates and populates a Call object which encapsulates the service endpoint, the identification of the SOAP operation to be invoked, the parameters that have to be sent, and so on.

Although this works for SOAP, it is limited in its use as a general model for invoking Web services for these reasons:

- **Web services are not just SOAP services.**

  You can deploy as a Web service any program with a WSDL description of its functional aspects and access protocols; and in the J2EE environment, the same component is available over multiple transports and protocols.

  For example, you can have a database stored procedure, which is then exposed as a stateless session bean, and then deployed into a SOAP router to become a SOAP service. At each stage, the fundamental service is the same. All that changes is the access mechanism: from JDBC to RMI-IIOP and then to SOAP.

  The WSDL specification defines a SOAP binding for Web services, but you can add binding extensions to the WSDL so that, for example, you can offer an enterprise bean as a Web service using RMI/IIOP as the access protocol. You can even treat a single Java class as a Web service, with in-thread Java method invocations as the access protocol. With this broader definition of a Web service, you need a binding-independent mechanism for service invocation.
- **Tying client code to a particular protocol implementation is restricting.**

  If your client code is tightly bound to a client library for a particular protocol implementation, it can become hard to maintain. For example if you move from Apache SOAP to a different SOAP implementation, the process can take a lot of time and effort. To avoid these problems, you need a protocol implementation-independent mechanism for service invocation.
- **Incorporating new bindings into client code is hard.**

  If you want to make an application that uses a custom protocol work as a Web service, you can add extensibility elements to WSDL to define the new bindings. But in practice, achieving this is hard. For example you have to design the client APIs for using this protocol; and if your application uses just the abstract interface of the Web service, you have to write tools to generate the stubs that enable an abstraction layer. These are tasks that can take a lot of time and effort. What you need is a service invocation mechanism that allows bindings to be updated or new bindings to be plugged in easily.
- **Multiple bindings can be used in flexible ways.**

  Imagine that you have successfully deployed an application that uses a Web service offering multiple bindings. For example, imagine that you have a SOAP binding for the service and a local Java binding that lets you to treat the local service implementation (a Java class) as a Web service.

  The local Java binding for the service can only be used if the client is deployed in the same environment as the service itself, and if this is the case it is far more efficient to communicate with the service by making direct Java calls than using the SOAP binding.

  If your clients could switch the actual binding used based on run-time information, they could choose the most efficient available binding for each situation. In order to take advantage of Web services that offer multiple bindings, you need a service invocation mechanism that allows you to switch between the available service bindings at runtime, without having to generate or recompile a stub.
- **A freer Web services environment enables intermediaries.**

  Web services offer application integrators a loosely-coupled paradigm. In such environments, intermediaries can be very powerful. Intermediaries can add value to the service invocation without specific programming. Facilities such as logging, high-availability and transformation can be provided by a intermediary. WSIF is designed to make building intermediaries both possible and simple.

The goals of WSIF are therefore:

- To give a binding-independent mechanism for Web service invocation.
- To free client code from the complexities of any particular protocol used to access a Web service.
- To enable dynamic selection between multiple bindings to a Web service.

- To help the development of Web service intermediaries.

An overview of WSIF

WSIF provides a Java API for invoking Web services, independent of the format of the service or the transport protocol through which it is invoked. It addresses all of the issues identified in the goals of WSIF.

WSIF provides these features:

- It has an API that provides binding-independent access to any Web service.
- It is closely based on WSDL, so it can invoke any service that can be described in WSDL.
- It allows stubless (completely dynamic) invocation of a Web service.
- You can plug a new or updated implementation of a binding into WSIF at run time.
- You can defer the choice of a binding until run time.

WSIF is designed to work both in an unmanaged environment (standalone) and inside a managed container. You can use JNDI to find the WSIF service, or else read in the WSDL definition.

For more conceptual information about WSIF and WSDL, see these topics:

"WSIF and WSDL"
This topic compares the semantics of Web Services Description Language (WSDL) and WSIF.

"WSIF architecture" on page 54
This topic describes the WSIF architecture.

"Use WSIF with Web services that offer multiple bindings" on page 55
This topic describes how to use WSIF with Web Services with multiple bindings.

"WSIF usage scenarios" on page 55
This topic describes two brief scenarios that illustrate the role that Web Services Invocation Framework (WSIF) plays in the emerging Web services environment.

"Dynamic invocation" on page 55
This topic describes dynamic invocation of WSIF.

WSIF and WSDL: In Web Services Description Language (WSDL), a service is defined in three distinct parts:

- **The PortType** The PortType defines the abstract interface offered by the service. A PortType defines a set of operations. Each operation can be In-Out (request-response), In-Only, Out-Only and Out-In (Solicit-Response). Each operation defines the input and output messages. A message is defined as a set of parts, and each part has a schema-defined type.
- **The Binding** A binding defines how to map between the abstract PortType and a real service format and protocol. For example, the Simple Object Access Protocol (SOAP) binding defines the encoding style, the SOAPAction header, and the namespace of the body (the targetURI).
- **The Port.** This defines the actual location (endpoint) of the available service. For example, the HTTP URL on which a SOAP service is available.

Currently in WSDL, each Port has one and only one binding, and each binding has a single PortType. But each Service (PortType) can have multiple Ports, each of which represents an alternative location and binding for accessing that service.

Web Services Invocation Framework (WSIF) follows the semantics of WSDL as much as possible:

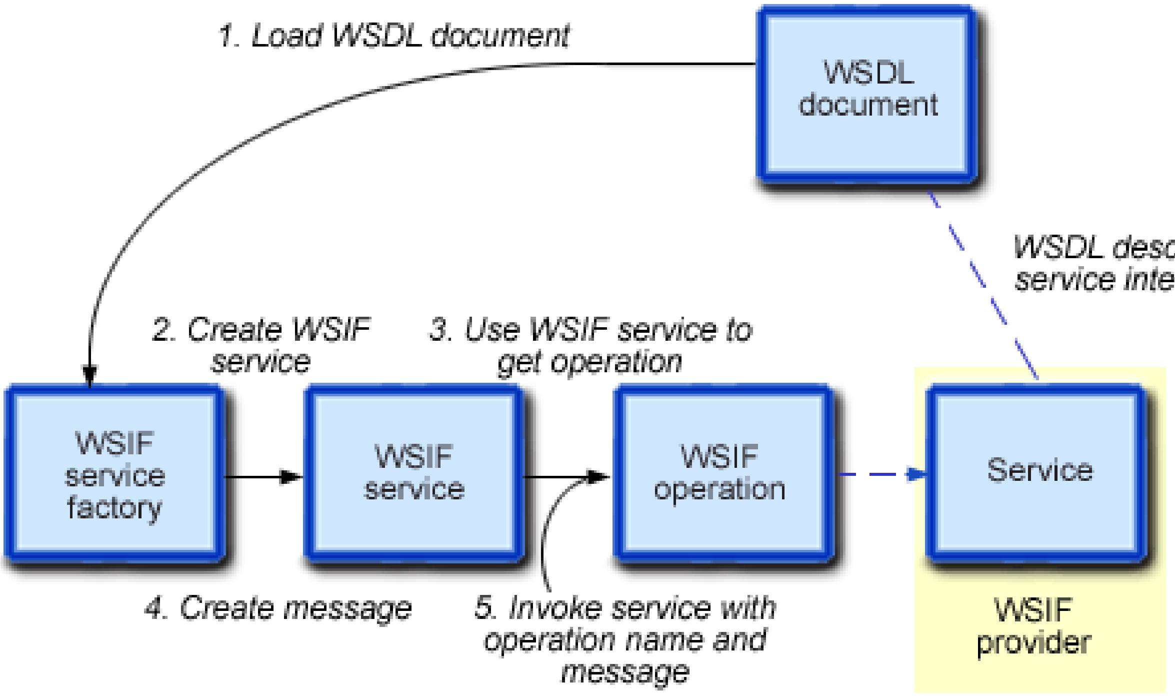
- The WSIF dynamic invocation API directly exposes run time equivalents of the model from WSDL. For example, invocation of an operation involves executing an operation with an Input Message.

- WSDL has extension points that allow new ports and bindings to be added so that WSDL can describe new systems. The equivalent concept in WSIF is a provider, that allows WSIF to understand a class of extensions, and therefore support new service implementation types.

As a metadata-based invocation framework, WSIF follows the design of the metadata. As WSDL is extended, WSIF is updated accordingly.

Note: The implicit and primary type system of WSIF is XML Schema, not Java. WSIF supports invocation using dynamic proxies, which support Java type systems, but when you use the WSIFMessage interface it is your responsibility to populate WSIFMessage objects with data based on the XML Schema types as defined in the WSDL document. You should define types of objects by a canonical and fixed mapping from schema types into the run time.

WSIF architecture: The WSIF architecture is shown in this figure. The components of this architecture are described after the figure.



WSIF architecture.
The WSIF architecture, shows a Web service invoked by loading a WSDL document, creating a WSIF service, using the service to get a WSIF operation, then invoking the target Web service by providing the WSIF operation with the target service operation's name and the message that it needs.

WSIF provider
A WSIF provider is an implementation of a WSDL binding that can run a WSDL operation through a binding-specific protocol. WebSphere Application Server - Express includes WSIF providers for SOAP over HTTP, and Java. For more information, see “Use the WSIF providers” on page 56.

WSIFOperation

The runtime representation of an operation, called WSIFOperation is responsible for invoking a service based on a particular binding.

WSIFService
The WSIFService is responsible for generating an instance of WSIFOperation to be used for a particular invocation of a service operation.

WSDL documents
The Web service WSDL document contains the location of the Web service. The binding document defines the protocol and format for operations and messages defined by a particular portType.

Use WSIF with Web services that offer multiple bindings: You can use WSIF to enable client applications to switch between service bindings at run time, to enable them to use the optimum binding, and to invoke operations on a Web service provider.

For example, a Web service provider could offer a SOAP binding for the service and a local Java binding that allows you to treat the local service implementation (a Java class) as a Web service. If the client is deployed in the same environment as the service, the local Java binding for the service can be used and provides more efficient communication with the service by making direct Java calls rather than using the SOAP binding.

WSIF usage scenarios: This topic describes two brief scenarios that illustrate the role that Web Services Invocation Framework (WSIF) plays in the emerging Web services environment.

Scenario: Redevelopment and redeployment

If you are implementing Web services today, you are probably working with simple prototypes. As your Web services move into production, you need to reimplement and redeploy them. WSIF uses the same API calls with different underlying technologies. If you use WSIF you can reimplement and redeploy your services without changing the client code, and you can use existing highly reliable and high-performance infrastructures like RMI-IIOP without sacrificing the location-independence that the Web service model offers.

Scenario: Service Flow composition

A service flow typically invokes a Web service, then passes the response from one Web service into the next Web service, perhaps performing some transformation in the middle.

There are two key aspects to this that WSIF provides:

- A representation of the service invocation based on the metadata in WSDL.
- The ability to build invocations based on the portType only, which can be used on any implementation.

For example, imagine that you build a meta-service that uses a number of services to build a process. Initially several of those services are simple JavaBean prototypes that are written and exposed through Simple Object Access Protocol (SOAP), but you plan to reimplement some of them as EJB components, and to out-source others.

If you use SOAP, it ties up multiple threads for every onward invocation, as they pass through the webserver and servlet engine into the SOAP router. If you use WSIF to call the beans directly, you get much better performance compared to SOAP, and you don’t lose access or location transparency. Using WSIF, you can move some of the Web services from local implementations to external SOAP services you just update the WSDL.

Dynamic invocation: In WSIF, dynamic invocation means providing these levels of support when invoking Web services:

1. Support, through the use of providers, for WSDL extensions and bindings that were not known at build time.

2. Support, by using the WSDL description to access the target service, for Web services that were not known at build time.

Use WSIF to invoke Web services

You invoke a Web service dynamically by using the WSIF API directly. You only specify the location of the WSDL file for the service, the name of the operation to be invoked, and any operation arguments needed. All the information needed to access the Web service is available through WSDL; the abstract interface, the binding, and the service endpoint.

This kind of invocation does not generate stub classes and does not need a separate compilation cycle.

More information on using WSIF to invoke Web services is given in these topics:

Pass SOAP messages with attachments using WSIF
This topic describes how to pass SOAP messages with attachments using WSIF.

“Use the WSIF providers”
This topic describes how to use these providers: the SOAP provider and the Java provider.

“Develop a WSIF service” on page 58
To develop a Web Services Invocation Framework (WSIF) service, you first develop the Web service. This topic describes how to develop a WSIF service.

“Use complex types” on page 58
This topic describes how to use complex types in your WSIF.

“Use JNDI” on page 59
This example task shows you how to use WSIF to bind a reference to a Web service, then look up the reference using JNDI.

“Interact with the WebSphere J2EE container” on page 60
This topic describes the interaction of WSIF with the J2EE container.

Use the WSIF providers: A WSIF provider is an implementation of a WSDL binding that can run a WSDL operation through a binding-specific protocol.

Providers implement the interface between the WSIF API and the actual implementation of a service. Providers are pluggable within the WSIF framework and are registered based upon the namespace of the WSDL extension that they implement.

WebSphere Application Server - Express includes these WSIF providers:

- **“Use the SOAP provider” on page 58**
- **“Use the Java provider”**

Note: Some providers use the J2EE programming model to utilize J2EE services. If a provider is available, but its required class libraries are not, the provider is disabled.

Use the Java provider: The WSIF Java Provider allows WSIF to invoke Java classes and JavaBeans. This means that in a thin-client environment, such as a Java virtual machine, you can define shortcuts to local Java code.

The WSIF Java Provider is not intended to be used in a J2EE environment. There is a difference between a client that uses the WSIF Java Provider to invoke a Java component and one that implements a Web service as a Java component on the server side.

The Java binding exploits the format binding for type mapping. The format binding allows WSDL to define the mapping between XML Schema types and Java types.

The Java provider requires the targeted Java classes to be in the class path of the client. The Java method is invoked synchronously, in-process, in-thread, with the current thread and ORB contexts.

The Java provider is not transactional.

The Java provider - writing the WSDL extension

The Java provider allows the invocation of a method on a local Java object. To use the Java provider, you require this binding specified in the WSDL:

Note: For legal information about this code example, see the Code example disclaimer.

```
<!-- Java binding -->
    <binding .... >
        <java:binding />
        <format:typeMapping style="Java" encoding="Java"/>?
            <format:typeMap name="qname" formatType="nmtoken"/>*
        </format:typeMapping>
        <operation>*
            <java:operation
                methodName="nmtoken"
                parameterOrder="nmtoken"
                returnPart="nmtoken"?
                methodType="instance|constructor" />
            <input name="nmtoken"? />?
            <output name="nmtoken"? />?
            <fault name="nmtoken"? />?
        </operation>
    </binding>
```

where *?* means optional and * means 0 or more.

Notes:

- The format:typeMap name attribute is a qualified name of a simple or complex type used by one of the Java operations.
- The format:typeMap formatType attribute is the fully qualified Class name for the Java Class that the element specified by name maps to.
- The java:operation methodName attribute is the name of the method on the Java object that is called by the operation.
- The java:operation parameterOrder attribute contains a whitespace-separated list of part names that define the order in which they are passed to the Java Object's method.
- The java:operation methodType attribute must be set to either instance or constructor. The value specifies whether the method being invoked on the object is an instance method or a constructor for the object.

```
<service ... >
        <port>*
            <java:address
                className="nmtoken"/>
        </port>
   </service>
```

Note: The java:address className attribute specifies the fully qualified class name of the object containing the method to invoke.

Use the SOAP provider: The SOAP provider allows WSIF stubs and dynamic clients to invoke SOAP services. The provider supports SOAP 1.1 over HTTP. The WSIF SOAP Provider utilizes ApacheSOAP 2.3 to parse and to create SOAP messages but is not limited to invoking services served by ApacheSOAP.

The WSIF SOAP provider supports:

- SOAP-ENC encoding
- RPC style

The SOAP provider is not transactional.

Note: Before you deploy a Web service that you expect to be used by multiple clients connecting over SOAP to WebSphere Application Server - Express, you must set up your application's deployment descriptor file (dds.xml) to handle multiple connections correctly.

Develop a WSIF service: To develop a Web Services Invocation Framework (WSIF) service, you first develop the Web service (or use an existing Web service), then develop the WSIF client based on the WSDL document for that Web service.

To develop a WSIF service, complete the following steps:

1. **Develop the Web service.**
   Use Web services tools to discover, create, and publish the Web service. You can develop Java bean and URL Web services. You can use Web service tools to create a skeleton Java bean and a sample application from a WSDL document. For example, you can use a Java class as a Web service, with local Java invocations as the access protocol.
2. **Develop the WSIF client.**

Use complex types: WSIF supports the use of user defined complex types through the mapping of complex types to Java classes. This mapping must be specified by the user. The method to use to create these mappings depends on the provider being used. For the Java provider, the mappings are specified in the wsdl file in the binding element. The syntax for specifying the mapping is as follows:

```
<binding .... >
        <ejb:binding|java:binding/>
          <format:typeMapping style="Java" encoding="Java"/>?
              <format:typeMap name="qname" formatType="nmtoken"/>*
          </format:typeMapping>
     ...
    </binding>
```

where *?* means optional and * means 0 or more.

The format:typeMap name attribute is a qualified name of a complex type or simple type used by one of the operations.

The format:typeMap formatType attribute is the fully qualified Class name for the Java Class that the element specified by name maps to.

If using the Apache SOAP provider then the mapping of a complex type to a Java Class is specified in the client code through two methods on the org.apache.wsif.WSIFService interface:

```
public void mapType(QName elementType, Class javaType)
```

and

```
public void mapPackage(String namespaceURI, String packageName)
```

The mapType allows you to specify a mapping between a WSDL element and method takes a QName representing the complex type or simple type and the corresponding Java Class it maps to.

The mapPackage method allows you to specify a more general mapping between a namespace and a Java package. Any custom complex or simple types whose namespace matches that of the mapping is mapped to a Java Class in the corresponding package. The name of the actual class is derived from the name of the complex type using standard xml to Java naming conventions.

Use JNDI: This example task shows you how to use WSIF to bind a reference to a Web service, then look up the reference using JNDI.

You access a Web service through information given in the WSDL document for the service. If you do not know where to find the WSDL document for the service, but you know that it has been registered in a UDDI registry, you look it up in the registry. Java programs access java objects and resources in a similar manner, but using a JNDI interface.

The following example shows how, using WSIF, you can bind a reference to a Web service then look up the reference using JNDI.

Specifying the argument values for the Web Service

The Web service is represented in WSIF by an instance of the org.apache.wsif.naming.WSIFServiceRef class. This simple Referencable object has the following constructor:

```
public WSIFServiceRef(
        String WSDL,
        String sNS,
        String sName,
        String ptNS,
        String ptName)
{
    [...]
}
```

where

- *WSDL* is the location of the WSDL file containing the definition of the service.
- *sNS* is the full namespace for the service definition (null can be specified if only one service is defined in the WSDL file).
- *sName* is the local name for the service definition (null can be specified if only one service is defined in the WSDL file).
- *ptNS* is the full namespace for the port type within the service that you want to use (null can be specified if only one port type is available for the service).
- *ptName* is the local name for the port type (null can be specified if only one port type is available for the service).

For example, if the WSDL file for the Web service is available from the URL http://localhost/WSDL/Example.WSDL and contains these service and port type definitions -

```
<definitions targetNamespace="http://hostname/namespace/example"
              xmlns:abc="http://hostname/namespace/abc"
[...]
    <portType name="ExamplePT">
      <operation name="exampleOp">
        <input name="exampleInput" message="tns:ExampleInputMsg"/>
      </operation>
    </portType>
[...]
    <service name="abc:ExampleService">
[...]
    </service>
[...]
  </definitions>
```

then you specify these argument values for WSIFServiceRef:

- WSDL is http://localhost/WSDL/Example.WSDL
- sNS is http://hostname/namespace/abc
- sName is ExampleService
- ptNS is http://hostname/namespace/example
- ptName is ExamplePT
- Binding the service using JNDI

To bind the service reference in the naming directory using JNDI, you can use the WebSphere Application Server - Express JndiHelper com.ibm.websphere.naming.JndiHelper class as follows:

```
[...]
    import com.ibm.websphere.naming.JndiHelper;
    import org.apache.wsif.naming.*;
[...]
    try {
       Context startingContext = new InitialContext();
       WSIFServiceRef ref = new WSIFServiceRef("http://localhost/WSDL/Example.WSDL,
                                         "http://localhost/WSDL/Example.WSDL",
                                         "http://hostname/namespace/abc"
                                         "ExampleService",
                                         "http://hostname/namespace/example",
                                         "ExamplePT");
       JndiHelper.recursiveRebind(startingContext, "myContext/mySubContext/myServiceRef", ref);

    }
    catch (NamingException e) {
       // Handle  error.
    }
[...]
```

Looking up the service using JNDI

This code fragment shows the lookup of a service using JNDI:

Note: For legal information about this code example, see the Code example disclaimer.

```
[...]
    try {
[...]
       InitialContext ic = new InitialContext();
       WSIFService myService = (WSIFService) ic.lookup("myContext/mySubContext/myServiceRef");
[...]
    }
    catch (NamingException e) {
       // Handle error.
    }
[...]
```

Interact with the WebSphere J2EE container: Interaction with a container is limited to these aspects:

1. The WebSphere administrative console and WCCM allow users to define Web services to WebSphere. As part of the definition of a service, the administrator may define a preferred port.
2. WSIF makes log and trace calls to the WebSphere Server JRAS services.
3. Some providers use the J2EE programming model to utilize J2EE services.
4. WSIF wraps the use of container services so that when WSIF is run in an unmanaged (thin) environment, the operation can succeed.

WSIF system management and administration

WSIF is provided as a standalone JAR file called wsif.jar. The JAR file contains the core WSIF classes, and the Java and SOAP over HTTP providers. Additional providers are packaged as separate JAR files.

When you install WebSphere Application Server - Express, wsif.jar is put on the WebSphere or Java virtual machine class path.

WSIF requires no further configuration. WSIF is a thin abstraction layer between application code and the relevant invocation infrastructure.

Maintaining the WSIF properties file

WSIF properties are stored in a properties file (in wsif.jar) called wsif.properties. This file is kept on the class path, so that WSIF can find it, and the client administrator can use it to configure WSIF.

Here are the initial contents of wsif.properties. All the possible properties are listed and described.

```
# Two properties are used to override which WSIFProvider is selected when there
# exists multiple providers supporting the same namespace URI. These properties are:
#
#    wsif.provider.default.CLASSNAME=N
#    wsif.provider.uri.M.CLASSNAME=URI
#
# CLASSNAME is the WSIFProvider class name
# N is the number of following default wsif.provider.uri.M.CLASSNAME properties
# M is a number from 1 to N to uniquely identify each wsif.provider.uri.M.CLASSNAME
#   property key.
# For example the following two properties would override the default SOAP provider
# to be the Apache SOAP provider:
#
# wsif.provider.default.org.apache.wsif.providers.soap.apacheaxis.WSIFDynamicProvider_ApacheAxis=1
# wsif.provider.uri.1.org.apache.wsif.providers.soap.apacheaxis.WSIFDynamicProvider_ApacheAxis=\
# http://schemas.xmlsoap.org/wsdl/soap/
#

# maximum number of milliseconds to wait for a response to a synchronous request.
# Default value if not defined is to wait forever.
wsif.syncrequest.timeout=10000

# maximum number of seconds to wait for a response to an async request.
# if not defined on invalid defaults to no timeout
wsif.asyncrequest.timeout=60
```

Enabling security for WSIF

This is how WSIF interacts with a security manager:

- WSIF runs in the current J2EE security context without modifying it.
- When WSIF is run under a J2EE container, Port implementations can utilize security context to pass on security tokens or credentials as necessary.
- WSIF implementations can automatically convert J2EE security context into appropriate context for onward services.

For WSIF to interact effectively with the WebSphere Application Server - Express's security manager, these permissions must be set in the server.policy file:

- FilePermission to load the WSDL (this is only required when a WSDL file is referred to using the file:/// protocol)
- RuntimePermission "getClassLoader" for the current thread's context class loader.
- RuntimePermission "accessDeclaredMembers" (this is required by both portions)
- PropertyPermission for system properties (this is required by SOAP and many others; read and write access is required for the SOAP and Java portion)
- NetPermission "specifyStreamHandler" (this must be in either the SOAP and Java portion, but it need not be in both).
- SocketPermission "*host_name*", "resolve" (this is not required by the SOAP and Java portion)

- SocketPermission "*host_name*:*port_no*", "connect" (this is required by both portions)

where *host_name* is your host name (for example localhost), and *port_no* is your port number (for example 9080).

Assemble Web services

Before you can deploy your Web services application, you must *assemble* (or package) the application. If you are using a development tool such as WebSphere Development Studio Client for iSeries, the tool automatically performs much of the assembly process for you. You need only specify any necessary assembly properties. You can then export your application EAR file for deployment. See your product documentation for more information.

You can also manually assemble your Web services application with command-line tools.

See the following topics for information about packaging your Web services application into a WAR or JAR file:

"Web services assembly properties" (Version 5.0.2 or later)
See this topic for a list of assembly properties for Web services applications.

"Assemble a WAR file for your Web services application" on page 64 (Version 5.0.2 or later)
This topic describes how to package your Web services application into a Web archive (WAR) file. If you are using a development tool, such as WebSphere Development Studio Client for iSeries, that automatically creates a WAR file for you, you can skip this step.

"Assemble a Web services client" on page 65 (Version 5.0.2 or later)
This topic describes how to assemble your Web services client application.

Use the WebSphere Development Studio Client for iSeries (or other development tool) to assemble the Web services-enabled WAR file into an EAR file. The EAR file can contain Web applications (WAR files) and metadata that describes the applications (application.xml files).

For information on assembling SOAP Web services, see "Deploy a programming component as a SOAP accessible Web service" on page 38 (Versions 5.0 and 5.0.1 only)

Web services assembly properties

ibm-webservices-bnd.xmi properties

The ibm-webservices-bnd.xmi file is a deployment descriptor for a Web Services-enabled Web module. It contains information for the Web services runtime that is either WebSphere product-specific or was not included in the Web services specification.

You can manually edit these assembly properties or use WebSphere Development Studio Client for iSeries (Version 5.1 or later).

To use the Development Studio client, open the webservices.xml file for your Web service module in the Web Services Editor. The assembly properties are located on the **Bindings** and **Binding Configurations** tabs.

Note: Although you open the webservices.xml file for editing, the Web Services Editor stores the values in the ibm-webservices-bnd.xmi file in your module.

The following user-definable assembly properties are supported:

- **wsDescNameLink**
  Attribute of the wsdescBindings element that specifies the link to the corresponding <webservice-description-name> in webservices.xml.
- **pc-name-link**
  Attribute of the pcBindings element that specifies the link to the <port-component-name> in the webservices.xml file.
- **scope**
  Attribute of the pcBindings element that specifies when new instances of implementation beans are created. Possible values are Request, Session, and Application. The value of scope for a deployed Web service can be changed using the administrative console. Navigate to the Web module of the Web service application and select **Web Services Implementation Scope**.

ibm-webservicesclient-bnd.xmi properties

The ibm-webservicesclient-bnd.xmi file contains information for the Web services run time that is WebSphere product-specific. The ibm-webservicesclient-bnd.xmi file is used for Web services clients or Web services that act as clients to another Web service.

You can manually edit these assembly properties or use WebSphere Development Studio Client for iSeries (Version 5.1 or later).

To use the Development Studio client, open the webservicesclient.xml file for your Web service module in the Web Services Client Editor. The assembly properties are located on the **Bindings** and **Port Bindings** tabs.

Note: Although you open the webservicesclient.xml file for editing, the Web Services Editor stores the values in the ibm-webservicesclient-bnd.xmi file in your module.

The following user-definable assembly properties are supported:

- **componentNameLink**
  Attribute of the componentScopedRefs element that specifies the link to the corresponding <component-scoped-refs> element in webservicesclient.xml file.
- **serviceRefLink**
  Attribute of the serviceRefs element that specifies the link to the <service-ref-name> in the webservicesclient.xml file.
- **deployedWSDLFile**
  Attribute of the serviceRefs element is optional and permits an alternate WSDL file to be used other than that specified in the <wsdl-file> element of webservicesclient.xml file. If this attribute is specified, the alternate WSDL file must be packaged in the same module and must be compatible with the development WSDL file. The deployedWSDLFile property is used to supply a new WSDL file containing a different endpoint URL than the original WSDL file.
- **defaultMappings element**
  Identifies which port should be used for a given portType when none is explicitly selected by the client. This element has the following attributes: portTypeNamespace, portTypeLocalName, portNamespace, portLocalName. These attributes identify which wsdl:port should be used for a wsdl:portType.
- **syncTimeout**
  Attribute of the portQnameBindings element that specifies how long, in seconds, to wait for a response from a synchronous call.
- **basicAuth**
  Element of the portQnameBindings element that can be used to authenticate a service client to the service endpoint, independent of the underlying transport that includes HTTP and HTTPS. Set the user ID and password attributes as needed.

- **sslConfig**
  Element of the portQnameBindings element that specifies the Secure Sockets Layer (SSL) configuration of an HTTPS outbound request. The name attribute is the name of a SSL configuration entry or alias defined in the SSL Configuration Repertoire.

  Note: This attribute is only used when the client is running in a WebSphere application server.

The values of deployedWSDLFile and the defaultMappings of a deployed Web service can also be changed using the administrative console. Using application management, navigate to the Web module and select **Web Services Client Bindings**.

Example bindings files

The following examples demonstrate the spelling and position of the various attributes. You cannot cut and paste these examples because they do not contain the required ID attributes. If you add elements to a binding file template generated by the WSDL2Java command, you must confirm that each element has an ID attribute whose value is a unique string. Review the template xmi files generated by the WSDL2Java command for examples of ID strings.

Example ibm-webservices-bnd.xmi file

```
<com.ibm.etools.webservice.wsbnd:WSBinding xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI" xmlns:com.ibm.etools.webservice.wsbnd=
    "http://www.ibm.com/websphere/appserver/schemas/5.0.2/wsbnd.xmi">
  <wsdescBindings wsDescNameLink="AddressBookService">
    <pcBindings pcNameLink="AddressBook" scope="Application"/>
  </wsdescBindings>
</com.ibm.etools.webservice.wsbnd:WSBinding>
```

Example ibm-webservicesclient-bnd.xmi file

```
<com.ibm.etools.webservice.wscbnd:ClientBinding xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI" xmlns:com.ibm.etools.webservice.wscbnd=
    "http://www.ibm.com/websphere/appserver/schemas/5.0.2/wscbnd.xmi">
  <componentScopedRefs componentNameLink="myComponent ref"/>
  <serviceRefs serviceRefLink="myService ref" deployedWSDLFile="META-INF/wsdl/alternate.wsdl">
    <defaultMappings portTypeLocalName="AddressBook"
        portTypeNamespace="http://www.com.ibm" portLocalName="AddressBookPort"
        portNamespace="http://www.com.ibm"/>
      <portQnameBindings portQnameNamespaceLink="http://www.com.ibm"
          portQnameLocalNameLink="AddressBookPort" syncTimeout="99">
        <basicAuth userid="myId" password="myPassword"/>
        <sslConfig name="mynode/DefaultSSLSettings"/>
      </portQnameBindings>
    </serviceRefs>
  </com.ibm.etools.webservice.wscbnd:ClientBinding>
```

Assemble a WAR file for your Web services application

This topic explains how to use the command-line tools to assemble a Web service-enabled WAR file.

If you converted an existing application in a Web service, perform the following steps to assemble a Web services-enabled WAR file:

1. Expand the WAR file into a directory.
2. Confirm that the WEB-INF/web.xml descriptor for the Web module contains a <servlet-class> element indicating the Java bean class that implements the service.
3. Place the WSDL file as specified by the deployment descriptor <wsdl-file> element of webservices.xml file in the WEB-INF/wsdl subdirectory.
4. Place the JAX-RPC mapping file as specified by the deployment descriptor <jaxrpc-mapping-file> element of webservices.xml in the WEB-INF subdirectory.

5. Place the webservices.xml and ibm-webservices-bnd.xmi deployment descriptors in the WEB-INF subdirectory.
6. (Optional) If you developed a service endpoint interface class, place it in a subdirectory corresponding to its Java package.
7. Run this command to add these files to the WAR file:
   ```
   jar -uvf WAR_file com WEB-INF/*
   ```

If you developed a new Web service application, perform the following steps to assemble a Web services-enabled WAR file:

1. Expand the WAR file into a directory.
2. Confirm that the WEB-INF/web.xml deployment descriptor for the Web module contains a <servlet> element including the <servlet-name> element. The <servlet-name> element can be any string and the <servlet-class> element specifies the Java bean class that implements the service.
3. Place the WSDL file as specified by the webservices.xml deployment descriptor <wsdl-file> element in the WEB-INF/wsdl subdirectory.
4. Place the JAX-RPC mapping file as specified by the webservices.xml deployment descriptor <jaxrpc-mapping-file> element in the WEB-INF subdirectory.
5. Place the webservices.xml and ibm-webservices-bnd.xmi deployment descriptors in the WEB-INF subdirectory.
6. Run this command to add these files to the WAR file:
   ```
   jar -uvf WAR_file com WEB-INF/*
   ```

Assemble a Web services client

The steps in this topic explain how to use the command-line tools to assemble a Web service-enabled client application.

To assemble a Web services client in a Web archive (WAR) file, follow these steps:

1. Expand the WAR file in the root directory.
2. Place the WSDL file in the WEB-INF/wsdl subdirectory.
3. Place the webservicesclient.xml and JAX-RPC mapping files in the WEB-INF subdirectory.
4. (Optional) If you use the ibm-webservicesclient-bnd.xmi file, place it in the WEB-INF subdirectory.
5. Add the files to the existing WAR file with the appropriate command for your platform.
   - On iSeries, run `jar -uvf existing.war WEB-INF/*` from Qshell.
   - For UNIX platforms, run `jar -uvf existing.war WEB-INF/*`.
   - For Windows platforms, run `jar -uvf existing.war WEB-INF\*`.

After you assemble the client application, test it to verify that it runs correctly.

Deploy Web services

You can use either the HTTP Server Administration interface, the administrative console or the wsadmin scripting interface to deploy a J2EE Web service.

Note: If the Web services in the application is previously deployed with the wsdeploy command, it is not necessary to specify Web services deployment during installation.

Use the HTTP Server Administration interface

To deploy the Web services-enabled application with the HTTP Server Administration interfaces, follow the installation process in Deploy and start a new application.

Use the administrative console

To deploy the Web services-enabled application with the administrative console, follow the installation process in Install and uninstall applications with the WebSphere administrative console. In step 1 of the **Install New Application** wizard, select **Deploy WebServices**.

Use wsadmin

To deploy the EAR file with wsadmin, follow these steps:

1. Start wsadmin.
2. At the wsadmin prompt, run the $AdminApp install command. You must specify the -deployws parameter. In this example, myWSApp is the name of the Web services-enabled EAR file:

   ```
   $AdminApp install myWSApp "-usedefaultbindings -deployws"
   ```

For information on deploying SOAP Web services, see “Deploy a programming component as a SOAP accessible Web service” on page 38 (Versions 5.0 and 5.0.1 only)

Configure Web services

See these topics for information about configuring Web services application and components:

“Web services tools”
This topic describes tools that you can use for Web services.

“Configure Web services security (Version 5.0.2 or later)” on page 77
This topic describes how to secure your Web services applications.

Web services tools

See these topics for tools you can use to administer Web services development:

WebSphere Development Studio Client Version 5.1
WebSphere Development Studio Client Version 5.1 includes a Web Services Description Language editor and other advanced Web services support. For more information, see the WebSphere Development Studio Client Version 5.1 documentation.

“Web services scripts”
WebSphere Application Server - Express ships with several scripts that you can use to develop Web services applications.

“Publish Web Services Description Language files” on page 74
See this topic for information about publishing your WSDL files.

Web services scripts

See the following topics for information about running the WebSphere Application Server - Express scripts for Web services:

“The Java2WSDL script” on page 67
The Java2WSDL command accepts a Java class as input and produces a WSDL file that represents the input class.

“The WSDL2Java script” on page 69
The WSDL2Java command tool creates Java classes and deployment descriptor templates from a WSDL file.

"The wsdeploy script" on page 72
The wsdeploy command line tool adds Websphere Application Server product-specific Web services deployment classes to a Web services enterprise archive (EAR) file or an application client Java archive (JAR) file.

"The setupWebServiceClientEnv script" on page 73
The setupWebServiceClientEnv script sets up a Java environment for Web services J2SE clients to use and sets the classpath variable for Web Services clients.

The Java2WSDL script: The Java2WSDL command tool maps a Java class to a Web Services Description Language (WSDL) file by following the Java API for XML-based remote procedure call (JAX-RPC) specification. The Java2WSDL command accepts a Java class as input and produces a WSDL file representing the input class. If there is an existing file at the output location, it is overwritten. The WSDL file generated by the Java2WSDL command contains WSDL and XML schema constructs that are automatically derived from the input class. You can override these default values with command-line arguments.

The WSDL file generated by the Java2WSDL command can contain unexpected elements. You can create WSDL files that cannot be compiled when regenerated into Java code using the WSDL2Java command because the JAX-RPC mapping from Java to WSDL is not reversible back to the original Java code. Inspect and modify the WSDL file if you encounter this problem.

Authority

To run this script, your user profile must have *RX authority.

Syntax

The syntax of the script is:

```
Java2WSDL class [argument...]
```

Parameters

The parameters of the script are:

- ***class***
  This is a required parameter. The value *class* represents the fully qualified name of one of the following Java classes:
  - Service Endpoint Interface that extends the java.rmi.Remote class
  - Java bean

  The Java2WSDL command locates the class in CLASSPATH.
- **-bindingName**
  This is an optional parameter. The value *name* specifies the name to use for the binding element. If not specified, the binding name is the portTypeName.
- **-help**
  Displays the help message.
- **-helpX**
  Displays the help message for extended options.
- **-debug**
  Displays debug messages.
- **-outputImpl**
  This is an optional parameter. The value *impl-wsdl* specifies if you want an interface and implementation WSDL file emitted.

- **-locationImport**
  This is an optional parameter. The value *location-uri* specifies the location of the interface WSDL file if you use the -outputImpl argument specified.
- **-MIMEStyle**
  This is an optional parameter. The value *style* specifies a style representing Multipurpose Internet Mail Extensions (MIME) information. Valid values are Axis and WSDL11. The default value is WSDL11.
- **-soapAction**
  This is an optional parameter. Valid arguments are:
  - **DEFAULT**
    Sets the soapAction field according to deployment information.
  - **NONE**
    Sets the soapAction field to "".
  - **OPERATION**
    Sets the soapAction field to the operation name.
- **-stopClasses**
  This is an optional parameter. If the -all argument is specified, the Java2WSDL command searches inherited classes and interfaces to construct the list of methods for WSDL file operations. The Java2WSDL command searches inherited classes and interfaces when generating extended complexTypes. The search stops when a class or interface is found within a package that begins with java or javax. The value *parent* specifies an additional class that cause the search to stop. To specify multiple classes, separate them with commas:

  ```
  -stopClasses class1,class2
  ```
- **-namespaceImpl**
  This is an optional parameter. The value *namespace* specifies the target namespace for the implementation WSDL if -outputImpl specified.
- **-voidReturn**
  Valid arguments are:
  - **ONEWAY**
    Methods with void returns are one-way.
  - **TWOWAY**
    Methods with void returns are two-way. This the default for HTTP transport.
- **-wrapped**
  This is an optional parameter. The value *boolean* specifies if the WSDL file should be generated according to wrapped rules. This is only valid if the value of the -use parameter is LITERAL. The option defaults to true.
- **-extraClasses**
  This is an optional parameter. The value *classes* specifies other classes that should be represented in the WSDL file.
- **-input**
  This is an optional parameter. The value *wsdl-uri* specifies the input WSDL file used to build an output WSDL file. Information from an existing WSDL file, whose name is specified in this option, is used with the input Java class to generate the desired output.
- **-implClass**
  This is an optional parameter. The Java2WSDL command uses method parameter names to construct the WSDL file message part names. The command automatically obtains the message names from the debug information in the class. If the class is compiled without debug information, or if the class is an interface, the method parameter names are not available. The value *impl-class* specifies an alternative class from which to obtain method parameter names. The specified class does not need to implement the class if the class is an interface, but it must implement the same methods as class.
- **-location**
  This is an optional parameter. The value *location* specifies the location or Uniform Resource Locator (URL) of the service. Typically, this value fills automatically when the Web service deploys. Use this

argument to specify the location if you want to generate a WSDL file containing a location URL without deploying. A warning displays to remind you that the generated WSDL file should not be published if the final location is not yet been determined. The name after the last slash or backslash is the name of the service port, unless the name is overridden by the -servicePortName argument. The service port address location attribute is assigned the specified value.

- **-namespace**
  This is an optional parameter. The value *targetNamespace* specifies the target namespace for the WSDL file being generated.
- **-output**
  This is an optional parameter. The value *wsdl-uri* specifies the path and file name of the output WSDL file. If not specified, the default file, class.wsdl, is written into the current directory.
-
- **-PkgtoNS**
  This is an optional parameter. If you specify this parameter, the script maps the package specified by *package* to the namespace specified by *namespace*. If there is a package without a namespace, the Java2WSDL command generates a namespace name. Specify this parameter once for each mapping.
- **-portTypeName**
  This is an optional parameter. The value *name* specifies the name to use for the portType element. If not specified, the class name is used.
- **-serviceElementName**
  This is an optional parameter. The value *name* specifies the name of the service element.
- **-servicePortName**
  This is an optional parameter. The value *name* specifies the name of the service. If not specified, the service name is derived from the -location parameter.
- **-style**
  This is an optional parameter. The value *style* specifies the WSDL style to use in the generated WSDL file. Valid values are RPC and DOCUMENT. This parameter is used with the -use parameter.
  - If RPC is specified with -use ENCODED, or omitting use, a style=rpc/use=encoded WSDL file is generated.
  - If RPC is specified with -use LITERAL, a style=rpc/use=literal WSDL file is generated.
  - If DOCUMENT is specified with -use LITERAL or omitting use, a style=document/use=literal WSDL file is generated.

  For more information about styles, see "Map between Java, WSDL, and XML" on page 13.
- **-transport**
  This is an optional parameter. The value *transport_type* specifies the type of tranpsort for which the script generates Simple Object Access Protocol (SOAP) bindings. The default value is `http`. You can specify the transport option only once.
- **-use**
  This is an optional parameter. The value *use* specifies the `use` that is generated into the WSDL file. Valid values are LITERAL and ENCODED. This parameter is used with the -style parameter.

  For more information, see "Map between Java, WSDL, and XML" on page 13.
- **-verbose**
  Displays verbose messages.

The WSDL2Java script: The WSDL2Java command tool creates Java classes and deployment descriptor templates from a Web Services Description Language (WSDL) file using the Java API for XML-based remote procedure call (JAX-RPC) 1.0 specification. See Mapping between Java, WSDL and XML for more information.

Classes and files generated

The WSDL2Java script generates these kinds of classes and files:

- For each portType in the WSDL document (<wsdl:portType> element tag):
  - Service Endpoint Interface
- For each service in the WSDL document (<wsdl:service> element tag):
  - Service Interface when the -role develop-client argument is specified.
  - ServiceLocator when the -role deploy-client argument is specified.
    This class is a WebSphere product-specific implementation of the service interface, and is not used directly.
  - webservices.xml deployment descriptor template when the -role develop-server argument is specified.
  - ibm-webservices-bnd.xmi deployment descriptor template when the -role develop-server argument is specified.
  - ibm-webservices-ext.xmi deployment descriptor template when the -role develop-server argument is specified.
  - wsdlfile_mapping.xml JAX-RPC mapping file when the -role develop-client or -role develop-server is specified.
  - webservicesclient.xml deployment descriptor template when the -role develop-client argument is specified.
  - ibm-webservicesclient-bnd.xmi deployment descriptor template when the -role develop-client argument is specified.
  - ibm-webservicesclient-ext.xmi deployment descriptor template when the -role develop-client argument is specified.

  When the role is a server role, the container argument specifies which J2EE container the implementation uses. When the -role develop-server -container web arguments are specified, the files are generated into the WEB-INF directory.
- For each binding in the WSDL file (<wsdl:binding> element tag):
  - A stub generates that implements the Service Endpoint Interface(deploy-client role).
  - An implementation template for the Java bean generates when the -role develop-server and -container-web arguments are specified.
- Other classes and files:
  - A Java bean representing the structure of the type when the -role develop-server or -role develop-client arguments are specified for each complexType or simpleType.
  - Three classes, *_Ser.java, *_Deser.java, and *_Helper.java, generate for each complexType to assist in converting the bean to Simple Object Access Protocol (SOAP) and back when the -role deploy-server or -role deploy-client argument is specified.
  - A *Holder.java class generates when the -role develop-server or -role develop-client arguments are specified for each out and inout parameter.

Authority

To run this script, your user profile must have *RX authority.

Syntax

The syntax of the script is:

```
WSDL2Java WSDL-URI [arguments]
```

Parameters

The parameters of the script are:

- ***WSDL-URI*** Specifies the location of the input WSDL document using a Universal Resource Identifier (URI). You can also use a regular file path if the WSDL file is on the local file system.

- **-container**
  This is an optional parameter. The value *j2ee-container* specifies the J2EE container to be used. Valid values are client, web, and none. If client is role, the default argument is none. If server is role, the container must be web. The same container option must be used for both development and deployment.
- **-deployScope**
  This is an optional parameter. The value *scope* specifies how to deploy the server implementation. These are the valid values:
  - Application - Uses one instance of the implementation class for all requests.
  - Request - Creates a new instance of the implementation class for each request.
- **-genResolver**
  Generates an absolute-import resolver class. The purpose of this class is to record the contents of the imported WSDL files used by the WSDL URI. This class is used by the runtime. It can also be used for future WSDL2Java command runs. This is desirable when the imported WSDL files are remote and can be inaccessible or slow to access. It also eliminates the possibility that a remote WSDL file might have different contents at run time than it did at development time. The generated class is named _AbsoluteImportResolver.java. You should compile and package this class with the other Java classes generated by the WSDL2Java command.
- **-help**
  Displays a help message.
- **-helpX**
  Displays a help message for extended options.
- **-inputMappingFile**
  This is an optional parameter. The value *mapfile* specifies the file name of the Java to WSDL mapping file.
- **-NStoPkg**
  This is an optional parameter. If you specify this parameter, the script maps the namespace specified by *namespace* to the package specified by *package*. Specify this parameter once for each unique namespace mapping. For example, if there is a namespace in the WSDL file called urn:AddressFetcher2, and you want files generated from the objects in this namespace to reside in the package samples.addr, provide the -NStoPkg urn:AddressFetcher2=samples.addr argument to the WSDL2Java command. By default, package names are automatically derived from the namespace strings in the WSDL file. For example, if the namespace is of the form http://x.y.com or urn:x.y.com, the corresponding package is com.y.x.

  Note: The default XML namespace to Java package mapping does not take the context root into account. If two namespaces are the same up to the first slash, they are mapped to the same Java package. For example, the XML namespaces http://www.ibm.com/foo and http://www.ibm.com/bar both map to the Java package com.ibm.www. Use the -NStoPkg option to specify the package for the fully qualified namespace.
- **-output**
  This is an optional parameter. The value *directory* sets the root directory for the files that the script generates.
- **-role**
  This is an optional parameter. The value *j2eeRole* specifies the J2EE development role that identifies which files to generate. Valid values are:
  - **client**
    Combination of develop-client and deploy-client.
  - **deploy-client**
    Generates binding files for client deployment.
  - **deploy-server**
    Generates binding files for server deployment.

  - **develop-client** (default)
    Generates files for client development. This is the default value.
  - **develop-server**
    Generates files for server development.
  - **server**
    Combination of develop-server and deploy-server.
- **-timeout**
  This is an optional parameter. The value *seconds* specifies how long the WSDL2Java command should wait, in seconds, for the WSDL-URI to respond before giving up. The default is 45 seconds, -1 disables the timeout.
- **-useResolver**
  This is an optional parameter. The value *resolver-class* specifies an absolute-import resolver class to use during parsing. This class must have been created during a previous execution of the WSDL2Java command using the -genResolver option. The class must be available in CLASSPATH.
- **-verbose**
  Displays processing information, including the names of the generated files.

The wsdeploy script: The wsdeploy command line tool adds Websphere Application Server - Express product-specific Web services deployment classes to a Web services enterprise archive (EAR) file. These classes include:

- Stubs
- Serializers and deserializers
- Implementations of service interfaces

This deployment step must be performed at least once, and can be performed more than once. Deployment can be performed separately using the wsdeploy command or when the application is installed. When using the wsadmin command for installation, specify the -deployws option. When using the administrative console for installation, select the Deploy Web services check box.

The wsdeploy command operates as follows:

1. Each module in the enterprise application is examined.
2. If the module contains Web services implementations, indicated by the presence of the webservices.xml deployment descriptor, the associated Web Services Description Language (WSDL) files are located and the WSDL2Java command is run with the role deploy-server.
3. If the module contains Web services clients, indicated by the presence of the webservicesclient.xml deployment descriptor, the associated WSDL files are located and the WSDL2Java command is run with the role deploy-client.
4. The files generated by the WSDL2Java command are compiled and repackaged.

See WSDL2Java command for more information about the files that are generated for deployment.

When the generated files are compiled, they can reference application-specific classes outside the EAR if the EAR is not self-contained. In this case, use -cp option to specify additional zip files to be added to CLASSPATH when the generated files are compiled.

Authority

To run this script, your user profile must have *RX authority.

Syntax

The syntax of the script is:

```
wsdeploy Input_filename Output_filename [options]
```

Parameters

The parameters of the script are:

- ***Input_filename***
  Specifies the path to the EAR to be deployed.
- ***Output_filename***
  Specifies the path of the deployed EAR file. If output_filename already exists, it is silently overwritten. The output_filename can be the same as the input_filename.
- **-jardir**
  This is an optional parameter. The value *directory* specifies a directory containing zip files. All zip files in this directory are added to the CLASSPATH used to compile the generated files. This option can be specified zero or more times.
- **-cp**
  This is an optional parameter. The value *entries* specifies entries to be added to CLASSPATH when the generated classes are compiled. Multiple entries are separated the same as they would be in the CLASSPATH environment variable, with a semicolon on Windows platforms and a colon for UNIX platforms.
- **-codegen**
  This is an optional parameter. Specifies that deployment code is to be generated, but not compiled. This option implicitly specifies the -keep option.
- **-debug**
  Includes debugging information when compiling, that is, use javac -g to compile.
- **-help**
  Displays a help message and exit.
- **-ignoreerrors**
  Do not stop deployment if validation or compilation errors are encountered.
- **-keep**
  Do not delete working directories containing generated classes. A message is displayed indicating the name of the working directory that is retained.
- **-novalidate**
  Do not validate the Web services deployment descriptors in the input file.
- **-trace**
  Displays processing information, including the names of the generated files.

Example

```
wsdeploy x.ear x_deployed.ear -trace -keep
Processing web service module x_client.jar.
Keeping directory: f:\temp\Base53383.tmp for module: x_client.jar.
Parsing XML file:f:\temp\Base53383.tmp\WarDeploy.wsdl
Generating f:\temp\Base53383.tmp\generatedSource\com\test\WarDeploy.java
Generating f:\temp\Base53383.tmp\generatedSource\com\test\WarDeployLocator.java
Generating f:\temp\Base53383.tmp\generatedSource\com\test\HelloWsBindingStub.java
Compiling f:\temp\Base53383.tmp\generatedSource\com\test\WarDeploy.java.
Compiling f:\temp\Base53383.tmp\generatedSource\com\test\WarDeployLocator.java.
Compiling f:\temp\Base53383.tmp\generatedSource\com\test\HelloWsBindingStub.java.
Done processing module x_client.jar.
```

Note: The wsdeploy always displays messages in English, regardless of the locale setting.

The setupWebServiceClientEnv script: The setupWebServiceClientEnv script sets up a Java environment for Web Services J2SE clients to use and sets the classpath variable for Web Services clients. After you set the classpath variable, you do not need to specify classpath values when you run a web service client application. For information on how to use the script, see “Set up a Web services client development environment” on page 11.

Authority

To run this script, your user profile must have *RX authority.

Syntax

The syntax of the script is:

```
setupWebServiceClientEnv
```

Publish Web Services Description Language files

The Web Services Description Language (WSDL) files for each Web services-enabled module are published to the file system location you specify. You can provide these WSDL files to clients that want to invoke your Web services.

You can publish WSDL files for the deployed EAR file in one of the following ways:

- "Publish Web Services Description Language files with the administrative console"
- "Publish Web Services Description Language files with wsadmin" on page 75
- "Publish Web Services Description Language files through a URL" on page 75

For more information, see "Multipart Web Services Description Language file best practices" on page 76.

Publish Web Services Description Language files with the administrative console: When you use the administrative console to publish Web Services Description Language (WSDL) files, you can specify default or custom HTTP URL prefixes.

To publish a WSDL file with the administrative console, follow these steps:

1. Start the administrative console.
2. In the topology tree, expand **Applications** and click **Enterprise Applications**.
3. Click the name of the application that contains the Web service for which you want to publish a WSDL file.
4. Under **Additional Properties**, click **Publish WSDL**.
5. On the **Publish WSDL files for Web Services** page, specify the default URL prefixes for the Web service.
   a. Select **HTTP URL prefix**.
   b. Select an entry from the drop down list. If you have multiple application modules, select the application module's checkbox on the module table.
   c. Click **Apply**. The URL prefix is copied to the selected module HTTP URL prefix field.
   d. Click **OK**.
   e. Click the exported *WSDL_zip_file* listed on the **Export WSDL Zip file** page.
   f. Follow your browser's instructions to download the zip file.
6. Specify custom URL prefixes for the Web service.
   a. Select **Custom HTTP URL prefix**.
   b. Type the name of the URL prefix in the **Custom HTTP URL prefix** field. The entry must be of the form http|https://*host_name*:*port_number*. For example, http://myHost:999.
   c. If you have multiple application modules, select the application module's checkbox on the module table.
   d. Click **Apply**. The URL prefix is copied to the selected module HTTP URL prefix field.
   e. Click **OK**.
   f. Click the exported *WSDL_zip_file* listed on the **Export WSDL Zip file** page.
   g. Follow your browser's instructions to download the zip file.

Publish Web Services Description Language files with wsadmin: The Web Services Description Language (WSDL) files in each Web services-enabled module are published to the file system location you specify. You can provide these WSDL files to the clients that want to invoke your Web services.

The wsadmin tool can publish the WSDL files in either local or remote mode. If you publish the WSDL file in local mode, the target application must be located at the same node where the wsadmin command is invoked.

To publish a WSDL file with wsadmin, follow these steps:

1. Start wsadmin.
2. At the wsadmin command prompt, run the $AdminApp publishWSDL command.
   - If you want to update the WSDL Simple Object Access Protocol (SOAP) address prefixes with the default values, run this command:
     ```
     $AdminApp publishWSDL app_Name path_Name
     ```
     where *app_Name* is the application name and *path_Name* is the fully-qualified absolute path to the zip file in which the command publishes the WSDL files.
   - If you do not want to update the WSDL Simple Object Access Protocol (SOAP) address prefixes with the default values, or if you want to customize the WSDL SOAP address for each module, run this command:
     ```
     $AdminApp publishWSDL app_Name path_Name {{module {{binding url-prefix}}}}
     ```
     where *app_Name* is the application name, *path_Name* is the fully-qualified absolute path to the zip file in which the command publishes the WSDL files, *module* is the name of a module for which you want to specify a WSDL SOAP address, *binding* is either `http` or `jms`, and *url-prefix* is the partial SOAP address for the associated SOAP binding.

     You can specify a different address prefix for each SOAP binding.

   Notes:
   - The zip file is saved in the application server machine. The directory structure in the zip file is *appName*/*moduleName*/WEB-INF/wsdl/*fileName*.wsdl, where *appName* is the name of the application EAR file, *moduleName* is the name of the module WAR file, and *fileName* is the name of the WSDL file.
   - For an HTTP binding the form is http://*host_name*:*port*/ or https://*host_name*:*port*, where *host_name* is the name of the machine that hosts the application and *port* is the port number used to access the application.

Publish Web Services Description Language files through a URL: The files referenced by the <wsdl-file> element in the webservices.xml file can or cannot import other Web Services Description Language (WSDL) or XSD files. Typically, all WSDL or XSD files are originally placed into the WEB-INF/wsdl directory when using Java beans. If your WSDL or XSD files are not placed in this directory, the file referenced by the <wsdl-file> and its imported files are located at the same directory and copied to the wsdl/ directory for publishing purposes.

To publish a WSDL file through a URL, follow these steps:

1. Retrieve the outermost WSDL file.
   The outermost WSDL file is the WSDL file defined by the <wsdl-file> element in the webservices.xml file.

   Each Web service has an endpoint address, like http://example.com/services/stockquote. You can retrieve the outermost WSDL file (defined by the <wsdl-file> element within the webservices.xml file) by appending the string "/wsdl" or "/wsdl/" to the endpoint address, for example,http://example.com/services/stockquote/wsdl.
2. Retrieve the imported WSDL files.
   When the outermost WSDL file imports other WSDL or XSD files, these imported files can be retrieved by appending the relative path to the URL, which is used to retrieve the outermost WSDL

file. This is also true for WSDL files that import other files. This process is similar to typical HTTP protocol. If an HTML document contains a hyperlink to other documents, the relative path is appended to create the URL to access the hyperlinked documents.

Example

Suppose you have an application with the following directory structure:

```
module-root/
  WEB-INF/
    webservices.xml
    web.xml
    ibm-webservices-bnd.xml
    jaxrpc-mapping-file
  wsdl/
    myServiceImpl.wsdl
    myService.wsdl
    myServiceTypes.xsd
```

- The webservices.xml file defines the myService service, and the <wsdl-file> element points to /wsdl/myServiceImpl.wsdl.
- The myServiceImpl.wsdl file imports myService.wsdl, which is an interface. This file is the outermost WSDL file.
- The myService.wsdl file imports the type definition for the interface.

If the SOAP address for the myService service is http://examples.com:9080/services/myService, you can retrieve the outermost WSDL with the this URL:

```
http://examples.com:9090/services/myService/wsdl
```

The URL is redirected to http://examples.com:9090/services/myService/wsdl/myServiceImpl.wsdl.

In this example, the myServiceImpl.wsdl file includes this <import> element:

```
<import namespace="http://examples.com/myService" location="a/b/myService.wsdl>
```

To obtain the myService.wsdl file, use this URL:

```
http://examples.com:9090/services/myService/wsdl/a/b/myService.wsdl
```

Multipart Web Services Description Language file best practices: WebSphere Application Server - Express supports deployment of Web services using a multipart Web Services Description Language (WSDL) file. That is, WSDL files import other WSDL files when the WSDL file listed in the <wsdl-file> element of the webservices.xml deployment descriptor contains all <wsdl:service> and <wsdl:port> elements. The WSDL file is divided into an implementation WSDL and an interface WSDL.

The <wsdl:import> element indicates a reference to another WSDL file. If the <wsdl:import> element location attribute does not contain a URL, that is, it contains only a file name, and does not begin with http://, https:// or file://, the imported file must be located in the same directory and must not contain a relative path component. For example, if WEB-INF/A_Impl.wsdl is in your module and contains the import statement <wsdl:import="A.wsdl" namespace="..."/>, the file, A.wsdl must also be located in the module WEB-INF directory.

It is recommended that all WSDL files be placed in the WEB-INF/wsdl directory if you are using Java beans, even if there are relative imports within the WSDL files. Otherwise, there are implications when the WSDL publication is involved with <location="../interfaces/A_Interface.wsdl" namespace="..."/>. Using a path like this fails due to the presence of the relative path, regardless of whether the file is located at that path or not. If the location is a URL, it must be readable at both deployment and server startup.

WSDL publication

The files located in the WEB-INF/wsdl directory can be published through either a URL or file, including WSDL or XSD files. For example, if the file referenced in the <wsdl:file> element of the webservices.xml deployment descriptor is located in the WEB-INF/wsdl directory, it is publishable. If the files imported by the <wsdl:file> are located in the wsd/ directory or its subdirectory, they are publishable.

If the WSDL file referenced by the <wsdl:file> element is located in a directory other than wsdl/, or its subdirectories, the file and its imported files, either WSDL or XSD files, which are in the same directory, are copied to the wsdl/ directory without modification when the application is installed. These types of files can also be published.

If the <wsdl:file> imports a file located in a different directory, the file is not copied to the wsdl/ directory and not available for publishing.

Configure Web services security (Version 5.0.2 or later)

Web services security for WebSphere Application Server - Express Version 5.0.2 and above is based on standards included in the Web services security specification

(http://www.ibm.com/developerworks/library/ws-secure/). Web services security is a message-level standard, based on securing Simple Object Access Protocol (SOAP) messages through XML digital signature, confidentiality through XML encryption and credential propagation through security tokens.

The specification proposes a standard set of Simple Object Access Protocol (SOAP) extensions that you can use to build secure Web services. These standards confirm integrity and confidentiality, which are generally provided with digital signature and encryption technologies. In addition, Web services security provides a general purpose mechanism for associating security tokens with messages. A typical example of the security token is a user name and password token, in which a user name and password are included as text. Web services security defines how to encode binary security tokens such as X.509 certificates and Kerberos tickets.

Note: If you are using Apache SOAP 2.3 (deprecated), see “Secure SOAP services” on page 43 for information about configuring Web services security.

For an explanation of Web services security and for instructions on how to configure WebSphere Application Server - Express, see the following topics:

“Overview of Web services security” on page 78
See this topic for information about how WebSphere Application Server - Express implements Web services security, including the architecture, scenarios, and sample configurations.

“Configure Web services authentication” on page 98
See this topic for instructions for configuring authentcation for Web services.

“Configure Web services for digital signing” on page 139
You can configure your Web services to digitally sign portions of a SOAP message. See this topic for more information.

“Configure Web services encryption and decryption” on page 159
WebSphere Application Server - Express supports the encryption and description of SOAP messages. See this topic for more information.

“Configure client-side SSL for Web services” on page 169
This topic describes how to configure SSL for Web services clients.

Overview of Web services security

See the following topics for information about Web services security:

"Web services security and WebSphere Application Server - Express"
This topic describes the specifications and Web services security elements that are supported by WebSphere Application Server - Express Version 5.0.2 and later.

"Web services security architecture" on page 80
See this topic for information about the Web services security model, including message interpretation, security programming interfaces (SPIs), and default runtime configuration.

"Web services security and J2EE role-based security" on page 85
See this topic for information about how WebSphere Application Server - Express supports the JSR 101 and JSR 109 specifications for J2EE Web services.

"Token type overview" on page 87
Web services security in WebSphere Application Server - Express uses various security tokens for authentication. See this topic for more information.

"Sample Web services security configurations" on page 90
See this topic for information about sample and default Web services security configurations that are provided with WebSphere Application Server - Express.

"Default bindings for Web services" on page 96
See this topic for information about WebSphere Application Server - Express default bindings, such as trust stores, key stores, and authentication method.

Web services security and WebSphere Application Server - Express: WebSphere Application Server - Express Version 5 and Version 5.0.1 support digital signature for Apache SOAP Version 2.3. However, the strategic direction for IBM is based on the Web services security specification, *Web Services Security* (WS-Security), proposed by IBM, Microsoft, and Verisign in April 2002. Starting with Version 5.0.2, WebSphere Application Server - Express supports Web services security. The implementation is based on the IBM Web services engine.

Web services security is a SOAP message-level security specification that is used to support security token propagation, message integrity, and message confidentiality. One intent of the specification is to address interoperability between different implementations of Web services security.

To realize the benefits of Web services security, it is recommended that an implementation of the specification is integrated with underlying security mechanisms. This implementation is fully integrated with the WebSphere Application Server - Express security infrastructure. Authorization, for example, is based on the J2EE security model. When a user ID and password are embedded in a request message, authentication is performed with the user ID and password. If successful, a user identity is established in the context and further resource access is authorized on that identity. After the user ID and password are authenticated by the Web services security run time, a J2EE container performs authorization.

WebSphere Application Server - Express provides an implementation of the key features of Web services security based on the following specifications:

- Specification: Web Services Security (WS-Security) Version 1.0 05 April 2002

  (http://www-106.ibm.com/developerworks/webservices/library/ws-secure/)
- Web Services Security Addendum 18 August 2002

(http://www-106.ibm.com/developerworks/webservices/library/ws-secureadd.html)

- Web Services Security: SOAP Message Security Working13 May 2003

  (http://www.oasis-open.org/committees/download.php/2314/WSS-SOAPMessageSecurity-13-050103-merged.pdf)
- Web Services Security: Username Token Profile Draft 2

  (http://www.oasis-open.org/apps/group_public/download.php/1003/documents/documents/WSS-Username-02-0223-merged.pdf)

The following is a summary of Web services security elements supported by WebSphere Application Server - Express:

Table 1: Web services security elements

Element	Notes
UsernameToken	Both the user name and password for the BasicAuth authentication method and the user name for the identity assertion authentication method are supported. WebSphere Application Server - Express does not support the Password Digest, Nonce, and Created attributes.
BinarySecurityToken	X.509 certificates and LTPA can be imbedded, but there is no implementation to imbed Kerberos tickets. However, the binary token generation and validation are pluggable and are based on the Java Authentication and Authorization Service (JAAS) APIs. You can extend this implementation to generate and validate other types of binary security tokens.
Signature	The X.509 certificate is imbedded as a BinarySecurityToken and can be referenced by the SecurityTokenReference. WebSphere Application Server - Express does not support shared, key-based signature.
Encryption	Both the EncryptedKey and ReferenceList XML tags are supported. KeyIdentifier specifies public keys and KeyName identifies the secret keys. WebSphere Application Server - Express has the capability to map an authenticated identity to a key for encryption or use the signer certificate to encrypt the response message.
Timestamp	WebSphere Application Server - Express supports the Created and Expires attributes. The freshness of the message is checked only if the Expires attribute is present in the message. WebSphere Application Server - Express does not support the Received attribute, which is defined in the addendum. Instead, WebSphere Application Server - Express uses the TimestampTrace Received attribute, which is defined in the OASIS specification.
XML based token	You can insert and validate an arbitrary format of XML tokens into a message. This format mechanism is based on the JAAS APIs.

Signing and encrypting attachments is not supported in WebSphere Application Server - Express. The namespaces used for sending a message were published by OASIS in draft 13. However, the Web services security run time in WebSphere Application Server - Express can accept any of the following namespaces:

- **April 2002 Specification**
  - http://schemas.xmlsoap.org/ws/2002/04/secext
- **August 2002 Addendum**
  - http://schemas.xmlsoap.org/ws/2002/07/secext
  - http://schemas.xmlsoap.org/ws/2002/07/utility
- **OASIS draft (published on 13 May 2003)**
  - http://schemas.xmlsoap.org/ws/2003/06/secext
  - http://schemas.xmlsoap.org/ws/2003/06/utility

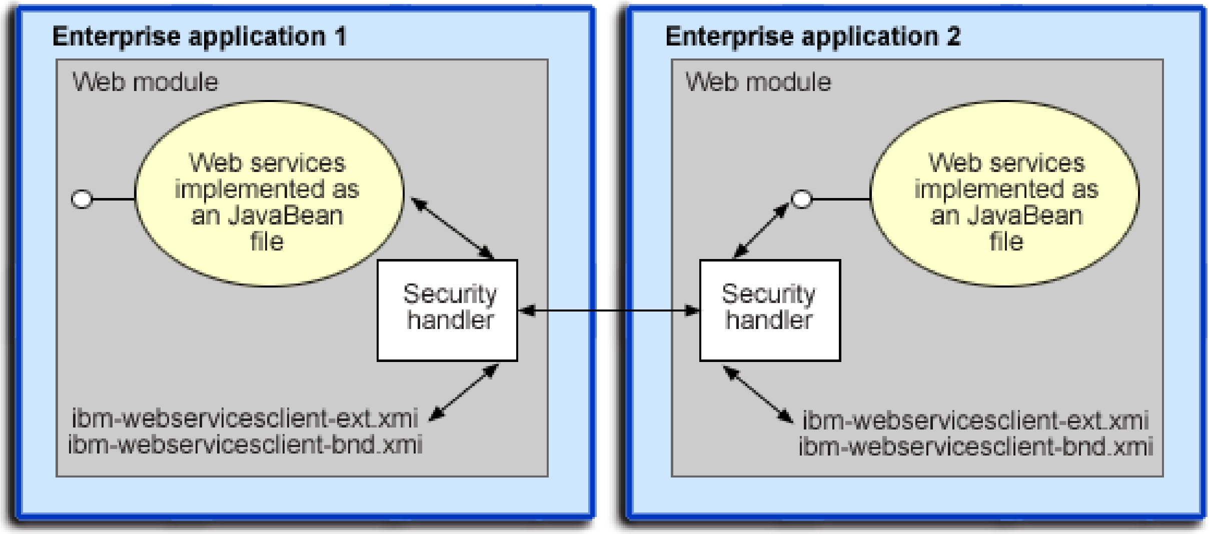
WebSphere Application Server - Express provides the following capability for Web services security:

- Integrity of the message
- Authenticity of the message
- Confidentiality of the message
- Privacy of the message
- Transport level security: provided by Secure Sockets Layer (SSL)
- Security token propagation (pluggable)
- Identity assertion

For a description of capabilities that are not supported, see Table 1: Web services security elements (page 79).

Web services security architecture: The Web services security model employed by WebSphere Application Server - Express is the declarative model. There are no APIs in for programmatically interacting with Web services security, but there are a few Server Provider Interfaces (SPIs) for extending some security-related behaviors.

Figure 1: Web services security model


Enterprise application 1
Web module
Web services implemented as an JavaBean file
Security handler
ibm-webservicesclient-ext.xmi
ibm-webservicesclient-bnd.xmi
Enterprise application 2
Web module
Web services implemented as an JavaBean file
Security handler
ibm-webservicesclient-ext.xmi
ibm-webservicesclient-bnd.xmi

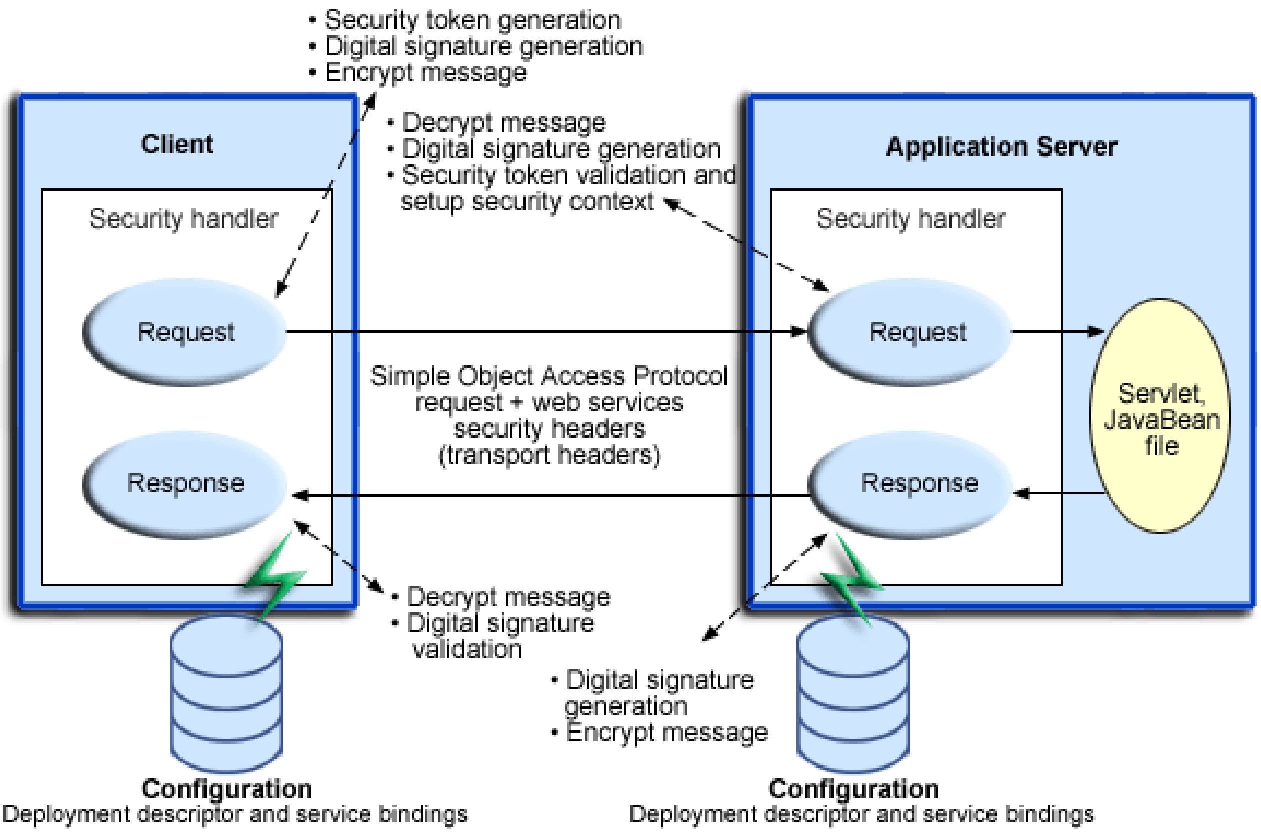
The security constraints for Web services security are specified in the IBM deployment descriptor extension for Web services. The Web services security run time acts on the constraints to enforce Web services security for the SOAP message. The scope of the IBM deployment descriptor extension is at the Web module level. Bindings are also associated with each of the following IBM deployment descriptor extensions:

- **Client** (A Web services client be either a stand-alone client or a Web service that acts as a client to another Web service.)
  - ibm-webservicesclient-ext.xmi
  - ibm-webservicesclient-bnd.xmi
- **Server**
  - ibm-webservices-ext.xmi
  - ibm-webservices-bnd.xmi

It is recommended that you use the tools provided by IBM (such as WebSphere Development Client for iSeries) to create the IBM deployment descriptor extension and bindings. After the bindings are created, you can use the tools or the WebSphere administrative console to specify the bindings.

Note: The binding information is collected after the application has been deployed, not during deployment itself. The alternative is to specify the required binding information before deploying your application.

Figure 2: Web services security message interpretation


• Security token generation
• Digital signature generation
• Encrypt message
Client
Security handler
Request
Response
• Decrypt message
• Digital signature generation
• Security token validation and setup security context
Application Server
Security handler
Request
Response
Simple Object Access Protocol request + web services security headers (transport headers)
Servlet, JavaBean file
• Decrypt message
• Digital signature validation
• Digital signature generation
• Encrypt message
Configuration
Deployment descriptor and service bindings
Configuration
Deployment descriptor and service bindings

The Web services security run time enforces or applies Web services security based on the defined security constraints in the deployment descriptor and binding files. In Figure 2, Web services security has the following points where it intercepts the message and acts on the security constraints that are defined:

- **Request sender**
  - Is defined in the ibm-webservicesclient-ext.xmi and ibm-webservicesclient-bnd.xmi files.
  - Applies the appropriate security constraints to the SOAP message (such as signing or encryption) before the message is sent across the wire, generating the time stamp or the required security token.
- **Request receiver**
  - Is defined in the ibm-webservices-ext.xmi and ibm-webservices-bnd.xmi files.
  - Verifies that the Web services security constraints are met.
  - Verifies the freshness of the message based on the time stamp.
  - Verifies the required signature.
  - Verifies that the message is encrypted and decrypts the message if encrypted.
  - Validates the security tokens and sets up the security context for the down-stream call.
- **Response sender**
  - Is defined in the ibm-webservices-ext.xmi and ibm-webservices-bnd.xmi files.
  - Applies the appropriate security constraints to the SOAP message response, like signing the message, encrypting the message, or generating the time stamp.
- **Response receiver**
  - Is defined in the ibm-webservicesclient-ext.xmi or ibm-webservicesclient-bnd.xmi file.
  - Verifies that the Web services security constraints are met.
  - Verifies the freshness of the message based on the time stamp.
  - Verifies the required signature.

  - Verifies that the message is encrypted and decrypts the message, if encrypted.

Web services security programming interfaces

SPIs are provided to extend the capability of the Web services security run time. The following SPIs and application programming interfaces (APIs) are available:

- **com.ibm.wsspi.wssecurity.config.KeyLocator**
  This SPI is an abstract class for obtaining the keys for digital signature and encryption. The following implementations are the defaults:
  - **com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator**
    Implements the Java key store.
  - **com.ibm.wsspi.wssecurity.config.WSIdKeyStoreMapKeyLocator**
    Provides a mapping of authenticated identity to a key for encryption, or uses the default key that is specified. This is typically used in the response sender configuration.
  - **com.ibm.wsspi.wssecurity.config.CertInRequestKeyLocator**
    Provides the capability of using the signer key for encryption in the response message. This is typically used in the response sender configuration.
- **com.ibm.wsspi.wssecurity.id.TrustedIDEvaluator**
  An interface that used to evaluate the trust for identity assertion. The following implementation is the default:
  - **com.ibm.wsspi.wssecurity.id.TrustedIDEvaluatorImpl**
    Enables you to define a list of trusted identities.
- **JAAS CallbackHandler APIs**
  Used for token generation by the request sender. These APIs can be extended to generate a custom token that is inserted in the Web services security header. The following implementations are the defaults that are provided by WebSphere Application Server - Express:
  - **com.ibm.wsspi.wssecurity.auth.callback.GUIPromptCallbackHandler**
    Presents a login prompt to gather the basic authentication data. Use this implementation in the client environment only.
  - **com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler**
    Collects the basic authentication data with Standard in (stdin). Use this implementation in the client environment only.
  - **com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler**
    Reads the basic authentication data from the application binding file. This may be used on the server side to generate a user name token.
  - **com.ibm.wsspi.wssecurity.auth.callback.LTPATokenCallbackHandler**
    Generates an LTPA token in the Web services security header as binary security token. If there is basic authentication data that is defined in the application binding file, this implementation is used to perform a login, extract the LTPA token from the WebSphere credentials, and insert the token in the Web services security header. Otherwise, it extracts the LTPA security token from the invocation credentials (RunAs identity) and inserts the token in the Web services security header.
- **JAAS LoginModule API**
  Used for token validation of the request receiver side of the message. You can implement a custom LoginModule to perform validation of the custom token on the request receiver of the message. After the token is verified and validated, the token is set as the caller (the RunAs identity in the WebSphere run time) and the identity is used for authorization checks by the containers before a J2EE resource is invoked.

  The following configurations are the default AuthMethod configurations that are provided by WebSphere Application Server - Express:
  - **BasicAuth**
    Validates a user name token.

- **Signature**
  Maps a distinguished name (DN) of a verified certificate to a JAAS subject.
- **IDAssertion**
  Maps a trusted identity to a JAAS subject.
- **LTPA**
  Validates an LTPA token received in the message and creates a JAAS subject.

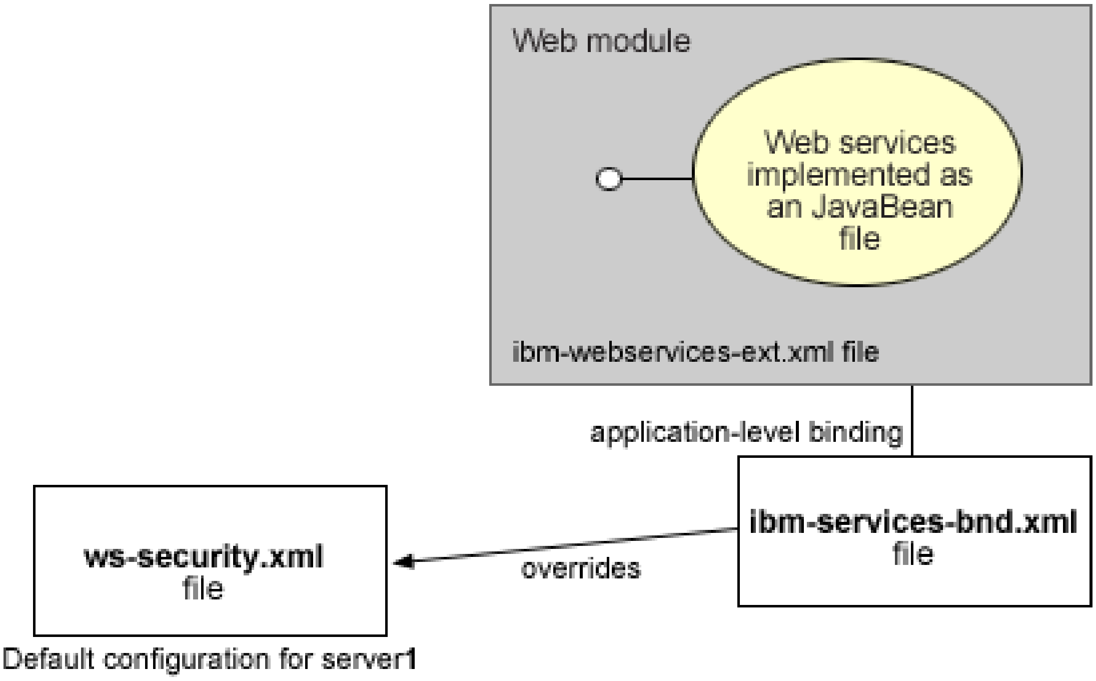
Default configuration (ws-security.xml) for WebSphere Application Server - Express

In WebSphere Application Server - Express, each application server has a copy of ws-security.xml, the file that defines the default binding information for Web services security. The following is a list of defaults defined in the ws-security.xml file:

- **Trust Anchors**
  Identifies the trusted root certificates for signature verification.
- **Certificate Stores**
  Contains certificate revocation lists (CRLs) and non-trusted certificates for verification.
- **KeyLocators**
  Locates the keys for digital signature and encryption.
- **TrustedIDEvaluators**
  Evaluates the trust of the received identity before identity assertion.
- **LoginMappings**
  Contains the JAAS configurations for AuthMethod token validation.

If the Web services security constraints that are specified in the deployment descriptors and the required bindings are not defined in the bindings file, the default constraints in the ws-security.xml file are used.

Figure 3: Runtime configuration

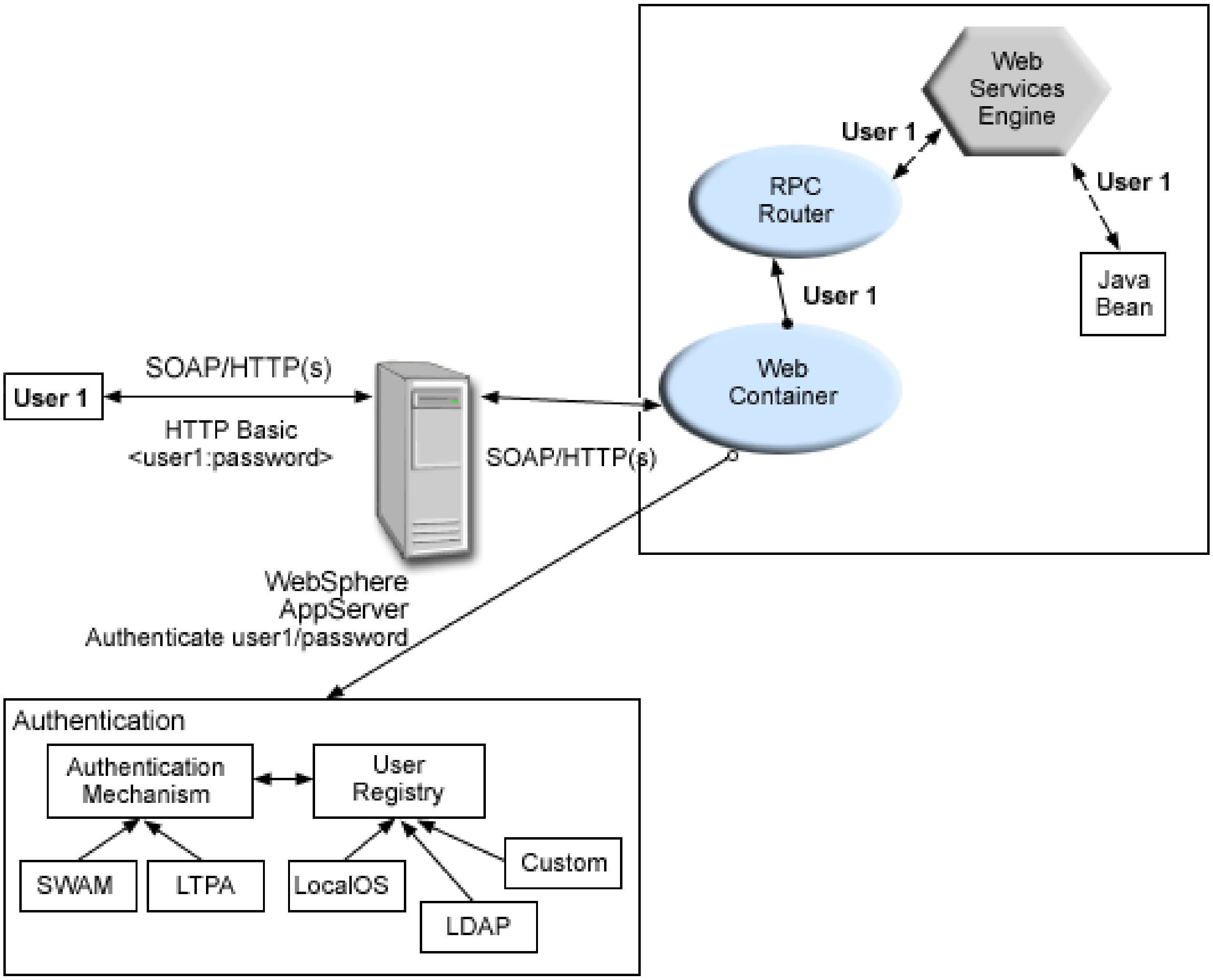

Web module
Web services implemented as an JavaBean file
ibm-webservices-ext.xml file
application-level binding
ibm-services-bnd.xml file
overrides
ws-security.xml file
Default configuration for server1

Web services security and J2EE role-based security: WebSphere Application Server - Express supports JSR 101 and JSR 109. These JSRs define Web services for Java 2 Platform Enterprise Edition (J2EE) architecture so you can develop and run Web services on the J2EE component architecture. This architecture allows Web services to leverage the infrastructure of the J2EE platform like the J2EE deployment model, scalability, security, transaction, and other quality of services provided by the J2EE platform. This document describes the relationship between Web services security (message level security) and J2EE platform security.

WebSphere Application Server - Express security (J2EE role-based security)

You can secure Web services using the existing security infrastructure of WebSphere Application Server - Express, J2EE role-based security, and SSL transport level security.

Figure 1: Simple object access protocol message flow through the existing security infrastructure of WebSphere Application Server - Express



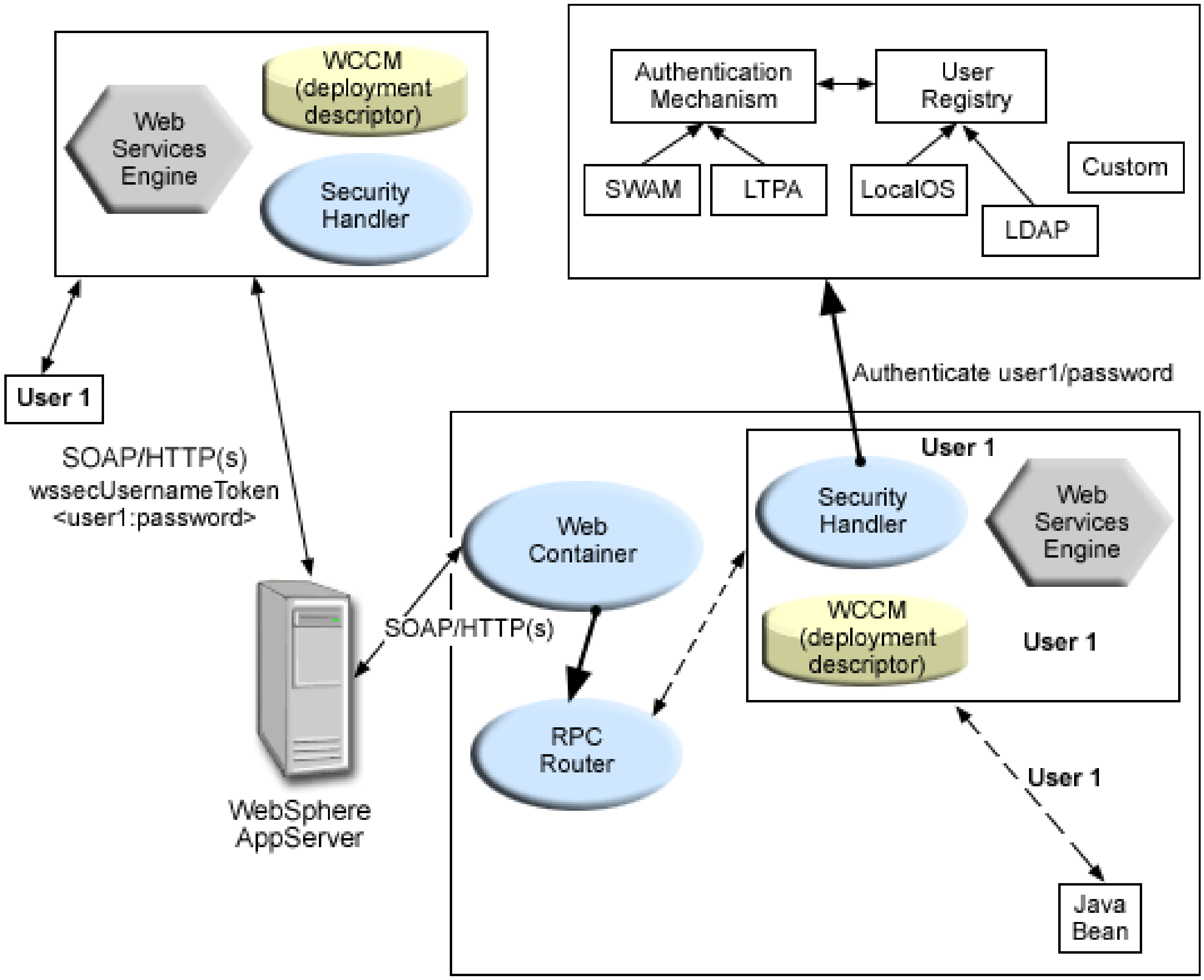
The Web services endpoint can be secured using J2EE role-based security. The Web services sender sends the basic authentication data using the HTTP header. SSL (HTTPS) can be used to secure the transport. When the WebSphere Application Server - Express receives the SOAP message, the Web container authenticates the user (in this example, user1) and sets the security context for the call. After this is complete, the SOAP router servlet sends the request to the implementation of the Web services (the implementation can be a JavaBean).

The Web services endpoint also can be secured using the J2EE role. Then, authorization is performed by the Web container before the SOAP request is dispatched to the Web services implementation.

Web services security

You can also secure Web services using Web services security at the message level. In this case, you can digitally sign or encrypt a certain part of the message. Web services security also supports security token propagation within the SOAP message. The following scenario assumes that the Web services endpoint is not secured with J2EE role-based security and the enterprise bean is secured with J2EE role-based security.

Figure 2: SOAP message flow and Web services security



In this case the Web services endpoint is not secured with J2EE role-based security. The Web services engine processes the SOAP message before the client sends the message to the Web services endpoint.

The Web services security run time acts on the security constraints, such as digitally signing, encrypting, or generating (and inserting) a security token in the SOAP header. In this case <wsse:UsernameToken> is generated with the user ID (user1) and password.

On the server-side (receiving), the Web service processes the incoming message and Web services security enforces security constraints. This includes ensuring messages are properly signed, properly encrypted, and decrypted, authenticating the security token, and setting up the security context with the authenticated identity. (In this case, user1 is the authenticated identity.) Finally, the SOAP message is dispatched to the Web services implementation. SSL also may be used in this scenario.

Web services security and J2EE role-based security

The previous scenario shows that Web services security can compliment J2EE role-based security. For example, SSL can be enabled at the transport level to provide a secure channel. The Web services security run time uses the security infrastructure in order to set the authenticated identity in the security context. The authenticated identity can be used in the downstream call to J2EE resources (or other resource types).

There are subtle consequences of combining the two scenarios. For example, the HTTP transport is sending basic authentication data with the user ID (user1) and password in the HTTP header, but <wsse:UsernameToken> with a different user ID (user99) and password is also inserted into the SOAP header. In the previous scenarios, there are two authentications performed. One is performed by Web container for authenticating user1, and the other is performed by Web services security for authenticating user99. Web services security run time runs after Web container authentication runs, so user99 is the authenticated identity that is set in the security context.

Token type overview: A security token represents a set of claims made by a client that may include a name, password, identity, key, certificate, group, or privilege. Web services security provides a general-purpose mechanism to associate security tokens with messages for single-message authentication. A specific type of security token is not required by Web services security. Web services security is designed to be extensible and support multiple security token formats to accommodate a variety of authentication mechanisms. For example, a client may provide proof of identity and proof that they have a particular business certification.

A security token is embedded in the Simple Object Access Protocol (SOAP) message within the SOAP header. The security token within in the SOAP header is propagated from the message sender to the intended message receiver. On the receiving side, the WebSphere Application Server - Express security handler authenticates the security token and sets up the caller identity on the thread.

The proposed Web services security draft defines these types of security tokens:

- "User name tokens" on page 88
  A username token consists of a user name and, optionally, password information. You can include a username token directly in the <Security> header within the message.
- "Binary security tokens" on page 89
  Binary tokens require a special encoding for inclusion. The Web services security specification describes how to encode binary security tokens such as X.509 certificates and Kerberos tickets; and how to include opaque encrypted keys. The specification also includes extensibility mechanisms that you can use to further describe the characteristics of the credentials that are included with a message. For more information, see Web Services Security (WS-Security)

  (http://schemas.xmlsoap.org/specs/ws-security/ws-security.htm).

WebSphere Application Server - Express Version 5.0.2 supports user name tokens. Basic authentication and identity assertion authentication both require user name tokens. The binary security token implementation supports both X.509 certificates and LTPA binary security. You can extended the implementation to generate other type of tokens. However, Kerberos tickets are not supported in WebSphere Application Server - Express.

Each type of token is processed by a corresponding token-generation and validation module. The binary token generation and validation modules are pluggable and are based on the Java Authentication and Authorization Service (JAAS) framework. For more information, see “Pluggable token support” on page 129. For example, arbitrary XML-based token format is supported using the JAAS pluggable framework. WebSphere Application Server - Express does not support an XML-based token that is used in SecurityTokenReference. For more information, see “XML tokens” on page 89.

You can define the types of tokens that the message can accept in the deployment descriptor extension file, ibm.webservices-ext.xmi. A message receiver may support one or more types of security tokens.

The following example shows that the receiver supports four types of security tokens:

```
<?xml version="1.0" encoding="UTF-8"?>
<com.ibm.etools.webservice.wsext:WsExtension xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
 xmlns:com.ibm.etools.webservice.wsext="http://www.ibm.com/websphere/appserver/schemas/5.0.2/wsext.xmi"
 xmi:id="WsExtension_1052760331306" routerModuleName="StockQuote.war">
  <wsDescExt xmi:id="WsDescExt_1052760331306" wsDescNameLink="StockQuoteFetcher">
    <pcBinding xmi:id="PcBinding_1052760331326" pcNameLink="urn:xmltoday-delayed-quotes" scope="Session">
      <serverServiceConfig xmi:id="ServerServiceConfig_1052760331326"actorURI= "myActorURI">
        <securityRequestReceiverServiceConfig xmi:id="SecurityRequestReceiverServiceConfig_1052760331326">
          <loginConfig xmi:id="LoginConfig_1052760331326">
            <authMethods xmi:id="AuthMethod_1052760331326" text="BasicAuth"/>
            <authMethods xmi:id="AuthMethod_1052760331327" text="IDAssertion"/>
            <authMethods xmi:id="AuthMethod_1052760331336" text="Signature"/>
            <authMethods xmi:id="AuthMethod_1052760331337" text="LTPA"/>
          </loginConfig>
        <idAssertion xmi:id="IDAssertion_1052760331336" idType="Username" trustMode="Signature"/>
        ...
```

The message sender may choose one of the token types that are supported by the receiver when sending a message. You can define the type of token to be used by the sending side in the client descriptor extension file, ibm-webservicesclient-ext.xmi.

The following example shows that the sender chooses to send a UsernameToken to the receiver:

```
<?xml version="1.0" encoding="UTF-8"?>
<com.ibm.etools.webservice.wscext:WsClientExtension xmi:version="2.0"
 xmlns:xmi="http://www.omg.org/XMI" xmlns:com.ibm.etools.webservice.wscext=
 "http://www.ibm.com/websphere/appserver/schemas/5.0.2/wscext.xmi"
 xmi:id="WsClientExtension_1052760331496">
  <serviceRefs xmi:id="ServiceRef_1052760331506" serviceRefLink="service/StockQuoteService">
    <portQnameBindings xmi:id="PortQnameBinding_1052760331506" portQnameLocalNameLink="StockQuote">
      <clientServiceConfig xmi:id="ClientServiceConfig_1052760331506" actorURI="myActorURI">
        <securityRequestSenderServiceConfig xmi:id="SecurityRequestSenderServiceConfig_1052760331506"
         actor="myActorURI">
          <loginConfig xmi:id="LoginConfig_1052760331506" authMethod="BasicAuth"/>
          ...
```

User name tokens: You can use the UsernameToken to propagate a user name and, optionally, password information. Also, you can use this token type to carry basic authentication information. Both a user name and password are used to authenticate the message. A UsernameToken containing the user name is used in identity assertion, which establishes the identity of the user based on the trust relationship.

The following example shows the the syntax of the UsernameToken element:

```
<UsernameToken Id="...">
 <Username>...</Username>
 <Password Type="...">...</Password>
</UsernameToken>
```

The Web services security specification defines the following password types:

- **wsse:PasswordText**
  (Default) This type is the actual password for the user name. WebSphere Application Server - Express supports this type.
- **wsse:PasswordDigest**
  This type is the digest of the password for the user name. The value is a base64-encoded SHA1 hash value of the UTF8-encoded password. WebSphere Application Server - Express does not support password digest because most user registry security policies do not expose the password to the application software.

The following example illustrates the use of the <UsernameToken> element:

```
<S:Envelope xmlns:S="http://www.w3.org/2001/12/soap-envelope"
 xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/04/secext">
  <S:Header>
    ...
    <wsse:Security>
      <wsse:UsernameToken>
        <wsse:Username>Joe</wsse:Username>
        <wsse:Password>ILoveJava</wsse:Password>
      </wsse:UsernameToken>
    </wsse:Security>
    ...
  </S:Header>
  ...
</S:Envelope>
```

The password is transmitted in unencrypted text. Therefore, it is recommended that you use a secure transmission channel between the sender and receiver. For example, you might consider using Secure Sockets Layer (SSL).

Binary security tokens: A binary security token has the following attributes that are used to interpret it:

- **ValueType**
  The ValueType attribute identifies the type of the security token, for example, an LTPA token.
- **Encoding Type**
  The EncodingType indicates how the security token is encoded, for example, `Base64Binary`. The BinarySecurityToken element defines a security token that is binary encoded.

The Web services security implementation for WebSphere Application Server - Express Version 5.0.2 and later supports both LTPA and X.509 certificate binary security tokens.

The following example shows an LTPA binary security token in a Web services security message header:

```
<wsse:BinarySecurityToken
 xmlns:ns7902342339871340177=http://www.ibm.com/websphere/appserver/tokentype/5.0.2"
 EncodingType="wsse:Base64Binary" ValueType="ns7902342339871340177:LTPA">
    MIZ6LGPt2CzXBQfio9wZTo1VotWov0NW3Za6lU5K7Li78DSnIK6iHj3hxXgrUn6p4wZI
    8Xg26havepvmSJ8XxiACMihTJuh1t3ufsrjbFQJOqh5VcRvI+AKEaNmnEgEV65jUYAC9
    C/iwBBWk5U/6DIk7LfXcTT0ZPAd+3D3nCS0f+6tnqMou8EG9mtMeTKccz/pJVTZjaRSo
    msu0sewsOKfl/WPsjW0bR/2g3NaVvBy18VlTFBpUbGFVGgzHRjBKAGo+ctkl80nlVLIk
    TUjt/XdYvEpOr6QoddGi4okjDGPyyoDxcvKZnReXww5UsoqlpfXwN4KG9as=
</wsse:BinarySecurityToken></wsse:Security></soapenv:Header>
```

As shown in the example, the token is Base64Binary encoded.

XML tokens: XML-based security tokens are growing in popularity. The following formats are well-known examples:

- Security Assertion Markup Language (SAML)
- Extensible Rights Markup Language (XrML)

The extensibility of the <wsse:Security> header in XML-based security tokens enables you to directly insert these security tokens into the header.

SAML assertions are attached to Web services security messages using Web services security by placing assertion elements inside the <wsse:Security> header. The following example illustrates a Web services security message with a SAML assertion token.

```
<S:Envelope xmlns:S="...">
  <S:Header>
    <wsse:Security xmlns:wsse="...">
      <saml:Assertion MajorVersion="1" MinorVersion="0" AssertionID="SecurityToken-ef375268"
       Issuer="elliotw1" IssueInstant="2002-07-23T11:32:05.6228146-07:00"
       xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion">
        ...
      </saml:Assertion>
        ...
    </wsse:Security>
  </S:Header>
  <S:Body>
    ...
  </S:Body>
</S:Envelope>
```

For more information on SAML and XrML, see WS-Security Profile for XML-based Tokens

(http://www-106.ibm.com/developerworks/library/ws-sectoken.html).

Sample Web services security configurations: WebSphere Application Server - Express provides the following sample key stores for sample configurations.

The following files are the sample key stores, which are located in the etc/ws-security/samples subdirectory of your instance, /QIBM/UserData/WebASE/ASE5/*instance*/etc/ws-security/samples/ (where *instance* is the name of your instance):

- **dsig-sender.ks** (key store password is `client`)
  - Trusted certificate with alias name `soapca`
  - Personal certificate with alias name `soaprequester` and key password `client`, issued by intermediatory Int CA2 (which is in turn issued by `soapca`)
- **dsig-receiver.ks** (key store password is `server`)
  - Trusted certificate with alias name `soapca`
  - Personal certificate with alias name `soapprovider` and key password `server`, issued by intermediatory Int CA2 (which is in turn issued by `soapca`)
- **enc-sender.jceks** (key store password storepass)
  - Secret key CN=Group1, alias name `Group1` and key password `keypass`
  - Public key CN=Bob, O=IBM, C=US, alias name `bob` and key password `keypass`
  - Private key CN=Alice, O=IBM, C=US, alias name `alice` and key password `keypass`
- **enc-receiver.jceks** (key store password is `storepass`)
  - Secret key CN=Group1, alias name `Group1` and key password `keypass`
  - Private key CN=Bob, O=IBM, C=US, alias name `bob` and key password `keypass`
  - Public key CN=Alice, O=IBM, C=US, alias name `alice` and key password `keypass`
- **intca2.cer**, the intermediatory Int CA2.

Note: These sample key stores are for testing and sample purpose only. Do not use them in production environment.

Default binding

WebSphere Application Server - Express provides the following default binding information:

- **Trust Anchors**
  Used to validate the trust of the signer certificate.
  - **SampleClientTrustAnchor**
    Used by response receiver to validate the signer certificate.
  - **SampleServerTrustAnchor**
    Used by request receiver to validate the signers certificate.
- **Collection Certificate Store**
  Used to validate the certificate path.
  - **SampleCollectionCertStore**
    Used by response receiver and request receiver to validate the signers certificate path.
- **Key Locators**
  Used to locating key for signature, encryption and decryption.
  - **SampleClientSignerKey**
    Used by requesting sender to sign the SOAP message. The signing key name is clientsignerkey, which can be referenced in the signing information as the signing key name.
  - **SampleServerSignerKey**
    Used by the responding sender to sign the SOAP message. The signing key name is serversignerkey, which can be referenced in the signing information as the signing key name.
  - **SampleSenderEncryptionKeyLocator**
    Used by the sender to encrypt the SOAP message. It is configured to use the enc-sender.jceks key store and the com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator key store key locator.
  - **SampleReceiverEncryptionKeyLocator**
    Used by the receiver to decrypt the encrypted SOAP message. It is configured to use the enc-receiver.jceks key store and the com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator key store key locator. It is configured for symmetric encryption (DES or TRIPLEDES). However, to use it for asymmetric encryption (RSA), you must add the private key CN=Bob, O=IBM, C=US, alias name `bob`, and key password `keypass`.
  - **SampleResponseSenderEncryptionKeyLocator**
    Used by response sender to encrypt the SOAP response message. It is configured to use the enc-receiver.jceks key store and the com.ibm.wsspi.wssecurity.config.WSIdKeyStoreMapKeyLocator key locator. This key locator maps an authenticated identity (of the current thread of execution) to a public key for encryption. By default was is configured to map to public key alice, and you must change was to the appropriate user. SampleResponseSenderEncryptionKeyLocator also has the capability to set a default key for encryption (by default it is configured to use public key alice as the default).
- **Trusted ID Evaluator**
  Used to establish trust before asserting to the identity in identity assertion.
  - **SampleTrustedIDEvaluator**
    Is configured to use com.ibm.wsspi.wssecurity.id.TrustedIDEvaluatorImpl. The default implementation of com.ibm.wsspi.wssecurity.id.TrustedIDEvaluator contains a list of trusted identities. The list is defined as properties with `trustedId_*` as the key and the value as the trusted identity. This can be defined in the WebSphere administration console in **Servers —> Application Servers —>** *server* **—> Web Services: Default bindings for Web Services Security —> Trusted ID Evaluators —> SampleTrustedIDEvaluator** for the server level (where *server* is the name of your application server) or **Security —> Web Services —> Trusted ID Evaluators —> SampleTrustedIDEvaluator** for the cell-level (Network Deployment only).
- **Login Mapping**
  Used to authenticate incoming security token in the Web services security SOAP header of a SOAP message.

- **BasicAuth authentication method**
  This method is used to authenticate user name security token (username and password).
- **Signature authentication method**
  This method is used to map a distinguished name (DN) into a WebSphere Application Server - Express Java Authentication and Authorization Server (JAAS) Subject.
- **IDAssertion authentication method**
  This method is used to map a trusted identity into a WebSphere ApplicationSserver JAAS Subject for identity assertion.
- **LTPA authentication method**
  This method is used to validate a Lightweight Third-party Authentication (LTPA) security token.

Note: These default bindings for trust anchors, collection certificate stores, and key locators are for testing or sample purpose only. Do not use it for production.

A sample configuration

The following examples demonstrate what IBM deployment descriptor extensions and bindings can do. The unnecessary information has been removed from the examples to improve clarity. Do not copy and paste these examples into your application's deployment descriptors or bindings. These examples serve as reference only and are not representative of the recommended configuration.

It is recommended that you use the following tools to create or edit IBM deployment descriptor extensions and bindings:

- Use WebSphere Development Studio for iSeries to create or edit the IBM deployment descriptor extensions.
- Use WebSphere Development Studio for iSeries or the WebSphere administrative console to create or edit the bindings file.

The following is an example of a scenario that performs the following actions:

- Signs the SOAP body, time stamp, and security token.
- Encrypts the body content and user name token.
- Sends the user name token (basic authentication data).
- Generates the time stamp for the request.

For the response, the SOAP body and time stamp are signed, the body content is encrypted, and the SOAP message freshness is checked using the time stamp.

Note: The request sender and request receiver are a pair. Similarly, the response sender and response receiver is a pair.

Note: It is recommended that you use the WebSphere Application Server - Express variables for specifying the path to key stores. In the WebSphere administrative console, click **Environment —> Manage WebSphere Variables**. This often ameliorates platform differences such as file-system naming conventions. The samples below use the `${USER_INSTALL_ROOT}` variable to replace /QIBM/UserData/WebASE/ASE5/*instance* (where *instance* is the name of your instance). For more information about setting the variables, see Manage substitution variables with the administrative console in the *Administration* topic.

Client-side IBM deployment descriptor extension

The client-side IBM deployment descriptor extension describes the following constraints:

- **Request Sender**
  - Signs the SOAP body, time stamp and security token

  - Encrypts the body content and user name token
  - Sends the basic authentication token (user name and password)
  - Generates the time stamp to be expired in 3 minutes
- **Response Receiver**
  - Verifies that the SOAP body and time stamp are signed
  - Verifies that the SOAP body content is encrypted
  - Verifies that the time stamp is present (also check for message freshness)

Example 1: Sample client IBM deployment descriptor extension.

Note: The xmi:id xmi:id statements have been removed for readability. They must be added in order for this example to work.

```
<?xml version="1.0" encoding="UTF-8"?>
<com.ibm.etools.webservice.wscext:WsClientExtension xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI" xmlns:com.ibm.etools.webservice.wscext=
  "http://www.ibm.com/websphere/appserver/schemas/5.0.2/wscext.xmi">
  <serviceRefs serviceRefLink="service/myServ">
    <portQnameBindings portQnameLocalNameLink="Port1">
      <clientServiceConfig actorURI="myActorURI">
        <securityRequestSenderServiceConfig actor="myActorURI">
          <integrity>
            <references part="body"/>
            <references part="timestamp"/>
            <references part="securitytoken"/>
          </integrity>
          <confidentiality>
            <confidentialParts part="bodycontent"/>
            <confidentialParts part="usernametoken"/>
          </confidentiality>
          <loginConfig authMethod="BasicAuth"/>
          <addCreatedTimeStamp flag="true" expires="PT3M"/>
        </securityRequestSenderServiceConfig>
        <securityResponseReceiverServiceConfig>
          <requiredIntegrity>
            <references part="body"/>
            <references part="timestamp"/>
          </requiredIntegrity>
          <requiredConfidentiality>
            <confidentialParts part="bodycontent"/>
          </requiredConfidentiality>
          <addReceivedTimeStamp flag="true"/>
        </securityResponseReceiverServiceConfig>
      </clientServiceConfig>
    </portQnameBindings>
  </serviceRefs>
</com.ibm.etools.webservice.wscext:WsClientExtension>
```

Client-side IBM extension bindings

The following is the client-side IBM extension bindings for the security constraints described previously in the discussion on client-side IBM deployment descriptor extensions.

The signer key and encryption (decryption) key for the message can be obtained from the key store key locator implementation (com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator). The signer key is used for encrypting the response. The sample is configured to use Java Certification Path API to validate the certificate path of the signer of the digital signature. The user name token (basic authentication) data is collected from the stdin using one of the default JAAS implementations:javax.security.auth.callback.CallbackHandler implementation (com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler).

Example 2: Sample client IBM extension binding

```
<?xml version="1.0" encoding="UTF-8"?>
<com.ibm.etools.webservice.wscbnd:ClientBinding xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI"
  xmlns:com.ibm.etools.webservice.wscbnd=
  "http://www.ibm.com/websphere/appserver/schemas/5.0.2/wscbnd.xmi">
  <serviceRefs serviceRefLink="service/MyServ">
    <portQnameBindings portQnameLocalNameLink="Port1">
      <securityRequestSenderBindingConfig>
        <signingInfo>
          <signatureMethod algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
          <signingKey name="clientsignerkey" locatorRef="SampleClientSignerKey"/>
          <canonicalizationMethod algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
          <digestMethod algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
        </signingInfo>
        <keyLocators name="SampleClientSignerKey"
         classname="com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator">
          <keyStore storepass="{xor}PDM2OjEr"
           path="$/{USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-sender.ks"
           type="JKS"/>
          <keys alias="soaprequester" keypass="{xor}PDM2OjEr" name="clientsignerkey"/>
        </keyLocators>
        <encryptionInfo name="EncInfo1">
          <encryptionKey name="CN=Bob, O=IBM, C=US"
           locatorRef="SampleSenderEncryptionKeyLocator"/>
          <encryptionMethod algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
          <keyEncryptionMethod algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
        </encryptionInfo>
        <keyLocators name="SampleSenderEncryptionKeyLocator"
         classname="com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator">
          <keyStore storepass="{xor}LCswLTovPiws"
           path="${USER_INSTALL_ROOT}/etc/ws-security/samples/enc-sender.jceks"
           type="JCEKS"/>
          <keys alias="Group1" keypass="{xor}NDomLz4sLA==" name="CN=Group1"/>
        </keyLocators>
        <loginBinding authMethod="BasicAuth" callbackHandler=
         "com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler"/>
      </securityRequestSenderBindingConfig>
      <securityResponseReceiverBindingConfig>
        <signingInfos>
          <signatureMethod algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
          <certPathSettings>
            <trustAnchorRef ref="SampleClientTrustAnchor"/>
            <certStoreRef ref="SampleCollectionCertStore"/>
          </certPathSettings>
          <canonicalizationMethod algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
          <digestMethod algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
        </signingInfos>
        <trustAnchors name="SampleClientTrustAnchor">
          <keyStore storepass="{xor}PDM2OjEr"
           path="${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-sender.ks"
           type="JKS"/>
        </trustAnchors>
        <certStoreList>
          <collectionCertStores provider="IBMCertPath" name="SampleCollectionCertStore">
            <x509Certificates
             path="${USER_INSTALL_ROOT}/etc/ws-security/samples/intca2.cer"/>
          </collectionCertStores>
        </certStoreList>
        <encryptionInfos name="EncInfo2">
          <encryptionKey locatorRef="SampleReceiverEncryptionKeyLocator"/>
          <encryptionMethod algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
          <keyEncryptionMethod algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
        </encryptionInfos>
        <keyLocators name="SampleReceiverEncryptionKeyLocator"
         classname="com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator">
          <keyStore storepass="{xor}PDM2OjEr"
          path="${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-sender.ks"
          type="JKS"/>
         <keys alias="soaprequester" keypass="{xor}PDM2OjEr" name="clientsignerkey"/>
       </keyLocators>
     </securityResponseReceiverBindingConfig>
   </portQnameBindings>
  </serviceRefs>
</com.ibm.etools.webservice.wscbnd:ClientBinding>
```

Server side IBM deployment descriptor extension

The server-side IBM deployment descriptor extension describes the following constraints:

- **Request Receiver** (ibm-webservices-ext.xmi and ibm-webservices-bnd.xmi)
  - Verifies that the SOAP body, time stamp, and security token are signed
  - Verifies that the SOAP body content and user name token are encrypted
  - Verifies that the basic authentication token (user name and password) is in the Web services security SOAP header
  - Verifies that the time stamp is present (also check for message freshness)
- **Response Sender** (ibm-webservices-ext.xmi and ibm-webservices-bnd.xmi)
  - Signs the SOAP body and time stamp
  - Encrypts the SOAP body content
  - Generates the time stamp to expire in 3 minutes

Example 3: Sample server IBM deployment descriptor extension

```
<?xml version="1.0" encoding="UTF-8"?>
<com.ibm.etools.webservice.wsext:WsExtension xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI" xmlns:com.ibm.etools.webservice.wsext=
  "http://www.ibm.com/websphere/appserver/schemas/5.0.2/wsext.xmi">
  <wsDescExt wsDescNameLink="MyServ">
    <pcBinding pcNameLink="Port1">
      <serverServiceConfig actorURI="myActorURI">
        <securityRequestReceiverServiceConfig>
          <requiredIntegrity>
            <references part="body"/>
            <references part="timestamp"/>
            <references part="securitytoken"/>
          </requiredIntegrity>
          <requiredConfidentiality">
            <confidentialParts part="bodycontent"/>
            <confidentialParts part="usernametoken"/>
          </requiredConfidentiality>
          <loginConfig>
            <authMethods text="BasicAuth"/>
          </loginConfig>
          <addReceivedTimestamp flag="true"/>
        </securityRequestReceiverServiceConfig>
        <securityResponseSenderServiceConfig actor="myActorURI">
          <integrity>
            <references part="body"/>
            <references part="timestamp"/>
          </integrity>
          <confidentiality>
            <confidentialParts part="bodycontent"/>
          </confidentiality>
          <addCreatedTimestamp flag="true" expires="PT3M"/>
        </securityResponseSenderServiceConfig>
      </serverServiceConfig>
    </pcBinding>
  </wsDescExt>
</com.ibm.etools.webservice.wsext:WsExtension>
```

Server-side IBM extension bindings

The following binding information is reusing some of the default binding information defined either at the server level or the cell level, which depends upon the installation. For example, request receiver is referencing the SampleCollectionCertStore certificate store and the SampleServerTrustAnchor trust store is defined in the default binding. However, the encryption information in the request receiver is references a SampleReceiverEncryptionKeyLocator key locator that is defined in the application-level binding (the same ibm-webservices-bnd.xmi file). The response sender is configured to use the signer key of the digital signature of the request to encrypt the response using one of the default key locator (com.ibm.wsspi.wssecurity.config.CertInRequestKeyLocator) implementations.

Example 4: Sample server IBM extension binding

```
<?xml version="1.0" encoding="UTF-8"?>
<com.ibm.etools.webservice.wsbnd:WSBinding xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI" xmlns:com.ibm.etools.webservice.wsbnd=
  "http://www.ibm.com/websphere/appserver/schemas/5.0.2/wsbnd.xmi">
  <wsdescBindings wsDescNameLink="MyServ">
    <pcBindings pcNameLink="Port1" scope="Session">
      <securityRequestReceiverBindingConfig>
        <signingInfos>
          <signatureMethod algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
          <certPathSettings>
            <trustAnchorRef ref="SampleServerTrustAnchor"/>
            <certStoreRef ref="SampleCollectionCertStore"/>
          </certPathSettings>
          <canonicalizationMethod algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
          <digestMethod algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
        </signingInfos>
        <encryptionInfos name="EncInfo1">
          <encryptionKey locatorRef="SampleReceiverEncryptionKeyLocator"/>
          <encryptionMethod algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
          <keyEncryptionMethod algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
        </encryptionInfos>
        <keyLocators name="SampleReceiverEncryptionKeyLocator"
         classname="com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator">
          <keyStore storepass="{xor}LCswLTovPiws"
           path="${USER_INSTALL_ROOT}/etc/ws-security/samples/enc-receiver.jceks"
           type="JCEKS"/>
          <keys alias="Group1" keypass="{xor}NDomLz4sLA==" name="CN=Group1"/>
          <keys alias="bob" keypass="{xor}NDomLz4sLA==" name="CN=Bob, O=IBM, C=US"/>
        </keyLocators>
      </securityRequestReceiverBindingConfig>
      <securityResponseSenderBindingConfig>
        <signingInfo>
          <signatureMethod algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
          <signingKey name="serversignerkey" locatorRef="SampleServerSignerKey"/>
          <canonicalizationMethod algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
          <digestMethod algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
        </signingInfo>
        <encryptionInfo  name="EncInfo2">
          <encryptionKey locatorRef="SignerKeyLocator"/>
          <encryptionMethod algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
          <keyEncryptionMethod algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
        </encryptionInfo>
        <keyLocators name="SignerKeyLocator"
         classname="com.ibm.wsspi.wssecurity.config.CertInRequestKeyLocator"/>
      </securityResponseSenderBindingConfig>
    </pcBindings>
  </wsdescBindings>
  <routerModules transport="http" name="StockQuote.war"/>
</com.ibm.etools.webservice.wsbnd:WSBinding>
```

Default bindings for Web services: Certain applications can share certain binding information. This includes trust stores, key stores, and authentication method (token validation). WebSphere Application

Server - Express provides support for default binding information. This means administrators can define binding information at the server level, and applications can refer to the binding information. The default binding information is defined in ws-security.xml and can be administered by either the administrative console or by scripting.

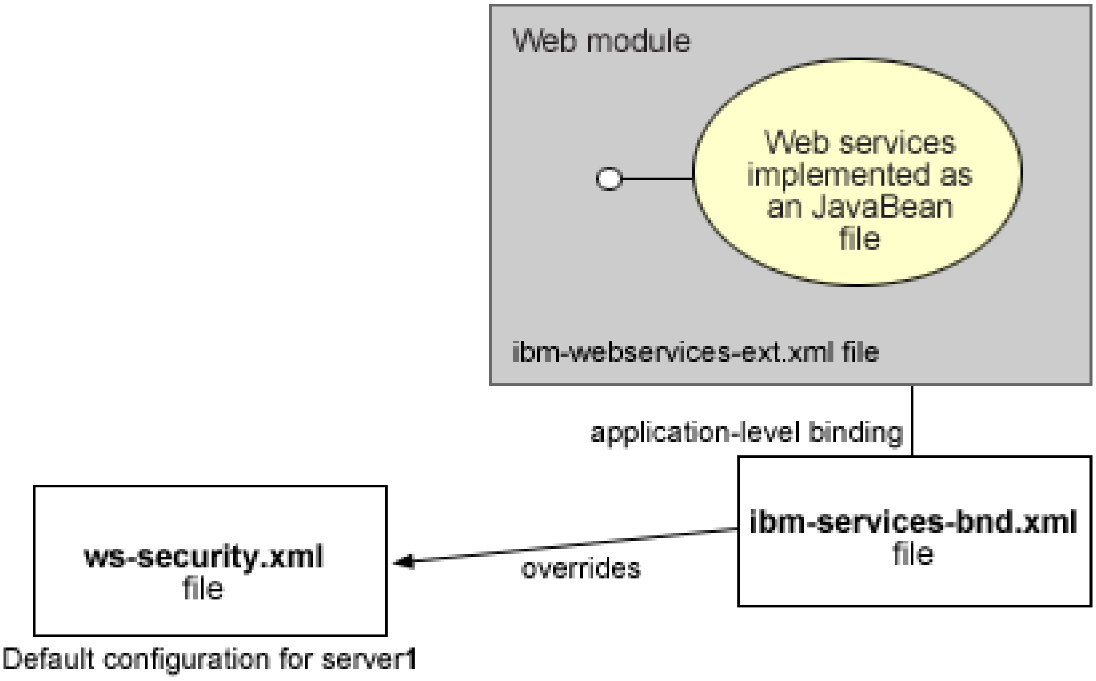
The following binding information can be defined in the ws-security.xml file:

- **Trust anchors (trust store)**
  - Trust anchors contain key store configuration information that has the root-trusted certificates. Trust anchors are used for certificate path validation of the incoming X.509-formatted security tokens.
  - The Trust Anchor Name is used in the binding file (ibm-webservices-bnd.xmi and ibm-webservicesclient-bnd-xmi when Web services is running as client) to refer to the trust anchor defined in the default binding information. The Trust Anchor Name must be unique in the trust anchor collection.
- **Collection certificate store**
  - The collection certificate store specifies a list of untrusted, intermediate certificates and is used for certificate path validation of incoming X.509-formatted security tokens. The default provider is IBMCertPath.
  - The Certificate Store Name is used in the binding file (ibm-webservices-bnd.xmi and ibm-webservicesclient-bnd-xmi when Web services is running as client) to refer to the certificate store defined in the default binding information. The Certificate Store Name must be unique to the collection certificate store collection.
- **Key locators**
  - Key locators specify implementation of the com.ibm.wsspi.wssecurity.config.KeyLocator interface. This interface is used to retrieve keys for signature or encryption. Customer implementation can be provided to extend the key locator interface to retrieve keys using other methods. WebSphere Application Server - Express provides implementations to retrieve a key from the key store, map an authenticated identity to a key in the key store, or retrieve a key from the signer certificate (the latter two are used for encrypting the response).
  - The Key Locator Name is used in the binding file (ibm-webservices-bnd.xmi and ibm-webservicesclient-bnd-xmi when Web services is running as client) to refer to the key locator defined in the default binding information. The Key Locator Name must be unique to the key locators collection in the default binding information.
- **Trusted ID evaluators**
  - Trusted ID evaluators are an implementation of the com.ibm.wsspi.wssecurity.id.TrustedIDEvaluator interface. This interface is used to make sure the identity-asserting authority is trusted. Additionally, you can extend the trusted identity evaluator to validate the trust. WebSphere Application Server - Express provides a default implementation for validating trust based on a pre-defined list of identities.
  - The Trusted ID Evaluator Name is used in the binding file (ibm-webservices-bnd.xmi) to refer to the trusted identity evaluator defined in the default binding information. The Trusted ID Evaluator Name must be unique to the Trusted ID Evaluator collection.
- **Login mappings**
  - The login mappings define the mapping of the AuthMethod to JAAS Login Configuration. The mappings are used to authenticate the incoming security token embedded in the Web services security SOAP message header. The JAAS Login Configuration is defined in the administrative console under **Security —> JAAS Configuration —> Application Logins**.
  - WebSphere Application Server - Express defines BasicAuth (authenticates user name and password), Signature (maps the subject distinguished name (DN) in the certificate to a WebSphere Application Server - Express credential), and IDAssertion (maps the identity to a WebSphere Application Server - Express credential). After identity authentication, the associated credential is used in the downstream call.

- This can be extended to authenticate custom security tokens by providing custom JAAS Login Configuration and using the com.ibm.wsspi.wssecurity.auth.module.WSSecurityMappingModule to create the principal and credential required by WebSphere Application Server - Express.
- If LoginConfig (AuthMethod) is defined in the IBM extension deployment descriptor (ibm-webservices-ext.xmi), but there are no login mapping bindings (ibm-webservices-bnd.xmi) defined for the AuthMethod, Web services security run time uses the login mapping defined in the default binding information.

In WebSphere Application Server - Express, each server has a copy of the ws-security.xml file (default binding information for Web services security). To navigate to the server-level default binding in the administrative console, click **Servers —> Application Servers —>** ***server_name*** **—> Web Services: Default bindings for Web Services Security**, where *server_name* is the name of your application server.

Figure 1: Web services security application level bindings and server level default binding information.


Web module
Web services implemented as an JavaBean file
ibm-webservices-ext.xml file
application-level binding
ws-security.xml file
overrides
ibm-services-bnd.xml file
Default configuration for server1

Web services security run time uses the binding information in the Web module binding file (ibm-webservices-bnd.xmi or ibm-webservicesclient-bnd.xmi if Web services is acting as client on the server) if the binding information is defined in the application level binding file. For example, if key locator K1 is defined in both the application level binding file and the default binding file (ws-security.xml), the K1 in the application level binding file is used.

Configure Web services authentication

WebSphere Application Server - Express provides the following authentication mechamisms for Web services:

- Basic authentication
- Identity assertion authentication
- Digital signature authentication

- Lightweight Third-party Authentication (LTPA)

For more information, see “Web services authentication method overview.”

You must configure a Web service and its clients to use the same authentication mechansim. The client creates a security token in the SOAP message, which is then extracted and validated by the server. For more information, see “Token type overview” on page 87.

You can configure a Web services server to support multiple authentication mechanisms. Additionally, a server can act as a client to another Web service, so in some cases you may need to configure both server-side and client-side authentication for a Web service application.

The authentication mechanism is configured in the Web service and Web services client deployment descriptors. You can use WebSphere Development Studio Client for iSeries (Version 5.1 or later) or the WebSphere Application Server Toolkit (Version 5.0.2 or later) to configure your deployment descriptors. These topics describe how to configure authentication mechanisms with the Development Studio Client. For more information, see “Configure your Web services application” on page 102.

See the following topics for information about configuring the various Web services authentication mechanisms:

“Configure basic authentication for Web services” on page 103
The basic authentication mechanism validates a security token with a user ID and text password. See this topic for more information.

“Configure identity assertion authentication” on page 108
The identity assertion mechanism validates a security token with an identity name only. The identity name can be a user name, a distinguished name (DN), or an X.509 certificate. See this topic for more information.

“Configure Web services digital signature authentication” on page 116
The digital signature mechanism uses a digital signature for authentication. See this topic for more information.

“Configure LTPA authentication for Web services” on page 122
The LTPA mechanism uses a binary security token for authentication. See this topic for more information.

As an alternative to the other, more complex Web services authentication mechanisms, you can use HTTP basic authentication to secure your Web services. For more information, see “Configure HTTP basic authentication for Web services” on page 137.

Web services authentication method overview: The Web Services Security implementation for WebSphere Application Server - Express supports the following authentication methods:

- **BasicAuth**
  When WebSphere Application Server - Express is configured to use the BasicAuth authentication method, the sender attaches the LTPA token as a BinarySecurityToken from the current security context or from basic authentication data configuration in the binding file in the Simple Object Access Protocol (SOAP) message header. The Web services security message receiver authenticates the sender by validating the user name and password against the configured user registry.
- **Identity assertion**
  The identity assertion authentication method, different from other three authentication methods, establishes the security credential of the sender based on the trust relationship. You can use the identity assertion authentication method, for example, when an intermediary server must invoke a service from a downstream server on behalf of the client, but does not have the client authentication information.

The intermediary server might establish a trust relationship with the downstream server and then assert the client identity to the same downstream server.

- **Digital signature**
  With the digital signature authentication method, the sender attaches a BinarySecurityToken from a X509 certificate to the Web services security message header along with a digital signature of the message body, time stamp, security token, or any combination of the three. The receiver authenticates the sender by verifying the validity of the X.509 certificate and the digital signature using the public key from the verified certificate.
- **Lightweight Third-Party Authentication (LTPA)**
  With the LTPA method, the sender attaches the LTPA BinarySecurityToken it previously received in the SOAP message header. The receiver authenticates the sender by validating the LTPA token and the token expiration time.

Web services security supports the following trust modes:

- BasicAuth
- Digital signature
- Presumed trust

When you use the BasicAuth and Digital signature trust modes, the intermediary server passes its own authentication information to the down stream server for authentication. The Presumed trust mode establishes a trust relationship using some external mechanism.For example, the intermediary server may pass Simple Object Access Protocol (SOAP) messages through a secure socket layers (SSL) connection with the downstream server and transport layer client certificate authentication.

The Web services security implementation for WebSphere Application Server - Express validates the trust relationship by following this procedure:

1. The downstream server first validates the authentication information of the intermediary server.
2. The downstream server verifies whether the authenticated intermediary server is authorized for identity assertion. For example, the intermediary server must be in the trust list for the downstream server.

The client identity may be represented by a name string, a distinguished name (DN), or an X.509 certificate. The client identity is attached in the Web services security message in a UsernameToken with just a user name, DN, or in a BinarySecurityToken of a certificate.

The following table summarizes the type of security token that is required for each authentication method.

Authentication Method	Security Token
BasicAuth	BasicAuth requires <wsse:UsernameToken> with <wsse:Username> and <wsse:Password>.
Signature	Signature requires <ds:Signature> and <wsse:BinarySecurityToken>.
IDAssertion	IDAssertion requires <wsse:UsernameToken> with <wsse:Username> or <wsse:BinarySecurityToken> with a X.509 certificate for client identity depending on <idType>. Also, it requires the following other security tokens according to the <trustMode>: • If the <trustMode> is BasicAuth, IDAssertion requires <wsse:UsernameToken> with <wsse:Username> and <wsse:Password>. • If the <trustMode> is Signature, IDAssertion requires <wsse:BinarySecurityToken>.

Authentication Method	Security Token
LTPA	LTPA requires <wsse:BinarySecurityToken> with an LTPA token.

Multiple authentication methods can be supported by a Web service simultaneously. The receiver-side Web services deployment descriptor can specify all the authentication methods that are supported in the ibm-webservices-ext.xmi XML file. The receiver-side Web services, as shown in the following example, is configured to accept all the authentication methods described previously:

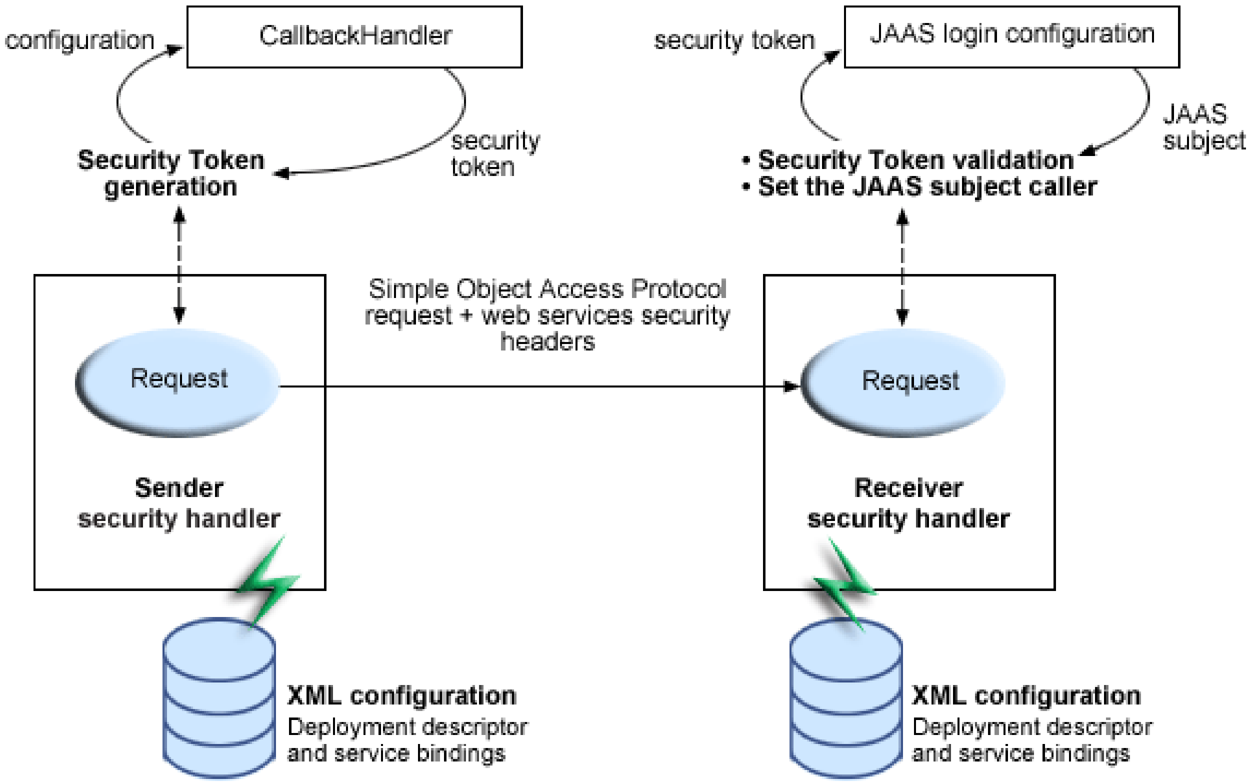
```
<loginConfig xmi:id="LoginConfig_1052760331326">
  <authMethods xmi:id="AuthMethod_1052760331326" text="BasicAuth"/>
  <authMethods xmi:id="AuthMethod_1052760331327" text="IDAssertion"/>
  <authMethods xmi:id="AuthMethod_1052760331336" text="Signature"/>
  <authMethods xmi:id="AuthMethod_1052760331337" text="LTPA"/>
</loginConfig>
<idAssertion xmi:id="IDAssertion_1052760331336" idType="Username" trustMode="Signature"/>
```

You can define only one authentication method in the sender-side Web services deployment descriptor. A Web service client can use any one of the authentication methods that are supported by the particular Web services application. The following example illustrates an identity assertion authentication method configuration in the Web service client deployment descriptor extension, ibm-webservicesclient-ext.xmi:

```
<loginConfig xmi:id="LoginConfig_1051555852697">
  <authMethods xmi:id="AuthMethod_1051555852698" text="IDAssertion"/>
</loginConfig>
<idAssertion xmi:id="IDAssertion_1051555852697" idType="Username" trustMode="Signature"/>
```

As shown in the previous example, the client identity type is `Username` and the trust mode is digital signature (`Signature`).

Figure 1: Security token generation and validation


configuration
CallbackHandler
security token
Security Token generation
security token
JAAS login configuration
JAAS subject
• Security Token validation
• Set the JAAS subject caller
Simple Object Access Protocol request + web services security headers
Request
Sender security handler
Request
Receiver security handler
XML configuration
Deployment descriptor and service bindings
XML configuration
Deployment descriptor and service bindings

The sender security handler invokes the handle() method of an implementation of the javax.security.auth.callback.CallbackHandler interface. The javax.security.auth.callback.CallbackHandler interface creates the security token and passes it back to the sender security handler. The sender security handler constructs the security token based on the authentication information in the callback array and inserts the security token into the Web services security message header.

The receiver security handler compares the token type in the message header with the expected token types configured in the deployment descriptor. If none of the expected token type are found in the Web services security header of the SOAP message, the request is rejected with an SOAP fault exception. Otherwise, the token type is used to map to a Java Authentication and Authorization Service (JAAS) login configuration for validating the token. If the authentication is successful, then a JAAS Subject is created and associated with the thread of execution. Otherwise, the request is rejected with an SOAP fault exception.

Configure your Web services application: WebSphere Development Studio Client for iSeries is a workstation-based graphical tool. It contains several WebSphere Studio tools that you can use to develop and configure your applications for WebSphere Application Server - Express. For more information, see WebSphere Development Studio Client for iSeries

(http://www.ibm.com/software/awdtools/wdt400/).

Note: You must use WebSphere Development Studio Client for iSeries, Version 5.1 or later.

To create or modify your Web services security configuration, you must edit the deployment descriptor for your application. Use the Web Services Editor in WebSphere Development Studio Client for iSeries.

If your application is a Web service, the deployment descriptor is the webservices.xml file. If your application is a Web services client, the deployment descriptor is the webservicesclient.xml file.

Perform the following steps in the Development Studio Client to open the deployment descriptor for editing:

1. In the Navigator panel, expand your Web module.

   If the Navigator is not shown, you can open it by clicking **Window —> Show View —> Navigator**.
2. Expand **WebContent —> WEB-INF**.
3. Open the deployment descriptor in the Web Services Editor:
   - If your application is a Web service, right-click the webservices.xml file, and select **Open With —> Web Services Editor**.
   - If your application is a Web service client, right-click the webservicesclient.xml file, and select **Open With —> Web Services Client Editor**.

In the Web Services Editor, there are several tabs at the bottom of the editor. Use the **Security Extensions** and **Bindings Configuration** tabs to configure Web services security.

Note: Although you open the webservices.xml or webservicesclient.xml file in the Web Services Editor, the Web services security configuration is written to the following files:

- For a Web service:
  - **ibm-webservices-ext.xmi**
    The security extensions configuration specifies what security is to be performed.
  - **ibm-webservices-bnd.xmi**
    The security bindings configuration indicates how to perform the security that is specified in the security extensions configuration.
- For a Web services client:
  - **ibm-webservicesclient-ext.xmi**
    The security extensions configuration specifies what security is to be performed.
  - **ibm-webservicesclient-bnd.xmi**
    The security bindings configuration indicates how to perform the security that is specified in the security extensions configuration.

Configure basic authentication for Web services: With the basic authentcation (BasicAuth) mechanism, the client generates a security token, based on user ID and password, and it imbeds the token in the SOAP message. The server extracts the token and uses a Java Authentication and Authorization Service (JAAS) login module to validate the token. For an overview of the basic authentication mechanism, see "Basic authentication for Web services."

Note: To use the basic authentication mechanism for Web services, you must configure WebSphere global security. For more information, see Configure global security in the *Security* topic.

Perform these steps to configure the basic authentication for your Web service:

1. "Configure basic authentication for the Web services client" on page 104
2. "Configure basic authentication for the Web services server" on page 106

Basic authentication for Web services: When you use the BasicAuth authentication method, the security token that is generated is a <wsse:UsernameToken> element with <wsse:Username> and <wsse:Password>elements. WebSphere Application Server - Express supports text passwords but not password digest because passwords are not stored and cannot be retrieved from the server.

On the request sender side, a callback handler is invoked to generate the security token. On the request receiver side, a Java Authentication and Authorization Service (JAAS) login module is used to validate the security token. These two operations, token generation and token validation, are described in the following topics.

BasicAuth token generation

The request sender generates a BasicAuth security token using a callback handler. The security token returned by the callback handler is inserted in the SOAP message. The callback handler that is used is specified in the <LoginBinding> element of the bindings file, ibm-webservicesclient-bnd.xmi. The following callback handler implementations are provided with WebSphere Application Server - Express and can be used with the BasicAuth authentication method:

- com.ibm.wsspi.wssecurity.auth.callback.GUIPromptCallbackHandler
- com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler
- com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler

You can add your own callback handlers that implement javax.security.auth.callback.CallbackHandler.

BasicAuth token validation

The request receiver retrieves the BasicAuth security token from the SOAP message and validates it using a JAAS login module. The <wsse:Username> and <wsse:Password> elements in the security token are used to perform the validation. If the validation is successful, the login module returns a JAAS Subject. This Subject then is set as the identity of the thread of execution. If the validation fails, the request is rejected with a SOAP fault exception.

The JAAS login configuration is specified in the <LoginMapping> element of the bindings file. There are default bindings specified in the ws-security.xml file. However, you can override these bindings using the application-specific ibm-webservices-bnd.xmi file.

The configuration information consists of a CallbackHandlerFactory and a ConfigName. The CallbackHandlerFactory specifies the name of a class that is used for creating the JAAS CallbackHandler object. WebSphere Application Server - Express provides the com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl CallbackHandlerFactory implementation. The ConfigName specifies a JAAS configuration name entry. WebSphere Application Server - Express searches the security.xml file for a matching configuration name entry. If a match is not found, it searches the wsjaas.conf file for a match. WebSphere Application Server - Express provides the WSLogin default configuration entry, which is suitable for the BasicAuth authentication method.

Configure basic authentication for the Web services client: This task is used to configure BasicAuth authentication. *BasicAuth* refers to the user ID and password of a valid user in the registry of the target server. Collection of BasicAuth information can occur in many ways including through a GUI prompt, a standard in (Stdin) prompt, or specified in the bindings, which prevents user interaction. For more information on BasicAuth authentication, see “Basic authentication for Web services” on page 103.

To select the BasicAuth authentication method for the Web services client, perform the following steps:

1. Open the webservicesclient.xml file in the Web Services Client Editor of the WebSphere Development Studio Client for iSeries. For more information, see “Configure your Web services application” on page 102.
2. Click the **Security Extensions** tab.
3. Expand the **Request Sender Configuration —> Login Config** settings. The only valid login configuration choices for a pure client are BasicAuth and Signature.

4. Select **BasicAuth** to authenticate the client using a user ID and password. This user ID and password must be specified in the target user registry. The other choice, **Signature**, attempts to authenticate the client with the certificate that is used to digitally sign the message.
5. Save the file.

Next, perform the following steps in the Web Services Client Editor to configure how the BasicAuth authentication information is collected:

1. Click the **Port Binding** tab.
2. Expand the **Security Request Sender Binding Configuration —> Login Binding** settings.
3. Click **Edit** or **Enable** to view the Login Binding information. The login binding information displays.
4. Configure the following settings:

Name	Purpose
Authentication method	The authentication method specifies the type of authentication that occurs. To use basic authentication, select **BasicAuth**.
Token value type URI and **Token value type local name**	When you select **BasicAuth**, you cannot edit the token value type URI and local name values. These values are specifically for custom authentication types. For BasicAuth authentication, you do not need to enter any information.
Callback handler	The callback handler specifies the Java Authentication and Authorization Server (JAAS) callback handler implementation for collecting the BasicAuth information. You can use the following default implementations for the callback handler: • **com.ibm.wsspi.wssecurity.auth.callback. StdinPromptCallbackHandler** This implementation is used for non-GUI console prompts. • **com.ibm.wsspi.wssecurity.auth.callback. GUIPromptCallbackHandler** This implementation is used for GUI panel prompts. • **com.ibm.wsspi.wssecurity.auth.callback. NonPromptCallbackHandler** This implementation is used when you plan to always enter the user ID and password in the BasicAuth user ID and password section that follows.
Basic Authentication user ID and **Basic Authentication password**	When values for BasicAuth user ID and password are entered, regardless of the default callback handler that is used, these user ID and password values are used to authenticate to the server for the Web services security authentication. If you leave these values blank, use either the GUIPromptCallbackHandler or the StdinPromptCallbackHandler implementation, but only on a pure client. Always fill in these values for any Web service that acts as a client to another Web service and you want to specify BasicAuth for authentication downstream. If you want the client identity of the originator to flow downstream, configure the Web service client to use ID assertion instead.

Name	Purpose
Property	This field enables you to enter properties and name and value pairs for use by custom callback handlers. For BasicAuth authentication, you do not need to enter any information.

5. (Optional) There is a basic authentication entry in the **Port Qualified Name Binding Details** section. This entry is used for HTTP transport authentication, which may be required if the router servlet is protected.

   Information specified in the **Web services security basic authentication** section overrides the basic authentication information specified in the **Port Qualified Name Binding Details** section for authorizing the Web service.

   For a server that acts as a client, do not specify a GUI or non-GUI prompt callback handler. To configure BasicAuth authentication from one Web service to a downstream Web service, select the **com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHander** implementation and explicitly specify the BasicAuth user ID and password.

   If you want the client identity of the originator to flow downstream, configure the Web service client to use identity assertion or Lightweight Third Party Authentication (LTPA) authentication instead.
6. Save the file.

Note: Examples may be wrapped for display purposes.

Configure basic authentication for the Web services server: This task is used to configure BasicAuth authentication at the server. BasicAuth refers to the user ID and password of a valid user in the registry of the target server. After a request is received that contains basic authentication information, the server needs to log in to form a credential. The credential is used for authorization. If the user ID and password supplied is invalid, an exception is thrown and the request ends without invoking the resource. For more information on BasicAuth authentication, see “Basic authentication for Web services” on page 103.

Perform the following steps to configure the server for BasicAuth authentication:

1. Open the webservices.xml deployment descriptor for your Web services application in the Web Services Editor of the WebSphere Development Studio Client for iSeries. For more information, see “Configure your Web services application” on page 102.
2. Click the **Security Extensions** tab.
3. Expand the **Request Receiver Service Configuration Details —> Login Config** settings. Select **BasicAuth** to authenticate the client using a user ID and password. The client must specify a valid user ID and password in the server user registry.

   Note: You can select multiple login configurations, which means that different types of security information might be received at the server. The order in which the login configurations are added decides the order in which they are processed when a request is received. This can cause problems if you have multiple login configurations added that have security tokens in common. For example, ID assertion contains a BasicAuth token. For ID assertion to work properly, list ID assertion ahead of BasicAuth in the processing list so the BasicAuth processing does not override the IDAssertion processing.

Next, use the Web Services Editor to specify how the BasicAuth authentication information is validated:

1. Click the **Binding Configurations** tab.
2. Expand the **Request Receiver Binding Configuration Details —> Login Mapping** settings.
3. Click **Edit** to view the login mapping information or click **Add** to add new login mapping information. The login mapping dialog appears.

4. Select or enter the following information:

Name	Purpose
Authentication method	The authentication method specifies the type of authentication that occurs. Select **BasicAuth** to use basic authentication.
Configuration name	This specifies the Java Authentication and Authorization Service (JAAS) login configuration name. For the BasicAuth authentication method, enter `WSLogin` for the JAAS login configuration name.
Use token value type	This option determines if you want to specify a custom token type. For the default authentication method selections, you do not need to specify this option.
Token value type URI and **Token value type URI local name**	When you select **BasicAuth**, you cannot edit the token value type URI and local name values. These values are specified for custom authentication types. For BasicAuth authentication, you do not need to enter any information for these fields.
Callback handler factory class name	This class name creates a JAAS CallbackHandler implementation that supports the following callbacks: • javax.security.auth.callback. NameCallback • javax.security.auth.callback. PasswordCallback • com.ibm.wsspi.wssecurity.auth.callback. BinaryTokenCallback • com.ibm.wsspi.wssecurity.auth.callback. XMLTokenReceiverCallback • com.ibm.wsspi.wssecurity.auth.callback. PropertyCallback For any of the default authentication methods (BasicAuth, ID assertion, and Signature), use the callback handler factory default implementation. Enter the following class name for any of the default Authentication methods including BasicAuth: `com.ibm.wsspi.wssecurity.auth.callback. WSCallbackHandlerFactoryImpl`. This implementation creates the correct callback handler for the default implementations.
Callback handler factory property name and **Callback handler factory property value**	This property is used to specify callback handler properties for custom callback handler factory implementations. You do not need to specify any properties for the default callback handler factory implementation. For BasicAuth, you do not need to enter any property values.
Login mapping property name and **Login mapping property value**	This property is used to specify properties for a custom login mapping to use. For the default implementations including BasicAuth, you do not need to enter any property values.

5. Save the file.

Note: Examples may be wrapped for display purposes.

Configure identity assertion authentication: With identity assertion authentication, the client generates a security token, based on user name, distinguished name (DN), or X.509 certificate, and imbeds it in the SOAP message. The server then extracts the token and validates it by using a Java Authentication and Authorization Service (JAAS) login module. For more information about identity assertion, see "Identity assertion" and "Identity authentication method for Web services" on page 109.

Identity assertion uses a trusted ID evaluator to determine if the name that is provided in the request message is to be trusted. You can use a default trusted ID evaluator, or you can develop your own. For more information, see "Trusted ID evaluator" on page 110.

Note: To use the identity assertion authentication mechanism for Web services, you must configure WebSphere global security. For more information, see Configure global security in the *Security* topic.

To configure the identity assertion authentication mechanism for your Web service, perform the following steps:

1. "Configure identity assertion authentication for a Web services client" on page 112
2. "Configure the server for Web services identity assertion authentication" on page 113

Identity assertion: Identity assertion is a method for expressing the identity of the sender (for example, user name) in a Simple Object Access Protocol (SOAP) message. When identity assertion is used as a authentication method, the authentication decision is performed based only on the name of the identity, but not on other information such as passwords and certificates.

ID type

The Web services security implementation in WebSphere Application Server - Express supports the following types of identity:

- **User name**
  Denotes the user name, such as the one in the local operating system (for example, `alice`). This name is embedded in the <Username> element within the <UsernameToken> element.
- **DN**
  Denotes the distinguished name (DN) for the user, such as `CN=alice, O=IBM, C=US`. This name is embedded in the <Username> element within the <UsernameToken> element.
- **X.509 certificate**
  Represents the identity of the user as a X.509 certificate instead of a string name. This certificate is embedded in the <BinarySecurityToken> element.

Managing trust

The intermediary host in the SOAP message itinerary can assert the initial sender's claimed identity. Two methods (called trust mode) are supported for this assertion:

- **Basic authentication**
  The intermediary adds its user name and password pair to the message.
- **Signature**
  The intermediary digitally signs the <UsernameToken> element of the initial sender.

Note: This trust mode does not support the X.509 certificate ID type.

In addition to the trust mode, the ultimate receiver can evaluate the trustworthiness of the asserting identity (rather than the initial sender identity) using the trusted ID evaluator. For the details about the trusted ID evaluator, see "Trusted ID evaluator" on page 110.

Typical scenario

ID assertion is typically used in the multi-hop environment where the SOAP message passes through one or more intermediary hosts. The intermediary host authenticates the initial sender. The following scenario describes the process:

1. The initial sender sends a SOAP message to the intermediary host with some embedded authentication information. This authentication information may be a user name and password pair and an LTPA token.
2. The intermediary host authenticates the initial sender according to the embedded authentication information.
3. The intermediary host removes the authentication information from the SOAP message and replaces it with the <UsernameToken> element, which contains a user name.
4. The intermediary host asserts the trust according to the trust mode.
5. The intermediary host sends the updated SOAP message to the ultimate receiver.
6. The ultimate receiver checks the trust against the intermediary host information according to the configured trust mode. Also, the trusted ID evaluator is invoked.
7. If trust is established by the ultimate receiver, it invokes the Web service under the authorization of the user name (that is, the initial sender) in the SOAP message.

Identity authentication method for Web services: When using the Identity Assertion (IDAssertion)authentication method, the security token generated is a <wsse:UsernameToken> element that contains a <wsse:Username> element. On the request sender side, a callback handler is invoked to generate the security token. On the request receiver side, the security token is validated. These two operations, token generation and token validation, are described in the following topics.

Identity Assertion token validation

The request receiver retrieves the IDAssertion security token from the Simple Object Access Protocol (SOAP) message and validates it using a Java Authentication and Authorization Service (JAAS) login module. With identity assertion, special processing is required to establish trust before asserting the identity as the established identity of the thread of execution. This special processing misdefined by the <IDAssertion> element in the deployment descriptor file, ibm-webservices-ext.xmi. If all the validation checks are successful,the asserted identity is set as the identity of the thread of execution. If the validation fails, the request is rejected with a SOAP fault exception.

The JAAS login configuration is specified in the <LoginMapping> element of the bindings file. There are default bindings specified in the ws-security.xml file. However, you can override these bindings using the application specific ibm-webservices-bnd.xmi file. The configuration information consists of a CallbackHandlerFactory and a ConfigName. The CallbackHandlerFactory specifies the name of a class that is used for creating the JAAS CallbackHandler object. WebSphere Application Server - Express provides the com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl CallbackHandlerFactory implementation. The ConfigName specifies a JAAS configuration name entry. WebSphere Application Server - Express searches the security.xml file for a matching configuration name entry. If a match is not found it searches the wsjaas.conf file. WebSphere Application Server - Express provides the system.wssecurity.IDAssertion default configuration entry, which is suitable for the identity assertion authentication method.

The <IDAssertion> element in the ibm-webservices-ext.xmi deployment descriptor file specifies the special processing required when using the identity assertion authentication method. The <IDAssertion> element is composed of two sub-elements:<IDType> and <TrustMode>.

The <IDType> element specifies the method for asserting the identity. The supported values for asserting the identity are:

- Username
- Distinguished Name (DN)
- X509Certificate

When <IDType> is `username`, a username token (for example, Bob) is provided. This user name is mapped to a user in the user registry and is the asserted identity after successful trust validation. When the <IDType> is `DN`, a user name token containing a distinguished name is provided (for example, cn=Bob Smith, o=ibm, c=us). This DN is mapped to a user in the user registry and this user is the asserted identity after successful trust validation. When the<IDType> is `X509Certificate`, a binary security token containing an X509certificate is provided and the SubjectDN from the certificate (for example, cn=Bob Smith, o=ibm, c=us) is extracted. This Subject DN is mapped to a user in the user registry and this user is the asserted identity after successful trust validation.

The <TrustMode> element specifies how the trust authority, or asserting authority, provides trust information. The supported values are:

- Signature
- BasicAuth
- (No value specified)

When <TrustMode> is `Signature`, the signature is validated. Then, the signer (for example, cn=IBM Authority, o=ibm, c=us) is mapped to an identity in the user registry (for example, IBMAuthority). To ensure that the asserting authority is trusted, the mapped identity (for example, IBMAuthority) is validated against a list of trusted identities. When the <TrustMode> is `BasicAuth`, there is a user name token with a username and password, which is the user name and password of the asserting authority. The user name and password are validated. If they are successfully validated, that user name (for example, IBMAuthority) is validated against a list of trusted identities. If a value is not specified for <TrustMode>, trust is presumed and additional trust validation is not performed. This type of identity assertion is called *presumed trust mode*. Use the presumed trust mode only in an environment where the trust is established using some other mechanism.

If all the validations described previously succeed, the asserted identity (for example, Bob) is set as the identity of the thread of execution. If any of the validations fail, the request is rejected with a SOAP fault exception.

Trusted ID evaluator: Trusted ID evaluator (com.ibm.wsspi.wssecurity.id.TrustedIDEvaluator) is a abstraction of the mechanism that evaluates whether the given ID name is trusted. Depending upon the implementation, various types of infrastructure can be used to store a list of the trusted IDs are stored, such as:

- Plain text file
- Database
- LDAP server

The trusted ID evaluator is typically used by the ultimate receiver in a multi-hop environment. The Web services security implementation invokes the trusted ID evaluator and passes the identity name of the intermediary as a parameter. If the identity is evaluated and deemed trustworthy, the procedure continues. Otherwise, an exception is thrown and the procedure is aborted.

Trusted ID evaluator default implementation

A trusted ID evaluator is used to determine if a given identity (ID) name is trusted. Trusted ID evaluators are implemented by providing a class that implements the com.ibm.wsspi.wssecurity.id.TrustedIDEvaluator interface.

The default implementation of a trusted ID evaluator is com.ibm.wsspi.wssecurity.id.TrustedIDEvaluatorImpl. This implementation is initialized with a list of trusted identity names. You can use `trustedId_`*n* as the property key name (where *n* is an integer greater than 0) to specify a list of trusted identities in the properties. When a name is to be evaluated, it is passed to the evaluate() method. The name is checked against the list of trusted names and returns `true` if it is in the list (this means it is trusted) and `false` if it is not in the list (this means it is not trusted). The trusted identities are specified as TrustedIDEvaluator properties of the Web Services Security binding file (ws-security.xml or ibm-webservices-bnd.xmi).

Developing a trusted ID evaluator

Perform the following steps to develop your own trusted ID evaluator:

1. Define the trusted ID evaluator class method. WebSphere Application Server - Express provides the trusted ID evaluator interface, com.ibm.wsspi.wssecurity.id.TrustedIDEvaluator, which defines the following methods:
   - `public void init(java.util.Map map) throws SoapSecurityException`
     This method initializes the object. The parameter map object contains name and value pairs.

     These pairs are specified in the WebSphere administrative console. Click **Application Servers —>** ***server_name*** **—> Web Services: Default bindings for Web Services Security —> Trusted ID Evaluators —>** ***trusted_ID_evaluator_name*** **—> Properties —> New**, where *server_name* is the name of your server and *trusted_ID_evaluator_name* is the name of your implementation.
   - `boolean evaluate(String id) throws TrustedIDEvaluatorException`
     This method evaluates whether the received ID is trusted. The parameter object is an ID that must be evaluated. You can specify the realm as "id@realm". The method returns a `true` value if the ID is trusted, otherwise, it returns a `false` value.

   You must configure the following methods that are implemented by the custom trusted ID evaluator implementation.

   Note: This listing only shows the methods and does not include any implementation.

   ```
   import com.ibm.wsspi.wssecurity.SoapSecurityException;
   import com.ibm.wsspi.wssecurity.id.TrustedIDEvaluator;
   import com.ibm.wsspi.wssecurity.id.TrustedIDEvaluatorException;
   import java.util.Map;

   public class MyTIEImpl implements TrustedIDEvaluator {
     public void init(Map map) throws SoapSecurityException {
       // Initialize the trusted ID evaluator object.
     }

     public boolean evaluate(String id) throws TrustedIDEvaluatorException {
       // Evaluate the given ID and return true if successful, or false otherwise.
     }
   }
   ```
2. Compile the implementation. Make sure that the /QIBM/ProdData/WebASE/ASE5/lib/was-wssecurity.jar file is in the compiler class path.
3. Copy the class file to a location in the class path, perferably in the /QIBM/UserData/WebASE/ASE5/*instance*/lib/ext directory, where *instance* is the name of your instance.
4. Restart your application server.
5. Delete the default trusted ID evaluator that is configured in the administrative console. Click **Application Servers —>** ***server_name*** **—> Web Services: Default bindings for Web Services Security —> Trusted ID Evaluators —>** ***trusted_ID_evaluator_name***, where *server_name* is the name of your application server, and *trusted_ID_evaluator_name* is the name of the default trusted ID evaluator.

   Select the box next to the specific trusted ID evaluator name and click **Delete**.
6. To add your custom trusted ID evaluator, click **New**. Verify that the class name is dot separated and appears in the class path.
7. Under **Additional Properties**, click **Properties** to add additional properties that are required to initialize the custom trusted ID evaluator. These properties are passed to the `init(java.util.Map)` method of your implementation when it extends the com.ibm.wsspi.wssecurity.id.TrustedIDEvaluator interface as described in the first step.

8. Save the configuration.
9. Restart the application server for the trusted ID evaluator to take effect.

Configure identity assertion authentication for a Web services client: This task is used to configure identity assertion authentication. The purpose of identity assertion is to assert the authenticated identity of the originating client from a Web service to a downstream Web service. Do not attempt to configure identity assertion from a pure client. Identity assertion works only when you configure on the client-side of a Web service acting as a client to a downstream Web service.

In order for the downstream Web service to accept the identity of the originating client (just the user name), you must supply a special trusted BasicAuth credential that the downstream Web service trusts and can authenticate successfully. You must specify the user ID of the special BasicAuth credential in a trusted ID evaluator on the downstream Web service configuration. For more information on trusted ID evaluators, see "Trusted ID evaluator" on page 110.

Perform the following steps in the WebSphere Development Studio Client for iSeries to specify identity assertion authentication for your Web services client:

1. Open the webservicesclient.xml file in the Web Services Client Editor of the WebSphere Development Studio Client for iSeries. For more information, see "Configure your Web services application" on page 102.
2. Click the **Security Extensions** tab.
3. Expand the **Request Sender Configuration —> Login Config** settings.
4. Select **IDAssertion** as the authentication method.
5. Expand the **Identity Assertion** section.
6. For the **ID Type**, select **Username**. This works with all registry types and originating authentication methods.
7. For the **Trust Mode**, select either **BasicAuth** or **Signature**.
   - If you select **BasicAuth**, you must include basic authentication information (user ID and password), which the downstream Web service has specified in the trusted ID evaluator as a trusted user ID. You specify the user ID and password information later, on the **Port Binding** tab.
   - If you select **Signature**, the certificate configured in the **Signature Information** section used to sign the data also is used as the trusted subject. The Signature is used to create a credential and the user ID, which the certificate mapped to the downstream registry, is used in the trusted ID evaluator as a trusted user ID.
8. Save the file.

Next, perform the following steps with the Web Services Client Editor to specify how the identity assertion informatino is collected:

1. Click the **Port Binding** tab.
2. Expand the **Security Request Sender Binding Configuration —> Login Binding** settings.
3. Click **Edit** to view the login binding information and select **IDAssertion**. The login binding dialog displays.
4. Select or enter the following information:

Name	Purpose
Authentication method	The authentication method specifies the type of authentication that occurs. Select **IDAssertion** to use identity assertion.

Name	Purpose
Token value type URI and **Token value type Local name**	When you select IDAssertion, you cannot edit the token value type URI and the local name. These values are specifically for custom authentication types. For IDAssertion authentication, you do not need to enter any information.
Callback handler	The callback handler specifies the Java Authentication and Authorization Service (JAAS) callback handler implementation for collecting the BasicAuth information. Specify the `com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler` implementation for IDAssertion.
Basic authentication User ID and **Basic authentication Password**	If the trust mode entered in the extensions is BasicAuth, you must specify the trusted user ID and password in these fields. The user ID specified must be an ID that is trusted by the downstream Web service. The Web service trusts the user ID if it is entered as a trusted ID in a trusted ID evaluator in the downstream Web service bindings. If the trust mode entered in the extensions is Signature, you do not need to specify any information in this field.
Property Name and **Property Value**	This field enables you to enter properties and name and value pairs, for use by custom callback handlers. For IDAssertion, you do not need to specify any information in this field.

5. Save the file.

Note: Examples may be wrapped for display purposes.

Configure the server for Web services identity assertion authentication: Use this task to configure identity assertion authentication. The purpose of identity assertion is to assert the authenticated identity of the originating client from a Web service to a downstream Web service. Do not attempt to configure identity assertion from a pure client.

For the downstream Web service to accept the identity of the originating client (user name only), you must supply a special trusted BasicAuth credential that the downstream Web service trusts and can authenticate successfully. You must specify the user ID of the special BasicAuth credential in a trusted ID evaluator on the downstream Web service configuration. For more information on trusted ID evaluators, see “Trusted ID evaluator” on page 110.

The server side passes the special BasicAuth credential into the trusted ID evaluator, which returns true or false that this ID is trusted. After it is trusted, the user name of the client is mapped to the credential, which is used for authorization.

Perform these steps to configure the server for identity assertion authentication:

1. Open the webservices.xml deployment descriptor for your Web services application in the Web Services Editor of the WebSphere Development Studio Client for iSeries. For more information, see “Configure your Web services application” on page 102.
2. Click the **Security Extensions** tab.
3. Expand the **Request Receiver Service Configuration Details —> Login Config** settings.
4. Select **IDAssertion** to authenticate the client using the identity assertion data provided. This user ID of the client must be in the target user registry configured in WebSphere Application Server - Express global security. You can select global security in the Administrative Console by clicking **Security —> Global security**.

Note: You can select multiple login configurations, which means that different types of security information can be received at the server. The order in which the login configurations are added decides the order in which they are processed when a request is received. This can cause problems if you have multiple login configurations added that have security tokens in common. For example, ID assertion contains a BasicAuth token, which is the token that is being trusted. For ID assertion to work properly, you must list ID assertion ahead of BasicAuth in the list or BasicAuth processing overrides ID assertion processing.

5. Expand the **IDAssertion** section. You need to select both the **ID Type** and **Trust Mode**:
   - For **ID Type**, the options are:
     - **Username**
     - **DN** (distinguished name)
     - **X509Certificate**

     These choices are just preferences and are not guaranteed. Most of the time **Username** is used. You must choose the same **ID Type** as the client.
   - The **Trust Mode** refers to the information sent by the client as the trusted ID. For **Trust Mode**, the options are as follows:
     - If you select **BasicAuth**, the client sends basic authentication data (user ID and password). This BasicAuth data is authenticated to the configured user registry. After the authentication occurs successfully, the user ID must be part of the trusted ID evaluator trust list.
     - If you select **Signature**, the client signing certificate is sent. This certificate must be mappable to the configured user registry. For **Local OS**, the common name (CN) of the distinguished name (DN) is mapped to a user ID in the registry. For **LDAP**, the DN is mapped to the registry for the ExactDN mode. If it is in the certificateFilter mode, attributes are mapped accordingly. In addition, the user name from the credential generated must be in the Trusted ID Evaluator trust list.
6. Save the file.

Next, perform the following steps in the Web Services Editor to specify how the identity assertion authentication information is validated.

1. Click the **Binding Configurations** tab.
2. Expand the **Request Receiver Binding Configuration Details —> Login Mapping** settings.
3. Click **Edit** to view the login mapping information. Click **Add** to add new login mapping information. The login mapping dialog displays.
4. Select or enter the following information:

Name	Purpose
Authentication method	The authentication method specifies the type of authentication that occurs. Select **IDAssertion** to use identity assertion authentication.
Configuration name	This specifies the JAAS login configuration name. For the IDAssertion authentication method, enter `system.wssecurity.IDAssertion` for the Java Authentication and Authorization Service (JAAS) login configuration name.
Use Token value type	This option determines if you want to specify a custom token type. For the default authentication method selections, you do not need to specify this option.
Token value type URI and **Token value type local name**	When you select ID assertion, you cannot edit these values. These values are specifically for custom authentication types. For the ID assertion authentication method, you do not need to enter any information in these fields.

Name	Purpose
Callback Handler Factory Class name	This class name creates a JAAS CallbackHandler implementation that supports the following callbacks: • javax.security.auth.callback.NameCallback • javax.security.auth.callback.PasswordCallback • com.ibm.wsspi.wssecurity.auth.callback.BinaryTokenCallback • com.ibm.wsspi.wssecurity.auth.callback.XMLTokenReceiverCallback • com.ibm.wsspi.wssecurity.auth.callback.PropertyCallback For any of the default Authentication methods (BasicAuth, IDAssertion, and Signature), use the callback handler factory default implementation. Enter the following class name for any of the default authentication methods including IDAssertion: `com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl`. This implementation creates the correct callback handler for the default implementations.
Callback handler factory property name and **Callback handler factory property value**	This property is used to specify callback handler properties for Custom callback handler factory implementations. The default callback handler factory implemetation does not need any properties to be specified. For ID assertion, you do not need to enter any values for this property.
Login mapping property name and **Login mapping property value**	This option is used to specify properties for a custom login mapping. For the default implementations including IDAssertion, you do not need to enter any properties for this option.

5. Expand the **Trusted ID Evaluator** section. Click **Edit** to see a dialog displaying all the trusted ID evaluator information. Specify or enter the following information:

Name	Purpose
Class name	The classname refers to the implementation of the trusted ID evaluator that you want to use. Enter the default implementation as `com.ibm.wsspi.wssecurity.id.TrustedIDEvaluatorImpl`. If you want to implement your own trusted ID evaluator, you must implement the com.ibm.wsspi.wssecurity.id.TrustedIDEvaluator interface.
Property name	The name is the name of this configuration. Enter `BasicIDEvaluator`.

Name	**Purpose**
Property value	The property defines name and value pairs that can be used by the trusted ID evaluator implementation. For the default implementation, the trusted list is defined here. When a request comes in and the trusted ID is verified, the user ID, as it appears in the user registry, must be listed in this property. Specify the property as a name and value pair where the name is `trustedId_`*`n`*, where *n* is an integer (starting from 0) and the value is the user ID associated with that name. Here is an example list of the trusted names: • `trustedId_0 = user1` • `trustedId_1 = user2` These values mean that both user1 and user2 are trusted. Both user 1 and user2 must be listed in the configured user registry.

6. Expand the **Trusted ID Evaluator Reference** section. Click **Enable** to add a new entry. The text you enter or the **Trusted ID Evaluator Reference** must be the same as the name entered previously in the **Trusted ID Evaluator** field. Make sure that the name matches exactly because as the information is case sensitive. If an entry is already specified, you can change it by clicking **Edit**.

Note: Examples may be wrapped for display purposes.

Configure Web services digital signature authentication: With digital signature authentication, the client generates a security token, based on a digital signature, and embeds it in the SOAP message. For more information about digital signatures, see “XML digital signature.”

The server then extracts the token and validates it by using a Java Authentication and Authorization Service (JAAS) login module. For more information, see “Digital signature authentication method for Web services” on page 117.

Note: To use the digital signature authentication mechanism for Web services, you must configure WebSphere global security. For more information, see Configure global security in the *Security* topic.

Perform the following steps to configure the digital signature authentication mechanism for your Web service:

1. “Configure the Web services client for signature authentication” on page 118
2. “Configure the server for Web service signature authentication” on page 120

XML digital signature: XML-Signature Syntax and Processing (XML signature) is a specification that defines XML syntax and processing rules to sign and verify digital signatures for digital content. The specification was developed jointly by the World Wide Web Consortium (W3C) and the Internet Engineering Task Force (IETF).

XML signature does not introduce new cryptographic algorithms. WebSphere Application Server - Express uses XML signature with existing algorithms such as RSA, HMAC, and SHA1. XML signature defines many methods for describing key information and enables a new method to be defined.

XML cannonicalization (c14n) is often needed when you use XML signature. Information can be represented in various ways within serialized XML documents. For example, although their octet representations are different, the following examples are identical:

- <person first=“John” last=“Smith”/>
- <person last=“Smith” first=“John”></person>

C14n is a process used to cannonicalize XML information. Select an appropriate c14n algorithm because the information that is cannonicalized is dependent upon this algorithm. One of major c14n algorithms, Exclusive XML Canonicalization, canonicalizes the character encoding scheme, attribute order, namespace declarations and so on. The algoritm does not canonicalize whitespace outside tags, namespace prefixes, or data type representation. For more information, see Exclusive XML Canonicalization

(http://www.w3.org/TR/2002/REC-xml-exc-c14n-20020718/).

XML Signature in Web Services Security-Core

The Web Services Security-Core (WSS-Core) specification defines a standard way for SOAP messages to incorporate an XML Signature. You can use almost all of th XML signature features in WSS-Core except enveloped signature and enveloping signature. However, WSS-Core has some recommendations such as exclusive canonicalization for the c14n algorithm and some additional features such as SecurityTokenReference and KeyIdentifier.

By including XML Signature in SOAP messages, the following goals are realized:

- **Message integrity**
  A message receiver can confirm that attackers or accidents have not altered parts of the message after they were signed by a key.
- **Authentication**
  You can assume that a valid signature is proof of possession. If a message has a digital certificate that is issued by a certificate authority and a signature in the message is validated successfully by a public key in the certificate, it is a proof that the signer has the corresponding private key. The receiver can authenticate the signer by checking the trustworthiness of the certificate.

XML signature in the current implementation

XML signature is supported in Web services security, however, an API is not available. The current implementation has many hard-coded behaviors and has some user-operable configuration items. To configure the client for digital signature, see “Configure the Web services client for response digital signature verification” on page 153. To configure the server for digital signature, see “Configure the Web services server for request digital signature verification” on page 155.

Security considerations

In a replay attack, an attacker taps the lines, receives a signed message, and then returns the message to the receiver. In this case, the receiver receives the same message twice and might process both of them if the signatures are valid. It can cause damage to the receiver if the message is a claim for money. If you have the signed generation time stamp and the signed expiration time in a message replay attacks may be reduced.

However, this is not a complete solution. A message must have a nonce value to prevent these attacks and the receiver must reject a message that contains a processed nonce. The current implementation does not provide a standard way to generate and check nonces in messages. Applications should handle nonces (such as serial numbers) and they should be signed.

Digital signature authentication method for Web services: When using the signature authentication method, the security token is generated with a <ds:Signature> and a <wsse:BinarySecurityToken> element. On the request sender side, a callback handler is invoked to generate the security token. On the request receiver side, a Java Authentication and Authorization Service (JAAS) login module is used to validate the security token. These two operations, token generation and token validation, are described in the following topics.

Signature token generation

The request sender generates a Signature security token using a callback handler. The security token returned by the callback handler is inserted in the SOAP message. The callback handler is specified in the <LoginBinding> element of the bindings file, ibm-webservicesclient-bnd.xmi. WebSphere Application Server - Express provides the following callback handler implementation that can be used with the Signature authentication method: com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler.

You can add your own callback handlers that implement javax.security.auth.callback.CallbackHandler.

Signature token validation

The request receiver retrieves the Signature security token from the SOAP message and validates it using a JAAS login module. The <ds:Signature> and <wsse:BinarySecurityToken> elements in the security token are used to perform the validation. If the validation is successful, the login module returns a JAAS Subject. This Subject then is set as the identity of the thread of execution. If the validation fails, the request is rejected with a SOAP fault exception.

The JAAS login configuration is specified in the <LoginMapping> element of the bindings file. There are default bindings specified in the ws-security.xml file. However, you can override these bindings using the application-specific ibm-webservices-bnd.xmi file.

The configuration information consists of a CallbackHandlerFactory and a ConfigName. The CallbackHandlerFactory specifies the name of a class that is used for creating the JAAS CallbackHandler object. WebSphere Application Server - Express provides the com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImp CallbackHandlerFactory implementation. The ConfigName specifies a JAAS configuration name entry. WebSphere Application Server - Express searches in the security.xml file for a matching configuration name entry. If a match is not found, it searches the wsjaas.conf file. WebSphere Application Server - Express provides the system.wssecurity.Signature default configuration entry, which is suitable for the signature authentication method.

Configure the Web services client for signature authentication: This task is used to configure signature authentication. A signature refers to the use of an X509 certificate to login on the target server. For more information on signature authentication, see “Digital signature authentication method for Web services” on page 117.

Perform the folowing steps in the WebSphere Development Studio Client for iSeries to specify signature authentication for your Web service client:

1. Open the webservicesclient.xml file in the Web Services Client Editor of the WebSphere Development Studio Client for iSeries. For more information, see “Configure your Web services application” on page 102.
2. Click the **Security Extensions** tab.
3. Expand the **Request Sender Configuration —> Login Config** settings. Select **Signature** to authenticate the client using the certificate used to digitally sign the request.
4. Save the file.

Next, perform the following steps in the Web Services Client Editor to specify how the signature authentication information is collected:

1. Click the **Port Binding** tab.
2. Expand **Security Request Sender Binding Configuration —> Signing Information** and click **Edit** to display and modify the signing key name and signing key locator.

To create new signing information, click **Enable**. The certificate that is sent to login at the server is the one configured in the Signing Information panel. For more information about how the signing key name maps to a key within the key locator entry, see "Configure a key locator" on page 139.

The following table describes the purpose of this information. Some of these definitions are based on the XML-Signature Syntax and Processing specification

(http://www.w3.org/TR/xmldsig-core).

Name	Purpose
Canonicalization method algorithm	The canonicalization method algorithm is used to canonicalize the SignedInfo element before it is digested as part of the signature operation.
Digest method algorithm	The digest method algorithm is the algorithm applied to the data after transforms are applied, if specified, to yield the <DigestValue>. The signing of the DigestValue binds resource content to the signer key. The algorithm that is selected for the client request sender configuration must match the algorithm that is selected in the server request receiver configuration.
Signature method algorithm	The signature method is the algorithm that is used to convert the canonicalized <SignedInfo> into the <SignatureValue>. The algorithm that is selected for the client request sender configuration must match the algorithm that is selected in the server request receiver configuration.
Signing key name	The signing key name represents the key entry associated with the signing key locator. The key entry refers to an alias of the key, which is used to sign the request.
Signing key locator	The signing key locator represents a reference to a key locator implementation. For more information on configuring key locators, see "Configure a key locator" on page 139.

3. Expand the **Security Request Sender Binding Configuration —> Login Binding** settings.
4. Click **Edit** to view the Login Binding information. The login binding information is displayed.
5. Select or enter the following information:

Name	Purpose
Authentication method	The authentication method specifies the type of authentication that occurs. Select **Signature** to use signature authentication.
Token value type URI and **Token value type URI local name**	When you select **Signature**, you cannot edit the **Token value type URI** and **Local name** values. These values are specifically for custom authentication types. For signature authentication, you do not need to enter any information.
Callback handler	The callback handler specifies the Java Authentication and Authorization Server (JAAS) callback handler implementation for collecting signature information. Enter the following callback handler for signature authentication: `com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler`. This callback handler is used because signature does not require user interaction.

Name	Purpose
Basic authentication User ID and **Basic authentication Password**	Do not enter anything in the BasicAuth fields when Signature authentication is desired.
Property Name and **Property Value**	This field enables you to enter properties and name and value pairs for use by custom callback handlers. For signature authentication, you do not need to enter any information.

6. (Optional) There is a basic authentication entry in the Port Qualified Name Binding Details section. This entry is used for HTTP transport authentication, which may be required if the router servlet is protected.

   Information that is specified in the Web services security signature authentication section overrides the basic authentication information that is specified in the Port Qualified Name Binding Details section for authorizing the Web service.

   If you want the signature identity of this client to flow downstream, configure the first Web service client to use ID assertion or Lightweight Third Party Authentication (LTPA) authentication instead.

Note: Examples may be wrapped for display purposes.

Configure the server for Web service signature authentication: This task is used to configure signature authentication at the server. *Signature* refers to the an X.509 certificate sent by the client to the server. The certificate is used to authenticate to the user registry configured at the server. After a request is received by the server that contains certificate, the server needs to log in to form a credential. The credential is used for authorization. If the certificate supplied cannot be mapped to an entry in the user registry, an exception is thrown and the request ends without invoking the resource. For more information, see "Digital signature authentication method for Web services" on page 117.

Perform the following steps in the WebSphere Development Studio Client for iSeries to configure the server for Web services signature authentication:

1. Open the webservices.xml deployment descriptor for your Web services application in the Web Services Editor of the WebSphere Development Studio Client for iSeries. For more information, see "Configure your Web services application" on page 102.
2. Click the **Security Extensions** tab.
3. Expand the **Request Receiver Service Configuration Details —> Login Config** settings. Select **Signature** to authenticate the client using an X509 certificate.

   The certificate that is sent from the client is the certificate used for signing the message. You must be able to map this certificate to the configured user registry. For Local OS, the common name (cn) of the distinguished name (DN) is mapped to a user ID in the registry. For LDAP, you can configure multiple mapping modes:

   - **EXACT_DN**
     This default mode directly maps the DN of the certificate to an entry in the LDAP server.
   - **CERTIFICATE_FILTER**
     With this mode, the LDAP advanced configuration has a place to specify a filter that maps specific attributes of the certificate to specific attributes of the LDAP server.
4. Save the file.

Next, perform the following steps in the Web Services Editor to specify how the signature authentication information is validated:

1. Click the **Binding Configurations** tab.
2. Expand the **Request Receiver Binding Configuration Details —> Login Mapping** settings.
3. Click **Edit** to view the login mapping information or click **Add** to add new login mapping information. The login mapping dialog is displayed.

4. Select or enter the following information:

Name	Purpose
Authentication method	The authentication method specifies the type of authentication that will occur. Select **Signature** to use signature authentication.
Configuration name	This specifies the Java Authentication and Authorization Service (JAAS) login configuration name. For the signature authentication method, enter `system.wssecurity.Signature` for the JAAS login configuration name. This specification logs in with the com.ibm.wsspi.wssecurity.auth.module.SignatureLoginModule JAAS login module.
Use Token value type	This determines if you want to specify a custom token type. For the default authentication method selections, you do not need to specify a value.
URI and **Local name**	When you select **Signature**, you cannot edit the token value type URI and local name values. These values are specifically for custom authentication types. For signature authentication, you do not need to enter any information.
Callback Handler factory class name	This class name creates a JAAS CallbackHandler implementation that understands the following callback handlers: • javax.security.auth.callback.NameCallback • javax.security.auth.callback.PasswordCallback • com.ibm.wsspi.wssecurity.auth.callback.BinaryTokenCallback • com.ibm.wsspi.wssecurity.auth.callback.XMLTokenReceiverCallback • com.ibm.wsspi.wssecurity.auth.callback.PropertyCallback For any of the default Authentication methods (BasicAuth, IDAssertion, Signature), use the callback handler factory default implementation. Enter the following class name for any of the default authentication methods including signature: `com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl`. This implementation creates the correct callback handler for the default implementations.
Callback handler factory property name and **Callback handler factory property value**	This field is used to specify callback handler properties for custom callback handler factory implementations. You do not need to specify any properties for the default callback handler factory implementation. For signature, you do not need to enter any properties for this field.
Login mapping property name and **Login mapping property value**	This field is used to specify properties for a custom login mapping to use. For the default implementations including signature, you do not need to enter any properties for this field.

5. Save the file.

Note: Examples may be wrapped for display purposes.

Configure LTPA authentication for Web services: With the LTPA authentication mechanism, the client generates a binary security token, and it imbeds the token in the SOAP message. The server extracts the token and uses a Java Authentication and Authorization Service (JAAS) login module to validate the token. For an overview of the LTPA authentication mechanism, see “Lightweight Third-party Authentication (LTPA) method for Web services.”

Note: LTPA authentication is supported for server Web services only, including Web service applications that act as clients to other Web services. A pure Web service client (that is, a client that is not also a Web service) cannot authenticate with LTPA.

However, you can configure multiple authentication mechanisms for a Web service. In a scenario with multiple Web services and Web services clients, you can configure the clients to authenticate with a different authentication mechanism. You can then configure the Web services to authenticate with LTPA.

Note: To use the LTPA authentication mechanism for Web services, you must configure WebSphere global security and the LTPA authentication mechanism. For more information, see Configure global security and Configure the authentication mechanism in the *Security* topic.

Perform these steps to configure LTPA authentication for your Web service:

1. “Configure the Web services client for LTPA token authentication” on page 123
   This topic describes how to configure LTPA authentication for a Web service that acts as a client.
2. “Configure the Web services server for LTPA token authentication” on page 124
   This topic describes how to configure LTPA authentication for your Web service application.
3. (Optional) “Configure a pluggable token” on page 126
   If you have developed custom token generation and validation, see this topic for information about configuring your pluggable token. For more information, see “Pluggable token support” on page 129.

Lightweight Third-party Authentication (LTPA) method for Web services: When you use the lightweight third party authentication (LTPA) method, the security token that is generated is <wsse:BinarySecurityToken>. On the request sender side, the security token is generated by invoking a callback handler. On the request receiver side, the security token is validated by a Java Authentication and Authorization Service (JAAS) login module. The token generation and token validation operations are described in the following topics.

LTPA token generation

The request sender uses a callback handler to generate an LTPA security token. The callback handler returns a security token that is inserted in the SOAP message. Specify the appropriate callback handler in the <LoginBinding> element of the bindings file (ibm-webservicesclient-bnd.xmi). The com.ibm.wsspi.wssecurity.auth.callback.LTPATokenCallbackHandler can be used with the LTPA authentication method. You can add your own callback handlers that implement the javax.security.auth.callback.CallbackHandler interface. For more information, see “Generating a pluggable token” on page 130.

When you use the LTPA authentication method (or any authentication method other than BasicAuth, Signature or IDAssertion), the TokenValueType attribute of the <LoginBinding> element in the bindings file (ibm-webservicesclient-bnd.xmi) must be specified.

The following values are used for the LTPA TokenValueType:

- `uri="http://www.ibm.com/websphere/appserver/tokentype/5.0.2"`
- `localName="LTPA"`

LTPA token validation

The request receiver retrieves the LTPA security token from the SOAP message and validates it using a JAAS login module. The security token, <wsse:BinarySecurityToken>, is used to perform the validation. If the validation is successful, the login module returns a JAAS Subject. Subsequently, this Subject is set as the identity of the thread of execution. If the validation fails, the request is rejected with a SOAP fault.

The appropriate JAAS login configuration to use is specified in the bindings file <LoginMapping> element. There are default bindings specified in the ws-security.xml file, but these can be overridden using the application-specific ibm-webservices-bnd.xmi file. The configuration information consists of the following properties:

- **CallbackHandlerFactory**
  The CallbackHandlerFactory specifies the name of a class to use to create the JAAS CallbackHandler object. A CallbackHandlerFactory implementation is provided: com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl.
- **ConfigName**
  The ConfigName specifies a JAAS configuration name entry. The Web services security run time first searches the security.xml file for a matching entry and if a matching entry is not found, the run time searches the wsjaas.conf file. A default configuration entry suitable for the LTPA authentication method is provided (WSLogin).
- **TokenValueType**
  There is an appropriate TokenValueType element in the LTPA LoginMapping section of the default ws-security.xml file.

For more information, see "Validating a pluggable token" on page 133.

Configure the Web services client for LTPA token authentication: When a client authenticates to WebSphere Application Server - Express, the credential that is created contains an LTPA token. You can configure a Web service to send the LTPA token when it calls a downstream Web service.

Note: You can only configure client LTPA authentication for a Web service that calls another Web service. Do not attempt to configure LTPA from a pure client. For the downstream Web service to validate the LTPA token, the LTPA keys must be the same for both servers.

Do not configure the client for LTPA token authentication unless LTPA is the configured authentication mechanism for WebSphere Application Server - Express. For more information, see Configure the authentication mechanism in the *Security* topic.

Perform the following steps to specify LTPA token authentication for your Web services client:

1. Open the webservicesclient.xml file in the Web Services Client Editor of the WebSphere Development Studio Client for iSeries. For more information, see "Configure your Web services application" on page 102.
2. Click the **Security Extensions** tab.
3. Expand the **Request Sender Configuration —> Login Config** settings.
4. Select **LTPA** as the authentication method.
5. Save the file.

Next, perform the following steps in the Web Services Client Editor to configure how the LTPA information is collected:

1. Click the **Port Binding** tab.
2. Expand the **Security Request Sender Binding Configuration —> Login Binding** settings.
3. Click **Edit** to view the login binding information and select **LTPA**. If **LTPA** is not listed, enter it as an option. The login binding dialog displays.

4. Select or enter the following information:

Name	Purpose
Authentication method	The authentication method specifies the type of authentication that occurs. Select **LTPA** to use identity assertion.
Token value type URI and **Token value type local name**	When you select **LTPA**, you must edit the **token value type URI** and the **local name** fields. These values are specified for custom authentication types, which are authentication methods that are not mentioned in the Web services security specification. • For **token value type URI**, enter `http://www.ibm.com/websphere/appserver /tokentype/5.0.2`. • For **local name**, enter `LTPA`.
Callback handler	The callback handler specifies the Java Authentication and Authorization Service (JAAS) callback handler implementation for collecting the LTPA information. Specify the `com.ibm.wsspi.wssecurity.auth.callback. LTPATokenCallbackHandler` implementation for LTPA.
Basic authentication user ID and **Basic authentication password**	For LTPA, you can leave these fields empty.
Property name and **Property value**	For LTPA, you can leave these fields empty.

Note: Examples may be wrapped for display purposes.

Configure the Web services server for LTPA token authentication: This task is used to configure Lightweight Third-Party Authentication (LTPA). LTPA is a type of authentication mechanism in WebSphere Application Server - Express security that defines a particular token format. The purpose of the LTPA token authentication is to send the LTPA token from the first Web service, which authenticated the originating client, to the downstream Web service.

After the downstream Web service receives the LTPA token, it validates the token to verify that the token has not been modified and has not expired. For validation to be successful, the LTPA keys that are used by both the sending and receiving servers must be the same.

Note: You can only configure client LTPA authentication for a Web service that calls another Web service. Do not attempt to configure LTPA from a pure client.

Perform the following steps in the WebSphere Development Studio Client for iSeries to configure the server for Web services signature authentication:

1. Open the webservices.xml deployment descriptor for your Web services application in the Web Services Editor of the WebSphere Development Studio Client for iSeries. For more information, see "Configure your Web services application" on page 102.
2. Click the **Security Extensions** tab.
3. Expand the **Request Receiver Service Configuration Details —> Login Configuration** settings.
4. Select **LTPA** to authenticate the client using the LTPA token received from the request.
5. Save the file.

Next, perform the following steps in the Web Services Editor to specify how the LTPA authentication information is validated:

1. Click the **Binding Configurations** tab.
2. Expand the **Request Receiver Binding Configuration Details —> Login Mapping** settings.

3. Click **Edit** to view the Login Mapping information. The login mapping information is displayed.
4. Select or enter the following information:

Name	Purpose
Authentication method	The authentication method specifies the type of authentication that occurs. Select **LTPA** to use LTPA token authentication.
Configuration name	This name specifies the Java Authentication and Authorization Service (JAAS) login configuration name. For the LTPA authentication method, enter `WSLogin` for the JAAS login configuration name. This configuration understands how to validate an LTPA token.
Use Token value type	This option determines if you want to specify a custom token type. For LTPA authentication, you must select this option because LTPA is considered a custom type. LTPA is not part of the Web services security specification.
Token value type URI and **local name**	If you select **Use Token value type** you must enter data into the **Token value Type URI** and **local name** fields. For **URI**, enter `http://www.ibm.com/websphere/appserver/tokentype/5.0.2`. For **local name**, enter `LTPA`.
Callback Handler Factory Class Name	This classname creates a JAAS CallbackHandler implementation that understands the following callback handlers: • javax.security.auth.callback.NameCallback • javax.security.auth.callback.PasswordCallback • com.ibm.wsspi.wssecurity.auth.callback.BinaryTokenCallback • com.ibm.wsspi.wssecurity.auth.callback.XMLTokenReceiverCallback • com.ibm.wsspi.wssecurity.auth.callback.PropertyCallback For any of the default Authentication methods (BasicAuth, IDAssertion, Signature, LTPA), use the callback handler factory default implementation. Enter `com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl` for any of the default authentication methods, including LTPA. This implementation creates the correct callback handler for the default implementations.
Callback Handler Factory Property	This field is used to specify callback handler properties for custom callback handler factory implementations. The default callback handler factory implementation does not need you to specify any properties. For LTPA, you do not need to enter any properties for this field.
Login Mapping Property	This field is used to specify properties for a custom login mapping. For the default implementations including LTPA, you do not need to enter any properties for this field.

Note: Examples may be wrapped for display purposes.

Configure a pluggable token: This topic describes how to configure the request sender to create security tokens in the Simple Object Access Protocol (SOAP) message and how to configure the request receiver to validate the security tokens found in the incoming SOAP message. You can use the authentication method defined in the login bindings and login mappings to generate security tokens in the request sender and validate security tokens in the request receiver.

WebSphere Application Server - Express supports pluggable security tokens. See the following topics for more information:

- "Pluggable token support" on page 129
- "Generating a pluggable token" on page 130
- "Validating a pluggable token" on page 133

Note: The pluggable token is required for the request sender and request receiver as they are a pair. The request sender and the request receiver must match for a request to be accepted by the receiver.

Prior to completing these steps, it is assumed that you have already created a Web services-enabled Java 2 Platform, Enterprise Edition (J2EE) with a Web Services for J2EE (JSR 109) enterprise application. If not, see "Develop Web services" on page 6 to create Web services-enabled J2EE with a JSR 109 enterprise application.

Perform the folowing steps in the WebSphere Development Studio Client for iSeries to configure a pluggable token for your Web service client:

1. Open the webservicesclient.xml file in the Web Services Client Editor of the WebSphere Development Studio Client for iSeries. For more information, see "Configure your Web services application" on page 102.
2. Click the **Security Extensions** tab. The Web Service Client Security Extensions editor displays. Specify the following settings:
   a. Under **Service References**, select an existing service reference or click **Add** to create a new one.
   b. Under **Port Qname Bindings**, select an existing port-qualified name for the selected service reference or click **Add** to create a new port name binding.
   c. Under **Request Sender Configuration: Login Config**, select an exiting authentication method or type in a new one in the editable list box. When a Web servics acts as a client, `LTPA` is a supported token generation format.
3. Click the **Web Services Client Binding** tab. The Web Services Client Binding editor displays. Specify the following settings:
   a. Under **Port Qualified Name Binding**, select an existing entry or click **Add** to add a new port name binding. The Web Services Client Binding editor displays for the selected port.
   b. Under **Login Binding**, click **Edit** or **Enable**. The Login Binding dialog displays. Specify the following settings:
      1) In the **Authentication Method** field, enter the authentication method. The authentication method that you enter in this field must match the authentication method defined on the **Security Extension** tab for the same Web service port. This field is mandatory.
      2) (Optional) Enter the token value type information in the **URI** and **Local name** fields. These fields are ignored for the BasicAuth, Signature, and IDAssertion authentication methods, but required for other authentication methods. The token value type information is inserted into the <wsse:BinarySecurityToken>@ValueType element for binary security token and is used as the namespace for the XML-based token.
      3) Enter an implementation of the Java Authentication and Authorization Service (JAAS) javax.security.auth.callback.CallbackHandler interface. See "Generating a pluggable token" on page 130 for information on how to develop a CallbackHandler that can generate a security token in the request sender. This is a mandatory field.

4) Enter the basic authentication information in the **User ID** and **Password** fields. The basic authentication information is passed to the constructor of the CallbackHandler implementation. The usage of the basic authentication information is up to the implementation of the CallbackHandler.

5) In the **Property** field, add name and value pairs. These pairs are passed to the constructor of the CallbackHandler implementation as java.util.Map data types.

6) Click **OK**.

4. Save the file.

Perform the folowing steps in the WebSphere Development Studio Client for iSeries to configure a pluggable token for your Web services application:

1. Open the webservices.xml deployment descriptor for your Web services application in the Web Services Editor of the WebSphere Development Studio Client for iSeries. For more information, see "Configure your Web services application" on page 102.
2. Click the **Security Extensions** tab. Specify the following settings:
   a. Under **Web Service Description Extension**, select an existing service reference or click **Add** to create a new extension.
   b. Under **Port Component Binding**, select an existing port-qualified name of the selected service reference or click **Add** to create a new one.
   c. Under **Request Receiver Service Configuration Details: Login Config**, select an exiting authentication method or click **Add** and enter a new method in the **Add AuthMethod** field that displays. You can select multiple authentication methods for the request receiver. The security token of the incoming message is authenticated against the authentication methods in the order that they are specified in the list.
3. Click the **Bindings** tab. The Web Services Bindings editor displays. Under **Web Service Description Bindings**, select an existing entry or click **Add** to add a new Web services descriptor.
4. Click the **Binding Configurations** tab. The Web Services Binding Configurations editor displays for the selected Web services descriptor. Under **Request Receiver Binding Configuration Details: Login Mapping**, click **Add** to create a new login mapping or click **Edit** to edit existing selected login mapping.

   The Login mapping dialog displays. Specify the following settings:
   a. In the **Authentication method** field, enter the authentication method. The information entered in this field must match the authentication method defined on the **Security Extensions** tab for the same Web service port. This is a mandatory field.
   b. In the **Configuration name** field, enter a JAAS login configuration name. You must define the JAAS login configuration name in the WebSphere administrative console under **Security —> JAAS Configuration —> Application Logins**). This is a mandatory field. For more information, see Configure JAAS login in the *Security* topic.
   c. (Optional) Select **Use Token value type** and enter the token value type information in the **URI** and **Local name** fields. This information is optional for BasicAuth, Signature and IDAssertion authentication methods, but required for any other authentication method. The token value type is used to validate the <wsse:BinarySecurityToken>@ValueType element for binary security tokens and to validate the namespace of the XML-based token.
   d. Under **Callback Handler Factory**, enter an implementation of the com.ibm.wsspi.wssecurity.auth.callback.CallbackHandlerFactory interface in the **Class name** field. This field is mandatory. See "Validating a pluggable token" on page 133 for instructions on how to develop a CallbackHandlerFactory and JAAS Login Configuration to validate the security token of the incoming message.
   e. Under **Callback Handler Factory Property**, click **Add** and enter the name and value pairs for the Callback Handler Factory Property. These name and value pairs are passed as a java.util.Map data

type to the com.ibm.wsspi.wssecurity.auth.callback.CallbackHandlerFactory.init() method. The usage of these name and value pairs is determined by the CallbackHandlerFactory implementation chosen.

   f. Under Login Mapping Property, click **Add** and enter the name and value pairs for the Login Mapping Property. These name and value pairs are available to the JAAS Login Module or Modules through thecom.ibm.wsspi.wssecurity.auth.callback.PropertyCallback JAAS Callback interface. Click **Remove** to delete selected login mapping.

   g. Click **OK**.

5. Save the file.

Configure pluggable tokens with WebSphere administrative console

Prior to completing these steps, it is assumed that you deployed a Web services-enabled enterprise application to the WebSphere Application Server - Express.

Perform the following steps in the administrative console:

1. Click **Applications —> Enterprise Applications —>** ***enterprise_application***, where *enterprise_application* is the name of your enterprise application.
2. Under **Related Items**, click **Web Modules —>** ***Uri***, where *Uri* is the URI of your Web services-enabled module.
3. (Optional) If the Web service is acting as a client, configure the client bindings. Under Additional Properties, click **Web Services: Client Security Bindings** to edit the response sender binding information, if Web services is acting as client. Specify the following settings:
   a. Under Response Sender Binding, click **Edit.**
   b. Under Additional Properties, click **Login Binding**.
   c. Select **Dedicated Login Binding** to define a new login binding. Specify the following settings:
      1) Enter the authentication method, this must match the authentication method defined in the IBM extension deployment descriptor. The authentication method must be unique in the binding file.
      2) Enter the name of your JAAS javax.security.auth.callback.CallbackHandler implementation. For more information, see “Generating a pluggable token” on page 130.
      3) Enter the basic authentication information (User ID and Password). The basic authentication information is passed to the construct of the CallbackHandler implementation. The usage of the basic authentication information defined by the implementation of the CallbackHandler.
      4) Enter the token value type, it is optional for BasicAuth, Signature and IDAssertion authentication methods but required for any other authentication method. The token value type is inserted into the <wsse:BinarySecurityToken>@ValueType for binary security token and used as the namespace of the XML-based token.
      5) Click **Properties**. Define the property with name and value pairs. These pairs are passed to the construct of the CallbackHandler implementation as java.util.Map data types.
4. Under **Additional Properties**, click **Web Services: Server Security Bindings** to edit the request receiver binding information. Specify the following settings:
   a. Under **Request Receiver Binding**, click **Edit**.
   b. Under **Additional Properties**, click **Login Mappings**. Click **New** to create new login mapping. Specify the following settings:
      1) Enter the authentication method, this must match the authentication method defined in the IBM extension deployment descriptor. The authentication method must be unique in the login mapping collection of the binding file.
      2) Enter a JAAS Login Configuration name. The JAAS Login Configuration must be defined in the **Security —> JAAS Configuration —> Application Logins** settings. For more information, see Configure JAAS login in the *Security* topic.

3) Enter the name of your com.ibm.wsspi.wssecurity.auth.callback.CallbackHandlerFactory implementation. See "Validating a pluggable token" on page 133 for more information. This is a mandatory field.
4) Enter the token value type. This setting is optional for BasicAuth, Signature and IDAssertion authentication methods but required for any other authentication method. The token value type is used to validate against the <wsse:BinarySecurityToken>@ValueType for binary security token and against the namespace of the XML-based token.
5) Enter the name and value pairs for the **Login Mapping Property** by clicking **Properties**. These name and value pairs are available to the JAAS login module or modules by the com.ibm.wsspi.wssecurity.auth.callback.PropertyCallback JAAS callback.
6) Enter the name and value pairs for the **Callback Handler Factory Property** These name and value pairs are passed as java.util.Map data types to the om.ibm.wsspi.wssecurity.auth.callback.CallbackHandlerFactory.init() method. The usage of these name and value pairs is dependent on the CallbackHandlerFactory implementation.

5. Save the configuration.

You can also define login mappings for the server-level and cell-level default binding configuration (ws-security.xml). To define the login mappings for the server-level default binding configuration, perform these steps in the administrative console:

1. Click **Servers —> Application Servers —>** ***server_name***, where *server_name* is the name of your application server.
2. Under Related Items, click **Web Services: Default bindings for Web Services Security** and then follow the steps outlined previously for creating or editing login mappings for **Web Services: Server Security Bindings**.
3. To define the login mappings for the cell-level default binding configuration, click **Security —> Web Services** and then follow the steps outlined previously for creating or editing login mappings for **Web Services: Server Security Bindings**.
4. Save the configuration.

Pluggable token support: You can extend the WebSphere Application Server - Express login mapping mechanism to handle new types of authentication tokens. WebSphere Application Server - Express provides a pluggable framework to generate security tokens on the sender-side of the message and to validate the security token on the receiver-side of the message. The framework is based on the Java Authentication and Authorization Service (JAAS) Application Programming Interfaces (APIs).

Pluggable security token support provides plug-in points to support customer security token types including token generation, token validation, and mapping a client identity to a WebSphere Application Server - Express identity that is used by the Java 2 Enterprise Edition (J2EE) authorization engine. Moreover, the pluggable token generation and validation framework allows XML-based tokens to be inserted into the Web service message header and validated on the receiver side. For more information, see "Generating a pluggable token" on page 130 and "Validating a pluggable token" on page 133.

Users can use the javax.security.auth.callback.CallbackHandler implementation to create a new type of security token following these guidelines:

- Use a constructor that takes a user name (of type String or `null`, if not defined), password (of type `char[]` or `null`, if not defined) and java.util.Map (empty, if properties are not defined).
- Use handle() methods that can process the following implementations:
  - javax.security.auth.callback.NameCallback
  - javax.security.auth.callback.PasswordCallback
  - com.ibm.websphere.security.auth.callback.WSCredTokenCallbackImpl
  - com.ibm.wsspi.wssecurity.auth.callback.XMLTokenCallback

If the NameCallback or the PasswordCallback implementation is populated with data, then a <wsse:UsernameToken> element is created. Otherwise, if WSCredTokenCallbackImpl is populated, the <wsse:BinarySecurityToken> element is created from the WSCredTokenCallbackImpl.

If XMLTokenCallback is populated, a XML-based token is created. This token is based on the Document Object Model (DOM) element that is returned from the XMLTokenCallback implementation. Encode the token byte by using the security handler and not the javax.security.auth.callback.CallbackHandler implementation.

You can implement the com.ibm.wsspi.wssecurity.auth.callback.CallbackHandlerFactory interface, which is a factory for instantiating the javax.security.auth.callback.CallbackHandler. For your own implementation, you must provide the javax.security.auth.callback.CallbackHandler interface. The Web service security run time instantiates the factory implementation class and passes the authentication information from the Web services message header to the factory class through the setter methods. The Web services security run time then invokes the newCallbackHandler() method of the factory implementation class to obtain an instance of the javax.security.auth.CallbackHandler object. The object is passed to the JAAS login configuration.

The following example is the definition of the CallbackHandlerFactory interface:

```
public interface com.ibm.wsspi.wssecurity.auth.callback.CallbackHandlerFactory {
  public void setUsername(String username);
  public void setRealm(String realm);
  public void setPassword(String password);
  public void setHashMap(Map properties);
  public void setTokenByte(byte[] token);
  public void setXMLToken(Element xmlToken);
  public CallbackHandler newCallbackHandler();
}
```

Generating a pluggable token: The Web services security run time uses the JAAS CallbackHandler interface as a plugin to generate security tokens on the client side or when a Web service is acting as client. This topic describes how to write a Java Authentication and Authorization Server (JAAS) javax.security.auth.callback.CallbackHandler to generate a binary security token (<wsse:BinarySecurityToken>) and an XML-based token.

See “Configure a pluggable token” on page 126 for information about configuring the pluggable token authentication for a request receiver.

Standard Java Authentication and Authorization Service CallbackHandler

WebSphere Application Server - Express provides a default implementation of the following JAAS callback handlers that you can use:

- **com.ibm.wsspi.wssecurity.auth.callback.GUIPromptCallbackHandler**
  If basic authentication data is not defined in the login binding (not to be confused with the HTTP basic authentication information), WebSphere Application Server - Express prompts for a user name and password in the graphical user interface (GUI) login panel. However, WebSphere Application Server - Express uses the basic authentication data that is defined in the login binding.

  Note: Use this callback handler with the BasicAuth authentication method only. Also, this implementation should only be used with Web services clients. The prompt behavior is not desirable in a server environment.
- **com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler**
  If basic authentication data is not defined in the login binding, WebSphere Application Server - Express prompts for a user name and password in Standard in (stdin). However, WebSphere Application Server - Express uses the basic authentication data that is defined in the login binding.

  Note: Use this callback handler with the BasicAuth authentication method only. Also, this implementation should only be used with Web services clients. The prompt behavior is not desirable in a server environment.

- **com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler**
  This callback handler does not prompt the user. It uses the basic authentication data that is defined in the login binding.

  Note: Use this callback handler with BasicAuth authentication method only. You can use this callback handler when a Web service is acting as a client and needs to send basic authentication information (<wsse:UsernameToken>) to a downstream call. You must define basic authentication data in the login binding for this callback handler.
- **com.ibm.wsspi.wssecurity.auth.callback.LTPATokenCallbackHandler**
  This callback handler generates LTPA tokens from the RunAs JAAS Subject (invocation subject) of the current WebSphere Application Server - Express security context. However, if basic authentication data is defined in the login binding, it authenticates with the basic authentication data and uses the LTPA token that is generated. The Web services security run time inserts the LTPA token as binary security token (<wsse:BinarySecurityToken>) into the Simple Object Access Protocol (SOAP) header of the message. The value type is mandatory and the value must be `http://www.ibm.com/websphere/appserver/tokentype/5.0.2/LTPA`.

  Note: Use this callback handler with the LTPA authentication method. Also, the **Token Type URI** and **Token Type Local Name** fields must be defined in the login binding for this callback handler. The token type values for both the sender and receiver must be the same. These values are defined in the binding configurations.

Developing a Java Authentication and Authorization Service callback handler

Because tokens are pluggable, you can also provide your own callback handler implementation.

Perform the following steps to develop your own JAAS callback handler:

1. Implement the javax.security.auth.callback.CallbackHandler interface. The implementation must provide a default constructor with the following method signature:

   ```
   MyCallbackHandler(String userid, char[] password, java.util.Map properties)
   ```

   where *userid* and *password* is the basic authentication data, and *properties* are the authentication properties that are defined in the login binding.
2. For the BasicAuth authentication method, the handler() method must handle the following javax.security.auth.callback.Callback implementation classes:
   - **javax.security.auth.callback.NameCallback**
     This is the standard JAAS callback and part of the JAAS default package. The implementation must set the user name using the javax.security.auth.callback.NameCallback.setName() method.
   - **javax.security.auth.callback.PasswordCallback**
     This is the standard JAAS Callback and part of the JAAS default package. The implementation must set the user name using the javax.security.auth.callback.PasswordCallback.setPassword() method.
3. For pluggable security token (other authentication methods), the handler() method must handle the following javax.security.auth.callback.Callback implementation classes:
   - **com.ibm.wsspi.wssecurity.auth.callback.BinaryTokenCallback**
     This is the implementation that is provided by WebSphere Application Server - Express. It is used to pass a binary security token to the Web services security run time. The implementation must set the binary security token as a byte[] data type using the com.ibm.wsspi.wssecurity.auth.callback.BinaryTokenCallback.setCredToken() method.
   - **com.ibm.wsspi.wssecurity.auth.callback.XMLTokenSenderCallback**
     This is the implementation that is provided by WebSphere Application Server - Express. It is used to pass XML-based tokens to the Web services security run time. The implementation must set the XML-based token as a org.w3c.dom.Element[] data type using the com.ibm.wsspi.wssecurity.auth.callback.XMLTokenSenderCallback.setXMLTokens() method.

   Note: If both the binary security token and XML-based token callback handlers are set, the binary security token takes precedence over the XML-based token. A binary security token is generated.

Sample implementation for BasicAuth authentication method

The following code is a sample callback handler implementation for generating the <wsse:UsernameToken> element. The error handling has been removed for clarity.

```
public class MyBACallbackHandler implements CallbackHandler {
  public MyBACallbackHandler() {
    super();
  }

  public MyBACallbackHandler(String userid, char[] password, Map properties) {
    super();
    tmpusername = userid;
    tmppassword = password;
    tmpMap = properties;
  }


  /**
   * This implementation of MyBACallbackHandler map the username and
   * password data defined in the Login binding to another user.
   */
  public void handle(Callback[] callbacks)
    throws IOException, UnsupportedCallbackException {

    if ((callbacks == null) || (callbacks.length == 0)) {
      return;
    }

    // call out to some server to perform mapping of
    // tmpusername and tmppassword to a mappeduser
    // and mappedpassword
    Result result = mapUser(tmpusername, tmppassword, tmpMap);
    String mappeduser = result.getMappedUser();
    char[] mappedpassword = result.getMappedPassword();

    for (int i = 0; i < callbacks.length; i++) {
      callback c = callbacks[i];

      if (c instanceof javax.security.auth.callback.namecallback) {
        ((javax.security.auth.callback.namecallback) c).setname(mappeduser);
      } else if (c instanceof javax.security.auth.callback.passwordcallback) {
        ((javax.security.auth.callback.passwordcallback) c).setpassword(
          (mappedpassword == null) ? new char[0] : mappedpassword);
      } else {
        throw new unsupportedcallbackexception(c, "Unsupported callback");
      }
    }
  }

  private string tmpusername = "";
  private char[] tmppassword = null;
  private map tmpmap = null;
}
```

The following sample code is a sample callback handler implementation for generating <wsse:BinarySecurityToken> element.

```
public class MyBSTCallbackHandler implements CallbackHandler {
  public MyBSTCallbackHandler() {
    super();
  }

  public MyBSTCallbackHandler(String userid, char[] password, Map properties) {
    super();
    tmpusername = userid;
    tmppassword = password;
    tmpMap = properties;
  }


  /**
   * This implementation of MyBSTCallbackHandler generates binary
   * security token based on the username and password data defined in the
   * Login binding to another user.
   */
  public void handle(Callback[] callbacks)
    throws IOException, UnsupportedCallbackException {

    if ((callbacks == null) || (callbacks.length == 0)) {
      return;
    }

    // call out to create binary security token
    // based on tmpusername and tmppassword
    byte[] token = login(tmpusername, tmppassword);

    for (int i = 0; i < callbacks.length; i++) {
      callback c = callbacks[i];

      if (c instanceof com.ibm.wsspi.wssecurity.auth.callback.binarytokencallback) {
        ((com.ibm.wsspi.wssecurity.auth.callback.binarytokencallback) c).setcredtoken(token);
      } else if (c instanceof com.ibm.wsspi.wssecurity.auth.callback.xmltokensendercallback) {
        continue;
      } else {
        throw new unsupportedcallbackexception(c, "Unsupported callback");
      }
    }
  }

  private string tmpusername = "";
  private char[] tmppassword = null;
  private map tmpmap = null;
}
```

Validating a pluggable token: This topic describes how to develop a Java Authentication and Authorization Service (JAAS) login module to authenticate the security token of an incoming request. The pluggable token is based on the JAAS programming model. You can develop and configure custom JAAS Login modules to authenticate custom security tokens.

See "Configure a pluggable token" on page 126 for information about configuring the pluggable token authentication for a request receiver.

Standard login mapping configuration

WebSphere Application Server - Express provides default implementations and configurations of the following login mappings:

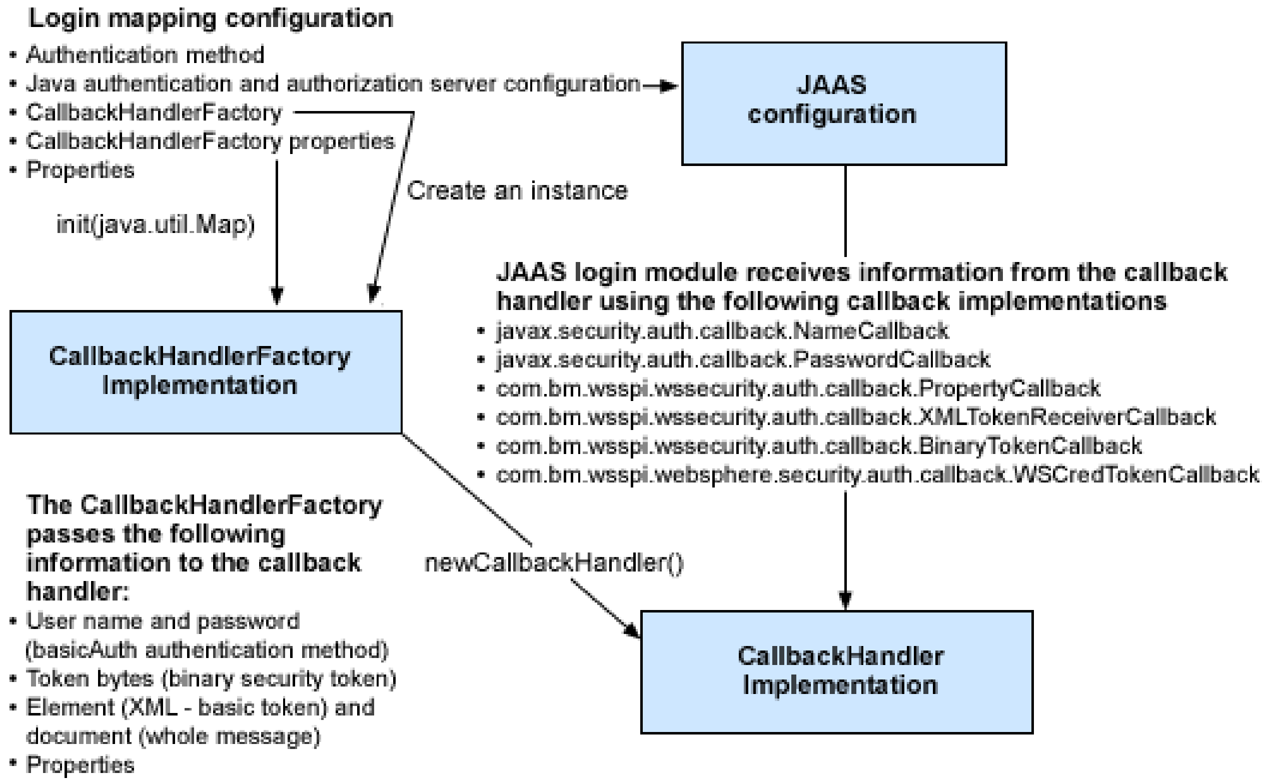
- **BasicAuth**
  BasicAuth is used to authenticate both a user name and a password.
- **Signature**
  Signature is used to map the distinguished name (DN) of the certificate to a JAAS Subject.
- **IDAssertion**
  IDAssertion is used to map a trusted identity to a JAAS Subject for identity assertion.
- **LTPA**
  Lightweight Third-party Authentication (LTPA) is used to authenticate LTPA security tokens. The value type of the LTPA is http://www.ibm.com/websphere/appserver/tokentype/5.0.2/LTPA

However, because the token is pluggable, you can can provide your own implementation. The token value is optional for the BasicAuth, Signature, and IDAssertion authentication methods. However, the token value is required by other types of authentication methods, including LTPA and the pluggable token. It is used to validate against the value type of binary security token (<wsse:BinarySecurityToken@ValueType>) and against the namespace for XML-based token. Therefore, the request sender must have the correct value type configured and inserted into the security tokens.

Developing a JAAS Login Module

The following figure shows the relationship between the login mapping configuration and the pluggable token validation.

Figure 1: Token validation


Login mapping configuration
• Authentication method
• Java authentication and authorization server configuration
• CallbackHandlerFactory
• CallbackHandlerFactory properties
• Properties
JAAS configuration
Create an instance
init(java.util.Map)
JAAS login module receives information from the callback handler using the following callback implementations
• javax.security.auth.callback.NameCallback
• javax.security.auth.callback.PasswordCallback
• com.bm.wsspi.wssecurity.auth.callback.PropertyCallback
• com.bm.wsspi.wssecurity.auth.callback.XMLTokenReceiverCallback
• com.bm.wsspi.wssecurity.auth.callback.BinaryTokenCallback
• com.bm.wsspi.websphere.security.auth.callback.WSCredTokenCallback
CallbackHandlerFactory Implementation
The CallbackHandlerFactory passes the following information to the callback handler:
• User name and password (basicAuth authentication method)
• Token bytes (binary security token)
• Element (XML - basic token) and document (whole message)
• Properties
newCallbackHandler()
CallbackHandler Implementation

The com.ibm.wsspi.wssecurity.auth.callback.CallbackHandlerFactory interface is a factory that is used to create an instance of the javax.security.auth.callback.CallbackHandler interface. For more information, see the Javadoc

(http://java.sun.com/j2ee/sdk_1.3/techdocs/api/javax/security/auth/callback/CallbackHandler.html). The various set() methods are used by the Web services security run time to pass various security tokens (<wsse:UsernameToken>, <BinarySecurityToken>, and XML-based security token and properties from the login binding) to the implementation, which can pass the security tokens to the new callback handler instance. The properties defined for the CallbackHandlerFactory in the login mapping are passed to the implementation through the com.ibm.wsspi.wssecurity.auth.callback.CallbackHandlerFactory.init() method.

WebSphere Application Server - Express provides a default implementation of the CallbackHandlerFactory interface, which is called com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl. The default implementation creates a callback handler that can handle the following javax.security.callback.Callback implementation classes:

- **javax.security.auth.callback.NameCallback** and **javax.security.auth.callback.PasswordCallback**
  The JAAS login module uses these callbacks to obtain basic authentication information. If the Simple Object Access Protocol (SOAP) header contains <wsse:UsernameToken>, the following actions occur:
  1. The Web services security run time passes the user name and password to the CallbackHandlerFactory implementation.
  2. The CallbackHandlerFactory passes the information to the CallbackHandler implementation. The CallbackHandler implementation can set the user name and password using the javax.security.auth.callback.NameCallback.getName() and javax.security.auth.callback.PasswordCallback.getPassword() methods respectively.
- **com.ibm.websphere.security.auth.callback.WSCredTokenCallbackImpl** or **com.ibm.wsspi.wssecurity.auth.callback.BinaryTokenCallback**
  The JAAS login module can use either of these implementations to obtain the token byte. If the SOAP header contains <wsse:BinarySecurityToken>, the following actions occur:
  1. The Web services security run time passes the token byte to the CallbackHandlerFactory implementation.
  2. The CallbackHandlerFactory implementation passed the token byte to the CallbackHandler iplementation. The CallbackHandler implementation returns from com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl set the token byte to both of the above Callbacks. (WSCallbackHandlerFactoryImpl passes tokens to CallbackHandler and CallbackHandler passes the tokens to the JAAS Login Module using Callbacks.) The JAAS Login Module can obtain the token byte using either com.ibm.websphere.security.auth.callback.WSCredTokenCallbackImpl.getCredToken() or com.ibm.wsspi.wssecurity.auth.callback.BinaryTokenCallback.getCredToken().
- **com.ibm.wsspi.wssecurity.auth.callback.XMLTokenReceiverCallback**
  The JAAS login module uses this callback to obtain the XML-based token. If the SOAP header contains a XML-based token, the Web services security run time passes the token as org.w3c.dom.Element and the whole SOAP message as org.w3c.dom.Document to CallbackHandlerFactory implementation. The JAAS login module can obtain the XML-based token and the whole SOAP message using the com.ibm.wsspi.wssecurity.auth.callback.XMLTokenReceiverCallback.getXMLToken() and com.ibm.wsspi.wssecurity.auth.callback.XMLTokenReceiverCallback.getSOAPMessage() methods respectively.
- **com.ibm.wsspi.wssecurity.auth.callback.PropertyCallback**
  If there are name and value pairs defined in the login mapping, the Web services security run time passes these pairs as java.util.Map to the CallbackHandlerFactory implementation, which, in turn, passes the pairs to the CallbackHandler implementation. The JAAS login module can obtain these properties by calling com.ibm.wsspi.wssecurity.auth.callback.PropertyCallback.getProperties()method.

com.ibm.wsspi.wssecurity.auth.module.WSSecurityMappingModule

WebSphere Application Server - Express also provides a default JAAS login module that you can use to map an identity to a principal and a credential. WebSphere Application Server - Express then can use the identity. The JAAS Login Module is called com.ibm.wsspi.wssecurity.auth.module.WSSecurityMappingModule. After the custom JAAS Login Module validates or authenticates the security token, the WSSecurityMappingModule can be used to map the identity to principal and credential format that can be used WebSphere Application Server - Express. You can configure the WSSecurityMappingModule module as the last JAAS login module in the JAAS login configuration using the stackable login module of JAAS. For more information on configuring a JAAS login, see Configure JAAS login in the *Security* topic.

The WSSecurityMappingModule.login() method looks for the identity using the com.ibm.wsspi.wssecurity.Constants.DN key from the shared state map (java.util.Map) of a JAAS login context. The shared state map is passed to the JAAS login module by the javax.security.auth.spi.LoginModule.initialize() method. After the credential is successfully created by the WSSecurityMappingModule.login() method, the WSSecurityMappingModule saves it in the shared state map by using the com.ibm.wsspi.wssecurity.Constants.WSCredential key. The other JAAS Login Modules can get the credential into their commit method. The credential is removed in the abort or commit method of the WSSecurityMappingModule.

Sample

The following sample code is a sample JAAS login module implementation that is used to validate the <wsse:BinarySecurityToken> element. The error handling was removed for clarity.

```
import javax.security.auth.*;
import javax.security.auth.callback.*;
import javax.security.auth.spi.*;
import com.ibm.websphere.security.auth.callback.*;
import java.util.*;
import javax.security.auth.login.*;
import com.ibm.websphere.security.cred.*;
import com.ibm.wsspi.wssecurity.auth.callback.*;

public class MyBSTLoginModule implements LoginModule {
  private Subject subject;
  private CallbackHandler callbackHandler;
  private Map sharedState;
  private Map options;

  private boolean succeeded = false;
  private boolean commitSucceeded = false;

  private byte[] token = null;
  private Map properties = null;
  private WSCredential credential = null;

  public MyBSTLoginModule() {
  }

  public void initialize(Subject subject, CallbackHandler callbackHandler,
                         Map sharedState, Map options) {
    this.subject = subject;
    this.callbackHandler = callbackHandler;
    this.sharedState = sharedState;
    this.options = options;
  }

  public boolean login() throws LoginException {
    if (callbackHandler == null)
      throw new LoginException("No CallbackHandler");

    succeeded = false;

    Callback[] callbacks = new Callback[2];
    callbacks[0] = new WSCredTokenCallbackImpl("Cred token: ");
    callbacks[1] = new PropertyCallback(null);

    try {
      callbackHandler.handle(callbacks);
      token = ((WSCredTokenCallbackImpl) callbacks[0]).getCredToken();
      // get the property in Login Mapping
      properties = ((PropertyCallback) callbacks[1]).getProperties();
    } catch (java.io.IOException ioe) {
      throw new LoginException(ioe.toString());
    } catch (UnsupportedCallbackException uce) {
      throw new LoginException(uce.getCallback().toString());
    }

    // validate the token and extract the id from the token
    succeeded = validate(token);
    String id = extractId(token);
    // put the identity in shared state
    sharedState.put(com.ibm.wsspi.wssecurity.Constants.WSSECURITY_DN, id);

    // ....

    return succeeded;
  }

  public boolean commit() throws LoginException {
    commitSucceeded = false;

    if (succeeded == true) {
      // set the custom token in the subject ....
      // to get the websphere credential
      credential = (WSCredential) sharedState.get(com.ibm.wsspi.wssecurity.Constants.WSSECURITY_CRED);
      // ....
      commitSucceeded = true;
    } else {
      // error;
    }

    return commitSucceeded;
  }

  private boolean validate(byte[] t) {
    // validate token
    // ....
    return true;
  }

  private String extractId(byte[] t) {
    // extract token id
    // ....
    return ...;
  }

  public boolean abort() throws LoginException {
    cleanup();
    return true;
  }

  public boolean logout() throws LoginException {
    cleanup();
    return true;
  }

  private void cleanup() {
    succeeded = false;
    commitSucceeded = false;
    // cleanup
  }
}
```

Configure HTTP basic authentication for Web services: HTTP basic authentication uses a username and password to authenticate a service client to a secure endpoint.

WebSphere Application Server - Express can have several resources, including Web services, protected by the J2EE security model.

A simple way to provide authentication data for the service client is to authenticate to the protected service endpoint to the HTTP basic authentication. The basic authentication is located in the HTTP header that carries the Simple Object Access Protocol (SOAP) request. When the application server receives the HTTP request, the username and password are retrieved and verified using the authentication mechanism specific to the server.

Note: To use HTTP basic authentication for Web services, you must configure WebSphere global security. For more information, see Configure global security in the *Security* topic.

Although the basic authentication data is base64-encoded, it is recommended that the data is sent over HTTPS. The integrity and confidentiality of the data can be protected by the Secured Sockets Layer (SSL) protocol.

In come cases, a firewall is present using the PASS-THRU HTTP proxy server. The HTTP proxy server forwards the basic authentication data into the J2EE application server. The proxy server can also be protected. Applications can specify the proxy data by setting properties in a stub object.

You can configure the username and password for HTTP basic authentication with the ibm-webservicesclient-bnd.xmi deployment descriptor, or using the properties mechanism for the configuration of a stub or call instance at run time.

- **Edit the ibm-webservicesclient-bnd.xmi deployment descriptor**
  Specify the attribute basicAuth for each `portQNameBindings` of each `serviceRef`. For example:
  ```
  >basicAuth userid="myID" password="myPassword"\>
  ```
- **Set properties**
  The values set by the properties mechanism take precedence over the values defined by the ibm-webservicesclient-bnd.xmi deployment descriptor.

  Change the following properties:
  - `javax.xml.rpc.Call.USERNAME_PROPERTY`
  - `javax.xml.rpc.Call.PASSWORD_PROPERTY`
  - `javax.xml.rpc.Stub.USERNAME_PROPERTY`
  - `javax.xml.rpc.Stub.PASSWORD_PROPERTY`

  You can also configure Proxy data using the properties mechanism described by using the following properties to configure your Web services application:
  - For HTTP:
    - `com.ibm.wsspi.webservices.HTTP_PROXYHOST_PROPERTY`
    - `com.ibm.wsspi.webservices.HTTP_PROXYPORT_PROPERTY`
    - `com.ibm.wsspi.webservices.HTTP_PROXYUSER_PROPERTY`
    - `com.ibm.wsspi.webservices.HTTP_PROXYPASSWORD_PROPERTY`
  - For HTTPS:
    - `com.ibm.wsspi.webservices.HTTPS_PROXYHOST_PROPERTY`
    - `com.ibm.wsspi.webservices.HTTPS_PROXYPORT_PROPERTY`
    - `com.ibm.wsspi.webservices.HTTPS_PROXYUSER_PROPERTY`
    - `com.ibm.wsspi.webservices.HTTPS_PROXYPASSWORD_PROPERTY`

After you have deployed your Web service, “Edit the HTTP basic authentication and SSL configuration for Web services” with the WebSphere administrative console.

Edit the HTTP basic authentication and SSL configuration for Web services: After you deploy your Web services application, edit the HTTP basic authentication (user ID and password) and Secure Sockets Layer (SSL) configuration for the HTTP outbound request in the client security bindings of the Web services.

You can edit the HTTP basic authentication and SSL configuration for the Web services from the administrative console by performing the following steps:

1. Expand **Applications**, and click **Enterprise Applications**.
2. Click the name of your application.
3. Under **Related Items**, click **Web Module**.
4. Click the name of your URI.
5. Click **Web Services: Client Security Bindings**.
6. Locate the **HTTP Basic Authentication** and **HTTP SSL Configuration** fields.
7. Configure HTTP Basic authentication:
   a. Click **Edit**.
   b. Enter the user ID and password.
   c. Click **OK**.
8. Configure HTTP SSL:
   a. Click **Edit**.
   b. Select **HTTP SSL Enabled**.
   c. Select an SSL alias for the HTTP SSL configuration.
   d. Click **OK**.
9. Save your configuration.
10. Restart your Web services application.

Configure Web services for digital signing

For purposes of integrity, you can configure your Web services to digitally sign and verify those digital signatures for the body, timestamp, or security token in a SOAP message.

To configure digital signing for your Web service, perform the following steps:

1. "Configure a key locator"
   Key locators are used to find keys for digital signature and encryption. WebSphere Application Server - Express provides default key locators that you can use with your digital signature configuration, or you can develop your own.
2. "Configure a collection certificate store" on page 146
   A collection certificate store contains CA certificates that are used to verify digital signatures. See this topic for information about configuring a collection certificate store for your Web services.
3. "Configure trust anchors" on page 149
   A trust anchor specifies key stores that contain root-trusted certificates that are used to validate the signer certificate of the digital signature. See this topic for information about configuring a trust anchor for your Web services.
4. "Configure the Web services client for request signing" on page 151
   Configure your Web services client to digitally sign its requests to the server.
5. "Configure the Web services client for response digital signature verification" on page 153
   Configure your Web services client to verify digital signatures in responses from the server.
6. "Configure the Web services server for request digital signature verification" on page 155
   Configure your Web service to verify digital signatures in requests it receives from the client.
7. "Configure the Web services server for response signing" on page 157
   Configure your Web service to digitally sign its responses to the client.

Configure a key locator: The purpose of key locators is to find keys or certificates. The method used to find keys or certificates depends upon the key locator implementation. WebSphere Application Server - Express provides the following default implementations:

- KeyStoreKeyLocator
- WSIdKeyStoreMapKeyLocator

- CertInRequestKeyLocator

Typically, the default implementation that is used for request sending, request receiving, and response receiving is the KeyStoreKeyLocator implementation. The implementation for response sender, however, is usually different because of the need to determine what key to use so that the client understands the response. The server communicates with many clients that might have different keys. Therefore, for the proper response, the response sender typically uses a special key locator implementation. The two key locator implementations that handle this problem for the response sending logic are as follows:

- WSIdKeyStoreMapKeyLocator
- CertInRequestKeyLocator

The WSIdKeyStoreMapKeyLocator implementation checks the client credentials to determine which key is mapped and then uses that key for the response. The CertInRequestKeyLocator implementation uses the certificate that signed the received request to encrypt the response.

You can choose which implementation to use for your environment or you can write your own. Custom key locators must implement the com.ibm.wsspi.wssecurity.config.KeyLocator interface. With this implementation, you can locate keys from any data source you choose.

This topic focuses on configuring a key locator. See the following topics for more information:

- "Key locators" on page 142
- "Key locator default implementation" on page 143
- "Develop a key locator" on page 144

For more information about creating a key store, see Use Java keystore files in the *Security* topic.

You can configure key locators with the WebSphere Development Studio Client for iSeries or the WebSphere administrative console. See these topics for more information:

- Configure a key locator in the WebSphere Development Studio Client for iSeries (page 140)
- Configure a key locator in the WebSphere administrative console (page 141)
- Configure default key locators at the server level in the administrative console (page 142)

Configure a key locator in the WebSphere Development Studio Client for iSeries

1. Open your deployment descriptor file in the WebSphere Development Studio Client for iSeries:
   - For a Web service application, open webservices.xml in the Web Services Editor.
   - For a Web service client, open webservicesclient.xml in the Web Services Client Editor.

   For more information, see "Configure your Web services application" on page 102.
2. Click the **Port Binding** tab in the Web Services Client Editor or the **Binding Configurations** tab in the Web Services Editor.
3. Expand one of the **Binding Configuration** sections. For example, expand **Security Request Sender Binding Configuration** section.
4. Expand the **Key Locators** section.
5. Click **Add** to create a new key locator, or click **Edit** to edit an existing one.
6. Enter a key locator name. The name entered for the key locator name is used to refer to the key locator from the **Encryption information** and **Signing Information** sections.
7. Enter a key locator class.The key locator class is the implementation of the KeyLocator interface. When using default implementations, select a class from the menu.
8. Determine whether to click **Use key store**. The default implementations all use key stores. Select this option when you use the default implementations. Specify the following information:
   a. Enter a **key store storepass**. The key store storepass is the password to access the key store.

   b. Enter a **key store path**. The key store path is the location on the file system where the key store resides. Make sure that the location can be found wherever you deploy the application.
   c. Enter a **key store type**.The valid types to enter are `JKS` and `JCEKS`. JKS is used when you are not using Java Cryptography Extensions (JCE). JCEKS is used when you are using JCE. Although the JCEKS type is more secure, it may decrease performance.
   d. Click **Add** to create an entry for a Key in the key store. Specify the following information:
      1) Enter a **key alias**. The key alias is a reference to this particular key from the **Signing Information** section.
      2) Enter a **keypass**. The keypass is the password that is associated with the certificate when it is created.
      3) Enter a **key name**. The key name refers to the alias of the certificate as found in the key store.
9. Click **Add** to create a custom property. The property can be used by custom implementations of KeyLocator. For example, you can use properties with the WSIdKeyStoreMapKeyLocator default implementation. The KeyLocator has the following property names:
   - **id_**, which maps to a credential user ID
   - **mappedName_**, which maps to the key alias to use for this user name
   - **default**, which maps to a Key alias to use when a credential does not have an associated id_ entry

   A typical set of properties for this key locator could be `id_1=user1`, `mappedName_1=key1`, `id_2=user2`, `mappedName_2=key2`, `default=key3`. If user1 or user2 authenticates, then the associated key1 or key2 is used, respectively. However, if none of the user properties authenticate or the user is not user1 or user2, then key3 is used.
   a. Enter a **Name**.The name entered is the property name.
   b. Enter a **Value**. This value entered is the property value.
10. Save the file.
11. Repeat the process until you have configured the necessary key locators for your applications.

Configure a key locator in the WebSphere administrative console

You can configure binding information in the administrative console, but for extensions, you must use the WebSphere Development Studio Client for iSeries.

Perform the following steps in the administrative console to configure a key locator for a specific application:

1. Click **Applications —> Enterprise Applications —>** *application_name*, where *application_name* is the name of your application. Under **Related Items**, click **Web Modules**.
2. Click the name of the module you are securing.
3. Under **Additional Properties**, click either **Web Services: Client Security Bindings** or **Web Services: Server Security Bindings** depending on whether you are adding the key locator to the client security bindings or the server security bindings.

   If you do not see any entries, return to the WebSphere Development Studio Client for iSeries and configure the security extensions.
4. Complete either of the following steps:
   - If you are editing your client security bindings, click **Edit** for either the **Request Sender Binding** or **Response Receiver Binding**.
   - If you are editing your server security bindings, click **Edit** for either the **Request Receiver Binding** or **Response Sender Binding**.
5. Click **Key Locators**. The same information that was used to configure a key locator with the WebSphere Development Studio Client for iSeries applies at this point in the steps. See step 5 in Configure a key locator in the WebSphere Development Studio Client for iSeries (page 140).

Configure default key locators at the server level in the administrative console

A key locator typically locates a key store in the file system. The location of key stores can vary from machine to machine so it is often helpful to configure a default key locator for a specific machine and reference it from within the encryption or signing information. This information is found within the binding configurations of any application installed on that machine. This suggestion enables you to define a single key locator for all applications that need to use the same keys.

Perform the following steps in the WebSphere administrative console to configure default key locators at the server level:

1. Click **Servers —> Application Servers —>** ***server_name***, where *server_name* is the name of your application server.
2. Under **Additional Properties**, click **Web Services: Default bindings for Web Services Security**.
3. Click **Key Locators**. The same information that was used to configure a key locator using the WebSphere Development Studio Client for iSeries applies at this point in the steps. See step 5 in Configure a key locator in the WebSphere Development Studio Client for iSeries (page 140).

Key locators: A key locator (com.ibm.wsspi.wssecurity.config.KeyLocator) is a abstraction of the mechanism that retrieves the key for digital signature and encryption. You can use any of the following infrastructure from which to retrieve the keys depending upon the implementation:

- Java key store file
- Database
- LDAP server

Key locators search the key using some type of a clue. The following types of clues are allowed:

- A string label of the key, which is explicitly passed through the application programming interface (API). The relationships between each key and its name (string label) is maintained inside the key locator.
- The execution context of the key locator; explicit information is not passed to the key locator. A key locators, by itself, determines the appropriate key according to their execution context.

For example, key locators can obtain the identity of the caller from the context and can retrieve the public key of the caller for response encryption.

Note: Current versions of key locators do not support the retrieval of verification keys because current Web services security implementations do not support the secret key-based signature. Since the key locators support the public key-based signature only, the key for verification is embedded in the X.509 certificate as a <BinarySecurityToken> element in the incoming message.

Usage scenarios

This topic describes the usage scenarios for key locators.

Signing

The name of the signing key is specified in the Web services security configuration. This value is passed to the key locator and the actual key is returned. The corresponding X.509 certificate can be returned also.

Verification

As described previously, key locators are not used in signature verification.

Encryption

The name of the encryption key is specified in the Web services security configuration. This value is passed to the key locator and the actual key is returned.

Decryption

The Web services security specification recommends the usage of the key identifier instead of the key name. However, while the algorithm for computing the identifier for the public keys is defined in Internet Engineering Task Force (IETF) Request for Comment (RFC) 3280, there is no agreed upon algorithm for the secret keys. Therefore, the current implementation of Web services security uses the identifier only when public key-based encryption is performed. Otherwise, the ordinal key name is used.

When you use public key-based encryption, the value of key identifier is embedded in the incoming encrypted message. Then, the Web services security implementation searches for all the keys managed by the key locator and decrypts the message using the key whose identifier value matches the one in the message.

When you use secret key-based encryption, the value of key name is embedded in the incoming encrypted message. The Web services security implementation asks the key locator for the key whose name matches the one in the message and decrypts the message using the key.

Key locator default implementation: A key locator is an abstraction of the mechanism that retrieves keys for digital signature and encryption. A key locator is implemented by providing a class that implements the com.ibm.wsspi.wssecurity.config.KeyLocator interface. WebSphere Application Server - Express provides the following key locator implementations:

- com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator
- com.ibm.wsspi.wssecurity.config.CertInRequestKeyLocator
- com.ibm.wsspi.wssecurity.config.WSIdKeyStoreMapKeyLocator

KeyStoreKeyLocator

The KeyStoreKeyLocator retrieves keys from a key store using the java.security.KeyStore class. To retrieve a key, the key locator uses the location, the type of the key store, and a name or label that specifies a particular key. The location and type of key store are provided in the <KeyLocator> element of the Web services security binding file (ws-security.xml, ibm-webservices-bnd.xmi, or ibm-webservicesclient-bnd.xmi).

The name or label of the key to use is determined by the sender or receiver. For example, a request sender that is going to digitally sign a request uses the name of the request receiver to retrieve the public key of the receiver. The KeyStoreKeyLocator is normally used for request sending, request receiving, and response receiving.

Response sending poses a special challenge. A server sends responses to many clients and some of those clients might have multiple keys, which can make it difficult for the server to retrieve the correct key. WebSphere Application Server - Express provides the following key locators to address this situation. These key locators are normally used for response sending.

- **CertInRequestKeyLocator**
  The CertInRequestKeyLocator uses the certificate that signed the received request to encrypt the response.
- **WSIdKeyStoreMapKeyLocator**
  The WSIdKeyStoreMapKeyLocator maps the identity of the current thread of execution to a public key name. This public key is then used to encrypt the response. The mapping between the identities and the public key names is specified by properties in the <KeyLocator> element within the Web services security binding file (ws-security.xml or ibm-webservices-bnd.xmi).

  Consider the following mapping for an authenticated user ID to a public key, where `id_`*n* represents the authenticated user ID and `mappedName_`*n* represents the public key, and where *n* has to be matched.

You can also specify a default mapping to map identities for which an explicit mapping is not found. To specify a default, use the default key in the property.

Develop a key locator: Perform the following steps to develop your own key locator:

1. Define the key locator class method. WebSphere Application Server - Express provides the com.ibm.wsspi.wssecurity.config.KeyLocator key locator interface, which defines the following methods:
   - `void init(java.util.Map map) throws SoapSecurityException`
     This method initializes the object. map is a map object that contains name and value pairs. You can specify these name and value pairs in the administrative console: click **Application Servers —>** ***server_name*** **—> Web Services: Default bindings for Web Services Security —> Key Locators —>** ***key_locator_name*** **—> Properties —> New**, where *server_name* is the name of your server, and *key_locator_name* is the name of your deployed key locator implementation.
   - `java.util.Set getNames(java.lang.Object context) throws KeyLocatorException`
     This method returns a Set object that contains all the abstract key name values. The input parameter is reserved for the future use.
   - `java.security.Key getEncryptionKey(java.lang.String name, java.lang.Object context) throws KeyLocatorException`
     This method returns an encryption key. For the input parameters, `name` is an abstract key name, and `context` is reserved for the future use.
   - `java.security.Key getDecryptionKey(java.lang.String name, java.lang.Object context) throws KeyLocatorException`
     This method returns an decryption key. For the input parameters, `name` is an abstract key name and `context` is reserved for the future use.
   - `java.security.Key getSigningKey(java.lang.String name) throws KeyLocatorException`
     This method returns a signing key. The input parameter is an abstract key name.
   - `java.security.Key getVerificationKey(java.lang.String name) throws KeyLocatorException`
     This method returns a verification key. This function is not implemented in current Web services security run time because the verification key is embedded in the received message as <BinarySecurityToken>. The input parameter is an abstract key name.
   - `java.lang.String getName(java.security.Key key) throws KeyLocatorException`
     This method returns an abstract key name that corresponds to the specified key. The input parameter is a key that can be retrieved through the KeyLocator object.
   - `java.security.cert.Certificate getCertificate(java.security.Key key) throws KeyLocatorException`
     This method returns a certificate object that corresponds to the specified key. The input parameter is a key that can be retrieved through the KeyLocator object.
   - `java.security.cert.Certificate getCertificate(java.lang.String name) throws KeyLocatorException`
     This method returns a certificate object that corresponds to the abstract key that is specified as the input parameter (an abstract key name).
   - `java.lang.String getName(java.lang.String name) throws KeyLocatorException`
     This method returns a concrete key name that corresponds to the given abstract key name, The key name is used as the value for the <KeyName> element. The input parameter is an abstract key name.

   You must configure the following methods implemented by the custom key locator implementation.

   Note: This listing only shows the methods and does not include an implementation.

```
import com.ibm.wsspi.wssecurity.SoapSecurityException;
import com.ibm.wsspi.wssecurity.config.KeyLocator;
import com.ibm.wsspi.wssecurity.config.KeyLocatorException;
import java.security.Key;
import java.security.cert.Certificate;
import java.util.Map;
import java.util.Set;
```

```
public class MyKeyLocatorImpl implements KeyLocator {
  public void init(Map map) throws SoapSecurityException {
    // Initialize the key locator object.
  }

  public Set getNames(Object context) throws KeyLocatorException {
    // Returns all the abstract key "name"s.
  }

  public Key getEncryptionKey(String name, Object context) throws KeyLocatorException {
    // Returns the encryption key that corresponds to the given abstract "name".
  }

  public Key getDecryptionKey(String name, Object context) throws KeyLocatorException {
    // Returns the decryption key that corresponds to the given abstract "name".
  }

  public Key getSigningKey(String name) throws KeyLocatorException {
    // Returns the signing key that corresponds to the given abstract "name".
  }

  public Key getVerificationKey(String name) throws KeyLocatorException {
    // Returns the verification key that corresponds to the given abstract "name".
  }

  public String getName(Key key) throws KeyLocatorException {
    // Returns the abstract "name" that corresponds to the given key.
  }

  public Certificate getCertificate(Key key) throws KeyLocatorException {
    // Returns the certificate object that corresponds to the given key.
  }

  public Certificate getCertificate(String name) throws KeyLocatorException {
    // Returns the certificate object that corresponds to the given abstract "name".
  }

  public String getName(String name) throws KeyLocatorException {
    // Returns the concrete "name" that corresponds to the given abstract "name".
  }
}
```

2. Compile the implementation. Make sure that /QIBM/ProdData/WebASE/ASE5/lib/was-wssecurity.jar is in the compiler class path.
3. Copy the class file to a location in the class path, preferably the /QIBM/UserData/WebASE/ASE5/*instance*/lib/ext directory, where *instance* is the name of your instance.
4. Restart the application server.
5. With the WebSphere administrative console, delete default key locator configuration. Click **Application Servers —>** *server_name* **Web Services: Default bindings for Web Services Security —> Key Locators —>** *key_locator_name*, where *server_name* is the name of your application server, and *key_locator_name* is the name of the default key locator.

   Select the checkbox next to specific key locator name and click **Delete**.
6. Add your custom key locator. Click **New**. Verify that the class name is dot-separated and appears in the class path.
7. Under **Additional Properties**, click **Properties** to add additional properties that are required to initialize the custom key locator. These properties are passed to the init(java.util.Map) method of your implementation when it extends the com.ibm.wsspi.wssecurity.config.KeyLocator interface as described in the first step.
8. Save the configuration.

9. Update the runtime configuration by clicking **Servers —> Application Servers —>** ***server_name* —> Web Services: Default bindings for Web Services Security** (where *server_name* is the name of your application server) or **Security —> Web services**.
10. Restart the application to use the new key locator implementation.

Configure a collection certificate store: A collection certificate store is a collection of non-root, certificate authority (CA) certificates and certificate revocation lists (CRLs). This collection of CA certificates and CRLs are used to check the signature of a digitally signed Simple Object Access Protocol (SOAP) message.

The collection certificate stores are utilized when processing a received SOAP message. They are configured in the securityRequestReceiverBindingConfig section of the binding file for servers and in the securityResponseReceiverBindingConfig section of the binding file for clients.

For more information, see "Collection certificate store" on page 149.

You can configure the collection certificate either by using the following tools:

- WebSphere Development Studio Client for iSeries
  - Configure server-side collection certificate stores (page 146)
  - Configure client-side collection certificate stores (page 147)
- WebSphere administrative console
  - Configure collection certificate stores (page 147)
  - Configured default collection certificate stores at the server level (page 148)
  - Configure default collection certificate stores at the cell level (page 148) (Network Deployment only)

Configure the server-side collection certificate store with the WebSphere Development Studio Client for iSeries

Perform these steps to configure the server-side collection certificate store:

1. Open the webservices.xml deployment descriptor for your Web services application in the Web Services Editor of the WebSphere Development Studio Client for iSeries. For more information, see "Configure your Web services application" on page 102.
2. Click the **Binding Configurations** tab.
3. Select one of the Web service description binding entries under the **Port Component Binding** section.
4. Expand the **Request Receiver Binding Configuration Details —> Certificate Store List —> Collection Certificate Store** section.
5. Click **Add** to create a new collection certificate store.
6. Enter a name in the **Name** field. This is a name that is referenced in the **Certificate store reference** field in the Signing information dialog.
7. Leave the **Provider** field as `IBMCertPath`.
8. Click **Add** to enter the path to your certificate store. For example, the path could be `${USER_INSTALL_ROOT}/etc/ws-security/samples/intca2.cer`.

   Note: It is recommended that you use the WebSphere Application Server - Express variables (such as `${USER_INSTALL_ROOT}`) for specifying the path to your certificate store. For more information about setting the variables, see Manage substitution variables with the administrative console in the *Administration* topic.
9. If you have additional certificate store paths, click **Add** to add them.
10. Click **OK** when you have added all necessary paths.
11. Save the file.

Configure client-side collection certificate stores with the WebSphere Development Studio Client for iSeries

Perform these steps to configure the client-side collection certificate store:

1. Open the webservicesclient.xml file in the Web Services Client Editor of the WebSphere Development Studio Client for iSeries. For more information, see “Configure your Web services application” on page 102.
2. Click the **PortBinding** tab
3. Select one of the **Port Qualified Name Binding** entries.
4. Expand the **Security Response Receiver Binding Configuration —> Certificate Store List —> Collection Certificate Store** section.
5. Click **Add** to create a new collection certificate store.
6. Enter a name in the **Name** field. This is a name that is referenced in the **Certificate store reference** field in the Signing information dialog.
7. Leave the **Provider** field as `IBMCertPath`.
8. Click **Add** to enter the path to your certificate store. For example, the path could be `${USER_INSTALL_ROOT}/etc/ws-security/samples/intca2.cer`.
9. If you have additional certificate store paths, click **Add** to add them.
10. Click **OK** when you have added all necessary paths.
11. Save the file.

Configure collection certificate stores in the WebSphere administrative console

Perform these steps in the WebSphere administrative console to configure a collection certificate store:

1. Click **Applications —> Enterprise Applications —>** ***application_name***, where *application_name* is the name of your Web services application.
2. Under **Related Items**, click **Web Modules**.
3. Click the name of the module you want to secure.
4. If you want to add the collection certificate store to the client sercurity bindings, click **Web Services: Client Security Bindings**.

   If you want to add the collection certificate store to the server security bindings, click **Web Services: Server Security Bindings**.

   If you do not see any entries, you must configure the security extensions in the deployment descriptor for your application and redeploy it. For more information, see “Configure the Web services client for response digital signature verification” on page 153 or “Configure the Web services server for request digital signature verification” on page 155.
5. If you are editing your client security bindings, click **Edit** for the **Response Receiver Binding**.

   If you are editing your server security bindings, click **Edit** for the **Request Receiver Binding**.
6. Click **Collection Certificate Store**.
7. Click a listed **Certificate Store Name** to edit an existing one, or **New** to add a new certificate store name.
8. Enter a name in the **Certificate Store Name** field. This is a name that is referenced in the **Certificate Store** field on the Signing information configuration page.
9. Leave the **Certificate Store Provider** field as `IBMCertPath`.
10. Click **Apply**.
11. Under **Additional Properties**, click **X.509 Certificates**
12. Click **New**.
13. Enter the path to your certificate store. For example, the path could be `${USER_INSTALL_ROOT}/etc/ws-security/samples/intca2.cer`.

14. Click **OK**.
15. If you have any additional certificate store paths to enter, click **New** and add the path names.
16. Save the configuration.

Configure default collection certificate stores at the server level in the administrative console

A certificate store typically refers to a certificate store located in the file system. The location of the certificate store can vary from machine to machine so you may configure a default collection certificate store for a specific machine and reference it from within the signing information. The signing information is found within the binding configurations of any application installed on the machine. This suggestion enables you to define a single collection certificate store for all of the applications that need to use the same certificates.

Perform the following steps in the WebSphere administrative console to configure default collection certificate stores at the server level:

1. Click **Servers —> Application Servers —>** ***server_name***, where *server_name* is the name of your application server.
2. Under **Additional Properties**, click **Web Services: Default bindings for Web Services Security**.
3. Click **Collection Certificate Store**.
4. Click a listed **Certificate Store Name** to edit an existing store or click **New** to add a new store.
5. Enter a name in the **Certificate Store Name** field. This is a name that is referenced in the **Certificate Store** field on the **Signing information** configuration page.
6. Leave the **Certificate Store Provider** field as `IBMCertPath`.
7. Click **Apply**.
8. Under **Additional Properties**, click **X.509 Certificates**.
9. Click **New**.
10. Enter the path to your certificate store. For example, the path could be `${USER_INSTALL_ROOT}/etc/ws-security/samples/intca2.cer`
11. Click **OK**.
12. If you have any additional certificate store paths to enter, click **New** and add the path names
13. Save the configuration.

Configure default collection certificate stores at the cell level in the administrative console (Network Deployment only)

Complete the following steps in the WebSphere administrative console to configure the default collection certificate stores at the cell-level in a Network Deployment environment:

1. Click **Security —> Web Services**.
2. Click **Collection Certificate Store**.
3. Click a listed **Certificate Store Name** to edit an existing store, or click **New** to add a new store.
4. Enter a name in the **Certificate Store Name** field. This is a name that is referenced in the **Certificate Store** field on the **Signing information** configuration page.
5. Leave the **Certificate Store Provider** field as `IBMCertPath`.
6. Click **Apply**.
7. Under **Additional Properties**, click **X.509 Certificates**.
8. Click **New**.
9. Enter the path to your certificate store. For example, the path could be `${USER_INSTALL_ROOT}/etc/ws-security/samples/intca2.cer`
10. Click **OK**.

11. If you have any additional certificate store paths to enter, click **New** and add the path names.
12. Save the configuration.

Collection certificate store: Collection certificate store is one kind of certificate store. A certificate store is defined as javax.security.cert.CertStore in Java CertPath API. A collection certification store contains both non-root certificate authority (CA) certificates and certificate revocation lists (CRLs). The Java CertPath API defines two types of certificate stores: collection certificate store and LDAP certificate store. A collection certificate store accepts the certificates and CRLs as java collection objects. The LDAP certificate store accepts certificates and CRLs as LDAP entries. CertPath API uses the certificate store and the trust anchor to validate the incoming X.509 certificate that is embedded in the SOAP message. For more information on trust anchors, see “Trust anchors” on page 151.

The Web services security implementation in the WebSphere Application Server - Express supports the collection certificate store. Each certificate and CRL is passed as a encoded file.

Configure trust anchors: This document describes how to create and configure trust anchors, or trust stores at the application level. The document does not provide information on how to create and configure trust anchors at the server level. Trust anchors defined at the application level have a higher precedence over trust anchors defined at the server level.

For more conceptual information, see “Default bindings for Web services” on page 96. For more conceptual information on trust anchors, see “Trust anchors” on page 151.

A trust anchor specifies key stores that contain root-trusted certificates, which validate the signer certificate. These key stores are used by the request receiver (as defined in the ibm-webservices-bnd.xmi file) and the response receiver (as defined in the ibm-webservicesclient-bnd.xmi file when Web services is acting as client) to validate the signer certificate of the digital signature. The key stores are critical to the integrity of the digital signature validation. If they are tampered with, the result of the digital signature verification is doubtful and comprised. Therefore, it is recommended that you secure these key stores. The binding configuration specified for the request receiver in the ibm-webservices-bnd.xmi file must match the binding configuration for the response receiver in the ibm-webservicesclient-bnd.xmi file.

You can create an application-level trust anchor and configure it using the WebSphere Development Studio Client for iSeries or the WebSphere administrative console. This topic describes both approaches.

The following steps assume that you have already created a Web services-enabled application the implements the Java 2 Platform, Enterprise Edition (J2EE) with JSR 109 specification.

Configuring a trust anchor with WebSphere Development Studio Client for iSeries

Perform the following steps to configure the client-side response receiver:

1. Open the webservicesclient.xml file in the Web Services Client Editor of the WebSphere Development Studio Client for iSeries. For more information, see “Configure your Web services application” on page 102.
2. Click the **Port Binding** tab.
3. Expand the **Port Qualified Name Binding** section and either select an existing entry or add a new port binding. Click **Add** to add a new port binding.
4. Expand the **Trust Anchor** section and click **Add**. Specify the following information:
   - Enter a unique name within the port binding for the **Trust anchor name**. The name is used to reference the trust anchor that is defined.
   - Enter the key store password, path, and key store type. The supported key store types are **JCE** and **JCEKS**.

   When you start the application, the configuration is validated in the run time while the binding information is loading.

5. Save the file.

Next, perform the following steps to configure the server-side request receiver:

1. Open the webservices.xml file with the Web Services Editor of the WebSphere Development Studio Client for iSeries. For more information, see “Configure your Web services application” on page 102.
2. Click the **Bindings** tab.
3. In the **Web Service Description Bindings** section, either select an existing entry or click **Add** and add a new Web services descriptor.
4. Click the **Binding Configurations** tab.
5. In the **Trust Anchor** section, click **Add** and enter the following information:
   - Enter a unique name within the binding for the **Trust anchor name**. This unique name is used to reference the trust anchor that is defined.
   - Enter the key store password, path, and key store type. The supported key store types are **JCE** and **JCEKS**.

   When you start the application, the configuration is validated in the run time while the binding information is loading.
6. Save the file.
7. “Configure the Web services server for request digital signature verification” on page 155.
8. (Optional) If the Web service is also acting as a client, complete the configuration process for the client-side response receiver. For more information, see “Configure the Web services client for response digital signature verification” on page 153.

Configure a trust anchor with the administrative console

Before completing the following steps, it is assumed that a Web services-enabled enterprise application was deployed to the WebSphere Application Server - Express.

Perform the following steps in the WebSphere administrative console to configure the client-side response receiver and the server-side request receiver:

1. Click **Applications —> Enterprise Applications —>** ***enterprise_application***, where *enterprise_application* is the name of your Web services application.
2. In the Related Links section, click **Web Modules**, and then click the Web services module.
3. (Optional) If the Web service is also acting as a client, edit the response receiver binding information:
   a. Click **Web Services: Client Security Bindings**.
   b. Under **Response Receiver Binding**, click **Edit**.
   c. Under **Additional Properties**, click **Trust Anchors**.
   d. Click **New** to create a new trust anchor, and enter the following information:
      - Enter a unique name within the request receiver binding for the **Trust anchor name** field. The name is used to reference the trust anchor that is defined.
      - Enter the key store password, path, and key store type.

   When you start the application, the configuration is validated in the run time while the binding information is loading.
4. Edit the request receiver binding information:
   a. Return to the main page for your Web services module.
   b. Click **Web Services: Server Security Bindings**.
   c. Under **Request Receiver Binding**, click **Edit**.
   d. Under **Additional Properties**, click **Trust Anchors**.
   e. Click **New** to create a new trust anchor, and enter the following information:

- Enter a unique name within the request receiver binding for the **Trust anchor name** field. The name is used to reference the trust anchor that is defined.
- Enter the key store password, path and key store type.

When you start the application, the configuration is validated in the run time while the binding information is loading.

5. Save the configuration.

Trust anchors: The trust anchor stores the trusted root certificate authority (CA) certificates. The trust anchor is defined as javax.security.cert.TrustAnchor in Java CertPath API. The Java CertPath API uses the trust anchor and the certificate store to validate the incoming X.509 certificate that is embedded in the Simple Object Access Protocol (SOAP) message. For more information on the certificate store, see "Collection certificate store" on page 149.

The Web services security implementation in WebSphere Application Server - Express supports this trust anchor. In WebSphere Application Server - Express, the trust anchor is represented as a Java key store object. The type, path, and password of the key store are passed to the implementation through the Administration Console or by scripting.

Configure the Web services client for request signing: This task provides the steps needed to configure the client for request signing. Use these steps to modify the extensions to indicate which parts of the request that you want to sign. Also, use the steps to configure the bindings to indicate how the parts of the request are to be signed.

Perform the following steps in the WebSphere Development Studio Client for iSeries to configure the parts of the Simple Object Access Protocol (SOAP) request that you want to digitally sign:

1. Open the webservicesclient.xml file in the Web Services Client Editor of the WebSphere Development Studio Client for iSeries. For more information, see "Configure your Web services application" on page 102.
2. Click the **Security Extensions** tab.
3. Expand **Request Sender Configuration —> Integrity**. *Integrity* refers to digital signature while confidentiality refers to encryption. Integrity decreases the risk of data modification while the data is transmitted across the Internet. For more information on digitally signing SOAP messages, see "XML digital signature" on page 116.
4. Select the parts of the message in which to sign by clicking **Add** and selecting one of the following message parts:
   - **Body**
     This is the user data portion of the message.
   - **Timestamp**
     The time stamp determines if the message is valid based on the time the message was sent and then received. If time stamp is selected, proceed to the next step to add a created time stamp to the message.
   - **Securitytoken**
     The security token authenticates the client. If **securitytoken** is selected, the message is signed.

   You can choose to digitally sign the message using a time stamp if the **Add Created Time Stamp** field is selected and configured. You can choose to digitally sign the message using a security token if a login configuration authentication method is selected.
5. Expand the **Add Created Time Stamp** section. Select this if you want a timestamp added to the message. You can additionally specify an expiration time for the timestamp. This helps defend against replay attacks.

   The lexical representation for duration is the ISO 8601 extended format `P`*n*`Y`*n*`M`*n*`DT`*n*`H`*n*`M`*n*`S`, where the following values apply:
   - *n*`Y` represents the number of years.

   - *n*M is the number of months.
   - *n*D is the number of days.
   - T is the date and time separator.
   - *n*H is the number of hours.
   - *n*M is the number of minutes.
   - *n*S is the number of seconds. The number of seconds can include decimal digits to arbitrary precision.

   For example, to indicate a duration of 1 year, 2 months, 3 days, 10 hours, and 30 minutes, set the expiration time to P1Y2M3DT10H30M. Typically, you configure a message timestamp for about 10 to 30 minutes. For an expiration of 10 minutes, specify P0Y0M0DT0H10M0S.

6. (Optional) If you have configured the client and server signing information correctly, but you receive a "Soap body not signed" error when you run the client, you may need to configure the actor on the client with the Web Services Client Editor:
   - Click **Security Extensions —> Client Service Configuration Details** and indicate the actor information in the **ActorURI** field.
   - Click **Security Extensions —> Request Sender Configuration section —> Details** and indicate the actor information in the **Actor** field.

   Also, configure the same actor strings for the Web service on the server, which processes the request and sends the response back. You can do this from the following locations:
   - Click **Security Extensions —> Server Service Configuration** section. Make sure that the **Actor URI** field contains the same actor string that is indicated on the client side.
   - Click **Security Extensions —> Response Sender Service Configuration Details —> Details** and indicate the actor information in the **Actor** field.

   The actor information on both the client and server must refer to the same exact string. When the actor fields on the client and server match, then the request or response is acted upon instead of being forwarded downstream. The actor fields may be different when you have Web services acting as a gateway to other Web services. However, in all other cases, make sure that the actor information matches on the client and server.

   When Web services are acting as a gateway and they do not have the same actor configured as the request passing through the gateway, Web services do not process the message from a client. Instead, these Web services send the request downstream. The downstream process that contains the correct actor string processes the request. The same situation occurs for the response. Therefore, it is important that you verify that the appropriate client and server actor fields are synchronized.
7. Save the file.

Next, perform the following steps in the Web Services Client Editor to configure the information that is needed to digitally sign the request parts:

1. Click the **Port Binding** tab.
2. Expand **Security Request Sender Binding Configuration —> Signing Information**.
3. Select **Edit** to view the signing information. The following table describes the purpose of this information. Some of these definitions are based on the XML-Signature Syntax and Processing specification

   (http://www.w3.org/TR/xmldsig-core).

Name	Purpose
Canonicalization method algorithm	The canonicalization method algorithm is used to canonicalize the SignedInfo element before it is digested as part of the signature operation.

Name	Purpose
Digest method algorithm	The digest method algorithm is the algorithm applied to the data after transforms are applied, if specified, to yield the <DigestValue> element. The signing of the DigestValue binds resource content to the signer key. The algorithm selected for the client request sender configuration must match the algorithm selected in the server request receiver configuration.
Signature method algorithm	The signature method is the algorithm that is used to convert the canonicalized <SignedInfo> into the <SignatureValue>. The algorithm selected for the client request sender configuration must match the algorithm selected in the server request receiver configuration.
Signing key name	The signing key name represents the key entry associated with the signing key locator. The key entry refers to an alias of the key (which is found in the key store or wherever the certificates are stored based upon the key locator implementation) that is used to sign the request.
Signing key locator	The signing key locator represents a reference to a key locator implementation that locates the correct key store where the alias and certificate reside. For more information on configuring key locators, see “Configure a key locator” on page 139.

4. Save the file.

Configure the Web services client for response digital signature verification: This task provides the steps needed to configure the client for response digital signature verification. Use these steps to modify the extensions that indicate which parts of the message must be verified. Also, use these steps to configure the bindings that indicate how these parts of the message must be verified.

Perform the following steps in the WebSphere Development Studio Client for iSeries to configure the parts of the SOAP message in which the digital signature must be verified:

1. Open the webservicesclient.xml file in the Web Services Client Editor of the WebSphere Development Studio Client for iSeries. For more information, see “Configure your Web services application” on page 102.
2. Click the **Security Extensions** tab.
3. Expand the **Response Receiver Configuration —> Required Integrity** settings. *Required Integrity* refers to message parts that require digital signature verification. Digital signature verification decreases the risk that the message parts have been modified while the message is transmitted across the Internet. For more conceptual information on digital signature, see “XML digital signature” on page 116.
4. Select the parts of the message that must be verified. You can determine which parts of the message to select by looking at the Web service response sender configuration. To add parts of the message, click **Add** and select one of the following three parts:
   - **Body**
     This is the user data portion of the message.
   - **Timestamp**
     The time stamp determines if the message is valid based on the time the message was sent and then received. If **timestamp** is selected, you can expand **Response Receiver Configuration —> Add Received Time Stamp** to add the received time stamp to the message.
   - **Securitytoken**
     The security token authenticates the client. If **Securitytoken** is selected, the message is signed.

5. (Optional) If you have configured the client and server signing information correctly, but you receive a “Soap body not signed” error when you run the client, you may need to configure the actor in the following locations on the client in the Web Services Client Editor:
   - Click **Security Extensions —> Client Service Configuration Details** and indicate the actor information in the **ActorURI** field.
   - Click **Security Extensions —> Request Sender Configuration section —> Details** and indicate the actor information in the **Actor** field.

   Also, configure the same actor strings for the Web service on the server, which processes the request and sends the response back. You can do this from the following location in the Web Services Editor:
   - Click **Security Extensions —> Server Service Configuration** section. Make sure that the **Actor URI** field contains the same actor string that is indicated on the client side.
   - Click **Security Extensions —> Response Sender Service Configuration Details —> Details** and indicate the actor information in the **Actor** field.

   The actor information on both the client and server must refer to the same exact string. When the actor fields on the client and server match, then the request or response is acted upon instead of being forwarded downstream. The actor fields might be different when you have Web services acting as a gateway to other Web services. However, in all other cases, make sure that the actor information matches on the client and server.

   When Web services are acting as a gateway and they do not have the same actor configured as the request passing through the gateway, Web services do not process the message from a client. Instead, these Web services send the request downstream. The downstream process that contains the correct actor string processes the request. The same situation occurs for the response. Therefore, it is important that you verify that the appropriate client and server actor fields are synchronized.
6. Save the file.

Next, perform the following steps in the Web Services Client Editor to configure the information that is needed to verify digital signatures:
1. Click the **Port Binding** tab.
2. Expand the **Security Response Receiver Binding Configuration —> Signing Information** settings. Click **Edit** to view the signing information. The following table describes the purpose for each of these selections. Some of these definitions are based on the XML-Signature Syntax and Processing specification

   (http://www.w3.org/TR/xmldsig-core).

Name	**Purpose**
Canonicalization method algorithm	The canonicalization method is the algorithm that is used to canonicalize the SignedInfo element before it is digested as part of the signature operation.
Digest method algorithm	The digest method algorithm is the algorithm applied to the data after transforms are applied, if specified, to yield the <DigestValue>. The signing of the DigestValue binds resource content to the signer key. The algorithm selected for the client response receiver configuration must match the algorithm selected in the server response sender configuration.
Signature method algorithm	The signature method is the algorithm that is used to convert the canonicalized <SignedInfo> into the <SignatureValue>. The algorithm selected for the client response receiver configuration must match the algorithm selected in the server response sender configuration.

Name	Purpose
Use certificate path reference or **Trust any certificate**	When a message is signed, the public key used to sign it is transmitted with the message. In order to validate this public key at the receiving end, you should configure a certificate path reference. By selecting **User certificate path reference**, you must configure a trust anchor reference and certificate store reference to validate the certificate sent with the message. By selecting **trust any certificate**, the signature is validated by the certificate sent with the message without the certificate itself being validated.
Use certificate path reference —> Trust anchor reference	A trust anchor is a configuration that refers to a key store containing trusted self-signed and certificate authority (CA) certificates. These are trusted certificates for any application in your deployment. Refer to "Configure trust anchors" on page 149 for more information.
Use certificate path reference —> Certificate store reference	A certificate store is a configuration that contains a collection of X.509 certificates that are not trusted for all applications in your deployment, but might be used to validate certificates for an application as an intermediary.

3. Save the file.

Configure the Web services server for request digital signature verification: Use this task to configure the server for request digital signature verification. The steps describe how to modify the extensions to indicate which parts of the request to verify. Also, the steps describe how to configure the bindings to indicate how to verify the parts of the request.

Perform the following steps in the WebSphere Development Studio Client for iSeries to configure the parts of the Simple Object Access Protocol (SOAP) request that the digital signature must verify:

1. Open the webservices.xml deployment descriptor for your Web services application in the Web Services Editor of the WebSphere Development Studio Client for iSeries. For more information, see "Configure your Web services application" on page 102.
2. Click the **Security Extensions** tab.
3. Expand the **Request Receiver Service Configuration Details —> Required Integrity** settings. *Required integrity* refers to the parts of the message that require digital signature verification. The purpose of digital signature verification is to make sure that the message parts have not been modified while it was transmitted across the Internet.
4. Select the parts of the message to verify. You can determine which parts of the message to verify by looking at the Web Service Request Sender Configuration in the client application. To add message parts to verify, click **Add** and select one of the following message parts:
   - **Body**
     This is the user data in the message.
   - **Timestamp**
     If selected, a timestamp is added to the message.
   - **SecurityToken**
     If selected, the authentication information is added to the message.
5. Expand the **Add Received Time Stamp** section. The **Add Received Time Stamp** field indicates to validate the **Add Created Time Stamp** that is configured by the client. You must select option this if you selected **Add Created Time Stamp** on the client. The time stamp ensures message integrity by indicating the freshness of the request. This option helps to defend against replay attacks.
6. (Optional) If you have configured the client and server signing information correctly, but you receive a "Soap body not signed" error when you run the client, you may need to configure the actor in the following locations on the client in the Web Services Client Editor:

- Click **Security Extensions —> Client Service Configuration Details** and indicate the actor information in the **ActorURI** field.
- Click **Security Extensions —> Request Sender Configuration section —> Details** and indicate the actor information in the **Actor** field.

Also, configure the same actor strings for the Web service on the server, which processes the request and sends the response back. You can do this from the following location in the Web Services Editor:

- Click **Security Extensions —> Server Service Configuration** section. Make sure the **Actor URI** field contains the same actor string that is indicated on the client side.
- Click **Security Extentions —> Response Sender Service Configuration Details —> Details** and indicate the actor information in the **Actor** field.

The actor information on both the client and server must refer to the same exact string. When the actor fields on the client and server match, then the request or response is acted upon instead of being forwarded downstream. The actor fields might be different when you have Web services acting as a gateway to other Web services. However, in all other cases, make sure that the actor information matches on the client and server.

When Web services are acting as a gateway and they do not have the same actor configured as the request passing through the gateway, Web services do not process the message from a client. Instead, these Web services send the request downstream. The downstream process that contains the correct actor string processes the request. The same situation occurs for the response. Therefore, it is important that you verify that the appropriate client and server actor fields are synchronized.

7. Save the file.

Next, perform the following steps in the Web Services Editor to configure the information that is needed to verify digital signatures:

1. Click the **Binding Configurations** tab.
2. Expand the **Security Request Receiver Binding Configuration Details —> Signing Information** settings.
3. Click **Edit** to view the signing information. For more conceptual information on digitally signing SOAP messages, see “XML digital signature” on page 116. The following table describes the purpose for each of these selections. Some of these definitions are based on the XML-Signature Syntax and Processing specification

   (http://www.w3.org/TR/xmldsig-core).

Name	Purpose
Canonicalization method algorithm	The canonicalization method algorithm is used to canonicalize the <SignedInfo> element before it is digested as part of the signature operation. The algorithm selected for the server request receiver configuration must match the algorithm selected in the client request sender configuration.
Digest method algorithm	The digest method algorithm is the algorithm applied to the data after transforms are applied, if specified, to yield the <DigestValue>. The signing of the DigestValue binds resource content to the signer key. The algorithm selected for the server request receiver configuration must match the algorithm selected in the client request sender configuration.

Name	Purpose
Signature method algorithm	The signature method is the algorithm that is used to convert the canonicalized <SignedInfo> into the <SignatureValue>. The algorithm selected for the server request receiver configuration must match the algorithm selected in the client request sender configuration.
Use certificate path reference or **Trust any certificate**	When a message is signed, the public key used to sign it is sent with the message. This public key or certificate might not be validated at the receiving end. By selecting **User certificate path reference**, you must configure a trust anchor reference and a certificate store reference to validate the certificate sent with the message. By selecting **Trust any certificate**, the signature is validated by the certificate sent with the message without the certificate itself being validated.
Use certificate path reference: Trust anchor reference	A trust anchor is a configuration that refers to a key store that contains trusted, self-signed certificates and certificate authority (CA) certificates. These certificates are trusted certificates that you can use with any applications in your deployment. See "Configure trust anchors" on page 149 for more information.
Use certificate path reference: Certificate store reference	A certificate store is a configuration that has a collection of X.509 certificates. These certificates are not trusted for all applications in your deployment, but might be used as an intermediary to validate certificates for an application.

4. Save the file.

Configure the Web services server for response signing: This task provides the steps needed configure the server for response signing. Use these steps to modify the extensions to indicate which parts of the response that you want to sign. Also, use the steps to configure the bindings to indicate how the parts of the response are to be signed.

Perform the following steps in the WebSphere Development Studio Client for iSeries to configure the security extensions for the parts of the Simple Object Access Protocol (SOAP) message that you want to digitally sign:

1. Open the webservices.xml deployment descriptor for your Web services application in the Web Services Editor of the WebSphere Development Studio Client for iSeries. For more information, see "Configure your Web services application" on page 102.
2. Click the **Security Extensions** tab.
3. Expand **Response Sender Service Configuration Details —> Integrity**. *Integrity* refers to digital signature while confidentiality refers to encryption. Integrity decreases the risk of data modification while the data is transmitted across the Internet. For more information on digitally signing SOAP messages, see "XML digital signature" on page 116.
4. Select the parts of the message in which to sign by clicking **Add** and selecting one of the following message parts:
   - **Body**
     This is the user data portion of the message.
   - **Timestamp**
     You can choose this if **Add Created Time Stamp** is selected and configured.
   - **Securitytoken**
     If security token is selected, the authentication information is added to the message.

5. Expand the **Add Created Time Stamp** section. Select this if you want a time stamp added to the message. Also, you can specify an expiration time for the time stamp, which helps defend against replay attacks.

   The lexical representation for duration is the ISO 8601 extended format `P`*n*`Y`*n*`M`*n*`DT`*n*`H`*n*`M`*n*`S`, where the following values apply:

   - *n*`Y` represents the number of years.
   - *n*`M` is the number of months.
   - *n*`D` is the number of days.
   - `T` is the date and time separator.
   - *n*`H` is the number of hours.
   - *n*`M` is the number of minutes.
   - *n*`S` is the number of seconds. The number of seconds can include decimal digits to arbitrary precision.

   For example, to indicate a duration of 1 year, 2 months, 3 days, 10 hours, and 30 minutes, set the expiration time to `P1Y2M3DT10H30M`. Typically, you configure a message timestamp for about 10 to 30 minutes. For an expiration of 10 minutes, specify `P0Y0M0DT0H10M0S`.
6. Repeat these steps for the response receiver configuration section. The client response receiver validates the parts of the response signed by the server. Because the response receiver must validate the message signed by the server, the **Response Receiver Configuration** section requires that you configure integrity. Refer to “Configure the Web services client for response digital signature verification” on page 153 for more information.
7. (Optional) If you have configured the client and server signing information correctly, but you receive a “Soap body not signed” error when you run the client, you may need to configure the actor in the following locations on the client in the Web Services Client Editor:

   - Click **Security Extensions —> Client Service Configuration Details** and indicate the actor information in the **ActorURI** field.
   - Click **Security Extensions —> Request Sender Configuration —> Details** and indicate the actor information in the **Actor** field.

   Also, configure the same actor strings for the Web service on the server, which processes the request and sends the response back. You can do this from the following location in the Web Services Editor:

   - Click **Security Extensions —> Server Service Configuration**. Make sure that the **Actor URI** field contains the same actor string that is indicated on the client side.
   - Click **Security Extensions —> Response Sender Service Configuration Details —> Details** and indicate the actor information in the **Actor** field.

   The actor information on both the client and server must refer to the same exact string. When the actor fields on the client and server match, then the request or response is acted upon instead of being forwarded downstream. The actor fields might be different when you have Web services acting as a gateway to other Web services. However, in all other cases, make sure that the actor information matches on the client and server.

   When Web services are acting as a gateway and they do not have the same actor configured as the request passing through the gateway, Web services do not process the message from a client. Instead, these Web services send the request downstream. The downstream process that contains the correct actor string processes the request. The same situation occurs for the response. Therefore, it is important that you verify that the appropriate client and server actor fields are synchronized.
8. Save the file.

Next, perform the following steps in the Web Services Editor to configure the bindings that are needed to sign the response parts:

1. Click the **Binding Configurations** tab.
2. Expand **Response Sender Binding Configuration Details —> Signing Information**.

3. Click **Edit** to view the signing information. The signing information dialog displays.
4. Select or enter the information that is described in the following table. Some of these definitions are based on the XML-Signature Syntax and Processing specification

   (http://www.w3.org/TR/xmldsig-core).

Name	Purpose
Canonicalization method algorithm	The canonicalization method algorithm is used to canonicalize the <SignedInfo> element before it is digested as part of the signature operation. The same algorithm used here should also be used on the client response receiver. The algorithm selected for the server response sender configuration must match the algorithm selected in the client response receiver configuration.
Digest method algorithm	The digest method algorithm is the algorithm applied to the data after transforms are applied, if specified, to yield the <DigestValue>. The signing of the DigestValue binds resource content to the signer key. The algorithm selected for the server response sender configuration must match the algorithm selected in the client response receiver configuration.
Signature method algorithm	The signature method is the algorithm that is used to convert the canonicalized <SignedInfo> into the <SignatureValue>. The algorithm selected for the server response sender configuration must match the algorithm selected in the client response receiver configuration.
Signing key name	The signing key name represents the key entry associated with the signing key locator. The key entry refers to an alias of the key (which is found in the key store or wherever the certificates are stored based upon the key locator implementation) that is used to sign the request.
Signing key locator	The signing key locator represents a reference to a key locator implementation that locates the correct key store where the alias and certificate reside. For more information on configuring key locators, see "Configure a key locator" on page 139.

5. Save the file.

Configure Web services encryption and decryption

WebSphere Application Server - Express supports the encryption and descryption of SOAP messages. For more information, see "XML encryption" on page 160.

To configure your Web services to encrypt and decrypt request and responses, perform the following steps:

1. "Configure a key locator" on page 139
   Key locators are used to find keys for digital signature and encryption. WebSphere Application Server - Express provides default key locators that you can use with your digital signature configuration, or you can develop your own.
2. "Configure the Web services client for request encryption" on page 163
   Configure your Web services client to encrypt its requests to the server.
3. "Configure the Web services client for response decryption" on page 164
   Configure your Web services client to decrypt responses that it receives from the server.

4. "Configure the Web services server for request decryption" on page 166
   Configure your Web service to decrypt requests from the client.
5. "Configure the Web services server for response encryption" on page 167
   Configure your Web service to encrypt its requests to the client.

XML encryption: XML Encryption is a specification that was developed by the World Wide Web Consortium (W3C) in 2002 that contains the following information:

- The steps to encrypt data.
- The steps to decrypt encrypted data.
- The XML syntax to represent encrypted data and the information used to decrypt the data.
- A list of encryption algorithms, such as triple DES, AES, and RSA.

You can apply XML encryption to an XML element, XML element content, and arbitrary data, including an XML document. For example, suppose that you need to encrypt the <CreditCard> element shown in Example 1.

Example 1: Sample XML document

```
<PaymentInfo xmlns='http://example.org/paymentv2'>
  <Name>John Smith</Name>
  <CreditCard Limit='5,000' Currency='USD'>
    <Number>4019 2445 0277 5567</Number>
    <Issuer>Example Bank</Issuer>
    <Expiration>04/02</Expiration>
  </CreditCard>
</PaymentInfo>
```

Example 2 shows the XML document after encryption. The EncryptedData element represents the encrypted CreditCard element. The EncryptionMethod element describes the applied encryption algorithm, which is triple DES in this example. The KeyInfo element contains the information to retrieve a decryption key, which is a KeyName element in this example. The CipherValue element contains the ciphertext obtained by serializing and encrypting the CreditCard element.

Example 2: XML document encrypted with a common secret key

```
<PaymentInfo xmlns='http://example.org/paymentv2'>
  <Name>John Smith</Name>
  <EncryptedData Type='http://www.w3.org/2001/04/xmlenc#Element'
    xmlns='http://www.w3.org/2001/04/xmlenc#'>
    <EncryptionMethod
      Algorithm='http://www.w3.org/2001/04/xmlenc#tripledes-cbc'/>
    <KeyInfo xmlns='http://www.w3.org/2000/09/xmldsig#'>
      <KeyName>John Smith</KeyName>
    </KeyInfo>
    <CipherData>
      <CipherValue>ydUNqHkMrD...</CipherValue>
    </CipherData>
  </EncryptedData>
</PaymentInfo>
```

In example 2, it is assumed that both the sender and recipient have a common secret key. If the recipient has a public and private key pair, which is a most likely the case, the CreditCard element can be encrypted as shown in example 3. The EncryptedData element is the same as the EncryptedData element found in example 2. However, the KeyInfo element contains an EncryptedKey element, which represents the encrypted secret key, instead of the KeyName element found in example 2.

Example 3: XML document encrypted with the public key of the recipient

```
<PaymentInfo xmlns='http://example.org/paymentv2'>
  <Name>John Smith</Name>
  <EncryptedData Type='http://www.w3.org/2001/04/xmlenc#Element'
```

```
    xmlns='http://www.w3.org/2001/04/xmlenc#'>
    <EncryptionMethod
      Algorithm='http://www.w3.org/2001/04/xmlenc#tripledes-cbc'/>
    <KeyInfo xmlns='http://www.w3.org/2000/09/xmldsig#'>
      <EncryptedKey xmlns='http://www.w3.org/2001/04/xmlenc#'>
        <EncryptionMethod
          Algorithm='http://www.w3.org/2001/04/xmlenc#rsa-1_5'/>
        <KeyInfo xmlns='http://www.w3.org/2000/09/xmldsig#'>
          <KeyName>Sally Doe</KeyName>
        </KeyInfo>
        <CipherData>
          <CipherValue>yMTEyOTA1M...</CipherValue>
        </CipherData>
      </EncryptedKey>
    </KeyInfo>
    <CipherData>
      <CipherValue>ydUNqHkMrD...</CipherValue>
    </CipherData>
  </EncryptedData>
</PaymentInfo>
```

XML Encryption in WSS-Core

WSS-Core is a specification under development by OASIS. The specification describes enhancements to SOAP messaging to provide quality of protection through message integrity, message confidentiality, and single message authentication. The message confidentiality is realized by encryption based on XML Encryption.

The WSS-Core specification allows encryption of any combination of body blocks, header blocks, their sub-structures, and attachments of a SOAP message. The specification also requires that when you encrypt parts of a SOAP message, you must prepend a reference from the security header block to the encrypted parts of the message. The reference can be a clue for a recipient to identify which encrypted parts of the message to decrypt.

The XML syntax of the reference varies according to what information is encrypted and how it is encrypted. For example, suppose that the CreditCard element in example 4 is encrypted with either a common secret key or the public key of the recipient.

Example 4: Sample SOAP message

```
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'
  xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'>
  <SOAP-ENV:Body>
    <PaymentInfo xmlns='http://example.org/paymentv2'>
      <Name>John Smith</Name>
      <CreditCard Limit='5,000' Currency='USD'>
        <Number>4019 2445 0277 5567</Number>
        <Issuer>Example Bank</Issuer>
        <Expiration>04/02</Expiration>
      </CreditCard>
    </PaymentInfo>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The resulting SOAP messages are shown in examples 5 and 6. In these example, the ReferenceList and EncryptedKey elements are used as references, respectively.

Example 5: SOAP message encrypted with a common secret key

```
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'
  xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'>
  <SOAP-ENV:Header>
```

```
    <Security SOAP-ENV:mustUnderstand='1'
      xmlns='http://schemas.xmlsoap.org/ws/2003/06/secext'>
      <ReferenceList xmlns='http://www.w3.org/2001/04/xmlenc#'>
        <DataReference URI='#ed1'/>
      </ReferenceList>
    </Security>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <PaymentInfo xmlns='http://example.org/paymentv2'>
      <Name>John Smith</Name>
      <EncryptedData Id='ed1'
        Type='http://www.w3.org/2001/04/xmlenc#Element'
        xmlns='http://www.w3.org/2001/04/xmlenc#'>
        <EncryptionMethod
          Algorithm='http://www.w3.org/2001/04/xmlenc#tripledes-cbc'/>
        <KeyInfo xmlns='http://www.w3.org/2000/09/xmldsig#'>
          <KeyName>John Smith</KeyName>
        </KeyInfo>
        <CipherData>
          <CipherValue>ydUNqHkMrD...</CipherValue>
        </CipherData>
      </EncryptedData>
    </PaymentInfo>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Example 6: SOAP message encrypted with public key of the recipient

```
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'
  xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'>
  <SOAP-ENV:Header>
    <Security SOAP-ENV:mustUnderstand='1'
      xmlns='http://schemas.xmlsoap.org/ws/2003/06/secext'>
      <EncryptedKey xmlns='http://www.w3.org/2001/04/xmlenc#'>
        <EncryptionMethod
          Algorithm='http://www.w3.org/2001/04/xmlenc#rsa-1_5'/>
        <KeyInfo xmlns='http://www.w3.org/2000/09/xmldsig#'>
          <KeyName>Sally Doe</KeyName>
        </KeyInfo>
        <CipherData>
          <CipherValue>yMTEyOTA1M...</CipherValue>
        </CipherData>
        <ReferenceList>
          <DataReference URI='#ed1'/>
        </ReferenceList>
      </EncryptedKey>
    </Security>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <PaymentInfo xmlns='http://example.org/paymentv2'>
      <Name>John Smith</Name>
      <EncryptedData Id='ed1'
        Type='http://www.w3.org/2001/04/xmlenc#Element'
        xmlns='http://www.w3.org/2001/04/xmlenc#'>
        <EncryptionMethod
          Algorithm='http://www.w3.org/2001/04/xmlenc#tripledes-cbc'/>
        <CipherData>
          <CipherValue>ydUNqHkMrD...</CipherValue>
        </CipherData>
      </EncryptedData>
    </PaymentInfo>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Relationship to Digital Signature

The WSS-Core specification also provides message integrity, which is realized by digital signature based on XML-Signature.

Note: A combination of encryption and digital signature over common data introduces cryptographic vulnerabilities. See Section 6.1 of the XML Encryption specification for the details.

Configure the Web services client for request encryption: This task provides the steps needed to configure the client for request encryption. Use these steps to modify the extensions to indicate which parts of the request you want to encrypt. Also, use these steps to configure the bindings to indicate how the parts of the request are to be encrypted.

Perform the following steps in the WebSphere Development Studio Client for iSeries to configure the parts of the Simple Object Access Protocol (SOAP) request that you want to encrypt:

1. Open the webservicesclient.xml file in the Web Services Client Editor of the WebSphere Development Studio Client for iSeries. For more information, see “Configure your Web services application” on page 102.
2. Click the **Security Extensions** tab.
3. Expand **Request Sender Configuration —> Confidentiality**. *Confidentiality* refers to encryption while integrity refers to digital signing. Confidentiality reduces the risk of someone being able to understand the message flowing across the Internet. With confidentiality specifications, the message is encrypted before it is sent and decrypted when it is received at the correct target. For more information on encrypting , see “XML encryption” on page 160.
4. Select the parts of the message that you want to encrypt by clicking **Add** and selecting one of the following message parts:
   - **Bodycontent**
     This is the user data portion of the message.
   - **Usernametoken**
     This is the basic authentication information, if selected.
5. Save the file.

Next, perform the following steps in the Web Services Client Editor to configure the information that is needed to encrypt the message parts:

1. Click the **Port Binding** tab.
2. Expand **Security Request Sender Binding Configuration —> Encryption Information**.
3. Select an encryption option and click **Edit** to view the encryption information or click **Add** to add another option. The following table describes the purpose of this information. Some of these definitions are based on the XML-Signature Syntax and Processing specification

   (http://www.w3.org/TR/xmldsig-core).

Name	Purpose
Encryption name	The encryption name refers to the name of the encryption information entry.
Data encryption method algorithm	The data encryption method algorithms are designed for encrypting and decrypting data in fixed size, multiple octet blocks.
Key encryption method algorithm	The key encryption method algorithms are public key encryption algorithms that are specified for encrypting and decrypting keys.

Name	Purpose
Encryption key name	The encryption key name represents a Subject (Owner field of the certificate) from a certificate found by the encryption key locator, which is used by the key encryption method algorithm to encrypt the private key. The private key is used to encrypt the data. **Note:** The chosen key must be a public key of the target. Encryption must be done using the public key and decryption must be done by the target using the private key (the personal certificate of the target).
Encryption key locator	The encryption key locator represents a reference to a key locator implementation. If you write a custom key locator, the encryption key name may be anything used by the key locator to find the correct encryption key. The encryption key locator references the implementation class that locates the correct key store where this alias and certificate exists. For more information on configuring key locators, see "Configure a key locator" on page 139.

4. Save the file.

The signing key name refers to a key entry associated with the signing key locator. The key entry has an alias, which is found in the key store or wherever the certificates are stored based upon the key locator implementation. The signing key locator references the implementation class that locates the correct key store where the alias and certificate exists.

Configure the Web services client for response decryption: This task provides the steps needed to configure the client for response decryption. Use these steps to modify the extensions to indicate which parts of the response that you want to decrypt. Before configuring the client for response decryption, you must know what server parts encrypt the response. The server response encryption and client response decryption configurations must match. The steps in this task also describe how to configure the bindings to indicate how to decrypt the parts of the response.

Perform the following steps in the WebSphere Development Studio Client for iSeries to configure the parts of the SOAP response that you must decrypt:

1. Open the webservicesclient.xml file in the Web Services Client Editor of the WebSphere Development Studio Client for iSeries. For more information, see "Configure your Web services application" on page 102.
2. Click the **Security Extensions** tab.
3. Expand the **Response Receiver Configuration —> Required Confidentiality** settings.
4. Select the parts of the message that you must decrypt by clicking **Add** and selecting one of the following two message parts:
   - **Bodycontent**
     This is the user data portion of the message.
   - **Usernametoken**
     This is the basic authentication information, if selected.

   The information selected in this step is encrypted by the server in the response sender.

   Note: A username token is typically not sent in the response. Thus, you usually do not need to select **Usernametoken**.
5. Save the file.

Next, perform the following steps in the Web Services Client Editor to configure the information needed to decrypt the required message parts:

1. Click the **Port Binding** tab.
2. Expand the **Security Response Receiver Binding Configuration —> Encryption Information** settings. For more information on encrypting and decrypting SOAP messages, see “XML encryption” on page 160.
3. Click **Edit** to view the encryption information. The following table describes the purpose for each of this information. Some of these definitions are based on the XML-Signature Syntax and Processing specification

   (http://www.w3.org/TR/xmldsig-core).

Name	Purpose
Encryption name	The encryption name refers to the alias used for the encryption information entry.
Data encryption method algorithm	The data encryption method algorithms are designed for encrypting and decrypting data in fixed size, multiple octet blocks.
Key encryption method algorithm	The key encryption method algorithms are public key encryption algorithms specified for encrypting and decrypting keys.
Encryption key name	The encryption key name represents a Subject from a certificate found by the encryption key locator. The Subject is used by the key encryption method algorithm to decrypt the secret key. The secret key is used to decrypt the data. **Note:** The key chosen must be a private key of the client. Encryption must be done using the public key and decryption must be done by the private key (personal certificate). For example, the personal certificate of the client is CN=Alice, O=IBM, C=US. Therefore, the client contains the public and private key pair. The target server that sends the response encrypts the secret key using the public key for CN=Alice, O=IBM, C=US. The client decrypts the secret key using the private key for CN=Alice, O=IBM, C=US.
Encryption key locator	The encryption key locator represents a reference to a key locator implementation. For more information on configuring key locators, see “Configure a key locator” on page 139.

4. Save the file.

Note: For decryption, the encryption key name that is chosen must refer to a personal certificate that can be located by the client key locator. The Subject (owner field of the certificate) of the personal certificate should be entered in the Encryption key name, this is typically a Distinguished Name (DN). The default key locator uses the Encryption key name to find the key within the keystore. If you write a custom key locator, the encryption key name can be anything used by the key locator to find the correct encryption key. The encryption key locator references the implementation class that locates the correct key store where this alias and certificate exists.

Configure the Web services server for request decryption: This task addresses configuring the server for request decryption. It describes modifying the extensions to indicate what parts of the request to decrypt. You need to know what parts the client encrypts when sending the request because the two configurations must match. It also describes configuring the bindings to indicate how to decrypt these parts.

For conceptual information on encrypting and decrypting Simple Object Access Protocol (SOAP) message, see "XML encryption" on page 160.

Perform the following steps in the WebSphere Development Studio Client for iSeries to configure the parts of the SOAP message that must be decrypted:

1. Open the webservices.xml deployment descriptor for your Web services application in the Web Services Editor of the WebSphere Development Studio Client for iSeries. For more information, see "Configure your Web services application" on page 102.
2. Click the **Security Extensions** tab.
3. Expand the **Request Receiver Service Configuration Details —> Required Confidentiality** settings.
4. Select the parts of the message to decrypt that the client encrypts in the request sender. You can do this by clicking **Add** and selecting either **bodycontent** (the user data of the message) or **usernametoken** (the basic authentication information).
5. Save the file.

Next, perform the following steps in the Web Services Editor to configure the information that is needed to decrypt the required parts:

1. Click the **Binding Configurations** tab.
2. Expand the **Request Receiver Binding Configuration Details —> Encryption Information** settings.
3. Click **Edit** to view the encryption information. The following table describes the purpose for each of these selections. Some of these definitions are based on the XML-Signature Syntax and Processing specification

   (http://www.w3.org/TR/xmldsig-core).

Name	Purpose
Encryption name	Encryption name is the name of this encryption information entry. This is an alias for the entry.
Data encryption method algorithm	Data encryption method algorithms are designed for encrypting and decrypting data in fixed size, multiple octet blocks. This algorithm must be the same as the algorithm selected in the client request sender configuration.
Key encryption method algorithm	Key encryption method algorithms are public key encryption algorithms specified for encrypting and decrypting keys. This algorithm must be the same as the algorithm selected in the client request sender configuration.

Name	Purpose
Encryption key name	Encryption key name represents a Subject (from a certificate) found by the encryption key locator. the Subject is used by the key encryption method algorithm to decrypt the secret key, and the secret key is used to decrypt the data. **Note**: The key chosen here should be a private key in the keystore configured by the key locator. The key should have the same Subject used by the client to encrypt the data. Encryption must be done using the public key and decryption by the private key (personal certificate). To ensure that the client encrypts the data with the correct public or private key, you must extract the public key from the server's keystore and add it to the keystore specified in the client request sender encryption configuration information. For example, the personal certificate of a server is CN=Bob, O=IBM, C=US. Therefore the server contains the public and private key pair. The client sending the request should encrypt the data using the public key for CN=Bob, O=IBM, C=US. The server decrypts the data using the private key for CN=Bob, O=IBM, C=US.
Encryption key locator	This represents a reference to a key locator implementation. For more information on configuring key locators, see "Configure a key locator" on page 139.

4. Save the file.

It is very important to note that for decryption, the encryption key name chosen must refer to a personal certificate that can be located by the key locator of the server referenced in the encryption information. Enter the Subject of the personal certificate here, which is typically a Distinguished Name (DN). The Subject uses the default key locator to find the key. If a custom key locator is written, the encryption key name can be anything used by the key locator to find the correct encryption key. The encryption key locator references the implementation class that finds the correct key store where this alias and certificate exist.

Configure the Web services server for response encryption: This task provides the steps needed to configure the server for response encryption. Use these steps to modify the extensions to indicate which parts of the response you want to encrypt. Also, use these steps to configure the bindings to indicate how the parts of the response are to be encrypted.

Perform the following steps in the WebSphere Development Studio Client for iSeries to configure the parts of the Simple Object Access Protocol (SOAP) request that you want to encrypt:

1. Open the webservices.xml deployment descriptor for your Web services application in the Web Services Editor of the WebSphere Development Studio Client for iSeries. For more information, see "Configure your Web services application" on page 102.
2. Click the **Security Extensions** tab.
3. Expand **Request Sender Configuration —> Confidentiality**. *Confidentiality* refers to encryption while integrity refers to digital signing. Confidentiality reduces the risk of someone being able to understand the message flowing across the Internet. With confidentiality specifications, the message is encrypted before it is sent and decrypted when it is received at the correct target. For more information on encrypting , see "XML encryption" on page 160.
4. Select the parts of the response that you want to encrypt by clicking **Add** and selecting one of the following message parts:

- **Bodycontent**
  This is the user data portion of the message.
- **Usernametoken**
  This is an option that you can select. However, a user name token does not appear in the response. You do not need to select this option for the response. If you select this option, make sure that you also select it for the client response receiver. If you do not select it, make sure that you do not select it for the client response receiver either.

5. Save the file.

Next, perform the following steps in the Web Services Editor to configure the information that is needed to encrypt the response parts (bindings):

1. Click the **Binding Configurations** tab.
2. Expand **Response Sender Binding Configuration Details —> Encryption Information**.
3. Click **Edit** to view the encryption information. The following table describes the purpose of this information. Some of these definitions are based on the XML-Signature Syntax and Processing specification

   (http://www.w3.org/TR/xmldsig-core).

Name	Purpose
Encryption name	The encryption name refers to the name of the encryption information entry.
Data encryption method algorithm	The data encryption method algorithms are designed for encrypting and decrypting data in fixed size, multiple octet blocks. The algorithm selected for the server response sender configuration must match the algorithm selected in the client response receiver configuration.
Key encryption method algorithm	The key encryption method algorithms are public key encryption algorithms that are specified for encrypting and decrypting keys. The algorithm selected for the server response sender configuration must match the algorithm selected in the client response receiver configuration.
Encryption key name	The encryption key name represents a Subject from a certificate found by the encryption key locator, which is used by the key encryption method algorithm to encrypt the private key. The private key is used to encrypt the data. **Note:** The key name chosen in the server response sender encryption information must be the public key of the key configured in the client response receiver encryption information. Encryption by the response sender must be done using the public key and decryption must be done by the response receiver using the associated private key (the personal certificate of the response receiver).
Encryption key locator	The encryption key locator represents a reference to a key locator implementation. For more information on configuring key locators, see “Configure a key locator” on page 139.

4. Save the file.

The encryption key name chosen must refer to a public key of the response receiver. For the encryption key name, use the Subject of the public key certificate, typically a Distinguished Name (DN). The name chosen is used by the default key locator to find the key. If you write a custom key locator, the encryption key name may be anything used by the key locator to find the correct encryption key (a public key). The encryption key locator references the implementation class that finds the correct key store where the alias and certificate exist.

Configure client-side SSL for Web services

Transport level security is based on Secured Sockets Layer (SSL) or Transport Layer Security (TLS) that runs beneath the HTTP protocol. Both provide security features including authentication, data protection, and cryptographic token support for secure HTTP connections. To run with HTTPS, the service endpoint address must be in the form of `https://`.

The integrity and confidentiality of transport data, including Simple Object Access Protocol (SOAP) messages and HTTP basic authentication, is confirmed when you use SSL and TLS. WebSphere Application Server - Express uses Java Secure Sockets Extension (JSSE) to support SSL and TLS.

The server-side, or service endpoint, transport level security is based on the Secured Sockets Layer (SSL) configuration of the WebSphere Application Server - Express Web container. See Configure SSL in WebSphere Application Server - Express in the *Security* topic for more information.

To configure the client-side transport level security, perform the following steps:

1. Create an SSL reperoire configuration entry for an existing service endpoint that acts as a service client. For more information, see Use SSL configuration repertoires in the *Security* topic.
2. Define the attribute `sslConfig` with the value of the alias name in the ibm-webservicesclient-bnd.xmi file. For example:

   ```
   <sslConfig name="default/DefaultSSLSettings"/>
   ```

   Note: If the attribute is not defined, the default SSL setting is used for JSSE.
3. Set the system property, com.ibm.webservices.sslConfigURL, to the property file. For example:

   ```
   Dcom.ibm.webservices.sslConfigURL=${USER_INSTALL_ROOT}/properties/sas.client.props
   ```

   Note: If the property `sslConfigURL` is not defined, the default SSL setting is used for JSSE.
4. (Optional) Set the system properties of an unmanaged service client by using the `-D` option of the Java command. Alternatively, you can call the System.setProperty (*propertyName*, *"propertyValue"*) method, where *propertyName* is the name of a property, and *propertyValue* is the value of the property.

   Using either method, set values for the following properties:
   - `java.protocol.handler.pkgs`
   - `javax.net.ssl.keyStore`
   - `javax.net.ssl.keyStorePassword`
   - `javax.net.ssl.trustStore`
   - `javax.net.ssl.trustStorePassword`
5. (Optional) Redirect the Simple Object Access Protocol (SOAP) request from a client to service endpoint to be over HTTPS. Complete this step if a transport guarantee of `CONFIDENTIAL` or `INTEGRAL` is configured for a secured Web application.

   To redirect the request set the system property `com.ibm.ws.webservices.HttpRedirectEnabled` to `true` for the entire Java virtual machine.

   Alternatively, you can set the property `com.ibm.wsspi.webservices.Constants.HTTP_REDIRECT_ENABLED`, to `true` in the stub or call instance, before the method is invoked.
6. After you have deployed your application, "Edit the HTTP basic authentication and SSL configuration for Web services" on page 138 with the WebSphere administrative console.

Web services resources

Use the following links to find relevant supplemental information about getting started with Web services. The information resides on IBM and non-IBM Internet sites, whose sponsors control the technical accuracy of the information.

These links are provided for convenience. Often, the information is not specific to the IBM WebSphere Application Server product, but is useful all or in part for understanding the product. When possible, links are provided to technical papers and Redbooks that supplement the broad coverage of the release documentation with in-depth examinations of particular product areas. The following sections are covered in this reference:

- Web services overview: Purpose, planning and designing to use Web services (page 170)
- Developing Web services Java API for XML-based remote procedure call (JAX-RPC) and the J2EE programming model (page 170)
- Security (page 171)
- Administration (page 173)
- Other references (page 173)

Web services overview: Purpose, planning and designing to use Web services

- **IBM Web Services architecture debuts**

  (http://www.ibm.com/developerworks/webservices/library/w-int.html?dwzone=webservices)

  Introducing IBM Web services, a distributed software architecture of service components. This brief overview and in-depth interview on IBM DeveloperWorks cover the fundamental concepts of Web services architecture and what they mean for developers. The interview with IBM professional Rod Smith explores which types of developers Web services targets, how Web services reduces development time, what developers could be doing with Web services now, and takes a glance at the economics of dynamically discoverable services.
- **Web services (r)evolution, Part 1**

  (http://www-106.ibm.com/developerworks/library/ws-peer1.html)

  This article focuses on the benefits and challenges of building Web services applications. Web services might be an evolutionary step in designing distributed applications, however, they are not without their problems. Outlined are the difficulties developers face in creating a truly workable distributed system of Web services. This article also outlines author Grahm Glass' plan for building peer-to-peer Web applications.

Developing Web services

- **JSR 109: Implementing Enterprise Web Services**

  (http://jcp.org/en/jsr/detail?id=109)

  This document describes the J2EE specification model.
- **Java API for XML-based RPC (JAX-RPC): Core Web Services API in the Java platform**

  (http://java.sun.com/xml/jaxrpc/)

  This document reviews the JAX-RPC which enables Java technology developers to develop SOAP based interoperable and portable Web services.

- **SOAP**

  (http://www.w3.org/TR/SOAP)

  This article is a detailed overview of SOAP, which includes programming specifications.
- **Web Services Description Language**

  (http://www.w3.org/TR/wsdl)

  This article is a detailed overview of Web Services Description Language (WSDL), which includes programming specifications.
- **Universal Description, Discovery and Integration**

  (http://www.uddi.org/about.html)

  This article is a detailed overview of Universal Description, Discovery and Integration (UDDI).
- **UDDI4J: Matchmaking for Web services**

  (http://www-106.ibm.com/developerworks/library/ws-uddi4j)

  Reviewed in this article are the basics of UDDI, the Java API to UDDI, and how you can use this technology to start building, testing, and deploying your own Web services.

Security

- **Security in a Web Services World: A Proposed Architecture and Roadmap**

  (http://www-106.ibm.com/developerworks/webservices/library/ws-secmap/)

  This document describes a proposed model for addressing security within a Web service environment. It defines a comprehensive Web Services Security model that supports, integrates, and unifies several popular security models, mechanisms, and technologies, including both symmetric and public key technologies, in a way that enables a variety of systems to securely interoperate in a platform and language-neutral manner. It also describes a set of specifications and scenarios that show how these specifications can be used together.
- **Web Services Security (WS-Security)**

  (http://www-106.ibm.com/developerworks/library/ws-secure/)

  The Web Services Security specifications describe enhancements to SOAP messaging to provide quality of protection through message integrity, message confidentiality, and single message authentication. These mechanisms can be used to accommodate a wide variety of security models and encryption technologies. Web Services Security also provides a general-purpose mechanism for associating security tokens with messages. Additionally, Web Services Security describes how to encode binary security tokens. Specifically, the specification describes how to encode X.509 certificates and Kerberos tickets, as well as how to include opaque encrypted keys. It also includes extensibility mechanisms that can be used to further describe the characteristics of the credentials that are included with a message.
- **Web Services Security Addendum**

  (http://www-106.ibm.com/developerworks/library/ws-secureadd.html)

This document describes clarifications, enhancements, best practices, and errata of the Web Services Security specification.

- **WS-Security Profile of the OASIS Security Assertion Markup Language (SAML) Working Draft 04, 10 September 2002**

  (http://www.oasis-open.org/committees/security/docs/draft-sstc-ws-sec-profile-04.pdf)

  This document proposes a set of standards for SOAP extentions used to increase message confidentiality.
- **Web Services Security: Soap Message Security Working Draft 12, Monday 21 April 2003**

  (http://www.oasis-open.org/committees/download.php/1686/WSS-SOAPMessageSecurity-12-04021.pdf)

  This document describes the support for multiple token formats, trust domains, signature formats, and encyrption technologies.
- **JSR 55:Certification Path API**

  (http://jcp.org/en/jsr/detail?id=55)

  This document provides a short description of the certification path API.
- **XML-Signature Syntax and Processing**

  (http://www.w3.org/TR/xmldsig-core/)

  This document specifies XML digital signature processing rules and syntax. XML signatures provide integrity, message authentication, or signer authentication services for data of any type, whether located within the XML that includes the signature or elsewhere.
- **Canonical XML Version 1.0**

  (http://www.w3.org/TR/xml-c14n)

  This specification describes a method for generating a physical representation, the canonical form, of an XML document that accounts for the permissible changes.
- **Exclusive XML Canonicalization Version 1.0**

  (http://www.w3.org/TR/xml-exc-c14n/)

  Canonical XML [XML-C14N] specifies a standard serialization of XML that, when applied to a subdocument, includes the subdocument's ancestor context including all of the namespace declarations and attributes in the ″xml:″namespace.
- **XML Encryption Syntax and Processing**

  (http://www.w3.org/TR/xmlenc-core/)

  This document specifies a process for encrypting data and representing the result in XML.
- **Decryption Transform for XML Signature**

(http://www.w3.org/TR/xmlenc-decrypt)

This document specifies an XML Signature "decryption transform" that enables XML Signature applications to distinguish between those XML Encryption structures that were encrypted before signing, and must not be decrypted, and those that were encrypted after signing, and must be decrypted, for the signature to validate.

- **WS-Security**

  (http://schemas.xmlsoap.org/ws/2002/04/secext/)

  This document specifies resources for the April 2002 Web Services Security Specification. The following addendums and drafts are available:

  - **http://schemas.xmlsoap.org/ws/2002/07/secext/**

    (http://schemas.xmlsoap.org/ws/2002/07/secext/)
    (http://schemas.xmlsoap.org/ws/2002/07/utility/)

  - **OASIS draft 12 for secext**

    (http://schemas.xmlsoap.org/ws/2003/06/secext/)

  - **OASIS draft 12 for utility**

    (http://schemas.xmlsoap.org/ws/2003/06/utility/)

Administration

- **SOAP Security Extensions: Digital Signature**

  (http://www.w3.org/TR/SOAP-dsig)

  This document specifies the syntax and processing rules of a SOAP header entry to carry digital signature information within a SOAP 1.1 Envelope

- **Apache Software Foundation**

  (http://www.apache.org)

Other references

- **Web services insider, Part 1: Reflections on SOAP**

  (http://www-106.ibm.com/developerworks/webservices/library/ws-ref1)

  What is the current state of the *Web services revolution*? Find out at this Web site that features the column *Web services insider, Part 1*. The author answers this question by reviewing the tools and technologies that have emerged over the past year, highlighting their differences and similarities.

- **The Web services insider, Part 2: A summary of the W3C Web Services Workshop**

  (http://www-106.ibm.com/developerworks/webservices/library/ws-ref2)

This is a brief summary of a W3C Web services workshop.

Appendix. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

```
IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY  10504-1785
U.S.A.
```

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

```
IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan
```

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION ″AS IS″ WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Software Interoperability Coordinator, Department YBWA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, IBM License Agreement for Machine Code, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming Interface Information

This WebSphere Application Server - Express publication documents intended Programming Interfaces that allow the customer to write programs to obtain the services of IBM i5/OS.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

```
AIX
AIX 5L
e(logo)server
eServer
i5/OS
IBM
IBM (logo)
iSeries
pSeries
WebSphere
xSeries
zSeries
```

Intel, Intel Inside (logos), MMX, and Pentium are trademarks of Intel Corporation in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

Terms and conditions

Permissions for the use of these publications is granted subject to the following terms and conditions.

Personal Use: You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative works of these publications, or any portion thereof, without the express consent of IBM.

Commercial Use: You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the p <?Pub Caret?>ublications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED ″AS-IS″ AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

Code license and disclaimer information

IBM grants you a nonexclusive copyright license to use all programming code examples from which you can generate similar function tailored to your own specific needs.

SUBJECT TO ANY STATUTORY WARRANTIES WHICH CANNOT BE EXCLUDED, IBM, ITS PROGRAM DEVELOPERS AND SUPPLIERS MAKE NO WARRANTIES OR CONDITIONS EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT, REGARDING THE PROGRAM OR TECHNICAL SUPPORT, IF ANY.

UNDER NO CIRCUMSTANCES IS IBM, ITS PROGRAM DEVELOPERS OR SUPPLIERS LIABLE FOR ANY OF THE FOLLOWING, EVEN IF INFORMED OF THEIR POSSIBILITY:

1. LOSS OF, OR DAMAGE TO, DATA;
2. DIRECT, SPECIAL, INCIDENTAL, OR INDIRECT DAMAGES, OR FOR ANY ECONOMIC CONSEQUENTIAL DAMAGES; OR
3. LOST PROFITS, BUSINESS, REVENUE, GOODWILL, OR ANTICIP <?Pub Caret?>ATED SAVINGS.

SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF DIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, SO SOME OR ALL OF THE ABOVE LIMITATIONS OR EXCLUSIONS MAY NOT APPLY TO YOU.

Printed in USA