WebSphere

Web Services Application Developer's Guide

Third Edition (March 2004)

Contents

Chapter 1. Introduction

You can use the Web Services client plug-in with WebSphere® Studio Device Developer (WSDD) to develop applications that consume and expose Web Services.

WSDD provides an integrated development environment (IDE) that you can use to create and test applications that you want to deploy on devices, such as cellular phones, Personal Data Assistants (PDA), and other pervasive devices. WSDD extends the WebSphere product suite by enabling you to build applications based on J2ME™ profiles and use configurations for running on "Java™ Powered™" devices. For more information on WSDD, visit http://www.ibm.com/pvc. In addition, WSDD provides support for building applications that use the IBM WebSphere Everyplace Custom Environment (WECE) class libraries and the OSGi Service Management Framework.

The Web Services support is an implementation of the Java 2 Micro Edition Web Services Specification (JSR-172). Further details on this specification can be found at the following site: http://jcp.org/jsr/detail/172.jsp.

To enable you to access remote web services, the Web Services for MIDP plug-in implements the features detailed in the specification. The Web Services for ESWE plug-in also allows you to consume web services, and to expose OSGi based bundles as Web Services.

An application that consumes Web Services needs to identify the service end-point and use the interface to that Web Service with the definition specified in the Web Services Description Language (WSDL) document.

Web Services provides support for the following Platform Profiles when creating Web Services consumers:

- WebSphere Everyplace Micro Environment (WEME) Foundation and Mobile Information Device Profile (MIDP)
- WebSphere Everyplace Custom Environment (WECE) Max and RM

Web Services provides support for the following Platform Profiles when creating Web Services providers:

- WebSphere Everyplace Micro Environment (WEME) Foundation on SMF
- WebSphere Everyplace Custom Environment (WECE) Max and RM on SMF

Chapter 2. Getting Started

This chapter discusses the necessary steps to enable you to start developing applications that consume Web Services.

Planning for the target execution environment

Using Web Services, you can develop Web Services consumer applications for a variety of target environments. Visit the following URL for complete information on the platforms supported for WebSphere Everyplace Custom Environment (WECE) and WebSphere Everyplace Micro Environment (WEME): http://www-3.ibm.com/software/wireless/wsdd/features.html

You can also develop Web Services and consumers that run on SMF.

WEME and WECE Web Services clients use a stub to interface with the Web Services runtime. However, for SMF you have the option of using a statically generated stub and the Web Services runtime, or you can use the Web Services runtime bundle instead, which dynamically creates a stub when the service is accessed.

Because Web Services clients communicate with Web Services using Hyper Text Transfer Protocol (HTTP), you need an Internet connection to access Web Services.

Prerequisites

Web Services requires the following software:

- WebSphere Studio Device Developer 5.7, which includes the SMF runtime
- Service Management Framework Bundle Development Kit 5.7
- WebSphere Everyplace Micro Environment Foundation class library 5.7
- WebSphere Everyplace Custom Environment Max class library 5.7
- WebSphere Everyplace Custom Environment RM Class library 5.7

Note: Only your target runtimes need to be installed in order to develop your application.

Chapter 3. Concepts

This chapter helps you understand Web Services client plug-in concepts.

Web Services

Web Services are applications that exist in a distributed environment, such as the Internet. Web Services accept a request, perform a function based on the request, and return a response. The request and the response can be part of the same operation, or they can occur separately, in which case the consumer does not need to wait for a response. Both the request and the response usually take the form of XML, a portable data-interchange format, and are delivered over a wire protocol, such as HTTP.

A unique attribute of Web Services is that their implementation details, such as programming language used to develop the service, are hidden from the clients that consume the service. In addition, Web services publish their public interfaces in a standardized method, using the Web Services Description Language (WSDL).

Web Services transactions are usually conducted between businesses or between businesses and consumers. A business that is a provider of one service can also be a consumer of another service. A Web Services consumer can be a client device, such as a thin client connecting to the Web Services provider using a lightweight protocol.

Web Services provides the following components that you can use to develop Web Services client and server applications:

- Web Services client plug-in that enables you to generate an application stub or a Java Interface file
- Runtime support for the development and target environments

Web Services installation options

When installing the Web Services support, you have the option of installing either the MIDP, the ESWE component, or both.

When you install Web Services support for MIDP, you are enabled with the capability to develop Web Services clients that use statically generated stubs using the Web Services wizard. You also have the option of selecting the WS-Security configuration for the subject service.

When you install Web Services support for ESWE, you are enabled with the following application options:

- Select **Mobile Web Service Client** to develop Web Services consumer applications that run as WEME, WECE applications or OSGi bundles, with statically generated stubs that use jclFoundation, jclMax, or jclRM class libraries.
- Select **Mobile Web Service Client for Extension Services** to develop Web Services consumer applications that run only as OSGi bundles with dynamically generated stubs that use jclFoundation, jclMax, or jclRM class libraries.

Web Services plug-in

The Web Services plug-in is a WebSphere Studio Device Developer (WSDD) plug-in that you can use to generate an application stub or an Interface file, which you include with your Java application.

You use the Web Services plug-in wizard to generate a Java stub inside an existing WSDD project. The wizard takes a WSDL document and generates a Java class (stub) that a program can call. The stub handles the Web Services request automatically by interfacing with the Web Services runtime and returns the result to the calling program.

If you are developing MIDP based Web services clients, then you must install the Web Services for MIDP feature.

If you are developing ESWE based Web services client or server applications, then you must install the Web Services for ESWE feature.

You can install both features as needed.

The Web Services plug-in contains:

- Web Services Stub generator for several application environments
- This guide
- Sample programs

Runtime support for the target environment

Web Services provides runtime support that enables you to run a Web Services consumer on the following target platforms:

- Pocket PC
- Microsoft® Windows® 2000
- Linux

Web Services includes the following runtime support for each of the supported target platforms: MIDP/CLDC, WebSphere Everyplace Custom Environment (WECE), and the OSGi Service Management Framework (SMF) environment. The runtime support is a collection of Java packages that you must install in the target environment. The packages provided in each runtime environment implement interfaces specified in the Java API for XML-based Remote Procedure Call (JAX-RPC) subset and the Java API for XML-based Parser (JAXP) subset as specified in the JSR-172 Specification. You can download the specification from www.jcp.org.

Note: The generated Java stub uses the interfaces specified by JSR-172. In Web Services, the stub and the interfaces that JAX-RPC uses are exposed to application developers. However, for consistent use of the interfaces, it is recommended that all applications use the automatically generated files, rather than directly using the JAX-RPC interfaces.

This release includes Web Services security (WS-Security) functions for use by Web Services clients in the supported platforms.

Support for OSGi Services

Web Services allows OSGi bundles to consume Web Services via a Web Services Gateway, and allows OSGi bundles to expose their interfaces as Web Services.

The limitations imposed by JSR-172 for J2ME Web Services are also imposed for SMF Web Services clients.

The Web Services Gateway dynamically creates a Web services stub for an OSGi Web services consumer. This dynamically generated stub is used by your OSGi application to interface with the Web Services runtime, and indirectly with the actual web service, regardless of where the service is running (i.e., local or remote).

The Web Services Gateway bundle is also capable of exposing your application as a Web Service by registering your application bundle with the Web Services Gateway bundle.

Web Services security

Web Services security (WS-Security) provides message confidentiality and integrity for commercial and enterprise customers. For example, as a commercial or enterprise customer, you might use Web Services security for financial transactions from wireless phones.

WS-Security uses the following encryption and key management technologies:

- Asymmetric encryption

  WS-Security uses RSA encryption to encrypt the symmetric key. The receiver provides a private key, and the sender encrypts the symmetric key with the public key. In most cases, the public key resides in a digital certificate.
- Symmetric encryption

  WS-Security uses DSA encryption to encrypt data contained in a SOAP message. Web services security is available only for Web services client applications.

Web Services security specifications

WS-Security is a standards-based architecture. The standards are a set of WS-Security specifications that address how to provide protection for messages that Web Services consumers exchange in a Web Services environment. WS-Security defines the core facilities that protect the integrity and confidentiality of a message and provides mechanisms for associating security-related claims with the message. To secure Web Services, you must consider a broad set of security requirements including authentication, authorization, privacy, trust, integrity, confidentiality, secure communications channels, federation, and delegation.

Web Services security architecture

The Web Services client wizard enables you to generate a client stub from a WSDL file. If you choose to configure security, the wizard creates a WS-Security configuration class. Refer to the following steps to understand how the Web Services runtime and the WS-Security runtime communicate securely when using static stubs.

1. When your application calls the Web Service, the Web Services runtime then calls the WS-Security configuration class, which defines the WS-Security configuration to the WS-Security runtime.
2. The Web Services runtime then constructs a SOAP message and calls the WS-Security runtime.
3. The WS-Security runtime processes the SOAP message according to the WS-Security configuration information.
4. The Web Services runtime then sends the SOAP message to the Web Services provider.

5. When the Web Services runtime receives a response to the SOAP message, the WS-Security runtime processes the security information in the SOAP message. For example, the WS-Security runtime verifies the digital signature and decrypts the data accordingly.
6. After the WS-Security runtime finishes processing the security information in the SOAP message, the Web Services runtime finishes processing the message and returns the results to your application.

Web Services security features

Web Services security (WS-Security) provides a set of mechanisms that you can use to secure Simple Object Access Protocol (SOAP) message exchanges. Specifically, WS-Security enhances SOAP messages by providing quality protection through the following features:

- "Message integrity" with "XML digital signatures" on page 10
- "Message confidentiality" on page 9 with "XML encryption and decryption" on page 10
- "Single message authentication" on page 9 with "Security tokens" on page 10

You can combine these functions in different ways to build security models with different cryptographic technologies.

The Web Services security support for WebSphere Studio Device Developer (WSDD) is based on the WS-Security Minimalist Profile specification maintained by the OASIS technical committee for Web Services security.

Message integrity

WS-Security provides Simple Object Access Protocol (SOAP) message integrity through "XML digital signatures" on page 10 that associate a key with data. The following steps describe how senders generate and receivers verify XML digital signatures:

1. The WS-Security runtime generates signatures by processing the elements in the SOAP messages to produce Digests.

   Note: The Digests reside in an XML element called `SignedInfo`.
2. The sender digests the data in the SignedInfo element and cryptographically signs the element to produce a signature.
3. WS-Security uses the private key of the requester to cryptographically sign the element.
4. When the receiver receives the signed SOAP message, the receiver verifies the Digests in the `SignedInfo` element by recalculating the digests of the elements in the SOAP message.
5. The receiver verifies the signature against the sender's public key.

   Because the SOAP message sender signs the message with a private key and the receiver verifies the signature using the public key, the receiver determines that only the private key holder could have signed the message.

The WS-Security support implements XML digital signatures with the Rivest, Shamir, Adleman (RSA) algorithm with Secure Hash Algorithm 1 (SHA1) and Digital Signature Algorithm (DSA) with SHA1. To send a digitally signed message, you must provide a keystore that contains an RSA or DSA private key and information to access the key. When the receiver receives a SOAP message, the WS-Security support uses a public key to verify the signature. Because the received message contains the information needed to reference the public key, you do not

need to provide configuration information to verify the signature of the received messages. For instructions to create a keystore, refer to “Creating a keystore” on page 22.

Note: The WS-Security support in Web Services does not permit symmetric key signature algorithms.

Message confidentiality

WS-Security protects message confidentiality by using XML encryption to encrypt the data in the message and represent the result in XML. The WS-Security support uses asymmetric encryption, also known as key exchange. The following steps describe how senders encrypt the information in the message and receivers decrypt the information:

1. During symmetric encryption, the sender generates a secret key that encrypts the data in the message.
2. The sender encrypts the secret key with a public key that the receiver can access.
3. The sender sends the secret key in the SOAP message to the receiver.
4. When the receiver receives the SOAP message, the receiver uses the public key to decrypt the secret key that was sent with the message.

The WS-Security support in Web Services for WSDD uses RSA for key encryption and decryption. To send an encrypted message, you must provide a keystore with the receiver certificate and information to access the certificate. The certificate contains the public key for the receiver. To decrypt the message you receive, you must provide a keystore that contains the private key and information to access the key. For instructions to create a keystore, refer to “Creating a keystore” on page 22.

Note: The WS-Security support in Web Services for WSDD does not support symmetric key encryption and symmetric message encryption.

Single message authentication

WS-Security supports single message authentication by enabling you to send authentication information in SOAP messages. Specifically, the authentication information resides in a security token in the SOAP message. Security tokens can be X.509 certificates, or a user name and password. The WS-Security support for WebSphere Studio Device Developer (WSDD) permits Basic authentication with user name and password and signature authentication with an X509-v3 certificate.

Web Services security supported functions

The WS-Security support in Web Services for WSDD is an implementation of the SOAP Message Security Minimalist Profile (MProf) specification. However, the WS-Security support differs from the MProf specification in the following ways:

- To support XML streaming, MProf prohibits backward references for signatures. Because Web Services is not a streaming-based architecture, it is not necessary to restrict the location of targeted message parts. As such, the WS-Security support permits you to use backward references for signatures by intra-document URIs.
- MProf assumes that the sender always canonicalizes SOAP messages. However, no servers exist that always canonicalize all of the targeted message parts for signatures. As such, the WS-Security support enables you to canonicalize SOAP messages for the messages you send and receive.
- WS-Security does not support MProf awareness.
- WS-Security supports only intra-document URIs for signatures. Enveloped signatures and SOAP attachment signatures are not supported.

XML digital signatures

The WS-Security support in Web Services for WebSphere Studio Device Developer (WSDD) supports the following algorithms and transforms:

- Digest method algorithm including the US Secure Hash Algorithm 1 (SHA1)
- Signature method algorithm
  - RSA with SHA1
  - DSA with SHA1
- Canonicalization method algorithm including the Exclusive XML Canonicalization (xml-exc-c14n)

  Note: XSLT, XPath, and enveloped signatures are not supported.

XML encryption and decryption

The WS-Security support in Web Services for WebSphere Studio Device Developer (WSDD) supports the following algorithms:

- RSA-v1.5 for key transport
- TripleDES for block encryption

Security tokens

The WS-Security support for WebSphere Studio Device Developer (WSDD) supports the following security tokens:

- X509-v3 as binary security tokens
- User name token

Chapter 4. Tasks

This chapter provides instructions for creating Web Services clients.

Creating a Web Services client for J2ME

To develop a Web Services client for Java 2 Micro Edition (J2ME), complete the tasks described in the following sections.

Creating a J2ME MIDP project

Before you can use the wizard to generate the client stub and sample MIDlet, refer to the following steps to create a Java 2 Micro Edition (J2ME) Mobile Information Device Profile (MIDP) project with the JCL MIDP library:

1. Select **File** -> **New** -> **Project**.
2. In the left frame, select **J2ME**. In the right frame, select **J2ME Project**. Click **Next**.
3. Specify a name for your new project. Click **Next**.
4. Select **WEME jclMidp** as the class library, and click **Finish**.

Generating a Java stub and sample MIDlet client

After you finish creating your Java 2 Micro Edition, Mobile Information Device Profile (MIDP) project, refer to the following steps to create a Java stub and MIDlet client:

1. Select **File** -> **New** -> **Other**.
2. In the left frame, select **Mobile Web Services Client**. In the right frame, select **Mobile Web Services Client for MIDP**. Click **Next**.

   The **Web Services Client Stub Generator** is displayed.
3. Specify the following information on the **Web Service Client Stub Generator** dialog:
   - Specify the source folder where you want the stub you generate and the sample MIDlet files to reside.

     Note: In most cases, select the `src` folder you created in "Creating a J2ME MIDP project."
   - Specify the name of the Java package where you want the code you generate to reside. For example, you might name your package `com.mysample.webservice`.

     Note: This is an optional field, and it is suggested that you leave this field blank.
   - Specify the address of the Web Services Description Language (WSDL) document that describes the Web Service for which you want to generate the stub.
   - Select the **Generate Midlet** checkbox if you want the wizard to create a sample midlet.
   - Select the **Configure security** checkbox if you want to enable security.

Creating a MIDlet suite

Refer to the WebSphere Studio Device Developer documentation for instructions to create a MIDlet suite that you can deploy to your device or emulator. The MIDlet suite enables you to execute the sample MIDlet.

Creating a MIDlet suite configuration

Before you can run your MIDlet suite on the MIDP emulator, refer to the WebSphere Studio Device Developer documentation for instructions to create a MIDlet suite configuration.

Running the client in the MIDP emulator

You can use the build you created in "Creating a MIDlet suite configuration" to launch the MIDlet suite in the MIDP emulator. Refer to the following instructions to launch your sample Web Services MIDlet on the WebSphere Studio Device Developer (WSDD) MIDP emulator:

1. On the main menu, select **Run** -> **Run** to display the **Run Configuration** dialog.
2. In the left frame, select **MIDlet suite** and click **New**.

   The MIDP run configuration editor is displayed.
3. Specify the name you want to assign to this configuration.
4. Verify that the **Project** field lists the name of your project.

   If the **Project** field does not list your project, click **Browse** and navigate to your project.
5. Verify that the **MIDlet suite** field lists the name of your JAD file.
6. Click **Run**.

   The **MIDP Emulator** window is displayed. You can use the Emulator to run the application you created.

Running the client on a PocketPC device

You can use the build you created in "Creating a MIDlet suite configuration" to launch the MIDlet suite on a PocketPC device. Before you can launch the MIDlet suite on a PocketPC device, verify that the PocketPC device connects to the host machine with ActiveSync and install the J9 Java Virtual Machine (JVM) on the device using the instructions in the WebSphere Studio Device Developer (WSDD) documentation.

Refer to the following instructions to launch your sample Web Services MIDlet on a PocketPC device:

1. On the main menu, select **Run** -> **Run** to display the **Run Configuration** dialog.
2. In the left frame, select **MIDlet Suite** and click **New**.

   The MIDP run configuration editor is displayed.
3. Specify the name you want to assign to this configuration.
4. Verify that the **Project** field lists the name of your project.

   If the **Project** field does not list your project, click **Browse** and navigate to your project.
5. Verify that the your PocketPC device is selected in the **Device** or **JRE** field.

   If your PocketPC device is not listed, complete the following steps:

   a. Click **Configure**.

b. Select **PocketPC Handheld** and click **New**.
c. Click **Browse** next to **J9 runtime location** to select the directory where the J9 resides.
d. Click **Browse** next to **Application install location** to select the directory where you want your application to reside.
e. Click **Browse** next to **Shortcut install location** to select the directory where you want the shortcut to reside.
f. Click **OK**.

6. Verify that the **MIDlet suite** field lists the name of your JAD file.
7. Click **Run**.

   WSDD transfers your build to the PocketPC device under the directory you selected in the **Application install location** field. Your application executes automatically after WSDD transfers the application to the device.

   Note: If your application does not execute successfully on the device, refer to "Enabling MIDP applications to run" on page 33 for additional information.

Creating a Mobile Web Services client

You can develop a Mobile Web Services client application that uses statically generated stubs that use the WECE environment jclMax library.

Setting up a WECE project for use with SMF

Set up a project into which to add a generated Web Services client stub by performing the following procedure:

1. From the WSDD toolbar, click **Open New Wizard**. The **New** dialog is displayed.
2. In the left frame, select **WECE** for J9. In the right frame, select **Create WECE Project**. Then click **Next**. The **New Project** dialog is displayed.
3. Type a name for your new project, for example: `My_Web_Service_Project`. Click **Next**.
4. Select the **WECE jclMax** class library configuration.
5. Click **Finish** to create the WECE project.
6. In the WSDD, expand the + sign to the left of `My_Web_Service_Project` to view the following files and folders:
   - **src**: Your Java code, including the stub should reside in this directory.
   - **VEHOME/lib/Max/***: The required libraries that are supplied with Max. The ″jar″ icon signifies that these libraries are in the Java build path (and not in your project).
   - **wsddbuild.xml**: This is the Ant script file that defines builds and launches. This file will be used in subsequent sections.

Creating a Web Services stub and interface file

You must add Web Services to an existing project. To do so, create a Web Services stub and interface file by performing the following procedure:

1. Select `My_Web_Service_Project` in the package explorer. Click **Open New Wizard** in the workbench toolbar, then select **Other**. The Web Services wizard is under the **Other** category. The **New** dialog is displayed.

2. In the left frame of the **New** dialog, select **Mobile Web Services Client**. In the right frame, select **Web Services Client**. Then click **Next**. The **Stub Generator** dialog is displayed.
3. Enter the following information into the Stub Generator Dialog:
   - Browse to the source directory of the project you want the stub to be placed. This example uses **My_Web_Service_Project**.
   - The Package is the name of the Java package in which you want the stub to be created, for example, mysample.webservice.com.
   - WSDL Location is the URL of the WSDL document that describes the Web Services for which you want to generate a stub. This example uses the a Web Services from www.acrosscommunications.com.
4. If the Web Service requires security, then select the **Configure Security** checkbox.

   If you choose to enable security, click **Next** to complete the security information on the dialog windows. Refer to "Creating a Web Services application that includes WS-Security" on page 21 for instructions to complete the security information.

   If you choose not to enable security, click **Finish**.

   The wizard generates the Java classes in the package you specified in the wizard. The class names reflect the binding you specified in the Web Services Description Language (WSDL) document.
5. Click **Finish**.

After you have created the stub and interface file, you need to create the rest of your WECE-based application to interface with the generated stub. Please refer to the help document for building ad running a WECE application for more information.

A Java class is created with the binding name taken from the WSDL document.

Creating a Web Services client for Extension Services (ESWE)

A Web Services client for ESWE enables SMF bundles to consume Web Services using a Web Services Gateway bundle that dynamically creates a Web Services stub. To create a Web Services client running as an SMF bundle, complete the tasks in this section. After you have completed these steps, see "Using Web Services with SMF" on page 16 for more information.

SMF is the IBM® implementation of the OSGi Alliance standard. SMF uses a single Java Virtual Machine (JVM) instance to run multiple applications. These applications are delivered to a device over a network as bundles. SMF Bundle Developer is a component of SMF that enables you to package bundles and create manifest files that contain the application's information.

Creating an SMF bundle

The following steps describe how to develop an example bundle called My_Web_Service_Bundle:

1. Create a new project for My_Web_Service_Bundle.

   When you use WSDD to create bundles, your bundle is contained in a WSDD project. The project stores the environment and all of the necessary files to build an SMF bundle. Use the following steps to create the project for the Web Services bundle.

a. Click **File** -> **New** -> **Project**. Select **Java** -> **Java Project**. Click **Next**.
b. Type a name for the new project, for example: `My_Web_Service_Bundle`. Click **Next**.
c. Click **Source**. Select **Use the project as source folder**.
d. Click **Finish**.

2. Create a new Bundle Folder for the My_Web_Service_Bundle project.

   SMF provides tools and files that can assist you in developing SMF bundles. In this example we will use the Manifest Editor and the Submit Bundle tools. To begin, create a Bundle Folder for the My_Web_Service_Bundle project with the following steps.

   a. Right-click the **My_Web_Service_Bundle** project in the Java perspective. Select **New** -> **Other**.
   b. Select **SMF** -> **Bundle Folder**. Click **Next**.
   c. Select the /My_Web_Service_Bundle bundle folder container. Click **Finish**.

3. Create a package for the bundle with the following steps.

   a. Right-click the `My_Web_Service_Bundle` project in the Java perspective.
   b. Select **New** ->**Package**.
   c. Name the package, for example `com.example.smf.web.service`, in the **Package** field.
   d. Click **Finish**.

4. Create the BundleActivator for the test service bundle with the following steps.

   a. Right-click the **My_Web_Service_Bundle** project in the Java perspective.
   b. Select **New** -> **Class**.
   c. Type `com.example.smf.web.service` in the **Package** field.
   d. Type `MyBundleActivator` in the **Name** field and click **Finish**.
   e. Type the source for `com/ibm/osg/example/My_Web_Service_Bundle/MyBundleActivator.java`. Refer to the code example in ″ Registering and unregistering a service with the OSGi Framework″ in the Service Management Framework Bundle Developer Tools, 5.5.2 User's Guide.
   f. Click **Save** to compile the class.

5. Modify the manifest file for the My_Web_Service_Bundle project.

   The Manifest Editor helps you to prevent errors when you create a manifest file for a bundle, and provides a user interface that you can use to create a manifest file. To use the Manifest Editor, use the following steps.

   a. In the My_Web_Service_Bundle project, double-click the META-INF folder and then double-click the `MANIFEST.MF` file. The GUI for the manifest editor is displayed.

      If the manifest editor does not display, close the `MANIFEST.MF` file and right-click the `MANIFEST.MF` file from the My_Web_Service_Bundle project. Select **Open With** -> **Bundle Manifest Editor**.

   b. Type `com.example.smf.web.service.MyBundleActivator` in the **Bundle-Activator** field.

      Alternatively, you can use the pull-down menu in the Bundle-Activator field to select the correct BundleActivator instead of typing it in.

   c. In the **Import Packages** area, click **Add**. Use the default (checked) values that are provided and click **OK**.

d. In the **Export Packages** area, click **Add**. Select the com.ibm.osg.example.My_Web_Service_Bundle package and click **OK**.
e. Click **Save** to save the manifest file.

6. Create the bundle jar for My_Web_Service_Bundle with the following steps.
   a. Right-click on **My_Web_Service_Bundle** in the **Package Explorer** panel. Select **SMF -> Submit Bundle...**.
   b. Select **Submit Jar**. Click **Add Directory** and select the directory to output the bundle jar file and click **OK**. Select the directory under **Export Targets**. Click **Finish**.
   c. The bundle jar is placed in the directory you specify. For example, the jar file is `MyTestService+1_0_0.jar`. Rename the file `MyTestService.jar`.

      Note: You can rename the jar file with a different name. However, this example assumes that you rename the file `MyTestService.jar`.
7. Start the SMF Bundle Server from a command prompt as described in the Service Management Framework Help. You should then be able to view details about the server from the SMF Bundle Servers tab of the SMF perspective.
8. Submit the bundles to the server with the following steps:
   a. Right-click on the **My_Web_Service_Bundle** project and select **SMF->Submit Bundle**.
   b. Check the **Submit jar** checkbox, select the SMF Bundle Server and click **Finish**. The MyTestService bundle is uploaded to the SMF Bundle Server.
9. Start the SMF Runtime with the following steps:
   a. Select **Run -> Run...**.
   b. Double-click **SMF Runtime**.
   c. Click **Run**. The SMF Runtime starts.

      Note: From the SMF Runtime view of the SMF perspective you can now view details about the currently running SMF Runtime.
10. Install the bundles from the SMF Bundle Server to the SMF Runtime with the following steps:
    a. Bring up the SMF Bundle Servers view of the SMF perspective (this is usually tabbed with the Package Explorer view).
    b. Expand **Bundles**.
    c. Right click on the **Web_Service_Bundle** bundle and select **Install Bundle**.
11. View Output - You can view the output of the Bundles from the Console view (from the SMF or Debug perspective).

Using Web Services with SMF

SMF provides a service registry to enable bundles to provide services to each other. A bundle registers a service using a class name, an object that is an instance of that class, and a set of properties. Other bundles can then look up that service using the class name, a query, or both against the set of properties.

Generally, services are registered under the class name of a well defined interface. Because only local bundles can register a service, the service registry will contain only local services.

The Web Services Gateway bundle (WSOSGi) enables bundles to use Web Services as though they were local bundles. Subject to the limitations of Web Services, bundles using Web Services need not be aware that the service is not a local service.

Installing the Web Services Gateway bundle

To use wsosgi, you must install and start the wsosgi bundle in the framework. For wsosgi to work, it must be run on a framework that supports the DynamicImport-Package manifest tag. This tag is part of the "OSGi R3 Framework Specification" and is implemented in SMF.

Preparing to register Web Services

In SMF you can only register services that implement an exported class in the service registry. When wsosgi parses the WSDL for Web Service, it generates the name of the class to be registered using the port name, and calculates the package name of the class using the host name of the namespace of the port. wsosgi will then dynamically import the package corresponding to the class. If the class is not already exported, the Web Services will not be registered.

You must use another tool, such as WSDL2java (which is included with Apache AXIS), to generate the needed classes, or you must construct the classes by hand. You should generate the classes before wsosgi tries to register a service, because any client bundle that uses the service is compiled against the classes that correspond to the service. You must install and export the classes so that the client bundle can be resolved and started.

Registering Web Services

For a programmatic interface to registering Web Services, you can use the WSProxyService. After the wsosgi bundle is started, it registers the com.ibm.pvcws.osgi.proxy.WSProxyService. This service uses two methods to register services:

```
boolean register(String url);
boolean register(String url, Dictionary properties);
```

Both methods take the URL of a WSDL resource that describes the Web Services to be registered in the framework. The WSDL will be retrieved and parsed to figure out the location of the Web Services provider, the names of the interfaces to register, and the data structures and methods used by the interfaces. A service will then be registered using the class names of the interfaces derived from the WSDL.

After the service is registered, it can be retrieved from the service registry and used just as any other local service.

If the Web Services are accessed using a user name and password, the two argument version of the register method must be used.

Web Services will stay registered until the virtual bundle for that service is stopped or uninstalled. If the unregister method of WSProxyService is used, the bundle registering the service will be started.

To register Web Services, install and start the wsosgi bundle and go to the http://server:port/wsman Web page.

Prepackaging Web Services

You can also prepackage Web Services in a bundle that you can install using bundle deployment methods rather than the WSProxyService. The prepackaged bundle must contain the `ProxyActivator` class and a Manifest in the following format:

```
Bundle-Activator: ProxyActivator
DynamicImport-Package: *
WSDL-URL: url
```

If the WSDL_URL is present, the WSDL used to define the service to be registered will be obtained from the given URL. Otherwise, the WSDL will be drawn from a resource in the bundle at `/wsdl`. Properties needed to access the service provided will be drawn from a resource in the bundle at `/wsdl-props`.

For convenience, `WSProxyServiceImpl.createVirtualBundle` is a public static method you can use to create a prepackaged bundle.

Using Web Services

Although **wsosgi** proxies the Web Services as a local service, and bundles use this service in the same way as any other local service, there are still some limitations when using these services. Briefly, these limitations are as follows:

- Method parameters must be able to be serialized into primitive types. This means that remote references cannot be passed to, or returned from a method. A specific example of such a reference would be an object passed to a method that has non-accessor type methods that can be invoked by the called class.
- If non primitive data types are to be returned by a method, or set in an output parameter, the data types must have default constructors and public member variables or public accessors for all members that are described the WSDL.
- Currently, self referential data structures are not supported. An example of such a structure is a circular linked list.

Input parameters to the methods of Web Services work as expected subject to the preceding limitations. Output methods present a problem because Java does not have the concept of output parameters.

wsosgi enables more flexible handling of output parameters than JSR-101. In addition to supporting Holder classes, it also enables instantiated output parameters to be passed to a method and then have its members filled in directly. An example of this follows:

If there is a class named Coord:

```
class Coord {
    int x;
    int y;
}
```

And if there is a method `void getCoord(Coord coord)`, you would normally need to use a Holder class with a member named value that would be set when `getCoord` returns. Rather than requiring these extra classes and logic, `getCoord` can be passed an instance of Coord whose members, x and y, will be set on return.

The decision to use Coord directly or use a Holder is made when the Java interface is defined. wsosgi will base its handling of output parameters at runtime by evaluating whether or not the parameter is an instance of Holder.

Creating a Web Services interface file

To create a Web Services interface file, you must have already created an SMF bundle. A Web Services client must be added to an existing project. These instructions describe how to generate an interface file for the My_Web_Service_Bundle project created in "Creating an SMF bundle" on page 14.

1. Select **My_ Web_Service_Project** in the Package Explorer. Click **Open the New Wizard** in the workbench toolbar, then select **Other**. The Web Services wizard is under the Other category. The New dialog is displayed.
2. In the left frame of the **New** dialog, select **Mobile Web Services Client**. In the right frame, select **Web Services Client for Extension Services**. Then click **Next**. The Generator dialog is displayed.
3. Enter the following information into the Stub Generator Dialog:
   - Browse to the source directory of the project where you want the stub to be placed. This example uses **My_Web_Service_Bundle.**
   - The Package is the name of the Java package in which you want the output files to be created, for example, `com.example.smf.web.service`.
   - WSDL Location is the URL of the WSDL document that describes the Web Services for which you want to generate a stub. This example uses Web Services from www.acrosscommunications.com.
4. Click **Finish**.
5. An interface file and a stub is generated that you can use with your application for interfacing with the Web Services.

Using Web Services directly

Bundle programmers can also use Web Services directly rather than indirectly through the service registry. This is accomplished using WSDLProxy class. The simplest way to use WSDLProxy is the public static method `getProxy`, which returns an object of the requested type that corresponds to a service described by the WSDL. To get more precise information and implementations you must use the non-static methods of WSDLProxy.

Hosting an OSGi Web Services provider

Hosting OSGi Web Services allows OSGi services to be designated as having a Web Services interface during service registration. This support is patterned after the capabilities specified in the J2ME Web Services specifications. At present, there is no WS-Security server model for using the Web Services Gateway bundle.

To host non-secured OSGi Web Services, perform the following steps:

1. Create an OSGi service. This is the service that you are about to expose as a Web Service.
2. Register the OSGi service and add any additional properties by referring to, "Registering the OSGi Service" on page 20 below.
3. Deploy the bundle containing the OSGi service, as well as the Web Services Gateway bundle. For more information, refer to the SMF documentation.

Using OSGi Web Services

To use the OSGi Web Services provided by the above steps, perform the following procedure:

1. Retrieve the WSDL for the service. Refer to, "Registering the OSGi Service" on page 20 for more information.
2. Use the WSDL to develop the client for the service.

The Web Services Gateway bundle, wsosgi, enables bundles to use Web Services as though they were local services. Subject to the limitations of Web Services, bundles using Web Services need not be aware that the service is not a local service.

Registering the OSGi Service

All procedures for registering the OSGi Service takes place in the `start() method`.

Note: You are not limited to performing these steps in the `start() method`, providing you have a reference to the `BundleContext` object for registering a service at a later time.

There are two properties to set on the OSGi service:

- com.ibm.pvcws.wsdl

  The value for this property is either an empty String, or a String containing the actual WSDL (not a URL to the WSDL, as above).

  com.ibm.pvcws.wsdl is an empty String indicating that the service should be exposed as a Web Service. You can also pass a String containing the WSDL that describes the service if the WSDL must be of a form that cannot be autogenerated.

  If you do not use the empty String, the WSDL must have a location attribute. The value of the location attribute is unimportant since wsosgi will correct the location when it is served to clients.
- `Constants.SERVICE_PID`

  `Constants.SERVICE_PID` is an optional String that can be used to construct a predetermined URL to the service.

The registerService method takes the following three arguments:

1. The name of the interface or class that the service is registered under.

   The service must be an instance of this interface or class. This argument allows other bundles to use the service as a local service. If the service is only being exposed as a web service, `java.lang.Object` can be used.
2. An instance of the service.
3. The service properties.

Refer to the following example:

```
package com.ibm.wstkmd.bundle;

import java.util.Hashtable;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.Constants;
import org.osgi.framework.ServiceReference;

import com.ibm.pvcws.osgi.proxy.WebServiceProvider;

public class Activate implements BundleActivator {
         public void start(BundleContext context) throws Exception {
                  //Register the properties
                  Hashtable props = new Hashtable(3);

                  // This property indicates that the service should be exposed
                 // as a web service. If the property is the empty string,
                 // the WSDL will be autogenerated.
```

```
                props.put("com.ibm.wstkmd.bundle", "");

                 //This allows us to access the service as /ws/pid/getMean
                 props.put(Constants.SERVICE_PID, "getMean");
                 context.registerService(MeanProvider.class.getName(), new
          GetMeanProvider(), props);
                 WebServiceProvider provider = getProvider(context);
                 provider.exportPid("getMean");

    }

    public void stop(BundleContext context) throws Exception {

    }

    private WebServiceProvider getProvider(BundleContext context) {
             String providerName = "com.ibm.pvcws.osgi.proxy.WebServiceProvider";
             ServiceReference ref = context.getServiceReference(providerName);
             if (ref == null) {
                 return null;
             }
             WebServiceProvider provider = (WebServiceProvider)
   context.getService(ref);
             return provider;
    }

}
```

Once the bundle is deployed, the WSDL for the service can be retrieved at the URL **http://localhost:6080/ws/pid/<servicepid>?wsdl**

Note: `<service_pid>` is the value of the `Constants.SERVICE_PID` property set in the above example.

For the example above, the URL would be:

http://localhost:6080/ws/pid/GetMean/wsdl/GetMean.wsdl

As a development aid, you can use Web Services tools to generate the stub for the service. The tools create the stub with an endpoint referring to your machine address. Since it is unlikely that your machine address will match the deployed machine address, change the endpoint address used by the generated stub using the `<stub>._setProperty( Stub.ENDPOINT_ADDRESS, <string> )` method.

Note: The Web Services provider runs on port 6080. This is not configurable at this time.

Creating a Web Services application that includes WS-Security

Web Services applications must contain the following items for WS-Security to function properly:

- A keystore
- A WS-Security configuration file
- An application stub

To create a Web Services application that includes WS-Security, create a keystore with the instructions in “Creating a keystore” on page 22. Then, refer to “Using the Web Services wizard to create an application that uses WS-Security” on page 22 for

instructions to use the wizard to create a configuration file and application stub. You can use the Web Services wizard to configure the security attributes when using static web services stubs.

If your web services client application uses the Web Services Gateway bundle on SMF, then you need to programmatically set the Web Services Security properties, described in "Using the Web Services wizard to create an application that uses WS-Security."

If you use the Web Services Gateway Utility (described in Appendix A, "The Web Services Gateway Utility (WSOSGI-UI)," on page 35), then you can set the web services client configuration in a similar way to using the Web Services wizard.

Creating a keystore

To send a digitally signed message, you must provide a keystore that contains private and public keys to sign, encrypt, and decrypt messages. You can use the WebSphere Device Developer Keytool to create a keystore. Refer to the WebSphere Device Developer documentation for instructions to create a keystore.

Web Services supports only JKS type keystores, and the keystore name must end with a .jks extension, such as `client.jks`. Refer to the following list to determine the requirements for the keystore you want to create:

1. To digitally sign a request message, the keystore must contain the RSA or DSA private key for your client.
2. To encrypt a request message, the keystore must contain the certificate for the RSA public key of the receiver of the message.
3. To decrypt a response message, the keystore must contain the RSA private key to decrypt the secret key.

Using the Web Services wizard to create an application that uses WS-Security

After you create a keystore with the WebSphere Application Server Keytool, refer to the following instructions to use the Web Services client wizard to create an application that uses WS-Security:

1. Select the **Configure Security** checkbox on the **Web Service Client Stub Generator** window.
2. Click **Next** to specify information about the keystore.
3. Specify the file name of the keystore that you created with the Web server keytool and specify the password for the keystore.

   Note: You can use the **Browse** button to navigate to the location where the file resides.
4. Click **Next**.
5. Specify whether or not Web Services requires a digital signature for the messages it receives.

   If the Web Service requires digital signatures, select the appropriate signature algorithm for your application.

   Note: If you specify DSA for the signature algorithm, the WS-Security wizard will not permit the SOAP message to be encrypted.
6. Specify the alias and password of the key that the Web service will use for the signature.

Note: You do not need to provide configuration information to verify the signature of the received messages because the received message contains the information needed to reference the public key.

7. Click **Next** to specify the authentication data for sending messages.
8. Complete the following information on the **Authentication Data for Sending Messages** window to add authentication information to the SOAP messages you send:
   a. Select **Yes** next to **Authentication** to add the authentication information.
   b. Select either **Basic Authentication** or **Signature Authentication** if you require authentication. Refer to the following table for the authentication method options:

Table 1. Authentication method options

Method	User name	Password	Other
Basic Authentication	yes	yes	no
Signature Authentication	no	no	X.509 certificate

Note: To authenticate messages using the Signature Authentication method, the message must be digitally signed by the sender and you must specify the alias on the **XML Digital Signature for Sending Messages** dialog to retrieve the X.509 Certificate.

9. Click **Next** to specify the XML encryption for sending messages.
10. Complete the following information on the **XML Encryption for Sending Messages** window to specify how to encrypt the SOAP messages you send.
    a. Click **Yes** to **Encryption** to encrypt the message.
    b. Specify the alias for the RSA public key that will be used for key encryption.

    Note: WS-Security supports only RSA-1.5 as the key encryption method. The wizard generates a TripleDES key automatically.
11. Click **Next** to specify the **XML Decryption for Receiving Messages** window.
12. Complete the following information on the **XML Decryption for Receiving Messages** window to specify how to decrypt the SOAP messages you receive in your application:
    a. Select whether or not to enable decryption.
    b. Specify the alias and password for the key.
    c. Click **Add** to register the key.

    Note: You can register additional keys by repeating the previous steps.
13. Click **Finish** to generate the WS-Security configuration file and the Web Services application stub.

Programming WS-Security Properties

When developing secured Web Services clients that use the Web Services Gateway, WS-Security properties need to be customized as appropriate.

The example below illustrates a simple forms based dialog for setting WS-Security properties. You may also refer to the Web Services examples included with the Web Services tooling for complete working examples.

The following list properties can be set, followed by an example that illustrates the method of setting these properties.

Table 2. Keystore list properties

Name of WS-Security property	required/ optional	value
WSSAttributeType.REQUESTER_KEYSTORE_FILE	required	key content encoded by Base64
WSSAttributeType.REQUESTER_KEYSTORE_PASS	required	

Table 3. Digital Signature (dsig) list properties

Name of WS-Security property	required/ optional	value
WSSAttributeType.REQUESTER_SIG_ALIAS		
WSSAttributeType.REQUESTER_SIG_PASS	optional: if applied dsig, required	
WSSAttributeType.REQUESTER_SIG_ALG	optional: if applied dsig, required	either WSSAttributeType.URI_DSIG_RSA_SHA1 or WSSAttributeType.URI_DSIG_DSA_SHA1
WSSAttributeType.REQUESTER_DIGEST_ALG	optional: if applied dsig, required	WSSAttributeType.URI_DSIG_SHA1

Table 4. Authentication list properties

Name of WS-Security property	required/ optional	value
WSSAttributeType.REQUESTER_LOGIN_UNAME	optional: if applied basic auth, required	
WSSAttributeType.REQUESTER_LOGIN_PASS	optional: if applied basic auth, required	

Table 5. Encryption list properties

Name of WS-Security property	required/ optional	value
WSSAttributeType.REQUESTER_ENC_ALIAS	optional: if applied encryption, requied	
WSSAttributeType.REQUESTER_ENC_PASS	optional: if applied encryption, requied	
WSSAttributeType.REQUESTER_ENC_DATA_ALG	optional: if applied encryption, requied	WSSAttributeType.URI_ENC_3DES
WSSAttributeType.REQUESTER_ENC_KEY_ALG	optional: if applied encryption, requied	WSSAttributeType.URI_ENC_RSA15

Table 6. Decryption Keys list properties

Name of WS-Security property	required/ optional	value
WSSAttributeType.REQUESTER_DEC_KEY_NUM	optional: may set,when user has private keys	if 0, it is unnecessary to set.
WSSAttributeType.REQUESTER_DEC_ALIAS + count(1,2,3,...)	optional: may set,when user has private keys	
WSSAttributeType.REQUESTER_DEC_PASS + count(1,2,3,...)	optional: may set,when user has private keys	

The following example illustrates a method of setting WS-Security properties for a Web Service client that uses OSGi Web Services:

```
package com.ibm.webservices.timeformat;

import java.io.InputStream;
import java.util.Hashtable;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceReference;
import org.osgi.util.tracker.ServiceTracker;

import com.ibm.pvcws.jaxrpc.encoding.PrimitiveSerializer;
import com.ibm.pvcws.osgi.proxy.WSProxyService;
import com.ibm.pvcws.wss.proxy.WSSAttributeType;
/**
 * @author admin
 *
 * To change the template for this generated type comment go to
 * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
 */
public class TimeFormatBundle implements BundleActivator {

    private BundleContext context;
    private ServiceTracker tracker;

    private static String WSDL = "http://localhost/TimeFormat/services
   /TimeFormat/wsdl/TimeFormat.wsdl";

    /* (non-Javadoc)
     * @see org.osgi.framework.BundleActivator#start
   (org.osgi.framework.BundleContext)
     */

    public void start(BundleContext context) throws Exception {
        System.out.println("log: TimeFormatBundle");
        this.context = context;
        Hashtable properties = new Hashtable();

        // set WS-Security configuration

        //  two essential properties for WS-Security
        properties.put("com.ibm.pvcws.jaxrpc.msg.handler",
   "com.ibm.pvcws.wss.WSSHandler");
        properties.put("com.ibm.pvcws.jaxrpc.msg.config",
   "com.ibm.pvcws.wss.proxy.WSSConfigProxyImpl");
        // set keystore properties
        String KEYSTORE_CONTENTS = convert("client.jks");
        properties.put(WSSAttributeType.REQUESTER_KEYSTORE_FILE, KEYSTORE_CONTENTS);
        properties.put(WSSAttributeType.REQUESTER_KEYSTORE_PASS, "client");
        // set DSignature properties
        properties.put(WSSAttributeType.REQUESTER_SIG_ALIAS, "client_rsa");
        properties.put(WSSAttributeType.REQUESTER_SIG_PASS, "client_rsa");
        properties.put(WSSAttributeType.REQUESTER_SIG_ALG,
   WSSAttributeType.URI_DSIG_RSA_SHA1);
        properties.put(WSSAttributeType.REQUESTER_DIGEST_ALG,
   WSSAttributeType.URI_DSIG_SHA1);
        // set Authentication properties - not apply in this case
        // properties.put(WSSAttributeType.REQUESTER_LOGIN_TYPE,
   WSSAttributeType.REQUESTER_AUTH_SIGNATURE);
        // properties.put(WSSAttributeType.REQUESTER_LOGIN_UNAME, "admin");
        // properties.put(WSSAttributeType.REQUESTER_LOGIN_PASS, "tester");

        // set Encryption properties
```

```
        properties.put(WSSAttributeType.REQUESTER_ENC_ALIAS, "server_rsa");
        properties.put(WSSAttributeType.REQUESTER_ENC_PASS, "server_rsa");
        properties.put(WSSAttributeType.REQUESTER_ENC_DATA_ALG,
    WSSAttributeType.URI_ENC_3DES);
        properties.put(WSSAttributeType.REQUESTER_ENC_KEY_ALG,
    WSSAttributeType.URI_ENC_RSA15);

        // set properties of private key for Decryption
        properties.put(WSSAttributeType.REQUESTER_DEC_KEY_NUM, "1");
        properties.put(WSSAttributeType.REQUESTER_DEC_ALIAS
    + "1", "client_rsa");
        properties.put(WSSAttributeType.REQUESTER_DEC_PASS
    + "1", "client_rsa");
        // properties.put(WSSAttributeType.REQUESTER_DEC_ALIAS
    + "2", "server_rsa");
        // properties.put(WSSAttributeType.REQUESTER_DEC_PASS
    + "2", "server_rsa");

        String service = "com.ibm.pvcws.osgi.proxy.WSProxyService";
        ServiceReference ref = context.getServiceReference(service);
        if (ref == null) {
            System.err.println("Error: WSManService does not exist.");
            return;
        }
        WSProxyService wsManImpl = (WSProxyService) context.getService(ref);

        // consume WSDL
        if (!wsManImpl.register(WSDL, properties)) {
            System.err.println("Unable to consume wsdl.");
            return;
        }

        tracker = new ServiceTracker(context, TimeFormat.class.getName(), null);
        tracker.open();

        Object svcs[] = tracker.getServices();
        if (svcs != null)
            for (int i = 0; i < svcs.length; i++) {
                TimeFormat dsvc = (TimeFormat) svcs[i];
                String [] argu = {"CHINA","JAPAN","GERMAN","FRANCE","ITALY","US"};
                for(int j = 0; j < argu.length; j++){
                    String temp = dsvc.getFormat(argu[j]);
                    System.out.println(temp);
                }

            }
    }

    /* (non-Javadoc)
     * @see org.osgi.framework.BundleActivator#stop
     (org.osgi.framework.BundleContext)
     */
    public void stop(BundleContext context) throws Exception {

    }

    /**
     * This method is used to convert a jks keystore into a base64

  * encoding that can be understood by the WS-Security
     * runtime in the web services gateway
     * The keystore should be accessible in the classpath
     * @param The URI of the keystore
     * @return
     */

    public String convert(String uri) {
```

```
        InputStream istream = this.getClass().getResourceAsStream(uri);
        int kssize = 0;
        byte[] buffer = null, ksenc = null;
        String kscontents = null;
        try {
            kssize = istream.available();
            buffer = new byte[kssize + 2];
            int ret = istream.read(buffer);
            istream.close();
            ksenc = PrimitiveSerializer.encode_base64(buffer);
            kscontents = new String(ksenc, "utf-8");
        } catch(Exception e) {
        }
        System.out.println("KeystoreConverter: uri="
        + uri + "  keystore=" + kscontents);

        return kscontents;
           }

    }
```

Chapter 5. Reference

This chapter contains reference information that you might find useful when you develop an application that consumes Web Services.

Soap implementations

This section describes the XML Parser and SOAP binding.

XML Parser

The J2ME SOAP stack implementation includes an optional XML parsing component. For more information, see http://www.jcp.org. This component conforms to a subset of Java API for XML-based Parser (JAXP) and offers a Simple API for XML (SAX) callback interface.

SOAP Binding

The `soap:body` binding element in WSDL provides information on how to assemble the different message parts in the Body element of the SOAP message. Both RPC-oriented and document-oriented messages use the `soap:body` element; however, the style of the enclosing operation has important effects on the structure of the `Body` section:

- If the operation style is **document**, there are no wrappers. The message parts display directly under the `soap:body` element.
- If the operation style is **RPC**, each part is a parameter or return value that appears inside a wrapper element within the body. The operation name and the wrapper element are named identically and its namespace is the value of the namespace attribute. Each message parameter is represented under the wrapper by an accessor that is named identically to the corresponding parameter of the call.

  J2ME Web Services support document/literal, and do not support the RPC operation style.

The required **use** attribute indicates whether or not the message parts are encoded, or the parts define the concrete schema of the message.

If **use** is **encoded**, each message part references an abstract type using the type attribute. These attributes apply an encoding that is specified by the **encodingStyle** attribute to produce a concrete message.

If **use** is **literal**, then each part references a concrete schema definition using either the **element** or **type** attribute. For document style bindings, if a part references the **element** attribute, the element referenced by the part is displayed directly under the **Body** element. For RPC style bindings, the **element** attribute is displayed under the accessor element named the same as the message part.

If a part references a **type** attribute, the type referenced becomes the schema type of the enclosing element, which is **Body** for document style, or **part accessor** element for RPC style.

The J2ME Web Services JAX-RPC Subset implementation must use the document style **literal use operation mode**.

The JAX-RPC Subset specification requires support for the following default representation of the SOAP:Body element for **document** style operations:

- The SOAP:Body element marshals to contain at most one message part (wsdl:part), defined by the element form at the abstract level.
- All message parts (either parameters or return value) appear inside a single wrapper element, which is the first child element of the SOAP:Body element. The wrapper element for the request has a name identical to the unique operation name. The name of the wrapper element for a request is used on the server side to resolve the method on the target service endpoint.

  The message part has an accessor with name corresponding to the name of the parameter and type corresponding to the type of the parameter.

  SOAP predefines one body element, which is the **fault** element used for reporting errors.

  The soap:fault element is patterned after the soap:body element in terms of literal uses and contains only a single message part. The fields of the fault element are defined as follows:

  – **Faultcode** is a code that indicates the type of the fault. SOAP defines the following set of faults:
    - **SOAP-ENV:Client**, indicates incorrectly formatted messages.
    - **SOAP-ENV:Server** indicates delivery problems.
    - **SOAP-ENV:VersionMismatch** reports any invalid namespace for the envelope element.
    - **SOAP-ENV:MustUnderstand** reports errors regarding the processing of header content.
    - **Faultstring** is a human-readable description of the fault. It must be present in the fault element.
    - **Faultactor** is an optional field that indicates the URL of the source of the fault. It is similar to the SOAP actor attribute except that it does not indicate the destination of the header entry.

  A SOAP fault is mapped to either a service specific exception class, or to a java.rmi.RemoteException. The only exception returned to an application is a JAXRPCException. If there is a service specific exception thrown on the server that contains the web service, a message contained within the JAXRPCException will contain the service specific message.

  Currently there is no provision in the interface to return a **RemoteException** to the caller.
- The implementation of JAX-RPC does not use SOAPAction. The Web Services for J2ME specification does not require support of the SOAPAction, however, to inter-operate with some servers, and in particular, .NET hosted services, it is necessary to set SOAPAction within the HTTP header.

External Interfaces

The external interfaces supported by Web Services comprise the JAX-RPC subset, and the JAXP subset.

The JAX-RPC subset interfaces

The JAX-RPC subset APIs are packaged in the javax.xml.rpc package as follows:

- **The javax.xml.rpc.Stub interface.**

- The interface javax.xml.rpc.Stub is the common base interface for the stub classes. All generated stub classes implement the javax.xml.rpc.Stub interface. An instance of a stub class represents a client side proxy or stub instance for the target service endpoint.
- The javax.xml.rpc.Stub interface provides an extensible property mechanism for the dynamic configuration of a stub instance.

• **The javax.xml.rpc.JAXRPCException class.**

Refer to the javadoc description for the specific API description.

IBM extensions

The IBM implementation of the JAXRPC subset provides additional configuration properties that you can specify. Normally, these are used only by the stub, and do not require the application change them.

SOAPAction: `SOAPAction` is an HTTP Header key required by some Web Services servers. The value required for `SOAPAction` is obtained from the WSDL, and set correctly within the stub class. If this value needs to be updated, you can change the value using the `_setProperty` method and specifying a property name of `com.ibm.pvcs.jaxrpc.SOAPAction`.

HTTPContentType: The HTTP Content type defaults to a String of `text/xml; charset="utf-8"`. Normally, this is sufficient for most Web Services servers. In the event that the content type needs to be changed, you can specify a valid content type string by using the _setProperty method and specifying a property name of `com.ibm.pvcs.jaxrpc.HTTPContentType`.

The Web Services client programming model

The Web Services client programming model uses a WSDD plug-in to generate the required stub classes to access a Web service.

In accordance with the programming model specified in the Web Services for J2ME specification, Web Services provides:

1. **A generated stub from the Web Services Description Language (WSDL) description of the service.**

   The WSDD Web Services plug-in generates the stub. The stub generator uses the WSDL that is exported from the web service as its input. The stub is generated during the development phase of the client application.

   The code generated by the stub uses the runtime Service Provider Interface (SPI), which interacts with the runtime component.
2. **Instantiation of the stub**

   The client application uses an instance of the stub to indirectly access the Web service from which the WSDL definition was derived.

   It is imperative that the WSDL definition reflects the actual interface to the Web service at runtime. The JAX-RPC subset does not perform any version control. Any differences between the defined WSDL and the instance of the Web service may produce unpredictable results.
3. **Invocation of stub methods that correspond to the implementation of service endpoint operations**

   The Web Services client application uses an instance of the stub to set stub properties, including the service endpoint.
4. **Packaging the stub with the client application**

The generated stub is provided in source form. It is used during application development.

Sample Web Services stub source code

To define the WS-Security configuration to the WS-Security runtime, the Web Services client wizard includes WS-Security information in the Web Services stub. Refer to the following code fragment for a sample of the stub source code generated by the Web Services client wizard.

Note: Because the you typically use the WS-Security wizard after you develop the Web service on the server, you do not need to consider the WS-Security implications when you develop the service client.

```
Public class Ping_Stub implements javax.xml.rpc.Stub, com.ibm.wstkme.demonidlet.PingPort1{

        public javax.microedition.xml.rpc.Operation operation=null;
    public java.util.Hashtable typeMap = new java.util.Hashtable();
    public java.util.Hashtable properties = new java.util.Hashtable(); {
      properties.put(javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY,
      "http://localhost:9081/PingServerEJB/services/PingPort1");
//properties.put(javax.xml.rpc.Stub.USERNAME_PROPERTY, "");
//properties.put(javax.xml.rpc.Stub.PASSWORD_PROPERTY, "");
//properties.put(javax.xml.rpc.Stub.SESSION_MAINTAIN_PROPERTY, new java.lang.Boolean(true));

/* The following two lines are includes from the WS-Security configuration. */

      properties.put("com.ibm.pvcws.jaxrpc.msg.handler" , "com.ibm.pvcws.wss.WSSHandler");
      properties.put("com.ibm.pvcws.jaxrpc.msg.config" , "xxx_wssConfig");
```

Web Services Samples

You can use the following sample applications to get started writing Web Services clients:

- RSAwithSHA1

  This sample application connects to a secured Web service using a digital signature with RSAwithSHA1.
- Stock Quote

  This sample application demonstrates a Web Services client that connects to a stock-quote service.
- Signed_Encryption

  This sample application connects to a secured Web service by using digital signature and encryption.
- Distance client

  This example demonstrates a client application that connects to a calculator service in another location.
- BabelFish

  This example demonstrates a client language dictionary application.
- Temperature

  This sample application requests temperature information from a service at a given location.
- Signed_DSAwithSHA1

  This sample application connects to a secure service using DSAwithSHA1 signature.

- Authentication_Encryption

  This sample application connects to a secure service using authentication and encryption.

Release notes

This section contains additional information if you were unable to successfully execute your application.

Enabling MIDP applications to run

The Web Services library extends the base `java.rmi package` library. To enable applications to execute successfully on midp devices, you might choose to disable Java security by modifying the security.policy. The security policy definitions reside in the `security.policy` file in the lib directory of the midp runtime libraries. For WSDD, this file is located in the `<wsdd_dir>\wsdd5.0\ive-2.2\runtimes\win32\x86\midp20\lib` directory. By default this file contains the following rules for the untrusted domain:

```
...
     domain: untrusted
     session(blanket): NetAccess
     blanket(session): LocalConnectivity
     oneshot(oneshot): WriteUserData
     oneshot(oneshot): ReadUserData
     blanket(session): FileConnection
     oneshot(oneshot): AppAutoInvoke
     ...
```

In order for Web Services to work, you need to modify this section to the following:

```
domain: untrusted
     session(blanket): NetAccess
     blanket(session): LocalConnectivity
     oneshot(oneshot): WriteUserData
     oneshot(oneshot): ReadUserData
     blanket(session): FileConnection
     oneshot(oneshot): AppAutoInvoke
```

Note: You might choose to modify the `security.policy` file according to your needs; however, the domain where your MIDP resides is named `untrusted`.

In order to execute Web Service Applications on MIDP devices, the `WebServicesMIDP.jar` file needs to be placed in the ext directory of the MIDP runtime library. The default location is `<wsdd_dir>\wsdd5.0\ive-2.2\runtimes\win32\x86\midp20\lib\jclMidp20\ext`.

Note: In order to execute Web Service Applications on Palm devices, the `WebServices.prc` and, optionally, the `WS-Security.prc` files need to be installed on the Palm device. These .prc files can be found in the `wsdd5.0\ive-2.2\runtimes\palmos50\arm\midp20\prc` directory, located in your installation of WSDD 5.7

Running secure Web Service clients on SMF

In order to run a secure Web Service client on the SMF platform using static stubs and the `*_wssConfig` object, `WS-Security.jar`, `WebServicesME.jar`, and the client bundle need to be added to the bootclasspath of the SMF runtime.

In order to run a secure Web Service client on the SMF platform using the Web Services Gateway, the WS-Security.jar file and the WebServicesME.jar file need to be placed on the bootclasspath of the SMF runtime.

Migration considerations

When migrating a project containing a Web Service client to the WSDD 5.7 platform, the build path will need to be manually updated. In earlier releases, the Web Services Runtime libraries were in the folder designated by the classpath variable WS_RUNTIME. In WSDD 5.7, the Web Services runtime libraries are stored in different locations.

When you install the Web Services for MIDP feature and start the tooling for the first time, the classpath variable WS_RUNTIME_MIDP will be create. This classpath variable will be added to a project when a new Web Services for MIDP client is generated using the tooling. For existing MIDP projects containing Web Services, you must edit the build classpath to use the WS_RUNTIME_MIDP classpath variable instead of the WS_RUNTIME classpath variable. Additionally, the Web Services for MIDP runtime library is now packaged as WebServicesMIDP.jar rather then WebServicesWME.jar. WS-Security.jar for use with MIDP can be found in the directory designated by the WS_RUNTIME_MIDP classpath variable as well.

When you install the Web Services for Extension Services feature, as part of the Extension Services feature, the classpath variables ESWE_BUNDLES and ESWE_FILES will be create. These classpath variables will be added to projects when a new Web Services for Extension Services client is generated using the tooling. For existing projects containing Web Services, you must edit the build classpath to use the ESWE_BUNDLES classpath variable instead of the WS_RUNTIME classpath variable to locate the Web Services runtime library. The Web Services for Extension Services runtime library is now packaged as WebServicesME.jar rather then WebServicesWCE.jar. WS-Security.jar for use with SMF can be found in the directory designated by the ESWE_FILES classpath variable, in the webservices directory.

Appendix A. The Web Services Gateway Utility (WSOSGI-UI)

The Web Services Gateway utility (wsosgi-ui) is a diagnostic tool that runs on SMF, and is used in conjunction with the Web Services Gateway. The purpose of the utility is to provide limited access to certain web services with some limitations without having to write any code at all. You may find this diagnostic utility useful before you write any code to consume your specific web service. Although the diagnostic utility can consume most web services, it is certainly not a substitute for your production-level web service client. Limitations in accessing secured webservices via this diagnostic utility may limit its usefulness. The utility is uninstalled into the bundle server by default.

- Consume Web Service clients
- List the Web Services clients that are currently being proxied via the Web Services Gateway
- Configure web service security for client Web Services

It has the ability to list local services that have been exposed as Web Services via the Web Services Gateway. Additionally, the diagnostic utility provides indirect access to almost all the functionality of the `wsosgi com.ibm.pvcws.osgi.proxy.WSProxyService`. The diagnostic utility provides a simple mechanism for creating web based tests of Web Services by implementing the `com.ibm.wsosgi.proxy.test.WSProxyTestService` interface. Finally, by enabling on-the-fly class generation when consuming wsdl and using the DynamicTest functionality, the diagnostic utility allows you to consume and test remote Web Services without writing any code.

Accessing the diagnostic utility

In order to access the diagnostic utility, the 'wsosgi', 'wsosgi-ui', and standard 'HttpService' bundles must be installed and started. The 'wsosgi-ui' bundle creates a servlet using the HttpService and registers itself as 'wsman'.

To access the utility, use the url: `http://smfserver:httpPort/wsman`, where `smfserver` is the SMF server name, and `httpPort` is the port # configured for the SMF server.

Consuming Web Services

The process of consuming a Web Service takes a valid WSDL URL, parses it, and creates and installs a virtual SMF bundle that proxies the Web Service. This is done using the *wsosgi com.ibm.pvcws.osgi.proxy.WSProxyService* service. The virtual bundles that are created remain installed on the SMF system until they are unregistered. That means that they will be present and started even when you restart SMF.

To consume Web Services, select the Create a **Web Service Client** link. You need to specify the URL to the WSDL for this service. This can be an http URL or a file based URL. You can either paste the WSDL URL into the **URL** field or use the **Quick Selection** drop-down. Every WSDL that is successfully consumed will go into the **Quick Selection** drop-down for future access. You can use the Clear Selected or Clear All to remove items from the quick selection box.

The Optional Parameters allow you to specify a username/password for the service or override the target endpoint that comes from the WSDL.

Select **Generate client class files in virtual bundle** parameter. If this is set to **Yes** (default), then the set of all classes that the Web Service uses will be generated dynamically and exported from that bundle. This allows you to create an SMF bundle Web Services client bundle without writing any code.

Listing Web Services Clients

Select the 'List Consumed' link to list all Web Services that have been registered with the Web Services Gateway bundle. For each Web Service client, the following actions can be performed:

unregister	Stops and uninstalls the virtual bundle for this web service.
test	If a *com.ibm.wsosgi.proxy.test.WSProxyTestService* is registered for this service then this allows you to use that to interactively test the Web Service. If none is registered then this reverts to dynamic test.
dynamic test	This allows you to run a dynamically created test for any of the methods of the Web Service. You will be able to choose which method to run and then fill in all of parameters via html forms. You can then execute the method and view the results.

Using the Web Services Gateway Utility to configure WS-Security properties

To configure WS-Security properties with the Web Services Gateway Utility, perform the following steps:

1. From the drop down menu, select **Yes** for **Set Web Service Signature configurations**.
2. Enter the **Keystore Contents** location.
3. For **Keystore Password** enter `client`.
4. From the drop down menu, select **Yes** for **Do you need authentication**.
5. For **Signature Algorithm**, select **RSA-SHA1**.
6. For **Option**, select **basic auth**.
7. For **Username**, enter `Admin`.
8. For **Password**, enter `tester`.
9. From the drop down menu, select **Yes** for **Do you need encryption**.
10. For **Alias**, enter **server rsa**.
11. If desired, create a Public Key by choosing and entering an **Alias** and **Password**. Each Alias/Password set creates one Public Key.
12. Click **set configuration** to complete the configuration process.

Note: In order to successfully register a secure WSDL, you must add the `WebServicesME.jar` and `WS-Security.jar` to the bootclasspath of the SMF runtime.

Testing Web Services Gateway Clients

Testing Web Services Gateway clients can be done in one of two ways. The first method is by running a dynamically generated test for the Web Services Gateway client. This works in most cases and is sufficient for testing simple Web Services. If you prefer a customized test for the service, you can also implement the *com.ibm.wsosgi.proxy.test.WSProxyTestService*. The test subsystem is designed to assist testing your Web Services Gateway clients. It remembers all previously used arguments for a given test in order to lessen the complexity of html forms based testing.

Dynamic testing

The dynamic test also implements the WSProxyTestService interface. It uses java reflection to attempt to dynamically generate a test for each of the web service's methods. Currently, dynamic test does not handle the input of all arrays. Rather, it defaults to array size of three every time. It is recommended that you try dynamic test before implementing a special WSProxyTestService, as this may be sufficient in many cases. In order to use this type of testing, from the **List of OSGi Web Services Clients Consumed** page simply click on the dynamic test link of the desired Web Service.

WSProxyTestService

WSProxyTestService allows you to create a test with various degrees of customization. In order to use this form of testing, from the **List of OSGi Web Services Clients Consumed** page simply click on the test link of the desired Web Services Gateway client.

Appendix B. Notices

This information was developed for products and services offered in the U.S.A. IBM might 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 might have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give 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, 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 information. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this information 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
Department LZKS
11400 Burnet Road
Austin, TX 78758
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 document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, 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 measurement 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. 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 IBM's application programming interfaces.

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 IBM's application programming interfaces.

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. *2004* All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, or other countries, or both:

IBM

Other company, product or service names may be trademarks or service marks of others.