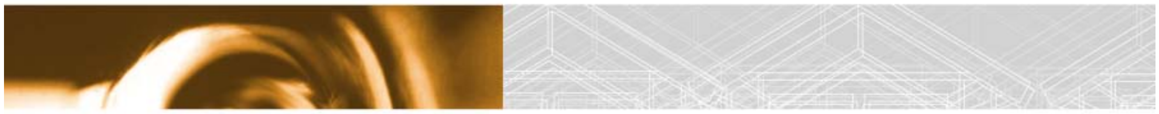
HP Web Services Platform



User's Guide

Early Access Release : 01/2002

Legal Notices

Corporate Offices:

Hewlett-Packard Co.
3000 Hanover St.
Palo Alto, CA 94304

Contents

List of Figures

Code Listings

List of Tables

About this Guide

Introduction

The *Hewlett-Packard Web Services Platform User's Guide* presents a detailed view of the services, functions and technologies implemented in this release.

Sample source code, XML files and Web Services Description Language (WSDL) documents are located in the `samples` subdirectory of the installation directory for this product.

In this chapter

Who Should Read This Guide

This document provides information about the HP Web Services platform. It describes how to expose an EJB or Java class as a web service, including deployment and configuration. It also describes the architecture of the HP Web Services platform. Although this guide is specifically intended for web service developers, it will be useful to anyone who would like to gain an understanding of web services and how they functions.

What You Should Know

This guide assumes a basic familiarity with Java™ development and object-oriented programming. A fundamental level of understanding in the following areas will also be useful:

- A working knowledge of Java 2 Platform, Enterprise Edition (J2EE), XML, Simple Object Access Protocol (SOAP) and WSDL.
- A general understanding of the Windows and/or UNIX operating systems.

Web Services Documentation

A set of documentation has been created for this Web Services Release, as follows:

Table p-1: Available Web Services Documentation

Document	Description
Web Services User's Guide	Contains fundamental information about the web services available, including design and operation.
Web Services API Javadocs	Describes the classes, inner classes, interfaces, constructors, methods, and fields of the web service API.
Web Services Release Notes	Contains the information needed to install this release of the HP Web Services platform. It includes system requirements and a list of the files that are installed.

Documentation Conventions

The following conventions are used in this guide:

Table p-2: Documentation Conventions

Convention	Description
<install_dir>	The installation directory of the web services product. The default installation directory is c:\hpmw\hpsoap.
Bold	Used to identify section references.
Italic	In paragraph text, *italic* identifies the titles of documents that are being referenced. When used in conjunction with the Code text described below, italics identify a variable that should be replaced by the user with an actual value.

Table p-2: Documentation Conventions

Convention	Description
`Code`	Text that represents programming code.
CTRL+*X*	A combination of keystrokes used to complete a function. For example, CTRL+C indicates that the user should press and hold the CTRL key while simultaneously pressing the C key.
Function | Function	A path to a function or dialog box within an interface. For example, "Select **File | Open**" indicates that you should select the **Open** function from the **File** menu.
[ ] and |	Brackets enclose optional items in command syntax. A vertical bar separates syntax items in a list of choices. For example, in the comand below, the last parameter is optional: `ClassToWebService <Java Class Name> <Service Endpoint URL> <Service Name> [<WSDL file>]`
Note and **Caution**	A **note** highlights important supplemental information. A **caution** highlights procedures or information necessary to avoid damage to equipment, damage to software, loss of data, or invalid test results.

Product Information and Support

Your comments and suggestions help us to provide accurate, quality documentation. If you have comments about any of the Web Services documentation, or if you would like to find additional information about our products, the following resources are available.

The HP Middleware Website

The starting point for all of HP Middleware's customer support and training is our Website. Visit it at www.hpmiddleware.com for information and support, including:

- The knowledge base.
- Support mailing lists.
- Add-ons and patches.
- Demos.
- Product evaluation support.
- Training.

HP Middleware Technical Support

If you need assistance or would like to send us your feedback, contact our Technical Support team.

Support Services
6000 Irwin Road
Mount Laurel, New Jersey 08054
Phone: 856.638.6000
www.hpmiddleware.com/support

Additional support features are available for users with current maintenance contracts, such as a toll-free, 24-hour customer support line. Visit HP Middleware's website for details about our maintenance contracts and the additional support we provide.

Chapter 1

Introduction to the HP Web Services Platform

What is a Web Service?

Web services are modular, reusable software components that are created by exposing a business application through a web service interface. For example, a business might have an application that creates an amortization schedule for any given mortgage amount. This application can be offered as a web service and made available to anyone over the Internet.

Web service interfaces are published in Universal Description, Discovery Integration (UDDI) registries and are described and discovered using the Web Services Description Language (WSDL). Businesses create descriptions of their web services in a WSDL file and publish that file to a UDDI registry. Users examine these WSDL file descriptions to discover available web services. Web services are accessed using the Simple Object Access Protocol (SOAP) through request/response SOAP-based message exchanges over standard transport protocols (primarily, HTTP).

Using standards based communication technologies, such as SOAP, UDDI, and WSDL, web services can communicate directly with other web services. In other words, application components, implemented as web services, can be accessed by customers, suppliers, and trading partners, independent of hardware operation system or programming environment. The result is a vastly improved collaboration environment as compared to today's Electronic Data Interchange (EDI) and Business-to-Business (B2B) solutions.

The HP Web Services platform creates an environment that enables users to leverage all of the functionality described above.

In this chapter

What is the HP Web Services Platform?

The HP Web Services platform is a software infrastructure for developing and deploying loosely coupled web services. These web services can be any mixture of internal and external services, and may include applications, business processes, computing resources, or information stores. The platform is extremely flexible and can help users to complete tasks, solve problems, or perform transactions. The HP Web Services platform is standards compliant, thereby making it widely accessible to users. Companies can also create secure private networks to serve a more controlled community (e.g., a supply chain) using the HP Web Services Registry Composer and the HP UDDI Registry.

The HP Web Services platform provides customers with the infrastructure and tools necessary for creating, deploying, registering, discovering, and accessing web services. More specifically, HP's Web Services platform includes support for the following:

- Web Services Description Language (WSDL) support for describing web services.
- An HP-SOAP server for responding to Simple Object Access Protocol (SOAP) requests from other applications.
- A Universal Description, Discovery and Integration (UDDI) browser tool (Registry Composer) for registering web services and for discovering other services.
- A UDDI registry for creating private networks of web services.

The HP Web Services platform provides all of the capabilities needed to create web services that expose new or existing applications (Java applications in our case) as web services. These services can be registered in either public or private UDDI registries, and invoked via SOAP servers.

This early access release of the platform does not include the UDDI Registry or the Registry Composer. These tools will be included in the next release, which will be available in the very near future.

An Architecture Designed for Web Services

The HP Web Services platform provides a single architecture for creating and deploying web services, and for publishing and discovering web services in public and private registries. It provides a consistent approach for creating web services from existing Java classes, EJBs, and Cocoon applications. HP's goal is to provide a plug-and-play platform that enables interoperability across a range of messaging profiles, from RosettaNet to Biztalk and ebXML.

Service interfaces published in a registry are programmatically discoverable and loosely coupled, making it easy for developers to locate and start using them immediately. This robust and modular web service infrastructure runs on top of the J2EE-compliant HP application server and provides interoperability with Microsoft .NET environments.

Compliant with Web Service Standards

The HP Web Services platform uses open Internet standards for peer-to-peer messaging. Its modular structure facilitates extension to accommodate new standards as they emerge. XML and Java, the core building blocks of the platform, are the dominant Internet standards for Web-based application development and interoperability. HP employs emerging standards for the creation, registration, and discovery of web services including:

- SOAP
- WSDL

- UDDI
- JAXM

SOAP

SOAP has emerged as the de-facto message format for XML-based communication in general, and web services in particular. It is a lightweight protocol with a framework that allows the user to define the content of a message and to provide processing hints. Although the design of the HP Web Services platform allows the receipt and processing of any XML document, its focus is on SOAP.

SOAP messages can be divided into two main categories: Remote Procedure Call (RPC) and Document Exchange (DE). The primary difference between the two categories is that the SOAP specification defines encoding rules and conventions for RPC. The document exchange model allows the exchange of arbitrary XML documents--a key ingredient of business-to-business (B2B) document exchange. The HP Web Services platform accommodates both categories of SOAP messages.

Note: For more information about SOAP and RPC, see sections four and five of the SOAP 1.2 Specification, Part 2.

UDDI

Universal Description, Discovery and Integration (UDDI) is a repository-based registry service for the automated lookup of web services. Think of UDDI as yellow pages that allow users to locate web services. A registry stores information regarding the suppliers of web services, the services they offer, and the appropriate contact information.

UDDI registries can be public or private. HP is one of three web service vendors that have agreed to provide a public UDDI registry, and the HP Web Services platform will also provide out-of-the box private registries. Private registries enable the creation of private networks of web services.

WSDL

WSDL (Web Services Description Language) is an XML-based language used to define web services and describe how to access them. An application trying to use a web service uses a WSDL document to discover the location (URL) of the service, the method calls available, and how to access them (for each method call, WSDL describes the format that the client must follow). Therefore, the client must first obtain a copy of the WSDL file from the server, and then use the information in this file to format the SOAP request.

JAXM

The Java API for XML Messaging (JAXM) is designed for the exchange of XML business documents over the Internet. Examples of XML documents that might typically be exchanged are purchase orders, order confirmations, and invoices. You can also send non-XML data by adding attachments to your message. The HP Web Services platform uses the JAXM API to provide messaging functionality.

Chapter 2

HP-SOAP Server Architecture

Introduction

The HP-SOAP server is designed to facilitate message exchanges according to the Simple Object Access Protocol (SOAP), and it has plug and play architecture. The HP-SOAP server provides a SOAP processing pipeline that handles both incoming and outgoing messages. The pipeline includes a set of header processors, adapter/handlers, data type management features, and XML digital signature security support. These features enable the HP-SOAP server to securely receive and route the SOAP message's business payload to back-end applications for processing and return an appropriate response. In addition, custom adapter/handlers can be created and added to the pipeline to expand its functionality. This provides the user with the plug and play capability needed to create and deploy a wide range of web services.

The current version of the HP-SOAP server uses the Apache Cocoon 2.0 framework; thus it takes advantage of Cocoon's pipeline controller features and application server-neutral services, such as component caching and error processing.

This chapter discusses the architecture of the HP-SOAP server.

In this chapter

HP-SOAP Server

The HP-SOAP server consists of a single processing pipeline that supports both RPC and Document Exchange services. The HP-SOAP server performs SOAP-header, SOAP-RPC body, and document exchange synchronous response message processing. This includes messages that contain attachments. Processing messages in HP-SOAP occurs in two places: the HP-SOAP Messaging Layer and the HP-SOAP Server Pipeline.

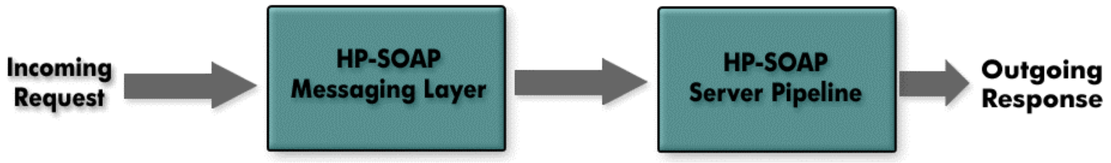


Figure 2-1: HP-SOAP message processing

HP-SOAP Messaging Layer

The HP-SOAP Messaging Layer manages the association/binding of soap messages to a transport protocol (e.g., HTTP, SMTP), the creation of a request context, and internal routing of the inbound messages to the processing pipeline. These tasks are facilitated through a collection of protocol specific listeners for transport level tasks, and an In-Box Servlet for context creation and internal routing. This messaging infrastructure represents HP's implementation of what the JAXM specification calls a JAXM Provider.

The HP-SOAP Messaging Layer uses the Java Services Framework™ (JSF) to implement the listener components. The Java Services Framework is an open, standard mechanism for assembling service components into Java server applications – more information can be obtained at http://www.jcp.org/jsr/detail/111.jsp.

The SOAP specification defines bindings to the HTTP protocol and allows for additional transport bindings. HP-SOAP supports SOAP over HTTP on all supported platforms. When HP-SOAP is combined with HP's application server (HP-AS), other protocols can be supported - due to HP-AS' implementation of the Java Services Framework. HP-AS can be extended and enable protocols such as SMTP and FTP to be used. In addition, listeners may be combined with a Load Balance Broker (LBB) to distribute the incoming requests among multiple application server instances, as seen in *Figure 2-2*

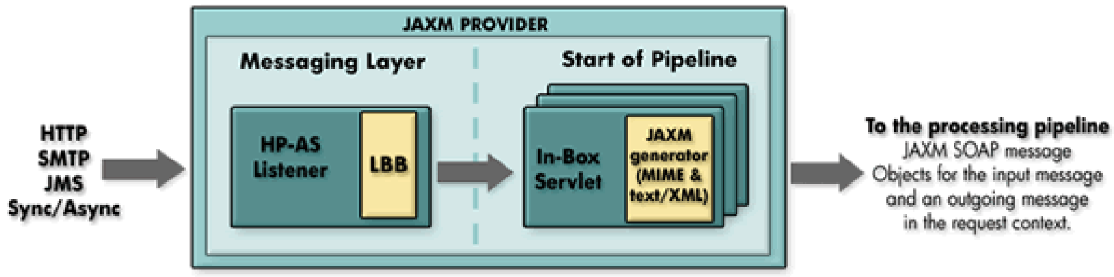

JAXM PROVIDER
Messaging Layer
Start of Pipeline
HTTP
SMTP
JMS
Sync/Async
HP-AS Listener
LBB
In-Box Servlet
JAXM generator (MIME & text/XML)
To the processing pipeline
JAXM SOAP message
Objects for the input message and an outgoing message in the request context.

Figure 2-2: HP Web services messaging framework

The server includes an implementation of the Java API for XML Messaging (JAXM) SOAP package (i.e., java.xml.soap). It uses JAXM as the primary API for gaining access to the contents of the incoming SOAP message (including attachments) and for manipulating the response message (including fault messages in the event of an exception). The in-box manages dispatching the incoming request to the pipeline. The server ships with a preconfigured in-box for both RPC and Document Exchange SOAP messages.

The JAXM generator signifies the beginning of a processing pipeline. It is responsible for creating a JAXM SOAPMessage object from input in the data stream and for creating an empty output SOAPMessage object so that various pipeline components can have access to both the request and response. After these objects are created, they are passed to the pipeline.

Note: All components in the server, except for the listener/LBB combination, are application server-neutral. The listener/LBB combination requires the Java Services Framework, which is currently implemented only in HP-AS.

HP-SOAP Server Pipeline

As shown in *Figure 2-3,* the HP-SOAP server pipeline consists of two stages: a SOAP header processing stage and a SOAP payload processing stage. At the header processing stage, the SOAP server parses the incoming request to determine which header processor contains the routing information needed to identify the web service that will process the message's business payload. The payload processing stage involves transforming the message payload from XML into a Java object so it can be processed, and then transforming the output back into XML. The message is then sent to a JAXM Serializer that transforms the response message to the HTTP protocol so that it can be sent across the wire to its destination.

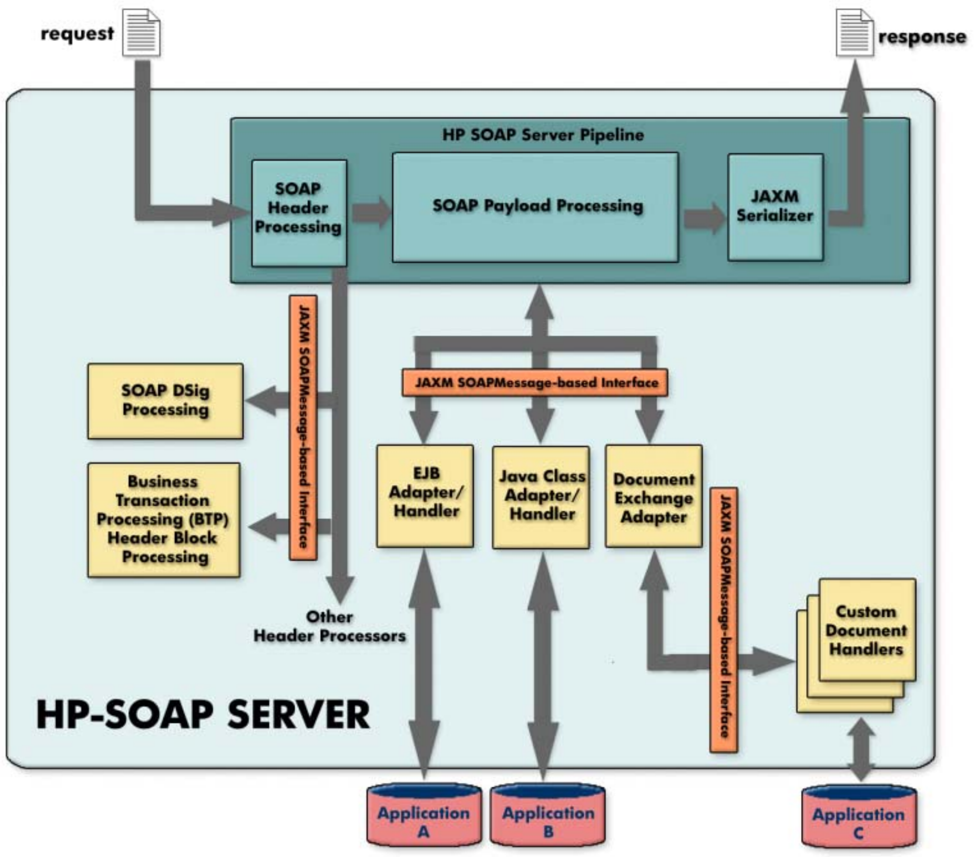


Figure 2-3: SOAP Pipeline

HP-SOAP Server Pipeline Processing

SOAP Header Processing

When an incoming XML message is received, the HP-SOAP server parses the message header to obtain information describing what should be done with the message payload. Generally, communicating parties have already agreed upon the semantics that govern the processing of the header element. So the SOAP server knows what to do with the message depending upon what it finds specified in the header.

SOAP provides an extensible header schema that allows communicating parties to embed their header extensions. These extensions are implemented as SOAP header blocks. An incoming SOAP message contains one or more `<header-block>` elements that specify which header processor(s) should handle the message. The HP-SOAP server maps the `<header-block>` values from the request to `<header-block>` values in a configuration file on the server and invokes the appropriate header processor.

For example, both ebXML and BizTalk define header blocks that are specific to their protocols. SOAP security extensions, such as SOAP-DSIG and the Business Transaction Protocol (BTP), define protocol-neutral header blocks. If a message is received that specifies one of these protocols, that header processor will be invoked.

The HP-SOAP server can facilitate both ebXML and BizTalk header blocks in its header processing stage. All header processors have access to the input and output JAXM `SOAPMessage` objects.

Header Processors

The header processing stage in the HP-SOAP server provides a configuration based dispatch mechanism to register header processors with the server. The header configuration file (`header-adapter.xconf`), maps the incoming SOAP message header blocks to their corresponding header processors.

The header configuration file contains the following elements:

- **Global SOAP Actor -** The global `<soap-actor>` element contains a URI that defines the global configuration information for all web services within a WAR file (a Web Archive file containing all of the web services being deployed. You can override these global settings at the web service level in the configuration file. See Web Service Specific Configuration below.
- **Header Groups -** Header Groups `<header-group>` are a convenient way to group related header blocks and associate them to a single header processor. In other words, there may be different header blocks specified in an incoming message that can be satisfied by the same header processor. The header-group tag makes it easy to list these blocks and assign them all to a single header processor. The `className` attribute identifies the header processor designated to process the group.

  Header Blocks – Header blocks `<header-block>` are child elements of header groups and contain the look-up information used to match the `<header-block>` elements contained within the incoming message. Each header block is identified by a `name` and `URI` attribute combination that tell the server which header processor to use and where it can be found. If the header block information does not match, header processing will not occur.
- **Web Service Specific Configuration** – The header configuration file allows you to make web service specific entries that can override the global soap-actor, header-group, and header-block configurations. The `<service>` element may contain a service specific `<soap-actor>` attribute that takes precedence over the global `<soap-actor>`. The `<service>` element can specify `<header-group>` and `<header-block>` elements that override the global <header-group> and <header-block> specifications.

Note: If you specify header processors at the web service level, they completely override the processors defined at the global level. In other words, global and service level configurations are not merged. For more information on the web service configuration files, refer to *Chapter 5, Deploying & Configuring Web Services.*

SOAP Node

A SOAP message may specify one or more recipients. These recipients are referred to in the SOAP specification as SOAP nodes. A SOAP node can be a server that contains many web services. A SOAP node's role can be the initial sender, the ultimate receiver, or an intermediary. What role it plays depends upon its position in the messages' path toward final processing. In other words, if the node is the final receiver of a message, its role is the ultimate receiver. If a node receives a message and will process it and then forwarding it to another receiver, it is acting as both an intermediary and a sender. If it is merely passing the message to the next node it is an intermediary.

In the diagram below, if at node 2 a web service performed some processing on the request and then forwarded it to the next node, node 2 is acting as an intermediary and as a sender.

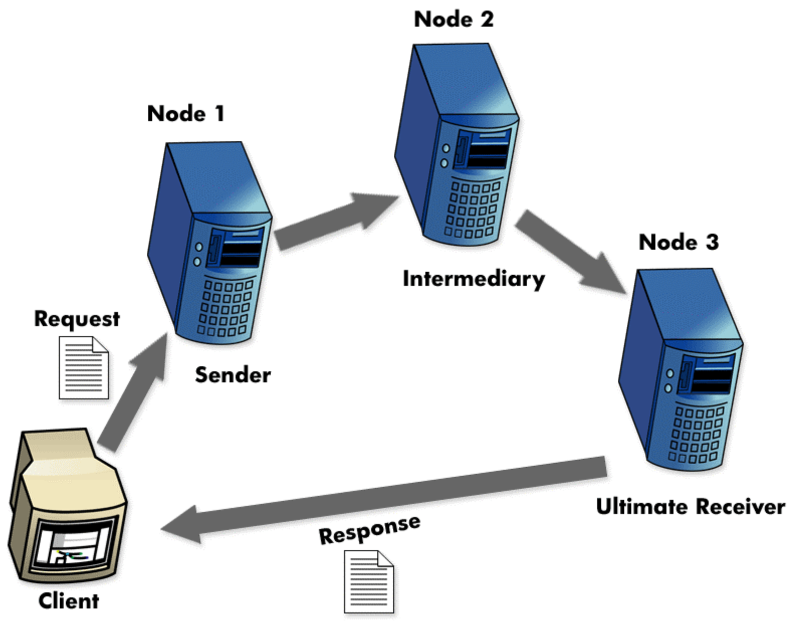


Figure 2-4: SOAP nodes

The idea of a SOAP node is important because it forms the basis for controlling the processing flow of the message. This greatly expands the ways a message can be handled and the types of services that can be performed on the message payload.

SOAP Actor

A SOAP node's role is defined by the `<SOAP-ENV:actor="…">` attribute in the SOAP message.

```
<SOAP-ENV:Header>
<h:echoMeStringRequest xmlns:h="http://soapinterop.org/echoheader/"
 SOAP-ENV:actor="http://schemas.xmlsoap.org/soap/actor/next">hello world
</h:echoMeStringRequest>
```

The SOAP actor attribute contains a Uniform Resource Identifier (URI) that identifies the next recipient of the SOAP message.

```
<SOAP-ENV:Header>
<h:echoMeStructRequest xmlns:h="http://soapinterop.org/echoheader/"
 SOAP-ENV:actor="http://schemas.xmlsoap.org/soap/actor/next"
 SOAP-ENV:mustUnderstand="1">
</h:echoMeStructRequest>
```

If there is no actor attribute in the SOAP message, then the default URI, `http://schemas.xmlsoap.org/SOAP/actor/next`, is used. This causes the SOAP message to be passed to the next recipient in the pipeline.

For a more detailed discussion of the SOAP actor attribute, see the SOAP specification 1.2, part 1, sections 1.4, 2.2, and 4.2.

SOAP Payload Processing

The HP-SOAP server is capable of processing SOAP messages with attachments and SOAP messages sent with content type set to plain text/xml. The business payload portion of the message can be embedded directly in the body portion of the message and/or delivered as attachment(s). The payload itself may represent either an RPC request or Document Exchange request. The payload processing stage of the pipeline allows you to process both types of requests through a single pipeline.

The payload processing stage of the pipeline can dynamically select one of three adapters: an EJB adapter, a Java class adapter or a Document Exchange adapter. The EJB adapter and the Java class adapter are designed to process SOAP-RPC requests mapped to an EJB or a Java class through a web service configuration file. These adapters perform the necessary serialization and deserialization of RPC parameters as well as method invocation. The Document Exchange adapter allows developers to take control of any SOAP request through a Document Handler.

If the SOAP message contains attachments, you can access them via the JAXM API.

Adapter/Handlers

Message Handler Interface

The message handler interface is the primary API between the processing pipeline and the adapter/handlers. All RPC handlers shipped with the HP-SOAP server have been implemented using this interface.

```
public interface SOAPMessageHandler {
    public void setWebServiceContext ( WebServiceContext context ) throws
Exception;
   public WebServiceContext getWebServiceContext ( );
   public void processMessage( SOAPMessage input, SOAPMessage output )
throws  Exception;
}
```

A handler can access the attachments and SOAP envelope information using the standard JAXM API. Note, that the header processing stage of the pipeline may have already removed certain header blocks from the input message.

Our current implementation of SOAP payload processing includes four ready-to-use adapters/handlers:

- A header-processing adapter.
- An RPC Java adapter/handler.
- An RPC EJB adapter/handler.
- A document exchange adapter.

The **header-processor** adapter extracts header information from the incoming message, reads the header configuration file, and tells the server which header processors to load. A header processor creates the context object that contains information needed to route the message to the processing application.

An **adapter/handler** is a software component that is responsible for receiving the business payload and passing it to the application that will process it. It also takes the processed message and passes it to its next destination. Adapter/handlers also deserialize the RPC parameters of the incoming XML message into Java objects so they can be processed, and then they serialize them from Java objects back into an XML response message. They also perform method invocation.

The **Document Exchange** adapter is different from the RPC adapter/handler. In RPC, the processing logic is contained in the adapter/handler component. In the document exchange paradigm, the adapter is a separate component from the handler. The document exchange adapter is only responsible for passing the message to the specified document handler. The document handler contains all of the processing logic. This model allows developers to take control of any SOAP request by writing custom document handlers that plug into a single document exchange adapter. If the SOAP message contains attachments, you can access them via the JAXM API.

To learn how to write your own custom handler, see *Writing a Document Handler* in Chapter 3.

SOAP Payload Processing in Action

After header processing is completed, the SOAP message body is dispatched to an appropriate adapter/handler for additional preprocessing and for routing to the business application that will satisfy the request. The server extracts the relevant payload information from the `WebServiceContext` object and passes it to the correct adapter. At this point, formatting can be applied to the text values and then converted into a Java object. The adapter invokes the handler, which passes the business payload to the business application that will process it.

After the payload is processed, the adapter/handler retrieves the message; it is converted back into an XML element, formatted, and dispatched to the JAXM Serializer. The Serializer transforms the data into a string so that it can be returned to the requester. The HP-SOAP server uses standard RPC formatters to format the input/output and Castor Converter classes to convert the SOAP input/output message into the desired formats.

Data Type Support

The HP-SOAP server provides extensive support for data types, including complex data types. Type converters are used to deserialize data from XML into Java objects and to serialize the Java objects back into XML. Value formatters are used to modify the value associated with a parameter or a return value. For more information, see *Chapter 4, Data Types in HP Web Services Platform*.

Digital Signatures

Secure and reliable exchange of SOAP messages is critical to the success of web services in the business world. SOAP messages have to be authenticated and possibly encrypted so that both the sender and receiver have assurance of secure message exchange. Authentication in this environment includes authentication of the sender and authentication of the message itself. While the former guarantees that the sender and recipient are who they say they are, the later ensures that the message was not modified by anyone during transmission. Digital signatures ensure authentication of the message and should be used in conjunction with other technologies, like SSL, to ensure the authentication of the sender.

The XML Signature specification at http://www.w3.org/TR/xmldsig-core/ is the core standard that allows digital signatures in the SOAP message. This standard defines the syntax and processing rules for creating and representing digital signatures that can be applied to any digital content, including XML. The SOAP Security Extensions specification at http://www.w3.org/TR/SOAP-dsig/ proposes a standard way to use XML Digital Signature to sign a SOAP document. In particular, the `SOAP-Dsig` extension defines a header entry `<SOAP-SEC:Signature>` that can specify the `SOAP actor` and the `mustUnderstand` attributes. In addition, the extension specifies the use of XML identifiers to refer to the signed part of the SOAP Envelope.

The following example shows the header entry along with the `ds:Reference` element that refers to the Body portion of the SOAP Envelope via the ID reference.

Listing 2-1: Sample of header entry for Dsig extension

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header>
    <SOAP-SEC:Signature
      xmlns:SOAP-SEC="http://schemas.xmlsoap.org/soap/security/2000-12"
      SOAP-ENV:actor="some-URI"
      SOAP-ENV:mustUnderstand="1">
      <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        <ds:SignedInfo>
          <ds:CanonicalizationMethod
            Algorithm="http://www.w3.org/TR/2000/CR-xml-c14n-20001026">
          </ds:CanonicalizationMethod>
          <ds:SignatureMethod
Algorithm="http://www.w3.org/2000/09/xmldsig#dsa-sha1"/>
          <ds:Reference URI="#Body">
            <ds:Transforms>
              <ds:Transform Algorithm="http://www.w3.org/TR/2000/CR-xml-c14n-
20001026"/>
            </ds:Transforms>
            <ds:DigestMethod
Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
            <ds:DigestValue>j6lwx3rvEPO0vKtMup4NbeVu8nk=</ds:DigestValue>
          </ds:Reference>
        </ds:SignedInfo>
        <ds:SignatureValue>MC0CFFrVLtRlk=...</ds:SignatureValue>
      </ds:Signature>
    </SOAP-SEC:Signature>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body
    xmlns:SOAP-SEC="http://schemas.xmlsoap.org/soap/security/2000-12"
    SOAP-SEC:id="Body">
    <m:GetLastTradePrice xmlns:m="some-URI">
      <m:symbol>IBM</m:symbol>
    </m:GetLastTradePrice>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

HP – XML Digital Signature API

HP provides an implementation of XML Digital signatures in the form of a library and an out-of-box integration of this library with the HP-SOAP server and client. On the client side, HP-SOAP client API provides the concept of an interceptor interface that can be used in conjunction with the SOAP client API to sign the document. The digital signature library itself can be used with any SOAP client implementation as long as you have access to the SOAP message before sending the request. For more information on the HP-XML Digital Signature API please consult *XML Digital Signatures in HP-SOAP in Appendix A*.

HP-SOAP Client – Interceptor

The HP-SOAP client library defines an interface `com.hp.soap.client.RequestMsgInterceptorIntf` that allows the users to intercept an outgoing SOAP message. This interface defines a *transform* method that allows you to get access to the outgoing message byte array and possibly transform it. The client side integration of digital signatures uses this interface to sign an outgoing message.

SOAP Envelope Signer / Verifier

HP-XML Digital Signature API includes the following utility class:

```
com.hp.security.xml.soap.hpsoap.EnvelopeSignerVerifier
```

This class provides utility methods to sign a SOAP envelope, sign the entire document, verify the signature embedded in a document, etc. Both the client and the server side signature integration use the `EnvelopeSignerVerifier` utility.

Chapter 3

Exposing a Web Service

Introduction

To expose a web service, you need to associate the Java class, EJB or Document Exchange handler with the appropriate adapter and describe your service. This can be accomplished by generating two files:

- a service description file.
- a service configuration file.

In order for the HP-SOAP server to recognize a web service as valid, the service description and configuration files must be contained in a predefined directory structure. To access a web service, you need to know the structure of the URL end point that corresponds to that web service.

Note: The description and configuration files for a web service as well as the directory and URL structure are described in detail in *Chapter 5, Deploying & Configuring Web Services.* Please refer to that chapter to learn more about the files and their content.

HP Web Services platform contains several command-line tools to help you generate and deploy the service description and configuration files.

The HP-SOAP server is equipped with several ready-to-use adapter/handler combinations, making the process of exposing a Java class or stateless session EJB as a web service fairly simple. RPC requests, in fact, do not require you to write any additional server code—you are able to use the adapter/handler combinations included in the HP Web Services platform package. For Document Exchange requests, you need to perform a few more tasks to expose the web service. HP Web Services platform ships with the following adapters:

- rpc-java adapter
- rpc-ejb adapter
- doc-handler adapter
- header adapter

In either case, the key to defining your web service lies in defining the payload processing. Header processing is optional; it does require some additional coding. If you define a header processor but don't define payload processing, your web service will not be exposed.

In this chapter

Processing SOAP Messages

HP Web Services platform defines the way in which SOAP messages are processed. There are several possible processing paths that you can follow.

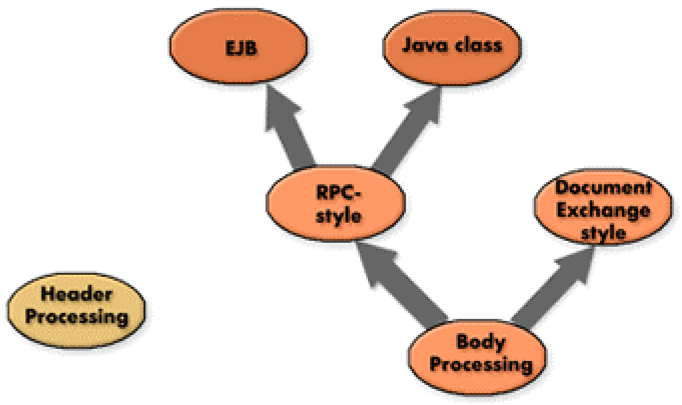


Figure 3-1: Possible processing paths for SOAP messages

Table 3-1 lists the possible paths that can be used to process a SOAP message for a web service.

Table 3-1: Possible combinations for RPC style documents

Path	Header Processing	Payload RPC style processing	
		EJB	Java
1	—	—	✓
2	—	✓	—

Table 3-1: Possible combinations for RPC style documents

Path	Header Processing	Payload RPC style processing	
		EJB	Java
3	✓	—	✓
4	✓	✓	—

The shaded cells in both tables indicate an out-of-box implementation. The unshaded cells require some custom code to complete the implementation.

Table 3-2: Possible combinations for Document Exchange style documents

Path	Header Processing	Payload Doc Exchange-style processing
5	—	✓
6	✓	✓

- **Path 1** in *Table 3-1* illustrates an RPC style web Service implemented as a Java class without header processing. **Path 3** is similar but includes header processing.
- **Path 2** in *Table 3-1* illustrates an RPC style web Service implemented as an EJB without header processing. **Path 4** is similar but includes header processing.
- **Path 5** in *Table 3-2* illustrates Document Exchange-style web service without header processing. **Path 6** is similar but includes header processing.

The following sections of this chapter explain the steps necessary to expose a web service. The complexity of this task depends on your needs and on the structure of your web service. For example, if your web service can be represented by **Path 1**, follow the steps outlined in the section titled *Exposing a Web Service through an RPC Java Adapter*. If instead your web service is represented by **Path 3**, you need to also perform the steps outlined in the *Header Processing* section. If your web service is represented by **Paths 5** or **6**, you need to perform the steps outlined in the section titled *Exposing a Web Service through a Document Exchange Adapter* and for **Path 6** you also need to perform the steps outlined in the *Header Processing* section.

Payload Processing

When you define the way in which the payload is processed, you also create the web service itself. The payload can be processed in one of three general ways: using a Java class, using an EJB, or using a document exchange model.

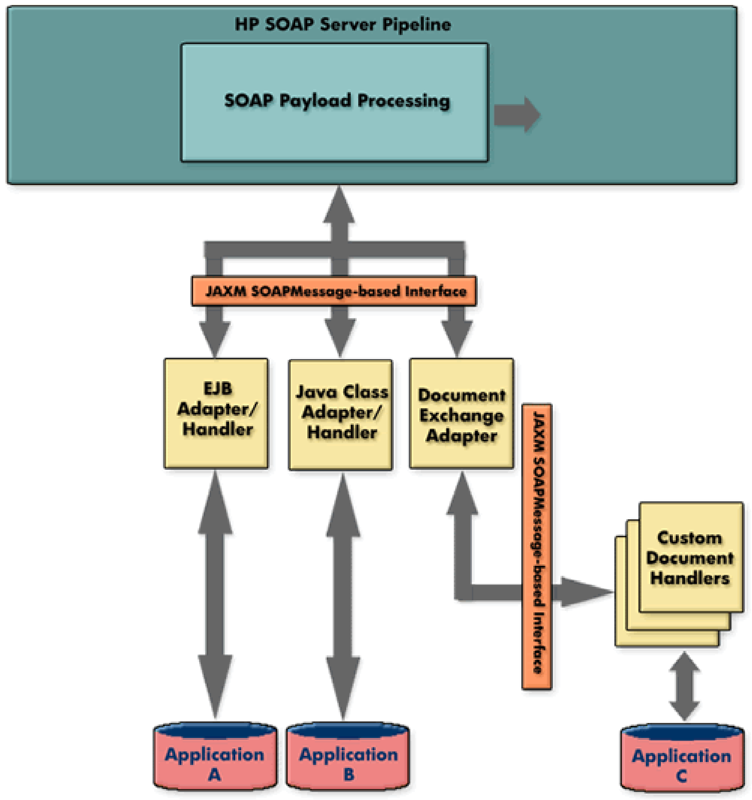


Figure 3-2: Payload pipeline stage

An incoming SOAP request is sequentially processed by pipeline stages. The pipeline stage dynamically loads the appropriate adapter/handler combination on a per-service basis and delegates the processing to the adapter. The adapter then delegates processing to the handler, which knows how to load the web services implementation object and how to process the specific type of request.

Exposing a Web Service through an RPC Java Adapter

To expose a Java class as a web service managed by the HP-SOAP server, you need to perform four steps:

1. Select a unique name for the new Service (<*WebServiceName*>).
2. Create and deploy a service description file (<*WebServiceName*>`.wsdl`).
3. Create and deploy a service configuration file (<*WebServiceName*>`.xml`).
4. Deploy the implementation object.

First, determine a name for your web service (e.g., *MyService*). To automatically generate and deploy the deployment descriptors for your web service, use the ClasstoWebService.bat command line tool located in the `<`*`install_dir`*`>\bin` directory. The syntax for using the tool is:

```
ClassToWebService <JavaClassName> <ServiceEndPoint> <ServiceName>
[<WSDLFile>]
```

Note: All dependencies for the Java class need to be in the classpath.

The ClassToWebService tool creates the directory structure needed to use the web service (`<install_dir>\WebApps\hpws\webservices\<serviceName>`)and copies the WSDL and XML files it creates to that directory.

Note: The WSDL file contents follow WSDL specification. The XML service configuration file is specific to the HP-SOAP server.

For more information about the classToWebService tool, refer to Chapter 6, Tools**.**

The web service configuration file the tool creates maps a web service to a specific adapter, which is dynamically loaded while a request is being processed. The file also maps the web service to the service implementation Java class. For a service implementing a Java class, the **rpc-adapter** is used.

The last step in the process is to deploy the Java class to the HP-SOAP server. There are two possible ways to deploy the class:

- If the Java class implementation is packaged as a JAR, place the implementation JAR in the `<install_dir>\WebApps\hpws\WEB-INF\lib` directory.
- If the Java class is in the form of a class(es), you need to place all implementation classes with their package structure in the `<install_dir>\WebApps\hpws\WEB-INF\classes` directory.

Example Java class-based Service

You can review a sample web service exposing a Java class by looking at the MathService sample shipped with the product. The MathService exposes two methods, add and subtract, with the following signature:

```
public int add(int arg1,int arg2)
public int subtract(int arg1,int arg2)
```

You can inspect the `MathService.wsdl` and `MathService.xml` files located in `<install_dir>\WebApps\hpws\webservices\MathService`.

Listing 3-1: MathService example file (MathService.xml)

```
<?xml version="1.0"?>
 <service-descriptor version="1.0">
   <webservice adapter="rpc-java"  wsdl-validation="false">
              <param name = "lookup-name"
             value = "com.hp.mw.soap.samples.rpc.MathService"/>
   </webservice>
</service-descriptor>
```

Note: For a detailed walk-through of this feature, visit the *Exposing a Java Class as an RPC-style Service* in the Exploring the HP-SOAP Server trailmap included with the platform.

Exposing a Web Service through an RPC EJB Adapter

To expose a stateless session EJB as a web service managed by the HP-SOAP server, you need to perform four steps:

1. Select a unique name for the new Service (`<WebServiceName>`).
2. Create and deploy a service description file (`<WebServiceName>.wsdl`).
3. Create and deploy a service configuration file (`<WebServiceName>.xml`).
4. Deploy the implementation object.

HP-SOAP server can manage the web service implemented by an EJB deployed locally or in a remote location. During design, when exposing an EJB as a web service, you need an EJB client JAR file. For runtime, you also need to have the EJB deployed in the J2EE application server.

Note: For more information about deploying an EJB and creating an EJB client JAR, refer to the documentation for the application server.

The EJB adapter shipped with the HP Web Services platform is configurable. The adapter is configured to recognize application server-specific JNDI lookup information needed to locate the EJB. Currently, the HP Web Services platform allows you to deploy an EJB under HP-AS or JBoss application servers.

Note: The adapter configuration file is named ejbadapter.xconf and is located in <*install_dir*>\WebApps\hpws\webservices\adapters directory.

The process of exposing an EJB as a web service is very similar to the process of exposing a Java class as a Service. Use the command line tool EJBtoWebService.bat located in the `<`*`install_dir`*`>\bin` directory to automatically generate and deploy web service configuration files. The syntax for using the tool is:

```
EJBtoWebService <EJBRemoteInterface> <ServiceEndPoint> <ServiceName>
<EJBWellKnownName> <EJBHomeInterface> <jndi-vendor> <ejbClient>.jar>
[<WSDLFile>]
```

Note: For more information about the EJBtoWebService tool, refer to Chapter 6.

The EJBtoWebService tool creates the directory structure needed to use the web services (`<`*`install_dir`*`>\WebApps\hpws\webservices\<`*`WebServiceName`*`>`) and copies the WSDL and XML files it creates to that directory.

Note: The WSDL file contents follow WSDL specification. The XML service configuration file is specific to the HP-SOAP server.

The web service configuration file for an EJB is more complex than that for a Java class. It contains more EJB specific information needed for loading EJBs and processing a request. For a service implemented by an EJB, the **rpc-ejb** adapter is used.

The last step in the process is to deploy the EJB client JAR to the HP-SOAP server. To deploy it, copy the `<`*`ejbclient`*`>.jar` to the `<`*`install_dir`*`>\WebApps\hpws\WEB-INF\lib` directory.

Example of EJB-based Service

You can review a sample web service exposing an EJB by looking at the LoanCalcService sample web service shipped with HP Web Services platform. The EJB used for this example exposes two public methods:

```
Double getTotalAmt (doubleloan, double interest, int years)
Double getMonthlyPayments (double loan, double interest, int years)
```

You can inspect the `LoanCalcService.WSDL` and `LoanCalcService.XML` files located in `<`*`install_dir`*`>\WebApps\hpws\webservices\LoanCalcService`. To see the Loan Calculator in action, go to: `<HostName>:<PortNumber>\hpws\soap\LoanCalcService`. If you enter the parameter values, the service will return the SOAP message containing the results.

Note: Make sure you deployed LoanCalcEJB in HP-AS application server prior to invoking this request.

Listing 3-2: LoanCalcService example file (LoanCalcService.xml)

```
<?xml version="1.0" encoding="UTF-8" ?>
  <service-descriptor version="1.0">
    <webservice adapter="rpc-ejb" wsdl-validation="false">
      <param name="lookup-name" value="LoanCalc"/>
      <param name="home"
            value="com.hp.mwlabs.wso.examples.ejb.LoanCalcHome"/>
      <param name="remote" value="com.hp.mwlabs.wso.examples.ejb.LoanCalc"/>
      <param name="ejb-type" value="ejb-stateless"/>
      <param name="jndi-vendor" value="HP"/>
    </webservice>
</service-descriptor>
```

The information in the web service configuration file is used to dynamically load an adapter and then look up the required EJB and perform the required RPC operations.

Note: For a detailed walk-through of this feature, visit the *Exposing an EJB as an RPC-style Service* in the **Exploring the HP-SOAP Server** trailmap included with the platform.

Exposing a Web Service through a Document Exchange Adapter

For RPC requests, the HP Web Service platform provides you with the full solution required to process messages. For Document Exchange requests, the HP Web Services platform requires additional steps in order to process message. For Document Exchange, then, you need to create a document handler that defines how to process messages. The document handler you write needs to work with the Document Exchange adapter (**doc-handler**) provided with the Web Services platform for seamless integration of Document Exchange requests.

Writing a Document Handler

A document handler is required for processing Document Exchange requests through the HP-SOAP server. The current version of the HP-SOAP server is designed to processes synchronous and asynchronous SOAP message requests.

To create a custom handler, you need to complete two steps:

1. Create an implementation object for the handler.
2. Map the custom handler to the web service of interest.

Creating an Implementation Object

HP Web Services platform uses both an adapter and a handler to process Document Exchange requests. In general, an adapter is a component able to load a document handler, while the handler is able process a message. The Web Services platform provides a generic adapter for Document Exchange called **doc-handler adapter**. There is one-to-one correspondence between an *adapter* and a *handler*. Basically, an adapter acquires an instance of a handler. When the request is dispatched, it extracts relevant JAXM messages from the context, initializes the web service context appropriately, and invokes the *handler*.

Developing a Document Handler

If you are developing a new handler to process the incoming request, you should place most of your logic in the handler. At the very least, the handler must implement:

```
com.hp.mw.soap.handler.SOAPMessageHandler
```

This interface contains three methods:

```
public void setWebServiceContext ( WebServiceContext context ) throws Exception;
public WebServiceContext getWebServiceContext ( );
public void processMessage( SOAPMessage input, SOAPMessage output ) throws  Exception;
```

For both synchronous and asynchronous requests, you need to implement the *processMessage* method. The document handler always receives incoming messages and typically responds by processing them and sending the result as the outgoing message. For synchronous messages, write the response messages to the *output* SOAPMessage object. The web service context is passed to the handler through the *setWebServiceContext* method.

The EchoDocument Web Service example demonstrates a custom handler working together with the Document Exchange adapter. You can use the implementation of:

`com.hp.mw.pipeline.soap.samples.doc.EchoDocument.java`

as a prototype for the custom handler.

Compiling the Handler

To compile the new custom handler, you will need the following JAR files as dependencies:

- `hpws.jar` for SOAPMessageHandler definition
- `jaxm.jar` for SOAPMessage

The JAR files are located in the `<`*`install_dir`*`>\WebApps\hpws\webservices\WEB-INF\lib` directory.

Note: Please refer to the JAXM documentation at java.sun.com/xml/jaxm to learn more about the JAXM API.

Note: If you are using NetBeans to compile, you may need to include the avalon-excalibur-4.0.jar and avalon-framework-4.0.jar files.

Deploying the Handler

The compiled handler code needs to be part of the HP-SOAP Server application. The simplest way to achieve this is to deploy the classes to the `<`*`install_dir`*`>\WebApps\hpws\webservices\WEB-INF\classes` directory.

Note: All dependencies for the Java class need to be in the classpath.

Mapping the Custom Handler to the Adapter

The next step in setting up the custom handler is to map it to the appropriate web service. The Document Exchange adapter allows the server to dynamically load the handler for the specified service and delegate further processing to that handler.

To automatically generate and deploy the deployment descriptors for your web service, use the HandlerWebService.bat command-line tool located in the `<`*`install_dir`*`>\bin` directory. The syntax for using the tool is:

```
HandlerToWebService.bat <Handler Class Name> <Service Endpoint
URL> <service name> <WSDL file>
```

To use the command-line tool, you ***must*** first create the service description file `<`*`WebServiceName`*`>.wsdl.` The tools that come with the platform cannot generate the WSDL file for Document Exchange-style processing. When passing parameters for `HandlerToWebService.bat`, the `<WSDL file>` parameter is the fully qualified path of the file you created.

The `HandlerToWebService` tool creates the directory structure needed to use the web service (`<install_dir>\WebApps\hpws\webservices\<serviceName>`) and copies the WSDL and XML files it creates to that directory.

Note: The WSDL file contents follows WSDL specification. The XML service configuration file is specific to the HP-SOAP server.

For more information about the HandlerToWebService tool, refer to Chapter 6**.**

You can inspect the EchoDocService`.wsdl` and EchoDocService`.xml` files located in `<install_dir>\WebApps\hpws\webservices\`EchoDocService

Listing 3-3 shows example code for registering a custom handler named EchoDocument and the doc-handler adapter. The handler is defined in the web service configuration file.

Listing 3-3: Sample code for defining a custom handler in the *<WebServiceName>*.xml file

```
<?xml version="1.0"?>
  <service-descriptor version="1.0">
     <webservice adapter="doc-handler“  wsdl-validation="false">
        <param name=”lookup-name”
               value=”com.hp.mw.pipeline.soap.samples.doc.EchoDocument”/>
     </webservice>
  </service-descriptor>
```

Note: For a detailed walk-through of this feature, visit the Developing your own Document Exchange-style Service in the Exploring the HP-SOAP Server trailmap included with the platform.

Header Processing

If your web service requires header processing, you need to write Java code that encapsulates the logic necessary to process the header. The header processor you need may not be included in the HP Web Services platform and you will need to write the processor yourself. This section explains how to create a header processor.

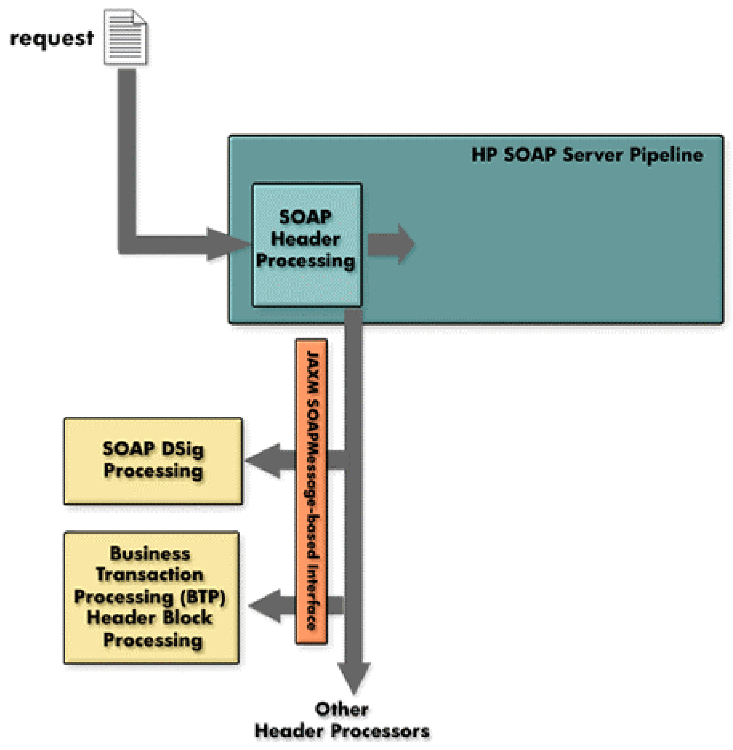


Figure 3-3: SOAP Header processing

To create a header processor, you need to complete three steps:

1. Determine the types of header blocks needed for the Service(s).
2. Implement the header processor.
3. Register the header processor and deploy.

Determine the Types of Header Blocks Needed for Your Service(s)

Before you develop a header processor, you need to determine what types of header blocks your service(s) requires. A header block is identified by its local name and a URI; it appears as a child node of the SOAP Header element. You need to create a header processor for each type of header block so that all services that use a particular type of header block can share a single header processor. For example, if you intend to develop a Digital Signature header processor, the header block in the SOAP message may look similar to the example in *Listing 3-4*.

Listing 3-4: Example of a header block in a SOAP message

```
<SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header>
     <SOAP-SEC:Signature
         xmlns:SOAP-SEC=
            "http://schemas.xmlsoap.org/soap/security/2000-12"
         SOAP-ENV:mustUnderstand="1">
       <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
          <ds:SignedInfo>
             <ds:CanonicalizationMethod
                Algorithm="http://www.w3.org/TR/2000/CR-xml-c14n-20001026"/>
            <ds:SignatureMethod
```

```
                     Algorithm="http://www.w3.org/2000/09/xmldsig#dsa-sha1"/>
                <ds:Reference URI="#Body">
                  <ds:Transforms>
                   <ds:Transform
                     Algorithm="http://www.w3.org/TR/2000/CR-xml-c14n-20001026"/>
                  </ds:Transforms>
                  <ds:DigestMethod
                     Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
                  <ds:DigestValue>j6lwx3rvEPO0vKtMup4NbeVu8nk=</ds:DigestValue>
                 </ds:Reference>
              </ds:SignedInfo>
              <ds:SignatureValue>MC0CFFrVLtRlk=...</ds:SignatureValue>
              <ds:KeyInfo>
                <ds:KeyName>Michael</ds:KeyName>
              </ds:KeyInfo>
           </ds:Signature>
         </SOAP-SEC:Signature>
      </SOAP-ENV:Header>
      <SOAP-ENV:Body
         xmlns:SOAP-SEC=
           "http://schemas.xmlsoap.org /soap/security/2000-12" SOAP-
    SEC:id="Body">
        ………
      </SOAP-ENV:Body>
    </SOAP-ENV:Envelope>
```

In this example, the local name of the SOAP-SEC header block is *Signature* and the URI is *http://schemas.xmlsoap.org/soap/security/2000-12*.

To develop and register a header processor for a particular type of header block, make note of the local name and URI. You will need this when configuring the HP-SOAP server.

In some cases, you may have a group of related header blocks. In such situations, you may wish to handle the entire group through a single header processor. The HP-SOAP server allows you to group related header blocks using a `<header-group>` element in the header adapter configuration file.

Listing 3-5: Sample header-group element in the header-adapter.xconf file

```
<header-group className="com.hp.mwlabs.soap.security.SOAPDSigProcessor">
             <header-block name="Signature"
uri="http://www.w3.org/2000/09/xmldsig#"/>
</header-group>
```

Implement the Header Processor

All header processors must implement the following Java interface:

`com.hp.mw.soap.HeaderElementProcessor`

This interface contains only one method:

```
public void processHeaderGroup(ArrayList elements,
            SOAPMessage input,
            SOAPMessage output,
            WebServiceContext context) throws SoapFaultException;
```

The API uses the JAXM API to provide standards-based access to the incoming SOAP message.

Note: Please refer to the JAXM documentation at java.sun.com/xml/jaxm to learn more about the JAXM API.

Create a Java class that implements this method and place your header processing logic in the body of the processHeaderGroup method. The first argument, *elements*, represents a collection of `javax.xml.soap.SOAPHeaderElement` objects, one per header block defined in the header configuration file (for an example of the complete header adapter file, see *Chapter 2, Architecture*). If you intend to process a single header block, you still need to create a group that includes a single header block definition, where *elements* is a single instance of the `javax.xml.soap.SOAPHeaderElement` object. The header block has already been extracted using the API `extractHeaderBlocks` (String actor) on the *input* `javax.xml.soap.SOAPHeaderElement` object.

When you write your header processor logic, you can access not only the block in which you are interested but also both the *input* and *output* messages. While the JAXM API allows you to manipulate these message objects, a header processor is not expected to modify the *input* message.

For example, if you were creating a header processor for Digital signature, the header processor would be tasked to extract and verify the embedded digital signature.

Defining the SOAP-actor Attribute for the SOAP Node

A SOAP header block may contain an optional `<SOAP-actor>` attribute that identifies the target SOAP node for which the header block is intended. If no actor is specified, the final recipient of the SOAP message is expected to process the block. An additional `<mustUnderstand>` attribute indicates that the designated SOAP node must process the block.

Note: For more information about the attributes required by SOAP, refer to the SOAP specification at http://www.w3.org/TR/SOAP/.

The deployment unit in the HP-SOAP server, a WAR file, represents the notion of a SOAP node. The HP-SOAP server allows you to configure a SOAP actor for either the entire WAR (and the services that it contains) or at a specific Service level. However, if you define a SOAP actor URI value (e.g., `http://mysoapserver/Order Processing`), any clients invoking the Order Processing Services must supply the `<SOAP-actor>` attribute in the SOAP request. Hence, the expected setting for the `<SOAP-actor>` attribute is an empty string.

Register the Header Processor and Deploy

The final step is to register your header processor within the context of the deployment WAR file (or within the context of an expanded WAR file on the file system). All header processors should be registered through the following configuration file:

```
<install_dir>\WebApps\hpws\webservices\adapters\ header-adapter.xconf
```

Specify a complete path to your header processor Java class via the *className* attribute of the *header-group* element as shown:

```
<header-group className="com.hp.mwlabs.soap.handler.SOAPHeaderEchoProcessor">
```

Deploy your directory or WAR file. For more information about configuring and deploying web services, refer to *Chapter 5* of this manual.

Using Digital Signatures in HP-SOAP

If you need to process digital signatures (Dsig), the HP Web Services platform includes a Dsig Header Processor for your use. To use digital signatures in the HP-SOAP server, you need to complete two steps:

1. Sign the document on the client side.
2. Configure HP-SOAP to verify signed SOAP messages.

Signing the Document on the Client Side

HP-SOAP ships with a client-side interceptor (`com.hp.security.xml.soap.hpsoap.RequestMsgSignerIceptor`) that signs just the Body portion of a SOAP message using the Envelope Signer/Verifier utility. If you are developing a client application using the HP-SOAP client API, you simply need to register the interceptor with the `SoapClient` object:

```
m_soapClient.setRequestMsgInterceptor(
         new RequestMsgSignerIceptor() );
```

where `m_soapClient` is an instance of `com.hp.soap.client.SoapClient` class.

For a complete example of a SOAP client using the interceptor, see *Chapter 2, Architecture* and look at `com.hp.security.xml.soap.hpsoap.EchoStringClient`.

Note: You could also sign other portions of the SOAP message using the HP-XML Digital Signature API. In such situations, you may want to implement your own interceptor and use the HP-XML Digital Signature API directly. Source code for both the RequestMsgSignerIceptor and EnvelopeSignerVerifier is included in the <*install_dir*>/samples directory.

Configuring HP-SOAP to Verify Signed SOAP Messages

HP-SOAP also ships with a header processor (`com.hp.security.xml.soap.hpsoap.SOAPSignatureProcessor`). See the previous section for more information about header processors in the HP-SOAP server look at `com.hp.security.xml.soap.hpsoap.EchoStringClient` and see *Chapter 2, Architecture*. To enable digital signature verification, you need to configure the HP-SOAP server through the header adapter configuration file:

```
<install dir>\WebApps\hpws\webservices\adapters\ header-adapter.xconf
```

To process signature header blocks that use the default setting for the SOAP Actor attribute (`http://schemas.xmlsoap.org/soap/actor/next`), you simply need to include the following entry in your `header-adapter.xconf` file.

Listing 3-6: Sample header-group element in the header-adapter.xconf file

```
<!-- ************************* -->
<!-- Global SOAP actor         -->
<!-- ************************* -->
   <soap-actor name=""/>
   <header-group
      className=”com.hp.security.xml.soap.hpsoap.SOAPSignatureProcessor ">
           <header-block name="Signature"
            uri="http://www.w3.org/2000/09/xmldsig#"/>
   </header-group>
```

The empty string value for the name attribute in `<soap-actor name=""/>` is interpreted by the SOAP server as the default SOAP actor (`http://schemas.xmlsoap.org/soap/actor/next`).

Chapter 4

Data Types in the HP Web Services Platform

Introduction

The serialization and deserialization of data between an incoming SOAP message and its target web service is executed by the HP Web Services platform data type system. Support for the various input and output data types of a web service are implemented through the `SoapType` base class and all of its derived classes. Each data type supported in the HP Web Services platform is implemented by its own SoapType derived class (`StringType, IntType, FloatType`, etc.).

The `SoapType` classes can also be configured to use converters and/or formatters to manipulate the data in the incoming or outgoing SOAP message. An input converter would be used to change the data type of a parameter in the incoming SOAP message before the parameter is passed to the web service, while an output converter would be used to change the return value from the web service to a different data type before being returned in the outgoing SOAP message. A formatter would be used to manipulate a value in the incoming or outgoing SOAP message without changing its data type.

In this chapter

Data Type Support

The HP Web Services platform supports various SOAP encoding schemes through a type abstraction. This includes an abstract base class, `SoapType,` which amongst other methods, includes the following:

Listing 4-1: Serialization/Deserialization related methods

```
String serialize();
Object deserialize();
Element serializeAsElement();
```

Listing 4-2: Meta data methods

```
Class getPrimitveClass();
String getTagName();
String getXSIType();
```

Listing 4-3: Type converters and formatters related methods

```
public void setConverter( TypeConverter converter );
public void setFormatter( Formatter formatter );
```

The HP-SOAP server enables you to convert data types in the incoming and outgoing messages into different data types. Type converters are used to deserialize data from XML into Java objects and to serialize the Java objects back into XML. Value formatters are used to modify the value associated with a parameter or a return value.

Supported Data Types

There is a consortium of SOAP server vendors that determines a set of data types to be implemented. The primary objective is to ensure a certain level of interoperability amongst various SOAP implementations. These data types cover all the native types specified in the SOAP encoding as well as some additional types. The HP-SOAP implementation closely tracks these data type requirements. The following list shows the current level of data type support.

Note: For more information on interoperability and data types go to www.whitemesa.com.

Table 4-1: Supported data types

• String	• StringArray	• 2DstringArray(Row-majored, non-ragged)
• Integer	• IntegerArray	• Float
• FloatArray	• Void	• Base64
• HexBinary	• Date	• Decimal
• Boolean	• Struct	• StructArray
• NestedStruct	• NestedArray	

Type Management

The HP-SOAP server facilitates type management at three different levels.

- A global type registry - global to a deployment unit (currently a WAR file)
- At a specific service level
- At a specific WSDL operation level

Type Management at the Global Level – Global Type Registry (soap-type.xml)

The Global Type Registry (`soap-type.xml`) contains mappings for all type-related configuration for the HP-SOAP server.

The root element is `<soap-type-registry>` and it contains the following child elements:

- `<type-maping>`
- `<type-formatters>`
- `<type-converters>`
- `<output-converters>`

`The <type-maping>` child elements contain the mappings for soap-type/java-class types to the deserialization/serialization implementations. These types are specified as attributes of the `<type-mapings>` element.

The `<type-formatters>` element contains the global Input/Output formatter registry. Configured formatters format the text value of an XML element. Input formatters modify the text values before they are deserialized into a Java object. Output formatters reformat the text values after they have been serialized from a Java object back into an XML element. Formatters always act upon soap-type XML elements**.**

The `<type-converters>` element contains the global Input/Output converter registry. Configured converters convert the XML element to a Java object or another XML element. Input converters take effect when deserializing the input soap-type XML element (after formatting). Output converters take effect when serializing the output Java object (before formatting).

By default, two converters are configured at the global level:

- **`org.apache.cocoon.processing.soap.conversion.ArrayRowMajorConverter`** - Transforms the incoming SOAP encoding array, which is in row-major, to a Java array object, and vice versa.
- **`org.apache.cocoon.processing.soap.conversion.CastorConverter`** -Transforms all complex types that are not addressed by the type-mapings configuration.

The `<output-converters>` element is different from `<type-converters>` in that it is only used if the data type of the output message specified on the WSDL is different from the type attribute on the serialized XML element of the returned Java object. For example, by default `<type-maping>` byte array byte is serialized in `Base64Type`, but the `echoHexBinary` on the WSDL specifies output in a Hexadecimal encoded byte array. Thus, if a parameter in an output message is hexBinary, the `<output-converter>` named `HexBinaryConverter` is used to serialize the output as a hexadecimal rather than as `Base64Type`. The intent is to provide some level of fault tolerance.

Type Management at the Service Level (WebServiceName.xml)

Each web service has a configuration file, `<`*`WebServiceName`*`>.xml`. This file enables you to register converters and formatters for a specific web service. If you specify a converter or formatter at this level, it overrides the converter or formatter defined in the Global Type Registry. To view an example of a web service registration file with a type converter and a value formatter specified at the service level, look at `<`*`install_dir`*`>/docs/samples/service-example.xml`.

Type Management at the WSDL Operation level (WebServiceName.wsdl)

The service configuration file also allows you to register type converters and formatters at the operation level. If a type converter or value formatter is specified at this level, it overrides the corresponding converter or formatter at both the global and service level. To view an example of a web service registration file with a type converter and a value formatter specified at the operation level, look at `<`*`install_dir`*`>`/docs/samples/`service-example.xml`.

Complex Type Support

If a service exposes complex data types, the server needs to support an extensible type system to address serialization and deserialization of such types. The HP-SOAP server is designed to address the following capabilities:

- Ability to map an XML schema type (global/service/operation level) to an existing Java class.
- Generate Java classes from a given XML schema.

The HP-SOAP Server and the Exolab Castor Project

The HP-SOAP server uses Exolab's Castor project to deserialize/serialize complex data types between Java objects and XML elements. The server defines the `CastorConverter` class to serialize and deserialize the data. The `CastorConverter` may be used to serialize either existing Java classes or schema data types.

This converter depends on configuration information in the `soap-type.xml` file to convert the data correctly. There must be a `<converter>` element that points to the `CastorConverter` class and contains property references to the map files that Castor uses to make its type conversions. An example of this is shown in the XML fragment below:

Listing 4-4: XML file showing property referencess

```
<type-converters>
  .................
    <converter name="struct"
     class="org.apache.cocoon.processing.soap.conversion.CastorConverter">
         <property name="http://soapinterop.org/xsd:SOAPStruct:map"
          value="soapstruct.xml" />
```

```
            <property name="http://soapinterop.org/xsd:SOAPStructStruct:class"
             value="org.apache.cocoon.processing.soap.types.SOAPStructStruct" />
            <property name="http://soapinterop.org/xsd:SOAPArrayStruct:map"
             value="soapstruct.xml" />
         </converter>
      ..................
   </type-converters>
```

The Castor Converter in Action

After the configuration has been edited, the `CastorConverter` is ready to be used on the new data type. When the HP-SOAP server receives an XML request that contains a custom or complex data type, the `CastorConverter` is created to deserialize (deserialize) this data to its Java class representation. The converter checks its property settings to determine if a map file name has been specified for this data type. The converter always searches for a map file first, because it can be used to override the default class mappings. If a map file name exists, it is used to deserialize the XML node. If the map file name is not specified, the converter checks its properties for the class name. If the class name exists, it is used for the deserializeing. If neither property exists, an exception is thrown. Otherwise, the newly created Java class object is returned from the converter and the payload can be passed to the web service to be processed.

After the payload is processed, a custom data object is returned and the `CastorConverter` is used to serialize this data to its XML node representation. The converter looks in its properties for the map file name referencing the class to use. If it exists, the map file is used to serialize the class. If the map file name does not exist, the converter attempts to use Castor's default serializeing functionality on the class to generate the XML node. The new node is returned from the converter to the caller.

Deserializing from a Text Value to a Java Object

Deserialization is the process of converting the incoming parameter to a Java object and initializing the object with the parameter value. The default sequence is to identify the xsi:type of a parameter in the incoming message and creating an instance of the appropriate derived class (of the abstract SoapType class). To complete the deserialization of a parameter, the deserialize() method is invoked. The xsi:type attribute is an optional attribute as per the SOAP encoding rules. Hence, HP-SOAP server uses the WSDL file to determine the data type if the incoming parameter does not contain this attribute.

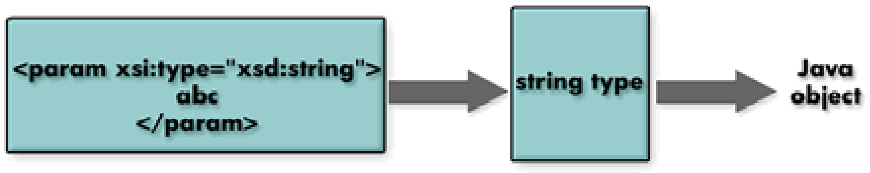
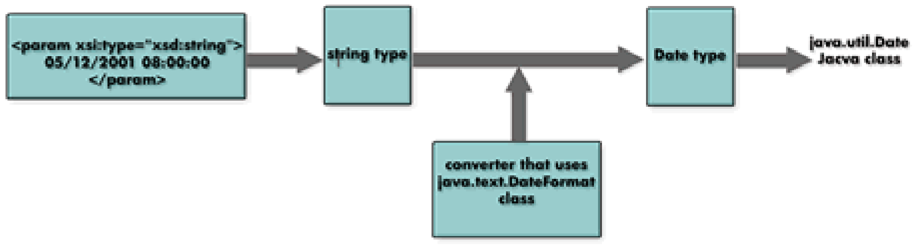


Figure 4-1: Default deserialization sequence

Deserializing with a Converter

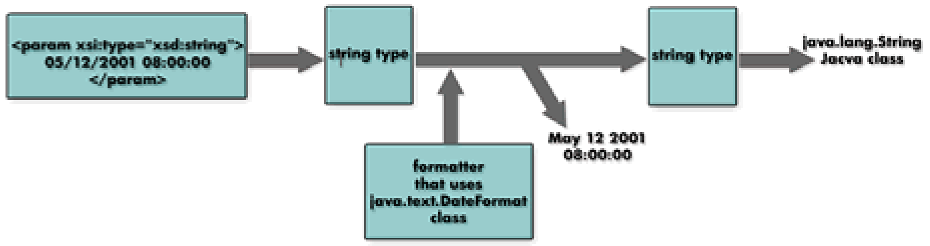


<param xsi:type="xsd:string"> 05/12/2001 08:00:00 </param>
string type
Date type
java.util.Date Jacva class
converter that uses java.text.DateFormat class

Figure 4-2: Deserialization sequence with a converter

Figure 4-2 illustrates how one could use a converter to create a Java class that best represents the underlying parameter value. In this example, a date value is encoded as a `xsi:string` type in the SOAP message. However, if the Java method that implements the operation takes a `java.uil.Date` type, the appropriate conversion is needed. In the HP-SOAP server, this is simply a matter of registering a type converter that performs the conversion from a value to `java.util.Date` type.

Deserializing with a Formatter



<param xsi:type="xsd:string"> 05/12/2001 08:00:00 </param>
string type
string type
java.lang.String Jacva class
May 12 2001 08:00:00
formatter that uses java.text.DateFormat class

Figure 4-3: Deserialization sequence with a formatter

Figure 4-3 shows the use of a formatter to convert a date value from `mm/dd/yyyy` format to a more verbose format, as shown.

Serializing from a Java Object to a Text Value

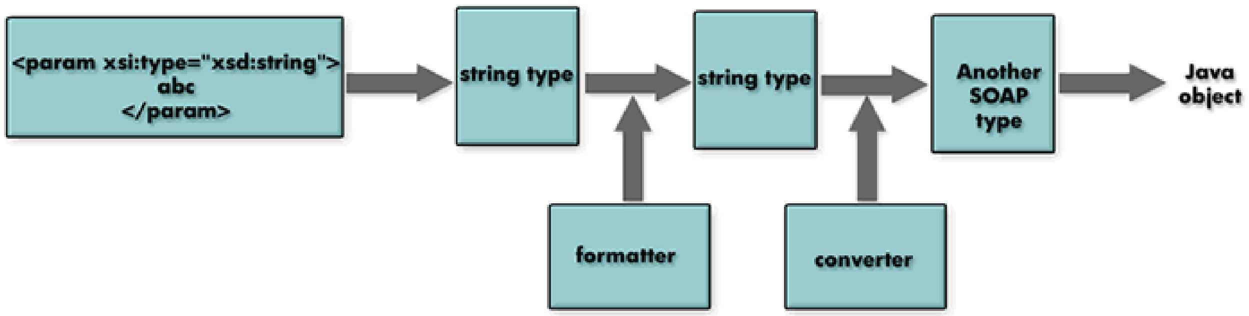


Figure 4-4: Serialization sequence with a converter and a formatter

The steps involved in the serialization sequence are the reverse of deserialization. If the Java object does not represent a native type, the initial type is determined based on the Java object or WSDL. In some cases, conversion may be needed in order to comply with the type specified in WSDL. You may also specify a formatter to reformat the return value.

Overriding the Service Configuration

As mentioned earlier in this guide, the HP-SOAP server allows you to configure formatters and converters at the global, service, and operation level. The following is an example of a service configuration:

Listing 4-5: Serialization/deserialization-related methods

```
<webservice-descriptor version="1.0">
  <webservice adapter="rpc-adapter"  wsdl-validation="false">
  <param name="lookup-name" value
="com.hp.mw.pipeline.soaprpc.services.MyService"/>
  <soap-type>
    <type-formatters>
      <formatter name="string"
class="org.apache.cocoon.soap.formation.StringFormatter"/>
    </type-formatters>
    <type-converters>
      <converter name="string"
class="org.apache.cocoon.processing.soap.conversion.MyConverter"/>
    </type-converters>
    <messages>
<message name="someOperationRequest">
   <formatter name="string"
      class="org.apache.cocoon.processing.soap.formation.StringFormatter"/>
   <converter name="string"
      class="org.apache.cocoon.processing.soap.conversion.MyConverter"/>
           </message>
           <message name="someOperationResponse">
              <formatter name="java.lang.String"

class="org.apache.cocoon.processing.soap.formation.StringFormatter"/>
              <converter name="java.lang.String"

class="org.apache.cocoon.processing.soap.conversion.MyConverter"/>
           </message>
   </soap-type>
 </webservice>
</webservice-descriptor>
```

For each specific service, you can configure formatters and converters at two levels: at the inidividual service level and for a specific message.

Configuring a Formatter or Converter at the Service Level

You configure formatters and converters by adding the `<type-formatters>` element and the `<type-converters>` element under `<soap-type>` element, which is a child of root element `<webservice>`. This configuration applies to all operations in this service. Using the sample configuration file above, any incoming parameter with a string type is formatted by `StringFormatter`, and then deserialized by `MyConverter`. Any returned Java object, which is an instance of `String`, is serialized by `MyConverter` and then formatted by `StringFormatter`.

Configuring a Formatter or Converter at the Message Level

At the message level, a `<soap-type>` element may have `<messages>` child elements. Furthermore, `<messages>` elements may have `<message>` child elements that are required to have the same name as a message declared in the WSDL. Under the `<message>` element formatter and converter, elements can be configured that apply to this message only. For example, `someOperationRequest` may be an input message of some operation in this service. If any of the parameter (part) is `string` type, it is formatted by `StringFormatter` and then deserialized by `MyConverter`.

While the service-level configuration of formatters and converters overrides the global-level configurations, message-level configuration overrides service level configuration. These overrides include any properties that are a part of the service or message elements. For example, suppose you configured a `StringFormatter` with a property `name=“prefix”` and a `value=“global"`, at the global level; and at the service level, you also have a `StringFormatter` with a property `name=“prefix”` and a `value=“service”`. Any incoming parameter for this service of type `string` is formatted by the `StringFormatter` using the prefix property with value of `“service”`, rather than of `“global”`. Any other service not having a service level formatter configuration would use the `StringFormatter` with the prefix property having the value “global”.

Invoking the Service

Both EJB and Java class adapters use a common set of classes to evaluate the RPC body, create an instance of a Java class and invoke the appropriate method corresponding to the operation or method specified in the SOAP message. The adapters rely on these helper classes to provide metadata, as well as serialization and deserialization capabilities. The `SoapRPCBody` class, in the `org.apache.cocoon.processing.soap` package, and other classes located in the subpackages (config, types, etc.,) make up the helper classes used by the RPC adapters.

When a SOAP message arrives, adapters extract the body portion of the message and initialize the `SoapRPCBody` class. The `SoapRPCBody` class extracts the parameters and makes an attempt to resolve the operation/method. If the incoming message contains the data type information, the method resolution is relatively straightforward. However, if the data type information is missing (partially or entirely), this information is looked up in the WSDL file.

Method Overloading

Method overloading occurs when two operations are allowed to have the same name but different input messages. These two operations can be mapped to two methods (of this service’s Java implementation) with the same name but different input parameters. Merging the data type information in the SOAP message with type information from the WSDL file allows the HP-SOAP Server to support method overloading.

However, in the ROUND 2 SOAP Interoperability Tests Specification, explicit typing information is not required for incoming messages. This may cause ambiguity for the method lookup (operation resolution). For example, consider the following RPC request:

```
<SOAP-ENV:Body>
   <echo>
       <input>1234</input>
   </echo>
</SOAP-ENV:Body>
```

Now, consider a corresponding service implementation that has the following methods, and **both** methods are exposed through the WSDL:

```
public int echo( int i )
```

```
{
      return i;
}

public String echo( String s )
{
      return s;
}
```

The service implementation has two methods named echo; one is an integer and the other is a string. When the SOAP request comes in, the server recognizes the `<echo>` tag and must decide whether to treat the data in this parameter as an integer or a string. So, there is an ambiguity in resolving the method to be invoked, even after using the information in WSDL. The incoming parameter `1234` can be deserialized to either an integer or a string. The HP-SOAP server resolves this problem by sequentially evaluating the parameters and the methods and selecting the first match.

In contrast, the request below is not ambiguous. The text '`ABCD`' must be a string.

```
<SOAP-ENV:Body>
   <echo>
          <input>ABCD</input>
   </echo>
</SOAP-ENV:Body>
```

In this case, the server will first try to deserialize the string as an integer and fail. Then it will deserialize it as a string and succeed.

The above example highlights the fact that in order to avoid ambiguity and to ensure an efficient method lookup, clients should be required to supply type information in SOAP requests. One way to enforce this is through WSDL validation.

WSDL Validation

In the web service configuration file, there is an attribute called `wsdl-validation`.

Listing 4-6: wsdl-validation in web service configuration file

```
<webservice-descriptor version="1.0">

  <webservice adapter="rpc-java"  wsdl-validation="false">

…………………

</webservice>
```

The default value for this attribute is false. If this value is set to true, explicit data type information is required for this service. If set to true and a request is submitted without data type information in the parameters, the request will be rejected. This can avoid possible ambiguity and force strong type checking.

In addition, this validation requires that the namespace URI specified in the incoming message be exactly the same as the one declared in WSDL. If a service has a namespace declaration, `xmlns:xsd=http://www.w3.org/2001/XMLSchema`, any incoming request with a different namespace, such as `xmlns:xsd=http://www.w3.org/1999/XMLSchema`, will be rejected.

Using Castor for Complex Data Types in the HP-SOAP Server

The HP SOAP server uses Exolab's Castor project to serialize data between Java objects and XML elements. The server defines the CastorConverter class to serialize and deserialize the data. The CastorConverter may be used to serialize either existing Java classes or schema data types. This converter depends on configuration information in the soap-type.xml file to serialize and deserialize the data correctly. The following two sections explain how to configure the CastorConverter.

Mapping XML Tags to Fields in an Existing Java Class

Note: For a detailed walk-through of this feature, visit the *Using complex data types defined by an existing Java Class* in the **Complex Data Type** trailmap included with the platform.

To use the CastorConverter for an existing Java class, a Castor map file must be created for the class. Castor provides a MappingTool utility class that will generate a basic map file for a specified Java class. Run the mapping tool using the following command:

```
java org.exolab.castor.tools.MappingTool -i class_name -o output_map_file
```

This map file specifies the mapping between the XML tags and the fields in the user's Java class. The user should closely examine the generated map file to verify that the XML tags created are the tags you want the mapping file to require of the incoming SOAP message.

Note: These values can be edited, but users are strongly encouraged to read the Castor documentation about mapping files before attempting to edit this file by hand. You can find Castor documentation at: http://www.castor.org/xml-mapping.html.

Once this map file is ready, it must be associated with the class in the soap-type.xml file.

1. Locate the `<type-converters>` tag in the `soap-type.xml` file.
2. Within the `<type-converters>` tag, attempt to locate the `<converter>` tag with the name attribute set to *`struct.`*
3. If that `<converter>` tag does not exist, create it as follows:

```
<converter name=”struct”
   class="org.apache.cocoon.processing.soap.conversion.CastorConverter">
</converter>
```

4. For each class to configure, add a `<property>` tag to the `<converter>` tag specified above. Set the name attribute of the tag to the namespace qualified Java class name, concatenated with the *:map* string. Set the value property to the map file name.
5. Copy the map file to `<`*`install_dir`*`>\WebApps\hpws`.
6. Copy the class to `<`*`install_dir`*`>\WebApps\hpws\WEB-INF\classes`.

Listing 4-7: Sample soap-type.xml file showing how to associate the map file to the Java class

```
<type-converters>
      <converter name="struct"
          class="org.apache.cocoon.processing.soap.conversion.CastorConverter">
          <property name = "http://soapinterop.org/xsd:SOAPStruct:map"
                    value = "soapstruct.xml" />
          …..…
          </converter>
</type-converters>
```

7. Copy the generated classes to `<`*`install_dir`*`>/WebApps/hpws/WEB-INF/classes.`

Generating Classes for Schema Types

Note: For a detailed walk-through of this feature, visit the *Using complex data types defined by an XML Schema* in the **Complex Data Type** trailmap included with the platform.

To use the CastorConverter for a schema-defined type, the user must first generate a Java class to represent this type. Castor provides the SourceGenerator utility class to accomplish this goal. Run the SourceGenerator using the following command:

```
java org.exolab.castor.builder.SourceGenerator -i schema_file  -f -types j2 -
package output_package_name
```

The SourceGenerator will produce two classes for each schema type: the descriptor class and the actual implementation class. Both of these classes must be made visible to the HP-SOAP server in order for the CastorConverter to function properly.

Once the class files have been generated, they must be associated with the schema type in the `soap-type.xml` file. Only the implementation classes must be specified in the configuration file. The descriptor classes are only needed for the actual serializeing and deserializeing.

1. Locate the `<type-converters>` tag in the soap-type.xml file.
2. Within the `<type-converters>` tag, attempt to locate the `<converter>` tag with the name attribute set to *`struct.`*
3. If that `<converter>` tag does not exist, create it as follows:
   ```
   <converter name=”struct”
   class="org.apache.cocoon.processing.soap.conversion.CastorConverter">
   </converter>
   ```
4. For each class to configure, add a `<property>` tag to the `<converter>` tag specified above. Set the name attribute of the tag to the namespace qualified java class name, concatenated with the *:class* string. Set the value property to the fully qualified java class name.

   The completed soap-type.xml `<type-converters>` section should look similar to this:

Listing 4-8: Sample soap-type.xml file showing how to associate a class file with the schema type.

```
<type-converters>
   <converter name="struct"
    class="org.apache.cocoon.processing.soap.conversion.CastorConverter">
       <property name ="http://soapinterop.org/xsd:SOAPStruct:class"
        value = "org.apache.cocoon.processing.soap.types.SOAPStruct" />
           .. …
   </converter>
</type-converters>
```

Chapter 5

Deploying & Configuring Web Services

Introduction

In order to use a web service in an application, you need to deploy it and configure any features that may be needed. There are several configuration files used to configure HP web services components.

Note: The keywords in the configuration files shown in this chapter are case sensitive. Comments are included between the <!-- *comment*--> tags. Values relative to the file system are relative to the working directory from which the server is launched.

In this chapter

Deploying Web Services

For HP Web Services platform, a web service is defined by its unique name and by two configuration files (an XML file and a WSDL file) which must bear the same name as the service. For instance, the MathService service must have a MathService.xml file and a MathService.wsdl file. These files provide information about the adapter(s) and handler(s) used by the service and also define other aspects of the service.

If you have your own Java class, EJB or Document Exchange handler and want to deploy it as a web service, follow these steps:

1. Select a name for the service you wish to create.
2. Using the tools shipped with HP's Web Services platform, create the deployment descriptor in the form of a WSDL file and XML file with the same name as the service you are creating. For more information about the tools, see *Chapter 6* of this Guide. For more information about the deployment descriptor, see the next section of this chapter.
3. Make sure the deployment descriptor files are in the `<install_dir>\WebApp\hpws\webservices\<WebServiceName>` directory.

Web Service Deployment Descriptor

A web service deployment descriptor consists of two files:

1. A WSDL file for the web service, in the form `<WebServiceName>.wsdl`
2. A web service configuration file, in the form `<WebServiceName>.xml`

Web Service URL Structure

The context path for the web service URL is determined by the name of Web Application directory deployed to HP-AS (by default, the hpws directory). The web.xml file shipped within the WEB-INF subdirectory contains a servlet mapping to the SOAP in-box (i.e., `/soap`).

Note: For more information about customization of web.xml, please see the section titled *Customizing the HP-SOAP Server WebApp Configuration*.

The rest of the URL is driven by the structure of the `webservices` subdirectory hierarchy. For example, all of the sample web services are deployed under the `webservices` directory as follows:

Listing 5-1: Directory of <install_dir>\WebApps\hpws\webservices

```
10/17/2001  01:49p      <DIR>          .
10/17/2001  01:49p      <DIR>          ..
10/17/2001  01:49p      <DIR>          adapters
10/17/2001  01:49p      <DIR>          dispatchers
10/17/2001  01:49p      <DIR>          EchoService
10/17/2001  01:49p      <DIR>          ExampleEJB
10/17/2001  01:49p      <DIR>          getDate
10/17/2001  01:49p      <DIR>          MathService
10/17/2001  01:49p      <DIR>          stocks
```

The structure of a web services URL should look like:

```
http://<HostName>:<PortNumber>/<ContextPath>/<ServletMapping>/<WebServiceName>
```

For example, the URL of for the **getDate** service may look like:

```
http://localhost:9090/hpws/soap/getDate
```

where

- **`/hpws`** is the context path
- **`/soap`** represents the servlet mapping in the `web.xml`
- **`/getDate`** is the name of the service directory

  As part of the sample files shipped with HP Web Services platform, the two stock-related web services are grouped under a subdirectory named **stocks**. Hence, the URLs for these web services will reflect the subhierarchy as follows:

```
http://localhost:9090/hpws/soap/stocks/getNASDAQStock
http://localhost:9090/hpws/soap/stocks/getStock
```

This features allows you to logically group and organize services but does require that your service name match the name of a physical directory. You may use servlet mapping capability to map a logical URL to this physical structure.

Setting Up an RPC-style Service Using a Java Class

At the very least, the service's XML file contains the adapter name, the class that implements the service and whether or not the message structure should be validated.

Table 5-1: XML file structure for a service using a Java class

Tag	Attribute	Value	Description
webservice	adapter	rpc-java	Name of the adapter used by the service.
	wsdl-validation	true or false	Validates SOAP incoming message, if needed. For RPC requests, if this is true, type information is required in each request to avoid invocation ambiguity if the target operation/method is overloaded. Default is false.
param	name	"lookup-name"	For all services with RPC requests, the parameter name is required and is expected to be this.
	value	fully qualified name of Java class	The Java class that implements the service.

Listing 5-2: Example of a service using a Java class

```
<?xml version="1.0"?>
<service-descriptor version="1.0">
  <webservice adapter="rpc-java" wsdl-validation="false" >
    <param name  = "lookup-name"
           value = "com.hp.mw.soap.samples.rpc.DateService"/>
  </webservice>
</service-descriptor>
```

The deployment descriptor for a web service includes a WSDL file as well as the XML file discussed here. However, the contents of the WSDL file are dependent upon to the Java class being exposed and, in the interest of conserving space, a sample file is not shown here.

Setting Up an RPC-style Service Using an EJB Adapter

If the service you are configuring uses an EJB adapter, there are a few additional parameters that must be defined in the XML file.

Table 5-2: Parameters for a service using an EJB adapter

Tag	Attribute	Value	Description
webservice	adapter	rpc-ejb	Name of the adapter used by the service.
	wsdl-validation	true or false	Validates SOAP incoming message, if needed. For RPC requests, if this is true, type information is required in each request to avoid invocation ambiguity if the target operation/method is overloaded. Default is false.
param	name	"lookup-name"	For all services with RPC requests, the parameter name is expected to be this.
	value	name of the EJB	The value should be set to the name of the EJB bean that implements the service.
param	name	"home"	For all services with EJB-RPC requests, this parameter is required and the name is expected to be this.
	value	fully qualified name of home interface	The fully qualified home interface.
param	name	"remote"	For all services with EJB-RPC requests, this parameter is required and the name is expected to be this.
	value	fully qualified name of remote interface	The fully qualified remote interface.
param	name	"ejb-type"	For all services with EJB-RPC requests, this parameter is required and the name is expected to be this.
	value	ejb-stateless	The value of this parameter should be set to ejb-stateless.
param	name	"jndi-vendor"	For all services with EJB-RPC requests, this parameter is required and the name is expected to be this.
	value	HP or JBOSS	The value can currently be set to HP or JBOSS.

Listing 5-3: Example of a service using an EJB adapter

```
<?xml version="1.0" encoding="UTF-8" ?>
 <service-descriptor version="1.0">
   <webservice adapter="rpc-ejb" wsdl-validation="false">
     <param name="lookup-name" value="LoanCalc"/>
     <param name="home" value="com.hp.mwlabs.wso.examples.ejb.LoanCalcHome"/>
     <param name="remote" value="com.hp.mwlabs.wso.examples.ejb.LoanCalc"/>
     <param name="ejb-type" value="ejb-stateless"/>
     <param name="jndi-vendor" value="HP"/>
   </webservice>
 </service-descriptor>
```

Setting Up a Document Exchange Service Using a Java Class

You can also set up a service based on document exchange. Parameters are listed in *Table 5-3:*

Table 5-3: Parameters for a service using document exchange

Tag	Attribute	Value	Description
webservice	adapter	doc-handler	Name of the adapter used by the service. For implementations using document exchange, the adapter is defined as doc-handler.
	wsdl-validation	true or false	Values are true or false. If true, all messages are checked by the service to ensure that they follow the structure outlined in the WSDL specification.
param	name	"lookup-name"	For all services with Document Exchange requests, the parameter name is expected to be this.
	value	Fully qualified Java class.	The name of the handler class.

Listing 5-4: Example of a service using document exchange

```
<?xml version="1.0"?>
 <service-descriptor version="1.0">
   <webservice adapter="doc-handler"  wsdl-validation="false">
      <param name = "lookup-name"
             value = "com.hp.mw.pipeline.soap.samples.doc.EchoDocument"/>
   </webservice>
 </service-descriptor>
```

Configuring Services

In addition to the tasks you must perform to deploy a Java class or EJB as a web service, there are several other features you can set in the configuration files.

Configuring a Service to Use a Type Converter

You can also configure a service that requires conversion of the incoming data from the incoming XML data to the data actually required by a back-end application. Type converters are used when all data of the specified type must be converted before it is sent to the web service. For instance, a complex data type represented by an XML node will need to be converted to the actual Java class of the data type before being sent to the back-end application.

Listing 5-5: Example of a service using a type converter

```
<?xml version="1.0" encoding="UTF-8" ?>
 <service-descriptor version="1.0">
   <webservice adapter="rpc-java" wsdl-validation="false">
      <param name = "lookup-name"
            value = "com.hp.mw.pipeline.soaprpc.services.ConverterService"/>
      <soap-type>
              <type-converters>
                 <converter name="struct"
        class="org.apache.cocoon.processing.soap.conversion.CastorConverter">
         <property name=http://soapinterop.org/xsd:SOAPStruct:map
                  value="soapstruct.xml" />
                 </converter>
              </type-converters>
              <result-converters>
                 …
              </result-converters>

              <messages>
                 <message name="s0:echoFormatConvertRequestInput1">
                    …
                 </message>
              </messages>
           </soap-type>
</webservice>
</service-descriptor>
```

Configuring a Service to Use an Input/Output Converter

You can also configure type converters for any message that is an input/output message of some operation(s) on the WSDL. This overrides the service level and global Type Converter registry.

Listing 5-6: Example of a service using an input/output converter

```
<?xml version="1.0" encoding="UTF-8" ?>
 <service-descriptor version="1.0">
   <webservice adapter="rpc-java" wsdl-validation="false">
      <param name = "lookup-name"
    value = "com.hp.mw.soap.samples.rpc.services.ConverterService"/>
      <soap-type>
              <type-converters>
                 …
              </type-converters>

              <result-converters>
                 …
              </result-converters>

        <!--
        Message level I/O formatter/converter registry: This overrides the
        service level and global registry
        Because WSDL use input/output messages to specify the I/O parameters,
        using message-level registry is more suitable the operation level.
        2 different operation may share the same input or output message
        -->
              <messages>
                 <message name="s0:echoFormatConvertRequestInput1">
                    <converter name="string"
```

```
          class="org.apache.cocoon.processing.soap.conversion.ToIntConverter">
                      </converter>
                  </message>
              </messages>
          </soap-type>
</webservice>
</service-descriptor>
```

Configuring a Service to Use a Result Converter

You can also configure a service that requires conversion of the outgoing XML message generated from serialization of the data returned from a back-end application to the required WSDL structure. For instance, a value identified in WSDL as a list of simple types may actually need to be converted from a complex data type that is returned by the back-end application.

Listing 5-7: Example of a service using a result converter

```
<?xml version="1.0" encoding="UTF-8" ?>
 <service-descriptor version="1.0">
   <webservice adapter="rpc-java" wsdl-validation="false">
      <param name = "lookup-name"
            value = "com.hp. mw.soap.samples.rpc.services.ConverterService"/>
      <soap-type>
              <type-converters>
                  …
              </type-converters>

         <!--
         Return element conversion registry: this converter applies to whole
         return xml element of the operation with the specified signature.
              For example:
                  <m:echoStructAsSimpleTypesResponse>
                    <return>
                         <varString>hello world</varString>
                         <varInt>42</varInt>
                         <varFloat>0.005</varFloat>
                    </return>
                  </m: echoStructAsSimpleTypesResponse>

                  Can be converted to:

                  <m:echoStructAsSimpleTypesResponse>
                      <outputString>hello world</outputString>
                      <outputInteger>42</outputInteger>
                      <outputFloat>0.005</outputFloat>
                  </m:echoStructAsSimpleTypesResponse>
              -->
              <result-converters>

                  <result-converter
                      operation-name="echoStructAsSimpleTypesResponse"
                      input-message="s0:echoStructAsSimpleTypesResponseInput"
                      output-message="s0:echoStructAsSimpleTypesResponseOutput"
      class="org.apache.cocoon.processing.soap.conversion.ResultConverter"/>

              </result-converters>
              <messages>
                  <message name="s0:echoFormatConvertRequestInput1">
                      …
                  </message>
              </messages>

          </soap-type>
```

```
</webservice>
</service-descriptor>
```

Customizing the HP-SOAP WebApp Configuration

HP-SOAP server is implemented as a servlet web application. The file `web.xml`**,** located in `<`*`install_dir`*`>\WebApps\hpws\WEB-INF` directory, is used to configure the basic features of the HP-SOAP Server, including the definition and mapping of the servlets. It is defined by the http://java.sun.com/j2ee/dtds/web-app_2_2.dtd schema.

Configuring the JAXM Inbox Servlet

There are only two parameters in the JAXM inbox servlet section (identified by servlet-name **JAXMRPCInbox**) that can be modified: logging and custom prefixes for outgoing messages. All other parameters for the JAXM servlet ***should not be modified***.

Note: If modified, HP cannot guarantee that the servlet will function as expected.

Logging

You can specify two levels of logging: *debug* or *none*. If you don't specify a level, the default level is *none*.

Listing 5-8: Sample code for logging levels

```
<init-param>
  <param-name>log-level</param-name>
  <param-value>NONE</param-value>
</init-param>
```

Using Custom Prefixes for Outgoing Messages

You can specify whether you want to include a custom prefix in messages going out from the server. Values are *true* or *false*.

- If *false* (default), the server uses its own default SOAP prefixes (envelope, header, body) in the message. If *false*, the `<response-prefix>` value is not used.
- If *true*, the value specified by the `<response-prefix>` is used as the prefix.
- If *true* and the `<response-prefix>` section has been commented out, the server uses the incoming message's prefix as the prefix for the outgoing message as well.

Listing 5-9: Sample code for custom prefixes

```
<init-param>
  <param-name>use-custom-prefix</param-name>
  <param-value>true</param-value>
</init-param>
<init-param>
  <param-name>response-prefix</param-name>
  <param-value>SOAP-HP</param-value>
</init-param>
```

Servlet Mapping

For each servlet available, you need a `<servlet-mapping>` tag that contains information about mapping URLs for incoming HTTP requests to specific servlets. The default configuration indicates that all requests containing `/soap/*` will be processed by **JAXMRPCInbox**.

If you need to create a custom URL to invoke the **JAXMRPCInbox**, add the appropriate servlet mapping in this section.

Note: We recommend that your installation directory is the default hpmw directory. If you deploy to a different directory, the default URL settings shipped with this product won't match and must be reset.

The tags for the Cocoon server are set up correctly and ***should not be modified***.

Note: If modified, HP cannot guarantee that the servlet will function as expected.

Listing 5-10: Sample code for mapping servlets

```
<servlet-mapping>
  <servlet-name>JAXMRPCInbox</servlet-name>
  <url-pattern>/soap/*</url-pattern>
</servlet-mapping>
```

Setting Global Logging Parameters

You can control logging of errors and other information globally through the `logkit.xconf` file, located in the `<install_dir>\WebApps\hpws\WEB-INF` directory. In that file, there are two sections that contain information about logging: `<priority-filter>` and `<categories>`. The log-level value for all of these sections can be either *debug* (logging is on) or *error* (logging is off). By default, log files are kept in the `WEB-INF\logs` directory.

Listing 5-11: Sample code for global logging parameters in the logkit.xconf file

```
.
.
.
<priority-filter id="filter" log-level="ERROR">
  <servlet>
    <format type="extended">%7.7{priority} %5.5{time}:
%{message}\n%{throwable}</format>
  </servlet>
</priority-filter>
  </targets>

  <categories>
    <category name="cocoon" log-level="ERROR">
      <log-target id-ref="cocoon"/>
    </category>
    <category name="root" log-level="ERROR">
      <log-target id-ref="root"/>
      <category name="store" log-level="ERROR">
         <log-target id-ref="components"/>
         <log-target id-ref="filter"/>
      </category>
```

Configuring Adapters

An adapter can have its own configuration file. Currently, the EJB adapter and the Header adapter have their own files (`ejbadapter.xconf` and `header-adapter.xconf`), which contain adapter-specific information. The EJB adapter file contains vendor-specific JNDI lookup information. Files shipped with the product contain HP and JBoss settings. The Header adapter file contains mappings of tags and processors responsible for processing the corresponding tag blocks. The adapter configuration files shipped with the web services platform are configured correctly and ***should not be modified***.

Note: If modified, HP cannot guarantee that the servlet will function as expected.

Testing Your Deployed Web Services

Now that you've deployed your web services, you can test them in one of two ways:

- Open an HTML browser and, in the **Address** field, type `http://localhost:9090\hpws\soap\<WebServiceName>.wsdl`. If the service has been successfully deployed, the browser should display the service's WSDL.
- Create a proxy using the WSDLtoClientProxyform. Pass in the endpoint URL for your service and optionally a package name for the generate class, in the form:

  ```
  WSDLtoClientProxy <ServiceEndpointURL>[-package <PackageName>]
  ```

  The utility will attempt to issue an HTTP `get` request to the URL, retrieve the WSDL and generate the client proxy.

Note: Before generating the client proxy, set the classpath using <install_dir>\bin\hpwsenv.bat, which includes the appropriate JAR files in the classpath.

Example

```
WSDLtoClientProxy http://localhost:9090/hpws/soap/<webServiceName>.WSDL -
package com.hp
```

Chapter 6

HP-SOAP Tools

Introduction

As described in the architecture and configuration sections of this document, the HP-SOAP server uses a web application for deployment of a group of web services. The HP-SOAP server tools are designed to facilitate easy creation, deployment, and UDDI registration of web services from existing business logic. This chapter introduces the command-line tools available and describes their usage.

The HP-SOAP server command line tools are shipped in the *bin* subdirectory of the product installation directory. Make sure that this directory is in your path.

Note: The early access release of the HP-SOAP server expands the WAR file template upon installation, along with sample web services. Users may use the expanded directory <*install_dir*>\WebApps\hpws as a template to create their own WAR files and deploy them to one of the supported application servers.

In this chapter

ClassToWebService

`ClassToWebService` is a command-line utility that generates the files that make up the HP-SOAP server deployment descriptor for a specified Java class. The deployment descriptor is made up of a WSDL file with a correct endpoint URL, and a web service configuration file. If necessary, the utility creates a subdirectory for the service in a default `webservices` directory and places the generated files in that directory. It provides a simple way to enable a Java class to be deployed as a web service on the HP-SOAP server.

Usage

```
ClassToWebService <Java Class Name> <Service Endpoint URL> <WebServiceName>
[<WSDL file>]
```

where

Table 6-1: Parameters for ClassToWebService utility

Parameter	Description
<Java Class Name>	The complete path to the Java class that you wish to expose as a web service. This class must be in the classpath. **Note:** Since the HP-SOAP server needs to dynamically load this class at run-time, we recommend that you place your Java classes in the *classes* subdirectory of *WEB-INF* directory. If your class is in a JAR, then simply place the jar file in the *lib* subdirectory.
<Service Endpoint URL>	This is the URL to a deployed service. It should be of the form: `http://<HostName>:<Port>/hpws/soap/ <WebServiceName>` Host and Port should represent your web server host and port. If you are using the HP Application Server (HP-AS) and the built-in HTTP listener, the default value for the port is 9090.
<WebServiceName>	The name of the web service. This must qualify as a directory name.
<WSDL file>	Optional. If this argument is not specified, a WSDL will be automatically generated. If you do specify this argument, provide a full path to the file.

Output

The utility creates a subdirectory for the service in the `webservices` subdirectory, under the root directory of the WAR file.

`<WebServiceName>.wsdl` in the service directory

`<WebServiceName>.xml` in the service directory

Example

```
<install_dir>\bin\ClassToWebService com.hp.mw.soap.samples.rpc.MathService
http://localhost:9090/hpws/soap/MathService MathService
```

This invocation will generate a web service configuration file and a WSDL file for `com.hp.mw.soap.samples.rpc.MathService` and place the two files in a `MathService` subdirectory under the `webservices` directory.

EJBToWebService

`EJBToWebService` is a command-line utility that generates the files that make up the HP-SOAP server deployment descriptor for a specified Enterprise Java Bean (EJB) class. The deployment descriptor is made up of a WSDL file with a correct endpoint URL, and a web service configuration file. If necessary, the utility creates a subdirectory for the service in a default `webservices` directory, and places the generated files in that directory. It provides a simple way for enabling an EJB class to be deployed as a web service on the HP SOAP server.

Usage

```
EJBToWebService <EJB Remote Interface Class Name> <Service Endpoint URL>
<WebServiceName> <EJB Well known name> <Home Interface Class Name> <jndi-
vendor> <ejb client jar> [<WSDL file>]
```

where

Table 6-2: Parameters for EJBToWebService utility

Parameter	Description
<EJB Remote Interface Class Name>	The fully qualified Java class that represents the remote interface to the EJB.
<Service Endpoint URL>	The URL to a deployed service. It should be of the form: http://<Host>:<Port>/hpws/soap/<WebServiceName>. <Host> and <Port> should represent your web server host and port. If you are using the HP Application Server and the built-in HTTP listener, the default value for the port is 9090.
<WebServiceName>	The name of the web service. This must qualify as a directory name.
<EJB Well Known Name>	The JNDI lookup name for the EJB.
<Home Interface Class Name>	Full path to the EJB home interface class.
<jndi-vendor>	Enter one of the supported vendor names. Currently, HP or JBoss.

Table 6-2: Parameters for EJBToWebService utility

Parameter	Description
<EJB Client Jar>	The complete file path and name of the EJB client jar generated by the application server deployment tool. This utility adds it to the internal classpath so that WSDL generation succeeds. **Note:** Since HP-SOAP server needs to dynamically load this class at run-time, we recommend that you place your Java classes in the `classes` subdirectory of WEB-INF directory. If your class is in a JAR, then simply place the jar file in the lib subdirectory.
<WSDL file>	Optional. If this argument is not specified, a WSDL is automatically generated. If you do specify this argument, provide a full path to the file.

Output

This utility creates a subdirectory for the service in the `webservices` subdirectory, under the root directory of the WAR file.

`<WebServiceName>.wsdl` in the service directory

`<WebServiceName>.xml` in the service directory

Example

```
<install_dir>\bin\EJBToWebService com.hp.mwlabs.wso.examples.ejb.LoanCalc
http://localhost:9090/hpws/soap/MyEJBLoanCalc MyEJBLoanCalc LoanCalc
com.hp.mwlabs.wso.examples.ejb.LoanCalcHome HP client_dep.jar
```

This invocation will generate a web service configuration file, and a WSDL file for `the LoanCalc` EJB. It will copy them to a `MyEJBLoanCalc` subdirectory under the `webservices` directory.

HandlerToWebService

`HandlerToWebService` is a command-line utility that generates the web service configuration file for a specified Document Handler class. The deployment descriptor in the HP-SOAP server is made up of a WSDL file with a correct endpoint URL, and a web service configuration file. In the case of Document Handlers, you should create a WSDL file for your service and include a correct endpoint URL. As shown below, this file is one of the required arguments to this utility. If necessary, the utility creates a subdirectory for the service in a default `webservices` directory and places the relevant files in that directory.

Usage

```
HandlerToWebService <Handler Class Name> <Service Endpoint URL>
<WebServiceName> <WSDL file>
```

where

Table 6-3: Parameters for HandlerToWebService utility

Parameter	Description
<Handler Class Name>	The complete path to the Java class that implements the com.hp.mw.soap.handlerSOAPMessageHandler interface. **Note:** Since the HP-SOAP server needs to dynamically load this class at run-time, we recommend that you place your Java classes in the `classes` subdirectory of WEB-INF directory. If your class is in a JAR, then simply place the jar file in the lib subdirectory.
<Service Endpoint URL>	The URL to a deployed service. It should be of the form: http://<Host>:<Port>/hpws/soap/<WebServiceName>. <Host> and <Port> should represent your web server host and port. If you are using the HP Application Server and the built-in HTTP listener, the default value for the port is 9090. **Note:** This URL is not currently used by this utility
<WebServiceName>	The name of the web service. This must qualify as a directory name.
<WSDL file>	Required. Specify the complete file path and name to your WSDL file. Ensure that the endpoint and service name information in the WSDL file is accurate. The format of the endpoint URL for a service deployed in HP-SOAP is as follows: This is the URL to a deployed service. It should be of the form: http://<Host>:<Port>/hpws/soap/<WebServiceName>. The <Host> and <Port> should represent your web server host and port. If you are using the HP Application Server and the built-in HTTP listener, the default value for the port is 9090.

Output

This utility creates a subdirectory for the service in the `webservices` subdirectory, under the root directory of the web application.

It copies the WSDL file that was passed in by the user to the service subdirectory with a new name in the service directory.

```
<WebServiceName>.wsdl
<WebServiceName>.xml
```

WSDLToClientProxy

WSDLToClientProxy is a command-line utility that reads the WSDL (Web Service Description Language) for a given web service and generates Java source code for a "proxy" class This code can be compiled and used as if the web service was a local class. It performs essentially the same task as Java's `rmic` command, but for SOAP web services.

Usage

```
WSDLToClientProxy <Service Endpoint URL> [<PackageName>]
```

where

Table 6-4: Parameters for WSDLToClientProxy utility

Parameter	Description
<Service Endpoint URL for WSDL>	The URL to get the WSDL for a deployed service. It should be of the form: http://<Host>:<Port>/hpws/soap/<WebServiceName>wsdl The <Host> and <Port> should represent your web server host and port. If you are using the HP Application Server and the built-in HTTP listener, the default value for the port is 9090.
<PackageName>	Optional. If specified, the generated proxy class will have the package included in its Java source file.

Output

This utility generates the client proxy Java class in the current directory.

ClassToWSDL

`ClassToWSDL` is a command-line utility that generates WSDL for a given Java class implementing a web service. It provides a simple way for publishing the interface of a web service implemented as a Java class over the Internet to potential users of the web service.

Usage

```
ClassToWSDL <Java Class Name> <Service Endpoint URL> <WebServiceName>
```

where

Table 6-5: Parameters for ClassToWSDL utility

Parameter	Description
<Java Class Name>	The complete path to the Java class that you wish to expose as a web service. This class must be in classpath. **Note:** Since HP-SOAP server needs to dynamically load this class at run-time. We recommend that you place your Java classes in the *classes* subdirectory of the *WEB-INF* directory. If your class is in a JAR, then simply place the JAR file in the *lib* subdirectory.
<Service Endpoint URL>	The URL to a deployed service. It should be of the form: http://<Host>:<Port>/hpws/soap/<WebServiceName> <Host> and <Port> should represent your web server host and port. If you are using the HP Application Server and the built-in HTTP listener, the default value for the port is 9090.
<WebServiceName>	The name of the web service. This must qualify as a directory name.

Output

```
<WebServiceName>.wsdl in the current directory.
```

WSDLToServerSkeleton

`WSDLToServerSkeleton` is a command-line utility which reads the WSDL for a given web service. It generates Java source code for the "skeleton" of a service that implements the interface. Users can edit the file and implement each method. The tool is primarily used for porting a web service from one language to another.

Usage

```
WSDLToServerSkeleton <Service Endpoint URL> <PackageName>
```

where

Table 6-6: Parameters for WSDLToServerSkeleton utility

Parameter	Description
<Service Endpoint URL>	The URL to a deployed service. It should be of the form: http://<Host>:<Port>/hpws/soap/ <WebServiceName> <Host> and <Port> should represent your web server host and port. If you are using the HP Application Server and the built-in HTTP listener, the default value for the port is 9090.
<PackageName>	The Java package name to be used in the generated skeleton service source file.

Index

Glossary

Adapter / Handler

A software component that receives the business payload of a SOAP message and passes it to the application that will process it.

B2B

Refers to Business-to-Business online interactions. The HP Web Services platform facilitates business-to-business interactions by enabling cross-platform communication and sharing of business information using a SOAP based message exchange implementation.

BizTalk

The Microsoft BizTalk Application Server automates the exchange of internal business data enabling companies to exchange business data with partners using EDI or XML. Primarily aimed at enabling smaller companies to do electronic business with larger companies that have EDI servers.

BTP

Business Transaction Protocol (BTP) is an XML-based specification for representing and managing complex, multi-step transactions over the Internet. BTP provides an open specification for XML message interfaces to support coordination of web services from different Internet trading partners.

Castor Converter

The Castor converter is an open source, data-binding framework that enables you to convert an XML document into a Java object, or the reverse, from a Java object back into an XML document.

Cocoon

Apache Cocoon is an XML publishing framework that is designed around pipelined SAX processing. The HP-SOAP server takes advantage of Cocoon's pipeline controller features and application server-neutral services.

Deserialize

The process of transforming XML data into a Java object that can be used by a back-end application. In the HP-SOAP server, the Castor converter performs this action.

Digital Signature

A unique keycode that provides security for parties exchanging information across the Internet. This keycode enables client / server authentication to occur before information is exchanged.

ebXML

ebXML is an international initiative established by UN/CEFACT and OASIS. "The United Nations Centre for Trade Facilitation and Electronic Business (UN/CEFACT) and the Organization for the Advancement of Structured Information Standards (OASIS) have joined forces to initiate a worldwide project to standardize XML business specifications. UN/CEFACT and OASIS have established the Electronic Business XML Working Group to develop a technical framework that will enable XML to be utilized in a consistent manner for the exchange of all electronic business data.

EDI

Electronic Data Interchange is a set of standards for controlling the exchange of business documents (such as purchase orders and invoices) between computers.

EJB

Enterprise JavaBeans is the server-side component architecture for the J2EE platform. EJB enables simplified development of distributed, transactional, secure and portable Java applications.

Element

An element is another name for an XML tag that can contain other elements or tags.

Formatter

Formatters in the HP-SOAP server allow the user to reformat the data from an incoming SOAP message before and after it has been processed.

GET

HTTP servers may serve GET requests as part of the HTTP protocol.

Handler

A Java object that implements one of the handler interfaces shipped with the HP-SOAP server that serves as a plugin of functionality.

HP-AS

Hewlett-Packard's J2EE compliant application server.

Interoperability

The ability to handle translations between data formats, communications protocols, and security mechanisms. The HP Web Services platform is built upon standards based technologies and therefore enables interoperability between diverse business partners that utilize these standards.

JAXM

The Java API for XML Messaging is an optional package that enables applications to send and receive document oriented XML messages using a pure Java API. JAXM implements the Simple Object Access Protocol (SOAP) 1.1, with Attachments messaging.

JAXM Serializer

A component in the HP-SOAP server that converts the XML output into bytes for transmission along the wire to a receiver.

JSF

The Java Services Framework (JSF) is an open, standard mechanism for assembling service components into Java server applications.

SOAP Pipeline

A pipeline refers to the various processing stages that a SOAP message undergoes from the time it is received by the HP-SOAP server until it is sent to its ultimate destination in the form of a response. The HP-SOAP server has consists of two stages in its pipeline: header processing and payload processing.

POST

HTTP servers may serve POST requests as part of the HTTP protocol.

Registry Composer

The HP-Registry Composer is a browser tool that enables easy registration of web services and for discovering existing web services on public or private UDDI servers.

RPC

Remote Procedure Call is a function call that takes place from a software client to a software server. It maintains procedure call semantics despite the fact that the server is not in the same process space as the client.

Serialize

The process of transforming a Java object into XML data. In the HP-SOAP server, the Castor converter performs this action. Serialization is also performed in another context. After the data has been converted to XML, the JAXM Serializer converts the XML data into a byte stream so that it can be sent to a receiver across the wire.

SOAP

Simple Object Access Protocol is a connectionless protocol that is built around the model of passing documents between objects on a network. SOAP also specifies RPC capabilities.

SOAP Actor

A SOAP actor has a server-side and client-side meaning. On the server-side, a SOAP actor defines the location of a set of web services. This actor is defined by an element in a global web service configuration file. On the client-side, an actor is an attribute in the <header-block> element of the requesting SOAP message. This actor attribute maps to the location defined in the actor element defined on the server-side.

SOAP Node

A SOAP node is the concept of a location of a set of web services. A SOAP message may traverse one or more SOAP nodes as a part of its processing.

UDDI Registry

Universal Description, Discovery and Integration (UDDI) is a repository-based registry service for the automated lookup of web services. It is a place where web service providers can publish their services and users can discover the services that are available.

WAR

A WAR (web application archive) file is an archive file that contains web components, server-side utility classes, and other needed files. The HP Web Services platform uses a WAR file to deploy its SOAP server and web services.

WSDL

Web Services Description Language (WSDL) is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information

XML Digital Signatures

Introduction

Digital Signatures for XML Documents is a work in progress from the World Wide Web Consortium (W3C) and the IETF--a work begun in 1999. The latest version of the candidate specification can be found at http://www.w3.org/TR/xmldsig-core. Its goal is to describe how to sign web resources, and in particular XML documents, to provide data integrity, signature assurance and non-repudiatability over web resources.

The HP Digital Signature Library classes are contained in the files `xml-dsig.jar, xml-common.jar` and `xml-c14n.jar`. These files are located in the `<install_dir>\lib` directory. The library also relies on the Sun Java Cryptography extension, `jce1_2_1.jar` and XML parser implementations `jaxp.jar, xcerces.java` and `xalan.jar` which are installed in the `<install_dir>\lib` directory. All of these files must be in your `CLASSPATH` to use the HP Digital Signature Library. This is done automatically for web services developed in the HP Web Services platform and/or HP-SOAP server. For standalone use, you can use the `HPWSenv.bat` batch file (or `HPWSenv.sh` shell script for Linux/UNIX users) which is provided in the `<install_dir>\bin` directory to set the `classpath` correctly.

In this chapter

Need for XML-based Digital Signature

Current security solutions for most web deployments are insufficient for securing business transactions. For business transactions carried out over the web, it is important to verify authenticity of the message (who sent this message?), check the integrity of the data contained in the message (has message been modified in transit?) and support non-repudiation (message sender should not be able to deny sending the message). In scenarios where an XML document is passed among multiple parties, each party may want to sign only the portion for which the party is responsible.

Using digital signatures in XML applies some additional requirements. While two XML documents can be logically equivalent for most applications (that is, they differ in Namespace or they are transformed with a different entity structure, attribute ordering or character encoding), without some sort of canonicalization the documents will generate different digital signature.

Following is an example scenario that shows how digital signature can be used. In this scenario, Alice, a client, and Bob, her banker, communicate over the Internet. Alice wants Bob to transfer $1000 from her account to Tom's account.

1. Alice sends the following message (M) to Bob: *"Bob, Transfer $1000 from my account to account 182376. Alice"*

Bob cannot know from this message alone whether Alice was the author of the message or whether someone else sent the message. To verify the author, digital signature can be used.

A digital signature is a mechanism that takes a key (`K`), an hash algorithm and a message (`M`) to produce signature output $\texttt{Sig}_M$ by using a signing function $\texttt{S}_K$ ($\texttt{Sig}_M = \texttt{S}_K(\texttt{hash(M)})$). This signature has a similar property as a seal. It is non-repudiation: Only the signer could generate the signature for that document (not reusable). In addition, anyone is able to verify the signature, and the signed document cannot be modified without breaking the signature.

In the PKI-based digital signature, Alice will use the private key to sign the hash (obtained from a one-way hash function) of the document and send the result ($\texttt{Sig}_M$) along with the original document. Once Alice has signed the document:

- Every person who has Alice's public key Pub can make sure that the pair `M`, $\texttt{Sig}_M$ is a valid pair by verifying $\texttt{Sig}_M$ is the signature for message `M`.
- Only the owner of the private key can produce a digital signature that can be verified through the public key.

2. Bob verifies the signature of the document as follows:

- Computes the hash of the document: `H`
- Bob verifies the signature using Alice's public key, recovering the original hash `H'` that Alice signed.
- Bob then compares `H` and `H'`: they should be identical to verify the signature.
- The example below shows the need for using canonicalization for XML digital signature. XML documents can undergo many mutations during the process of communication. For example, the document could be modified when it is transferred from the *client* to *Server B* through *Server A*.

Client to Server A

```
<Body>
  <Account>12345</Account>
    <Transfer>
      <Sum>1000</Sum>
      <Destination>
        <Account>62785</Account>
```

Server A to Server B

```
<Transaction ns="http://ServerB/bank>
  <Body>
    <Account>12345</Account>
    <Transfer>
      <Sum>1000</Sum>
      <Destination>
```

```
        </Destination>
    <Transfer>
    <Today/>
<Body>
```

```
        <Account>62785</Account>
      </Destination>
    <Transfer>
    <Today></Today>
  <Body>
  <Approved/>
</Transaction>
```

The original document is embedded inside another one, which causes its nodes to belong to a new namespace, and the element `"Today"` has been expanded.

The digital signature situation involving Alice yields a different value for a change of even one bit in the message; therefore, it cannot be used as a solution. The expansion of `"Today"` does not change the document from an XML standpoint, but it does change from a binary standpoint. A more flexible scheme is required that would render the same signature valid under some acceptable transformation. The XML digital signature specification provides a scheme for handling this type of scenario and also provides an option for signing the selected elements of the XML document.

Structure of an XML Signature

Code Listing A-1: Structure of an XML Signature

```
<Signature>
  <SignedInfo>
    (CanonicalizatrionMethod)
    (SignatureMethod)
    (<Reference (URI=)?>
      (Transforms)?
      (DigestMethod)
      (DigestValue)
   (Reference>)+
 <SignedInfo>
  (SignatureValue)
  (KeyInfo)?
 (Object)*
</Signature>
```

The elements of the XML signature are explained in the following table.

Table A-1: Elements of an XML signature

Element	Description
SignedInfo	The information that is actually signed.
CanonicalizationMethod	Represents the algorithm that is used to canonicalize the SignedInfo element.
Reference	Each resource to be signed has its own <Reference> element, identified by the URI attribute. It includes the digest method and resulting digest value calculated over the identified data object.
Transform	Specifies an ordered list of processing steps that were applied to the referenced resource's content before it was digested.
DigestValue	Carries the value of the digest of the reference resource.
SignatureValue	Carries the value of the digest of the signedInfo element.

Table A-1: Elements of an XML signature

Element	Description
KeyInfo	Indicates the key to be used to validate the signature. Possible forms for identification include certificates, keynames and key agreement algorithms and information. Optional.

Steps to Create XML Signature

To create a digital signature, you need to complete the following steps:

1. Identify the data objects to be signed.
2. Generate the reference element(s).
3. Create the SignedInfo element.
4. Generate the signature.

Identifying Data Objects to be Signed

This step involves identifying all the data objects, which are identified through the Uniform Resource Identifier (URI). The data objects can refer to a specific element in XML document, to the whole XML document or to any web resource identified by a URI.

Reference Generation

In the signature, each data object is referenced by a reference element.

1. To generate a reference element, first apply any optional transform to the data object . For example, you may want to use Base64 transformation for binary pages.
2. Next, feed the transformed stream into a digest algorithm. The algorithm is specified as value for `<DigestMethod>` element (e.g., SHA-1). The result is placed as the value for the `<DigestValue>` element. The transforms element contains the list of transforms applied on the Data Object before calculating the digest.
3. Based on the above elements, a Reference element is created.

Code Listing A-2: Sample code for generating a reference element

```
<Reference URI="http://www.hp.com/purchase/purchase_order.xml">
 <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
 <DigestValue>4JLaUVVmpdfobC3y+GxoqGUZlB0=</DigestValue>
</Reference>
<Reference
  URI=" http://www.hp.com/purchase/buy.xml">
 <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
 <DigestValue>2jmj7l5rSw0yVb/vlWAYkK/YBwk=</DigestValue>
</Reference>
```

Creating the SignedInfo Element

The `SignedInfo` element is created using the `<SignatureMethod>` element (which is an algorithm describing the digital signature algorithm), the `<CanonicalizationMethod>` element (which defines the Canonicalization algorithm used for canonicalization of the SignedInfo element) and one or more reference element(s).

Code Listing A-3: Sample code for creating the SignedInfo element

```
<SignedInfo Id="DsigExample">
 <CanonicalizationMethod
    Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
 <SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#dsa-sha1" />
 <Reference URI="http://www.hp.com/purchase/purchase_order.xml">
 <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
 <DigestValue>4JLaUVVmpdfobC3y+GxoqGUZlB0=</DigestValue>
</Reference>
<Reference
  URI=" http://www.hp.com/purchase/buy.xml">
 <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
 <DigestValue>2jmj7l5rSw0yVb/vlWAYkK/YBwk=</DigestValue>
</Reference>
</SignedInfo>
```

Generating the Signature

After the `SignedInfo` element is canonicalized, the next step is to calculate the digest of canonicalized `SignedInfo` element and sign the digest using the `<SignatureMethod>` element value. The result is then placed in the `<SignatureValue>` element.

If key information is needed, it can be placed in the `<KeyInfo>` element. `<KeyInfo>` contains the public key corresponding to the private key used for signing. `<SignedInfo>`, `<SignatureValue>` and optional `<KeyInfo>` elements are added into the `<Signature>` element to complete the XML Signature.

A reference that contains a null URI attribute (`URI=""`) signs the entire document. Enveloped signatures are over data within the same XML document as the signature; detached signatures are over data external to the signature element.

Code Listing A-4: Sample code for signature generation

```
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
SignedInfo Id="DsigExample">
 <CanonicalizationMethod
    Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
 <SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#dsa-sha1" />
 <Reference URI="http://www.hp.com/purchase/purchase_order.xml">
 <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
 <DigestValue>4JLaUVVmpdfobC3y+GxoqGUZlB0=</DigestValue>
</Reference>
<Reference
  URI=" http://www.hp.com/purchase/buy.xml">
 <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
 <DigestValue>2jmj7l5rSw0yVb/vlWAYkK/YBwk=</DigestValue>
</Reference>
</SignedInfo>
<SignatureValue>h134^67</SignatureValue>
<KeyInfo>
<X509Data>
<X509SubjectName>CN=first Last,O=HP,L=Cupertino, S=CA,C=US</X509SubjectName>
<X509Certificate>
MDIOFJDD…DDD
</X509Certificate>
</X509Data>
</KeyInfo>
</Signature>
```

Verifying the Signature

Verification is a symmetrical process. To verify the digital signature, follow these steps:

1. For each reference element, recalculate the digests of the reference element and compare it to the value in the <DigestValue> element of each <Reference> element.
2. Verify the Signature of the <SignedInfo> element. In this step, calculate the digest of <SignedInfo> and use this and the public key to verify the value of the <SignatureValue> element.

HP XML Digital Signature Library

HP XML Digital Signature provides the implementation in line with the current W3C specification. The library is developed in Java and provides Java API for generating XML Signature and for verification. This section provides the code sample for using the HP XML Digital Signature library.

Creating the Signature

To create the signature, follow the steps here:

1. Parse the XML document you would like to sign and Store in the Document object. Identify the element object in which the signature will be added. The code below assumes that the Signature will be added to `sigParentElement`.

```
Element sigParentElement = null;
sigParentElement  = <Element to which Signature tag will be appended>
```

2. Select the name of the signature algorithm and the canonicalization and create `KeyInfo` element if key is needed to be added to Signature.

```
// create a key info from the key pair.

KeyPair keyPairForSigning = (Generate RSA Based Key Pair…)
RSAPublicKey rsaPublicKey
= (RSAPublicKey) keyPairForSigning.getPublic();
KeyValue keyValue = new KeyValue( rsaPublicKey );
KeyInfo keyInfo = new KeyInfo( keyValue );

// create a signature object which will sign the xml document.
// the first argument is the element to which the <Signature> tag
//will be appended as child, after signing.

Signature signature = null;
signature = new Signature(sigParentElement,
"http://www.w3.org/2000/09/xmldsig# rsa-sha1”,
C14nAlgorithm.WITHOUT_COMMENTS, keyInfo );
```

where, in the Signature object constructor:

- `sigParentElement` is the XML element to which Signature tag will be appended i.e. added as a child element.
- 2nd element represent the signature algorithm (Corresponding to `<SignatureMethod>` element
- 3rd parameter represents Canonicalization Algorithm corresponding to `<CanonicalizationMethod>`

3. Adding the reference element

```
// add the reference.
Reference reference = null;
Reference = signature.addReference( refDigestMethod, refId,refURI
,refType );
```

where:

- `refDigestMethod` refers to the Digest algorithm for Reference Element. (e.g., `http://www.w3.org/2000/09/xmldsig#sha1`)
- `refId` is used to specify the ID for the reference.
- `refURI` represents the reference URI.
- `RefType` can be null other values are also possible. It allow the digest to be calculated regardless of the `SignatureElement.`

4. Add the transform algorithm for the reference. It is important to specify the `ENVELOPED_SIGNATURE` algorithm in the case of Enveloped Signature. This step should be omitted in the case of detached signature.

```
// set the algorithm to be used for TRANSFORMation
reference.addTransform( TransformAlgorithm.ENVELOPED_SIGNATURE );
```

5. Sign the document using the private key in the `keyinfo` passed.

```
signature.sign( keyPairForSigning.getPrivate() );
```

After this step is complete, the signature element is appended to `sigParentElement` object.

Verifying the Signature

To verify the digital signature, follow these steps:

1. For the verification process, in the received XML document, check the document to find `<Signature>` element. The code sample assumes that `<Signature>` element is stored in the element.

```
Element Sigelement = null;
SigElement = Signature.findFirst( docToVerify, Signature.getTagName() );
```

Where `findFirst` method finds the first `Signature` element. `DocToVerify` is a Document object which needs to be verified.

2. Create a signature element using the above element.

```
Signature signature = new Signature( siglement );
```

3. In the case `<Signature>` contains `<KeyInfo>` the public credentials in it can be used for verification, otherwise external public key needs to be provided for verification.

```
Boolean verified = signature.verify().isSuccess();
// The above API assumes that key is specified in the <KeyInfo> element
          and will return true if the signature is verified.
OR
Boolean verified = signature.verify(pubKey).isSuccess();
// where pubKey refers to externally provided Key.
```