The Java™ Web Services Tutorial

For Java Web Services Developer's Pack, v1.6

June 14, 2005

Contents

About This Tutorial

THE Java™ Web Services Tutorial is a guide to developing Web applications with the Java Web Services Developer Pack (Java WSDP). The Java WSDP is an all-in-one download containing key technologies to simplify building of Web services using the Java 2 Platform. This tutorial requires a full installation (Typical, not Custom) of the Java WSDP, v1.6 with the Sun Java System Application Server Platform Edition 8.1 2005Q2 UR2 (hereafter called the Application Server). Here we cover all the things you need to know to make the best use of this tutorial.

Who Should Use This Tutorial

This tutorial is intended for programmers who are interested in developing and deploying Web services and Web applications on the Sun Java System Application Server Platform Edition 8.1.

Prerequisites

Before proceeding with this tutorial you should have a good knowledge of the Java programming language. A good way to get to that point is to work through all the basic and some of the specialized trails in *The Java™ Tutorial*, Mary Campione et al., (Addison-Wesley, 2000). In particular, you should be familiar with relational database and security features described in the trails listed in Table 1.

Table 1 Prerequisite Trails in *The Java™ Tutorial*

Trail	**URL**
JDBC	`http://java.sun.com/docs/books/tutorial/jdbc`
Security	`http://java.sun.com/docs/books/tutorial/security1.2`

How to Use This Tutorial

The *Java Web Services Tutorial* is an adjunct to the *J2EE 1.4 Tutorial*, which you can download from the following location:

`http://java.sun.com/j2ee/1.4/download.html#tutorial`

The *Java Web Services Tutorial* addresses the following technology areas, which are *not* covered in the *J2EE 1.4 Tutorial*:

- The Java Architecture for XML Binding (JAXB)
- The StAX APIs and the Sun Java Streaming XML Parser implementation
- XML and Web Services Security (XWS Security)
- XML Digital Signature
- Service Registry

All of the examples for this tutorial are installed with the Java WSDP 1.6 bundle and can be found in the subdirectories of the `<JWSDP_HOME>/<`*`technology`*`>/samples` directory, where *`JWSDP_HOME`* is the directory where you installed the Java WSDP 1.6 bundle.

The *J2EE 1.4 Tutorial* opens with three introductory chapters that you should read before proceeding to any specific technology area. Java WSDP users should look at Chapters 2 and 3, which cover XML basics and getting started with Web applications.

When you have digested the basics, you can delve into one or more of the following main XML technology areas:

- The Java XML chapters cover the technologies for developing applications that process XML documents and implement Web services components:
    - The Java API for XML Processing (JAXP)
    - The Java API for XML-based RPC (JAX-RPC)
    - SOAP with Attachments API for Java (SAAJ)
    - The Java API for XML Registries (JAXR)
- The Web-tier technology chapters cover the components used in developing the presentation layer of a J2EE or stand-alone Web application:
    - Java Servlet
    - JavaServer Pages (JSP)
    - JavaServer Pages Standard Tag Library (JSTL)
    - JavaServer Faces
    - Web application internationalization and localization
- The platform services chapters cover system services used by all the J2EE component technologies. Java WSDP users should look at the Web-tier section of the Security chapter.

After you have become familiar with some of the technology areas, you are ready to tackle a case study, which ties together several of the technologies discussed in the tutorial. The Coffee Break Application (Chapter 35) describes an application that uses the Web application and Web services APIs.

Finally, the following appendixes contain auxiliary information helpful to the Web Services application developer:

- Java encoding schemes (Appendix A)
- XML Standards (Appendix B)
- HTTP overview (Appendix C)

About the Examples

This section tells you everything you need to know to install, build, and run the examples.

Required Software

Java WSDP 1.6 Bundle

The example source for the technologies in this tutorial is contained in the Java WSDP 1.6 bundle. If you are viewing this online, you need to download the Java WSDP 1.6 bundle from:

```
http://java.sun.com/webservices/download/webservicespack.html
```

After you have installed the Java WSDP 1.6 bundle, the example source code is in the subdirectories of the `<JWSDP_HOME>/<`*`technology`*`>/samples/` directory. For example, the examples for JAXB are included in the Java WSDP in the subdirectories of the `<JWSDP_HOME>/jaxb/samples` directory.

Application Server

Sun Java System Application Server Platform Edition 8.1 2005Q2 UR2 is the build and runtime environment for the tutorial examples. To build, deploy, and run the examples, you need a copy of the Application Server and the Java 2 Software Development Kit, Standard Edition (J2SE SDK) 1.4.2 or higher (J2SE 5.0 is recommended). The Application Server and J2SE SDK are contained in the J2EE 1.4 SDK. If you already have a copy of the J2SE SDK, you can download the Application Server from:

```
http://java.sun.com/j2ee/1.4/download.html#sdk
```

You can also download the J2EE 1.4 SDK—which contains the Application Server and the J2SE SDK—from the same site.

Building the Examples

Most of the examples in the Java WSDP are distributed with a build file for Ant, a portable build tool contained in the Java WSDP. For information about Ant, visit `http://ant.apache.org/`. Directions for building the examples are provided in each chapter. Most of the tutorial examples in the *J2EE 1.4 Tutorial* are distributed with a configuration file for `asant`, a portable build tool contained in the Application Server. This tool is an extension of the Ant tool developed by the Apache Software Foundation (`http://ant.apache.org`). The `asant` utility contains additional tasks that invoke the Application Server administration utility `asadmin`. Directions for building the examples are provided in each chapter.

In order to run the Ant scripts, you must configure your environment and properties files as follows:

- Add the `bin` directory of your J2SE SDK installation to the front of your path.
- Add `<JWSDP_HOME>/jwsdp-shared/bin` to the front of your path so the Java WSDP 1.6 scripts that are shared by multiple components override other installations.
- Add `<JWSDP_HOME>/apache-ant/bin` to the front of your path so that the Java WSDP 1.6 Ant script overrides other installations.

Further Information

This tutorial includes the basic information that you need to deploy applications on and administer the Application Server.

For reference information on the tools distributed with the Application Server, see the man pages at `http://docs.sun.com/db/doc/819-0082`.

See the *Sun Java™ System Application Server Platform Edition 8.1 2005Q1 Developer's Guide* at `http://docs.sun.com/db/doc/819-0079` for information about developer features of the Application Server.

See the *Sun Java™ System Application Server Platform Edition 8.1 2005Q1 Administration Guide* at `http://docs.sun.com/db/doc/819-0076` for information about administering the Application Server.

For information about the PointBase database included with the Application Server, see the PointBase Web site at `www.pointbase.com`.

How to Print This Tutorial

To print this tutorial, follow these steps:

1. Ensure that Adobe Acrobat Reader is installed on your system.
2. Open the PDF version of this book.
3. Click the printer icon in Adobe Acrobat Reader.

Typographical Conventions

Table 2 lists the typographical conventions used in this tutorial.

Table 2 Typographical Conventions

Font Style	Uses
italic	Emphasis, titles, first occurrence of terms
`monospace`	URLs, code examples, file names, path names, tool names, application names, programming language keywords, tag, interface, class, method, and field names, properties
`italic monospace`	Variables in code, file paths, and URLs
<*`italic monospace`*>	User-selected file path components

Feedback

Please send comments, broken link reports, errors, suggestions, and questions about this tutorial to the tutorial team at `users@jwsdp.dev.java.net`.

1

Binding XML Schema to Java Classes with JAXB

THE Java™ Architecture for XML Binding (JAXB) provides a fast and convenient way to bind XML schemas to Java representations, making it easy for Java developers to incorporate XML data and processing functions in Java applications. As part of this process, JAXB provides methods for unmarshalling XML instance documents into Java content trees, and then marshalling Java content trees back into XML instance documents.

What this all means is that you can leverage the flexibility of platform-neutral XML data in Java applications without having to deal with or even know XML programming techniques. Moreover, you can take advantage of XML strengths without having to rely on heavyweight, complex XML processing models like SAX or DOM. JAXB hides the details and gets rid of the extraneous relationships in SAX and DOM—generated JAXB classes describe only the relationships actually defined in the source schemas. The result is highly portable XML data joined with highly portable Java code that can be used to create flexible, lightweight applications and Web services.

This chapter describes the JAXB architecture, functions, and core concepts. You should read this chapter before proceeding to Chapter 2, which provides sample code and step-by-step procedures for using JAXB.

JAXB Architecture

This section describes the components and interactions in the JAXB processing model. After providing a general overview, this section goes into more detail about core JAXB features. The topics in this section include:

- Architectural Overview
- The JAXB Binding Process
- JAXB Binding Framework
- More About javax.xml.bind
- More About Unmarshalling
- More About Marshalling
- More About Validation

Architectural Overview

Figure 1–1 shows the components that make up a JAXB implementation.

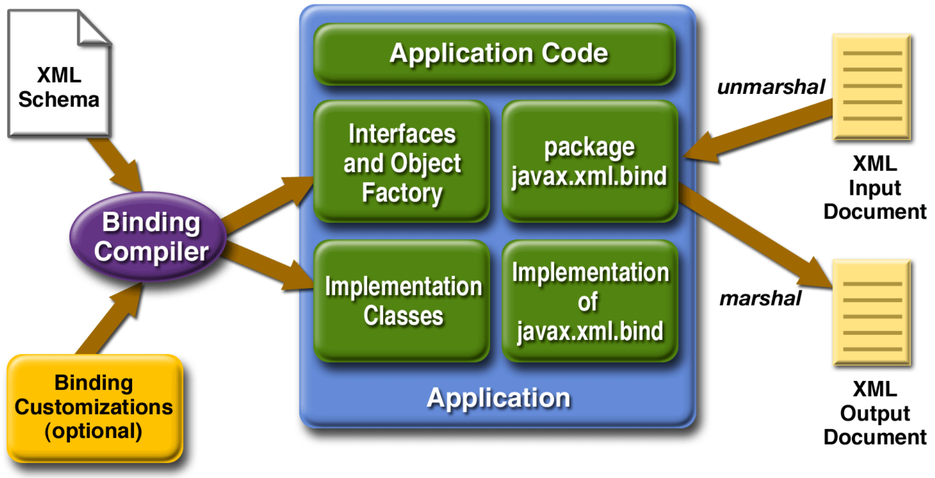

Figure 1–1 JAXB Architectural Overview

As shown in Figure 1–1, a JAXB implementation comprises the following eight core components.

Table 1–1 Core Components in a JAXB Implementation

Component	**Description**
XML Schema	An XML schema uses XML syntax to describe the relationships among elements, attributes and entities in an XML document. The purpose of an XML schema is to define a class of XML documents that must adhere to a particular set of structural rules and data constraints. For example, you may want to define separate schemas for chapter-oriented books, for an online purchase order system, or for a personnel database. In the context of JAXB, an XML document containing data that is constrained by an XML schema is referred to as a *document instance*, and the structure and data within a document instance is referred to as a *content tree*.
Binding Customizations	By default, the JAXB binding compiler binds Java classes and packages to a source XML schema based on rules defined in Section 5, “Binding XML Schema to Java Representations,” in the *JAXB Specification*. In most cases, the default binding rules are sufficient to generate a robust set of schema-derived classes from a wide range of schemas. There may be times, however, when the default binding rules are not sufficient for your needs. JAXB supports customizations and overrides to the default binding rules by means of *binding customizations* made either inline as annotations in a source schema, or as statements in an external binding customization file that is passed to the JAXB binding compiler. Note that custom JAXB binding customizations also allow you to customize your generated JAXB classes beyond the XML-specific constraints in an XML schema to include Java-specific refinements such as class and package name mappings.
Binding Compiler	The JAXB binding compiler is the core of the JAXB processing model. Its function is to transform, or bind, a source XML schema to a set of JAXB *content classes* in the Java programming language. Basically, you run the JAXB binding compiler using an XML schema (optionally with custom binding declarations) as input, and the binding compiler generates Java classes that map to constraints in the source XML schema.
Implementation of `javax.xml.bind`	The JAXB binding framework implementation is a runtime API that provides interfaces for unmarshalling, marshalling, and validating XML content in a Java application. The binding framework comprises interfaces in the `javax.xml.bind` package.
Schema-Derived Classes	These are the schema-derived classes generated by the binding JAXB compiler. The specific classes will vary depending on the input schema.

Table 1–1 Core Components in a JAXB Implementation (Continued)

Component	Description
Java Application	In the context of JAXB, a Java application is a client application that uses the JAXB binding framework to unmarshal XML data, validate and modify Java content objects, and marshal Java content back to XML data. Typically, the JAXB binding framework is wrapped in a larger Java application that may provide UI features, XML transformation functions, data processing, or whatever else is desired.
XML Input Documents	XML content that is unmarshalled as input to the JAXB binding framework -- that is, an XML instance document, from which a Java representation in the form of a content tree is generated. In practice, the term “document” may not have the conventional meaning, as an XML instance document does not have to be a completely formed, selfstanding document file; it can instead take the form of streams of data passed between applications, or of sets of database fields, or of *XML infosets*, in which blocks of information contain just enough information to describe where they fit in the schema structure. In JAXB, the unmarshalling process supports *validation* of the XML input document against the constraints defined in the source schema. This validation process is optional, however, and there may be cases in which you know by other means that an input document is valid and so you may choose for performance reasons to skip validation during unmarshalling. In any case, validation before (by means of a third-party application) or during unmarshalling is important, because it assures that an XML document generated during marshalling will also be valid with respect to the source schema. Validation is discussed more later in this chapter.
XML Output Documents	XML content that is marshalled out to an XML document. In JAXB, marshalling involves parsing an XML content object tree and writing out an XML document that is an accurate representation of the original XML document, and is valid with respect the source schema. JAXB can marshal XML data to XML documents, SAX content handlers, and DOM nodes.

The JAXB Binding Process

Figure 1–2 shows what occurs during the JAXB binding process.

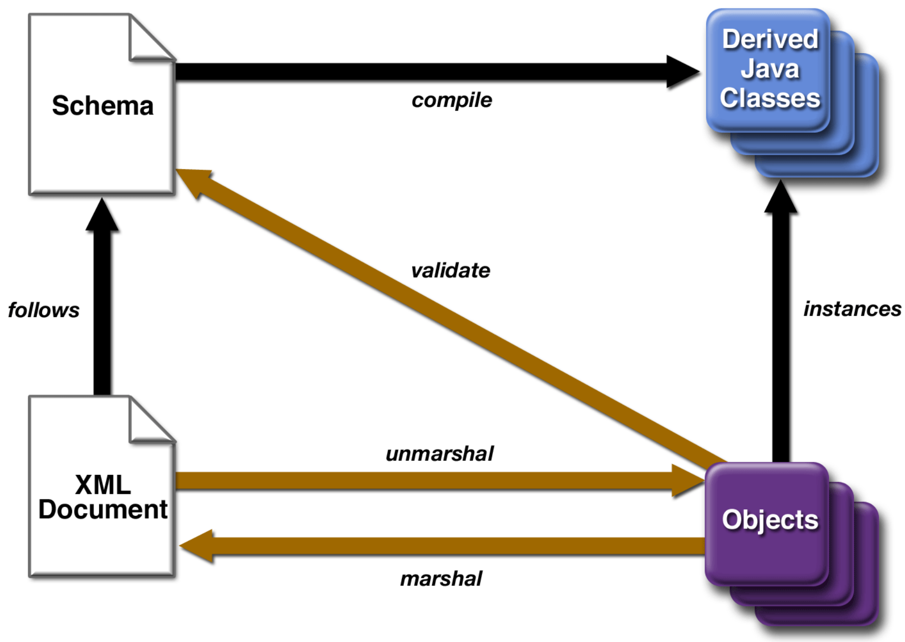

Figure 1–2 Steps in the JAXB Binding Process

The general steps in the JAXB data binding process are:

1. Generate classes. An XML schema is used as input to the JAXB binding compiler to generate JAXB classes based on that schema.
2. Compile classes. All of the generated classes, source files, and application code must be compiled.
3. Unmarshal. XML documents written according to the constraints in the source schema are unmarshalled by the JAXB binding framework. Note that JAXB also supports unmarshalling XML data from sources other than files/documents, such as DOM nodes, string buffers, SAX Sources, and so forth.
4. Generate content tree. The unmarshalling process generates a content tree of data objects instantiated from the generated JAXB classes; this content tree represents the structure and content of the source XML documents.

5. Validate (optional). The unmarshalling process optionally involves validation of the source XML documents before generating the content tree. Note that if you modify the content tree in Step 6, below, you can also use the JAXB Validate operation to validate the changes before marshalling the content back to an XML document.
6. Process content. The client application can modify the XML data represented by the Java content tree by means of interfaces generated by the binding compiler.
7. Marshal. The processed content tree is marshalled out to one or more XML output documents. The content may be validated before marshalling.

To summarize, using JAXB involves two discrete sets of activities:

- Generate and compile JAXB classes from a source schema, and build an application that implements these classes
- Run the application to unmarshal, process, validate, and marshal XML content through the JAXB binding framework

These two steps are usually performed at separate times in two distinct phases. Typically, for example, there is an application development phase in which JAXB classes are generated and compiled, and a binding implementation is built, followed by a deployment phase in which the generated JAXB classes are used to process XML content in an ongoing "live" production setting.

Note: Unmarshalling is not the only means by which a content tree may be created. Schema-derived content classes also support the programmatic construction of content trees by direct invocation of the appropriate factory methods. Once created, a content tree may be revalidated, either in whole or in part, at any time. See Create Marshal Example (page 47) for an example of using the `ObjectFactory` class to directly add content to a content tree.

JAXB Binding Framework

The JAXB binding framework is implemented in three Java packages:

- The `javax.xml.bind` package defines abstract classes and interfaces that are used directly with content classes.

  The `javax.xml.bind` package defines the `Unmarshaller`, `Validator`, and `Marshaller` classes, which are auxiliary objects for providing their respective operations.

The `JAXBContext` class is the entry point for a Java application into the JAXB framework. A `JAXBContext` instance manages the binding relationship between XML element names to Java content interfaces for a JAXB implementation to be used by the unmarshal, marshal and validation operations.

The `javax.xml.bind` package also defines a rich hierarchy of validation event and exception classes for use when marshalling or unmarshalling errors occur, when constraints are violated, and when other types of errors are detected.

- The `javax.xml.bind.util` package contains utility classes that may be used by client applications to manage marshalling, unmarshalling, and validation events.
- The `javax.xml.bind.helper` package provides partial default implementations for some of the javax.xml.bind interfaces. Implementations of JAXB can extend these classes and implement the abstract methods. These APIs are not intended to be directly used by applications using JAXB architecture.

The main package in the JAXB binding framework, `javax.xml.bind`, is described in more detail below.

More About javax.xml.bind

The three core functions provided by the primary binding framework package, `javax.xml.bind`, are marshalling, unmarshalling, and validation. The main client entry point into the binding framework is the `JAXBContext` class.

`JAXBContext` provides an abstraction for managing the XML/Java binding information necessary to implement the unmarshal, marshal and validate operations. A client application obtains new instances of this class by means of the `newInstance(contextPath)` method; for example:

```
JAXBContext jc = JAXBContext.newInstance(
"com.acme.foo:com.acme.bar" );
```

The `contextPath` parameter contains a list of Java package names that contain schema-derived interfaces—specifically the interfaces generated by the JAXB binding compiler. The value of this parameter initializes the `JAXBContext` object to enable management of the schema-derived interfaces. To this end, the JAXB provider implementation must supply an implementation class containing a method with the following signature:

```
public static JAXBContext createContext( String contextPath,
ClassLoader classLoader )

      throws JAXBException;
```

Note: The JAXB provider implementation must generate a `jaxb.properties` file in each package containing schema-derived classes. This property file must contain a property named `javax.xml.bind.context.factory` whose value is the name of the class that implements the `createContext` API.

The class supplied by the provider does not have to be assignable to `javax.xml.bind.JAXBContext`, it simply has to provide a class that implements the `createContext` API. By allowing for multiple Java packages to be specified, the `JAXBContext` instance allows for the management of multiple schemas at one time.

More About Unmarshalling

The `Unmarshaller` class in the `javax.xml.bind` package provides the client application the ability to convert XML data into a tree of Java content objects. The `unmarshal` method for a schema (within a namespace) allows for any global XML element declared in the schema to be unmarshalled as the root of an instance document. The `JAXBContext` object allows the merging of global elements across a set of schemas (listed in the `contextPath`). Since each schema in the schema set can belong to distinct namespaces, the unification of schemas to an unmarshalling context should be namespace-independent. This means that a client application is able to unmarshal XML documents that are instances of any of the schemas listed in the `contextPath`; for example:

```
JAXBContext jc = JAXBContext.newInstance(
  "com.acme.foo:com.acme.bar" );

Unmarshaller u = jc.createUnmarshaller();

FooObject fooObj =
  (FooObject)u.unmarshal( new File( "foo.xml" ) ); // ok

BarObject barObj =
  (BarObject)u.unmarshal( new File( "bar.xml" ) ); // ok
```

```
BazObject bazObj =
  (BazObject)u.unmarshal( new File( "baz.xml" ) );
  // error, "com.acme.baz" not in contextPath
```

A client application may also generate Java content trees explicitly rather than unmarshalling existing XML data. To do so, the application needs to have access and knowledge about each of the schema-derived `ObjectFactory` classes that exist in each of Java packages contained in the `contextPath`. For each schema-derived Java class, there will be a static factory method that produces objects of that type. For example, assume that after compiling a schema, you have a package `com.acme.foo` that contains a schema-derived interface named `PurchaseOrder`. To create objects of that type, the client application would use the following factory method:

```
ObjectFactory objFactory = new ObjectFactory();

com.acme.foo.PurchaseOrder po =
  objFactory.createPurchaseOrder();
```

Note: Because multiple `ObjectFactory` classes are generated when there are multiple packages on the `contextPath`, if you have multiple packages on the `contextPath`, you should use the complete package name when referencing an `ObjectFactory` class in one of those packages.

Once the client application has an instance of the schema-derived object, it can use the mutator methods to set content on it.

Note: The JAXB provider implementation must generate a class in each package that contains all of the necessary object factory methods for that package named `ObjectFactory` as well as the `newInstance(javaContentInterface)` method.

More About Marshalling

The `Marshaller` class in the `javax.xml.bind` package provides the client application the ability to convert a Java content tree back into XML data. There is no difference between marshalling a content tree that is created manually using the factory methods and marshalling a content tree that is the result an unmarshal operation. Clients can marshal a Java content tree back to XML data to a java.io.OutputStream or a java.io.Writer. The marshalling process can alternatively produce SAX2 event streams to a registered ContentHandler or produce a DOM Node object.

A simple example that unmarshals an XML document and then marshals it back out is a follows:

```
JAXBContext jc = JAXBContext.newInstance( "com.acme.foo" );

// unmarshal from foo.xml
Unmarshaller u = jc.createUnmarshaller();
FooObject fooObj =
  (FooObject)u.unmarshal( new File( "foo.xml" ) );

// marshal to System.out
Marshaller m = jc.createMarshaller();
m.marshal( fooObj, System.out );
```

By default, the Marshaller uses UTF-8 encoding when generating XML data to a java.io.OutputStream or a java.io.Writer. Use the setProperty API to change the output encoding used during these marshal operations. Client applications are expected to supply a valid character encoding name as defined in the W3C XML 1.0 Recommendation (http://www.w3.org/TR/2000/REC-xml-20001006#charencoding) and supported by your Java Platform.

Client applications are not required to validate the Java content tree prior to calling one of the marshal APIs. There is also no requirement that the Java content tree be valid with respect to its original schema in order to marshal it back into XML data. Different JAXB Providers can support marshalling invalid Java content trees at varying levels, however all JAXB providers must be able to marshal a valid content tree back to XML data. A JAXB provider must throw a MarshalException when it is unable to complete the marshal operation due to invalid content. Some JAXB providers will fully allow marshalling invalid content, others will fail on the first validation error.

Table 1–2 shows the properties that the `Marshaller` class supports.

Table 1–2 Marshaller Properties

Property	**Description**
`jaxb.encoding`	Value must be a `java.lang.String`; the output encoding to use when marshalling the XML data. The `Marshaller` will use "UTF-8" by default if this property is not specified.
`jaxb.formatted.output`	Value must be a `java.lang.Boolean`; controls whether or not the `Marshaller` will format the resulting XML data with line breaks and indentation. A `true` value for this property indicates human readable indented XML data, while a `false` value indicates unformatted XML data. The `Marshaller` defaults to `false` (unformatted) if this property is not specified.
`jaxb.schemaLocation`	Value must be a `java.lang.String`; allows the client application to specify an `xsi:schemaLocation` attribute in the generated XML data. The format of the `schemaLocation` attribute value is discussed in an easy to understand, non-normative form in Section 5.6 of the *W3C XML Schema Part 0: Primer* and specified in Section 2.6 of the *W3C XML Schema Part 1: Structures*.
`jaxb.noNamespaceSchemaLocation`	Value must be a `java.lang.String`; allows the client application to specify an `xsi:noNamespaceSchemaLocation` attribute in the generated XML data.

More About Validation

The `Validator` class in the `javax.xml.bind` package is responsible for controlling the validation of content trees during runtime. When the unmarshalling process incorporates validation and it successfully completes without any validation errors, both the input document and the resulting content tree are guaranteed to be valid. By contrast, the marshalling process does not actually perform validation. If only validated content trees are marshalled, this guarantees that generated XML documents are always valid with respect to the source schema.

Some XML parsers, like SAX and DOM, allow schema validation to be disabled, and there are cases in which you may want to disable schema validation to improve processing speed and/or to process documents containing invalid or incomplete content. JAXB supports these processing scenarios by means of the exception handling you choose implement in your JAXB-enabled application. In general, if a JAXB implementation cannot unambiguously complete unmarshalling or marshalling, it will terminate processing with an exception.

Note: The `Validator` class is responsible for managing On-Demand Validation (see below). The `Unmarshaller` class is responsible for managing Unmarshal-Time Validation during the unmarshal operations. Although there is no formal method of enabling validation during the marshal operations, the `Marshaller` may detect errors, which will be reported to the `ValidationEventHandler` registered on it.

A JAXB client can perform two types of validation:

- **Unmarshal-Time validation** enables a client application to receive information about validation errors and warnings detected while unmarshalling XML data into a Java content tree, and is completely orthogonal to the other types of validation. To enable or disable it, use the `Unmarshaller.setValidating` method. All JAXB Providers are required to support this operation.
- **On-Demand validation** enables a client application to receive information about validation errors and warnings detected in the Java content tree. At any point, client applications can call the `Validator.validate` method on the Java content tree (or any sub-tree of it). All JAXB Providers are required to support this operation.

If the client application does not set an event handler on its `Validator`, `Unmarshaller`, or `Marshaller` prior to calling the validate, unmarshal, or marshal methods, then a default event handler will receive notification of any errors or warnings encountered. The default event handler will cause the current operation to halt after encountering the first error or fatal error (but will attempt to continue after receiving warnings).

There are three ways to handle events encountered during the unmarshal, validate, and marshal operations:

- Use the default event handler.

The default event handler will be used if you do not specify one via the `setEventHandler` APIs on `Validator`, `Unmarshaller`, or `Marshaller`.

- Implement and register a custom event handler.

  Client applications that require sophisticated event processing can implement the `ValidationEventHandler` interface and register it with the `Unmarshaller` and/or `Validator`.

- Use the `ValidationEventCollector` utility.

  For convenience, a specialized event handler is provided that simply collects any `ValidationEvent` objects created during the unmarshal, validate, and marshal operations and returns them to the client application as a `java.util.Collection`.

Validation events are handled differently, depending on how the client application is configured to process them. However, there are certain cases where a JAXB Provider indicates that it is no longer able to reliably detect and report errors. In these cases, the JAXB Provider will set the severity of the `ValidationEvent` to `FATAL_ERROR` to indicate that the unmarshal, validate, or marshal operations should be terminated. The default event handler and `ValidationEventCollector` utility class must terminate processing after being notified of a fatal error. Client applications that supply their own `ValidationEventHandler` should also terminate processing after being notified of a fatal error. If not, unexpected behavior may occur.

XML Schemas

Because XML schemas are such an important component of the JAXB processing model—and because other data binding facilities like JAXP work with DTDs instead of schemas—it is useful to review here some basics about what XML schemas are and how they work.

XML Schemas are a powerful way to describe allowable elements, attributes, entities, and relationships in an XML document. A more robust alternative to DTDs, the purpose of an XML schema is to define classes of XML documents that must adhere to a particular set of structural and data constraints—that is, you may want to define separate schemas for chapter-oriented books, for an online purchase order system, or for a personnel database. In the context of JAXB, an XML document containing data that is constrained by an XML schema is referred to as a *document instance*, and the structure and data within a document instance is referred to as a *content tree*.

Note: In practice, the term “document” is not always accurate, as an XML instance document does not have to be a completely formed, selfstanding document file; it can instead take the form of streams of data passed between applications, or of sets of database fields, or of *XML infosets* in which blocks of information contain just enough information to describe where they fit in the schema structure.

The following sample code is taken from the W3C's *Schema Part 0: Primer* (`http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/`), and illustrates an XML document, `po.xml`, for a simple purchase order.

```
<?xml version="1.0"?>
<purchaseOrder orderDate="1999-10-20">
  <shipTo country="US">
    <name>Alice Smith</name>
    <street>123 Maple Street</street>
    <city>Mill Valley</city>
    <state>CA</state>
    <zip>90952</zip>
  </shipTo>
  <billTo country="US">
    <name>Robert Smith</name>
    <street>8 Oak Avenue</street>
    <city>Old Town</city>
    <state>PA</state>
    <zip>95819</zip>
  </billTo>
<comment>Hurry, my lawn is going wild!</comment>
  <items>
    <item partNum="872-AA">
      <productName>Lawnmower</productName>
      <quantity>1</quantity>
      <USPrice>148.95</USPrice>
      <comment>Confirm this is electric</comment>
    </item>
    <item partNum="926-AA">
      <productName>Baby Monitor</productName>
      <quantity>1</quantity>
      <USPrice>39.98</USPrice>
      <shipDate>1999-05-21</shipDate>
    </item>
  </items>
</purchaseOrder>
```

The root element, `purchaseOrder`, contains the child elements `shipTo`, `billTo`, `comment`, and `items`. All of these child elements except `comment` contain other child elements. The leaves of the tree are the child elements like `name`, `street`, `city`, and `state`, which do not contain any further child elements. Elements that contain other child elements or can accept attributes are referred to as *complex types*. Elements that contain only `PCDATA` and no child elements are referred to as *simple types*.

The complex types and some of the simple types in `po.xml` are defined in the purchase order schema below. Again, this example schema, `po.xsd`, is derived from the W3C's *Schema Part 0: Primer* (`http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/`).

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="purchaseOrder" type="PurchaseOrderType"/>
<xsd:element name="comment" type="xsd:string"/>
<xsd:complexType name="PurchaseOrderType">
  <xsd:sequence>
    <xsd:element name="shipTo" type="USAddress"/>
    <xsd:element name="billTo" type="USAddress"/>
    <xsd:element ref="comment" minOccurs="0"/>
    <xsd:element name="items" type="Items"/>
  </xsd:sequence>
  <xsd:attribute name="orderDate" type="xsd:date"/>
</xsd:complexType>

<xsd:complexType name="USAddress">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="street" type="xsd:string"/>
    <xsd:element name="city" type="xsd:string"/>
    <xsd:element name="state" type="xsd:string"/>
    <xsd:element name="zip" type="xsd:decimal"/>
  </xsd:sequence>
  <xsd:attribute name="country" type="xsd:NMTOKEN"
       fixed="US"/>
</xsd:complexType>

<xsd:complexType name="Items">
  <xsd:sequence>
    <xsd:element name="item" minOccurs="1"
                 maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="productName"
                       type="xsd:string"/>
          <xsd:element name="quantity">
            <xsd:simpleType>
              <xsd:restriction base="xsd:positiveInteger">
```

```
                    <xsd:maxExclusive value="100"/>
                  </xsd:restriction>
                </xsd:simpleType>
              </xsd:element>
              <xsd:element name="USPrice" type="xsd:decimal"/>
              <xsd:element ref="comment" minOccurs="0"/>
              <xsd:element name="shipDate" type="xsd:date"
                           minOccurs="0"/>
            </xsd:sequence>
            <xsd:attribute name="partNum" type="SKU"
                           use="required"/>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>

    <!-- Stock Keeping Unit, a code for identifying products -->
    <xsd:simpleType name="SKU">
      <xsd:restriction base="xsd:string">
        <xsd:pattern value="\d{3}-[A-Z]{2}"/>
      </xsd:restriction>
    </xsd:simpleType>

    </xsd:schema>
```

In this example, the schema comprises, similar to a DTD, a main or root `schema` element and several child elements, `element`, `complexType`, and `simpleType`. Unlike a DTD, this schema also specifies as attributes data types like `decimal`, `date`, `fixed`, and `string`. The schema also specifies constraints like `pattern value`, `minOccurs`, and `positiveInteger`, among others. In DTDs, you can only specify data types for textual data (`PCDATA` and `CDATA`); XML schema supports more complex textual and numeric data types and constraints, all of which have direct analogs in the Java language.

Note that every element in this schema has the prefix `xsd:`, which is associated with the W3C XML Schema namespace. To this end, the namespace declaration, `xmlns:xsd="http://www.w3.org/2001/XMLSchema"`, is declared as an attribute to the `schema` element.

Namespace support is another important feature of XML schemas because it provides a means to differentiate between elements written against different schemas or used for varying purposes, but which may happen to have the same name as other elements in a document. For example, suppose you declared two namespaces in your schema, one for `foo` and another for `bar`. Two XML documents are combined, one from a billing database and another from an shipping database, each of which was written against a different schema. By specifying namespaces in your schema, you can differentiate between, say, `foo:address` and `bar:address`.

Representing XML Content

This section describes how JAXB represents XML content as Java objects. Specifically, the topics in this section are as follows:

- Binding XML Names to Java Identifiers
- Java Representation of XML Schema

Binding XML Names to Java Identifiers

XML schema languages use *XML names*—strings that match the *Name* production defined in *XML 1.0 (Second Edition)* (`http://www.w3.org/XML/`) to label schema components. This set of strings is much larger than the set of valid Java class, method, and constant identifiers. To resolve this discrepancy, JAXB uses several name-mapping algorithms.

The JAXB name-mapping algorithm maps XML names to Java identifiers in a way that adheres to standard Java API design guidelines, generates identifiers that retain obvious connections to the corresponding schema, and is unlikely to result in many collisions.

Refer to Chapter 2 for information about changing default XML name mappings. See Appendix C in the *JAXB Specification* for complete details about the JAXB naming algorithm.

Java Representation of XML Schema

JAXB supports the grouping of generated classes and interfaces in Java packages. A package comprises:

- A name, which is either derived directly from the XML namespace URI, or specified by a binding customization of the XML namespace URI
- A set of Java content interfaces representing the content models declared within the schema
- A Set of Java element interfaces representing element declarations occurring within the schema

- An `ObjectFactory` class containing:
    - An instance factory method for each Java content interface and Java element interface within the package; for example, given a Java content interface named `Foo`, the derived factory method would be:

      ```
      public Foo createFoo() throws JAXBException;
      ```

    - Dynamic instance factory allocator; creates an instance of the specified Java content interface; for example:

      ```
      public Object newInstance(Class javaContentInterface)
         throws JAXBException;
      ```

    - `getProperty` and `setProperty` APIs that allow the manipulation of provider-specified properties
- Set of typesafe enum classes
- Package javadoc

Binding XML Schemas

This section describes the default XML-to-Java bindings used by JAXB. All of these bindings can be overridden on global or case-by-case levels by means of a custom binding declaration. The topics in this section are as follows:

- Simple Type Definitions
- Default Data Type Bindings
- Default Binding Rules Summary

See the *JAXB Specification* for complete information about the default JAXB bindings.

Simple Type Definitions

A schema component using a simple type definition typically binds to a Java property. Since there are different kinds of such schema components, the following Java property attributes (common to the schema components) include:

- Base type
- Collection type, if any

- Predicate

The rest of the Java property attributes are specified in the schema component using the `simple` type definition.

Default Data Type Bindings

The Java language provides a richer set of data type than XML schema. Table 1–3 lists the mapping of XML data types to Java data types in JAXB.

Table 1–3 JAXB Mapping of XML Schema Built-in Data Types

XML Schema Type	Java Data Type
`xsd:string`	`java.lang.String`
`xsd:integer`	`java.math.BigInteger`
`xsd:int`	`int`
`xsd.long`	`long`
`xsd:short`	`short`
`xsd:decimal`	`java.math.BigDecimal`
`xsd:float`	`float`
`xsd:double`	`double`
`xsd:boolean`	`boolean`
`xsd:byte`	`byte`
`xsd:QName`	`javax.xml.namespace.QName`
`xsd:dateTime`	`java.util.Calendar`
`xsd:base64Binary`	`byte[]`
`xsd:hexBinary`	`byte[]`
`xsd:unsignedInt`	`long`
`xsd:unsignedShort`	`int`
`xsd:unsignedByte`	`short`

Table 1–3 JAXB Mapping of XML Schema Built-in Data Types (Continued)

XML Schema Type	Java Data Type
`xsd:time`	`java.util.Calendar`
`xsd:date`	`java.util.Calendar`
`xsd:anySimpleType`	`java.lang.String`

Default Binding Rules Summary

The JAXB binding model follows the default binding rules summarized below:

- Bind the following to Java package:
  - XML Namespace URI
- Bind the following XML Schema components to Java content interface:
  - Named complex type
  - Anonymous inlined type definition of an element declaration
- Bind to typesafe enum class:
  - A named simple type definition with a basetype that derives from "`xsd:NCName`" and has enumeration facets.
- Bind the following XML Schema components to a Java Element interface:
  - A global element declaration to a Element interface.
  - Local element declaration that can be inserted into a general content list.
- Bind to Java property:
  - Attribute use
  - Particle with a term that is an element reference or local element declaration.
- Bind model group with a repeating occurrence and complex type definitions with mixed `{content type}` to:
  - A general content property; a List content-property that holds Java instances representing element information items and character data items.

Customizing JAXB Bindings

The default JAXB bindings can be overridden at a global scope or on a case-by-case basis as needed by using custom binding declarations. As described previously, JAXB uses default binding rules that can be customized by means of binding declarations made in either of two ways:

- As inline annotations in a source XML schema
- As declarations in an external binding customizations file that is passed to the JAXB binding compiler

Custom JAXB binding declarations also allow you to customize your generated JAXB classes beyond the XML-specific constraints in an XML schema to include Java-specific refinements such as class and package name mappings.

You do not need to provide a binding instruction for every declaration in your schema to generate Java classes. For example, the binding compiler uses a general name-mapping algorithm to bind XML names to names that are acceptable in the Java programming language. However, if you want to use a different naming scheme for your classes, you can specify custom binding declarations to make the binding compiler generate different names. There are many other customizations you can make with the binding declaration, including:

- Name the package, derived classes, and methods
- Assign types to the methods within the derived classes
- Choose which elements to bind to classes
- Decide how to bind each attribute and element declaration to a property in the appropriate content class
- Choose the type of each attribute-value or content specification

---

Note: Relying on the default JAXB binding behavior rather than requiring a binding declaration for each XML Schema component bound to a Java representation makes it easier to keep pace with changes in the source schema. In most cases, the default rules are robust enough that a usable binding can be produced with no custom binding declaration at all.

Code examples showing how to customize JAXB bindings are provided in Chapter 2.

Scope

When a customization value is defined in a binding declaration, it is associated with a *scope*. A scope of a customization value is the set of schema elements to which it applies. If a customization value applies to a schema element, then the schema element is said to be covered by the scope of the customization value.

Table 1–4 lists the four scopes for custom bindings.

Table 1–4 Custom Binding Scopes

Scope	**Description**
Global	A customization value defined in `<globalBindings>` has global scope. A global scope covers all the schema elements in the source schema and (recursively) any schemas that are included or imported by the source schema.
Schema	A customization value defined in `<schemaBindings>` has schema scope. A schema scope covers all the schema elements in the target name space of a schema.
Definition	A customization value in binding declarations of a type definition and global declaration has definition scope. A definition scope covers all schema elements that reference the type definition or the global declaration.
Component	A customization value in a binding declaration has component scope if the customization value applies only to the schema element that was annotated with the binding declaration.

Scope Inheritance

The different scopes form a taxonomy. The taxonomy defines both the inheritance and overriding semantics of customization values. A customization value defined in one scope is inherited for use in a binding declaration covered by another scope as shown by the following inheritance hierarchy:

- A schema element in schema scope inherits a customization value defined in global scope.
- A schema element in definition scope inherits a customization value defined in schema or global scope.
- A schema element in component scope inherits a customization value defined in definition, schema or global scope.

Similarly, a customization value defined in one scope can override a customization value inherited from another scope as shown below:

- Value in schema scope overrides a value inherited from global scope.
- Value in definition scope overrides a value inherited from schema scope or global scope.
- Value in component scope overrides a value inherited from definition, schema or global scope.

What is Not Supported

See Section E.2, “Not Required XML Schema Concepts,” in the *JAXB Specification* for the latest information about unsupported or non-required schema concepts.

JAXB APIs and Tools

The JAXB APIs and tools are shipped in the `jaxb` subdirectory of the Java WSDP. This directory contains sample applications, a JAXB binding compiler (`xjc`), and implementations of the runtime binding framework APIs contained in the `javax.xml.bind` package. For instructions on using the JAXB, see Chapter 2.

2 Using JAXB

THIS chapter provides instructions for using several of the sample Java applications that were included in the Java WSDP. These examples demonstrate and build upon key JAXB features and concepts. It is recommended that you follow these procedures in the order presented.

After reading this chapter, you should feel comfortable enough with JAXB that you can:

- Generate JAXB Java classes from an XML schema
- Use schema-derived JAXB classes to unmarshal and marshal XML content in a Java application
- Create a Java content tree from scratch using schema-derived JAXB classes
- Validate XML content during unmarshalling and at runtime
- Customize JAXB schema-to-Java bindings

The primary goals of the basic examples are to highlight the core set of JAXB functions using default settings and bindings. After familiarizing yourself with these core features and functions, you may wish to continue with Customizing JAXB Bindings (page 54) for instructions on using five additional examples that demonstrate how to modify the default JAXB bindings.

Note: The Purchase Order schema, `po.xsd`, and the Purchase Order XML file, `po.xml`, used in these samples are derived from the W3C XML Schema Part 0: Primer (`http://www.w3.org/TR/xmlschema-0/`), edited by David C. Fallside.

General Usage Instructions

This section provides general usage instructions for the examples used in this chapter, including how to build and run the applications using the Ant build tool, and provides details about the default schema-to-JAXB bindings used in these examples.

Description

This chapter describes ten examples; the basic examples (Unmarshal Read, Modify Marshal, Create Marshal, Unmarshal Validate, Validate-On-Demand) demonstrate basic JAXB concepts like ummarshalling, marshalling, and validating XML content, while the customize examples (Customize Inline, Datatype Converter, External Customize, Fix Collides, Bind Choice) demonstrate various ways of customizing the binding of XML schemas to Java objects. Each of the examples in this chapter is based on a *Purchase Order* scenario. With the exception of the Bind Choice and the Fix Collides examples, each uses an XML document, `po.xml`, written against an XML schema, `po.xsd`.

Table 2–1 Sample JAXB Application Descriptions

Example Name	**Description**
Unmarshal Read Example	Demonstrates how to unmarshal an XML document into a Java content tree and access the data contained within it.
Modify Marshal Example	Demonstrates how to modify a Java content tree.
Create Marshal Example	Demonstrates how to use the *`ObjectFactory`* class to create a Java content tree from scratch and then marshal it to XML data.
Unmarshal Validate Example	Demonstrates how to enable validation during unmarshalling.
Validate-On-Demand Example	Demonstrates how to validate a Java content tree at runtime.
Customize Inline Example	Demonstrates how to customize the default JAXB bindings by means of inline annotations in an XML schema.

Table 2–1 Sample JAXB Application Descriptions

Example Name	Description
Datatype Converter Example	Similar to the Customize Inline example, this example illustrates alternate, more terse bindings of XML `simpleType` definitions to Java datatypes.
External Customize Example	Illustrates how to use an external binding declarations file to pass binding customizations for a read-only schema to the JAXB binding compiler.
Fix Collides Example	Illustrates how to use customizations to resolve name conflicts reported by the JAXB binding compiler. It is recommended that you first run `ant fail` in the application directory to see the errors reported by the JAXB binding compiler, and then look at `binding.xjb` to see how the errors were resolved. Running `ant` alone uses the binding customizations to resolve the name conflicts while compiling the schema.
Bind Choice Example	Illustrates how to bind a `choice` model group to a Java interface.

Note: These examples are all located in the `$JWSDP_HOME/jaxb/samples` directory.

Each example directory contains several base files:

- `po.xsd` is the XML schema you will use as input to the JAXB binding compiler, and from which schema-derived JAXB Java classes will be generated. For the Customize Inline and Datatype Converter examples, this file contains inline binding customizations. Note that the Bind Choice and Fix Collides examples use `example.xsd` rather than `po.xsd`.
- `po.xml` is the *Purchase Order* XML file containing sample XML content, and is the file you will unmarshal into a Java content tree in each example. This file is almost exactly the same in each example, with minor content

differences to highlight different JAXB concepts. Note that the Bind Choice and Fix Collides examples use `example.xml` rather than `po.xml`.

- `Main.java` is the main Java class for each example.
- `build.xml` is an Ant project file provided for your convenience. Use Ant to generate, compile, and run the schema-derived JAXB classes automatically. The `build.xml` file varies across the examples.
- `MyDatatypeConverter.java` in the `inline-customize` example is a Java class used to provide custom datatype conversions.
- `binding.xjb` in the External Customize, Bind Choice, and Fix Collides examples is an external binding declarations file that is passed to the JAXB binding compiler to customize the default JAXB bindings.
- `example.xsd` in the Fix Collides example is a short schema file that contains deliberate naming conflicts, to show how to resolve such conflicts with custom JAXB bindings.

Using the Examples

As with all applications that implement schema-derived JAXB classes, as described above, there are two distinct phases in using JAXB:

1. Generating and compiling JAXB Java classes from an XML source schema
2. Unmarshalling, validating, processing, and marshalling XML content

In the case of these examples, you perform these steps by using `Ant` with the `build.xml` project file included in each example directory.

Configuring and Running the Samples

The `build.xml` file included in each example directory is an Ant project file that, when run, automatically performs the following steps:

1. Updates your `CLASSPATH` to include the necessary schema-derived JAXB classes.
2. Runs the JAXB binding compiler to generate JAXB Java classes from the XML source schema, `po.xsd`, and puts the classes in a package named `primer.po`.
3. Generates API documentation from the schema-derived JAXB classes using the Javadoc tool.

4. Compiles the schema-derived JAXB classes.
5. Runs the `Main` class for the example.

Solaris/Linux

1. Set the following environment variables:

   ```
   export JAVA_HOME=<your J2SE installation directory>
   export JWSDP_HOME=<your JWSDP installation directory>
   ```

2. Change to the desired example directory.

   For example, to run the Unmarshal Read example:

   ```
   cd <JWSDP_HOME>/jaxb/samples/unmarshal-read
   ```

   (*<JWSDP_HOME>* is the directory where you installed the Java WSDP bundle.)

3. Run ant:

   ```
   $JWSDP_HOME/apache-ant/bin/ant -emacs
   ```

4. Repeat these steps for each example.

Windows NT/2000/XP

1. Set the following environment variables:

   ```
   set JAVA_HOME=<your J2SE installation directory>
   set JWSDP_HOME=<your JWSDP installation directory>
   ```

2. Change to the desired example directory.

   For example, to run the Unmarshal Read example:

   ```
   cd <JWSDP_HOME>\jaxb\samples\unmarshal-read
   ```

   (*<JWSDP_HOME>* is the directory where you installed the Java WSDP bundle.)

3. Run ant:

   ```
   %JWSDP_HOME%\apache-ant\bin\ant -emacs
   ```

4. Repeat these steps for each example.

The schema-derived JAXB classes and how they are bound to the source schema is described in About the Schema-to-Java Bindings (page 32). The methods used for building and processing the Java content tree are described in Basic Examples (page 43).

JAXB Compiler Options

The JAXB schema binding compiler is located in the `<JWSDP_HOME>/jaxb/bin` directory. There are two scripts in this directory: `xjc.sh` (Solaris/Linux) and `xjc.bat` (Windows).

Both `xjc.sh` and `xjc.bat` take the same command-line options. You can display quick usage instructions by invoking the scripts without any options, or with the `-help` switch. The syntax is as follows:

```
   xjc [-options ...] <schema>
```

The `xjc` command-line options are listed in Table 2–2.

Table 2–2 `xjc` Command-Line Options

Option or Argument	**Description**
`<schema>`	One or more schema files to compile.
`-nv`	Do not perform strict validation of the input schema(s). By default, `xjc` performs strict validation of the source schema before processing. Note that this does not mean the binding compiler will not perform any validation; it simply means that it will perform less-strict validation.
`-extension`	By default, `xjc` strictly enforces the rules outlined in the Compatibility chapter of the *JAXB Specification*. Specifically, Appendix E.2 defines a set of W3C XML Schema features that are not completely supported by JAXB v1.0. In some cases, you may be able to use these extensions with the `-extension` switch. In the default (strict) mode, you are also limited to using only the binding customizations defined in the specification. By using the `-extension` switch, you can enable the JAXB Vendor Extensions.

Table 2–2 `xjc` Command-Line Options (Continued)

Option or Argument	Description

`-b` *`<file>`*	Specify one or more external binding files to process (each binding file must have it's own `-b` switch). The syntax of the external binding files is extremely flexible. You may have a single binding file that contains customizations for multiple schemas, or you can break the customizations into multiple bindings files; for example:
	
	`xjc schema1.xsd schema2.xsd schema3.xsd -b bindings123.xjb`
	`xjc schema1.xsd schema2.xsd schema3.xsd -b bindings1.xjb -b bindings2.xjb -b bindings3.xjb`
	
	Note that the ordering of schema files and binding files on the command line does not matter.

`-d` *`<dir>`*	By default, `xjc` will generate Java content classes in the current directory. Use this option to specify an alternate output directory. The directory must already exist; `xjc` will not create it for you.
`-p` *`<pkg>`*	Specifies the target package for schema-derived classes. This option overrides any binding customization for package name as well as the default package name algorithm defined in the *JAXB Specification*.
`-host` *`<proxyHost>`*	Set `http.proxyHost` to *`<proxyHost>`*.
`-port` *`<proxyPort>`*	Set `http.proxyPort` to *`<proxyPort>`*.
`-classpath` *`<arg>`*	Specify where to find client application class files used by the `<jxb:javaType>` and `<xjc:superClass>` customizations.
`-catalog` *`<file>`*	Specify catalog files to resolve external entity references.Supports TR9401, XCatalog, and OASIS XML Catalog format.
`-readOnly`	Generated source files will be marked read-only. By default, `xjc` does not write-protect the schema-derived source files it generates.
`-use-runtime` *`<pkg>`*	Suppress the generation of the `impl.runtime` package and refer to another existing runtime in the specified package. This option is useful when you are compiling multiple independent schemas. Because the generated impl.runtime packages are identical, except for their package declarations, you can reduce the size of your generated codebase by telling the compiler to reuse an existing `impl.runtime` package.

Table 2–2 `xjc` Command-Line Options (Continued)

Option or Argument	**Description**
`-xmlschema`	Treat input schemas as W3C XML Schema (default). If you do not specify this switch, your input schemas will be treated as W3C XML Schema.
`-relaxng`	Treat input schemas as RELAX NG (experimental, unsupported). Support for RELAX NG schemas is provided as a JAXB Vendor Extension.
`-dtd`	Treat input schemas as XML DTD (experimental, unsupported). Support for RELAX NG schemas is provided as a JAXB Vendor Extension.
`-help`	Display this help message.

The command invoked by the `xjc.sh` and `xjc.bat` scripts is equivalent to the Java command:

```
$JAVA_HOME/bin/java -jar $JAXB_HOME/lib/jaxb-xjc.jar
```

About the Schema-to-Java Bindings

When you run the JAXB binding compiler against the `po.xsd` XML schema used in the basic examples (Unmarshal Read, Modify Marshal, Create Marshal, Unmarshal Validate, Validate-On-Demand), the JAXB binding compiler generates a Java package named `primer.po` containing eleven classes, making a total of twelve classes in each of the basic examples:

Table 2–3 Schema-Derived JAXB Classes in the Basic Examples

Class	**Description**
`primer/po/Comment.java`	Public interface extending `javax.xml.bind.Element`; binds to the global schema `element` named `comment`. Note that JAXB generates element interfaces for all global element declarations.
`primer/po/Items.java`	Public interface that binds to the schema `complexType` named `Items`.

Table 2–3 Schema-Derived JAXB Classes in the Basic Examples (Continued)

Class	Description
`primer/po/ObjectFactory.java`	Public class extending `com.sun.xml.bind.DefaultJAXBContextImpl`; used to create instances of specified interfaces. For example, the `ObjectFactory` `createComment()` method instantiates a `Comment` object.
`primer/po/PurchaseOrder.java`	Public interface extending `javax.xml.bind.Element`, and `PurchaseOrderType`; binds to the global schema `element` named `PurchaseOrder`.
`primer/po/PurchaseOrderType.java`	Public interface that binds to the schema `complexType` named `PurchaseOrderType`.
`primer/po/USAddress.java`	Public interface that binds to the schema `complexType` named `USAddress`.
`primer/po/impl/CommentImpl.java`	Implementation of `Comment.java`.
`primer/po/impl/ItemsImpl.java`	Implementation of `Items.java`
`primer/po/impl/PurchaseOrderImpl.java`	Implementation of `PurchaseOrder.java`
`primer/po/impl/PurchaseOrderTypeImpl.java`	Implementation of `PurchaseOrderType.java`
`primer/po/impl/USAddressImpl.java`	Implementation of `USAddress.java`

Note: You should never directly use the generated implementation classes—that is, `*Impl.java` in the *`<packagename>`*`/impl` directory. These classes are not directly referenceable because the class names in this directory are not standardized by the JAXB specification. The `ObjectFactory` method is the only portable means to create an instance of a schema-derived interface. There is also an `ObjectFactory.newInstance(Class JAXBinterface)` method that enables you to create instances of interfaces.

These classes and their specific bindings to the source XML schema for the basic examples are described below.

Table 2–4 Schema-to-Java Bindings for the Basic Examples

XML Schema	JAXB Binding
`<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">`	
`<xsd:element name="purchaseOrder" type="PurchaseOrderType"/>`	`PurchaseOrder.java`
`<xsd:element name="comment" type="xsd:string"/>`	`Comment.java`

`<xsd:complexType name="PurchaseOrderType">`
  `<xsd:sequence>`
    `<xsd:element name="shipTo" type="USAddress"/>`
    `<xsd:element name="billTo" type="USAddress"/>`
    `<xsd:element ref="comment" minOccurs="0"/>`
    `<xsd:element name="items" type="Items"/>`
  `</xsd:sequence>`
  `<xsd:attribute name="orderDate" type="xsd:date"/>`
`</xsd:complexType>`
	`PurchaseOrder-`
	`Type.java`

`<xsd:complexType name="USAddress">`
  `<xsd:sequence>`
    `<xsd:element name="name" type="xsd:string"/>`
    `<xsd:element name="street" type="xsd:string"/>`
    `<xsd:element name="city" type="xsd:string"/>`
    `<xsd:element name="state" type="xsd:string"/>`
    `<xsd:element name="zip" type="xsd:decimal"/>`
  `</xsd:sequence>`
`<xsd:attribute name="country" type="xsd:NMTOKEN" fixed="US"/>`
`</xsd:complexType>`
	`USAddress.java`

`<xsd:complexType name="Items">`
  `<xsd:sequence>`
    `<xsd:element name="item" minOccurs="1" maxOc-`
`curs="unbounded">`
	`Items.java`

Table 2–4 Schema-to-Java Bindings for the Basic Examples (Continued)

XML Schema	JAXB Binding
`      <xsd:complexType>`	
`        <xsd:sequence>`	
`         <xsd:element name="productName" type="xsd:string"/>`	
`         <xsd:element name="quantity">`	
`           <xsd:simpleType>`	
`             <xsd:restriction base="xsd:positiveInteger">`	
`               <xsd:maxExclusive value="100"/>`	
`             </xsd:restriction>`	`Items.ItemType`
`           </xsd:simpleType>`	
`         </xsd:element>`	
`         <xsd:element name="USPrice" type="xsd:decimal"/>`	
`         <xsd:element ref="comment" minOccurs="0"/>`	
`<xsd:element name="shipDate" type="xsd:date" minOccurs="0"/>`	
`        </xsd:sequence>`	
`  <xsd:attribute name="partNum" type="SKU" use="required"/>`	
`      </xsd:complexType>`	
`    </xsd:element>` `  </xsd:sequence>` `</xsd:complexType>`	
`<!-- Stock Keeping Unit, a code for identifying products -->`	
`<xsd:simpleType name="SKU">` `  <xsd:restriction base="xsd:string">` `    <xsd:pattern value="\d{3}-[A-Z]{2}"/>` `  </xsd:restriction>` `</xsd:simpleType>`	
`</xsd:schema>`	

Schema-Derived JAXB Classes

The code for the individual classes generated by the JAXB binding compiler for the basic examples is listed below, followed by brief explanations of its functions. The classes listed here are:

- `Comment.java`
- `Items.java`
- `ObjectFactory.java`
- `PurchaseOrder.java`
- `PurchaseOrderType.java`
- `USAddress.java`

Comment.java

In `Comment.java`:

- The `Comment.java` class is part of the `primer.po` package.
- `Comment` is a public interface that extends `javax.xml.bind.Element`.
- Content in instantiations of this class bind to the XML schema element named `comment`.
- The `getValue()` and `setValue()` methods are used to get and set strings representing XML `comment` elements in the Java content tree.

The `Comment.java` code looks like this:

```
package primer.po;

public interface Comment
    extends javax.xml.bind.Element
{

    String getValue();
    void setValue(String value);
}
```

Items.java

In `Items.java`, below:

- The `Items.java` class is part of the `primer.po` package.
- The class provides public interfaces for `Items` and `ItemType`.
- Content in instantiations of this class bind to the XML ComplexTypes `Items` and its child element `ItemType`.
- `Item` provides the `getItem()` method.
- `ItemType` provides methods for:
  - `getPartNum();`
  - `setPartNum(String value);`
  - `getComment();`
  - `setComment(java.lang.String value);`
  - `getUSPrice();`
  - `setUSPrice(java.math.BigDecimal value);`
  - `getProductName();`
  - `setProductName(String value);`
  - `getShipDate();`

- `setShipDate(java.util.Calendar value);`
- `getQuantity();`
- `setQuantity(java.math.BigInteger value);`

The `Items.java` code looks like this:

```
package primer.po;

public interface Items {
   java.util.List getItem();

   public interface ItemType {
        String getPartNum();
        void setPartNum(String value);
        java.lang.String getComment();
        void setComment(java.lang.String value);
        java.math.BigDecimal getUSPrice();
        void setUSPrice(java.math.BigDecimal value);
        String getProductName();
        void setProductName(String value);
        java.util.Calendar getShipDate();
        void setShipDate(java.util.Calendar value);
        java.math.BigInteger getQuantity();
        void setQuantity(java.math.BigInteger value);
    }
}
```

ObjectFactory.java

In `ObjectFactory.java`, below:

- The `ObjectFactory` class is part of the `primer.po` package.
- `ObjectFactory` provides factory methods for instantiating Java interfaces representing XML content in the Java content tree.
- Method names are generated by concatenating:
  - The string constant `create`
  - If the Java content interface is nested within another interface, then the concatenation of all outer Java class names
  - The name of the Java content interface
  - JAXB implementation-specific code was removed in this example to make it easier to read.

For example, in this case, for the Java interface `primer.po.Items.ItemType`, `ObjectFactory` creates the method `createItemsItemType()`.

The ObjectFactory.java code looks like this:

```
package primer.po;

public class ObjectFactory
    extends com.sun.xml.bind.DefaultJAXBContextImpl {

    /**
     * Create a new ObjectFactory that can be used to create
     * new instances of schema derived classes for package:
     * primer.po
     */
    public ObjectFactory() {
        super(new primer.po.ObjectFactory.GrammarInfoImpl());
    }

    /**
     * Create an instance of the specified Java content
     * interface.
     */
    public Object newInstance(Class javaContentInterface)
        throws javax.xml.bind.JAXBException
    {
        return super.newInstance(javaContentInterface);
    }

    /**
     * Get the specified property. This method can only be
     * used to get provider specific properties.
     * Attempting to get an undefined property will result
     * in a PropertyException being thrown.
     */
    public Object getProperty(String name)
        throws javax.xml.bind.PropertyException
    {
        return super.getProperty(name);
    }

    /**
     * Set the specified property. This method can only be
     * used to set provider specific properties.
     * Attempting to set an undefined property will result
     * in a PropertyException being thrown.
     */
    public void setProperty(String name, Object value)
        throws javax.xml.bind.PropertyException
    {
        super.setProperty(name, value);
```

```
}

/**
 * Create an instance of PurchaseOrder
 */
public primer.po.PurchaseOrder createPurchaseOrder()
    throws javax.xml.bind.JAXBException
{
    return ((primer.po.PurchaseOrder)
        newInstance((primer.po.PurchaseOrder.class)));
}

/**
 * Create an instance of ItemsItemType
 */
public primer.po.Items.ItemType createItemsItemType()
    throws javax.xml.bind.JAXBException
{
    return ((primer.po.Items.ItemType)
        newInstance((primer.po.Items.ItemType.class)));
}

/**
 * Create an instance of USAddress
 */
public primer.po.USAddress createUSAddress()
    throws javax.xml.bind.JAXBException
{
    return ((primer.po.USAddress)
        newInstance((primer.po.USAddress.class)));
}

/**
 * Create an instance of Comment
 */
public primer.po.Comment createComment()
    throws javax.xml.bind.JAXBException
{
    return ((primer.po.Comment)
        newInstance((primer.po.Comment.class)));
}

/**
 * Create an instance of Comment
 */
public primer.po.Comment createComment(String value)
    throws javax.xml.bind.JAXBException
{
```

```
            return new primer.po.impl.CommentImpl(value);
        }

        /**
         * Create an instance of Items
         */
        public primer.po.Items createItems()
            throws javax.xml.bind.JAXBException
        {
            return ((primer.po.Items)
                newInstance((primer.po.Items.class)));
        }

        /**
         * Create an instance of PurchaseOrderType
         */
        public primer.po.PurchaseOrderType
    createPurchaseOrderType()
            throws javax.xml.bind.JAXBException
        {
            return ((primer.po.PurchaseOrderType)
                newInstance((primer.po.PurchaseOrderType.class)));
        }
    }
```

PurchaseOrder.java

In `PurchaseOrder.java`, below:

- The `PurchaseOrder` class is part of the `primer.po` package.
- `PurchaseOrder` is a public interface that extends `javax.xml.bind.Element` and `primer.po.PurchaseOrderType`.
- Content in instantiations of this class bind to the XML schema element named `purchaseOrder`.

The `PurchaseOrder.java` code looks like this:

```
package primer.po;

public interface PurchaseOrder
extends javax.xml.bind.Element, primer.po.PurchaseOrderType{
}
```

PurchaseOrderType.java

In `PurchaseOrderType.java`, below:

- The `PurchaseOrderType` class is part of the `primer.po` package.
- Content in instantiations of this class bind to the XML schema child element named `PurchaseOrderType`.
- `PurchaseOrderType` is a public interface that provides the following methods:
  - `getItems();`
  - `setItems(primer.po.Items value);`
  - `getOrderDate();`
  - `setOrderDate(java.util.Calendar value);`
  - `getComment();`
  - `setComment(java.lang.String value);`
  - `getBillTo();`
  - `setBillTo(primer.po.USAddress value);`
  - `getShipTo();`
  - `setShipTo(primer.po.USAddress value);`

The `PurchaseOrderType.java` code looks like this:

```
package primer.po;

public interface PurchaseOrderType {
    primer.po.Items getItems();
    void setItems(primer.po.Items value);
    java.util.Calendar getOrderDate();
    void setOrderDate(java.util.Calendar value);
    java.lang.String getComment();
    void setComment(java.lang.String value);
    primer.po.USAddress getBillTo();
    void setBillTo(primer.po.USAddress value);
    primer.po.USAddress getShipTo();
    void setShipTo(primer.po.USAddress value);
}
```

USAddress.java

In `USAddress.java`, below:

- The `USAddress` class is part of the `primer.po` package.
- Content in instantiations of this class bind to the XML schema element named `USAddress`.
- `USAddress` is a public interface that provides the following methods:
    - `getState();`
    - `setState(String value);`
    - `getZip();`
    - `setZip(java.math.BigDecimal value);`
    - `getCountry();`
    - `setCountry(String value);`
    - `getCity();`
    - `setCity(String value);`
    - `getStreet();`
    - `setStreet(String value);`
    - `getName();`
    - `setName(String value);`

The `USAddress.java` code looks like this:

```
package primer.po;

public interface USAddress {
    String getState();
    void setState(String value);
    java.math.BigDecimal getZip();
    void setZip(java.math.BigDecimal value);
    String getCountry();
    void setCountry(String value);
    String getCity();
    void setCity(String value);
    String getStreet();
    void setStreet(String value);
    String getName();
    void setName(String value);
}
```

Basic Examples

This section describes five basic examples (Unmarshal Read, Modify Marshal, Create Marshal, Unmarshal Validate, Validate-On-Demand) that demonstrate how to:

- Unmarshal an XML document into a Java content tree and access the data contained within it
- Modify a Java content tree
- Use the `ObjectFactory` class to create a Java content tree from scratch and then marshal it to XML data
- Perform validation during unmarshalling
- Validate a Java content tree at runtime

Unmarshal Read Example

The purpose of the Unmarshal Read example is to demonstrate how to unmarshal an XML document into a Java content tree and access the data contained within it.

1. The `<`*`JWSDP_HOME`*`>/jaxb/samples/unmarshal-read/Main.java` class declares imports for four standard Java classes plus three JAXB binding framework classes and the `primer.po` package:

```
import java.io.FileInputStream
import java.io.IOException
import java.util.Iterator
import java.util.List
import javax.xml.bind.JAXBContext
import javax.xml.bind.JAXBException
import javax.xml.bind.Unmarshaller
import primer.po.*;
```

2. A `JAXBContext` instance is created for handling classes generated in `primer.po`.

```
JAXBContext jc = JAXBContext.newInstance( "primer.po" );
```

3. An `Unmarshaller` instance is created.

```
Unmarshaller u = jc.createUnmarshaller();
```

4. `po.xml` is unmarshalled into a Java content tree comprising objects generated by the JAXB binding compiler into the `primer.po` package.

```
PurchaseOrder po =
   (PurchaseOrder)u.unmarshal(
      new FileInputStream( "po.xml" ) );
```

5. A simple string is printed to `system.out` to provide a heading for the purchase order invoice.

```
System.out.println( "Ship the following items to: " );
```

6. `get` and `display` methods are used to parse XML content in preparation for output.

```
USAddress address = po.getShipTo();
displayAddress(address);
Items items = po.getItems();
displayItems(items);
```

7. Basic error handling is implemented.

```
} catch( JAXBException je ) {
   je.printStackTrace();
} catch( IOException ioe ) {
   ioe.printStackTrace();
```

8. The `USAddress` branch of the Java tree is walked, and address information is printed to `system.out`.

```
public static void displayAddress( USAddress address ) {
   // display the address
   System.out.println( "\t" + address.getName() );
   System.out.println( "\t" + address.getStreet() );
   System.out.println( "\t" + address.getCity() +
      ", " + address.getState() +
      " "  + address.getZip() );
   System.out.println( "\t" + address.getCountry() + "\n");
}
```

9. The `Items` list branch is walked, and item information is printed to `system.out`.

```
public static void displayItems( Items items ) {
   // the items object contains a List of
   //primer.po.ItemType objects
   List itemTypeList = items.getItem();
```

10. Walking of the `Items` branch is iterated until all items have been printed.

```
for(Iterator iter = itemTypeList.iterator();
      iter.hasNext();) {
    Items.ItemType item = (Items.ItemType)iter.next();
    System.out.println( "\t" + item.getQuantity() +
      " copies of \"" + item.getProductName() +
      "\"" );
}
```

Sample Output

Running `java Main` for this example produces the following output:

```
Ship the following items to:
   Alice Smith
   123 Maple Street
   Cambridge, MA 12345
   US

   5 copies of "Nosferatu - Special Edition (1929)"
   3 copies of "The Mummy (1959)"
   3 copies of "Godzilla and Mothra: Battle for Earth/Godzilla
     vs. King Ghidora"
```

Modify Marshal Example

The purpose of the Modify Marshal example is to demonstrate how to modify a Java content tree.

1. The `<*JWSDP_HOME*>/jaxb/samples/modify-marshal/Main.java` class declares imports for three standard Java classes plus four JAXB binding framework classes and `primer.po` package:

```
import java.io.FileInputStream;
import java.io.IOException;
import java.math.BigDecimal;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;
import javax.xml.bind.Unmarshaller;
import primer.po.*;
```

2. A `JAXBContext` instance is created for handling classes generated in `primer.po`.

```
JAXBContext jc = JAXBContext.newInstance( "primer.po" );
```

3. An `Unmarshaller` instance is created, and `po.xml` is unmarshalled.

```
Unmarshaller u = jc.createUnmarshaller();
PurchaseOrder po =
  (PurchaseOrder)u.unmarshal(
    new FileInputStream( "po.xml" ) );
```

4. `set` methods are used to modify information in the `address` branch of the content tree.

```
USAddress address = po.getBillTo();
address.setName( "John Bob" );
address.setStreet( "242 Main Street" );
address.setCity( "Beverly Hills" );
address.setState( "CA" );
address.setZip( new BigDecimal( "90210" ) );
```

5. A `Marshaller` instance is created, and the updated XML content is marshalled to `system.out`. The `setProperty` API is used to specify output encoding; in this case formatted (human readable) XML format.

```
Marshaller m = jc.createMarshaller();
m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT,
  Boolean.TRUE);
m.marshal( po, System.out );
```

Sample Output

Running `java Main` for this example produces the following output:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<purchaseOrder orderDate="1999-10-20-05:00">
<shipTo country="US">
<name>Alice Smith</name>
<street>123 Maple Street</street>
<city>Cambridge</city>
<state>MA</state>
<zip>12345</zip>
</shipTo>
<billTo country="US">
<name>John Bob</name>
<street>242 Main Street</street>
<city>Beverly Hills</city>
<state>CA</state>
<zip>90210</zip>
</billTo>
<items>
<item partNum="242-NO">
```

```
<productName>Nosferatu - Special Edition (1929)</productName>
<quantity>5</quantity>
<USPrice>19.99</USPrice>
</item>
<item partNum="242-MU">
<productName>The Mummy (1959)</productName>
<quantity>3</quantity>
<USPrice>19.98</USPrice>
</item>
<item partNum="242-GZ">
<productName>
Godzilla and Mothra: Battle for Earth/Godzilla vs. King Ghidora
</productName>
<quantity>3</quantity>
<USPrice>27.95</USPrice>
</item>
</items>
</purchaseOrder>
```

Create Marshal Example

The Create Marshal example demonstrates how to use the `ObjectFactory` class to create a Java content tree from scratch and then marshal it to XML data.

1. The `<JWSDP_HOME>/jaxb/samples/create-marshal/Main.java` class declares imports for four standard Java classes plus three JAXB binding framework classes and the `primer.po` package:

   ```
   import java.math.BigDecimal;
   import java.math.BigInteger;
   import java.util.Calendar;
   import java.util.List;
   import javax.xml.bind.JAXBContext;
   import javax.xml.bind.JAXBException;
   import javax.xml.bind.Marshaller;
   import primer.po.*;
   ```

2. A `JAXBContext` instance is created for handling classes generated in `primer.po`.

   ```
   JAXBContext jc = JAXBContext.newInstance( "primer.po" );
   ```

3. The `ObjectFactory` class is used to instantiate a new empty `PurchaseOrder` object.

   ```
   // creating the ObjectFactory
   ObjectFactory objFactory = new ObjectFactory();
   ```

```
// create an empty PurchaseOrder
PurchaseOrder po = objFactory.createPurchaseOrder();
```

4. Per the constraints in the `po.xsd` schema, the `PurchaseOrder` object requires a value for the `orderDate` attribute. To satisfy this constraint, the `orderDate` is set using the standard `Calendar.getInstance()` method from `java.util.Calendar`.

```
po.setOrderDate( Calendar.getInstance() );
```

5. The `ObjectFactory` is used to instantiate new empty `USAddress` objects, and the required attributes are set.

```
USAddress shipTo = createUSAddress( "Alice Smith",
               "123 Maple Street",
               "Cambridge",
               "MA",
               "12345" );
  po.setShipTo( shipTo );

USAddress billTo = createUSAddress( "Robert Smith",
               "8 Oak Avenue",
               "Cambridge",
               "MA",
               "12345" );
po.setBillTo( billTo );
```

6. The `ObjectFactory` class is used to instantiate a new empty `Items` object.

```
Items items = objFactory.createItems();
```

7. A `get` method is used to get a reference to the `ItemType` list.

```
List itemList = items.getItem();
```

8. `ItemType` objects are created and added to the `Items` list.

```
itemList.add( createItemType(
             "Nosferatu - Special Edition (1929)",
             new BigInteger( "5" ),
             new BigDecimal( "19.99" ),
             null,
             null,
             "242-NO" ) );
itemList.add( createItemType( "The Mummy (1959)",
             new BigInteger( "3" ),
             new BigDecimal( "19.98" ),
             null,
             null,
             "242-MU" ) );
itemList.add( createItemType(
             "Godzilla and Mothra: Battle for Earth/Godzilla
                vs. King Ghidora",
             new BigInteger( "3" ),
             new BigDecimal( "27.95" ),
             null,
             null,
             "242-GZ" ) );
```

9. The `items` object now contains a list of `ItemType` objects and can be added to the po object.

```
po.setItems( items );
```

10. A `Marshaller` instance is created, and the updated XML content is marshalled to `system.out`. The `setProperty` API is used to specify output encoding; in this case formatted (human readable) XML format.

```
Marshaller m = jc.createMarshaller();
m.setProperty( Marshaller.JAXB_FORMATTED_OUTPUT,
   Boolean.TRUE );
m.marshal( po, System.out );
```

11. An empty `USAddress` object is created and its properties set to comply with the schema constraints.

```
public static USAddress createUSAddress(
             ObjectFactory objFactory,
             String name, String street,
             String city,
             String state,
             String zip )
     throws JAXBException {

   // create an empty USAddress objects
   USAddress address = objFactory.createUSAddress();

   // set properties on it
   address.setName( name );
   address.setStreet( street );
   address.setCity( city );
   address.setState( state );
   address.setZip( new BigDecimal( zip ) );

   // return it
```

```
        return address;
    }
```

12. Similar to the previous step, an empty `ItemType` object is created and its properties set to comply with the schema constraints.

```
    public static Items.ItemType createItemType(ObjectFactory
         objFactory,
                 String productName,
                 BigInteger quantity,
                 BigDecimal price,
                 String comment,
                 Calendar shipDate,
                 String partNum )
         throws JAXBException {

        // create an empty ItemType object
        Items.ItemType itemType =
        objFactory.createItemsItemType();

        // set properties on it
        itemType.setProductName( productName );
        itemType.setQuantity( quantity );
        itemType.setUSPrice( price );
        itemType.setComment( comment );
        itemType.setShipDate( shipDate );
        itemType.setPartNum( partNum );

        // return it
        return itemType;
    }
```

Sample Output

Running `java Main` for this example produces the following output:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<purchaseOrder orderDate="2002-09-24-05:00">
<shipTo>
<name>Alice Smith</name>
<street>123 Maple Street</street>
<city>Cambridge</city>
<state>MA</state>
<zip>12345</zip>
</shipTo>
<billTo>
<name>Robert Smith</name>
<street>8 Oak Avenue</street>
```

```
<city>Cambridge</city>
<state>MA</state>
<zip>12345</zip>
</billTo>
<items>
<item partNum="242-NO">
<productName>Nosferatu - Special Edition (1929)</productName>
<quantity>5</quantity
<USPrice>19.99</USPrice>
</item>
<item partNum="242-MU">
<productName>The Mummy (1959)</productName>
<quantity>3</quantity>
<USPrice>19.98</USPrice>
</item>
<item partNum="242-GZ">
<productName>Godzilla and Mothra: Battle for Earth/Godzilla vs.
King Ghidora</productName>
<quantity>3</quantity>
<USPrice>27.95</USPrice>
</item>
</items>
</purchaseOrder>
```

Unmarshal Validate Example

The Unmarshal Validate example demonstrates how to enable validation during unmarshalling (*Unmarshal-Time Validation*). Note that JAXB provides functions for validation during unmarshalling but not during marshalling. Validation is explained in more detail in More About Validation (page 11).

1. The `<`*`JWSDP_HOME`*`>/jaxb/samples/unmarshal-validate/Main.java` class declares imports for three standard Java classes plus seven JAXB binding framework classes and the `primer.po` package:

   ```
   import java.io.FileInputStream;
   import java.io.IOException;
   import java.math.BigDecimal;
   import javax.xml.bind.JAXBContext;
   import javax.xml.bind.JAXBException;
   import javax.xml.bind.Marshaller;
   import javax.xml.bind.UnmarshalException;
   import javax.xml.bind.Unmarshaller;
   import javax.xml.bind.ValidationEvent;
   import javax.xml.bind.util.ValidationEventCollector;
   import primer.po.*;
   ```

2. A `JAXBContext` instance is created for handling classes generated in `primer.po`.

   ```
   JAXBContext jc = JAXBContext.newInstance( "primer.po" );
   ```
3. An `Unmarshaller` instance is created.

   ```
   Unmarshaller u = jc.createUnmarshaller();
   ```
4. The default JAXB Unmarshaller `ValidationEventHandler` is enabled to send to validation warnings and errors to `system.out`. The default configuration causes the unmarshal operation to fail upon encountering the first validation error.

   ```
   u.setValidating( true );
   ```
5. An attempt is made to unmarshal `po.xml` into a Java content tree. For the purposes of this example, the `po.xml` contains a deliberate error.

   ```
   PurchaseOrder po =
     (PurchaseOrder)u.unmarshal(
       new FileInputStream("po.xml"));
   ```
6. The default validation event handler processes a validation error, generates output to `system.out`, and then an exception is thrown.

   ```
   } catch( UnmarshalException ue ) {
     System.out.println( "Caught UnmarshalException" );
   } catch( JAXBException je ) {
     je.printStackTrace();
   } catch( IOException ioe ) {
     ioe.printStackTrace();
   ```

Sample Output

Running `java Main` for this example produces the following output:

```
DefaultValidationEventHandler: [ERROR]: "-1" does not satisfy
the "positiveInteger" type
Caught UnmarshalException
```

Validate-On-Demand Example

The Validate-On-Demand example demonstrates how to validate a Java content tree at runtime (*On-Demand Validation*). At any point, client applications can call the `Validator.validate` method on the Java content tree (or any subtree of it). All JAXB Providers are required to support this operation. Validation is explained in more detail in More About Validation (page 11).

1. The `<JWSDP_HOME>/jaxb/samples/ondemand-validate/Main.java` class declares imports for five standard Java classes plus nine JAXB Java classes and the `primer.po` package:

```
import java.io.FileInputStream;
import java.io.IOException;
import java.math.BigDecimal;
import java.math.BigInteger;
import java.util.List;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;
import javax.xml.bind.UnmarshalException;
import javax.xml.bind.Unmarshaller;
import javax.xml.bind.ValidationEvent;
import javax.xml.bind.ValidationException;
import javax.xml.bind.Validator;
import javax.xml.bind.util.ValidationEventCollector;
import primer.po.*;
```

2. A `JAXBContext` instance is created for handling classes generated in `primer.po`.

```
JAXBContext jc = JAXBContext.newInstance( "primer.po" );
```

3. An `Unmarshaller` instance is created, and a valid `po.xml` document is unmarshalled into a Java content tree. Note that `po.xml` is valid at this point; invalid data will be added later in this example.

```
Unmarshaller u = jc.createUnmarshaller();
PurchaseOrder po =
   (PurchaseOrder)u.unmarshal( new FileInputStream( "po.xml"
) );
```

4. A reference is obtained for the first item in the purchase order.

```
Items items = po.getItems();
List itemTypeList = items.getItem();
Items.ItemType item = (Items.ItemType)itemTypeList.get( 0 );
```

5. Next, the item quantity is set to an invalid number. When validation is enabled later in this example, this invalid quantity will throw an exception.

```
item.setQuantity( new BigInteger( "-5" ) );
```

Note: If `@enableFailFastCheck` was `"true"` and the optional `FailFast` validation method was supported by an implementation, a `TypeConstraintException` would be thrown here. Note that the JAXB implementation does not support the `FailFast` feature. Refer to the *JAXB Specification* for more information about `FailFast` validation.

---

6. A `Validator` instance is created, and the content tree is validated. Note that the `Validator` class is responsible for managing On-Demand validation, whereas the `Unmarshaller` class is responsible for managing Unmarshal-Time validation during unmarshal operations.

```
Validator v = jc.createValidator();
boolean valid = v.validateRoot( po );
System.out.println( valid );
```

7. The default validation event handler processes a validation error, generates output to `system.out`, and then an exception is thrown.

```
} catch( ValidationException ue ) {
   System.out.println( "Caught ValidationException" );
} catch( JAXBException je ) {
   je.printStackTrace();
} catch( IOException ioe ) {
   ioe.printStackTrace();
}
```

Sample Output

Running `java Main` for this example produces the following output:

```
DefaultValidationEventHandler: [ERROR]: "-5" does not satisfy
the "positiveInteger" type
Caught ValidationException
```

Customizing JAXB Bindings

The remainder of this chapter describes several examples that build on the concepts demonstrated in the basic examples.

The goal of this section is to illustrate how to customize JAXB bindings by means of custom binding declarations made in either of two ways:

- As annotations made inline in an XML schema
- As statements in an external file passed to the JAXB binding compiler

Unlike the examples in Basic Examples (page 43), which focus on the Java code in the respective `Main.java` class files, the examples here focus on customizations made to the XML schema *before* generating the schema-derived Java binding classes.

Note: Although JAXB binding customizations must currently be made by hand, it is envisioned that a tool/wizard may eventually be written by Sun or a third party to make this process more automatic and easier in general. One of the goals of the JAXB technology is to standardize the format of binding declarations, thereby making it possible to create customization tools and to provide a standard interchange format between JAXB implementations.

This section just begins to scratch the surface of customizations you can make to JAXB bindings and validation methods. For more information, please refer to the *JAXB Specification* (`http://java.sun.com/xml/downloads/jaxb.html`).

Why Customize?

In most cases, the default bindings generated by the JAXB binding compiler will be sufficient to meet your needs. There are cases, however, in which you may want to modify the default bindings. Some of these include:

- Creating API documentation for the schema-derived JAXB packages, classes, methods and constants; by adding custom Javadoc tool annotations to your schemas, you can explain concepts, guidelines, and rules specific to your implementation.
- Providing semantically meaningful customized names for cases that the default XML name-to-Java identifier mapping cannot handle automatically; for example:
  - To resolve name collisions (as described in Appendix C.2.1 of the *JAXB Specification*). Note that the JAXB binding compiler detects and reports all name conflicts.
  - To provide names for typesafe enumeration constants that are not legal Java identifiers; for example, enumeration over integer values.
  - To provide better names for the Java representation of unnamed model groups when they are bound to a Java property or class.
  - To provide more meaningful package names than can be derived by default from the target namespace URI.
- Overriding default bindings; for example:
  - Specify that a model group should be bound to a class rather than a list.

- Specify that a fixed attribute can be bound to a Java constant.
- Override the specified default binding of XML Schema built-in datatypes to Java datatypes. In some cases, you might want to introduce an alternative Java class that can represent additional characteristics of the built-in XML Schema datatype.

Customization Overview

This section explains some core JAXB customization concepts:

- Inline and External Customizations
- Scope, Inheritance, and Precedence
- Customization Syntax
- Customization Namespace Prefix

Inline and External Customizations

Customizations to the default JAXB bindings are made in the form of *binding declarations* passed to the JAXB binding compiler. These binding declarations can be made in either of two ways:

- As inline annotations in a source XML schema
- As declarations in an external binding customizations file

For some people, using inline customizations is easier because you can see your customizations in the context of the schema to which they apply. Conversely, using an external binding customization file enables you to customize JAXB bindings without having to modify the source schema, and enables you to easily apply customizations to several schema files at once.

Note: You can combine the two types of customizations—for example, you could include a reference to an external binding customizations file in an inline annotation—but you cannot declare both an inline and external customization on the same schema element.

Each of these types of customization is described in more detail below.

Inline Customizations

Customizations to JAXB bindings made by means of inline *binding declarations* in an XML schema file take the form of `<xsd:appinfo>` elements embedded in schema `<xsd:annotation>` elements (`xsd:` is the XML schema namespace prefix, as defined in W3C *XML Schema Part 1: Structures*). The general form for inline customizations is shown below.

```
<xs:annotation>
   <xs:appinfo>
      .
      .
      binding declarations
      .
      .
   </xs:appinfo>
</xs:annotation>
```

Customizations are applied at the location at which they are declared in the schema. For example, a declaration at the level of a particular element would apply to that element only. Note that the XMLSchema namespace prefix must be used with the `<annotation>` and `<appinfo>` declaration tags. In the example above, `xs:` is used as the namespace prefix, so the declarations are tagged `<xs:annotation>` and `<xs:appinfo>`.

External Binding Customization Files

Customizations to JAXB bindings made by means of an external file containing binding declarations take the general form shown below.

```
<jxb:bindings schemaLocation = "xs:anyURI">
   <jxb:bindings node = "xs:string">*
      <binding declaration>
   <jxb:bindings>
</jxb:bindings>
```

- `schemaLocation` is a URI reference to the remote schema
- `node` is an XPath 1.0 expression that identifies the schema node within `schemaLocation` to which the given binding declaration is associated.

For example, the first `schemaLocation`/`node` declaration in a JAXB binding declarations file specifies the schema name and the root schema node:

```
<jxb:bindings schemaLocation="po.xsd" node="/xs:schema">
```

A subsequent `schemaLocation/node` declaration, say for a `simpleType` element named `ZipCodeType` in the above schema, would take the form:

```
<jxb:bindings node="//xs:simpleType[@name='ZipCodeType']">
```

Binding Customization File Format

Binding customization files should be straight ASCII text. The name or extension does not matter, although a typical extension, used in this chapter, is `.xjb`.

Passing Customization Files to the JAXB Binding Compiler

Customization files containing binding declarations are passed to the JAXB Binding compiler, `xjc`, using the following syntax:

```
xjc -b <file> <schema>
```

where *<file>* is the name of binding customization file, and *<schema>* is the name of the schema(s) you want to pass to the binding compiler.

You can have a single binding file that contains customizations for multiple schemas, or you can break the customizations into multiple bindings files; for example:

```
xjc schema1.xsd schema2.xsd schema3.xsd -b bindings123.xjb

xjc schema1.xsd schema2.xsd schema3.xsd -b bindings1.xjb -b
bindings2.xjb -b bindings3.xjb
```

Note that the ordering of schema files and binding files on the command line does not matter, although each binding customization file must be preceded by its own `-b` switch on the command line.

For more information about `xjc` compiler options in general, see JAXB Compiler Options (page 30).

Restrictions for External Binding Customizations

There are several rules that apply to binding declarations made in an external binding customization file that do not apply to similar declarations made inline in a source schema:

- The binding customization file must begin with the `jxb:bindings version` attribute, plus attributes for the JAXB and XMLSchema namespaces:

  ```
  <jxb:bindings version="1.0"
     xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
     xmlns:xs="http://www.w3.org/2001/XMLSchema">
  ```

- The remote schema to which the binding declaration applies must be identified explicitly in XPath notation by means of a `jxb:bindings` declaration specifying `schemaLocation` and `node` attributes:
  - `schemaLocation` – URI reference to the remote schema
  - `node` – XPath 1.0 expression that identifies the schema node within `schemaLocation` to which the given binding declaration is associated; in the case of the initial `jxb:bindings` declaration in the binding customization file, this node is typically `"/xs:schema"`

  For information about XPath syntax, see *XML Path Language*, James Clark and Steve DeRose, eds., W3C, 16 November 1999. Available at `http://www.w3.org/TR/1999/REC-xpath-19991116`.

- Similarly, individual nodes within the schema to which customizations are to be applied must be specified using XPath notation; for example:

  ```
  <jxb:bindings node="//xs:complexType[@name='USAddress']">
  ```

  In such cases, the customization is applied to the node by the binding compiler as if the declaration was embedded inline in the node's `<xs:appinfo>` element.

To summarize these rules, the external binding element `<jxb:bindings>` is only recognized for processing by a JAXB binding compiler in three cases:

- When its parent is an `<xs:appinfo>` element
- When it is an ancestor of another `<jxb:bindings>` element
- When it is root element of a document—an XML document that has a `<jxb:bindings>` element as its root is referred to as an external binding declaration file

Scope, Inheritance, and Precedence

Default JAXB bindings can be customized or overridden at four different levels, or *scopes*, as described in Table 2–4.

Figure 2–1 illustrates the inheritance and precedence of customization declarations. Specifically, declarations towards the top of the pyramid inherit and supersede declarations below them. For example, Component declarations inherit from and supersede Definition declarations; Definition declarations inherit and supersede Schema declarations; and Schema declarations inherit and supersede Global declarations.

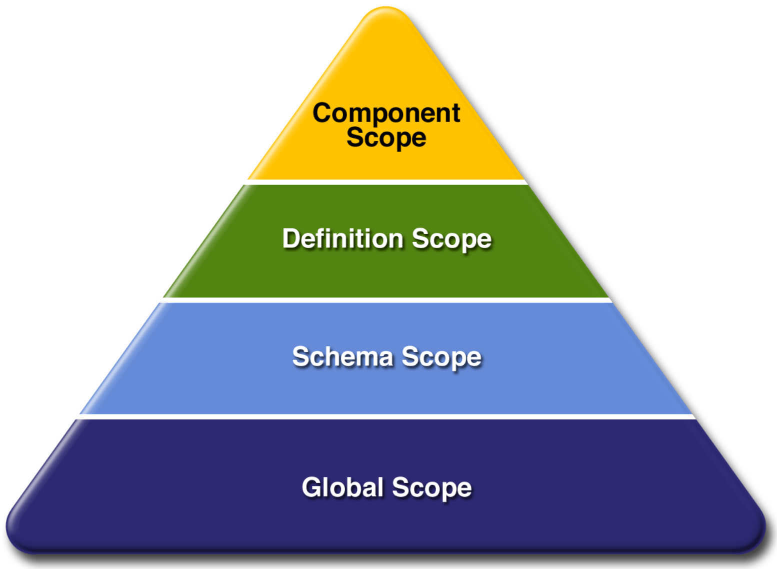

Figure 2–1 Customization Scope Inheritance and Precedence

Customization Syntax

The syntax for the four types of JAXB binding declarations, as well as the syntax for the XML-to-Java datatype binding declarations and the customization namespace prefix are described below.

- Global Binding Declarations
- Schema Binding Declarations
- Class Binding Declarations
- Property Binding Declarations
- <javaType> Binding Declarations
- Typesafe Enumeration Binding Declarations
- <javadoc> Binding Declarations
- Customization Namespace Prefix

Global Binding Declarations

Global scope customizations are declared with `<globalBindings>`. The syntax for global scope customizations is as follows:

```
<globalBindings>
   [ collectionType = "collectionType" ]
   [ fixedAttributeAsConstantProperty= "true" | "false" | "1" | "0" ]
   [ generateIsSetMethod= "true" | "false" | "1" | "0" ]
   [ enableFailFastCheck = "true" | "false" | "1" | "0" ]
   [ choiceContentProperty = "true" | "false" | "1" | "0" ]
   [ underscoreBinding = "asWordSeparator" | "asCharInWord" ]
   [ typesafeEnumBase = "typesafeEnumBase" ]
   [ typesafeEnumMemberName = "generateName" | "generateError" ]
   [ enableJavaNamingConventions = "true" | "false" | "1" | "0" ]
   [ bindingStyle = "elementBinding" | "modelGroupBinding" ]
   [ <javaType> ... </javaType> ]*
</globalBindings>
```

- `collectionType` can be either `indexed` or any fully qualified class name that implements `java.util.List`.
- `fixedAttributeAsConstantProperty` can be either `true`, `false`, `1`, or `0`. The default value is `false`.
- `generateIsSetMethod` can be either `true`, `false`, `1`, or `0`. The default value is `false`.
- `enableFailFastCheck` can be either `true`, `false`, `1`, or `0`. If `enableFail-FastCheck` is `true` or `1` and the JAXB implementation supports this optional checking, type constraint checking is performed when setting a

property. The default value is `false`. Please note that the JAXB implementation does not support failfast validation.

- `choiceContentProperty` can be either `true`, `false`, `1`, or `0`. The default value is `false`. `choiceContentProperty` is not relevant when the `bindingStyle` is `elementBinding`. Therefore, if `bindingStyle` is specified as `elementBinding`, then the `choiceContentProperty` must result in an invalid customization.
- `underscoreBinding` can be either `asWordSeparator` or `asCharInWord`. The default value is `asWordSeparator`.
- `enableJavaNamingConventions` can be either `true`, `false`, `1`, or `0`. The default value is `true`.
- `typesafeEnumBase` can be a list of QNames, each of which must resolve to a simple type definition. The default value is `xs:NCName`. See Typesafe Enumeration Binding Declarations (page 66) for information about localized mapping of `simpleType` definitions to Java `typesafe enum` classes.
- `typesafeEnumMemberName` can be either `generateError` or `generateName`. The default value is `generateError`.
- `bindingStyle` can be either `elementBinding`, or `modelGroupBinding`. The default value is `elementBinding`.
- `<javaType>` can be zero or more javaType binding declarations. See <javaType> Binding Declarations (page 64) for more information.

`<globalBindings>` declarations are only valid in the `annotation` element of the top-level `schema` element. There can only be a single instance of a `<globalBindings>` declaration in any given schema or binding declarations file. If one source schema includes or imports a second source schema, the `<globalBindings>` declaration must be declared in the first source schema.

Schema Binding Declarations

Schema scope customizations are declared with `<schemaBindings>`. The syntax for schema scope customizations is:

```
<schemaBindings>
  [ <package> package </package> ]
  [ <nameXmlTransform> ... </nameXmlTransform> ]*
</schemaBindings>

<package [ name = "packageName" ]
  [ <javadoc> ... </javadoc> ]
</package>
```

```
<nameXmlTransform>
  [ <typeName [ suffix="suffix" ]
              [ prefix="prefix" ] /> ]
  [ <elementName [ suffix="suffix" ]
                 [ prefix="prefix" ] /> ]
  [ <modelGroupName [ suffix="suffix" ]
                    [ prefix="prefix" ] /> ]
  [ <anonymousTypeName [ suffix="suffix" ]
                       [ prefix="prefix" ] /> ]
</nameXmlTransform>
```

As shown above, `<schemaBinding>` declarations include two subcomponents:

- `<package>...</package>` specifies the name of the package and, if desired, the location of the API documentation for the schema-derived classes.
- `<nameXmlTransform>...</nameXmlTransform>` specifies customizations to be applied.

Class Binding Declarations

The `<class>` binding declaration enables you to customize the binding of a schema element to a Java content interface or a Java `Element` interface. `<class>` declarations can be used to customize:

- A name for a schema-derived Java interface
- An implementation class for a schema-derived Java content interface.

The syntax for `<class>` customizations is:

```
<class [ name = "className"]
   [ implClass= "implClass" ] >
   [ <javadoc> ... </javadoc> ]
</class>
```

- `name` is the name of the derived Java interface. It must be a legal Java interface name and must not contain a package prefix. The package prefix is inherited from the current value of package.
- `implClass` is the name of the implementation class for `className` and must include the complete package name.
- The `<javadoc>` element specifies the Javadoc tool annotations for the schema-derived Java interface. The string entered here must use `CDATA` or `<` to escape embedded HTML tags.

Property Binding Declarations

The <property> binding declaration enables you to customize the binding of an XML schema element to its Java representation as a property. The scope of customization can either be at the definition level or component level depending upon where the <property> binding declaration is specified.

The syntax for <property> customizations is:

```
<property[ name = "propertyName"]
   [ collectionType = "propertyCollectionType" ]
   [ fixedAttributeAsConstantProperty = "true" | "false" | "1" | "0" ]
   [ generateIsSetMethod = "true" | "false" | "1" | "0" ]
   [ enableFailFastCheck ="true" | "false" | "1" | "0" ]
   [ <baseType> ... </baseType> ]
   [ <javadoc> ... </javadoc> ]
</property>

<baseType>
   <javaType> ... </javaType>
</baseType>
```

- name defines the customization value propertyName; it must be a legal Java identifier.
- collectionType defines the customization value propertyCollectionType, which is the collection type for the property. propertyCollectionType if specified, can be either indexed or any fully-qualified class name that implements java.util.List.
- fixedAttributeAsConstantProperty defines the customization value fixedAttributeAsConstantProperty. The value can be either true, false, 1, or 0.
- generateIsSetMethod defines the customization value of generateIsSetMethod. The value can be either true, false, 1, or 0.
- enableFailFastCheck defines the customization value enableFailFastCheck. The value can be either true, false, 1, or 0. Please note that the JAXB implementation does not support failfast validation.
- <javadoc> customizes the Javadoc tool annotations for the property's getter method.

<javaType> Binding Declarations

The <javaType> declaration provides a way to customize the translation of XML datatypes to and from Java datatypes. XML provides more datatypes than

Java, and so the `<javaType>` declaration lets you specify custom datatype bindings when the default JAXB binding cannot sufficiently represent your schema.

The target Java datatype can be a Java built-in datatype or an application-specific Java datatype. If an application-specific datatype is used as the target, your implementation must also provide parse and print methods for unmarshalling and marshalling data. To this end, the JAXB specification supports a `parseMethod` and `printMethod`:

- The `parseMethod` is called during unmarshalling to convert a string from the input document into a value of the target Java datatype.
- The `printMethod` is called during marshalling to convert a value of the target type into a lexical representation.

If you prefer to define your own datatype conversions, JAXB defines a static class, `DatatypeConverter`, to assist in the parsing and printing of valid lexical representations of the XML Schema built-in datatypes.

The syntax for the `<javaType>` customization is:

```
<javaType name= "javaType"
      [ xmlType= "xmlType" ]
      [ hasNsContext = “true” | “false” ]
      [ parseMethod= "parseMethod" ]
      [ printMethod= "printMethod" ]>
```

- `name` is the Java datatype to which `xmlType` is to be bound.
- `xmlType` is the name of the XML Schema datatype to which `javaType` is to bound; this attribute is required when the parent of the `<javaType>` declaration is `<globalBindings>`.
- `parseMethod` is the name of the parse method to be called during unmarshalling.
- `printMethod` is the name of the print method to be called during marshalling.
- `hasNsContext` allows a namespace context to be specified as a second parameter to a print or a parse method; can be either `true`, `false`, `1`, or `0`. By default, this attribute is `false`, and in most cases you will not need to change it.

The `<javaType>` declaration can be used in:

- A `<globalBindings>` declaration
- An annotation element for simple type definitions, `GlobalBindings`, and `<basetype>` declarations.
- A `<property>` declaration.

See MyDatatypeConverter Class (page 73) for an example of how `<javaType>` declarations and the `DatatypeConverterInterface` interface are implemented in a custom datatype converter class.

Typesafe Enumeration Binding Declarations

The typesafe enumeration declarations provide a localized way to map XML `simpleType` elements to Java `typesafe enum` classes. There are two types of typesafe enumeration declarations you can make:

- `<typesafeEnumClass>` lets you map an entire `simpleType` class to `typesafe enum` classes.
- `<typesafeEnumMember>` lets you map just selected members of a `simpleType` class to `typesafe enum` classes.

In both cases, there are two primary limitations on this type of customization:

- Only `simpleType` definitions with enumeration facets can be customized using this binding declaration.
- This customization only applies to a single `simpleType` definition at a time. To map sets of similar `simpleType` definitions on a global level, use the `typesafeEnumBase` attribute in a `<globalBindings>` declaration, as described Global Binding Declarations (page 61).

The syntax for the `<typesafeEnumClass>` customization is:

```
<typesafeEnumClass[ name = "enumClassName" ]
  [ <typesafeEnumMember> ... </typesafeEnumMember> ]*
  [ <javadoc> enumClassJavadoc </javadoc> ]
</typesafeEnumClass>
```

- `name` must be a legal Java Identifier, and must not have a package prefix.
- `<javadoc>` customizes the Javadoc tool annotations for the enumeration class.
- You can have zero or more `<typesafeEnumMember>` declarations embedded in a `<typesafeEnumClass>` declaration.

The syntax for the `<typesafeEnumMember>` customization is:

```
<typesafeEnumMember name = "enumMemberName">
            [ value = "enumMemberValue" ]
  [ <javadoc> enumMemberJavadoc </javadoc> ]
</typesafeEnumMember>
```

- `name` must always be specified and must be a legal Java identifier.
- `value` must be the enumeration value specified in the source schema.
- `<javadoc>` customizes the Javadoc tool annotations for the enumeration constant.

For inline annotations, the `<typesafeEnumClass>` declaration must be specified in the annotation element of the `<simpleType>` element. The `<typesafeEnumMember>` must be specified in the annotation element of the enumeration member. This allows the enumeration member to be customized independently from the enumeration class.

For information about typesafe enum design patterns, see the sample chapter of Joshua Bloch's *Effective Java Programming* on the Java Developer Connection.

<javadoc> Binding Declarations

The `<javadoc>` declaration lets you add custom Javadoc tool annotations to schema-derived JAXB packages, classes, interfaces, methods, and fields. Note that `<javadoc>` declarations cannot be applied globally—that is, they are only valid as a sub-elements of other binding customizations.

The syntax for the `<javadoc>` customization is:

```
<javadoc>
  Contents in &lt;b>Javadoc&lt;\b> format.
</javadoc>
```

or

```
<javadoc>
  <<![CDATA[
  Contents in <b>Javadoc<\b> format
  ]]>
</javadoc>
```

Note that documentation strings in `<javadoc>` declarations applied at the package level must contain `<body>` open and close tags; for example:

```
<jxb:package name="primer.myPo">
       <jxb:javadoc><![CDATA[<body>Package level documentation
for generated package primer.myPo.</body>]]>
</jxb:javadoc>
      </jxb:package>
```

Customization Namespace Prefix

All standard JAXB binding declarations must be preceded by a namespace prefix that maps to the JAXB namespace URI (`http://java.sun.com/xml/ns/jaxb`). For example, in this sample, `jxb:` is used. To this end, any schema you want to customize with standard JAXB binding declarations *must* include the JAXB namespace declaration and JAXB version number at the top of the schema file. For example, in `po.xsd` for the Customize Inline example, the namespace declaration is as follows:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
            jxb:version="1.0">
```

A binding declaration with the `jxb` namespace prefix would then take the form:

```
<xsd:annotation>
   <xsd:appinfo>
      <jxb:globalBindings binding declarations />
      <jxb:schemaBindings>
         .
         .
         binding declarations
         .
         .
      </jxb:schemaBindings>
   </xsd:appinfo>
</xsd:annotation>
```

Note that in this example, the `globalBindings` and `schemaBindings` declarations are used to specify, respectively, global scope and schema scope customizations. These customization scopes are described in more detail in Scope, Inheritance, and Precedence (page 60).

Customize Inline Example

The Customize Inline example illustrates some basic customizations made by means of inline annotations to an XML schema named `po.xsd`. In addition, this example implements a custom datatype converter class, `MyDatatypeConverter.java`, which illustrates print and parse methods in the `<javaType>` customization for handling custom datatype conversions.

To summarize this example:

1. `po.xsd` is an XML schema containing inline binding customizations.
2. `MyDatatypeConverter.java` is a Java class file that implements print and parse methods specified by `<javaType>` customizations in `po.xsd`.
3. `Main.java` is the primary class file in the Customize Inline example, which uses the schema-derived classes generated by the JAXB compiler.

Key customizations in this sample, and the custom `MyDatatypeConverter.java` class, are described in more detail below.

Customized Schema

The customized schema used in the Customize Inline example is in the file `<`*`JAVA_HOME`*`>/jaxb/samples/inline-customize/po.xsd`. The customizations are in the `<xsd:annotation>` tags.

Global Binding Declarations

The code below shows the `globalBindings` declarations in `po.xsd`:

```
<jxb:globalBindings
        fixedAttributeAsConstantProperty="true"
        collectionType="java.util.Vector"
        typesafeEnumBase="xsd:NCName"
        choiceContentProperty="false"
        typesafeEnumMemberName="generateError"
        bindingStyle="elementBinding"
        enableFailFastCheck="false"
        generateIsSetMethod="false"
        underscoreBinding="asCharInWord"/>
```

In this example, all values are set to the defaults except for `collectionType`.

- Setting `collectionType` to `java.util.Vector` specifies that all lists in the generated implementation classes should be represented internally as vectors. Note that the class name you specify for `collectionType` must implement `java.util.List` and be callable by `newInstance`.
- Setting `fixedAttributeAsConstantProperty` to true indicates that all fixed attributes should be bound to Java constants. By default, fixed attributes are just mapped to either simple or collection property, which ever is more appropriate.
- Please note that the JAXB implementation does not support the `enable-FailFastCheck` attribute.
- If `typesafeEnumBase` to `xsd:string` it would be a global way to specify that all `simple` type definitions deriving directly or indirectly from `xsd:string` and having enumeration facets should be bound by default to a `typesafe enum`. If `typesafeEnumBase` is set to an empty string, `""`, no `simple` type definitions would ever be bound to a `typesafe enum` class by default. The value of `typesafeEnumBase` can be any atomic simple type definition except `xsd:boolean` and both binary types.

Note: Using typesafe enums enables you to map schema enumeration values to Java constants, which in turn makes it possible to do compares on Java constants rather than string values.

Schema Binding Declarations

The following code shows the schema binding declarations in `po.xsd`:

```
<jxb:schemaBindings>
      <jxb:package name="primer.myPo">
         <jxb:javadoc>
   <![CDATA[<body> Package level documentation for
generated package primer.myPo.</body>]]>
         </jxb:javadoc>
      </jxb:package>
      <jxb:nameXmlTransform>
          <jxb:elementName suffix="Element"/>
      </jxb:nameXmlTransform>
    </jxb:schemaBindings>
```

- `<jxb:package name="primer.myPo"/>` specifies the `primer.myPo` as the package in which the schema-derived classes should be generated.

- `<jxb:nameXmlTransform>` specifies that all generated Java element interfaces should have `Element` appended to the generated names by default. For example, when the JAXB compiler is run against this schema, the element interfaces `CommentElement` and `PurchaseOrderElement` will be generated. By contrast, without this customization, the default binding would instead generate `Comment` and `PurchaseOrder`.

  This customization is useful if a schema uses the same name in different symbol spaces; for example, in global element and type definitions. In such cases, this customization enables you to resolve the collision with one declaration rather than having to individually resolve each collision with a separate binding declaration.
- `<jxb:javadoc>` specifies customized Javadoc tool annotations for the `primer.myPo` package. Note that, unlike the `<javadoc>` declarations at the class level, below, the opening and closing `<body>` tags must be included when the `<javadoc>` declaration is made at the package level.

Class Binding Declarations

The following code shows the class binding declarations in `po.xsd`:

```
<xsd:complexType name="PurchaseOrderType">
      <xsd:annotation>
      <xsd:appinfo>
         <jxb:class name="POType">
            <jxb:javadoc>
            A &lt;b>Purchase Order&lt;/b> consists of
addresses and items.
            </jxb:javadoc>
         </jxb:class>
      </xsd:appinfo>
      </xsd:annotation>
      .
      .
      .
</xsd:complexType>
```

The Javadoc tool annotations for the schema-derived `POType` class will contain the description `"A &lt;b>Purchase Order&lt;/b> consists of addresses and items."` The `&lt;` is used to escape the opening bracket on the `<b>` HTML tags.

Note: When a `<class>` customization is specified in the `appinfo` element of a `complexType` definition, as it is here, the `complexType` definition is bound to a Java content interface.

Later in `po.xsd`, another `<javadoc>` customization is declared at this class level, but this time the HTML string is escaped with `CDATA`:

```
<xsd:annotation>
 <xsd:appinfo>
    <jxb:class>
      <jxb:javadoc>
     <![CDATA[ First line of documentation for a
<b>USAddress</b>.]]>
       </jxb:javadoc>
    </jxb:class>
  </xsd:appinfo>
  </xsd:annotation>
```

Note: If you want to include HTML markup tags in a `<jaxb:javadoc>` customization, you must enclose the data within a `CDATA` section or escape all left angle brackets using `&lt;`. See *XML 1.0 2nd Edition* for more information (`http://www.w3.org/TR/2000/REC-xml-20001006#sec-cdata-sect`).

Property Binding Declarations

Of particular interest here is the `generateIsSetMethod` customization, which causes two additional property methods, `isSetQuantity` and `unsetQuantity`, to be generated. These methods enable a client application to distinguish between schema default values and values occurring explicitly within an instance document.

For example, in `po.xsd`:

```
<xsd:complexType name="Items">
   <xsd:sequence>
      <xsd:element name="item" minOccurs="1"
maxOccurs="unbounded">
         <xsd:complexType>
            <xsd:sequence>
           <xsd:element name="productName" type="xsd:string"/>
            <xsd:element name="quantity" default="10">
            <xsd:annotation>
```

```
                <xsd:appinfo>
                   <jxb:property generateIsSetMethod="true"/>
                </xsd:appinfo>
             </xsd:annotation>
          .
          .
          .
          </xsd:complexType>
       </xsd:element>
    </xsd:sequence>
 </xsd:complexType>
```

The `@generateIsSetMethod` applies to the `quantity` element, which is bound to a property within the `Items.ItemType` interface. `unsetQuantity` and `isSetQuantity` methods are generated in the `Items.ItemType` interface.

MyDatatypeConverter Class

The `<`*`JWSDP_HOME`*`>/jaxb/samples/inline-customize/MyDatatypeConverter` class, shown below, provides a way to customize the translation of XML datatypes to and from Java datatypes by means of a `<javaType>` customization.

```
package primer;
import java.math.BigInteger;
import javax.xml.bind.DatatypeConverter;

public class MyDatatypeConverter {

  public static short parseIntegerToShort(String value) {
     BigInteger result =
        DatatypeConverter.parseInteger(value);
     return (short)(result.intValue());
  }

  public static String printShortToInteger(short value) {
     BigInteger result = BigInteger.valueOf(value);
     return DatatypeConverter.printInteger(result);
  }

  public static int parseIntegerToInt(String value) {
     BigInteger result =
     DatatypeConverter.parseInteger(value);
  return result.intValue();
  }
```

```
    public static String printIntToInteger(int value) {
      BigInteger result = BigInteger.valueOf(value);
      return DatatypeConverter.printInteger(result);
    }
  };
```

The following code shows how the `MyDatatypeConverter` class is referenced in a `<javaType>` declaration in `po.xsd`:

```
<xsd:simpleType name="ZipCodeType">
  <xsd:annotation>
     <xsd:appinfo>
        <jxb:javaType name="int"
parseMethod="primer.MyDatatypeConverter.parseIntegerToInt"
printMethod="primer.MyDatatypeConverter.printIntTo Integer" />
     </xsd:appinfo>
  </xsd:annotation>
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="10000"/>
    <xsd:maxInclusive value="99999"/>
  </xsd:restriction>
</xsd:simpleType>
```

In this example, the `jxb:javaType` binding declaration overrides the default JAXB binding of this type to `java.math.BigInteger`. For the purposes of the Customize Inline example, the restrictions on `ZipCodeType`—specifically that legal US ZIP codes are limited to five digits—make it so all valid values can easily fit within the Java primitive datatype `int`. Note also that, because `<jxb:javaType name="int"/>` is declared within `ZipCodeType`, the customization applies to all JAXB properties that reference this `simpleType` definition, including the `getZip` and `setZip` methods.

Datatype Converter Example

The Datatype Converter example is very similar to the Customize Inline example. As with the Customize Inline example, the customizations in the Datatype Converter example are made by using inline binding declarations in the XML schema for the application, `po.xsd`.

The global, schema, and package, and most of the class customizations for the Customize Inline and Datatype Converter examples are identical. Where the Datatype Converter example differs from the Customize Inline example is in the parseMethod and printMethod used for converting XML data to the Java int datatype.

Specifically, rather than using methods in the custom MyDataTypeConverter class to perform these datatype conversions, the Datatype Converter example uses the built-in methods provided by javax.xml.bind.DatatypeConverter:

```
<xsd:simpleType name="ZipCodeType">
  <xsd:annotation>
     <xsd:appinfo>
        <jxb:javaType name="int"
   parseMethod="javax.xml.bind.DatatypeConverter.parseInt"
   printMethod="javax.xml.bind.DatatypeConverter.printInt"/>
     </xsd:appinfo>
  </xsd:annotation>
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="10000"/>
    <xsd:maxInclusive value="99999"/>
  </xsd:restriction>
</xsd:simpleType>
```

External Customize Example

The External Customize example is identical to the Datatype Converter example, except that the binding declarations in the External Customize example are made by means of an external binding declarations file rather than inline in the source XML schema.

The binding customization file used in the External Customize example is <*JWSDP_HOME*>/jaxb/samples/external-customize/binding.xjb.

This section compares the customization declarations in bindings.xjb with the analogous declarations used in the XML schema, po.xsd, in the Datatype Converter example. The two sets of declarations achieve precisely the same results.

- JAXB Version, Namespace, and Schema Attributes
- Global and Schema Binding Declarations
- Class Declarations

JAXB Version, Namespace, and Schema Attributes

All JAXB binding declarations files must begin with:

- JAXB version number
- Namespace declarations
- Schema name and node

The version, namespace, and schema declarations in `bindings.xjb` are as follows:

```
<jxb:bindings version="1.0"
               xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
               xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <jxb:bindings schemaLocation="po.xsd" node="/xs:schema">
        .
        <binding_declarations>
        .
  </jxb:bindings>
<!-- schemaLocation="po.xsd" node="/xs:schema" -->
</jxb:bindings>
```

JAXB Version Number

An XML file with a root element of `<jaxb:bindings>` is considered an external binding file. The root element must specify the JAXB version attribute with which its binding declarations must comply; specifically the root `<jxb:bindings>` element must contain either a `<jxb:version>` declaration or a `version` attribute. By contrast, when making binding declarations inline, the JAXB version number is made as attribute of the `<xsd:schema>` declaration:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
            jxb:version="1.0">
```

Namespace Declarations

As shown in JAXB Version, Namespace, and Schema Attributes (page 76), the namespace declarations in the external binding declarations file include both the JAXB namespace and the XMLSchema namespace. Note that the prefixes used in this example could in fact be anything you want; the important thing is to consistently use whatever prefixes you define here in subsequent declarations in the file.

Schema Name and Schema Node

The fourth line of the code in JAXB Version, Namespace, and Schema Attributes (page 76) specifies the name of the schema to which this binding declarations file will apply, and the schema node at which the customizations will first take effect. Subsequent binding declarations in this file will reference specific nodes within the schema, but this first declaration should encompass the schema as a whole; for example, in `bindings.xjb`:

```
<jxb:bindings schemaLocation="po.xsd" node="/xs:schema">
```

Global and Schema Binding Declarations

The global schema binding declarations in `bindings.xjb` are the same as those in `po.xsd` for the Datatype Converter example. The only difference is that because the declarations in `po.xsd` are made inline, you need to embed them in `<xs:appinfo>` elements, which are in turn embedded in `<xs:annotation>` elements. Embedding declarations in this way is unnecessary in the external bindings file.

```
<jxb:globalBindings
    fixedAttributeAsConstantProperty="true"
    collectionType="java.util.Vector"
    typesafeEnumBase="xs:NCName"
    choiceContentProperty="false"
    typesafeEnumMemberName="generateError"
    bindingStyle="elementBinding"
    enableFailFastCheck="false"
    generateIsSetMethod="false"
    underscoreBinding="asCharInWord"/>
<jxb:schemaBindings>
    <jxb:package name="primer.myPo">
        <jxb:javadoc><![CDATA[<body>Package level
documentation for generated package primer.myPo.</body>]]>
        </jxb:javadoc>
    </jxb:package>
    <jxb:nameXmlTransform>
        <jxb:elementName suffix="Element"/>
    </jxb:nameXmlTransform>
</jxb:schemaBindings>
```

By comparison, the syntax used in `po.xsd` for the Datatype Converter example is:

```
<xsd:annotation>
  <xsd:appinfo>
    <jxb:globalBindings
        .
        <binding_declarations>
        .
    <jxb:schemaBindings>
        .
        <binding_declarations>
        .
    </jxb:schemaBindings>
  </xsd:appinfo>
</xsd:annotation>
```

Class Declarations

The class-level binding declarations in `bindings.xjb` differ from the analogous declarations in `po.xsd` for the Datatype Converter example in two ways:

- As with all other binding declarations in bindings.xjb, you do not need to embed your customizations in schema `<xsd:appinfo>` elements.
- You must specify the schema node to which the customization will be applied. The general syntax for this type of declaration is:

  `<jxb:bindings node="//<node_type>[@name='<node_name>']">`

For example, the following code shows binding declarations for the `complexType` named `USAddress`.

```
<jxb:bindings node="//xs:complexType[@name='USAddress']">
  <jxb:class>
    <jxb:javadoc>
<![CDATA[First line of documentation for a <b>USAddress</b>.]]>
    </jxb:javadoc>
  </jxb:class>

  <jxb:bindings node=".//xs:element[@name='name']">
    <jxb:property name="toName"/>
  </jxb:bindings>

  <jxb:bindings node=".//xs:element[@name='zip']">
```

```
      <jxb:property name="zipCode"/>
    </jxb:bindings>
  </jxb:bindings>
  <!-- node="//xs:complexType[@name='USAddress']" -->
```

Note in this example that `USAddress` is the parent of the child elements `name` and `zip`, and therefore a `</jxb:bindings>` tag encloses the `bindings` declarations for the child elements as well as the class-level `javadoc` declaration.

Fix Collides Example

The Fix Collides example illustrates how to resolve name conflicts—that is, places in which a declaration in a source schema uses the same name as another declaration in that schema (namespace collisions), or places in which a declaration uses a name that does translate by default to a legal Java name.

Note: Many name collisions can occur because XSD Part 1 introduces six unique symbol spaces based on type, while Java only has only one. There is a symbols space for type definitions, elements, attributes, and group definitions. As a result, a valid XML schema can use the exact same name for both a type definition and a global element declaration.

For the purposes of this example, it is recommended that you remove the `binding` parameter to the `xjc` task in the `build.xml` file in the `<`*`JWSDP_HOME`*`>/jaxb/samples/fix-collides` directory to display the error output generated by the `xjc` compiler. The XML schema for the Fix Collides, `example.xsd`, contains deliberate name conflicts.

Like the External Customize example, the Fix Collides example uses an external binding declarations file, `binding.xjb`, to define the JAXB binding customizations.

- The example.xsd Schema
- Looking at the Conflicts
- Output From Running the ant Task Without Using a Binding Declarations File
- The binding.xjb Declarations File
- Resolving the Conflicts in example.xsd

The example.xsd Schema

The XML schema, <*JWSDP_HOME*>/jaxb/samples/fix-collides /example.xsd, used in the Fix Collides example illustrates common name conflicts encountered when attempting to bind XML names to unique Java identifiers in a Java package. The schema declarations that result in name conflicts are highlighted in bold below.

```
<xs:schema xmlns:xs=”http://www.w3.org/2001/XMLSchema”
           xmlns:jxb=”http://java.sun.com/xml/ns/jaxb”
           jxb:version=”1.0”>

  <xs:element name=”Class” type=”xs:int”/>
  <xs:element name=”FooBar” type=”FooBar”/>
  <xs:complexType name=”FooBar”>
    <xs:sequence>
      <xs:element name=”foo” type=”xs:int”/>
      <xs:element ref=”Class”/>
      <xs:element name=”zip” type=”xs:integer”/>
    </xs:sequence>
    <xs:attribute name=”zip” type=”xs:string”/>
  </xs:complexType>
</xs:schema>
```

Looking at the Conflicts

The first conflict in example.xsd is the declaration of the element name Class:

```
<xs:element name=”Class” type=”xs:int”/>
```

Class is a reserved word in Java, and while it is legal in the XML schema language, it cannot be used as a name for a schema-derived class generated by JAXB.

When this schema is run against the JAXB binding compiler with the ant fail command, the following error message is returned:

```
[xjc] [ERROR] Attempt to create a property having the same
name as the reserved word "Class".
[xjc] line 6 of example.xsd
```

The second conflict is that there are an `element` and a `complexType` that both use the name `Foobar`:

```
<xs:element name=”FooBar” type=”FooBar”/>
<xs:complexType name=”FooBar”>
```

In this case, the error messages returned are:

```
[xjc] [ERROR] A property with the same name "Zip" is
generated from more than one schema component.
[xjc] line 22 of example.xsd
[xjc] [ERROR] (Relevant to above error) another one is
generated from this schema component.
[xjc] line 20 of example.xsd
```

The third conflict is that there are an `element` and an `attribute` both named `zip`:

```
<xs:element name=”zip” type=”xs:integer”/>
<xs:attribute name=”zip” type=”xs:string”/>
```

The error messages returned here are:

```
[xjc] [ERROR] A property with the same name "Zip" is
generated from more than one schema component.
[xjc] line 22 of example.xsd
[xjc] [ERROR] (Relevant to above error) another one is
generated from this schema component.
[xjc] line 20 of example.xsd
```

Output From Running the ant Task Without Using a Binding Declarations File

Here is the output that is returned if you run the `ant` task in the *`<JWSDP_HOME>`*`/jaxb/samples/fix-collides` directory without specifying the `binding` parameter to the `xjc` task in the `build.xml` file:

```
[echo] Compiling the schema w/o external binding file
(name collision errors expected)...
[xjc] Compiling file:/C:/Sun/jwsdp-1.5/jaxb/samples/
fix-collides/example.xsd
[xjc] [ERROR] Attempt to create a property having the same
name as the reserved word "Class".
[xjc]   line 14 of example.xsd
```

```
[xjc] [ERROR] A property with the same name "Zip" is
generated from more than one schema component.
[xjc]   line 17 of example.xsd
[xjc] [ERROR] (Relevant to above error) another one is
generated from this schema component.
[xjc]   line 15 of example.xsd
[xjc] [ERROR] A class/interface with the same name
"generated.FooBar" is already in use.
[xjc]   line 9 of example.xsd
[xjc] [ERROR] (Relevant to above error) another one is
generated from here.
[xjc]   line 18 of example.xsd
```

The binding.xjb Declarations File

The `<JWSDP_HOME>/jaxb/samples/fix-collides/binding.xjb` binding declarations file resolves the conflicts in `examples.xsd` by means of several customizations.

Resolving the Conflicts in example.xsd

The first conflict in `example.xsd`, using the Java reserved name `Class` for an element name, is resolved in `binding.xjb` with the `<class>` and `<property>` declarations on the schema element node `Class`:

```
<jxb:bindings node="//xs:element[@name='Class']">
  <jxb:class name="Clazz"/>
  <jxb:property name="Clazz"/>
</jxb:bindings>
```

The second conflict in `example.xsd`, the namespace collision between the `element FooBar` and the `complexType FooBar`, is resolved in `binding.xjb` by using a `<nameXmlTransform>` declaration at the `<schemaBindings>` level to append the suffix `Element` to all `element` definitions.

This customization handles the case where there are many name conflicts due to systemic collisions between two symbol spaces, usually named type definitions and global element declarations. By appending a suffix or prefix to every Java identifier representing a specific XML symbol space, this single customization resolves all name collisions:

```
<jxb:schemaBindings>
  <jxb:package name="example"/>
    <jxb:nameXmlTransform>
      <jxb:elementName suffix="Element"/>
    </jxb:nameXmlTransform>
</jxb:schemaBindings>
```

The third conflict in `example.xsd`, the namespace collision between the `element zip` and the `attribute zip`, is resolved in `binding.xjb` by mapping the `attribute zip` to property named `zipAttribute`:

```
<jxb:bindings node=".//xs:attribute[@name='zip']">
  <jxb:property name="zipAttribute"/>
</jxb:bindings>
```

If you add the `binding` parameter you removed back to the `xjc` task in the `build.xml` file and then run `ant` in the `<JWSDP_HOME>/jaxb/samples/fix-collides` directory, the customizations in `binding.xjb` will be passed to the `xjc` binding compiler, which will then resolve the conflicts in `example.xsd` in the schema-derived Java classes.

Bind Choice Example

The Bind Choice example shows how to bind a `choice` model group to a Java interface. Like the External Customize and Fix Collides examples, the Bind Choice example uses an external binding declarations file, `binding.xjb`, to define the JAXB binding customization.

The schema declarations in `<JWSDP_HOME>/jaxb/samples/bind-choice /example.xsd` that will be globally changed are highlighted in bold below.

```
<xs:schema xmlns:xs=”http://www.w3.org/2001/XMLSchema”
           xmlns:jxb=”http://java.sun.com/xml/ns/jaxb”
           jxb:version=”1.0”>

  <xs:element name=”FooBar”>
  <xs:complexType>
    <xs:sequence>
      <xs:element name=”foo” type=”xs:int”/>
      <xs:element ref=”Class”/>
      <xs:choice>
```

```
                <xs:element name="phoneNumber" type="xs:string"/>
                <xs:element name="speedDial" type="xs:int"/>
            </xs:choice>
            <xs:group ref="ModelGroupChoice"/>
        </xs:sequence>
        <xs:attribute name="zip" type="xs:string"/>
    </xs:complexType>
</xs:element>

    <xs:group name="ModelGroupChoice">
        <xs:choice>
            <xs:element name="bool" type="xs:boolean"/>
            <xs:element name="comment" type="xs:string"/>
            <xs:element name="value" type="xs:int"/>
        </xs:choice>
    </xs:group>
</xs:schema>
```

Customizing a choice Model Group

The *<JWSDP_HOME>*`/jaxb/samples/bind-choice/binding.xjb` binding declarations file demonstrates one way to override the default derived names for `choice` model groups in `example.xsd` by means of a `<jxb:globalBindings>` declaration:

```
<jxb:bindings schemaLocation="example.xsd" node="/xs:schema">
  <jxb:globalBindings bindingStyle="modelGroupBinding"/>
    <jxb:schemaBindings/>
      <jxb:package name="example"/>
    </jxb:schemaBindings>
  </jxb:bindings
</jxb:bindings>
```

This customization results in the `choice` model group being bound to its own content interface. For example, given the following `choice` model group:

```
<xs:group name="ModelGroupChoice">
  <xs:choice>
    <xs:element name="bool" type="xs:boolean"/>
    <xs:element name="comment" type="xs:string"/>
    <xs:element name="value" type="xs:int"/>
  </xs:choice>
</xs:group>
```

the globalBindings customization shown above causes JAXB to generate the following Java class:

```
/**
 * Java content class for model group.
 */
  public interface ModelGroupChoice {
        int getValue();
        void setValue(int value);
        boolean isSetValue();

        java.lang.String getComment();
        void setComment(java.lang.String value);
        boolean isSetComment();

        boolean isBool();
        void setBool(boolean value);
        boolean isSetBool();

        Object getContent();
        boolean isSetContent();
        void unSetContent();
    }
```

Calling getContent returns the current value of the Choice content. The setters of this choice are just like radio buttons; setting one unsets the previously set one. This class represents the data representing the choice.

Additionally, the generated Java interface FooBarType, representing the anonymous type definition for element FooBar, contains a nested interface for the choice model group containing phoneNumber and speedDial.

3

Streaming API for XML

This chapter focuses on the Streaming API for XML (StAX), a streaming Java-based, event-driven, pull-parsing API for reading and writing XML documents. StAX enables you to create bidrectional XML parsers that are fast, relatively easy to program, and have a light memory footprint.

StAX provides is the latest API in the JAXP family, and provides an alternative to SAX, DOM, TrAX, and DOM for developers looking to do high-performance stream filtering, processing, and modification, particularly with low memory and limited extensibility requirements.

Note: To synopsize, StAX provides a standard, bidirectional *pull parser* interface for streaming XML processing, offering a simpler programming model than SAX and more efficient memory management than DOM. StAX enables developers to parse and modify XML streams as events, and to extend XML information models to allow application-specific additions. More detailed comparisons of StAX with several alternative APIs are provided below, in “Comparing StAX to Other JAXP APIs.”

Why StAX?

The StAX project was spearheaded by BEA with support from Sun Microsystems, and the JSR 173 specification passed the Java Community Process final approval ballot in March, 2004 (`http://jcp.org/en/jsr/detail?id=173`). The primary goal of the StAX API is to give “parsing control to the programmer by exposing a simple iterator based API. This allows the programmer to ask for the next event (pull the event) and allows state to be stored in procedural fashion." StAX was created to address limitations in the two most prevalent parsing APIs, SAX and DOM.

Streaming Versus DOM

Generally speaking, there are two programming models for working with XML infosets: document *streaming* and the *document object model* (DOM).

The *DOM* model involves creating in-memory objects representing an entire document tree and the complete infoset state for an XML document. Once in memory, DOM trees can be navigated freely and parsed arbitrarily, and as such provide maximum flexibility for developers. However the cost of this flexibility is a potentially large memory footprint and significant processor requirements, as the entire representation of the document must be held in memory as objects for the duration of the document processing. This may not be an issue when working with small documents, but memory and processor requirements can escalate quickly with document size.

Streaming refers to a programming model in which XML infosets are transmitted and parsed serially at application runtime, often in real time, and often from dynamic sources whose contents are not precisely known beforehand. Moreover, stream-based parsers can start generating output immediately, and infoset elements can be discarded and garbage collected immediately after they are used. While providing a smaller memory footprint, reduced processor requirements, and higher performance in certain situations, the primary trade-off with stream processing is that you can only see the infoset state at one location at a time in the document. You are essentially limited to the "cardboard tube" view of a document, the implication being that you need to know what processing you want to do before reading the XML document.

Streaming models for XML processing are particularly useful when your application has strict memory limitations, as with a cellphone running J2ME, or when your application needs to simultaneously process several requests, as with an application server. In fact, it can be argued that the majority of XML business logic can benefit from stream processing, and does not require the in-memory maintenance of entire DOM trees.

Pull Parsing Versus Push Parsing

Streaming *pull parsing* refers to a programming model in which a client application calls methods on an XML parsing library when it needs to interact with an XML infoset—that is, the client only gets (pulls) XML data when it explicitly asks for it.

Streaming *push parsing* refers to a programming model in which an XML parser sends (pushes) XML data to the client as the parser encounters elements in an XML infoset—that is, the parser sends the data whether or not the client is ready to use it at that time.

Pull parsing provides several advantages over push parsing when working with XML streams:

- With pull parsing, the client controls the application thread, and can call methods on the parser when needed. By contrast, with push processing, the parser controls the application thread, and the client can only accept invocations from the parser.
- Pull parsing libraries can be much smaller and the client code to interact with those libraries much simpler than with push libraries, even for more complex documents.
- Pull clients can read multiple documents at one time with a single thread.
- A StAX pull parser can filter XML documents such that elements unnecessary to the client can be ignored, and it can support XML views of non-XML data.

StAX Use Cases

The StAX specification defines a number of uses cases for the API:

- **Data binding**
  - Unmarshalling an XML document
  - Marshalling an XML document
  - Parallel document processing
  - Wireless communication
- **SOAP message processing**
  - Parsing simple predictable structures
  - Parsing graph representations with forward references

    - Parsing WSDL
- **Virtual data sources**
    - Viewing as XML data stored in databases
    - Viewing data in Java objects created by XML data binding
    - Navigating a DOM tree as a stream of events
- **Parsing specific XML vocabularies**
- **Pipelined XML processing**

A complete discussion of all these use cases is beyond the scope of this chapter. Please refer to the StAX specification for further information.

Comparing StAX to Other JAXP APIs

As an API in the JAXP family, StAX can be compared, among other APIs, to SAX, TrAX, and JDOM. Of the latter two, StAX is not as powerful or flexible as TrAX or JDOM, but neither does it require as much memory or processor load to be useful, and StAX can, in many cases, outperform the DOM-based APIs. The same arguments outlined above, weighing the cost/benefits of the DOM model versus the streaming model, apply here.

With this in mind, the closest comparisons between can be made between StAX and SAX, and it is here that StAX offers features that are beneficial in many cases; some of these include:

- StAX-enabled clients are generally easier to code than SAX clients. While it can be argued that SAX parsers are marginally easier to write, StAX parser code can be smaller and the code necessary for the client to interact with the parser simpler.
- StAX is a bidirectional API, meaning that it can both read and write XML documents. SAX is read only, so another API is needed if you want to write XML documents.
- SAX is a push API, whereas StAX is pull. The trade-offs between push and pull APIs outlined above apply here.

Table 3–1 synopsizes the comparative features of StAX, SAX, DOM, and TrAX (table adapted from “Does StAX Belong in Your XML Toolbox?” (`http://www.developer.com/xml/article.php/3397691`) by Jeff Ryan).

Table 3–1 XML Parser API Feature Summary

Feature	**StAX**	**SAX**	**DOM**	**TrAX**
API Type	Pull, streaming	Push, streaming	In memory tree	XSLT Rule
Ease of Use	High	Medium	High	Medium
XPath Capability	No	No	Yes	Yes
CPU and Memory Efficiency	Good	Good	Varies	Varies
Forward Only	Yes	Yes	No	No
Read XML	Yes	Yes	Yes	Yes
Write XML	Yes	No	Yes	Yes
Create, Read, Update, Delete	No	No	Yes	No

StAX API

The StAX API exposes methods for iterative, event-based processing of XML documents. XML documents are treated as a filtered series of events, and infoset states can be stored in a procedural fashion. Moreover, unlike SAX, the StAX API is bidirectional, enabling both reading and writing of XML documents.

The StAX API is really two distinct API sets: a *cursor* API and an *iterator* API. These two API sets explained in greater detail later in this chapter, but their main features are briefly described below.

Cursor API

As the name implies, the StAX *cursor* API represents a cursor with which you can walk an XML document from beginning to end. This cursor can point to one thing at a time, and always moves forward, never backward, usually one infoset element at a time.

The two main cursor interfaces are XMLStreamReader and XMLStreamWriter. XMLStreamReader includes accessor methods for all possible information retrievable from the XML Information model, including document encoding, element names, attributes, namespaces, text nodes, start tags, comments, processing instructions, document boundaries, and so forth; for example:

```
public interface XMLStreamReader {
  public int next() throws XMLStreamException;
  public boolean hasNext() throws XMLStreamException;
  public String getText();
  public String getLocalName();
  public String getNamespaceURI();
  // ... other methods not shown
}
```

You can call methods on XMLStreamReader, such as getText and getName, to get data at the current cursor location. XMLStreamWriter provides methods that correspond to StartElement and EndElement event types; for example:

```
public interface XMLStreamWriter {
  public void writeStartElement(String localName) \
     throws XMLStreamException;
  public void writeEndElement() \
     throws XMLStreamException;
  public void writeCharacters(String text) \
     throws XMLStreamException;
// ... other methods not shown
}
```

The cursor API mirrors SAX in many ways. For example, methods are available for directly accessing string and character information, and integer indexes can be used to access attribute and namespace information. As with SAX, the cursor API methods return XML information as strings, which minimizes object allocation requirements.

Iterator API

The StAX *iterator* API represents an XML document stream as a set of discrete event objects. These events are pulled by the application and provided by the parser in the order in which they are read in the source XML document.

The base iterator interface is called XMLEvent, and there are subinterfaces for each event type listed in Table 3–2, below. The primary parser interface for reading iterator events is XMLEventReader, and the primary interface for writing iterator events is XMLEventWriter. The XMLEventReader interface contains five methods, the most important of which is nextEvent(), which returns the next event in an XML stream. XMLEventReader implements java.util.Iterator, which means that returns from XMLEventReader can be cached or passed into routines that can work with the standard Java Iterator; for example:

```
public interface XMLEventReader extends Iterator {
  public XMLEvent nextEvent() throws XMLStreamException;
  public boolean hasNext();
  public XMLEvent peek() throws XMLStreamException;
  ...
}
```

Similarly, on the output side of the iterator API, you have:

```
public interface XMLEventWriter {
  public void flush() throws XMLStreamException;
  public void close() throws XMLStreamException;
  public void add(XMLEvent e) throws XMLStreamException;
  public void add(Attribute attribute) \
    throws XMLStreamException;
  ...
}
```

Iterator Event Types

Table 3–2 lists the thirteen XMLEvent types defined in the event iterator API.

Table 3–2 XMLEvent Types

Event Type	Description
StartDocu-ment	Reports the beginning of a set of XML events, including encoding, XML version, and standalone properties.
StartEle-ment	Reports the start of an element, including any attributes and namespace declarations; also provides access to the prefix, namespace URI, and local name of the start tag.
EndElement	Reports the end tag of an element. Namespaces that have gone out of scope can be recalled here if they have been explicitly set on their corresponding StartElement.

Table 3–2 `XMLEvent` Types (Continued)

Event Type	**Description**
`Characters`	Corresponds to XML `CData` sections and `CharacterData` entities. Note that ignorable whitespace and significant whitespace are also reported as `Character` events.
`EntityReference`	Character entities can be reported as discrete events, which an application developer can then choose to resolve or pass through unresolved. By default, entities are resolved. Alternatively, if you do not want to report the entity as an event, replacement text can be substituted and reported as `Characters`.
`ProcessingInstruction`	Reports the target and data for an underlying processing instruction.
`Comment`	Returns the text of a comment
`EndDocument`	Reports the end of a set of XML events.
`DTD`	Reports as `java.lang.String` information about the DTD, if any, associated with the stream, and provides a method for returning custom objects found in the DTD.
`Attribute`	Attributes are generally reported as part of a `StartElement` event. However, there are times when it is desirable to return an attribute as a standalone `Attribute` event; for example, when a namespace is returned as the result of an `XQuery` or `XPath` expression.
`Namespace`	As with attributes, namespaces are usually reported as part of a `StartElement`, but there are times when it is desirable to report a namespace as a discrete `Namespace` event.

Note that the `DTD`, `EntityDeclaration`, `EntityReference`, `NotationDeclaration`, and `ProcessingInstruction` events are only created if the document being processed contains a DTD.

Sample Event Mapping

As an example of how the event iterator API maps an XML stream, consider the following XML document:

```
<?xml version="1.0"?>
<BookCatalogue xmlns="http://www.publishing.org">
  <Book>
    <Title>Yogasana Vijnana: the Science of Yoga</Title>
    <ISBN>81-40-34319-4</ISBN>
    <Cost currency="INR">11.50</Cost>
  </Book>
</BookCatalogue>
```

This document would be parsed into eighteen primary and secondary events, as shown below. Note that secondary events, shown in curly braces ({}), are typically accessed from a primary event rather than directly.

Table 3–3 Sample Iterator API Event Mapping

#	Element/Attribute	Event
1	`version="1.0"`	`StartDocument`

2	`isCData = false`
	`data = “\n”`
	`IsWhiteSpace = true`
		`Characters`

3	`qname = BookCatalogue:http://www.publishing.org`
	`attributes = null`
	`namespaces = {BookCatalogue” -> http://www.publishing.org”}`
		`StartElement`

4	`qname = Book`
	`attributes = null`
	`namespaces = null`
		`StartElement`

5	`qname = Title`
	`attributes = null`
	`namespaces = null`
		`StartElement`

6	`isCData = false`
	`data = “Yogasana Vijnana: the Science of Yoga\n\t”`
	`IsWhiteSpace = false`
		`Characters`

7	`qname = Title`
	`namespaces = null`
		`EndElement`

Table 3–3 Sample Iterator API Event Mapping (Continued)

#	Element/Attribute	Event

8	`qname = ISBN`
	`attributes = null`
	`namespaces = null`
		`StartElement`

9	`isCData = false`
	`data = “81-40-34319-4\n\t”`
	`IsWhiteSpace = false`
		`Characters`

10	`qname = ISBN`
	`namespaces = null`
		`EndElement`

11	`qname = Cost`
	`attributes = {“currency” -> INR}`
	`namespaces = null`
		`StartElement`

12	`isCData = false`
	`data = “11.50\n\t”`
	`IsWhiteSpace = false`
		`Characters`

13	`qname = Cost`
	`namespaces = null`
		`EndElement`

14	`isCData = false`
	`data = “\n”`
	`IsWhiteSpace = true`
		`Characters`

15	`qname = Book`
	`namespaces = null`
		`EndElement`

16	`isCData = false`
	`data = “\n”`
	`IsWhiteSpace = true`
		`Characters`

17	`qname = BookCatalogue:http://www.publishing.org`
	`namespaces = {BookCatalogue” -> http://www.publishing.org”}`
		`EndElement`

18		`EndDocument`

There are several important things to note in the above example:

- The events are created in the order in which the corresponding XML elements are encountered in the document, including nesting of elements, opening and closing of elements, attribute order, document start and document end, and so forth.

- As with proper XML syntax, all container elements have corresponding start and end events; for example, every `StartElement` has a corresponding `EndElement`, even for empty elements.
- `Attribute` events are treated as secondary events, and are accessed from their corresponding `StartElement` event.
- Similar to `Attribute` events, `Namespace` events are treated as secondary, but appear twice and are accessible twice in the event stream, first from their corresponding `StartElement` and then from their corresponding `EndElement`.
- `Character` events are specified for all elements, even if those elements have no character data. Similarly, `Character` events can be split across events.
- The StAX parser maintains a namespace stack, which holds information about all XML namespaces defined for the current element and its ancestors. The namespace stack is exposed through the `javax.xml.namespace.NamespaceContext` interface, and can be accessed by namespace prefix or URI.

Choosing Between Cursor and Iterator APIs

It is reasonable to ask at this point, “What API should I choose? Should I create instances of `XMLStreamReader` or `XMLEventReader`? Why are there two kinds of APIs anyway?”

Development Goals

The authors of the StAX specification targeted three types of developers:

- **Library and infrastructure developers** – Create application servers, JAXM, JAXB, JAX-RPC and similar implementations; need highly efficient, low-level APIs with minimal extensibility requirements.
- **J2ME developers** – Need small, simple, pull-parsing libraries, and have minimal extensibility needs.

- **J2EE and J2SE developers** – Need clean, efficient pull-parsing libraries, plus need the flexibility to both read and write XML streams, create new event types, and extend XML document elements and attributes.

Given these wide-ranging development categories, the StAX authors felt it was more useful to define two small, efficient APIs rather than overloading one larger and necessarily more complex API.

Comparing Cursor and Iterator APIs

Before choosing between the cursor and iterator APIs, you should note a few things that you can do with the iterator API that you cannot do with cursor API:

- Objects created from the `XMLEvent` subclasses are immutable, and can be used in arrays, lists, and maps, and can be passed through your applications even after the parser has moved on to subsequent events.
- You can create subtypes of `XMLEvent` that are either completely new information items or extensions of existing items but with additional methods.
- You can add and remove events from an XML event stream in much simpler ways than with the cursor API.

Similarly, keep some general recommendations in mind when making your choice:

- If you are programming for a particularly memory-constrained environment, like J2ME, you can make smaller, more efficient code with the cursor API.
- If performance is your highest priority—for example, when creating low-level libraries or infrastructure—the cursor API is more efficient.
- If you want to create XML processing pipelines, use the iterator API.
- If you want to modify the event stream, use the iterator API.
- If you want to your application to be able to handle pluggable processing of the event stream, use the iterator API.
- In general, if you do not have a strong preference one way or the other, using the iterator API is recommended because it is more flexible and extensible, thereby "future-proofing" your applications.

Using StAX

In general, StAX programmers create XML stream readers, writers, and events by using the `XMLInputFactory`, `XMLOutputFactory` and `XMLEventFactory` classes. Configuration is done by setting properties on the factories, whereby implementation-specific settings can be passed to the underlying implementation using the `setProperty()` method on the factories. Similarly, implementation-specific settings can be queried using the `getProperty()` factory method.

The `XMLInputFactory`, `XMLOutputFactory` and `XMLEventFactory` classes are described below, followed by discussions of resource allocation, namespace and attribute management, error handling, and then finally reading and writing streams using the cursor and iterator APIs.

StAX Factory Classes

XMLInputFactory

The `XMLInputFactory` class lets you configure implementation instances of XML stream reader processors created by the factory. New instances of the abstract class `XMLInputFactory` are created by calling the `newInstance()` method on the class. The static method `XMLInputFactory.newInstance()` is then used to create a new factory instance.

Deriving from JAXP, the `XMLInputFactory.newInstance()` method determines the specific `XMLInputFactory` implementation class to load by using the following lookup procedure:

1. Use the `javax.xml.stream.XMLInputFactory` system property.
2. Use the `lib/xml.stream.properties` file in the JRE directory.
3. Use the Services API, if available, to determine the classname by looking in the `META-INF/services/javax.xml.stream.XMLInputFactory` files in jars available to the JRE.
4. Use the platform default `XMLInputFactory` instance.

After getting a reference to an appropriate XMLInputFactory, an application can use the factory to configure and create stream instances. Table 3–4 lists the properties supported by `XMLInputFactory`. See the StAX specification for a more detailed listing.

Table 3–4 XMLInputFactory Properties

Property	**Description**
javax.xml.stream.isValidating	Turns on implementation specific validation.
javax.xml.stream.isCoalescing	*(Required)* Requires the processor to coalesce adjacent character data.
javax.xml.stream.isNamespaceAware	Turns off namespace support. All implementations must support namespaces supporting non-namespace aware documents is optional.
javax.xml.stream.isReplacingEntityReferences	*(Required)* Requires the processor to replace internal entity references with their replacement value and report them as characters or the set of events that describe the entity.
javax.xml.stream.isSupportingExternalEntities	*(Required)* Requires the processor to resolve external parsed entities.
javax.xml.stream.reporter	*(Required)* Sets and gets the implementation of the XMLReporter
javax.xml.stream.resolver	*(Required)* Sets and gets the implementation of the XMLResolver interface
javax.xml.stream.allocator	*(Required)* Sets/gets the implementation of the XMLEventAllocator interface

XMLOutputFactory

New instances of the abstract class `XMLOutputFactory` are created by calling the `newInstance()` method on the class. The static method `XMLOutputFactory.newInstance()` is then used to create a new factory instance. The algorithm used to obtain the instance is the same as for `XMLInputFactory` but references the `javax.xml.stream.XMLOutputFactory` system property.

`XMLOutputFactory` supports only one property, `javax.xml.stream.isRepairingNamespaces`. This property is required, and its purpose is to create default prefixes and associate them with Namespace URIs. See the StAX specification for a more information.

XMLEventFactory

New instances of the abstract class `XMLEventFactory` are created by calling the `newInstance()` method on the class. The static method `XMLEventFactory.newInstance()` is then used to create a new factory instance. This factory references the `javax.xml.stream.XMLEventFactory` property to instantiate the factory. The algorithm used to obtain the instance is the same as for `XMLInputFactory` and `XMLOutputFactory` but references the `javax.xml.stream.XMLEventFactory` system property.

There are no default properties for `XMLEventFactory`.

Resources, Namespaces, and Errors

The StAX specification handles resource allocation, attributes and namespace, and errors and exceptions as described below.

Resource Resolution

The `XMLResolver` interface provides a means to set the method that resolves resources during XML processing. An application sets the interface on `XMLInputFactory`, which then sets the interface on all processors created by that factory instance.

Attributes and Namespaces

Attributes are reported by a StAX processor using lookup methods and strings in the cursor interface and `Attribute` and `Namespace` events in the iterator interface. Note here that namespaces are treated as attributes, although namespaces are reported separately from attributes in both the cursor and iterator APIs. Note also that namespace processing is optional for StAX processors. See the StAX specification for complete information about namespace binding and optional namespace processing.

Error Reporting and Exception Handling

All fatal errors are reported by way of `javax.xml.stream.XMLStreamException`. All nonfatal errors and warnings are reported using the `javax.xml.stream.XMLReporter` interface.

Reading XML Streams

As described earlier in this chapter, the way you read XML streams with a StAX processor—and more importantly, what you get back—varies significantly depending on whether you are using the StAX cursor API or the event iterator API. The following two sections describe how to read XML streams with each of these APIs.

Using XMLStreamReader

The `XMLStreamReader` interface in the StAX cursor API lets you read XML streams or documents in a forward direction only, one item in the infoset at a time. The following methods are available for pulling data from the stream or skipping unwanted events:

- Get the value of an attribute
- Read XML content
- Determine whether an element has content or is empty
- Get indexed access to a collection of attributes
- Get indexed access to a collection of namespaces
- Get the name of the current event (if applicable)
- Get the content of the current event (if applicable)

Instances of `XMLStreamReader` have at any one time a single current event on which its methods operate. When you create an instance of `XMLStreamReader` on a stream, the initial current event is the `START_DOCUMENT` state.The `XMLStreamReader.next()` method can then be used to step to the next event in the stream.

Reading Properties, Attributes, and Namespaces

The `XMLStreamReader.next()` method loads the properties of the next event in the stream. You can then access those properties by calling the `XMLStreamReader.getLocalName()` and `XMLStreamReader.getText()` methods.

When the XMLStreamReader cursor is over a StartElement event, it reads the name and any attributes for the event, including the namespace. All attributes for an event can be accessed using an index value, and can also be looked up by namespace URI and local name. Note, however, that only the namespaces declared on the current StartEvent are available; previously declared namespaces are not maintained, and redeclared namespaces are not removed.

XMLStreamReader Methods

XMLStreamReader provides the following methods for retrieving information about namespaces and attributes:

```
int getAttributeCount();
String getAttributeNamespace(int index);
String getAttributeLocalName(int index);
String getAttributePrefix(int index);
String getAttributeType(int index);
String getAttributeValue(int index);
String getAttributeValue(String namespaceUri,String
localName);
boolean isAttributeSpecified(int index);
```

Namespaces can also be accessed using three additional methods:

```
int getNamespaceCount();
String getNamespacePrefix(int index);
String getNamespaceURI(int index);
```

Instantiating an XMLStreamReader

This example, taken from the StAX specification, shows how to instantiate an input factory, create a reader, and iterate over the elements of an XML stream:

```
XMLInputFactory f = XMLInputFactory.newInstance();
XMLStreamReader r = f.createXMLStreamReader( ... );
while(r.hasNext()) {
  r.next();
}
```

Using XMLEventReader

The XMLEventReader API in the StAX event iterator API provides the means to map events in an XML stream to allocated event objects that can be freely reused, and the API itself can be extended to handle custom events.

XMLEventReader provides four methods for iteratively parsing XML streams:

- `next()` – Returns the next event in the stream
- `nextEvent()` – Returns the next typed XMLEvent
- `hasNext()` – Returns true if there are more events to process in the stream
- `peek()` – Returns the event but does not iterate to the next event

For example, the following code snippet illustrates the `XMLEventReader` method declarations:

```
package javax.xml.stream;
import java.util.Iterator;
public interface XMLEventReader extends Iterator {
  public Object next();
  public XMLEvent nextEvent() throws XMLStreamException;
  public boolean hasNext();
  public XMLEvent peek() throws XMLStreamException;
...
}
```

To read all events on a stream and then print them, you could use the following:

```
while(stream.hasNext()) {
XMLEvent event = stream.nextEvent();
System.out.print(event);
}
```

Reading Attributes

You can access attributes from their associated `javax.xml.stream.StartElement`, as follows:

```
public interface StartElement extends XMLEvent {
  public Attribute getAttributeByName(QName name);
  public Iterator getAttributes();
}
```

You can use the `getAttributes()` method on the `StartElement` interface to use an `Iterator` over all the attributes declared on that `StartElement`.

Reading Namespaces

Similar to reading attributes, namespaces are read using an `Iterator` created by calling the `getNamespaces()` method on the `StartElement` interface. Only the namespace for the current `StartElement` is returned, and an application can get the current namespace context by using `StartElement.getNamespaceContext()`.

Writing XML Streams

StAX is a bidirectional API, and both the cursor and event iterator APIs have their own set of interfaces for writing XML streams. As with the interfaces for reading streams, there are significant differences between the writer APIs for cursor and event iterator. The following sections describe how to write XML streams using each of these APIs.

Using XMLStreamWriter

The `XMLStreamWriter` interface in the StAX cursor API lets applications write back to an XML stream or create entirely new streams. XMLStreamWriter has methods that let you:

- Write well-formed XML
- Flush or close the output
- Write qualified names

Note that `XMLStreamWriter` implementations are not required to perform well-formedness or validity checks on input. While some implementations my perform strict error checking, others may not. The rules you choose to implement are set on properties provided by the `XMLOutputFactory` class.

The `writeCharacters(...)` method is used to escape characters such as `&`, `<`, `>`, and “. Binding prefixes can be handled by either passing the actual value for the prefix, by using the `setPrefix()` method, or by setting the property for defaulting namespace declarations.

The following example, taken from the StAX specification, shows how to instantiate an output factory, create a writer and write XML output:

```
XMLOutputFactory output = XMLOutputFactory.newInstance();
XMLStreamWriter writer = output.createXMLStreamWriter( ... );
writer.writeStartDocument();
writer.setPrefix("c","http://c");
writer.setDefaultNamespace("http://c");
writer.writeStartElement("http://c","a");
writer.writeAttribute("b","blah");
writer.writeNamespace("c","http://c");
writer.writeDefaultNamespace("http://c");
writer.setPrefix("d","http://c");
writer.writeEmptyElement("http://c","d");
writer.writeAttribute("http://c","chris","fry");
writer.writeNamespace("d","http://c");
writer.writeCharacters("foo bar foo");
writer.writeEndElement();
writer.flush();
```

This code generates the following XML (new lines are non-normative)

```
<?xml version='1.0' encoding='utf-8'?>
<a b="blah" xmlns:c="http://c" xmlns="http://c">
<d:d d:chris="fry" xmlns:d="http://c"/>foo bar foo</a>
```

Using XMLEventWriter

The `XMLEventWriter` interface in the StAX event iterator API lets applications write back to an XML stream or create entirely new streams. This API can be extended, but the main API is as follows:

```
public interface XMLEventWriter {
  public void flush() throws XMLStreamException;
  public void close() throws XMLStreamException;
  public void add(XMLEvent e) throws XMLStreamException;
  // ... other methods not shown.
}
```

Instances of `XMLEventWriter` are created by an instance of `XMLOutputFactory`. Stream events are added iteratively, and an event cannot be modified after it has been added to an event writer instance.

Attributes, Escaping Characters, Binding Prefixes

StAX implementations are required to buffer the last `StartElement` until an event other than `Attribute` or `Namespace` is added or encountered in the stream. This means that when you add an `Attribute` or a `Namespace` to a stream, it is appended the current `StartElement` event.

You can use the `Characters` method to escape characters like `&`, `<`, `>`, and “.

The `setPrefix(...)` method can be used to explicitly bind a prefix for use during output, and the `getPrefix(...)` method can be used to get the current prefix. Note that by default, `XMLEventWriter` adds namespace bindings to its internal namespace map. Prefixes go out of scope after the corresponding `EndElement` for the event in which they are bound.

Sun's Streaming Parser Implementation

The JWSDP 1.6 includes an Early Access (EA) release of Sun Microsystem's JSR 173 (StAX) implementation, called the Sun Java Streaming XML Parser (SJSXP). The SJSXP is a high-speed, non-validating, W3C XML 1.0 and Namespace 1.0-compliant streaming XML pull parser built upon the Xerces2 codebase.

In Sun's SJSXP implementation, the Xerces2 lower layers, particularly the Scanner and related classes, have been redesigned to behave in a pull fashion. In addition to the changes the lower layers, the SJSXP includes additional StAX-related functionality and many performance-enhancing improvements. The SJSXP is implemented in `sjsxp.jar`, which is located in the `<JWSDP_HOME>/sjsxp/lib` directory.

Included with this SJSXP EA distribution are code samples that illustrate how the implementation works. These samples are described in the Sample Code section, later in this chapter.

Before proceeding with the sample code, there are three important aspects of the SJSXP about which you should be aware:

- SJSXP JAR Files
- Reporting CDATA Events
- SJSXP Factories Implementation

These three topics are discussed below.

SJSXP JAR Files

There are two JAR files in the SJSXP implementation. Both of these JARs are located in the *<JWSDP_HOME>*`/sjsxp/lib` directory:

- **sjsxp.jar** – Sun implementation JAR for SJSXP
- **jsr173_api.jar** – Standard API JAR for JSR 173

Complete listings of the contents of these two JARs are provided in Appendix B, "SJSXP JAR Files."

Reporting CDATA Events

The javax.xml.stream.XMLStreamReader implemented in the SJSXP does not report CDATA events. If you have an application that needs to receive such events, configure the XMLInputFactory to set the following implementation-specific "report-cdata-event" property:

```
XMLInputFactory factory = XMLInptuFactory.newInstance();
factory.setProperty("report-cdata-event", Boolean.TRUE);
```

SJSXP Factories Implementation

Most applications do not need to know the factory implementation class name. Just adding the sjsxp.jar file to the classpath is sufficient for most applications because sjsxp.jar supplies the factory implementation classname for various SJSXP properties under the META-INF/services directory—for example, javax.xml.stream.XMLInputFactory, javax.xml.stream.XMLOutputFactory, and javax.xml.stream.XMLEventFactory—which is the third step of a lookup operation when an application asks for the factory instance. See the javadoc for the XMLInputFactory.newInstance() method for more information about the lookup mechanism.

However, there may be scenarios when an application would like to know about the factory implementation class name and set the property explicitly. These scenarios could include cases where there are multiple JSR 173 implementations in the classpath and the application wants to choose one, perhaps one that has superior performance, contains a crucial bug fix, or suchlike.

If an application sets the SystemProperty, it is the first step in a lookup operation, and so obtaining the factory instance would be fast compared to other options; for example:

```
javax.xml.stream.XMLInputFactory -->
com.sun.xml.stream.ZephyrParserFactory
javax.xml.stream.XMLOutputFactory -->
com.sun.xml.stream.ZephyrWriterFactor
javax.xml.stream.XMLEventFactory -->
com.sun.xml.stream.events.ZephyrEventFactory
```

Sample Code

This section steps through the sample StAX code included in the JWSDP 1.6 bundle. All sample directories used in this section are located off the `<JWSDP_HOME>/sjsxp/samples` directory. The sample XML file used here is located in the `data` directory off of `samples`.

There are seven sample directories distributed with JWSDP 1.6:

- **cursor** contains `CursorParse.java`, which illustrates how to use the `XMLStreamReader` (cursor) API to read an XML file.
- **cursor2event** contains `CursorApproachEventObject.java`, which illustrates how an application can get information as an `XMLEvent` object when using cursor API.
- **data** contains `BookCatalogue.xml`, which is the XML document used by the sample classes.
- **event** contains `EventParse.java`, which illustrates how to use the `XMLEventReader` (event iterator) API to read an XML file.
- **filter** contains `MyStreamFilter.java`, which illustrates how to use the Stax Stream Filter APIs. In this example, the filter accepts only `StartElement` and `EndElement` events and filters out the remainder of the events.
- **readnwrite** contains `EventProducerConsumer.java`, which illustrates how the StAX producer/consumer mechanism can be used to simultaneously read and write XML streams.
- **writer** contains `CursorWriter.java`, which illustrates how to use `XMLStreamWriter` to write an XML file programatically.

Configuring Your Environment for Running the Samples

The instructions for configuring your environment are basically the same as those required for running the JWSDP in general. In addition to these general instructions, you should also set the following environment variables:

- `PATH=`*`<JWSDP_HOME>`*`/apache-ant/bin:$PATH`
- `ANT_HOME=`*`<JWSDP_HOME>`*`/apache-ant`
- `CLASSPATH=`*`<JWSDP_HOME>`*`/sjsxp/lib/:$CLASSPATH`

Running the Samples

The samples can be run either manually or by means of several Ant targets, defined in the `<JWSDP_HOME>/sjsxp/samples/build.xml` file. It is easiest to run the samples using the Ant targets.

When you run any of the samples, the compiled class files are placed in a directory named `./build`. This directory is created if it does not exist already.

Running the Samples Using Ant

Use the Ant build file (`build.xml`) in the `<JWSDP_HOME>/sjsxp/samples` directory to run the SJSXP samples. There are eight targets defined in SJSXP `build.xml` file:

- **all** – Compile and run all classes; default target
- **compile** – Only compile classes; do not run
- **cursor.CursorParse** – Compile and run `./cursor/CursorParse.java`
- **cursor2event.CursorApproachEventObject** – Compile and run `./cursor2event/CursorApproachEventObject.java`
- **event.EventParse** – Compile and run `./event/`EventParse.java
- **filter.MyStreamFilter** – Compile and run `./filter/MyStreamFilter.java`
- **readnwrite.EventProducerConsumer** – Compile and run `./readnwrite/EventProducerConsumer.java`
- **writer.CursorWriter** – Compile and run `./writer/CursorWriter.java`

To run any of the Ant targets, change to the `<JWSDP_HOME>/sjsxp/samples` directory and invoke the target you want; for example:

```
cd jwsdp.home/sjsxp/samples
ant cursor.CursorParse
```

Note: If the StAX (JSR 173) API JAR file is not named `jsr173_api.jar`, or is not in the same directory as the `sjsxp.jar` file, you will get an error when you run the samples. If this occurs, you should tell Ant the location of the StAX APIs by overriding the `stax.api.jar` property as shown:

```
ant -Dstax.api.jar="<JSR 173 API LOCATION>" cursor.CursorParse
```

If Ant cannot find the `sjsxp.jar` file, override the `sjsxp.jar` property as shown:

```
ant -Dsjsxp.jar="sjsxp.jar location" cursor.CursorParse
```

Running the Samples Manually

You can also run the samples manually. To do so, go to the <JWSDP_HOME/sjsxp/samples directory and change to the directory that contains the sample you want to run. For example, to run the CursorParse.java sample:

1. Change to the directory containing the CursorParse.java file:
   ```
   cd <JWSDP_HOME>/sjsxp/samples/cursor
   ```
2. Compile CursorParse.java:
   ```
   javac -classpath ../lib/jsr173_api.jar CursorParse.java
   ```
   Note that if the jsr173_api.jar is in your CLASSPATH, you do not need to use the -classpath option here.
3. Run the CursorParse sample:
   ```
   java -classpath .:../lib/sjsxp.jar:../lib/jsr173_api.jar
   cursor.CursorParse -x 1 ./samples/data/BookCatalogue.xml
   ```
   Again, if the jsr173_api.jar and sjsxp.jar files are in your CLASSPATH, you do not need to use the -classpath option here.

Sample XML Document

The sample XML document, BookCatalogue.xml, used by most of the SJSXP sample classes is located in the <JWSDP_HOME>/sjsxp/samples/data directory, and is a simple book catalog based on the common BookCatalogue namespace. The contents of BookCatalogue.xml are listed below:

```
<?xml version="1.0"?>
<BookCatalogue xmlns="http://www.publishing.org">
  <Book>
     <Title>Yogasana Vijnana: the Science of Yoga</Title>
     <Author>Dhirendra Brahmachari</Author>
     <Date>1966</Date>
     <ISBN>81-40-34319-4</ISBN>
     <Publisher>Dhirendra Yoga Publications</Publisher>
     <Cost currency="INR">11.50</Cost>
  </Book>
```

```
  <Book>
    <Title>The First and Last Freedom</Title>
    <Author>J. Krishnamurti</Author>
    <Date>1954</Date>
    <ISBN>0-06-064831-7</ISBN>
    <Publisher>Harper &amp; Row</Publisher>
    <Cost currency="USD">2.95</Cost>
  </Book>
</BookCatalogue>
```

CursorParse.java

Located in the `<JWSDP_HOME>/sjsxp/samples/cursor` directory, `CursorParse.java` demonstrates using the StAX cursor API to read an XML document.

Stepping Through Events

In this example, the client application pulls the next event in the XML stream by calling the `next()` method on the parser; for example:

```
try
  {
    for(int i =0 ; i< count ; i++)
      {
        //pass the file name.. allrelativeentity
        //references will be resolved againstthis as
        //base URI.
        XMLStreamReader xmlr=
xmlif.createXMLStreamReader(filename, new
FileInputStream(filename));
        //when XMLStreamReader is created, it is positioned
at START_DOCUMENT event.
        int eventType = xmlr.getEventType();
        //printEventType(eventType);
        printStartDocument(xmlr);
        //check if there aremore eventsinthe input stream
        while(xmlr.hasNext())
          {
            eventType =xmlr.next();
            //printEventType(eventType);
            //these functionsprints the information about
theparticular event by calling relevant function
            printStartElement(xmlr);
            printEndElement(xmlr);
```

```
                printText(xmlr);
                printPIData(xmlr);
                printComment(xmlr);
            }
        }
```

Note that `next()` just returns an integer constant corresponding to the event underlying the current cursor location. The application calls the relevant function to get more information related to the underlying event. There are various accessor methods which can be called when the cursor is at particular event.

Returning String Representations

Because the `next()` method only returns integers corresponding to underlying event types, you typically need to map these integers to string representations of the events; for example:

```
public final staticString getEventTypeString(inteventType)
{
  switch(eventType)
    {
         case XMLEvent.START_ELEMENT:
           return "START_ELEMENT";
         case XMLEvent.END_ELEMENT:
           return "END_ELEMENT";
         case XMLEvent.PROCESSING_INSTRUCTION:
           return "PROCESSING_INSTRUCTION";
         case XMLEvent.CHARACTERS:
           return "CHARACTERS";
         case XMLEvent.COMMENT:
           return "COMMENT";
         case XMLEvent.START_DOCUMENT:
           return "START_DOCUMENT";
         case XMLEvent.END_DOCUMENT:
           return "END_DOCUMENT";
         case XMLEvent.ENTITY_REFERENCE:
           return "ENTITY_REFERENCE";
         case XMLEvent.ATTRIBUTE:
           return "ATTRIBUTE";
         case XMLEvent.DTD:
           return "DTD";
         case XMLEvent.CDATA:
           return "CDATA";
         case XMLEvent.SPACE:
                return "SPACE";
        }
      return"UNKNOWN_EVENT_TYPE , "+ eventType;
   }
```

Running the Sample

When you run the CursorParse sample, the class is compiled, and the XML stream is parsed and returned to STDOUT.

CursorApproachEventObject.java

Located in the <JWSDP_HOME>/sjsxp/samples/cursor2event directory, CursorApproachEventObject.java demonstrates how to get information returned by an XMLEvent object even when using the cursor API.

The idea here is that the cursor API's XMLStreamReader returns integer constants corresponding to particular events, where as the event iterator API's XMLEventReader returns immutable and persistent event objects. XMLStreamReader is more efficient, but XMLEventReader is easier to use, as all the information related to a particular event is encapsulated in a returned XMLEvent object. However, the disadvantage of event approach is the extra overhead of creating objects for every event, which consumes both time and memory.

With this mind, XMLEventAllocator can be used to get event information as an XMLEvent object, even when using the cursor API.

Instantiating an XMLEventAllocator

The first step is to create a new XMLInputFactory and instantiate an XMLEventAllocator:

```
XMLInputFactory xmlif = XMLInputFactory.newInstance();
System.out.println("FACTORY: " + xmlif);
xmlif.setEventAllocator(new XMLEventAllocatorImpl());
allocator = xmlif.getEventAllocator();
XMLStreamReader xmlr = xmlif.createXMLStreamReader(filename,
new FileInputStream(filename));
```

Creating an Event Iterator

The next step is to create an event iterator:

```
int eventType = xmlr.getEventType();
while(xmlr.hasNext()){
  eventType = xmlr.next();
  //Get all "Book" elements as XMLEvent object
  if(eventType == XMLStreamConstants.START_ELEMENT &&
xmlr.getLocalName().equals("Book")){
     //get immutable XMLEvent
     StartElement event = getXMLEvent(xmlr).asStartElement();
     System.out.println("EVENT: " + event.toString());
  }
}
```

Creating the Allocator Method

The final step is to create the `XMLEventAllocator` method:

```
private static XMLEvent getXMLEvent(XMLStreamReader reader)
throws XMLStreamException{
  return allocator.allocate(reader);
}
```

Running the Sample

When you run the `CursorApproachEventObject` sample, the class is compiled, and the XML stream is parsed and returned to `STDOUT`. Note how the `Book` events are returned as strings.

EventParse.java

Located in the `<JWSDP_HOME>/sjsxp/samples/event` directory, `EventParse.java` demonstrates how to use the StAX cursor API to read an XML document.

Creating an Input Factory

The first step is to create a new instance of `XMLInputFactory`:

```
XMLInputFactory factory = XMLInputFactory.newInstance();
System.out.println("FACTORY: " + factory);
```

Creating an Event Reader

The next step is to create an instance of XMLEventReader:

```
XMLEventReader r = factory.createXMLEventReader(filename, new
FileInputStream(filename));
```

Creating an Event Iterator

The third step is to create an event iterator:

```
XMLEventReader r = factory.createXMLEventReader(filename, new
FileInputStream(filename));
while(r.hasNext()) {
  XMLEvent e = r.nextEvent();
  System.out.println(e.toString());
}
```

Getting the Event Stream

The final step is to get the underlying event stream:

```
public final static String getEventTypeString(int eventType)
{
  switch (eventType)
     {
        case XMLEvent.START_ELEMENT:
           return "START_ELEMENT";
        case XMLEvent.END_ELEMENT:
           return "END_ELEMENT";
        case XMLEvent.PROCESSING_INSTRUCTION:
           return "PROCESSING_INSTRUCTION";
        case XMLEvent.CHARACTERS:
           return "CHARACTERS";
        case XMLEvent.COMMENT:
           return "COMMENT";
        case XMLEvent.START_DOCUMENT:
           return "START_DOCUMENT";
        case XMLEvent.END_DOCUMENT:
           return "END_DOCUMENT";
        case XMLEvent.ENTITY_REFERENCE:
           return "ENTITY_REFERENCE";
        case XMLEvent.ATTRIBUTE:
           return "ATTRIBUTE";
        case XMLEvent.DTD:
           return "DTD";
        case XMLEvent.CDATA:
           return "CDATA";
        case XMLEvent.SPACE:
           return "SPACE";
     }
  return "UNKNOWN_EVENT_TYPE " + "," + eventType;
}
```

Running the Sample

When you run the `EventParse` sample, the class is compiled, and the XML stream is parsed as events and returned to `STDOUT`. For example, an instance of the `Author` element is returned as:

```
<['http://www.publishing.org']::Author>
Dhirendra Brahmachari
</['http://www.publishing.org']::Author>
```

Note in this example that the event comprises an opening and closing tag, both of which include the namespace. The content of the element is returned as a string within the tags.

Similarly, an instance of the `Cost` element is returned as:

```
<['http://www.publishing.org']::Cost currency='INR'>
11.50
</['http://www.publishing.org']::Cost>
```

In this case, the `currency` attribute and value are returned in the opening tag for the event.

See earlier in this chapter, in the “Iterator API” and “Reading XML Streams” sections, for a more detailed discussion of StAX event parsing.

CursorWriter.java

Located in the `<JWSDP_HOME>/sjsxp/samples/writer` directory, `CursorWriter.java` demonstrates how to use the StAX cursor API to write an XML stream.

Creating the Output Factory

The first step is to create an instance of `XMLOutputFactory`:

```
XMLOutputFactory xof =  XMLOutputFactory.newInstance();
```

Creating a Stream Writer

The next step is to create an instance of `XMLStreamWriter`:

```
XMLStreamWriter xtw = null;
```

Writing the Stream

The final step is to write the XML stream. Note that the stream is flushed and closed after the final `EndDocument` is written:

```
xtw = xof.createXMLStreamWriter(new FileWriter(fileName));
xtw.writeComment("all elements here are explicitly in the HTML
namespace");
xtw.writeStartDocument("utf-8","1.0");
xtw.setPrefix("html", "http://www.w3.org/TR/REC-html40");
xtw.writeStartElement("http://www.w3.org/TR/REC-
html40","html");
xtw.writeNamespace("html", "http://www.w3.org/TR/REC-html40");
xtw.writeStartElement("http://www.w3.org/TR/REC-
html40","head");
xtw.writeStartElement("http://www.w3.org/TR/REC-
html40","title");
xtw.writeCharacters("Frobnostication");
xtw.writeEndElement();
xtw.writeEndElement();
xtw.writeStartElement("http://www.w3.org/TR/REC-
html40","body");
xtw.writeStartElement("http://www.w3.org/TR/REC-html40","p");
xtw.writeCharacters("Moved to");
xtw.writeStartElement("http://www.w3.org/TR/REC-html40","a");
```

```
    xtw.writeAttribute("href","http://frob.com");
    xtw.writeCharacters("here");
    xtw.writeEndElement();
    xtw.writeEndElement();
    xtw.writeEndElement();
    xtw.writeEndElement();
    xtw.writeEndDocument();
    xtw.flush();
    xtw.close();
```

Running the Sample

When you run the `CursorWriter` sample, the class is compiled, and the XML stream is parsed as events and written to a file named `CursorWriter-Output`:

```
    <!--all elements here are explicitly in the HTML namespace-->
    <?xml version="1.0" encoding="utf-8"?>
    <html:html xmlns:html="http://www.w3.org/TR/REC-html40">
    <html:head>
    <html:title>Frobnostication</html:title></html:head>
    <html:body>
    <html:p>Moved to
    <html:a href="http://frob.com">here</html:a>
    </html:p>
    </html:body>
    </html:html>
```

Note that in the actual `CursorWriter-Output` file, this stream is written without any linebreaks; the breaks have been added here to make the listing easier to read. In this example, as with the object stream in the `EventParse.java` sample, the namespace prefix is added to both the opening and closing HTML tags. This is not required by the StAX specification, but it is good practice when the final scope of the output stream is not definitively known.

MyStreamFilter.java

Located in the `<JWSDP_HOME>/sjsxp/samples/filter` directory, `MyStream-Filter.java` demonstrates how to use the StAX stream filter API to filter out events not needed by your application. In this example, the parser filters out all events except `StartElement` and `EndElement`.

Implementing the StreamFilter Class

The MyStreamFilter implements `javax.xml.stream.StreamFilter`:

```
public class MyStreamFilter implements
javax.xml.stream.StreamFilter{
```

Creating an Input Factory

The next step is to create an instance of XMLInputFactory. In this case, various properties are also set on the factory:

```
XMLInputFactory xmlif = null ;
try{
xmlif = XMLInputFactory.newInstance();
xmlif.setProperty(XMLInputFactory.IS_REPLACING_ENTITY_REFERENC
ES,Boolean.TRUE);
xmlif.setProperty(XMLInputFactory.IS_SUPPORTING_EXTERNAL_ENTIT
IES,Boolean.FALSE);
xmlif.setProperty(XMLInputFactory.IS_NAMESPACE_AWARE ,
Boolean.TRUE);
xmlif.setProperty(XMLInputFactory.IS_COALESCING ,
Boolean.TRUE);
}catch(Exception ex){
  ex.printStackTrace();
}
System.out.println("FACTORY: " + xmlif);
System.out.println("filename = "+ filename);
```

Creating the Filter

The next step is to instantiate a file input stream and create the stream filter:

```
FileInputStream fis = new FileInputStream(filename);

XMLStreamReader xmlr =
xmlif.createFilteredReader(xmlif.createXMLStreamReader(fis),
new MyStreamFilter());

int eventType = xmlr.getEventType();
printEventType(eventType);
while(xmlr.hasNext()){
  eventType = xmlr.next();
  printEventType(eventType);
  printName(xmlr,eventType);
```

```
    printText(xmlr);
    if(xmlr.isStartElement()){
      printAttributes(xmlr);
    }
    printPIData(xmlr);
    System.out.println("-----------------------------");
  }
```

Capturing the Event Stream

The next step is to capture the event stream. This is done in basically the same way as in the `EventParse.java` sample.

Filtering the Stream

The final step is the filter the stream:

```
  public boolean accept(XMLStreamReader reader) {
    if(!reader.isStartElement() && !reader.isEndElement())
      return false;
    else
      return true;
  }
```

Running the Sample

When you run the `MyStreamFilter` sample, the class is compiled, and the XML stream is parsed as events and returned to `STDOUT`. For example an `Author` event is returned as follows:

```
EVENT TYPE(1):START_ELEMENT
HAS NAME: Author
HAS NO TEXT
HAS NO ATTRIBUTES
-----------------------------
EVENT TYPE(2):END_ELEMENT
HAS NAME: Author
HAS NO TEXT
-----------------------------
```

Similarly, a `Cost` event is returned as follows:

```
EVENT TYPE(1):START_ELEMENT
HAS NAME: Cost
HAS NO TEXT

HAS ATTRIBUTES:
ATTRIBUTE-PREFIX:
ATTRIBUTE-NAMESP: null
ATTRIBUTE-NAME:   currency
ATTRIBUTE-VALUE:  INR
ATTRIBUTE-TYPE:  CDATA

-----------------------------
EVENT TYPE(2):END_ELEMENT
HAS NAME: Cost
HAS NO TEXT
-----------------------------
```

See earlier in this chapter, in the “Iterator API” and “Reading XML Streams” sections, for a more detailed discussion of StAX event parsing.

EventProducerConsumer.java

Located in the `<JWSDP_HOME>/sjsxp/samples/`readnwrite directory, `EventProducerConsumer.java` demonstrates how to use a StAX parser simultaneously as both a producer and a consumer.

The StAX `XMLEventWriter` API extends from the `XMLEventConsumer` interface, and is referred to as an *event consumer*. By contrast, `XMLEventReader` is an *event producer*. StAX supports simultaneous reading and writing, such that it is possible to read from one XML stream sequentially and simultaneously write to another stream.

This sample shows how the StAX producer/consumer mechanism can be used to read and write simultaneously. This sample also shows how a stream can be modified, and new events can be added dynamically and then written to different stream.

Creating an Event Producer/Consumer

The first step is to instantiate an event factory and then create an instance of an event producer/consumer:

```
XMLEventFactory m_eventFactory=XMLEventFactory.newInstance();
public EventProducerConsumer() {
}
.
.
.
try{
  EventProducerConsumer ms = new EventProducerConsumer();

  XMLEventReader reader =
XMLInputFactory.newInstance().createXMLEventReader(new
java.io.FileInputStream(args[0]));
  XMLEventWriter writer =
XMLOutputFactory.newInstance().createXMLEventWriter(System.out
);
```

Creating an Iterator

The next step is to create an iterator to parse the stream:

```
while(reader.hasNext())
  {
     XMLEvent event = (XMLEvent)reader.next();
     if(event.getEventType() == event.CHARACTERS)
        {

writer.add(ms.getNewCharactersEvent(event.asCharacters()));
        }
     else
        {
           writer.add(event);
        }
  }
writer.flush();
```

Creating a Writer

The final step is to create a stream writer in the form of a new `Character` event:

```
Characters getNewCharactersEvent(Characters event){
  if(event.getData().equalsIgnoreCase("Name1")){
     return
m_eventFactory.createCharacters(Calendar.getInstance().getTime
().toString());

  }
  //else return the same event
  else return event;
}
```

Running the Sample

When you run the `EventProducerConsumer` sample, the class is compiled, and the XML stream is parsed as events and written back to `STDOUT`:

```
<?xml version="1.0" encoding="UTF-8"?>
<BookCatalogue xmlns="http://www.publishing.org">
  <Book>
     <Title>Yogasana Vijnana: the Science of Yoga</Title>
     <Author>Dhirendra Brahmachari</Author>
     <Date>1966</Date>
     <ISBN>81-40-34319-4</ISBN>
     <Publisher>Dhirendra Yoga Publications</Publisher>
     <Cost currency="INR">11.50</Cost>
  </Book>

  <Book>
     <Title>The First and Last Freedom</Title>
     <Author>J. Krishnamurti</Author>
     <Date>1954</Date>
     <ISBN>0-06-064831-7</ISBN>
     <Publisher>Harper &amp; Row</Publisher>
     <Cost currency="USD">2.95</Cost>
  </Book>
</BookCatalogue>
```

Further Information

For more information about StAX, see:

- Java Community Process page:
  `http://jcp.org/en/jsr/detail?id=173`.
- W3C Recommendation “Extensible Markup Language (XML) 1.0”:
  `http://www.w3.org/TR/REC-xml`
- XML Information Set:
  `http://www.w3.org/TR/xml-infoset/`
- JAXB specification:
  `http://java.sun.com/xml/jaxb`
- JAX-RPC specification:
  `http//java.sun.com/xml/jaxrpc`
- W3C Recommendation “Document Object Model”:
  `http://www.w3.org/DOM/`
- SAX “Simple API for XML”:
  `http://www.saxproject.org/`
- DOM “Document Object Model”:
  `http://www.w3.org/TR/2002/WD-DOM-Level-3-Core-20020409/core.html#ID-B63ED1A3`
- W3C Recommendation “Namespaces in XML”:
  `http://www.w3.org/TR/REC-xml-names/`

For some useful articles about working with StAX, see:

- Jeff Ryan, “Does StAX Belong in Your XML Toolbox?”:
  `http://www.developer.com/xml/article.php/3397691`
- Elliotte Rusty Harold, “An Introduction to StAX”:
  `http://www.xml.com/pub/a/2003/09/17/stax.html`
- “More efficient XML parsing with the Streaming API for XML”:
  `http://www-106.ibm.com/developerworks/xml/library/x-tipstx/`

4

Introduction to XML and Web Services Security

THIS addendum discusses using XML and Web Services Security (XWS-Security) for *message-level security*. In message-level security, security information is contained within the SOAP message and/or SOAP message attachment, which allows security information to travel along with the message or attachment. For example, a portion of the message may be signed by a sender and encrypted for a particular receiver. When the message is sent from the initial sender, it may pass through intermediate nodes before reaching its intended receiver. In this scenario, the encrypted portions continue to be opaque to any intermediate nodes and can only be decrypted by the intended receiver. For this reason, message-level security is also sometimes referred to as *end-to-end security*.

Overview

This release includes the following XWS-Security features:

- Support for securing JAX-RPC applications at the service, port, and operation levels.
- XWS-Security APIs for securing both JAX-RPC applications and stand-alone applications that make use of SAAJ APIs only for their SOAP messaging.

Note: The XWS-Security EA 2.0 APIs are intended to insulate XWS-Security users from possible changes in the internal APIs, however, these APIs are subject to minor changes between 2.0 EA and 2.0 FCS.

- A sample security framework within which a JAX-RPC application developer will be able to secure applications by signing, verifying, encrypting, and/or decrypting parts of SOAP messages and attachments.

  The message sender can also make claims about the security properties by associating security tokens with the message. An example of a security claim is the identity of the sender, identified by a user name and password.
- Support for SAML Tokens and the WSS SAML Token Profile (partial).
- Support for securing attachments based on the WSS SwA Profile Draft.
- Partial support for sending and receiving WS-I Basic Security Profile (BSP) 1.0 compliant messages. For more information about BSP, read Interoperability with Other Web Services.
- Enhancements to the `SecurityConfiguration` Schema from the previous release.
- Sample programs that demonstrate using the framework.
- Command-line tools that provide specialized utilities for keystore management, including `pkcs12import` and `keyexport`.

The XWS-Security release contents are arranged in the structure shown in Table 4–1 within the Java WSDP release:

Table 4–1 XWS-Security directory structure

Directory Name	**Contents**
`<`*`JWSDP_HOME`*`>/xws-security/etc/`	Keystore files, property files, configuration files used for the examples.
`<`*`JWSDP_HOME`*`>/xws-security/docs/`	Release documentation for the XWS-Security framework. For the latest updates to this documentation, visit the web site at http://java.sun.com/webservices/docs/1.6/xws-security/index.html.
`<`*`JWSDP_HOME`*`>/xws-security/docs/api`	API documentation for the XWS-Security framework.
`<`*`JWSDP_HOME`*`>/xws-security/lib/`	JAR files containing the XWS-Security framework implementation and dependent libraries.
`<`*`JWSDP_HOME`*`>/xws-security/samples/`	Example code. This release includes sample applications. For more information on the samples, read Are There Any Sample Applications Demonstrating XWS-Security?
`<`*`JWSDP_HOME`*`>/xws-security/bin/`	Command-line tools that provide specialized utilities for keystore management. For more information on these, read Useful XWS-Security Command-Line Tools.

This implementation of XWS-Security is based on the Oasis Web Services Security (WSS) specification, which can be viewed at the following URL:

http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf

Some of the material in this chapter assumes that you understand basic security concepts. To learn more about these concepts, we recommend that you explore the following resources before you begin this chapter.

- The Java 2 Standard Edition discussion of security, which can be viewed from

  http://java.sun.com/j2se/1.5.0/docs/guide/security/index.html
- The *J2EE 1.4 Tutorial* chapter titled *Security*, which can be viewed from

  http://java.sun.com/j2ee/1.4/docs/tutorial-update2/doc/index.html

Does XWS-Security Implement Any Specifications?

XWS-Security is an implementation of the Web Services Security (WSS) specification developed at OASIS. WSS defines a SOAP extension providing quality of protection through message integrity, message confidentiality, and message authentication. WSS mechanisms can be used to accommodate a wide variety of security models and encryption technologies.

The WSS specification defines an end to end security framework that provides support for intermediary security processing. Message integrity is provided by using XML Signature in conjunction with security tokens to ensure that messages are transmitted without modifications. Message confidentiality is granted by using XML Encryption in conjunction with security tokens to keep portions of SOAP messages confidential.

In this release, the XWS-Security framework provides the following options for securing JAX-RPC applications:

- XML Digital Signature (DSig)

  This implementation of XML and Web Services Security uses JSR-105 (XML Digital Signature APIs) for signing and verifying parts of a SOAP message or attachment. JSR-105 can be viewed at http://www.jcp.org/en/jsr/detail?id=105

  Samples containing code for signing and/or verifying parts of the SOAP message are included with this release in the directory `<JWSDP_HOME>/xws-security/samples/simple/`. Read Simple Security Configurations Sample Application for more information on these sample applications.

- XML Encryption (XML-Enc)

  This implementation of XML and Web Services Security uses Apache's XML-Enc implementation, which is based on the XML Encryption W3C standard. This standard can be viewed at `http://www.w3.org/TR/xmlenc-core/`.

  Samples containing code for encrypting and/or decrypting parts of the SOAP message are included with this release in the directory `<JWSDP_HOME>/xws-security/samples/simple/`. Read Simple Security Configurations Sample Application for more information on these sample applications.

- UsernameToken Verification

Username token verification specifies a process for sending `UserNameTokens` along with the message. Sending these tokens with a message binds the identity of the tokens (and any other claims occurring in the security token) to the message.

This implementation of XML and Web Services Security provides support for Username Token Profile, which is based on OASIS WSS Username Token Profile 1.0 (which can be read at `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0.pdf`) and X.509 Certificate Token Profile, which is based on OASIS WSS X.509 Certificate Token Profile 1.0 (which can be read at `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0.pdf`).

Samples containing code for sending user name and X.509 certificate tokens along with the SOAP message are included with this release in the directory `<`*`JWSDP_HOME`*`>/xws-security/samples/simple/`. Read Simple Security Configurations Sample Application for more information on these sample applications.

- XWS-Security Framework APIs

  This implementation of XML and Web Services Security provides APIs that can be used to secure stand-alone Web services applications as well as JAX-RPC applications. These new APIs can be used to secure an outbound `SOAPMessage` and verify the security in an inbound `SOAPMessage`.

  Because some of the Java standards for XWS-Security technologies are currently undergoing definition under the Java Community Process, the security solution that is provided in Java WSDP 1.6 is based on *non-standard* APIs, which are subject to change with new revisions of the technology.

  To insulate stand alone XWS-Security users from the possible changes in the internal APIs, this release includes a sample interface definition that abstracts out some of the internal implementation details.

  Samples containing code for using these APIs are included with this release in the directory `<`*`JWSDP_HOME`*`>/xws-security/samples/api-sample/`. Read XWS-Security APIs Sample Application for more information on this sample application.

On Which Technologies Is XWS-Security Based?

XWS-Security APIs are used for securing Web services based on JAX-RPC and on stand-alone applications based on SAAJ. This release of XWS-Security is based on standard XML Digital Signature and non-standard XML Encryption APIs, which are subject to change with new revisions of the technology. As standards are defined in the Web Services Security space, the non-standard APIs will be replaced with standards-based APIs.

JSR-105 (XML Digital Signature) APIs are included in this release of the Java WSDP. JSR 105 is a standard API (in progress, at Proposed Final Draft) for generating and validating XML Signatures as specified by the W3C recommendation. It is an API that should be used by Java applications and middleware that need to create and/or process XML Signatures. It is used by this release of Web Services Security and can be used by non-Web Services technologies, for example, documents stored or transferred in XML. Both JSR-105 and JSR-106 (XML Digital Encryption) APIs are core-XML security components.

XWS-Security does not use the JSR-106 APIs because, currently, the Java standards for XML Encryption are undergoing definition under the Java Community Process. This Java standard is JSR-106-XML Digital Encryption APIs, which you can read at `http://www.jcp.org/en/jsr/detail?id=106`.

XWS-Security uses the Apache libraries for XML-Encryption. In future releases, the goal of XWS-Security is to move toward using the JSR-106 APIs.

Table 4–2 shows how the various technologies are stacked upon one another:

Table 4–2 API/Implementation Stack Diagram

XWS-Security
JSR-105 XML Signature and W3C XML Encryption Specifications *(W3C spec. may be replaced with JSR-106 in a future release)*
Apache XML Security implementation.
J2SE Security (JCE/JCA APIs)

The *Apache XML Security* project is aimed at providing implementation of security standards for XML. Currently the focus is on the W3C standards. More information on Apache XML Security can be viewed at:

http://xml.apache.org/security/

Java security includes the *Java Cryptography Extension (JCE)* and the *Java Cryptography Architecture (JCA)*. JCE and JCA form the foundation for public key technologies in the Java platform. The JCA API specification can be viewed at http://java.sun.com/j2se/1.5.0/docs/guide/security/CryptoSpec.html. The JCE documentation can be viewed at http://java.sun.com/products/jce/reference/docs/index.html.

Interoperability with Other Web Services

One of the goals of XML and Web Services Security technology is to enable applications to be able to securely interoperate with clients and web service endpoints deployed on other Java application servers and other web services platforms.

To accomplish this interoperability, an open industry organization, Web Services-Interoperability (WS-I) Organization, was chartered to promote Web services interoperability across platforms, operating systems, and programming languages. WS-I is developing an interoperability profile, WS-I Basic Security Profile 1.0 (BSP), that deals with transport security, SOAP messaging security, and other Basic-Profile-oriented Web services security considerations. XWS-Security EA 2.0 provides partial support for BSP (complete support is planned for the FCS release of 2.0.)

What is Basic Security Profile (BSP)?

In terms of XWS-Security, Basic Security Profile (BSP) support means that BSP-compliant requests will be generated and BSP-compliant requests will be accepted.

BSP restrictions and rules are only applicable for those features explicitly supported by XWS-Security. For outgoing messages, BSP-compliant messages are created by default. The only instance where BSP-compliant messages are not created by default is in the case of exclusive canonicalization transform in signatures. For performance reasons, this transform is not added by default, but can be added explicitly to the list of transforms.

For incoming messages, you can set the `compliance` attribute to `bsp` if you want to check for compliance in messages received from other applications or implementations. Non-compliant incoming messages are flagged when this option is set.

What is the XWS-Security Framework?

The XWS-Security framework is used to secure JAX-RPC and stand-alone SAAJ applications. Use XWS-Security to secure SOAP messages (requests and responses) through signing some parts, or encrypting some parts, or sending username-password authentication info, or some combination of these. Some example applications that use the technology are discussed in Are There Any Sample Applications Demonstrating XWS-Security?.

Use the XWS-Security framework to secure JAX-RPC applications by using the `-security` option of the `wscompile` tool. When you create an `asant` (or `ant`) target for JAX-RPC clients and services, the `wscompile` utility generates stubs, ties, serializers, and WSDL files.

Note: For the 2.0 release of JAX-RPC, JAX-RPC will be renamed to JAX-WS. JAX-WS will be part of the XWS-Security 2.0 FCS later this year. When this renaming occurs, the `wscompile` tool will be replaced, and these steps and the `build.xml` files for the sample applications will need to be modified accordingly.

XWS-Security has been integrated into JAX-RPC through the use of security configuration files. The code for performing the security operations on the client and server is generated by supplying the security configuration files to the JAX-RPC `wscompile` tool. The `wscompile` tool is instructed to generate security code via the `-security` option which specifies the security configuration file. See Configuring Security Configuration Files for more information on creating and using security configuration files.

To use the XWS-Security framework, set up the client and server-side infrastructure. A critical component of setting up your system for XWS-Security is to set up the appropriate database for the type of security (DSig, XML-Enc, UserName Token) to be used. Depending on the structure of your application, these databases could be any combination of keystore files, truststore files, and username-password files.

Configuring Security Configuration Files

XWS-Security makes it simple to specify client and server-side configurations describing security settings using security configuration files. In this tutorial, build, package, and deploy targets are defined and run using the `asant` tool. The `asant` tool is version of the Apache Ant Java-based build tool used specifically with the Sun Java System Application Server (Application Server). If you are deploying to a different container, you can use the Apache Ant tool instead.

To configure a security configuration file, follow these steps:

1. Create a security configuration file. Creating security configuration files is discussed in more detail in Understanding Security Configuration Files. Sample security configuration files are located in the directory `<JWSDP_HOME>/xws-security/samples/simple/config/`.
2. Create an `asant` (or `ant`) target in the `build.xml` file for your application that passes in and uses the security configuration file(s). This step is discussed in more detail in How Do I Specify the Security Configuration for the Build Files?
3. Create a property in the `build.properties` file to specify a security configuration file to be used on the client side and a security configuration file to be used on the server side. This step is discussed in more detail in How Do I Specify the Security Configuration for the Build Files?

Understanding Security Configuration Files

Security configuration files are written in XML. The elements within the XML file that specify the security mechanism(s) to use for an application are enclosed within `<xwss:SecurityConfiguration></xwss:SecurityConfiguration>` tags. The complete set of child elements along with the attributes that can be placed within these elements are described informally in XWS-Security Configuration File Schema. The formal schema definition (XSD) for XWS-Security Configuration can be viewed in the appendix A XWS-Security Formal Schema Definition. Many example security configuration files, along with descriptions each, are described in Simple Sample Security Configuration Files. This section describes a few of these options.

If you are using XWS-Security under JAX-RPC, the first set of elements of the security configuration file contain the declaration that this file is a security configuration file. The elements that provide this declaration look like this:

```
<xwss:JAXRPCSecurity xmlns:xwss="http://java.sun.com/xml/ns/
xwss/config">
   <xwss:Service>
      <xwss:SecurityConfiguration>
```

Note: If you are using XWS-Security in a stand-alone SAAJ environment, the root element of the security configuration file is `<xwss:SecurityConfiguration>`. An example application that uses XWS-Security in a stand-alone SAAJ environment is described in XWS-Security APIs Sample Application.

Within these declaration elements are elements that specify which type of security mechanism is to be applied to the SOAP message. For example, to apply XML Digital Signature, the security configuration file would include an `xwss:Sign` element, along with a keystore alias that identifies the private key/certificate associated with the sender's signature. A simple client security configuration file that requires digital signatures would look like this:

```
<xwss:JAXRPCSecurity xmlns:xwss="http://java.sun.com/xml/ns/
xwss/config">

    <xwss:Service>
        <xwss:SecurityConfiguration dumpMessages="true">
            <!--
             Note that in the <Sign> operation, a Timestamp is
exported
              in the security header and signed by default.
            -->
            <xwss:Sign>
               <xwss:X509Token certificateAlias="xws-security-
client"/>
            </xwss:Sign>
            <!--
              Signature requirement. No target is specified,
hence the
              soap body is expected to be signed. Also, by
default, a
              Timestamp is expected to be signed.
            -->
            <xwss:RequireSignature/>
        </xwss:SecurityConfiguration>
    </xwss:Service>

    <xwss:SecurityEnvironmentHandler>
        com.sun.xml.wss.sample.SecurityEnvironmentHandler
    </xwss:SecurityEnvironmentHandler>

</xwss:JAXRPCSecurity>
```

The xwss elements can be listed sequentially so that more than one security mechanism can be applied to the SOAP message. For example, for a client to first sign a message and then encrypt it, create an xwss element with the value Sign (to do the signing first), and then create an xwss element with the value of Encrypt (to encrypt after the signing). Building on the previous example, to add encryption to the message after the message has been signed, the security configuration file would be written like this example:

```
<xwss:JAXRPCSecurity xmlns:xwss="http://java.sun.com/xml/ns/
xwss/config">

    <xwss:Service>
        <xwss:SecurityConfiguration dumpMessages="true">
            <xwss:Sign/>
            <xwss:Encrypt>
                <xwss:X509Token certificateAlias="s1as"
keyReferenceType="Identifier"/>
            </xwss:Encrypt>
            <!--
              Requirements on messages received:
            -->
            <xwss:RequireEncryption/>
            <xwss:RequireSignature/>
        </xwss:SecurityConfiguration>
    </xwss:Service>

    <xwss:SecurityEnvironmentHandler>
        com.sun.xml.wss.sample.SecurityEnvironmentHandler
    </xwss:SecurityEnvironmentHandler>

</xwss:JAXRPCSecurity>
```

The xwss:RequireSignature element present in the two examples shown is used by the client to indicate that it expects the Response to be a signed response. Similarly the xwss:RequireEncryption element in a client configuration file indicates that the client expects an encrypted response. In the second example, a RequireEncryption and a RequireSignature element specified in that order implies that the client expects the response to be signed and then encrypted.

The xwss:RequireSignature and xwss:RequireEncryption elements appearing in a server configuration file similarly indicate that the server expects the request to be signed and encrypted respectively. The normal behavior of a client or server when it specifies a requirement of the form xwss:RequireSignature or xwss:RequireEncryption is to throw an exception if the requirement is not met by the received response or request.

The xwss:SecurityEnvironmentHandler element appearing under xwss:SecurityConfiguration is a compulsory child element that needs to be specified. The value of this element is the name of a Java class that implements the javax.security.auth.callback.CallbackHandler interface and handles a set of Callbacks defined by XWS-Security. There are a set of callbacks that are mandatory and that every CallbackHandler needs to implement. A few callbacks are optional and can be used to supply some finer-grained information to the XWS-Security run-time. The SecurityEnvironmentHandler and the Callbacks are described in Writing SecurityEnvironmentHandlers. The SecurityEnvironmentHandler is essentially a CallbackHandler which is used by the XWS-Security run-time to obtain the private-keys, certificates, symmetric keys, etc. to be used in the signing and encryption operations from the application. For more information, refer to the API documentation for the com.sun.xml.wss.impl.callback package, which is located in the *<JWSDP_HOME>*/xws-security/docs/api directory, to find the list of mandatory and optional callbacks and the details of the Callback classes.

When XWS-Security is used in a stand-alone SAAJ environment, the developer can choose to implement the com.sun.xml.wss.SecurityEnvironment interface instead of a callback handler that handles XWS-Security callbacks. In this situation, an instance of the SecurityEnvironment implementation can be set into the ProcessingContext instance. For an example application that demonstrates this, refer to the XWS-Security APIs Sample Application. For more details on the SecurityEnvironment interface, refer to the javadocs at *<JWSDP_HOME>*/xws-security/docs/api/com/sun/xml/wss/SecurityEnvironment.html.

Another type of security mechanism that can be specified in the security configuration file is *user name authentication*. In the case of user name authentication, the user name and password of a client need to be authenticated against the user/password database of the server. The xwss element specifies that the security mechanism to use is UsernameToken. On the server-side, refer to the documentation for your server regarding how to set up a user/password database for the server, or read Setting Up To Use XWS-Security With the Sample Applications for a summary. A client-side security configuration file that specifies `UsernameToken` authentication would look like this:

```
<xwss:JAXRPCSecurity xmlns:xwss="http://java.sun.com/xml/ns/
xwss/config">

    <xwss:Service>
        <xwss:SecurityConfiguration dumpMessages="true">
            <!--
              Default: Digested password will be sent.
            -->
            <xwss:UsernameToken name="Ron" password="noR"/>
        </xwss:SecurityConfiguration>
    </xwss:Service>

    <xwss:SecurityEnvironmentHandler>
        com.sun.xml.wss.sample.SecurityEnvironmentHandler
    </xwss:SecurityEnvironmentHandler>

</xwss:JAXRPCSecurity>
```

The `simple` sample application includes a number of example security configuration files. The sample configuration files are located in the directory `<`*`JWSDP_HOME`*`>/xws-security/samples/simple/config/`. Further discussion of the example security configurations can be found in Simple Sample Security Configuration Files.

Other sample configuration files that are provided in this release include:

- Simple Sample Security Configuration Files
- JAAS Sample Security Configuration Files
- SwA Sample Configuration Files
- SAML Interop Sample Configuration Files
- Security Configuration Files for Enabling Dynamic Policy
- Security Configuration Files for Enabling Dynamic Response

XWS-Security Configuration File Schema

When creating a security configuration file, there is a hierarchy within which the XML elements must be listed. This section contains an abstract sketch of the schema for the data for security configuration files. The formal schema definition can be viewed at A XWS-Security Formal Schema Definition.

Figure 4–1 shows the XML schema. The tables in Semantics of Security Configuration File Elements provide more information on the elements contained within the schema. The following notations are used to describe the schema:

- | means OR
- & means AND
- * means zero or more of these elements allowed
- + means 1 required, more allowed
- ? means zero or one element allowed
- (*value*) means that this value is the default value for the element, so specifying this attribute is optional.

Figure 4–1 XWS-Security Abstract Configuration File Schema

```
<JAXRPCSecurity>
      +<Service/>
      <SecurityEnvironmentHandler/>
</JAXRPCSecurity>

 <Service  ?name=service_identifier
            ?id=unique_identifier
            ?conformance="bsp"
            ?useCache=("false") | "true">
      ?<SecurityConfiguration/>
      *<Port/>
      ?<SecurityEnvironmentHandler/>
</Service>

<SecurityConfiguration
            ?dumpMessages=("false")|"true"
            ?enableDynamicPolicy=("false")|"true">
      *SecurityConfigurationElements
</SecurityConfiguration>

*SecurityConfigurationElements =
      ?<Timestamp/> |
      ?<SAMLAssertion type="SV"/> |
      ?<RequireSAMLAssertion type="SV"/> |
      ?<UsernameToken/> |
      ?<RequireUsernameToken /> |
      ?<RequireTimestamp /> |
       ?<OptionalTargets /> |
      <Sign/> |
      <Encrypt/> |
      <RequireSignature/> |
      <RequireEncryption/>
```

```
<Port name="port-name" ?conformance="bsp">
      *<Operation ?name="op-name">
            *<SecurityConfiguration/>
      </Operation>
</Port>

<SecurityEnvironmentHandler>
      handler-classname
</SecurityEnvironmentHandler>

<Operation name="operation_name" >
      *<SecurityConfiguration/>
</Operation>

<Timestamp ?id=unique_policy_identifier
            ?timeout=("300")/>

<UsernameToken ?id=unique_policy_identifier
            ?name=user_name // User name and password can also
be
                     //obtained dynamically from the
                                //SecurityEnvironment
            ?password=password
            ?useNonce=("true")|"false"
            ?digestPassword=("true")|"false"/>

 <RequireUsernameToken
            ?id=unique_policy_identifier
            ?nonceRequired=("true")|"false"
            ?passwordDigestRequired=("true")|"false"
            ?maxClockSkew=("60")
            ?timestampFreshnessLimit=("300")
            ?maxNonceAge=("900")/>

<Encrypt
      ?id=unique_policy_identifier >
      ?Key-Bearing-Token
      ?<KeyEncryptionMethod
            algorithm=("http://www.w3.org/2001/04/xmlenc#rsa-
oaep-mgf1p")|
                        "http://www.w3.org/2001/04/xmlenc#kw-
tripledes"|
                        "http://www.w3.org/2001/04/xmlenc#kw-
aes128" |
                        "http://www.w3.org/2001/04/xmlenc#kw-
aes256" |
                        "http://www.w3.org/2001/04/xmlenc#rsa-
```

```
1_5" />
      ?<DataEncryptionMethod
            algorithm=("http://www.w3.org/2001/04/
xmlenc#aes128-cbc")|
                        "http://www.w3.org/2001/04/
xmlenc#tripledes-cbc"|
                        "http://www.w3.org/2001/04/
xmlenc#aes256-cbc" />
      *<Target/>  // of type Target or EncryptionTarget
</Encrypt>

<EncryptionTarget
            ?type=("qname")|"uri"|"xpath"
            ?contentOnly=("true")|"false"
            ?enforce=("true")|"false"
            value=an_appropriate_ target_identifier>
      *<Transform/>
</EncryptionTarget>

<RequireEncryption
            ?id=unique_policy_identifier />
      ?Key-Bearing-Token
      ?<KeyEncryptionMethod
            algorithm=("http://www.w3.org/2001/04/xmlenc#rsa-
oaep-mgf1p") |
                        "http://www.w3.org/2001/04/xmlenc#kw-
tripledes" |
                        "http://www.w3.org/2001/04/xmlenc#kw-
aes128" |
                        "http://www.w3.org/2001/04/xmlenc#kw-
aes256" |
                        "http://www.w3.org/2001/04/xmlenc#rsa-
1_5" />
      ?<DataEncryptionMethod
            algorithm=("http://www.w3.org/2001/04/
xmlenc#aes128-cbc") |
                        "http://www.w3.org/2001/04/
xmlenc#tripledes-cbc" |
                        "http://www.w3.org/2001/04/
xmlenc#aes128-cbc" |
                        "http://www.w3.org/2001/04/
xmlenc#aes256-cbc" />
      *<Target/>//of type Target and/or EncryptionTarget
</RequireEncryption>

Key-Bearing-Token=
      <X509Token/> |
      <SAMLAssertion type="HOK"/> |
      <SymmetricKey/>

<X509Token
      ?id=any_legal_id //Must be unique within the resulting
XML
      ?strId=legal_id
      ?certificateAlias=alias_SecurityEnvironment_understands
      ?keyReferenceType=("Direct")|"Identifier"|"IssuerSerialN
umber"
      ?encodingType=("http://docs.oasis-open.org/wss/2004/01/
                  oasis-200401-wss-soap-message-security-
1.0#Base64Binary")
      ?valueType>

<SAMLAssertion
      ?id=unique_policy_identifier
      ?authorityId=URI_of_Issuing_Authority}
      ?strId=unique_policy_identifier
      ?keyIdentifier=identifier_for_Attester_Key
      ?keyReferenceType=("Identifier")|"Embedded"
      type="HOK"|"SV"
</SAMLAssertion>

<RequireSAMLAssertion
      ?id=unique_policy_identifier
      ?authorityId=URI_of_Issuing_Authority>
      ?strId=unique_policy_identifier
      type="SV"
      ?keyReferenceType=("Identifier")|"Embedded"
</RequireSAMLAssertion>

<SymmetricKey keyAlias= alias/keyname_of_a_shared_key />

keyReferenceType=
             "Direct"|"Identifier"|"IssuerSerialNumber"|
"Embedded"

EncodingType=(#Base64Binary |
                         other-wss-defined-encoding-type

ValueType=token-profile-specific-value-types

<Sign ?id=unique_policy_identifier
      ?includeTimestamp=("true")|"false">
      ?Key-Bearing-Token
      ?<CanonicalizationMethod
                  algorithm="http://www.w3.org/2001/10/xml-
exc-c14n#" | others/>
```

```
      ?<SignatureMethod
                  algorithm=("http://www.w3.org/2000/09/
xmldsig#rsa-sha1") | others/>
      *<Target/> //of type Target or SignatureTarget
</Sign>

<SignatureTarget
            ?type=("qname")|"uri"|"xpath"
            ?enforce=("true")|"false"
            value=an_appropriate_target_identifier>
      ?<DigestMethod algorithm=("http://www.w3.org/2000/09/
xmldsig#sha1") | others/>
      *<Transform/>
</SignatureTarget>

<RequireSignature
            ?id=unique_policy_identifier
            ?requireTimestamp=("true")|"false">
      ?Key-Bearing-Token
      ?<CanonicalizationMethod
                        algorithm=("http://www.w3.org/2001/10/
xml-exc-c14n#") | others/>
      ?<SignatureMethod
                        algorithm=("http://www.w3.org/2000/09/
xmldsig#rsa-sha1") | others/>
      *<Target/> //of type Target and/or SignatureTarget
</RequireSignature>

<Transform algorithm=supported-algorithms>
      *<AlgorithmParameter name="name" value="value"/>
</Transform>

<RequireTimestamp
      ?id=unique_policy_id
      ?maxClockSkew=("60")
      ?timestampFreshnessLimit=("300")/>

<RequireUsernameToken
      ?id=unique_policy_id
      ?nonceRequired=("true")|"false"
      ?passwordDigestRequired=("true")|"false"
      ?maxClockSkew=("60")
      ?timestampFreshnessLimit=("300")
      ?maxNonceAge=("900") >
</RequireUsernameToken>

<OptionalTargets>
      *<Target>
```

```
</OptionalTargets>

<Target ?type=("qname")|"uri"|"xpath"
        ?contentOnly=("true")|"false"
        ?enforce=("true")|"false">
        value
</Target>
```

Semantics of Security Configuration File Elements

This section contains a discussion regarding the semantics of security configuration file elements.

JAXRPCSecurity

The `<JAXRPCSecurity>` element is the top-level XML element for XWS-Security configuration files for applications that use JAX-RPC. The top-level XML element for stand-alone SAAJ applications is `<SecurityConfiguration>`. Table 4–3 provides a description of the sub-elements of `<JAXRPCSecurity>`.

Table 4–3 Sub-elements of JAXRPCSecurity element

Sub-elements of **JAXRPCSecurity**	**Description**
Service	Indicates a JAX-RPC service within the XWS-Security environment for which XWS-Security can be configured. In this release, multiple services per configuration file are supported.
SecurityEnvironmentHandler	Specifies the implementation class name of the security environment handler (Required).

Service

The `<Service>` element indicates a JAX-RPC service within the XWS-Security environment for which XWS-Security can be configured.

Note: Although the XWS-Security configuration schema allows multiple `<Service>` elements to appear under a `<JAXRPCSecurity>` element, the current release does not support this feature. The configuration reader will throw an `IllegalStateException` if multiple services are specified.

Table 4–4 provides a description of its attributes, Table 4–5 provides a description of its sub-elements.

Table 4–4 Attributes of Service element

Attributes of **Service**	**Description**
`name`	The name of the JAX-RPC service (optional).
`id`	The id of the JAX-RPC service (optional).
`conformance`	Type of conformance. In this release, the choice for this attribute is restricted to `bsp` (optional).
`useCache`	Determines whether caching is enabled. Default is `false` (optional). This flag is unused in the current release and has been introduced for future enhancements.

Table 4–5 Sub-elements of Service element

Sub-elements of **Service**	**Description**
SecurityConfiguration	Indicates that what follows is the security configuration for the service.
Port	A port within a JAX-RPC service. Any (including zero) number of these elements may be specified.
SecurityEnvironmentHandler	Specifies the implementation class name of the security environment handler. If specified, overrides the `SecurityEnvironmentHandler` specified at the parent level. (Optional)

Port

The `<Port>` element represents a port within a JAX-RPC service. Table 4–6 provides a description of its attributes, Table 4–7 provides a description of its sub-elements.

Table 4–6 Attributes of Port element

Attributes of **Port**	**Description**
`name`	Name of the port as specified in the wsdl (Required).
`conformance`	Type of conformance. In this release, the choice for this attribute is restricted to `bsp`. In this release, XWS-Security is conformant to Basic Security Profile (BSP) for messages that are created and sent. When `conformance` is set to `bsp`, messages are checked for BSP compliance before being sent. For more information on BSP, read What is Basic Security Profile (BSP)? This EA implementation of this feature will be more complete in the FCS release (optional).

Table 4–7 Sub-elements of Port element

Sub-elements of **Port**	**Description**
SecurityConfiguration	Indicates that what follows is security configuration for the port. This over-rides any security configured for the service.
Operation	Indicates a port within a JAX-RPC service. Any (including zero) number of these elements may be specified.

Operation

The `<Operation>` element creates a security configuration at the operation level, which takes precedence over port and service-level security configurations. Table

4–8 provides a description of its attributes, Table 4–9 provides a description of its sub-elements.

Table 4–8 Attributes of Operation

Attributes of **Operation**	**Description**
`name`	Name of the operation as specified in the WSDL file, for example, `name="{http://xmlsoap.org/Ping}Ping0"`. (Required)

Table 4–9 Sub-elements of Operation

Sub-elements of **Operation**	**Description**
SecurityConfiguration	This element indicates that what follows is security configuration for the operation. This overrides any security configured for the port and the service.

SecurityConfiguration

The `<SecurityConfiguration>` element specifies a security configuration. Table 4–10 provides a description of its attributes, Table 4–11 provides a description of its sub-elements. The sub-elements of `SecurityConfiguration` can appear in any order. The order in which they appear determines the order in which they are executed, with the exception of the `OptionalTargets` element.

Table 4–10 Attributes of SecurityConfiguration

Attributes of **SecurityConfiguration**	**Description**
`dumpMessages`	If `dumpMessages` is set to `true`, all incoming and outgoing messages are printed at the standard output. The default value is `false` (Optional).

Table 4–10 Attributes of SecurityConfiguration (Continued)

Attributes of **SecurityConfiguration**	**Description**
`enableDynamicPolicy`	If `enableDynamicPolicy` is set to `true`, all incoming and outgoing messages use a dynamic security policy. The default value is `false` (Optional). For an example that uses this attribute, see Dynamic Policy Sample Application.

Table 4–11 Sub-elements of SecurityConfiguration

Sub-elements of **SecurityConfiguration**	**Description**
Timestamp	Indicates that a timestamp must be sent in the outgoing messages.
UsernameToken	Indicates that a username token must be sent in the outgoing messages.
Sign	Indicates that a sign operation needs to be performed on the outgoing messages.
Encrypt	Indicates that an encrypt operation needs to be performed on the outgoing messages.
SAMLAssertion	Indicates that a SAML assertion of subject confirmation type Sender-Vouches (SV) must be sent in the security header of the outgoing messages.
RequireTimestamp	Indicates that a timestamp must be present in the incoming messages.
RequireUsernameToken	Indicates that a username token must be present in the incoming messages.
RequireSignature	Indicates that the incoming messages must contain a signature.
RequireEncryption	Indicates that the incoming messages must be encrypted.
RequireSAMLAssertion	Indicates that the incoming message must contain a SAML assertion of subject confirmation type Sender-Vouches (SV).

Table 4–11 Sub-elements of SecurityConfiguration

Sub-elements of **SecurityConfiguration**	**Description**
OptionalTargets	Specifies a list of elements on which security operations are not required in the incoming messages, but are allowed.

Timestamp

The `<Timestamp>` element specifies that a timestamp must be sent in outgoing messages. For a discussion of using the `Timestamp` element with the `includeTimestamp` attribute of `Sign`, see Using Timestamp and includeTimestamp. Table 4–12 provides a description of its attributes.

Table 4–12 Attributes of Timestamp

Attributes of **Timestamp**	**Description**
`timeout`	Value in seconds after which the timestamp should be considered expired. Default value is `300`.

UsernameToken

The `<UsernameToken>` element is used when a `UsernameToken` should be sent with outgoing messages. This `UsernameToken` contains the sender's user and password information. Table 4–13 provides a description of its attributes.

Table 4–13 Attributes of UsernameToken

Attributes of **UsernameToken**	**Description**
`name`	The name of the user. If not specified, security environment handler must provide it at runtime.
`password`	The password of the user. If not specified, attempt would be made to obtain it from the security environment handler at runtime.

Table 4–13 Attributes of UsernameToken (Continued)

Attributes of **UsernameToken**	**Description**
`digestPassword`	Indicates whether to send password in digest form or not. Default value is `true`.
`useNonce`	Indicates whether to send a nonce inside the username token or not. Sending a nonce helps in preventing replay attacks. Default value is `true`.
`id`	The id to be set on the username token in the message to be sent. This is also useful in referring to the token from other places in the security configuration file.

Sign

The `<Sign>` element is used to indicate that a sign operation needs to be performed on the outgoing messages. Table 4–14 provides a description of its attributes, Table 4–15 provides a description of its sub-elements.

Table 4–14 Attributes of Sign

Attributes of **Sign**	**Description**
`id`	The id to be set on the signature of the message to be sent. This is also useful in referring to the signature from other places in the security configuration file.
`includeTimestamp`	Indicates whether to also sign a timestamp as part of this signature or not. This is a mechanism useful in preventing replay attacks. The default value is `true`. Note that a `true` value for this attribute makes sure that a timestamp will be sent in the outgoing messages even if the <Timestamp> element has not been specified. Also note that at most one timestamp is sent in a message. When `includeTimestamp` is `true`, a `Timestamp` element with the default value is added and is signed (i.e., `Timestamp` is added as one of the targets in the corresponding signature element.)

Table 4–15 Sub-elements of Sign

Sub-elements of **Sign**	**Description**
X509Token	Indicates the certificate corresponding to the private key used for signing. If this element is not present, attempt is made to get the default certificate from the security environment handler. Only one of the `X509Token`, `SAMLAssertion`, and `SymmetricKey` elements may be present at a time.
SAMLAssertion	Indicates the certificate corresponding to the SAML assertion used for signing. Only one of the `X509Token`, `SAMLAssertion`, and `SymmetricKey` elements may be present at a time.
SymmetricKey	Indicates the symmetric key corresponding to the private key used for signing. Only one of the `X509Token`, `SAMLAssertion`, and `SymmetricKey` elements may be present at a time. (`SymmetricKey` signatures are not supported for signatures in this release.)
CanonicalizationMethod	Indicates the canonicalization algorithm applied to the `<SignedInfo>` element prior to performing signature calculations.
SignatureMethod	Indicates the algorithm used for signature generation and validation.
Target	Specifies the target message part to be signed. `Target` has been deprecated and is included only for backward compatibility.
SignatureTarget	Specifies the target message part to be signed.

Using Timestamp and includeTimestamp

The following configurations of `Timestamp` and the `includeTimestamp` attribute of the `Sign` element have the following effect:

1. If a `<Timestamp>` element is configured, a timestamp will be sent in the message.
2. If the `includeTimestamp` attribute on `<Sign>` has value `true` and `<Timestamp>` is not configured, a timestamp (with default `timeout` value) will be sent in the message and included in the signature.
3. If the `includeTimestamp` attribute on `<Sign>` has value `true` and `<Timestamp>` is configured, a timestamp with the properties (e.g, `timeout`) spec-

ified on the `<Timestamp>` will be sent in the message and also be included in the signature.

4. If the `includeTimestamp` attribute on `<Sign>` has value `false`, a timestamp is not included in the signature.

Encrypt

The `<Encrypt>` element is used to indicate that an encrypt operation needs to be performed on the outgoing messages. Table 4–16 provides a description of its sub-elements.

Table 4–16 Sub-elements of Encrypt

Sub-elements of **Encrypt**	**Description**
X509Token	Indicates the certificate to be used for encryption. If this element is not present, attempt is made to get the default certificate from the security environment handler. This element must not be specified if the <SymmetricKey> or <SAMLAssertion> sub-element of <Encrypt> is specified.
SymmetricKey	Indicates the symmetric key to be used for encryption. This element must not be specified if the <X509Token> or <SAMLAssertion> sub-element of <Encrypt> is present.
SAMLAssertion	Indicates the SAML assertion to be used for encryption. This element must not be specified if the <X509Token> or <SymmetricKey> sub-element of <Encrypt> is present.
KeyEncryptionMethod	Specifies the public key encryption algorithm to be used for encrypting and decrypting keys.
DataEncryptionMethod	Specifies the encryption algorithm to be applied to the cipher data.
Target	Identifies the resource that needs to be encrypted. The `Target` element has been deprecated and is provided only for backward compatibility.
EncryptionTarget	Identifies the resource that needs to be encrypted.

SAMLAssertion

The `<SAMLAssertion>` element is used to define the SAML assertion to be transferred from identity providers to service providers. These assertions include statements that service providers use to make access control decisions. The SAML Sample Application provides some examples of using this element. Table 4–17 provides a description of attributes of the `<SAMLAssertion>` element.

Table 4–17 Attributes of SAMLAssertion

Attributes of **SAMLAssertion**	**Description**
`id`	Identifier for an assertion.
`authorityId`	Defines the ID that may be used to acquire the identified assertion at a SAML assertion authority or responder.
`strID`	Element content of the string identifier for the `keyIdentifier`.
`keyIdentifier`	The ID for a token reference for the key identifier that references a local SAML assertion.
`encodingType`	A parameter used to identify the security reference. When the `keyIdentifier` is used, this attribute is prohibited. (Prohibited)
`keyReferenceType`	Indicates whether the token reference identifies a token by URI (`Identifier`) or by an embedded reference (`Embedded`). The default value is `Identifier`.

Table 4–17 Attributes of SAMLAssertion (Continued)

Attributes of **SAMLAssertion**	**Description**
type	Indicates the type of SAML assertion to use. The choices are Holder-of-Key (HOK) and Sender-Vouches (SV). The SV confirmed assertion may not be contained in the message. The Security Token Reference (STR) identified in strID becomes a remote reference to the SV confirmed assertion. The HOK assertion contained in the message identifies the attesting entity and its signing key. Whether you choose type HOK or SV depends on where this token is located in the configuration file. A standalone <SAMLAssertion> element under <SecurityConfiguration> should be of type SV. An assertion of type HOK can appear as a child of a <Sign> or <Encrypt> element, indicating the presence of a confirmation key that can be used for the operation. (Required)

RequireTimestamp

If the <RequireTimestamp> element is present, a timestamp, in the form of a wsu:Timestamp element, must be present in the incoming messages. If the RequireTimestamp element is not specified, a Timestamp is not required. A timestamp specifies the particular point in time it marks. You may also want to consider using a nonce, which is a value that you should never receive more than once. Table 4–18 provides a description of its attributes.

Table 4–18 Attributes of RequireTimestamp

Attributes of **RequireTimestamp**	**Description**
id	The id assigned to the timestamp.
maxClockSkew	The maximum number of seconds the sending clock can deviate from the receiving clock. Default is 60.
timestampFreshness-Limit	The maximum number of seconds the time stamp remains valid. Default is 300.

RequireUsernameToken

The <RequireUsernameToken> element is used to specify that a username token must be present in the incoming messages. Table 4–19 provides a description of its attributes.

Table 4–19 Attributes of RequireUsernameToken

Attributes of **RequireUsernameToken**	**Description**
id	The identifier for the UsernameToken.
passwordDigestRequired	Indicates whether the username tokens in the incoming messages are required to contain the passwords in digest form or not. Default value is true. (See also: digestPassword attribute on <UsernameToken>)
nonceRequired	Indicates whether a nonce is required to be present in the username tokens in the incoming messages. Default value is true. (See also: useNonce attribute on <UsernameToken>)
maxClockSkew	The maximum number of seconds the sending clock can deviate from the receiving clock. Default is 60.
timestampFreshnessLimit	The maximum number of seconds the time stamp remains valid. Default is 300.
maxNonceAge	The maximum number of seconds the nonce is cached by the server for detecting a nonce replay. Default is 900.

RequireSignature

The <RequireSignature> element is specified when a digital signature is required for all specified targets. If no signature is present, an exception is thrown. In this release, the only sub-elements of RequireSignature that are verified while validating an incoming message are Target and SignatureTarget.

Table 4–20 provides a description of its attributes, Table 4–21 provides a description of its sub-elements.

Table 4–20 Attributes of RequireSignature

Attributes of **RequireSignature**	**Description**
`id`	The id to be set on the signature of the message to be sent. This is also useful in referring to the signature from other places in the security configuration file.
`requireTimestamp`	Indicates whether a timestamp must be included in the signatures in the incoming messages. Default value is `true`. (See also: `includeTimestamp` attribute on <Sign>)

Table 4–21 Sub-elements of RequireSignature

Sub-elements of **RequireSignature**	**Description**
X509Token	Indicates the certificate corresponding to the private key used for signing. If this element is not present, attempt is made to get the default certificate from the security environment handler. Only one of the `X509Token`, `SAMLAssertion`, and `SymmetricKey` elements may be present at a time.
SAMLAssertion	Indicates the certificate corresponding to the SAML assertion used for signing. Only one of the `X509Token`, `SAMLAssertion`, and `SymmetricKey` elements may be present at a time.
SymmetricKey	Indicates the symmetric key corresponding to the private key used for signing. Only one of the `X509Token`, `SAMLAssertion`, and `SymmetricKey` elements may be present at a time.
CanonicalizationMethod	Indicates the canonicalization algorithm applied to the <`SignedInfo`> element prior to performing signature calculations.
SignatureMethod	Indicates the algorithm used for signature generation and validation.

Table 4–21 Sub-elements of RequireSignature (Continued)

Sub-elements of **RequireSignature**	**Description**
Target	Specifies the target message part which was expected to be signed. Target has been deprecated and is only provided for backward compatibility.
SignatureTarget	Specifies the target message part which was expected to be signed.

RequireEncryption

The `<RequireEncryption>` element is used when encryption is required for all incoming messages. If encryption is not present, an exception is thrown. In this release, the only sub-elements of `RequireEncryption` that are verified during validation of encrypted data in incoming messages are `Target` and `Encryption-Target`. Table 4–22 provides a description of its attributes, Table 4–23 provides a description of its sub-elements.

Table 4–22 Attributes of RequireEncryption

Attributes of **RequireEncryption**	**Description**
`id`	The id to be set on the message to be sent.

Table 4–23 Sub-elements of RequireEncryption

Sub-elements of **RequireEncryption**	**Description**
X509Token	Indicates the certificate to be used for encryption. If this element is not present, attempt is made to get the default certificate from the security environment handler. Only one of the `X509Token`, `SAMLAssertion`, and `SymmetricKey` elements may be present at a time.

Table 4–23 Sub-elements of RequireEncryption (Continued)

Sub-elements of **RequireEncryption**	**Description**
SAMLAssertion	Indicates the certificate corresponding to the SAML assertion used for encryption. Only one of the `X509Token`, `SAMLAssertion`, and `SymmetricKey` elements may be present at a time.
SymmetricKey	Indicates the symmetric key corresponding to the private key used for encryption. Only one of the `X509Token`, `SAMLAssertion`, and `SymmetricKey` elements may be present at a time.
CanonicalizationMethod	Indicates the canonicalization algorithm applied to the `<Encrypt>` element prior to performing encrypt calculations.
DataEncryptionMethod	Indicates the encryption algorithm to be applied to the cipher data.
Target	Identifies the resource that was expected to be encrypted. Target has been deprecated and is only provided for backward compatibility.
EncryptionTarget	Identifies the resource that was expected to be encrypted.

RequireSAMLAssertion

The `<RequireSAMLAssertion>` element is used when a Sender-Vouches (SV) SAML assertion is required for all incoming messages. If a SAML assertion is not present, an exception is thrown. Table 4–24 provides a description of its attributes.

Table 4–24 Attributes of RequireSAMLAssertion

Attributes of **RequireSAMLAssertion**	**Description**
`id`	Identifier for an assertion. (Optional)
`authorityId`	Defines an abstract identifier for the assertion-issuing authority.

Table 4–24 Attributes of RequireSAMLAssertion (Continued)

Attributes of **RequireSAMLAssertion**	**Description**
strID	Element content of the string identifier for the keyIdentifier.
keyReferenceType	Indicates whether the token reference identifies a token by AssertionId (Identifier) or by an embedded reference (Embedded). The default value is Identifier.
type	Indicates to use the SV type of SAML assertion. The SV confirmed assertion is not contained in the message. (Required)

OptionalTargets

The <OptionalTargets> element is used when an operation is optional for a specific target. Table 4–25 provides a description of its sub-elements.

Table 4–25 Sub-elements of OptionalTargets

Sub-elements of **OptionalTargets**	**Description**
Target	Indicates that a security operation is allowed to be performed on this target, but it is not required. One or more of these elements can be specified. The augmented cid:* syntax is not allowed as the value of the Target when Target is a sub-element of OptionalTargets.

Transform

The <Transform> element is an optional ordered list of processing steps to be applied to the resource's content before it is digested. Transforms can include operations such as canonicalization, encoding/decoding, XSLT, XPath, XML schema validation, or XInclude. The recommendation that discusses this method is the W3C XML-Signature Syntax and Processing recommendation, which can be viewed at http://www.w3.org/TR/xmldsig-core/#sec-Transforms. The following types of transform algorithms can be used: canonicalization, Base64, xpath filtering, envelope signature transform, and XSLT transform. The XWS-Security APIs Sample Application provides some examples of configuration files that use this element.

Table 4–26 provides a description of its attributes, Table 4–27 provides a description of its sub-elements.

Table 4–26 Attributes of Transform

Attributes of **Transform**	**Description**
`algorithm`	The algorithm to be used for signing. (Required)

Table 4–27 Sub-elements of Transform

Sub-elements of **Transform**	**Description**
AlgorithmParameter	Identifies the parameters to be supplied to the transform algorithm.

AlgorithmParameter

Algorithms are identified by URIs that appear as an attribute to the element that identifies the algorithms' role (DigestMethod, Transform, SignatureMethod, or CanonicalizationMethod). All algorithms used herein take parameters but in many cases the parameters are implicit. Explicit additional parameters to an algorithm appear as content elements within the algorithm role element. Such parameter elements have a descriptive element name, which is frequently algorithm specific, and MUST be in the XML Signature namespace or an algorithm specific namespace. The XWS-Security APIs Sample Application provides some examples of configuration files that use this element.

Table 4–28 provides a description of its attributes.

Table 4–28 Attributes of AlgorithmParameter

Attributes of **AlgorithmParameter**	**Description**
`name`	The name of the algorithm parameter. (Required)

Table 4–28 Attributes of AlgorithmParameter (Continued)

Attributes of **AlgorithmParameter**	**Description**
`value`	The value of the algorithm parameter. (Required)

X509Token

The `<X509Token>` element is used to specify the certificate to be used for encryption (for the case of encryption) or the certificate corresponding to the private key used for signing (for the case of signature). This element must not be specified if the <SymmetricKey> or <SAMLAssertion> sub-elements are present. Table 4–29 provides a description of its attributes.

Table 4–29 Attributes of X509Token

Attributes of **X509Token**	**Description**
`id`	The id to be assigned to this token in the message. This attribute is useful in referring the token from other places in the security configuration file. (Optional)
`strID`	If specified, it denotes the `wsu:Id` to be assigned to the Security Token Reference (STR) to be generated and inserted into the message. The inserted STR would reference the X509 token.
`certificateAlias`	The alias associated with the token (certificate).
`keyReferenceType`	The reference mechanism to be used for referring to the X509 token (certificate) which was involved in the security operation, in the outgoing messages. The default value is `Direct`. The list of allowed values for this attribute and their description is as follows: 1. `Direct -` certificate is sent along with the message. 2. `Identifier -` subject key identifier extension value of the certificate is sent in the message. 3. `IssuerSerialNumber -` issuer name and serial number of the certificate are sent in the message.

Table 4–29 Attributes of X509Token (Continued)

Attributes of **X509Token**	**Description**
encodingType	The type of encoding to be used for the token. The default value is http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#Base64Binary.
valueType	The type of value to expect. The valueType can be #X509v3, #X509PKIPathv1, or #PKCS7. This release does not support #PKCS7.

Target

Note: In this release the Target sub-element is deprecated and is supported only for backward compatibility. The Target sub-element is being replaced with Signature-Target and EncryptionTarget.

The <Target>*target_value*</Target> sub-element contains a string that can be used to identify the resource that needs to be signed or encrypted. If a Target sub-element is not specified, the default value is a target that points to the contents of the SOAP body of the message. The value of this element is specified as a text node inside this element.

You can specify attachments as targets by setting the type attribute to uri and specifying the target value as cid:<*part-name*>, which specifies the value of the Content-ID (CID) header of the attachment. When the Content-ID is not know until runtime, such as when auto-generated CIDs are run under JAX-RPC, the attachment can be referenced by setting the type attribute to uri and specifying the target value as attachmentRef:<*part-name*>, where *part-name* is the WSDL part name of the AttachmentPart. Auto-generated CIDs in JAX-RPC following the form <*partname*>=<*UUID*>@<*Domain*>. The special value cid:* can be used to refer to all attachments of a SOAPMessage.

The attributes of the <Target> element are described in Table 4–30.

Table 4–30 Attributes of Target

Attributes of **Target**	**Description**
type	Indicates the type of the target value. Default value is qname. The list of allowed values for this attribute and their description is as follows: 1. qname - If the target element has a local name Name and a namespace URI some-uri, the target value is {some-uri}Name. 2. xpath - Indicates that the target value is the xpath of the target element. 3. uri - If the target element has an id some-id, then the target value is #some-id.
contentOnly	Indicates whether the complete element or only the contents needs to be encrypted (or is required to be encrypted). The default value is true. (Relevant only for <Encrypt> and <RequireEncryption> targets)
enforce	If true, indicates that the security operation on the target element is definitely required. Default value is true. (Relevant only for <RequireSignature> and <RequireEncryption> targets)

SignatureTarget

The <SignatureTarget> sub-element is called by the <SignatureMethod> element to identify the resource that needs to be signed. If neither the <SignatureTarget> nor <Target> sub-element are specified, the default value is a target that points to the contents of the SOAP body of the message. The target value is a string that specifies the object to be signed, and which is specified between the <SignatureTarget>*target_value*</SignatureTarget> elements. The XWS-Security APIs Sample Application provides some examples of configuration files that use this element.

You can specify attachments as targets by setting the type attribute to uri and specifying the target value as cid:<*part-name*>, which specifies the value of the Content-ID (CID) header of the attachment. When the Content-ID is not know until runtime, such as when auto-generated CIDs are run under JAX-RPC, the attachment can be referenced by setting the type attribute to uri and specifying the target value as `attachmentRef:<part-name>`, where *`part-name`* is the WSDL part name of the `AttachmentPart`. Auto-generated CIDs in JAX-RPC following the form *`<partname>=<UUID>@<Domain>`*. The special value `cid:*` can be used to refer to all attachments of a `SOAPMessage`.

The attributes of `<SignatureTarget>` are described in Table 4–31, its sub-elements are described in Table 4–32.

Table 4–31 Attributes of SignatureTarget

Attributes of **SignatureTarget**	**Description**
`type`	Indicates the type of the target value. Default value is `qname`. The list of allowed values for this attribute and their description is as follows: 1. `qname` - If the target element has a local name `Name` and a namespace URI `some-uri`, the target value is `{some-uri}Name`. 2. `xpath` - Indicates that the target value is the xpath of the target element. 3. `uri` - If the target element has an id `some-id`, then the target value is `#some-id`. This is the option that is used to secure message attachments.
`value`	Indicates whether the value needs to be encrypted (or is required to be encrypted). The default value is `true`. (Relevant only for <Encrypt> and <RequireEncryption> targets)
`enforce`	If `true`, indicates that the security operation on the target element is definitely required. Default value is `true`. (Relevant only for <RequireSignature> and <RequireEncryption> targets)

Table 4–32 Sub-elements of SignatureTarget

Sub-elements of **SignatureTarget**	**Description**
DigestMethod	Identifies the digest algorithm to be applied for signing the object.

Table 4–32 Sub-elements of SignatureTarget (Continued)

Sub-elements of **SignatureTarget**	**Description**
Transform	Identifies the transform algorithm to be applied before signing the object.

EncryptionTarget

The `<EncryptionTarget>` sub-element identifies the type of encrypted structure being described. If neither the `<EncryptionTarget>` nor `<Target>` sub-elements are specified, the default value is a target that points to the contents of the SOAP body of the message. The target value is a string that specifies the object to be encrypted, and which is specified between the `<EncryptionTarget>`*`target_value`*`</EncryptionTarget>` elements.

You can specify attachments as targets by setting the `type` attribute to `uri` and specifying the target value as `cid:<`*`part-name`*`>`, which specifies the value of the Content-ID (CID) header of the attachment. When the Content-ID is not know until runtime, such as when auto-generated CIDs are run under JAX-RPC, the attachment can be referenced by setting the `type` attribute to `uri` and specifying the target value as `attachmentRef:<`*`part-name`*`>`, where *`part-name`* is the WSDL part name of the `AttachmentPart`. Auto-generated CIDs in JAX-RPC following the form `<`*`partname`*`>=<`*`UUID`*`>@<`*`Domain`*`>`. The special value `cid:*` can be used to refer to all attachments of a `SOAPMessage`.

The attributes of `<EncryptionTarget>` are described in Table 4–33, its sub-elements are described in Table 4–34.

Table 4–33 Attributes of EncryptionTarget

Attributes of **EncryptionTarget**	**Description**
`type`	Indicates the type of the target value. Default value is `qname`. The list of allowed values for this attribute and their description is as follows: 1. `qname` - If the target element has a local name `Name` and a namespace URI `some-uri`, the target value is `{some-uri}Name`. 2. `xpath` - Indicates that the target value is the xpath of the target element. 3. `uri` - If the target element has an id `some-id`, then the target value is `#some-id`. This option is used to secure message attachments.
`contentOnly`	Indicates whether the complete element or only the contents need to be encrypted (or is required to be encrypted). The default value is `true`. (Relevant only for <Encrypt> and <RequireEncryption> targets)
`value`	Indicates whether the value needs to be encrypted (or is required to be encrypted). The default value is `true`. (Required)
`enforce`	If `true`, indicates that the security operation on the target element is definitely required. Default value is `true`. (Relevant only for <RequireSignature> and <RequireEncryption> targets)

Table 4–34 Sub-elements of EncryptionTarget

Sub-elements of **EncryptionTarget**	**Description**
Transform	Identifies the transform algorithm to be applied to the object to be encrypted.

SymmetricKey

The <SymmetricKey> element indicates the symmetric key to be used for encryption. This element must not be specified if the <X509Token> or <SAMLAssertion> sub-elements are present. Its attributes are discussed in Table 4–35.

Table 4–35 Attributes of SymmetricKey

Attributes of **SymmetricKey**	**Description**
keyAlias	The alias of the symmetric key to be used for encryption. This attribute is required.

CanonicalizationMethod

The <CanonicalizationMethod> element specifies the canonicalization algorithm to be applied to the <SignedInfo> element prior to performing signature calculations. When specified, the canonical XML [XML-C14N] standard, which is an algorithm that standardizes the way XML documents should be ordered and structured, should be applied. The recommendation that discusses this method is the W3C XML-Signature Syntax and Processing recommendation, which can be viewed at http://www.w3.org/TR/xmldsig-core/#sec-CanonicalizationMethod. Its attributes are discussed in Table 4–36.

Table 4–36 Attributes of CanonicalizationMethod

Attributes of **CanonicalizationMethod**	**Description**
algorithm	The algorithm to be used for signing. There is no default value. You must explicitly add `http://www.w3.org/2001/10/xml-exc-c14n#` to the transforms list in the configuration file if you want to use it. The prefix list is computed by the implementation and does not need to be specified in the configuration file. This transform will be added as the last transform regardless of its placement in the configuration file.

SignatureMethod

The <`SignatureMethod`> element specifies the algorithm used for signature generation and validation. A `SignatureMethod` is implicitly given two parameters: the keying info and the output of `CanonicalizationMethod`. The recommendation that discusses this method is the W3C XML-Signature Syntax and Processing recommendation, which can be viewed at http://www.w3.org/TR/xmldsig-core/#sec-SignatureMethod. Its attributes are discussed in Table 4–37.

Table 4–37 Attributes of SignatureMethod

Attributes of **SignatureMethod**	**Description**
`algorithm`	The algorithm to be used for signing. The default value is `http://www.w3.org/2000/09/xmldsig#rsa-sha1`.

DigestMethod

The <`DigestMethod`> element specifies the algorithm used for generating the digest of the object to be signed. The recommendation that discusses this method is the W3C XML-Signature Syntax and Processing recommendation, which can be viewed at http://www.w3.org/TR/xmldsig-core/#sec-DigestMethod. The attributes of <`DigestMethod`> are discussed in Table 4–38.

Table 4–38 Attributes of DigestMethod

Attributes of **DigestMethod**	**Description**
`algorithm`	Identifies the digest algorithm to be applied to the signed object. The default value is `http://www.w3.org/2000/09/xmldsig#sha1`.

DataEncryptionMethod

The <`DataEncryptionMethod`> element specifies the encryption algorithm to be applied to the cipher data. The recommendation that discusses this method is the W3C XML Encryption Syntax and Processing recommendation, which can be viewed at http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/#sec-EncryptionMethod. The attributes of <`DataEncryptionMethod`> are discussed in Table 4–39.

Table 4–39 Attributes of DataEncryptionMethod

Attributes of **DataEncryptionMethod**	**Description**
`algorithm`	The algorithm to be used for encrypting data. The default value is "`http://www.w3.org/2001/04/xmlenc#aes128-cbc`"). Other options include: "`http://www.w3.org/2001/04/xmlenc#aes256-cbc`"; and "`http://www.w3.org/2001/04/xmlenc#tripledes-cbc`".

Note: Although the schema indicates that `http://www.w3.org/2001/04/xmlenc#aes128-cbc` is the default algorithm for <`DataEncryptionMethod`>, for backward compatibility this implementation still uses `http://www.w3.org/2001/04/xmlenc#tripledes-cbc` as the default.

KeyEncryptionMethod

The <`KeyEncryptionMethod`> element specifies the public key encryption algorithm to be used for encrypting and decrypting keys. Its attributes are discussed in Table 4–40.

Table 4–40 Attributes of KeyEncryptionMethod

Attributes of **KeyEncryptionMethod**	**Description**
`algorithm`	Specifies the `KeyTransport`/`KeyWrap` algorithms to be used to encrypt/decrypt a public key or secret key (key used to encrypt the data) respectively. The default value is `http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p`. Other options include: "`http://www.w3.org/2001/04/xmlenc#rsa-1_5`"; "`http://www.w3.org/2001/04/xmlenc#kw-triple-des`"; "`http://www.w3.org/2001/04/xmlenc#kw-aes128`"; and "`http://www.w3.org/2001/04/xmlenc#kw-aes256`".

SecurityEnvironmentHandler

The <`SecurityEnvironmentHandler`> element specifies the implementation class name of the security environment handler. Read Writing SecurityEnvironmentHandlers for more information on `SecurityEnvironmentHandlers`.

How Do I Specify the Security Configuration for the Build Files?

After the security configuration files are created, you can easily specify which of the security configuration files to use for your application. In the `build.properties` file for your application, create a property to specify which security configuration file to use for the client, and which security configuration file to use for the server. An example from the `simple` sample application does this by listing all of the alternative security configuration files, and uncommenting only the configuration to be used. The `simple` sample uses the following properties:

```
# #look in config directory for alternate security
configurations
# Client Security Config. file
client.security.config=config/dump-client.xml
#client.security.config=config/user-pass-authenticate-
client.xml
#client.security.config=config/encrypted-user-pass-client.xml
#client.security.config=config/encrypt-usernameToken-
client.xml
#client.security.config=config/sign-client.xml
#client.security.config=config/encrypt-client.xml
#client.security.config=config/encrypt-using-symmkey-
client.xml
#client.security.config=config/sign-encrypt-client.xml
#client.security.config=config/encrypt-sign-client.xml
#client.security.config=config/sign-ticket-also-client.xml
#client.security.config=config/timestamp-sign-client.xml
#client.security.config=config/flexiblec.xml
#client.security.config=config/method-level-client.xml

# Server Security Config. file
server.security.config=config/dump-server.xml
#server.security.config=config/user-pass-authenticate-
server.xml
#server.security.config=config/encrypted-user-pass-server.xml
#server.security.config=config/encrypt-usernameToken-
server.xml
#server.security.config=config/sign-server.xml
#server.security.config=config/encrypt-server.xml
#server.security.config=config/sign-encrypt-server.xml
#server.security.config=config/encrypt-sign-server.xml
#server.security.config=config/sign-ticket-also-server.xml
#server.security.config=config/timestamp-sign-server.xml
#server.security.config=config/flexibles.xml
#server.security.config=config/method-level-server.xml
```

As you can see from this example, several security scenarios are listed in the `build.properties` file. To run a particular security configuration option, simply uncomment one of the entries for a client configuration file, uncomment the corresponding entry for the server configuration file, and comment all of the other options.

In general, the client and server configuration files should match. However, in some cases, more than one client configuration can be used with a server configuration. For example, either `encrypt-using-symmkey-client.xml` or `encrypt-client.xml` can be used with `encrypt-server.xml`. This combination works because the server requirement is the same (the body contents must be encrypted) when the client-side security configuration is either `encrypt-using-symmkey-client.xml` or `encrypt-client.xml`. The difference in the two client configurations is the key material used for encryption.

After the property has been defined in the `build.properties` file, you can refer to it from the file that contains the `asant` (or `ant`) targets, which is `build.xml`.

When you create an `asant` (or `ant`) target for JAX-RPC clients and services, you use the `wscompile` utility to generate stubs, ties, serializers, and WSDL files. XWS-Security has been integrated into JAX-RPC through the use of security configuration files. The code for performing the security operations on the client and server is generated by supplying the configuration files to the JAX-RPC `wscompile` tool. The `wscompile` tool can be instructed to generate security code by making use of the `-security` option and supplying the security configuration file.

Note: For the 2.0 release of JAX-RPC, JAX-RPC will be renamed to JAX-WS. JAX-WS will become part of the XWS-Security 2.0 FCS later this year. When this renaming occurs, the `wscompile` tool will be replaced, and these steps and the `build.xml` files for the sample applications will need to be modified accordingly.

An example of the target that runs the `wscompile` utility with the `-security` option pointing to the security configuration file specified in the `build.properties` file to generate server artifacts, from the `simple` sample application, looks like this:

```
<target name="gen-server" depends="prepare"
          description="Runs wscompile to generate server 
artifacts">
    <echo message="Running wscompile...."/>
    <wscompile verbose="${jaxrpc.tool.verbose}"
               xPrintStackTrace="true"
               keep="true" fork="true"
       security="${server.security.config}"
               import="true"
               model="${build.home}/server/WEB-INF/
${model.rpcenc.file}"
               base="${build.home}/server/WEB-INF/classes"
               classpath="${app.classpath}"
               config="${config.rpcenc.file}">
```

```
        <classpath>
          <pathelement location="${build.home}/server/WEB-INF/
classes"/>
          <path refid="app.classpath"/>
        </classpath>
      </wscompile>
    </target>
```

An example of the target that runs the `wscompile` utility with the `security` option pointing to the security configuration file specified in the `build.properties` file to generate the client-side artifacts, from the `simple` sample application, looks like this:

```
<target name="gen-client" depends="prepare"
          description="Runs wscompile to generate client side
artifacts">
    <echo message="Running wscompile...."/>
    <wscompile fork="true" verbose="${jaxrpc.tool.verbose}"
keep="true"
               client="true"
               security="${client.security.config}"
               base="${build.home}/client"
               features=" "
               config="${client.config.rpcenc.file}">
      <classpath>
        <fileset dir="${build.home}/client">
            <include name="secenv-handler.jar"/>
        </fileset>
        <path refid="app.classpath"/>
      </classpath>
    </wscompile>
  </target>
```

Refer to the documentation for the `wscompile` utility in Useful XWS-Security Command-Line Tools for more information on `wscompile` options.

Are There Any Sample Applications Demonstrating XWS-Security?

This release of the Java WSDP includes many example applications that illustrate how a JAX-RPC or stand-alone SAAJ application developer can use the XML and Web Services Security framework and APIs. The example applications can be found in the `<JWSDP_HOME>/xws-security/samples/<sample_name>/` directory. Before you can run the sample applications, you must follow the setup instructions in Setting Up To Use XWS-Security With the Sample Applications.

The sample applications print out both the client and server request and response SOAP messages. The output from the server may be viewed in the appropriate container's log file. The output from the client is sent to `stdout` or whichever stream is used by the configured log handler. Messages are logged at the `INFO` level.

Note: In some of the sample security configuration files, no security is specified for either a request or a response. In this case, the response is a simple JAX-RPC response. When XWS-Security is enabled for an application by providing the `-security` option to `wscompile`, and a request or response not containing a `wsse:Security Header` is received, the message `WSS0202: No Security element in the message` will display in the output to warn that a nonsecure response was received.

In these examples, the server-side code is found in the `<`*`JWSDP_HOME`*`>/xws-security/samples/<`*`sample_name`*`>/server/src/<`*`sample_name`*`>/` directory. Client-side code is found in the `<`*`JWSDP_HOME`*`>/xws-security/samples/<`*`sample_name`*`>/client/src/<`*`sample_name`*`>/` directory. The `asant` (or `ant`) targets build objects under the `/build/server/` and `/build/client/` directories.

These examples can be deployed onto any of the following containers. For the purposes of this tutorial, only deployment to the Sun Java System Application Server Platform Edition 8.1 will be discussed. The `README.txt` file for each example provides more information on deploying to the other containers. The following containers can be downloaded from http://java.sun.com/webservices/containers/index.html.

- Sun Java System Application Server Platform Edition 8.1 (Application Server)
- Sun Java System Web Server 6.1 (Web Server)

  If you are using the Java SDK version 5.0 or higher, download service pack 4 for the Web Server. If you are using version 1.4.2 of the Java SDK, download service pack 2 or 3.
- Tomcat 5 Container for Java WSDP (Tomcat)

These examples use keystore and truststore files that are included in the `<`*`JWSDP_HOME`*`>/xws-security/etc/` directory. For more information on using keystore and truststore files, read the `keytool` documentation at http://java.sun.com/j2se/1.5.0/docs/tooldocs/solaris/keytool.html. Refer to the application's `README.txt` file if deploying on the Web Server or Tomcat.

The following list provides the name, a short description, and a link to further discussion of each of the sample applications available in this release:

- `simple`
  This sample application lets you plug in different client and server-side configurations describing security settings. This example has support for digital signatures, XML encryption/decryption, and username token verification. This example allows and demonstrates combinations of these basic security mechanisms through configuration files. The section Simple Security Configurations Sample Application provides examples and descriptions of configuration files used in the sample application, along with instructions for compiling and running the application.
- `api-sample`
  This sample application shows how to use the XWS-Security 2.0 APIs in a stand-alone mode. This sample defines the `XWSSProcessor` interface, which is used to insulate the API user from changes that may occur in future releases of the API, and provides an implementation for it. The `Client.java` file uses the `XWSSProcessor` APIs to secure a SOAP message. The section XWS-Security APIs Sample Application provides further description of the sample application, along with instructions for compiling and running the application.
- `jaas-sample`
  This sample demonstrates how to plug in a JAAS `LoginModule` for username-password authentication. Read more about JAAS at http://java.sun.com/products/jaas/. The section JAAS Sample Application provides further description of the sample application, along with instructions for compiling and running the application.
- `swainterop`
  This sample application demonstrates the Soap Messages with Attachments (SwA) interoperability scenarios. The section Soap With Attachments Sample Application provides further description of the sample application, along with instructions for compiling and running the application.
- `samlinterop`
  This sample application demonstrates support for OASIS WSS Security Assertion Markup Language (SAML) Token Profile 1.0 in XWS-Security. The section

SAML Sample Application provides further description of the sample application, along with instructions for compiling and running the application.

- `dynamic-policy`
  This sample application demonstrates how the request and response security policies can be set at runtime from the `SecurityEnvironmentHandler` callback. The section Dynamic Policy Sample Application provides further description of the sample application, along with instructions for compiling and running the application.
- `dynamic-response`
  This sample application demonstrates using the certificate that arrived in a signed request to encrypt the response back to the requester. The section Dynamic Response Sample Application provides further description of the sample application, along with instructions for compiling and running the application.

Writing SecurityEnvironmentHandlers

The signing and encryption operations require private-keys and certificates. An application can obtain such information in various ways, such as looking up a keystore with an alias, using the default key-pairs available with the container, looking up a truststore with an alias, etc. Similarly if an application wants to send a username-password in a `UsernameToken`, it can choose to obtain the username-password pair in various ways, such as reading from a file, prompting the user on the console, using a popup window, etc. The authentication of the username-password on the receiving application can similarly be done by plugging into existing authentication infrastructure, using a proprietary username-password database, etc.

To support these possibilities, XWS-Security defines a set of `CallBack` classes and requires the application to define a `CallBackHandler` to handle these callbacks. The `xwss:SecurityEnvironmentHandler` element is a compulsory child element that needs to be specified. The value of this element is the class name of a Java class that implements the `javax.security.auth.callback.CallbackHandler` interface and handles the set of callbacks defined by XWS-Security. There are a set of callbacks that are mandatory and every `CallbackHandler` needs to implement them. A few callbacks are optional and can be used to supply some fine-grained property information to the XWS-Security run-time.

When using the XWS-Security APIs for securing both JAX-RPC applications and stand-alone applications that make use of SAAJ APIs only for their SOAP messaging, you have the option of either implementing a CallbackHandler or implementing the com.sun.xml.wss.SecurityEnvironment interface. Once implemented, the appropriate instance of the CallbackHandler or SecurityEnvironment interface implementation needs to be set into an instance of com.sun.xml.wss.ProcessingContext. For example code uses the XWS-Security APIs, refer to XWS-Security APIs Sample Application. The SecurityEnvironment interface is evolving and is subject to refinement in a later release.

Because information such as private keys and certificates for signing and encryption can be obtained in various ways (looking up a keystore with an alias, using the default key-pairs available with the container, looking up a truststore with an alias, etc.), every callback defines a set of Request inner classes and a callback can be initialized with any of its request inner classes. A tagging Request interface is also defined within the callback to tag all Request classes. For example, the XWS-Security configuration schema defines an xwss:X509Token element containing an optional attribute certificateAlias. When the xwss:X509Token element embedded inside a xwss:Sign element has a certificateAlias attribute specified as shown in the following code snippet, the XWS-Security run-time would invoke the SecurityEnvironmentHandler of the application with a SignatureKeyCallback object to obtain the private-key required for the signing operation.

```
<xwss:Sign>
    <xwss:X509Token certificateAlias="xws-security-client"/>
</xwss:Sign>
```

The SignatureKeyCallback will be initialized by XWS-Security run-time with an AliasPrivKeyCertRequest in the following manner:

```
SignatureKeyCallback sigKeyCallback = new
SignatureKeyCallback(new
   SignatureKeyCallback.AliasPrivKeyCertRequest(alias));
```

The application's SecurityEnvironmentHandler implementation then needs to handle the SignatureKeyCallback and use the alias to locate and set the private-key and X.509 certificate pair on the AliasPrivKeyCertRequest. The following code shows how this callback is handled in the handle() method of SecurityEnvironmentHandler shipped with the simple sample.

```
} else if (callbacks[i] instanceof SignatureKeyCallback) {
   SignatureKeyCallback cb =
(SignatureKeyCallback)callbacks[i];
```

```
        if (cb.getRequest() instanceof
SignatureKeyCallback.AliasPrivKeyCertRequest) {
          SignatureKeyCallback.AliasPrivKeyCertRequest request
=
             (SignatureKeyCallback.AliasPrivKeyCertRequest)
cb.getRequest();
          String alias = request.getAlias();
          if (keyStore == null)
             initKeyStore();
             try {
                X509Certificate cert = (X509Certificate)
keyStore.getCertificate(alias);
                request.setX509Certificate(cert);
                // Assuming key passwords same as the keystore
password
                PrivateKey privKey =
                   (PrivateKey) keyStore.getKey(alias,
keyStorePassword.toCharArray());
                request.setPrivateKey(privKey);
             } catch (Exception e) {
                   throw new IOException(e.getMessage());
                }
                } else {
                      throw  new
UnsupportedCallbackException(null, "Unsupported Callback
                         Type Encountered");
                }
             }
```

This handler uses a keystore to locate the private key and certificate pair, and sets it using `AliasPrivKeyCertRequest`.

As shown in the sample code, the `SecurityEnvironmentHandler` should throw an `UnsupportedCallbackException` whenever it cannot handle a `Callback` or a particular `Request` type of a `Callback`.

The type of `Request` with which the `Callback` is initialized often depends on the information specified in the security configuration file of the application. For example if the `xwss:X509Token` specified under an `xwss:Sign` element did not contain the `certificateAlias` attribute, XWS-Security would invoke the application's `SecurityEnvironmentHandler` with `SignatureKeyCallback.DefaultPrivKeyCertRequest` to try and obtain the default private-key and certificate pair. If the `SecurityEnvironmentHandler` does not handle this request and throws an `UnsupportedCallbackException`, the signature operation would fail.

For more information, read the API documentation for callbacks from the `<JWSDP_HOME>/xws-security/docs/api/com/sun/xml/wss/impl/callback/package-summary.html`. This documentation includes the list of mandatory and optional callbacks and the details of the `Callback` classes and supported methods. Table 4–41 provides a brief summary of all the mandatory `Callback` classes and their associated `Request` types.

Table 4–41 Summary of `Callback` classes and their `Request` types

Callback	Description	Request Inner Classes Defined	Methods in the Request Classes

Signature Key Callback	Used by XWS-Security runtime to obtain the private key to be used for signing the corresponding X.509 certificate. There are two ways in which an application can supply the private-key and certificate information.
	1. Lookup a keystore using an alias.
	2. Obtain the default private-key and certificate from the container/environment in which the application is running.
	3. Obtain the private key and certificate given the public key. This kind of request is used in scenarios where the public key appears as a `KeyValue` under a `ds:KeyInfo` and needs to be used for signing.
	
	Accordingly, there are three `Request` inner classes with which the `SignatureKeyCallback` can be initialized.
		1. `AliasPrivKeyCertRequest`: A `Callback` initialized with this request should be handled if the private key to be used for signing is mapped to an alias.
		2. `DefaultPrivKeyCertRequest`: A `Callback` initialized with this request should be handled if there's some default private key to be used for signing.
		3. `PublicKeyBasedPrivateKeyCertRequest`: A callback initialized with this request should be handled if the private key to be used for signing is to be retrieved given the public key.
			The following four methods are present in all `Request` Classes of this `Callback`:
			`public void setPrivateKey(`
			`PrivateKey privateKey)`
			`public PrivateKey getPrivateKey()`
			
			`public void`
			`setX509Certificate(`
			`X509Certificate certificate)`
			`public X509Certificate`
			`getX509Certificate()`

Table 4–41 Summary of `Callback` classes and their `Request` types (Continued)

Callback	Description	Request Inner Classes Defined	Methods in the Request Classes

`SignatureVerificationKeyCallback`	Obtains the certificate required for signature verification. There are currently two situations in which XWS-Security would require this Callback to resolve the certificate:
	1. When the signature to be verified references the key using an X.509 `SubjectKeyIdentifier`. For example, when the sender specifies the attribute `xwss:keyReferenceType="Identifier"` on the `xwss:X509Token` child of the `xwss:Sign` element.
	2. When the signature to be verified references the key using an X.509 `IssuerSerialNumber`. For example, when the sender specifies the attribute `xwss:keyReferenceType="IssuerSerialNumber"` on the `xwss:X509Token` child of the `xwss:Sign` element.
	3. When ds:KeyInfo contains a key value, use the public key to obtain the X.509 certificate.
	
	Accordingly, there are three `Request` inner classes with which a `SignatureVerificationKeyCallback` can be initialized.
	Note: Additional `Requests` may be defined in a future release.
		1. `X509SubjectKeyIdentifierBasedRequest`: Request for an X.509 certificate whose X.509 `SubjectKeyIdentifier` value is given.
		2. `X509IssuerSerialBasedRequest`: Request for an X.509 certificate whose issuer name and serial number values are given.
		3. `PublicKeyBasedRequest`: Request for an X.509 certificate for a given public key.
			The following two methods are present in all the `Request` classes of this `Callback`:
			
			`public void setX509Certificate(`
			`   X509Certificate certificate)`
			`public X509Certificate`
			`   getX509Certificate()`

Table 4–41 Summary of `Callback` classes and their `Request` types (Continued)

Callback	Description	Request Inner Classes Defined	Methods in the Request Classes

`EncryptionKeyCallback`	Obtains the certificate for key-encryption or a symmetric-key for data encryption. The three situations for which XWS-Security would require this `Callback` for performing encryption:
	1. When the `xwss:Encrypt` element contains an `xwss:X509Token` child with `certificateAlias` attribute set to an alias. The `certificateAlias` indicates that a random symmetric key is used for encryption of the specified message part and the certificate is then used to encrypt the random symmetric-key to be sent along with the message.
	2. When the `xwss:Encrypt` element contains an `xwss:X509Token` child with no `certificateAlias` attribute set on it. XWS-Security tries to obtain a default certificate from the `Callback` to be used for encrypting the random symmetric key.
	3. When the `xwss:Encrypt` element contains an `xwss:SymmetricKey` child specifying the `keyAlias` attribute. This alias indicates that a symmetric key corresponding to this alias needs to be located and used for encryption of the specified message part.
	4. When an X.509 certificate needs to be obtained for a given public key.
		The following are the `Request` inner classes with which an `EncryptionKeyCallback` can be initialized.
		1. `AliasX509CertificateRequest`: A `Callback` initialized with this request should be handled if the X.509 certificate to be used for encryption is mapped to an alias.
		2. `DefaultX509CertificateRequest`: A `Callback` initialized with this request should be handled if there's a default X.509 certificate to be used for encryption.
		3. `AliasSymmetricKeyRequest`: A `Callback` initialized with this request should be handled if the symmetric key to be used for encryption is mapped to an alias.
		4. `PublicKeyBasedRequest`: Request for an X.509 certificate for a given public key.
			The following two methods are present in the `AliasX509CertificateRequest` and `DefaultX509CertificateRequest Request` classes of this `Callback`:
			
			`public void setX509Certificate(`
			`   X509Certificate certificate)`
			`public X509Certificate`
			`   getX509Certificate()`
			
			The following methods are present in the `AliasSymmetricKeyRequest` class of this `Callback`:
			
			`public void setSymmetricKey(`
			`   javax.crypto.SecretKey`
			`      symmetricKey)`
			`public javax.crypto.SecretKey`
			`   getSymmetricKey()`

Table 4–41 Summary of `Callback` classes and their `Request` types (Continued)

Callback	Description	Request Inner Classes Defined	Methods in the Request Classes

`DecryptionKeyCallback`	Obtains the symmetric key to be used for decrypting the encrypted data or obtaining the private-key for decrypting the encrypted random symmetric key that was sent with the message (along with the encrypted data).
	There are currently four situations in which XWS-Security will require this `Callback` to perform decryption.
	1. When the `EncryptedKey` references the key (used for encrypting the symmetric key) using an X.509 `SubjectKeyIdentifier`. For example, when the sender specifies the attribute `keyReferenceType="Identifier"` on the `xwss:X509Token` child of the `xwss:Encrypt` element.
	2. When the `EncryptedKey` references the key (used for encrypting the symmetric key) using an X.509 `IssuerSerialNumber`. For example, when the sender specifies the attribute `keyReferenceType="IssuerSerialNumber"` on the `xwss:x509Token` child of `xwss:Encrypt` element.
		1. `X509SubjectKeyIdentifierBasedRequest`: Request for a private-key when the X.509 `SubjectKeyIdentifier` value for a corresponding X.509 certificate is given.
		2. `X509IssuerSerialBasedRequest`: Request for a private key when the issuer name and serial number values for a corresponding X.509 certificate are given.
		3. `X509CertificateBasedRequest`: Request for a private key when a corresponding X.509 certificate is given.
			The following two methods are present in the `X509SubjectKeyIdentifierBasedRequest`, `X509IssuerSerialBasedRequest`, and `X509CertificateBasedRequest` `Request` classes of this `Callback`:
			
			`public void setPrivateKey(`
			   `PrivateKey privateKey)`
			`public PrivateKey`
			   `getPrivateKey()`

Table 4–41 Summary of `Callback` classes and their `Request` types (Continued)

Callback	Description	Request Inner Classes Defined	Methods in the Request Classes

`DecryptionKeyCallback` (continued)	3. When the `EncryptedKey` contains a `wsse:Direct` reference to the key used for encrypting the symmetric key. This means the X.509 certificate is present as a `wsse:BinarySecurityToken` in the message. For example, when the sender specifies the attribute `keyReferenceType="Direct"` on the `xwss:x509Token` child of `xwss:Encrypt` element.
	4. When the `EncryptedData` contains a `ds:keyName` reference to the symmetric key that was used for encryption. For example, when the sender specifies the `xwss:SymmetricKey` child of `xwss:Encrypt` and specifies the `keyAlias` attribute on it.
	5. When the `EncryptedKey` contains a `ds:KeyInfo` with a key value child.
	Accordingly, there are five `Request` classes with which a `DecryptionKeyCallback` can be initialized.
		4. `AliasSymmetricKeyRequest`: A `Callback` initialized with this request should be handled if the symmetric key to be used for decryption is mapped to some alias.
		5. `PublicKeyBasedPrivateKeyRequest`: Request for a private key given the public key.
			The following methods are present in the `AliasSymmetricKeyRequest` class of this `Callback`:
			
			`public void setSymmetricKey(`
			   `javax.crypto.SecretKey`
			      `symmetricKey)`
			`public javax.crypto.SecretKey`
			   `getSymmetricKey()`

Table 4–41 Summary of `Callback` classes and their `Request` types (Continued)

Callback	Description	Request Inner Classes Defined	Methods in the Request Classes

`Password Validation Callback`	Username-Password validation. A validator that implements the `PasswordValidator` interface should be set on the callback by the callback handler. There are currently two situations in which XWS-Security will require this `Callback` to perform username-password validation:
	1. When the receiver gets a `UsernameToken` with plain-text user name and password.
	2. When the receiver gets a `UsernameToken` with a digested password (as specified in the WSS Username-Token Profile).
	Accordingly there are two Request classes with which the `PasswordValidationCallback` can be initialized.
	Note: A validator for WSS Digested Username-Password is provided as part of this callback, with classname `PasswordValidationCallback.DigestPasswordValidator`.
	This class implements WSS digest password validation. The method for computing password digest is described in `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0.pdf`.
	For more information, see the `ServerSecurityEnvironmentHandler` in <*JWSDP_HOME*>`/xws-security/samples/jaas-sample/src/com/sun/xml/wss/sample`.
		1. `PlainTextPasswordRequest`: Represents a validation request when the password in the username token is in plain text.
		2. `DigestPasswordRequest`: Represents a validation request when the password in the username token is in digested form.
			The following methods are present in the `PlainTextPasswordRequest`:
			`public String getUsername()`
			`public String getPassword()`
			
			The following methods are present in the `DigestPasswordRequest`:
			`public void setPassword(String password)`
			
			This method must be invoked by the `CallbackHandler` while handling a `Callback` initialized with `DigestPasswordRequest` to set the plain-text password on the `Callback`.
			
			`public java.lang.String getPassword()`
			`public java.lang.String getUsername()`
			`public java.lang.String getDigest()`
			`public java.lang.String getNonce()`
			`public java.lang.String getCreated()`

Table 4–41 Summary of `Callback` classes and their `Request` types (Continued)

Callback	Description	Request Inner Classes Defined	Methods in the Request Classes

`UsernameCallback`	To supply the user name for the `UsernameToken` at runtime. It contains the following two methods:
	`public void setUsername(`
	`    String username)`
	`public String getUsername()`
	
	Refer to the `ClientSecurityEnvironmentHandler` of the `jaas-sample` located in <*JWSDP_HOME*>`/xws-security/samples/jaas-sample/src/com/sun/xml/wss/sample` for more details on using the `UsernameCallback`.		

`PasswordCallback`	To supply the password for the username token at runtime. It contains the following two methods:
	
	`public void setPassword(String`
	`   password)`
	`public String getPassword()`
	Refer to the `ClientSecurityEnvironmentHandler` of the `jaas-sample` located in <*JWSDP_HOME*>`/xws-security/samples/jaas-sample/src/com/sun/xml/wss/sample` for more details on using the `PasswordCallback`.		

Table 4–41 Summary of `Callback` classes and their `Request` types (Continued)

Callback	Description	Request Inner Classes Defined	Methods in the Request Classes
`PropertyCallback`	Optional callback to specify the values of properties configurable with XWS-Security run-time. Refer to the API documentation at *`<JWSDP_HOME>`*`/xws-security/docs/api/com/sun/xml/wss/impl/callback/PropertyCallback.html` for a list of configurable properties and methods supported by this callback.		This callback has been deprecated and disabled in this release. To get similar functionality, use the `maxClockSkew` and `timestampFreshnessLimit` attributes on `<RequireTimestamp>`, or the `maxClockSkew`, `timestampFreshnessLimit`, and `maxNonceAge` attributes on `<RequireUsernameToken>`.

Table 4–41 Summary of `Callback` classes and their `Request` types (Continued)

Callback	Description	Request Inner Classes Defined	Methods in the Request Classes
`Prefix Namespace Mapping Callback`	Optional callback to register any prefix versus namespace-uri mappings that the developer wants to make use of in the security configuration (while specifying `Targets` as `xpaths`). Refer to the API documentation at *<JWSDP_HOME>*`/xws-security/docs/api/com/sun/xml/wss/impl/callback/PrefixNamespaceMappingCallback.html` for more details.		The `PrefixNamespaceMappingCallback` has been deprecated and disabled in this release. When specifying XPath expressions for targets in XWS-Security configuration files, you are required to make use of the elongated syntax of the form `local-name()="Body"` and `namespace-uri()="http://schemas.xmlsoap.org/soap/envelope/"`, etc., if the prefix involved is anything other than the following: 1. The prefix of the SOAP envelope in the message. 2. One of the following prefixes: `SOAP-ENV`, `env`, `S11` to mean `http://schemas.xmlsoap.org/soap/envelope/`. 3. Prefix `ds` to mean `http://www.w3.org/2000/09/xmldsig#` 4. Prefix `xenc` to mean `http://www.w3.org/2001/04/xmlenc#` 5. Prefix `wsse` to mean `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd` 6. Prefix `wsu` to mean `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd` 7.Prefix `wsu` to mean `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd` The use of XPath expressions is discouraged in XWS-Security EA 2.0 because it impacts performance. Users are advised to make use of fragment URI's and QNames to identify targets of signature and encryption.

Table 4–41 Summary of `Callback` classes and their `Request` types (Continued)

Callback	Description	Request Inner Classes Defined	Methods in the Request Classes
`CertificateValidationCallback`	This callback is intended for X.509 certificate validation. A validator that implements the `CertificateValidator` interface should be set on the callback by the callback handler. Currently this callback is invoked by the XWS-Security runtime whenever an X.509 certificate is present in an incoming message in the form of a `BinarySecurityToken`.		
`DynamicPolicyCallback`	This callback is intended for dynamic policy resolution. `DynamicPolicyCallback` is made by the XWS runtime to allow the application and/or handler to decide the incoming and/or outgoing `SecurityPolicy` at runtime. When the `SecurityPolicy` set on the callback is a `DynamicSecurityPolicy`, the `CallbackHandler` is expected to set a `com.sun.xml.wss.impl.configuration.MessagePolicy` instance as the resolved policy. The `MessagePolicy` instance can contain policies generated by the `PolicyGenerator` obtained from the `DynamicSecurityPolicy`.		

The following code snippet shows the `handle()` method skeleton for an application's `SecurityEnvironmentHandler` that handles all the mandatory `Callbacks` (except `UsernameCallback` and `PasswordCallback`) and associated `Requests` defined by XWS-Security. A particular application may choose to throw an `UnsupportedCallbackException` for any of the `Callbacks` or its `Requests` that it cannot handle. The `UsernameCallback` and `PasswordCallback` are useful for obtaining a username-password pair at run-time and are explained later in this section.

Note: In this release of XWS-Security, users will have to ensure that the `SecurityEnvironmentHandler` implementation they supply is thread safe.

```
public  class SecurityEnvironmentHandler implements
CallbackHandler {

   public void handle(Callback[] callbacks) throws IOException,
      UnsupportedCallbackException {

      for (int i=0; i < callbacks.length; i++) {

         if (callbacks[i] instanceof
PasswordValidationCallback) {
            PasswordValidationCallback cb =
(PasswordValidationCallback) callbacks[i];
             if (cb.getRequest() instanceof
               PasswordValidationCallback.PlainTextPasswordReq
uest) {
               // setValidator for plain-text password
validation on callback cb
               } else if (cb.getRequest() instanceof
                     PasswordValidationCallback.DigestPassword
Request) {
                        PasswordValidationCallback.DigestPassw
ordRequest request =
                           (PasswordValidationCallback.DigestP
asswordRequest) cb.getRequest();
                        // set plaintext password on  request
                        // setValidator for digest password
validation on cb

               } else {
                     // throw unsupported;
                  }

               } else if (callbacks[i] instanceof
SignatureVerificationKeyCallback) {
                     SignatureVerificationKeyCallback cb =
                        (SignatureVerificationKeyCallback)call
backs[i];

                     if (cb.getRequest() instanceof
                       SignatureVerificationKeyCallback.X509Su
bjectKeyIdentifierBasedRequest) {
                       // subject keyid request
                       SignatureVerificationKeyCallback.X509Su
bjectKeyIdentifierBasedRequest
                          request =
                       (SignatureVerificationKeyCallback.X509S
ubjectKeyIdentifierBasedRequest)
                           cb.getRequest();
                       // locate and setX509Certificate on the
request
                        } else if (cb.getRequest() instanceof
                           SignatureVerificationKeyCallback.X5
09IssuerSerialBasedRequest) {
                        // issuer serial request
                        SignatureVerificationKeyCallback.X509I
ssuerSerialBasedRequest request =
                           (SignatureVerificationKeyCallback.X
509IssuerSerialBasedRequest)
                           cb.getRequest();
                        // locate and setX509Certificate on the
request

                     } else  {
                        // throw unsupported;
                        }

                     } else if (callbacks[i] instanceof
SignatureKeyCallback) {
                           SignatureKeyCallback cb =
(SignatureKeyCallback)callbacks[i];
                          if (cb.getRequest() instanceof
                             SignatureKeyCallback.DefaultPrivK
eyCertRequest) {
                             // default priv key cert req

SignatureKeyCallback.DefaultPrivKeyCertRequest request =

   (SignatureKeyCallback.DefaultPrivKeyCertRequest)
cb.getRequest();
                             // locate and set default
privateKey and X509Certificate on request
```

```
                      } else if (cb.getRequest() instanceof

SignatureKeyCallback.AliasPrivKeyCertRequest) {
                               // Alias priv key cert req

   SignatureKeyCallback.AliasPrivKeyCertRequest request =

      (SignatureKeyCallback.AliasPrivKeyCertRequest)
cb.getRequest();
                               // locate and set default
privateKey and X509Certificate on request

                      } else {
                         // throw unsupported;
                         }

                      } else if (callbacks[i] instanceof
DecryptionKeyCallback) {
                         DecryptionKeyCallback cb =
(DecryptionKeyCallback)callbacks[i];

                         if (cb.getRequest() instanceof
                            DecryptionKeyCallback.X509Subject
KeyIdentifierBasedRequest) {
                               //ski  request
                            DecryptionKeyCallback.X509Subject
KeyIdentifierBasedRequest request =
                               (DecryptionKeyCallback.X509Sub
jectKeyIdentifierBasedRequest)
                         cb.getRequest();
                         // locate and set the privateKey on
the request

                      } else if (cb.getRequest() instanceof
                            DecryptionKeyCallback.X509IssuerS
erialBasedRequest) {
                               // issuer serial request
                               DecryptionKeyCallback.X509Issu
erSerialBasedRequest request =
                                  (DecryptionKeyCallback.X509
IssuerSerialBasedRequest)
                                     cb.getRequest();
                               // locate and set the
privateKey on the request
                      } else if (cb.getRequest() instanceof
                            DecryptionKeyCallback.X509Certifi
cateBasedRequest) {
                               // X509 cert request
```

```
                                DecryptionKeyCallback.X509Cert
ificateBasedRequest request =
                                  (DecryptionKeyCallback.X509C
ertificateBasedRequest)
                                      cb.getRequest();
                                // locate and set private key
on the request
                       } else if (cb.getRequest() instanceof
                             DecryptionKeyCallback.AliasSymmet
ricKeyRequest) {
                                DecryptionKeyCallback.AliasSym
metricKeyRequest request =
                                   (DecryptionKeyCallback.Alia
sSymmetricKeyRequest)
                                      cb.getRequest();
                                // locate and set symmetric key
on request

                       } else  {
                          // throw unsupported;
                          }

                       } else if (callbacks[i] instanceof
EncryptionKeyCallback) {
                             EncryptionKeyCallback cb =
(EncryptionKeyCallback)callbacks[i];
                             if (cb.getRequest() instanceof
                                EncryptionKeyCallback.AliasX50
9CertificateRequest) {
                                EncryptionKeyCallback.AliasX50
9CertificateRequest request =
                                   (EncryptionKeyCallback.Alia
sX509CertificateRequest)
                                      cb.getRequest();
                                // locate and set certificate
on request
                       } else if (cb.getRequest() instanceof
                              EncryptionKeyCallback.AliasSymme
tricKeyRequest) {
                                 EncryptionKeyCallback.AliasSy
mmetricKeyRequest request =
                                    (EncryptionKeyCallback.Ali
asSymmetricKeyRequest)
                                       cb.getRequest();
                                 // locate and set symmetric
key on request

                       } else {
                          // throw unsupported;
                          }

                      } else if (callbacks[i] instanceof
CertificateValidationCallback) {
                             CertificateValidationCallback cb
=
                                (CertificateValidationCallback
)callbacks[i];
                             // set an X509 Certificate
Validator on the callback
                      } else {
                          // throw unsupported;
                          }
            }
      }
}
```

An application can also choose not to handle certain callbacks if it knows that the particular application will never require those callbacks. For example if the security application only deals with signing the requests and does not deal with encryption or username tokens, its `handle()` method only needs to worry about `SignatureKeyCallback` (with its associated `Requests`) and `SignatureVerificationKeyCallback` (with its associated `Requests`). It can then throw an `UnsupportedCallbackException` for any other callback. The following code shows the `handle()` method skeleton for such an application:

```
public  class SecurityEnvironmentHandler implements
CallbackHandler {

   public void handle(Callback[] callbacks) throws IOException,
      UnsupportedCallbackException {

         for (int i=0; i < callbacks.length; i++) {
            if (callbacks[i] instanceof
SignatureVerificationKeyCallback) {
                  (SignatureVerificationKeyCallback)callbacks[
i];

                  if (cb.getRequest() instanceof
                     SignatureVerificationKeyCallback.X509Subj
ectKeyIdentifierBasedRequest) {
                     // subject keyid request
                     SignatureVerificationKeyCallback.X509Subj
ectKeyIdentifierBasedRequest
                         request =
                         (SignatureVerificationKeyCallback.X50
9SubjectKeyIdentifierBasedRequest)
                            cb.getRequest();
                     // locate and setX509Certificate on the
request
                  } else if (cb.getRequest() instanceof
                        SignatureVerificationKeyCallback.X509I
ssuerSerialBasedRequest) {
                        // issuer serial request
                        SignatureVerificationKeyCallback.X509I
ssuerSerialBasedRequest request =
                           (SignatureVerificationKeyCallback.X
509IssuerSerialBasedRequest)
                              cb.getRequest();
                     // locate and setX509Certificate on the
request

                  } else  {
                     // throw unsupported;
                      }

                  } else if (callbacks[i] instanceof
SignatureKeyCallback) {
                        SignatureKeyCallback cb =
(SignatureKeyCallback)callbacks[i];
                        if (cb.getRequest() instanceof
                           SignatureKeyCallback.DefaultPrivKey
CertRequest) {
                        // default priv key cert req
                        SignatureKeyCallback.DefaultPrivKeyCer
tRequest request =
                           (SignatureKeyCallback.DefaultPrivKe
yCertRequest) cb.getRequest();
                        // locate and set default privateKey
and X509Certificate on request
                  } else if (cb.getRequest() instanceof
                        SignatureKeyCallback.AliasPrivKeyCertR
equest) {
                        // Alias priv key cert req
                        SignatureKeyCallback.AliasPrivKeyCertR
equest request =
                           (SignatureKeyCallback.AliasPrivKeyC
ertRequest) cb.getRequest();
                        // locate and set default privateKey
and X509Certificate on request

                  } else {
                     //  throw unsupported;
                     }
```

```
                    } else {
                      // throw unsupported;
                    }
                }
            }
    }
```

Similarly, an application dealing only with `UsernameToken` but not signature or encryption requirements can simply throw `UnsupportedCallbackException` for all non-username related callbacks.

The `SecurityEnvironmentHandler` implementation for the `simple` sample is located in the directory *<JWSDP_HOME>*`/xws-security/samples/simple/src/com/sun/xml/wss/sample`. The `simple` sample uses the same `SecurityEnvironmentHandler` for both the client and server side.

The `jaas-sample` requires a different set of callbacks to be handled on the client and server side. The `CallbackHandlers` for the `jaas-sample` are located in the directory *<JWSDP_HOME>*`/xws-security/samples/jaas-sample/src/com/sun/xml/wss/sample`. The two `CallbackHandlers` defined for the `jaas-sample` are:

- A `ClientSecurityEnvironmentHandler` that handles only the `UsernameCallback` and `PasswordCallback` for retrieving the username and password to be sent in a WSS `UsernameToken`.
- A `ServerSecurityEnvironmentHandler` that handles only the `PasswordValidationCallback` to validate the username-password pair that it received in the WSS `UsernameToken`.

Using the SubjectAccessor API

XWS-Security applications might require access to the authenticated subject of the sender from within the SEI implementation methods. The `SubjectAccessor` API contains a single method:

```
public static Subject getRequesterSubject(Object context)
   throws XWSSecurityException
public static Subject getRequesterSubject()
```

The `getRequesterSubject(Object context)` method returns the `Subject` if one is available or else it returns `NULL`. The context argument to be passed into this method should either be a `ServletEndpointContext`, which is available with the SEI implementation class, or a com.sun.xml.wss.ProcessingContext. For an example on how the SubjectAccessor is used to obtain the authenticated sender subject, refer to the PingImpl.java class in the jaas-sample located at *<JWSDP_HOME>*/xws-security/samples/jaas-sample/server/src/sample. The API for SubjectAccessor viewed from *<JWSDP_HOME>*/xws-security/docs/api/com/sun/xml/wss/SubjectAccessor.html.

The getRequesterSubject() method returns the requester subject from the context if available, and returns null if not available. This method should be used by the receiver response processing to access the subject of the requester. This method will work only for the Synchronous Request-Response Message Exchange Pattern (SRRMEP). For an example that uses this method, see Dynamic Response Sample Application.

Useful XWS-Security Command-Line Tools

In this release, the following command-line tools are included. These tools provide specialized utilities for keystore management or for specifying security configuration files:

- pkcs12import
- keyexport
- wscompile

For more information on keystore management, read the Application Server Administration Guide topic Working with Certificates and SSL.

pkcs12import

The pkcs12import command allows *Public-Key Cryptography Standards version 12* (PKCS-12) files (sometimes referred to as PFX files) to be imported into a keystore, typically a keystore of type *Java KeyStore* (JKS).

When would you want to do this? One example would be a situation where you want to obtain a new certificate from a certificate authority. In this scenario, one option is to follow this sequence of steps:

1. Generate a key-pair.
2. Generate a certificate request

3. Send the request to the authority for its signature
4. Get the signed certificate and import it into this keystore.

Another option is to let the certificate authority generate a key-pair. The authority would return a generated certificate signed by itself along with the corresponding private key. One way the certificate authority can return this information is to bundle the key and the certificate in a PKCS-12 formatted file (generally `pfx` extension files). The information in the PKCS-12 file would be encrypted using a password that would be conveyed to the user by the authority. After receiving the PKCS-12 formatted file, you would import this key-pair (certificate/private-key pair) into your private keystore using the `pkcs12import` tool. The result of the import is that the private-key and the corresponding certificate in the PKCS-12 file are stored as a key entry inside the keystore, associated with some alias.

The `pkcs12import` tool can be found in the directory `<`*`JWSDP_HOME`*`>/xws-security/bin`, and can be run from the command line by executing `pkcs12import.sh` (on Unix systems) or `pkcs12import.bat` (on Windows systems). The options for this tool listed in Table 4–42.

Table 4–42 Options for `pkcs12import` tool

Option	**Description**
`-file` *`pkcs12-file`*	Required. The location of the PKCS-12 file to be imported.
`[ -pass` *`pkcs12-password`* `]`	The password used to protect the PKCS-12 file. The user is prompted for this password if this option is omitted.
`[ -keystore` *`keystore-file`* `]`	Location of the keystore file into which to import the contents of the PKCS-12 file. If no value is given, defaults to `${`*`user-home`*`}/.keystore`.
`[ -storepass` *`store-password`* `]`	The password of the keystore. User is prompted for the password of the truststore if this option is omitted.
`[ -keypass` *`key-password`* `]`	The password to be used to protect the private key inside the keystore. The user is prompted for this password if this option is omitted.
`[ -alias` *`alias`* `]`	The alias to be used to store the key entry (private key and the certificate) inside the keystore.

`keyexport`

This tool is used to export a private key in a keystore (typically of type Java Keystore (JKS)) into a file.

Note: The exported private key is not secured with a password, so it should be handled carefully. For example, you can export a private key from a keystore and use it to sign certificate requests obtained through any means using other key/certificate management tools. These certificate requests are then sent to a certificate authority for validation and certificate generation.

The `keyexport` tool can be found in the directory `<`*`JWSDP_HOME`*`>/xws-security/bin/`, and can be run from the command line by executing `keyexport.sh` (on Unix systems) or `keyexport.bat` (on Windows systems). The options for this tool are listed in Table 4–43.

Table 4–43 Options for `keyexport` tool

Option	**Description**
`-keyfile` *`key-file`*	Required. The location of the file to which the private key will be exported.
`[ -outform` *`output-format`* `]`	This specifies the output format. The options are DER and PEM. The DER format is the DER encoding (binary format) of the certificate. The PEM format is the base64-encoding of the DER encoding with header and footer lines added.
`[ -keystore` *`keystore-file`* `]`	Location of the keystore file containing the key. If no value is given, this option defaults to `${`*`user-home`*`}/.keystore`.
`[ -storepass` *`store-password`* `]`	Password of the keystore. User is prompted for the password if this option is omitted.
`[ -keypass` *`key-password`* `]`	The password used to protect the private key inside the keystore. User is prompted for the password if this option is omitted.
`[ -alias` *`alias`* `]`	The alias of the key entry inside the keystore.

`wscompile`

The `wscompile` tool generates the client stubs and server-side ties for the service definition interface that represents the Web service interface. Additionally, it generates the WSDL description of the Web service interface which is then used to generate the implementation artifacts.

XWS-Security has been integrated into JAX-RPC through the use of security configuration files. The code for performing the security operations on the client and server is generated by supplying the configuration files to the JAX-RPC `wscompile` tool. The `wscompile` tool can be instructed to generate security code by making us of the `-security` option to specify the location of the security configuration file that contains information on how to secure the messages to be sent. An example of using the `-security` option with `wscompile` is shown in How Do I Specify the Security Configuration for the Build Files?.

Note: For the 2.0 release of JAX-RPC, JAX-RPC will be renamed to JAX-WS. JAX-WS will become part of the XWS-Security 2.0 FCS later this year. When this renaming occurs, the `wscompile` tool will be replaced, and these steps and the `build.xml` files for the sample applications will need to be modified accordingly.

The syntax for this option is as follows:

```
wscompile [-security {location of security configuration
file}]
```

For more description of the `wscompile` tool, its syntax, and examples of using this tool, read:
`http://docs.sun.com/source/817-6092/hman1m/wscompile.1m.html`

Troubleshooting XWS-Security Applications

This section lists some possible errors and the possible causes for these errors. For more troubleshooting information, read the online release notes at http://java.sun.com/webservices/docs/1.6/xws-security/ReleaseNotes.html.

Error: at XMLCipher.getInstance (Unknown Source)

```
[java] Exception in thread "main"
java.lang.NullPointerException
[java] at
com.sun.org.apache.xml.security.encryption.XMLCipher.getInstan
ce(Unknown Source)
```

Solution: Configure a JCE provider as described in Configuring a JCE Provider.

Error: UnsupportedClassVersionError

```
java.lang.UnsupportedClassVersionError: com/sun/tools/javac/
Main (Unsupported major.minor version 49.0)
```

Solution: Install version 1.4.2_04 or higher of Java 2 Standard Edition (J2SE). If you had an older version of the JDK, you will also have to reinstall the Application Server so that it recognizes this as the default version of the JDK.

Error: DeployTask not found

Solution: Verify that the `jwsdp.home` property in the `build.properties` file for the sample is set correctly to the location where you installed the Java WSDP version 1.6, as described in Setting Build Properties. A common error is to not escape the backslash character when running on the Microsoft Windows platform.

Compiler Errors

If you use a version of the Application Server prior to 2005Q1 for the container, you may get compiler errors because this version of the Application Server has an earlier version of XWS-Security bundled into it. The compilation errors that you see are because these classes do not exist in the earlier version of XWS-Security shipped in these earlier versions of the Application Server.

Further Information

- Java 2 Standard Edition, v.1.5.0 security information
  http://java.sun.com/j2se/1.5.0/docs/guide/security/index.html
- Java Servlet specification
  http://java.sun.com/products/servlet/
- Information on SSL specifications
  http://wp.netscape.com/eng/security/
- OASIS Standard 200401: Web Services Security: SOAP Message Security 1.0
  http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf
- XML Encryption Syntax and Processing
  http://www.w3.org/TR/xmlenc-core/
- Digital Signatures Working Draft
  http://www.w3.org/Signature/
- JSR 105-XML Digital Signature APIs
  http://www.jcp.org/en/jsr/detail?id=105
- JSR 106-XML Digital Encryption APIs
  http://www.jcp.org/en/jsr/detail?id=106
- Public-Key Cryptography Standards (PKCS)
  http://www.rsasecurity.com/rsalabs/pkcs/index.html
- Java Authentication and Authorization Service (JAAS)
  http://java.sun.com/products/jaas/
- WS-I Basic Security Profile Version 1.0
  http://www.ws-i.org/Profiles/BasicSecurityProfile-1.0-2005-01-20.html
- Web Services Security: SOAP Messages with Attachments (SwA) Profile 1.0
  http://www.oasis-open.org/committees/download.php/10090/wss-swa-profile-1.0-draft-14.pdf
- Web Services Security: SOAP Messages with Attachments (SwA) Profile 1.0, Interop 1 Scenarios
  http://lists.oasis-open.org/archives/wss/200410/pdf00003.pdf
- Web Services Security: Security Assertion Markup Language (SAML) Token Profile 1.0
  http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.0.pdf
- Web Services Security: Security Assertion Markup Language (SAML) Interop Scenarios

http://www.oasis-open.org/apps/org/workgroup/wss/download.php/7011/wss-saml-interop1-draft-11.doc

5

Understanding and Running the XWS-Security Sample Applications

THIS addendum discusses the XML and Web Services Security (XWS-Security) sample applications that are shipped with Java WSDP 1.6. For each of the sample applications, there is an explanation of what is being demonstrated, how the application is secured, and how to compile and run the application.

Introduction to XML and Web Services Security provides an introduction to how to use XWS-Security in this release. Setting Up To Use XWS-Security With the Sample Applications provides information on how to configure your system to run the sample applications.

The following sample applications are discussed in this chapter:

- Simple Security Configurations Sample Application
- JAAS Sample Application
- XWS-Security APIs Sample Application
- Soap With Attachments Sample Application
- SAML Sample Application
- Dynamic Policy Sample Application
- Dynamic Response Sample Application

Setting Up To Use XWS-Security With the Sample Applications

This addendum discusses creating and running applications that use the XWS-Security framework, and deploying these applications onto the Sun Java System Application Server Platform Edition 8.1. For deployment onto other containers, read the `README.txt` file for the example applications for more information.

Follow these steps to set up your system to compile, run, and deploy the sample applications included in this release that use the XWS-Security framework.

1. Make sure that you have installed the Java 2 Platform, Standard Edition version 1.4.2 or higher. If not, you can download the JDK from the following URL:

   http://java.sun.com/j2se/

   If you are using version 1.4.x of the Java SDK, configure a version of a JCE provider that supports RSA encryption. Information on doing this is discussed in Configuring a JCE Provider.
2. Make sure that you have a container installed. For more information on containers, read http://java.sun.com/webservices/containers/index.html.
3. Make sure that you have installed Java WSDP 1.6. If not, you can download the JWSDP from the following URL:

   http://java.sun.com/webservices/jwsdp/index.jsp
4. Set system properties as described in Setting System Properties.
5. Read the information in Setting Up the Application Server For the Examples.

Setting System Properties

The asant (or ant) build files for the XWS-Security samples shipped with this release rely on certain environment variables being set correctly. Make sure that the following environment variables are set to the locations specified in this list. If you are not sure how to set these environment variables, refer to the file <*JWSDP_HOME*>/xws-security/docs/samples.html for more specific information. This file includes instructions for both the Unix and Microsoft Windows platforms. Throughout this document, instructions for running on the Unix platform will be provided.

1. Set JAVA_HOME to the location of your J2SE installation directory, for example, /home/<*your_name*>/j2sdk1.4.2_04/.
2. Set JWSDP_HOME to the location of your Java WSDP 1.6 installation directory, for example, /home/<*your_name*>/jwsdp-1.6/.
3. Set SJSAS_HOME to the location of your Application Server installation directory, for example, /home/<*your_name*>/SUNWappserver/. If you are deploying onto a different container, set SJSWS_HOME or TOMCAT_HOME instead.
4. Set ANT_HOME to the location where the asant (or ant) executable can be found. If you are running on the Application Server, this will be <*SJSAS_HOME*>/bin/. If you are running on a different container, this location will probably be <*JWSDP_HOME*>/apache-ant/bin/.
5. Set the PATH variable so that it contains these directories: <*JWSDP_HOME*>/jwsdp-shared/bin/, <*SJSAS_HOME*>/bin/, <*ANT_HOME*>/, and <*JAVA_HOME*>/bin/.

Configuring a JCE Provider

You only need to perform the steps in this section if you are running Java WSDP 1.6 on J2SE 1.4.x.

The Java Cryptography Extension (JCE) provider included with J2SE 1.4.x does not support RSA encryption. Because the XWS-Security sample applications use RSA encryption, you must download and install a JCE provider that does support RSA encryption in order for these sample applications to run, if you are using encryption, and if you are using a version of the Java SDK prior to version 1.5.0.

Note: RSA is public-key encryption technology developed by RSA Data Security, Inc. The acronym stands for Rivest, Shamir, and Adelman, the inventors of the technology.

If you are running the Application Server on version 1.5 of the Java SDK, the JCE provider is already configured properly. If you are running the Application Server on version 1.4.x of the Java SDK, follow these steps to add a JCE provider statically as part of your JDK environment:

1. Download and install a JCE provider JAR (Java ARchive) file. The following URL provides a list of JCE providers that support RSA encryption:

   http://java.sun.com/products/jce/jce14_providers.html
2. Copy the JCE provider JAR file to `<`*`JAVA_HOME`*`>/jre/lib/ext/`.
3. Stop the Application Server (or other container). If the Application Server is not stopped, and restarted later in this process, the JCE provider will not be recognized by the Application Server.
4. Edit the `<`*`JAVA_HOME`*`>/jre/lib/security/java.security` properties file in any text editor. Add the JCE provider you've just downloaded to this file. The `java.security` file contains detailed instructions for adding this provider. Basically, you need to add a line of the following format in a location with similar properties:

   `security.provider.<`*`n`*`>=<`*`provider class name`*`>`

   In this example, <*n*> is the order of preference to be used by the Application Server when evaluating security providers. Set <*n*> to `2` for the JCE provider you've just added.

   For example, if you've downloaded ABC JCE provider, and the Java class name of the ABC provider's main class is `org.abc.ABCProvider`, add this line.

   ```
   security.provider.2=org.abc.ABCProvider
   ```

   Make sure that the Sun security provider remains at the highest preference, with a value of 1.

   ```
   security.provider.1=sun.security.provider.Sun
   ```

   Adjust the levels of the other security providers downward so that there is only one security provider at each level.

The following is an example of a `java.security` file that provides the necessary JCE provider and keeps the existing providers in the correct locations.

```
          security.provider.1=sun.security.provider.Sun
          security.provider.2=org.abc.ABCProvider
          security.provider.3=com.sun.net.ssl.internal.ssl.P
rovider
          security.provider.4=com.sun.rsajca.Provider
          security.provider.5=com.sun.crypto.provider.SunJCE
          security.provider.6=sun.security.jgss.SunProvider
```

5. Save and close the file.
6. Restart the Application Server (or other container).

Setting Up the Application Server For the Examples

To set up the container for running the XWS-Security sample applications included with this release, you need to specify on which container you are running the `asant` (or `ant`) build targets (as discussed in Setting Build Properties), and you must point the container to the keystore and truststore files to be used to run the XWS-Security sample applications. For the sample applications, these are the keystore and truststore files included in the `/xws-security/etc/` directory. For further discussion of using keystores and truststores with XWS-Security applications, read Keystore and Truststore Files with XWS-Security.

Keystore and Truststore Files with XWS-Security

For the `simple` sample, the keystore, truststore, and symmetric-key databases used by that example are located in the `<JWSDP_HOME>/xws-security/etc/` directory. The locations of these files have been configured in the `<JWSDP_HOME>/xws-security/etc/client-security-env.properties` and `<JWSDP_HOME>/xws-security/etc/server-security-env.properties` files for the client and server respectively. These property files are used by the `SecurityEnvironmentHandler` to handle the `Callbacks`.

To plug in your own keystores and truststores for an application, make sure that the certificates are of version 3, and that the client truststore contains the certificate of the certificate authority that issued the server's certificate, and vice versa.

XWS-Security requires version 3 (v3) certificates when the `keyReferenceType` attribute (specified on a `xwss:X509Token` element) has a value of `Identifier`, which indicates the use of an X.509 `SubjectKeyIdentifier` extension. For all other values of the `keyReferenceType` attribute, a v1 certificate can also be used. Version 3 includes requirements specified by the WSS X509 Token Profile.

Setting Build Properties

To run the sample applications, you must edit the sample `build.properties` file for that sample application and specify information that is unique to your system and to your installation of Java WSDP 1.6 and the Application Server (or other container).

To edit the `build.properties` file for the example you want to run, follow these steps:

1. Change to the directory for the sample application you want to run: `<`*`JWSDP_HOME`*`>/xws-security/samples/<`*`example`*`>/`.
2. Copy the `build.properties.sample` file to `build.properties`.
3. Edit the `build.properties` file, checking that the following properties (where applicable) are set correctly for your system:
    - `javahome`: Set this to the directory where J2SE version 1.4.2 or higher is installed.

Note: When running on Microsoft Windows, you must escape any backslashes in the `javahome`, `jwsdp.home`, and `sjsas.home` properties with another backslash or use forward slashes as a path separator. So, for example, if your Application Server installation is `C:\Sun\AppServer`, you must set `sjsas.home` as follows:

```
sjsas.home = C:\\Sun\\AppServer
```

or

```
sjsas.home=C:/Sun/AppServer
```

- `sjsas.home`: To specify that you are running under the Application Server, set this property to the directory where the Application Server is installed and make sure there is not a comment symbol (`#`) to the left of this entry. If you are running under a different container, set the location for its install directory under the appropriate property name (`tom-`

`cat.home` or `sjsws.home`) and uncomment that entry instead. Only one of the container home properties should be uncommented at any one time.

- `username`, `password`: Enter the appropriate username and password values for a user assigned to the role of `admin` for the container instance being used for this sample. A user with this role is authorized to deploy applications onto the Application Server.
- `endpoint.host`, `endpoint.port`: If you changed the default host and/or port during installation of the Application Server (or other container), change these properties to the correct values for your host and port. If you installed the Application Server using the default values, these properties will already be set to the correct values.
- `VS.DIR`=If you are running under the Sun Java System Web Server, enter the directory for the virtual server. If you are running under any other container, you do not need to modify this property.
- `jwsdp.home`: Set this property to the directory where Java WSDP is installed. The keystore and truststore URL's for the client are configured relative to this property.
- `http.proxyHost`, `http.proxyPort`: If you are using remote endpoints, set these properties to the correct proxy server address and port. If you are not using remote endpoints, put a comment character (`#`) before these properties. A proxy server will follow the format of `myserver.mycompany.com`. The proxy port is the port on which the proxy host is running, for example, `8080`.

4. Save and exit the `build.properties` file.

Simple Security Configurations Sample Application

The `simple` sample application is a fully-developed sample application that demonstrates various configurations that can be used to exercise XWS-Security framework code. To change the type of security that is being used for the client and/or the server, simply modify two properties in the `build.properties` file for the example. The types of security configurations possible in this example include XML Digital Signature, XML Encryption, `UserNameToken` verification, and combinations thereof. This example allows and demonstrates combinations of these basic security mechanisms through the specification of the appropriate security configuration files.

The application prints out both the client and server request and response SOAP messages. The output from the server may be viewed in the appropriate container's log file. The output from the client is sent to `stdout` or whichever stream is used by the configured log handler. Messages are logged at the `INFO` level.

In this example, server-side code is found in the `/simple/server/src/simple/` directory. Client-side code is found in the `/simple/client/src/simple/` directory. The `asant` (or `ant`) targets build objects under the `/build/server/` and `/build/client/` directories.

This example uses keystores and truststores which are included in the `/xws-security/etc/` directory. For more information on using keystore and truststore files, read the `keytool` documentation at the following URL:
http://java.sun.com/j2se/1.5.0/docs/tooldocs/solaris/keytool.html

Plugging in Security Configurations

This example makes it simple to plug in different client and server-side configurations describing security settings. This example has support for digital signatures, XML encryption/decryption, and username/token verification. This example allows and demonstrates combinations of these basic security mechanisms through configuration files. See Simple Sample Security Configuration Files, for further description of the security configuration options defined for the `simple` sample application.

To specify which security configuration option to use when the sample application is run (see Running the Simple Sample Application), follow these steps:

1. Open the `build.properties` file for the example. This file is located at `<`*JWSDP_HOME*`>/xws-security/samples/simple/build.properties`.
2. To set the security configuration that you want to run for the client, locate the `client.security.config` property, and uncomment one of the client security configuration options. The client configuration options are listed in Simple Sample Security Configuration Files, and also list which client and server configurations work together. For example, if you want to use XML Encryption for the client, you would uncomment this option:

   ```
   # Client Security Config. file
   client.security.config=config/encrypt-client.xml
   ```

   Be sure to uncomment only one client security configuration at a time.

3. To set the security configuration that you want to run for the server, locate the `server.security.config` property, and uncomment one of the server security configuration options. The server configuration options, and which server options are valid for a given client configuration, are listed in Simple Sample Security Configuration Files. For example, if you want to use XML Encryption for the server, you would uncomment this option:

   ```
   # Server Security Config. file
   server.security.config=config/encrypt-server.xml
   ```

   Be sure to uncomment only one client security configuration at a time.
4. Save and exit the `build.properties` file.
5. Run the sample application as described in Running the Simple Sample Application.

Simple Sample Security Configuration Files

The configuration files available for this example are located in the `/xws-security/samples/simple/config/` directory. The configuration pairs available under this sample include configurations for both the client and server side. Some possible combinations are discussed in more detail in the referenced sections.

Dumping the Request and/or the Response

The security configuration pair `dump-client.xml` and `dump-server.xml` have no security operations. These options enable the following tasks:

- Dump the request before it leaves the client.
- Dump the response upon receipt from the server.

The container's server logs also contain the dumps of the server request and response. See Running the Simple Sample Application for more information on viewing the server logs.

Encrypting the Request and/or the Response

The security configuration pair `encrypt-client.xml` and `encrypt-server.xml` enable the following tasks:

- Client encrypts the request body and sends it.
- Server decrypts the request and sends back a response.

The `encrypt-client.xml` file looks like this:

```
<xwss:JAXRPCSecurity xmlns:xwss="http://java.sun.com/xml/ns/
xwss/config">

    <xwss:Service>
        <xwss:SecurityConfiguration dumpMessages="true">
            <!--
              Since no targets have been specified below, the
contents of
              the soap body would be encrypted by default.
            -->
            <xwss:Encrypt>
                <xwss:X509Token certificateAlias="s1as"/>
            </xwss:Encrypt>
        </xwss:SecurityConfiguration>
    </xwss:Service>

    <xwss:SecurityEnvironmentHandler>
          sample.SecurityEnvironmentHandler
    </xwss:SecurityEnvironmentHandler>

</xwss:JAXRPCSecurity>
```

Signing and Verifying the Signature

The security configuration pair `sign-client.xml` and `sign-server.xml` enable the following tasks:

- Client signs the request body.
- Server verifies the signature and sends its response.

The `sign-client.xml` file looks like this:

```
<xwss:JAXRPCSecurity xmlns:xwss="http://java.sun.com/xml/ns/
xwss/config">

    <xwss:Service>
```

```
        <xwss:SecurityConfiguration dumpMessages="true">
            <!--
             Note that in the <Sign> operation, a Timestamp is
exported
              in the security header and signed by default.
            -->
            <xwss:Sign>
               <xwss:X509Token certificateAlias="xws-security-
client"/>
            </xwss:Sign>
            <!--
              Signature requirement. No target is specified,
hence the
              soap body is expected to be signed. Also, by
default, a
              Timestamp is expected to be signed.
            -->
            <xwss:RequireSignature/>
        </xwss:SecurityConfiguration>
   </xwss:Service>

    <xwss:SecurityEnvironmentHandler>
          sample.SecurityEnvironmentHandler
    </xwss:SecurityEnvironmentHandler>

</xwss:JAXRPCSecurity>
```

Signing then Encrypting the Request, Decrypting then Verifying the Signature

The security configuration pair `sign-encrypt-client.xml` and `sign-encrypt-server.xml` enable the following tasks:

- Client signs and then encrypts and sends the request body.
- Server decrypts and verifies the signature.
- Server signs and then encrypts and sends the response.

The `sign-encrypt-client.xml` file looks like this:

```
<xwss:JAXRPCSecurity xmlns:xwss="http://java.sun.com/xml/ns/
xwss/config">

    <xwss:Service>
        <xwss:SecurityConfiguration dumpMessages="true">
            <xwss:Sign/>
            <xwss:Encrypt>
```

```
                <xwss:X509Token certificateAlias="s1as"
keyReferenceType="Identifier"/>
            </xwss:Encrypt>
            <!--
              Requirements on messages received:
            -->
            <xwss:RequireEncryption/>
            <xwss:RequireSignature/>
        </xwss:SecurityConfiguration>
    </xwss:Service>

    <xwss:SecurityEnvironmentHandler>
        sample.SecurityEnvironmentHandler
    </xwss:SecurityEnvironmentHandler>

</xwss:JAXRPCSecurity>
```

Encrypting then Signing the Request, Verifying then Decrypting the Signature

The security configuration pair `encrypt-sign-client.xml` and `encrypt-sign-server.xml` enable the following tasks:

- Client encrypts the request body, then signs and sends it.
- Server verifies the signature and then decrypts the request body.
- Server sends its response.

The `encrypt-sign-client.xml` file looks like this:

```
<xwss:JAXRPCSecurity xmlns:xwss="http://java.sun.com/xml/ns/
xwss/config">

    <xwss:Service>
        <xwss:SecurityConfiguration dumpMessages="true">
            <!--
              First encrypt the contents of the soap body
            -->
            <xwss:Encrypt>
                <xwss:X509Token keyReferenceType="Identifier"
certificateAlias="s1as"/>
            </xwss:Encrypt>
            <!--
              Secondly, sign the soap body using some default
private key.
            The sample CallbackHandler implementation has code
to handle
```

```
              the default signature private key request.
            -->
            <xwss:Sign/>
        </xwss:SecurityConfiguration>
    </xwss:Service>

    <xwss:SecurityEnvironmentHandler>
        sample.SecurityEnvironmentHandler
    </xwss:SecurityEnvironmentHandler>

</xwss:JAXRPCSecurity>
```

Signing a Ticket

The security configuration pair `sign-ticket-also-client.xml` and `sign-ticket-also-server.xml` enable the following tasks:

- Client signs the ticket element, which is inside the message body.
- Client signs the message body.
- Server verifies signatures.

The `sign-ticket-also-client.xml` file looks like this:

```
<xwss:JAXRPCSecurity xmlns:xwss="http://java.sun.com/xml/ns/
xwss/config">

    <xwss:Service>
        <xwss:SecurityConfiguration dumpMessages="true">
            <!--
              Signing multiple targets as part of the same
ds:Signature
              element in the security header
            -->
            <xwss:Sign>
                <xwss:Target type="qname">{http://xmlsoap.org/
Ping}ticket</xwss:Target>
            <xwss:Target type="xpath">//env:Body</xwss:Target>
            </xwss:Sign>
        </xwss:SecurityConfiguration>
    </xwss:Service>

    <xwss:SecurityEnvironmentHandler>
        sample.SecurityEnvironmentHandler
    </xwss:SecurityEnvironmentHandler>

</xwss:JAXRPCSecurity>
```

Adding a Timestamp to a Signature

The security configuration pair `timestamp-sign-client.xml` and `timestamp-sign-server.xml` enable the following tasks:

- Client signs the request, including a timestamp in the request.

The `timestamp-sign-client.xml` file looks like this:

```
<xwss:JAXRPCSecurity xmlns:xwss="http://java.sun.com/xml/ns/
xwss/config">

    <xwss:Service>
        <xwss:SecurityConfiguration dumpMessages="true">
            <!--
              Export a Timestamp with the specified timeout
interval (in sec).
            -->
            <xwss:Timestamp timeout="120"/>
            <!--
           The above Timestamp would be signed by the following
Sign
              operation by default.
            -->
            <xwss:Sign>
                <xwss:Target type="qname">{http://xmlsoap.org/
Ping}ticket</xwss:Target>
            </xwss:Sign>
        </xwss:SecurityConfiguration>
    </xwss:Service>

    <xwss:SecurityEnvironmentHandler>
        sample.SecurityEnvironmentHandler
    </xwss:SecurityEnvironmentHandler>

</xwss:JAXRPCSecurity>
```

Symmetric Key Encryption

The security configuration pair `encrypt-using-symmkey-client.xml` and `encrypt-server.xml` enable the following tasks:

- Client encrypts the request using the specified symmetric key.

This is a case where the client and server security configuration files do not match. This combination works because the server requirement is the same (the body contents must be encrypted) when the client-side security configuration is either `encrypt-using-symmkey-client.xml` or `encrypt-client.xml`. The difference in the two client configurations is the key material used for encryption.

The `encrypt-using-symmkey-client.xml` file looks like this:

```
<xwss:JAXRPCSecurity xmlns:xwss="http://java.sun.com/xml/ns/
xwss/config">

    <xwss:Service>
        <xwss:SecurityConfiguration dumpMessages="true">
            <!--
             Encrypt using a symmetric key associated with the
given alias
            -->
            <xwss:Encrypt>
                <xwss:SymmetricKey keyAlias="sessionkey"/>
            </xwss:Encrypt>
        </xwss:SecurityConfiguration>
    </xwss:Service>

    <xwss:SecurityEnvironmentHandler>
        sample.SecurityEnvironmentHandler
    </xwss:SecurityEnvironmentHandler>

</xwss:JAXRPCSecurity>
```

Adding a Username Password Token

The security configuration pair `user-pass-authenticate-client.xml` and `user-pass-authenticate-server.xml` enable the following tasks:

- Client adds a username-password token and sends a request.
- Server authenticates the username and password against a username-password database.
- Server sends response.

The `user-pass-authenticate-client.xml` file looks like this:

```
<xwss:JAXRPCSecurity xmlns:xwss="http://java.sun.com/xml/ns/
xwss/config">

    <xwss:Service>
        <xwss:SecurityConfiguration dumpMessages="true">
            <!--
              Default: Digested password will be sent.
            -->
            <xwss:UsernameToken name="Ron" password="noR"/>
        </xwss:SecurityConfiguration>
    </xwss:Service>

    <xwss:SecurityEnvironmentHandler>
        sample.SecurityEnvironmentHandler
    </xwss:SecurityEnvironmentHandler>

</xwss:JAXRPCSecurity>
```

Encrypt Request Body and a UserNameToken

The security configuration pair `encrypt-usernameToken-client.xml` and `encrypt-usernameToken-server.xml` enable the following tasks:

- Client encrypts request body.
- Client encrypts the `UsernameToken` as well before sending the request.
- Server decrypts the encrypted message body and encrypted `UsernameToken`.
- Server authenticates the user name and password against a username-password database.

The `encrypt-usernameToken-client.xml` file looks like this:

```
<xwss:JAXRPCSecurity xmlns:xwss="http://java.sun.com/xml/ns/
xwss/config">

    <xwss:Service>
        <xwss:SecurityConfiguration dumpMessages="true">
            <!--
             Export a username token into the security header.
Assign it
               the mentioned wsu:Id
            -->
            <xwss:UsernameToken name="Ron" password="noR"
id="username-token"/>
            <xwss:Encrypt>
                <xwss:X509Token certificateAlias="s1as"/>
                <xwss:Target type="xpath">//SOAP-ENV:Body</
xwss:Target>
                <!--
                  The username token has been refered as an
encryption
                  target using a URI fragment
                -->
                <xwss:Target type="uri">#username-token</
```

```
xwss:Target>
            </xwss:Encrypt>
        </xwss:SecurityConfiguration>
    </xwss:Service>

    <xwss:SecurityEnvironmentHandler>
        sample.SecurityEnvironmentHandler
    </xwss:SecurityEnvironmentHandler>

</xwss:JAXRPCSecurity>
```

In this sample, the `UsernameToken` is assigned an id `username-token`. This id is used to refer to the token as an encryption target within the `<xwss:Encrypt>` element. The id becomes the actual `wsu:id` of the `UsernameToken` in the generated `SOAPMessage`.

Adding a UserName Password Token, then Encrypting the UserName Token

The security configuration pair `encrypted-user-pass-client.xml` and `encrypted-user-pass-server.xml` enable the following tasks:

- Client adds a `UsernameToken`.
- Client encrypts the `UsernameToken` before sending the request.
- Server decrypts the `UsernameToken`.
- Server authenticates the user name and password against a username-password database.

The `encrypted-user-pass-client.xml` file looks like this:

```
<xwss:JAXRPCSecurity xmlns:xwss="http://java.sun.com/xml/ns/
xwss/config">

    <xwss:Service>
        <xwss:SecurityConfiguration dumpMessages="true">
            <xwss:UsernameToken name="Ron" password="noR"/>
            <xwss:Encrypt>
                <xwss:X509Token certificateAlias="s1as"
keyReferenceType="Identifier"/>
                <xwss:Target type="qname">
                    {http://docs.oasis-open.org/wss/2004/01/
oasis-200401-wss-wssecurity-
                        secext-1.0.xsd}UsernameToken
                </xwss:Target>
            </xwss:Encrypt>
        </xwss:SecurityConfiguration>
    </xwss:Service>

    <xwss:SecurityEnvironmentHandler>
        sample.SecurityEnvironmentHandler
    </xwss:SecurityEnvironmentHandler>

</xwss:JAXRPCSecurity>
```

Flexibility in Positions of Timestamps and Tokens

The security configuration pair `flexiblec.xml` and `flexibles.xml` demonstrate the flexibility in the position of Timestamps and tokens allowed in the receiver-side processing of a message. The tokens that can be used include `UsernameToken`, `BinarySecurityToken`, `SAMLAssertion`, and others. The position of <`Require`*`XXX`*> elements for these tokens can vary in the receiver-side configuration file regardless of the position of the tokens in the incoming message.

This flexibility does not apply to the relative position of `Signature` and `EncryptedData` elements in the incoming message, which have to follow the strict order in which the `<RequireSignature>` and `<RequireEncryption>` elements appear in the configuration file.

The `flexiblec.xml` file looks like this:

```
<xwss:JAXRPCSecurity xmlns:xwss="http://java.sun.com/xml/ns/
xwss/config">

   <xwss:Service>
      <xwss:SecurityConfiguration dumpMessages="true"
      xmlns:xwss="http://java.sun.com/xml/ns/xwss/config">
         <xwss:Sign includeTimestamp="false"/>
         <xwss:UsernameToken name="Ron" password="noR"
useNonce="true"
         digestPassword="false"/>
         <xwss:Timestamp timeout="300"/>
      </xwss:SecurityConfiguration>
   </xwss:Service>

   <xwss:SecurityEnvironmentHandler>
```

```
        sample.SecurityEnvironmentHandler
    </xwss:SecurityEnvironmentHandler>

</xwss:JAXRPCSecurity>
```

Adding Security at the Method Level

The security configuration pair `method-level-client.xml` and `method-level-server.xml` enable the following tasks:

- Configures different security policies for different WSDL methods of the application and different port instances.

The `simple` sample's WSDL file contains two operations, `Ping` and `Ping0`, and two port instances of type `PingPort`. The port names are `Ping` and `Ping0`. The method level security configuration file demonstrates how different sets of security operations can be configured for the operations `Ping` and `Ping0` under each of the two `Port` instances `Ping` and `Ping0`.

The `method-level-client.xml` file looks like this:

```
<xwss:JAXRPCSecurity xmlns:xwss="http://java.sun.com/xml/ns/
xwss/config">
    <xwss:Service>
          <!--
            Service-level security configuration
            -->
          <xwss:SecurityConfiguration dumpMessages="true">
              <xwss:Encrypt>
                  <xwss:X509Token certificateAlias="s1as"/>
              </xwss:Encrypt>
          </xwss:SecurityConfiguration>
          <xwss:Port name="{http://xmlsoap.org/Ping}Ping">
              <!--
             Port-level security configuration. Takes precedence
over the
                service-level security configuration
              -->
              <xwss:SecurityConfiguration dumpMessages="true"/>
           <xwss:Operation name="{http://xmlsoap.org/Ping}Ping">
                  <!--
                  Operation-level security configuration. Takes
precedence
                    over port-level and service-level security
configurations.
                  -->
                <xwss:SecurityConfiguration dumpMessages="true">
```

```
                    <xwss:UsernameToken name="Ron"
                                        password="noR"
                                        digestPassword="false"
                                        useNonce="false"/>
                    <xwss:Sign>
                        <xwss:Target type="qname">{http://
xmlsoap.org/Ping}ticket</xwss:Target>
                        <xwss:Target type="qname">{http://
xmlsoap.org/Ping}text</xwss:Target>
                    </xwss:Sign>
                    <xwss:Encrypt>
                     <xwss:X509Token certificateAlias="s1as"/>
                    </xwss:Encrypt>
                </xwss:SecurityConfiguration>
            </xwss:Operation>

            <xwss:Operation name="{http://xmlsoap.org/
Ping}Ping0">
              <xwss:SecurityConfiguration dumpMessages="true">
                    <xwss:Encrypt>
                     <xwss:X509Token certificateAlias="s1as"/>
                    </xwss:Encrypt>
                </xwss:SecurityConfiguration>
            </xwss:Operation>
         </xwss:Port>

        <xwss:Port name="{http://xmlsoap.org/Ping}Ping0">
            <xwss:SecurityConfiguration dumpMessages="true">
                <xwss:Encrypt>
                    <xwss:X509Token certificateAlias="s1as"/>
                </xwss:Encrypt>
                <xwss:RequireSignature/>
            </xwss:SecurityConfiguration>
               <xwss:Operation name="{http://xmlsoap.org/
Ping}Ping"/>
               <xwss:Operation name="{http://xmlsoap.org/
Ping}Ping0"/>
         </xwss:Port>
      </xwss:Service>
<xwss:SecurityEnvironmentHandler>
        sample.SecurityEnvironmentHandler
    </xwss:SecurityEnvironmentHandler>
</xwss:JAXRPCSecurity>
```

In this example, the following has been configured for the `Ping` operation under port instance `Ping`:

- Inserts a `UsernameToken` into the request.
- Signs the ticket and text child elements of the request body.
- Encrypts the contents of the request body.

The following has been configured for the `Ping0` operation under port instance `Ping`:

- Encrypt the content of the body of the message.

When the `xwss:Encrypt` element is specified with no child elements of type `xwss:Target`, it implies that the default `Target` (which is `SOAP-ENV:Body`) has to be encrypted. The same rule applies to `xwss:Sign` elements with no child elements of type `xwss:Target`.

The configuration file in this example also configures the following security for all the WSDL operations under port instance `Ping0`:

- Encrypts the request body.
- Expects a signed response from the server.Username

Running the Simple Sample Application

To run the `simple` sample application, follow these steps:

1. Complete the tasks defined in the following sections of this addendum:
   - Setting System Properties
   - Configuring a JCE Provider
   - Setting Build Properties
2. Start the selected container and make sure the server is running. To start the Application Server,
   a. From a Unix machine, enter the following command from a terminal window: `asadmin start-domain domain1`
   b. From a Windows machine, choose Start→Programs→Sun Microsystems→J2EE 1.4→Start Default Server.
3. Modify the `build.properties` file to set up the security configuration that you want to run for the client and/or server. See Simple Sample Security Configuration Files for more information on the security configurations options that are already defined for the sample application.

4. Build and run the application from a terminal window or command prompt.
    - On the Application Server, the command to build and run the application is: `asant run-sample`
    - On the other containers, the command to build and run the application is: `ant run-sample`

Note: To run the sample against a remote server containing the deployed endpoint, use the `run-remote-sample` target in place of the `run-sample` target. In this situation, make sure that the `endpoint.host`, `endpoint.port`, `http.proxyHost`, `http.proxyPort`, and `service.url` properties are set correctly in the `build.properties` file (as discussed in Setting Build Properties) before running the sample.

If the application runs successfully, you will see a message similar to the following:

```
[echo] Running the client program....
[java] ==== Sending Message Start ====
...
[java] ==== Sending Message End ====
[java] ==== Received Message Start ====
...
[java] ==== Received Message End ====
```

You can view similar messages in the server logs:

```
<SJSAS_HOME>/domains/<domain-name>/logs/server.log
<TOMCAT_HOME>/logs/launcher.server.log
<SJSWS_HOME>/<Virtual-Server-Dir>/logs/errors
```

JAAS Sample Application

The Java Authentication and Authorization Service (JAAS) is a set of APIs that enable services to authenticate and enforce access controls upon users. It implements a Java technology version of the standard Pluggable Authentication Module (PAM) framework, and supports user-based authorization.

The <*JWSDP_HOME*>/xws-security/samples/jaas-sample application demonstrates the following functionality:

- Obtaining a user name and password at run-time and sending it in a Web Services Security (WSS) UsernameToken to the server.
- Using JAAS authentication to authenticate the user name and password in the server application.
- Accessing the authenticated sender's subject from within the endpoint implementation methods.

The application prints out both the client and server request and response SOAP messages. The output from the server may be viewed in the appropriate container's log file. The output from the client is sent to stdout or whichever stream is used by the configured log handler. Messages are logged at the INFO level.

In this example, server-side code is found in the /jaas-sample/server/src/jaas-sample/ directory. Client-side code is found in the /jaas-sample/client/src/jaas-sample/ directory. The asant (or ant) targets build objects under the /build/server/ and /build/client/ directories.

JAAS Sample Security Configuration Files

The security configuration pair user-pass-authenticate-client.xml and user-pass-authenticate-server.xml enable the following tasks:

- Client adds a username-password token and sends a request.
- Server authenticates the username and password against a username-password database.
- Server sends response.

The username-password database must be set up before this security configuration pair will run properly. Refer to Setting Up For the JAAS-Sample for instructions on setting up this database.

The user-pass-authenticate-client.xml file looks like this:

```
<xwss:JAXRPCSecurity xmlns:xwss="http://java.sun.com/xml/ns/
xwss/config">

    <xwss:Service>
        <xwss:SecurityConfiguration dumpMessages="true">
```

```
                <xwss:UsernameToken digestPassword="false"/>
            </xwss:SecurityConfiguration>
        </xwss:Service>

        <xwss:SecurityEnvironmentHandler>
            sample.ClientSecurityEnvironmentHandler
        </xwss:SecurityEnvironmentHandler>

    </xwss:JAXRPCSecurity>
```

If you compare this security configuration file to the similar one in the `simple` sample, as discussed in Adding a Username Password Token, you'll see that this security configuration file does not hard-code the user name and password. In this example, the username and password are obtained by reading a system property, `username.password`, that is configured in the `build.xml` file of the `jaas-sample` under the `run-sample` target as a `sysproperty`. The properties and a section of the target from this example are configured like this:

```
<target name="run-sample"
        depends="clean, prepare, build-server, deploy-forced,
build-client"
          description="Runs the example client">
    <echo message="Running the ${client-class} program...."/>
    <java fork="on" classname="${client-class}">
      <sysproperty key="java.endorsed.dirs"
value="${java.endorsed.dirs}"/>
     <sysproperty key="endpoint.host" value="${endpoint.host}"/
>
     <sysproperty key="endpoint.port" value="${endpoint.port}"/
>
      <sysproperty key="service.url" value="${service.url}"/>
      <sysproperty key="username.password" value="Ron noR"/>
```

The client-side `SecurityEnvironmentHandler` of this sample is the entity that actually reads the system property at run-time and populates the username and password `Callback` objects passed to it by the XWS-Security run-time. A different `SecurityEnvironmentHandler` can be plugged into this sample to obtain the username and password at run-time from a different source (possibly by popping up a dialog box where the user can enter the username and password).

This sample's server-side `SecurityEnvironmentHandler` makes use of a JAAS login module that takes care of authenticating the user name and password. The sample demonstrates how JAAS authentication can be plugged into applications that use the XWS-Security framework. The source of the JAAS login module, `UserPassLoginModule.java`, is located at *<JWSDP_HOME>*`/xws-security/`

samples/jaas-sample/src/com/sun/xml/wss/sample directory. The JAAS-Validator.java class in the same directory does the actual JAAS authentication by creating a LoginContext and calling the LoggingContext.login() method. The UserPassLoginModule makes use of a username-password XML database located at <*JWSDP_HOME*>/xws-security/etc/userpasslist.xml when performing the actual authentication in its login() method.

Setting Up For the JAAS-Sample

Before the sample application will run correctly, you must have completed the tasks defined in the following sections of this addendum:

- Setting System Properties
- Setting Build Properties

In addition, follow the steps in this section that are specific to the jaas-sample application.

1. Stop the Application Server.
2. Set the user name and password for the example.

   Because the samples are run using ASAnt tasks, the user name and password for this example are set as a system property. The build.xml file for the jaas-sample example includes the following line under the run-sample target that uses a user name and password supplied in the <*JWSDP_HOME*>/xws-security/etc/userpasslist.xml file.

   ```
   <sysproperty key=”username.password” value=”Ron noR”/>
   ```

   The JAAS login module also makes use of the userpasslist.xml file, so make sure that this file exists and contains the user name and password specified in the build.xml file.

3. Add the following JAAS policy to the JAAS policy file of the Application Server. This file can be found at <*SJSAS_HOME*>/domains/domain1/config/login.conf. Add the following code near the end of the file:

   ```
   /** Login Configuration for the Sample Application **/
   XWS_SECURITY_SERVER{com.sun.xml.wss.sample.UserPassLogin-
   Module REQUIRED debug=true;
   };
   ```

4. Add the following permissions to the server policy file of the Application Server. This file can be found at <*SJSAS_HOME*>/domains/domain1/config/server.policy. Add the following code near the end of the file:

```
grant  codeBase  "file:${com.sun.aas.instanceRoot}/applica-
tions/j2ee-modules/jaassample/WEB-INF/-" {
        permission javax.security.auth.AuthPermission "modi-
fyPrincipals";
        permission javax.security.auth.AuthPermission "modi-
fyPrivateCredentials";
        permission javax.security.auth.PrivateCredentialPer-
mission "* * \"*\"","read";
      permission javax.security.auth.AuthPermission "getSub-
ject";
      permission javax.security.auth.AuthPermission
         "createLoginContext.XWS_SECURITY_SERVER";
};
```

5. Save and exit all files.
6. Restart the Application Server.

Running the JAAS-Sample Application

To run the `jaas-sample` application, follow these steps:

1. Follow the steps in Setting Up For the JAAS-Sample.
2. Start the selected container and make sure the server is running. To start the Application Server,
   a. From a Unix machine, enter the following command from a terminal window: `asadmin start-domain domain1`
   b. From a Windows machine, choose Start→Programs→Sun Microsystems→Application Server→Start Default Server.
3. Build and run the application from a terminal window or command prompt.
   - On the Application Server, the command to build and run the application is: `asant run-sample`
   - On the other containers, the command to build and run the application is: `ant run-sample`

Note: To run the sample against a remote server containing the deployed endpoint, use the `run-remote-sample` target in place of the `run-sample` target. In this situation, make sure that the `endpoint.host`, `endpoint.port`, `http.proxyHost`, `http.proxyPort`, and `service.url` properties are set correctly in the `build.properties` file (as discussed in Setting Build Properties) before running the sample.

If the application runs successfully, you will see a message similar to the following:

```
          [echo] Running the sample.TestClient program....
          [java] Service URL=http://localhost:8080/jaassample/Ping
          [java] Username read=Ron
          [java] Password read=noR
          [java] INFO: ==== Sending Message Start ====
          [java] <?xml version="1.0" encoding="UTF-8"?>
          [java] <env:Envelope xmlns:env="http://
     schemas.xmlsoap.org/soap/envelope/" xmlns:enc="http://
     schemas.xmlsoap.org/soap/encoding/" xmlns:ns0="http://
     xmlsoap.org/Ping" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
          [java] <env:Header>
          [java] <wsse:Security xmlns:wsse="http://docs.oasis-
     open.org/wss/2004/01/oasis-200401-
          wss-wssecurity-secext-1.0.xsd" env:mustUnderstand="1">
          [java] <wsse:UsernameToken>
          [java] <wsse:Username>Ron</wsse:Username>
          [java] <wsse:Password>****</wsse:Password>
          [java] <wsse:Nonce EncodingType="http://docs.oasis-
     open.org/wss/2004/01/oasis-
          200401-wss-soap-message-security-
          1.0#Base64Binary">qdKj8WL0U3r21rcgOiM4H76H</wsse:Nonce>
         [java] <wsu:Created xmlns:wsu="http://docs.oasis-open.org/
     wss/2004/01/oasis-200401-
          wss-wssecurity-utility-1.0.xsd">2004-11-05T02:07:46Z</
     wsu:Created>
          [java] </wsse:UsernameToken>
          [java] </wsse:Security>
          [java] </env:Header>
          [java] <env:Body>
          [java] <ns0:Ping>
          [java] <ns0:ticket>SUNW</ns0:ticket>
          [java] <ns0:text>Hello !</ns0:text>
          [java] </ns0:Ping>
          [java] </env:Body>
          [java] </env:Envelope>
          [java] ==== Sending Message End ====

          [java] INFO: ==== Received Message Start ====
          [java] <?xml version="1.0" encoding="UTF-8"?>
          [java] <env:Envelope xmlns:env="http://
```

```
schemas.xmlsoap.org/soap/envelope/"
     xmlns:enc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:ns0="http://xmlsoap.org/
     Ping" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
     [java] <env:Body>
     [java] <ns0:PingResponse>
     [java] <ns0:text>Hello !</ns0:text>
     [java] </ns0:PingResponse>
     [java] </env:Body>
     [java] </env:Envelope>
     [java] ==== Received Message End ====
```

The server code in `server/src/sample/PingImpl.java` makes use of a `SubjectAccessor` to access and print the authenticated `Subjects` principal from within the business method `Ping()`.

You can view similar messages in the server logs:

```
<SJSAS_HOME>/domains/<domain-name>/logs/server.log
<TOMCAT_HOME>/logs/launcher.server.log
<SJSWS_HOME>/<Virtual-Server-Dir>/logs/errors
```

XWS-Security APIs Sample Application

The focus of `api-sample` is to demonstrate how to use XWS-Security APIs to secure and validate SOAP messages in a stand-alone (non-JAX-RPC) SAAJ application. The XWS-Security APIs can be used to secure JAX-RPC applications, too, but because securing JAX-RPC applications can be easily accomplished using the security configuration files, this sample application focuses on securing stand-alone, non-JAX-RPC applications.

This sample uses configuration files that start with `<xwss:SecurityConfiguration>` as the root element, as opposed to the other XWS-Security samples that are based on JAX-RPC and use `<xwss:JAXRPCSecurity>` as the root element.

Documentation for XWS-Security 2.0 EA APIs is located in the `/xws-security/docs/api` directory.

The *<JWSDP_HOME>*/xws-security/samples/api-sample application demonstrates the following functionality:

- Defines an ease-of-use interface, XWSSProcessor interface. This interface is intended to insulate API users from changes to the APIs in future releases.
- Provides an implementation of XWSSProcessor interface for XWS-Security 2.0 EA.
- The client (com.sun.wss.sample.Client) code uses the XWSSProcessor APIs to secure SOAP messages according to the security policy inferred from the SecurityConfiguration with which this XWSSProcessor was initialized.
- Server verifies the secured message.

The application prints out the client request and response SOAP messages. The output from the client is sent to stdout or whichever stream is used by the configured log handler. Messages are logged at the INFO level.

The example code is found in the /api-sample/com/sun/wss/sample/ directory.

The XWSSProcessor Interface

The XWSSProcessor interface defines methods for securing an outbound SOAPMessage and verifying the security in an inbound SOAPMessage. An XWSSProcessor can add and/or verify security in a SOAPMessage in the ways defined by the OASIS WSS 1.0 specification.

The XWSSProcessor interface contains the following methods:

- secureOutboundMessage
  This method adds security to an outbound SOAPMessage according to the security policy inferred from the SecurityConfiguration with which this XWSSProcessor was initialized.
- verifyInboundMessage
  This method verifies security in an inbound SOAPMessage according to the security policy inferred from the SecurityConfiguration with which this XWSSProcessor was initialized.

API-Sample Client Code

The client code (samples/api-sample/com/sun/wss/sample/Client.java) uses the XWSSProcessor APIs to secure SOAP messages according to the security policy inferred from the SecurityConfiguration with which this XWSSProcessor was initialized. The following code demonstrates how this is done:

```
public static void main(String[] args) throws Exception {

        FileInputStream clientConfig = null;
        FileInputStream serverConfig = null;
        try {
            //read client-side security configuration
            clientConfig = new java.io.FileInputStream(
                    new
java.io.File(System.getProperty("client.configfile")));
            //read server-side security configuration
            serverConfig = new java.io.FileInputStream(
                    new
java.io.File(System.getProperty("server.configfile")));
        } catch (Exception e) {
            e.printStackTrace();
            throw e;
        }

        //Create a XWSSProcessFactory.
        XWSSProcessorFactory factory =
XWSSProcessorFactory.newInstance();

       //Create XWSSProcessor to secure outgoing soap messages.
        //Sample SecurityEnvironment is configured to
        //use client-side keystores.

        XWSSProcessor cprocessor =
                factory.createForSecurityConfiguration(
                clientConfig, new
SecurityEnvironmentHandler("client"));

       //Create XWSSProcessor to veriy incoming soap messages.
        //Sample SecurityEnvironment is configured to
        //use server-side keystores.

        XWSSProcessor sprocessor =
                factory.createForSecurityConfiguration(
                serverConfig, new
SecurityEnvironmentHandler("server"));
        try{
            clientConfig.close();
            serverConfig.close();
        }catch(Exception ex){
            ex.printStackTrace();
           return;
        }

        for(int i=0;i<1;i++){

            // create SOAPMessage
            SOAPMessage msg =
MessageFactory.newInstance().createMessage();
            SOAPBody body = msg.getSOAPBody();
            SOAPBodyElement sbe = body.addBodyElement(
                    SOAPFactory.newInstance().createName(
                    "StockSymbol",
                    "tru",
                    "http://fabrikam123.com/payloads"));
            sbe.addTextNode("QQQ");

            //Create processing context and set the soap
            //message to be processed.
           ProcessingContext context = new ProcessingContext();
            context.setSOAPMessage(msg);

            //secure the message.
            SOAPMessage secureMsg = cprocessor.secureOutbound-
Message(context);

            //verify the secured message.
            context = new ProcessingContext();
            context.setSOAPMessage(secureMsg);

            SOAPMessage verifiedMsg= null;
            try{
             verifiedMsg= sprocessor.verifyInboundMessage(con-
text);
                //System.out.println("\nRequester Subject " +
SubjectAccessor.getRequesterSubject(context));

            }catch(Exception ex){
                ex.printStackTrace();
               //context.getSOAPMessage().writeTo(System.out);
            }
        }
```

The API Sample Security Configuration Files

The client (`com.sun.wss.sample.Client`) code uses the `XWSSProcessor` APIs to secure SOAP messages according to the security policy inferred from the `SecurityConfiguration` with which this `XWSSProcessor` was initialized. The `api-sample` contains many different example security configuration files. The following pairs should be used when specifying the client and server configuration files in `build.properties`. The client configuration to specify is listed first, the server configuration second:

- `sign-rsign.xml`, `sign-rsign.xml`
- `username.xml`, `username.xml`
- `encryptv1.xml`, `encryptv1.xml`
- `encryptv2.xml`, `encryptv2.xml`
- `signv1.xml`, `signv1.xml`
- `signv2.xml`, `signv1.xml`
- `signv3.xml`, `signv1.xml`
- `signv4.xml`, `signv1.xml`
- `str_transform.xml`, `str_transform.xml`

Note: The configuration files `strid.xml` and `no_security.xml` have syntax errors and should not be used.

Remember, when using the XWS-Security APIs to secure stand-alone application, we will use configuration files that start with `<xwss:SecurityConfiguration>` as the root element, as opposed to the other XWS-Security samples that are based on JAX-RPC and use `<xwss:JAXRPCSecurity>` as the root element.

Encrypting the SOAP Message

The security configuration files `encryptv1.xml` and `encryptv2.xml` enable the following tasks:

- Client encrypts an outbound `SOAPMessage` and sends it.
- Client verifies that the inbound `SOAPMessage` is encrypted.

The encryptv1.xml file looks like this:

```
<xwss:SecurityConfiguration dumpMessages="true"
   xmlns:xwss="http://java.sun.com/xml/ns/xwss/config">
   <xwss:Encrypt>
      <xwss:X509Token certificateAlias="s1as"/>
   </xwss:Encrypt>
</xwss:SecurityConfiguration>
```

The encryptv2.xml file does the same thing, but specifies the following:

- The public key encryption algorithm to be used for encrypting and decrypting keys.
- The encryption algorithm to be applied to the cipher data.
- Specifically identifies the type of encrypted structure being described.

It looks like this:

```
<xwss:SecurityConfiguration dumpMessages="true"
   xmlns:xwss="http://java.sun.com/xml/ns/xwss/config">
   <xwss:Encrypt>
        <xwss:X509Token certificateAlias="s1as"/>
        <xwss:KeyEncryptionMethod
              algorithm="http://www.w3.org/2001/04/xmlenc#rsa-
oaep-mgf1p"/>
        <xwss:DataEncryptionMethod
               algorithm="http://www.w3.org/2001/04/
xmlenc#aes128-cbc"/>
        <xwss:EncryptionTarget type="xpath" value=".//SOAP-
ENV:Body"/>
   </xwss:Encrypt>
</xwss:SecurityConfiguration>
```

Signing the SOAP Message

The security configuration files signv1.xml, signv2.xml, and signv3.xml enable the following tasks:

- Client signs an outbound SOAPMessage and sends it.
- Client verifies that the inbound SOAPMessage is signed.

The `signv1.xml` file looks like this:

```
<xwss:SecurityConfiguration dumpMessages="true"
xmlns:xwss="http://java.sun.com/xml/ns/xwss/config" >
    <!--
     Note that in the <Sign> operation, a Timestamp is exported
      in the security header and signed by default.
    -->
    <xwss:Sign>
      <xwss:X509Token certificateAlias="xws-security-client"/>
    </xwss:Sign>
</xwss:SecurityConfiguration>
```

The `signv2.xml` file does the same thing, except that it also includes the following:

- Specifies the canonicalization algorithm to be applied to the `<Sign>` element prior to performing signature calculations.
- Specifies the algorithm used for signature generation and validation.
- Provides a list of processing steps to be applied to the resource's content before it is digested.

It looks like this:

```
<xwss:SecurityConfiguration dumpMessages="true"
xmlns:xwss="http://java.sun.com/xml/ns/xwss/config" >
    <xwss:Sign>
      <xwss:X509Token certificateAlias="xws-security-client"/>
     <xwss:CanonicalizationMethod algorithm=
        "http://www.w3.org/2001/10/xml-exc-c14n#"/>
        <xwss:SignatureMethod
              algorithm="http://www.w3.org/2000/09/
xmldsig#rsa-sha1"/>
             <xwss:SignatureTarget type="uri" value="">
                  <xwss:DigestMethod
                       algorithm="http://www.w3.org/2000/09/
xmldsig#sha1"/>
                  <xwss:Transform
                  algorithm="http://www.w3.org/TR/1999/REC-
xpath-19991116">
                         <xwss:AlgorithmParameter name="XPATH"
                         value="./SOAP-ENV:Envelope/SOAP-
ENV:Header/wsse:Security/
                        ds:Signature[1]/ds:KeyInfo/
wsse:SecurityTokenReference"/>
                  </xwss:Transform>
                  <xwss:Transform algorithm="http://
```

```
docs.oasis-open.org/wss/2004/01/
                  oasis-200401-wss-soap-message-security-
1.0#STR-Transform">
                        <xwss:AlgorithmParameter
name="CanonicalizationMethod"
                        value="http://www.w3.org/2001/10/xml-
exc-c14n#"/>
                  </xwss:Transform>
             </xwss:SignatureTarget>
    </xwss:Sign>
</xwss:SecurityConfiguration>
```

The `signv3.xml` file looks the same as `signv1.xml` file, except that it sends the subject key identifier extension value of the certificate, instead of the actual certificate, along with the message. It looks like this:

```
<xwss:SecurityConfiguration dumpMessages="true"
xmlns:xwss="http://java.sun.com/xml/ns/xwss/config" >
    <!--
     Note that in the <Sign> operation, a Timestamp is exported
      in the security header and signed by default.
    -->
    <xwss:Sign>
       <xwss:X509Token certificateAlias="xws-security-client"
              keyReferenceType="Identifier"/>
    </xwss:Sign>
</xwss:SecurityConfiguration>
```

The `sign-rsign.xml` file looks like the `signv1.xml` file, except that it requires a signature. It looks like this:

```
<xwss:SecurityConfiguration dumpMessages="true"
xmlns:xwss="http://java.sun.com/xml/ns/xwss/config" >
    <xwss:Sign>
      <xwss:X509Token certificateAlias="xws-security-client"/>
    </xwss:Sign>
    <xwss:RequireSignature/>
</xwss:SecurityConfiguration>
```

The `str-transform.xml` file uses a Security Token Reference (STR) Dereference Transform, which is an option for referencing information to be signed. Other methods for referencing information to be signed include referencing URIs, IDs and XPaths. Use an STR-Transform when a token format does not allow tokens to be referenced using URIs or IDs and an XPath is undesirable.

STR-Transform allows you to create your own unique reference mechanism. It looks like this:

```
<xwss:SecurityConfiguration dumpMessages="true"
xmlns:xwss="http://java.sun.com/xml/ns/xwss/config" >
      <xwss:Sign>
            <xwss:X509Token certificateAlias="xws-security-
client"/>
      </xwss:Sign>
      <xwss:Sign>
            <xwss:X509Token certificateAlias="xws-security-
client"/>
            <xwss:Target
                  type="qname">{http://schemas.xmlsoap.org/
soap/envelope/}Body
                  </xwss:Target>
            <xwss:SignatureTarget type="xpath"
            value="/SOAP-ENV:Envelope/SOAP-ENV:Header/
wsse:Security/ds:Signature[1]/
            ds:KeyInfo/wsse:SecurityTokenReference">
                        <xwss:DigestMethod
                        algorithm="http://www.w3.org/2000/09/
xmldsig#sha1"/>
                        <xwss:Transform algorithm="http://
docs.oasis-open.org/wss/2004/01/
                        oasis-200401-wss-soap-message-secu-
rity-1.0#STR-Transform">
                              <xwss:AlgorithmParameter
name="CanonicalizationMethod"
                              value="http://www.w3.org/2001/
10/xml-exc-c14n#"/>
                  </xwss:Transform>
            </xwss:SignatureTarget>
      </xwss:Sign>
</xwss:SecurityConfiguration>
```

Sending a Username Token with the SOAP Message

The security configuration `username.xml` enables the following tasks:

- Client adds a username-password token to an outbound `SOAPMessage` and sends a request.
- Client verifies that the inbound `SOAPMessage` contains a `UsernameToken`.

The `username.xml` file looks like this:

```
<xwss:SecurityConfiguration dumpMessages="true"
xmlns:xwss="http://java.sun.com/xml/ns/xwss/config">
            <xwss:UsernameToken name="Ron" password="noR"/>
            <xwss:RequireUsernameToken/>
</xwss:SecurityConfiguration>
```

Building and Running the API Sample Application

This sample does not require that a container be running, so there is no need to start the Application Server for this example.

To run the `api-sample` application, follow these steps:

1. Complete the tasks defined in the following sections of this addendum:
   - Setting System Properties
   - Setting Build Properties
2. Modify the `client.configfile` and `server.configfile` properties in the `build.properties` file so that they points to a valid security configuration pair you want to run.
3. Build and run the application from a terminal window or command prompt.
   - On the Application Server, the command to build and run the application is: `asant run-sample`
   - On the other containers, the command to build and run the application is: `ant run-sample`

Note: To run the sample against a remote server containing the deployed endpoint, use the `run-remote-sample` target in place of the `run-sample` target. In this situation, make sure that the `endpoint.host`, `endpoint.port`, `http.proxyHost`, `http.proxyPort`, and `service.url` properties are set correctly in the `build.properties` file (as discussed in Setting Build Properties) before running the sample.

Soap With Attachments Sample Application

The swainterop sample application demonstrates the Soap with Attachments (SwA) interoperability scenarios. SwA describes how a web service consumer can secure SOAP attachments using XWS-Security for attachment integrity, confidentiality and origin authentication, and how a receiver may process such a message. Read more about SwA at http://www.oasis-open.org/committees/download.php/10090/wss-swa-profile-1.0-draft-14.pdf.

This sample application was used as Sun's entry in a virtual interoperability demonstration sponsored by OASIS. This sample implements a set of interop scenarios required by the event. The scenarios addressed in this sample are described in The SwA Interop Scenarios. Read more about the SwA interop scenarios at http://lists.oasis-open.org/archives/wss/200410/pdf00003.pdf.

The application prints out both the client and server request and response SOAP messages. The output from the server may be viewed in the appropriate container's log file. The output from the client is sent to stdout or whichever stream is used by the configured log handler. Messages are logged at the INFO level.

In this example, server-side code is found in the /swainterop/server/src/swainterop/ directory. Client-side code is found in the /swainterop/client/src/swainterop/ directory. The asant (or ant) targets build objects under the /build/server/ and /build/client/ directories.

The SwA Interop Scenarios

All four SwA interop scenarios use the same, simple application. The Requester sends a Ping element with a value of a string as the single child to a SOAP request. The value is the organization that has developed the software and the number of the scenario, for example, in this application the value is "Sun Microsystems – Scenario #1". The Responder returns a PingResponse element with a value of the same string. Each interaction includes a SOAP attachment secured via one of the content-level security mechanisms described in the WSS SwA Profile.

The following is a summary of each of the SwA interop scenarios demonstrated in this sample application. You will need the numbers for the scenarios when you run the sample application.

1. Scenario #1 - Attachment Signature

   Scenario #1 tests the interoperability of a signed attachment using an X.509 certificate. The certificate used to verify the signature shall be present in the SOAP header. No security properties are applied to any part of the SOAP envelope.

2. Scenario #2 - Attachment Encryption

   The SOAP request has an attachment that has been encrypted. The encryption is done using a symmetric cipher. The symmetric encryption key is further encrypted for a specific recipient identified by an X.509 certificate. The certificate associated with the key encryption is provided to the requestor out-of-band. No security properties are applied to any part of the SOAP envelope.

3. Scenario #3 - Attachment Signature and Encryption

   The SOAP request contains an attachment that has been signed and then encrypted. The certificate associated with the encryption is provided out-of-band to the requestor. The certificate used to verify the signature is provided in the header. The Response Body is not signed or encrypted. There are two certificates in the request message. One identifiers the recipient of the encrypted attachment and one identifies the signer.

4. Scenario #4 - Attachment Signature and Encryption with MIME Headers

   The SOAP request contains an attachment that has been signed and then encrypted. The certificate associated with the encryption is provided out-of-band to the requestor. The certificate used to verify the signature is provided in the header. The Response Body is not signed or encrypted. There are two certificates in the request message. One identifies the recipient of the encrypted attachment and one identifies the signer. This scenario differs from the first three scenarios in that it covers MIME headers in the signature and encryption. This means that it uses the Attachment-Complete Signature Reference Transform and Attachment-Complete EncryptedData Type.

   Aside from these two changes, Scenario #4 is identical to Scenario #3.

SwA Sample Configuration Files

The security configuration pair `swa-client.xml` and `swa-server.xml` are used to secure message attachments. Each file contains the security configuration for each of the four scenarios.

You specify attachments as targets by specifying the value of the Content-ID (CID) header of the attachment. To do this, set the `type` attribute to `uri` and specify the target value as `cid:`*`<part-name>`*, where *`part-name`* is the WSDL part name of the `AttachmentPart`.

The `swa-client.xml` file looks like this:

```
<xwss:JAXRPCSecurity xmlns:xwss="http://java.sun.com/xml/ns/
xwss/config">

    <xwss:Service>

    <!-- Port 1: SwA Scenario 1 Sign Attachment Only-->
      <xwss:Port name="{http://xmlsoap.org/Ping}Ping1">
        <xwss:SecurityConfiguration dumpMessages="true">
            <!--
             Note that in the <Sign> operation, a Timestamp is
exported
              in the security header and signed by default.
            -->
            <xwss:Sign includeTimestamp="false">
               <xwss:X509Token certificateAlias="xws-security-
client"/>
                <xwss:CanonicalizationMethod
                   algorithm="http://www.w3.org/2001/10/xml-
exc-c14n#"/>
                <xwss:SignatureMethod
                   algorithm="http://www.w3.org/2000/09/
xmldsig#rsa-sha1"/>
                <xwss:SignatureTarget type="uri"
value="cid:foobar">
                    <xwss:DigestMethod algorithm="http://
www.w3.org/2000/09/xmldsig#sha1"/>
                 <xwss:Transform algorithm="http://docs.oasis-
open.org/wss/2004/XX/
                       oasis-2004XX-wss-swa-profile-
1.0#Attachment-Complete-Transform"/>
                </xwss:SignatureTarget>
            </xwss:Sign>
        </xwss:SecurityConfiguration>
      </xwss:Port>
     <!-- Port 2: SwA Scenario 2 Encrypt Attachment Only -->
      <xwss:Port name="{http://xmlsoap.org/Ping}Ping2">
        <xwss:SecurityConfiguration dumpMessages="true">
            <xwss:Encrypt>
                <xwss:X509Token certificateAlias="s1as"
keyReferenceType="Direct"/>
              <xwss:Target type="uri">cid:foobar</xwss:Target>
            </xwss:Encrypt>
        </xwss:SecurityConfiguration>
      </xwss:Port>

   <!-- Port 3: SwA Scenario 3 Attachment Signature and Encryp-
tion -->
      <xwss:Port name="{http://xmlsoap.org/Ping}Ping3">
        <xwss:SecurityConfiguration dumpMessages="true">
            <xwss:Sign includeTimestamp="false">
               <xwss:X509Token certificateAlias="xws-security-
client"/>
                <xwss:CanonicalizationMethod
                     algorithm="http://www.w3.org/2001/10/xml-
exc-c14n#"/>
                <xwss:SignatureMethod
                     algorithm="http://www.w3.org/2000/09/
xmldsig#rsa-sha1"/>
                <xwss:SignatureTarget type="uri"
value="cid:foobar">
                    <xwss:DigestMethod algorithm="http://
www.w3.org/2000/09/xmldsig#sha1"/>
                 <xwss:Transform algorithm="http://docs.oasis-
open.org/wss/2004/XX/
                        oasis-2004XX-wss-swa-profile-
1.0#Attachment-Complete-Transform"/>
                </xwss:SignatureTarget>
            </xwss:Sign>
            <xwss:Encrypt>
                <xwss:X509Token certificateAlias="s1as"
keyReferenceType="Direct"/>
              <xwss:Target type="uri">cid:foobar</xwss:Target>
            </xwss:Encrypt>
        </xwss:SecurityConfiguration>
      </xwss:Port>

      <!-- Port 4: SwA Scenario 4 Attachment Signature and
Encryption
                          With MIME Headers-->
      <xwss:Port name="{http://xmlsoap.org/Ping}Ping4">
        <xwss:SecurityConfiguration dumpMessages="true">
```

```
            <xwss:Sign includeTimestamp="false">
               <xwss:X509Token certificateAlias="xws-security-
client"/>
                <xwss:CanonicalizationMethod
                    algorithm="http://www.w3.org/2001/10/xml-
exc-c14n#"/>
                <xwss:SignatureMethod
                    algorithm="http://www.w3.org/2000/09/
xmldsig#rsa-sha1"/>
                <xwss:SignatureTarget type="uri"
value="cid:foobar">
                    <xwss:DigestMethod algorithm="http://
www.w3.org/2000/09/xmldsig#sha1"/>
                 <xwss:Transform algorithm="http://docs.oasis-
open.org/wss/2004/XX/
                         oasis-2004XX-wss-swa-profile-1.0#
                         Attachment-Content-Only-Transform"/>
                </xwss:SignatureTarget>
            </xwss:Sign>
            <xwss:Encrypt>
                <xwss:X509Token certificateAlias="s1as"
keyReferenceType="Direct"/>
                <xwss:Target type="uri" conten-
tOnly="false">cid:foobar</xwss:Target>
            </xwss:Encrypt>
        </xwss:SecurityConfiguration>
      </xwss:Port>

   </xwss:Service>

    <xwss:SecurityEnvironmentHandler>
        com.sun.xml.wss.sample.SecurityEnvironmentHandler
    </xwss:SecurityEnvironmentHandler>

</xwss:JAXRPCSecurity>
```

The security configuration file for the server is `swa-server.xml`. Ideally, each scenario would contain a `RequireSignature` and/or `RequireEncryption` element, but we have not done this yet. The `swa-server.xml` file looks like this:

```
<xwss:JAXRPCSecurity xmlns:xwss="http://java.sun.com/xml/ns/
xwss/config">

    <xwss:Service>

    <!-- Port 1: SwA Scenario 1 Sign Attachment Only-->
       <xwss:Port name="{http://xmlsoap.org/Ping}Ping1">
         <xwss:SecurityConfiguration dumpMessages="true">
```

```
</xwss:SecurityConfiguration>
       </xwss:Port>

    <!-- Port 2: SwA Scenario 2 Encrypt Attachment Only-->
       <xwss:Port name="{http://xmlsoap.org/Ping}Ping2">
         <xwss:SecurityConfiguration dumpMessages="true">
        </xwss:SecurityConfiguration>
       </xwss:Port>

   <!-- Port 3: SwA Scenario 3 Attachment Signature and Encryp-
tion-->
       <xwss:Port name="{http://xmlsoap.org/Ping}Ping3">
        <xwss:SecurityConfiguration dumpMessages="true"/>
       </xwss:Port>

  <!-- Port 4: SwA Scenario 4 Attachment Signature and Encryp-
tion
         With MIME Headers -->
       <xwss:Port name="{http://xmlsoap.org/Ping}Ping4">
        <xwss:SecurityConfiguration dumpMessages="true"/>
       </xwss:Port>

    </xwss:Service>

    <xwss:SecurityEnvironmentHandler>
        com.sun.xml.wss.sample.SecurityEnvironmentHandler
    </xwss:SecurityEnvironmentHandler>

</xwss:JAXRPCSecurity>
```

Running the SwA Sample Application

To run the `swainterop` sample application, follow these steps:

1. Complete the tasks defined in the following sections of this addendum:
   - Setting System Properties
   - Configuring a JCE Provider
   - Setting Build Properties
2. Start the selected container and make sure the server is running. To start the Application Server,
   a. From a Unix machine, enter the following command from a terminal window: `asadmin start-domain domain1`

   b. From a Windows machine, choose Start→Programs→Sun Microsystems→J2EE 1.4→Start Default Server.

3. Make sure that you have modified the `/swainterop/build.properties` file for this sample as described in Setting Build Properties.
4. Build and run the application from a terminal window or command prompt.
   - On the Application Server, the command to build and run the application is: `asant run-sample-<number>`
   - On the other containers, the command to build and run the application is: `ant run-sample-<number>`

   Where the *<number>* variable is the number of the interop scenario you want to run.

Note: To run the sample against a remote server containing the deployed endpoint, use the `run-remote-sample` target in place of the `run-sample` target. In this situation, make sure that the `endpoint.host`, `endpoint.port`, `http.proxyHost`, `http.proxyPort`, and `service.url` properties are set correctly in the `build.properties` file (as discussed in Setting Build Properties) before running the sample.

If the application runs successfully, you will see a message similar to the following. The final response will have `Sun Microsystems Scenario#<scenario-number>`.

```
[echo] Running the client program....
[java] ==== Sending Message Start ====
...
[java] ==== Sending Message End ====
[java] ==== Received Message Start ====
...
[java] ==== Received Message End ====
```

You will see similar messages in the server log files, which are located in the following files:

```
<SJSAS_HOME>/domains/<domain-name>/logs/server.log
<TOMCAT_HOME>/logs/launcher.server.log
<SJSWS_HOME>/<Virtual-Server-Dir>/logs/errors
```

SAML Sample Application

The samlinterop sample application demonstrates support for OASIS WSS SAML Token Profile 1.0 in XWS-Security. Security Assertion Markup Language (SAML) assertions are used as security tokens. SAML provides a means by which security assertions about messages can be exchanged between communicating service endpoints. SAML is also considered important for promoting interoperable Single-Sign-On (SSO) and Federated Identity. This release, JWSDP 1.6, adds partial support for SAML Token Profile 1.0.

This sample application was used as Sun's entry in a virtual interoperability demonstration sponsored by OASIS. This sample implements three out of the four interop scenarios required by the event and described in the WSS SAML Interop Scenarios document. The scenarios addressed in this interop are described in SAML Interop Scenarios. Read more about the SAML interop scenarios at the following URL: http://www.oasis-open.org/apps/org/workgroup/wss/download.php/7011/wss-saml-interop1-draft-11.doc.

The application prints out both the client and server request and response SOAP messages. The output from the server may be viewed in the appropriate container's log file. The output from the client is sent to stdout or whichever stream is used by the configured log handler. Messages are logged at the INFO level.

In this example, server-side code is found in the /swainterop/server/src/swainterop/ directory. Client-side code is found in the /swainterop/client/src/swainterop/ directory. The asant (or ant) targets build objects under the /build/server/ and /build/client/ directories.

SAML Interop Scenarios

All four SAML interop scenarios invoke the same, simple application. The Requester sends a Ping element with a value of a string. The value of the string should be the name of the organization that has developed the software and the number of the scenario, e.g. "Sun Microsystems – Scenario #1". The Responder returns a PingResponse element with a value of the same string. These scenarios use the Request/Response Message Exchange Pattern (MEP) with no intermediaries. All scenarios use SAML v1.1 Assertions.

To validate and process an assertion, the receiver needs to establish the relationship between the subject of each SAML subject statement and the entity providing the evidence to satisfy the confirmation method defined for the statements.

The two methods for establishing this correspondence are Holder-of-Key (HOV) and Sender-Vouches (SV). For more information on these confirmation methods, read SAML Token Profile 1.0.

The following is a summary of each of the SAML interop scenarios.

- Scenario #1 - Sender-Vouches: Unsigned

  The request contains a minimal sender-vouches SAML assertion with no optional elements included. There are no signatures or certificates required. The response does not contain a security header.

  In this scenario, there is no technical basis for trust because the messages are sent in the clear with no content or channel protection. This scenario is intended only to demonstrate message structure interoperability and is not intended for production use.

- Scenario #2 - Sender-Vouches: Unsigned: SSL (sample not provided)

  The request contains a `sender-vouches` SAML assertion. There are no signatures required. This scenario is tested over SSL, and certificates are required to support SSL at the transport layer. The response does not contain a security header.

  In this scenario, the basis of trust is the Requester's client certificate used to establish the SSL link. The Responder relies on the Requester who vouches for the contents of the User message and the SAML Assertion.

  This scenario is not demonstrated in this sample application.

- Scenario #3 - Sender-Vouches: Signed

  The request contains a `sender-vouches` SAML assertion. The `Assertion` and the `Body` elements are signed. A reference to the certificate used to verify the signature is provided in the header. The response does not contain a security header.

  In this scenario, the basis of trust is the Requester's certificate. The Requester's private key is used to sign both the SAML Assertion and the message Body. The Responder relies on the Requester, who vouches for the contents of the User message and the SAML Assertion.

- Scenario #4 - Holder-of-Key

  The request contains a `holder-of-key` SAML assertion. The assertion is signed by the assertion issuer with an enveloped signature. The certificate used to verify the issuer signature is contained within the assertion signature. The message body is signed by the Requester. The certificate used to verify the Requester’s signature is contained in the assertion `Subject-Confirmation`. The response does not contain a security header.

In this scenario, the basis of trust is the Assertion Issuer’s certificate. The Assertion Issuer’s private key is used to sign the SAML Assertion for the User. The Responder relies on the Assertion Issuer to have issued the assertion to an authorized User.

SAML Interop Sample Configuration Files

The following sections provide the example configuration files for SAML interop scenarios 1, 3, and 4:

- Sender-Vouches Sample Configuration Files
- Holder-Of-Key Sample Configuration Files

Sender-Vouches Sample Configuration Files

The security configuration pair `sv-saml-client3.xml` and `sv-saml-server3.xml` enable the following tasks, as required by Scenario #3:

- Client contains a sender-vouches SAML assertion.
- Client signs the assertion and the body elements.
- Client includes a reference to the certificate used to verify the signature in the header.
- Client sends the request body.
- Server verifies that a SAML assertion is received.
- Server verifies the signature.
- Server sends the response, which does not contain a security header.

The `sv-saml-client3.xml` file looks like this:

```
<xwss:JAXRPCSecurity xmlns:xwss="http://java.sun.com/xml/ns/
xwss/config">
    <xwss:Service>
        <xwss:SecurityConfiguration dumpMessages="true">
            <xwss:SAMLAssertion type="SV" strId="SV-123"/>
            <xwss:Sign includeTimestamp="false">
               <xwss:X509Token certificateAlias="xws-security-
client"/>
                <xwss:Target type="qname">
                      {http://schemas.xmlsoap.org/soap/
envelope/}Body
                </xwss:Target>
              <xwss:SignatureTarget type="uri" value="SV-123">
                 <xwss:Transform algorithm="http://docs.oasis-
open.org/wss/2004/01/
                          oasis-200401-wss-soap-message-
security-1.0#STR-Transform">
                        <xwss:AlgorithmParameter
name="CanonicalizationMethod"
                             value="http://www.w3.org/2001/10/
xml-exc-c14n#"/>
                   </xwss:Transform>
                </xwss:SignatureTarget>
            </xwss:Sign>
            <xwss:Timestamp />
        </xwss:SecurityConfiguration>
   </xwss:Service>
    <xwss:SecurityEnvironmentHandler>
        sample.SecurityEnvironmentHandler
    </xwss:SecurityEnvironmentHandler>
</xwss:JAXRPCSecurity>
```

The `sv-saml-server3.xml` file looks like this:

```
<xwss:JAXRPCSecurity xmlns:xwss="http://java.sun.com/xml/ns/
xwss/config">
    <xwss:Service>
        <xwss:SecurityConfiguration dumpMessages="true">
         <xwss:RequireTimestamp />
         <xwss:RequireSAMLAssertion type="SV"/>
         <xwss:RequireSignature requireTimestamp="false">
         <xwss:X509Token />
         <xwss:Target type="qname">
               {http://schemas.xmlsoap.org/soap/envelope/}Body
         </xwss:Target>
         <xwss:SignatureTarget type="uri" value="SV-123"/>
        </xwss:RequireSignature>
        </xwss:SecurityConfiguration>
   </xwss:Service>
    <xwss:SecurityEnvironmentHandler>
        sample.SecurityEnvironmentHandler
    </xwss:SecurityEnvironmentHandler>
</xwss:JAXRPCSecurity>
```

The other security configuration files in the `/samlinterop/config/` directory that contain a sender-vouches type of assertion are those in support of Scenario #1, the `sv-saml-client.xml` and `sv-saml-server.xml` pair.

Holder-Of-Key Sample Configuration Files

The security configuration pair `hok-saml-client.xml` and `hok-saml-server.xml` enable the following tasks, as required by Scenario #4:

- Client contains a holder-of-key SAML assertion.
- Client has the assertion signed by the assertion issuer with an enveloped signature.
- Client includes the certificate used to verify the issuer signature in the assertion signature.
- Client signs the request body.
- Server verifies that a SAML assertion is received.
- Server verifies the signature.
- Server sends the response, which does not contain a security header.

The `hok-saml-client.xml` file looks like this:

```
<xwss:JAXRPCSecurity xmlns:xwss="http://java.sun.com/xml/ns/
xwss/config">
    <xwss:Service>
        <xwss:SecurityConfiguration dumpMessages="true">
            <xwss:Sign includeTimestamp="false">
                <xwss:SAMLAssertion type="HOK"/>
                <xwss:Target type="qname">
                     {http://schemas.xmlsoap.org/soap/
envelope/}Body
                </xwss:Target>
            </xwss:Sign>
            <xwss:Timestamp />
        </xwss:SecurityConfiguration>
   </xwss:Service>
    <xwss:SecurityEnvironmentHandler>
        sample.SecurityEnvironmentHandler
    </xwss:SecurityEnvironmentHandler>
</xwss:JAXRPCSecurity>
```

The `hok-saml-server.xml` file looks like this:

```
<xwss:JAXRPCSecurity xmlns:xwss="http://java.sun.com/xml/ns/
xwss/config">
    <xwss:Service>
        <xwss:SecurityConfiguration dumpMessages="true">
         <xwss:RequireTimestamp />
         <xwss:RequireSignature requireTimestamp="false">
         <xwss:SAMLAssertion type="HOK"/>
```

```
                <xwss:Target type="qname">
                    {http://schemas.xmlsoap.org/soap/envelope/}Body
                 </xwss:Target>
               </xwss:RequireSignature>
               </xwss:SecurityConfiguration>
          </xwss:Service>
           <xwss:SecurityEnvironmentHandler>
               sample.SecurityEnvironmentHandler
           </xwss:SecurityEnvironmentHandler>
       </xwss:JAXRPCSecurity>
```

Running the SAML Interop Sample

To run the `samlinterop` sample application, follow these steps:

1. Complete the tasks defined in the following sections of this addendum:
   - Setting System Properties
   - Configuring a JCE Provider
   - Setting Build Properties
2. Start the selected container and make sure the server is running. To start the Application Server,
   a. From a Unix machine, enter the following command from a terminal window: `asadmin start-domain domain1`

   b. From a Windows machine, choose Start→Programs→Sun Microsystems→J2EE 1.4→Start Default Server.
3. Modify the `build.properties` file to set up the security configuration that you want to run for the client and/or server. To do this, remove the comment character (#) from beside the client and server configuration pair to be used, and make sure the other security configuration files have the comment character beside them. See SAML Interop Sample Configuration Files for more information on the security configurations options defined for this sample application.
4. Build and run the application from a terminal window or command prompt.
   - On the Application Server, the command to build and run the application is: `asant run-sample`
   - On the other containers, the command to build and run the application is: `ant run-sample`

Note: To run the sample against a remote server containing the deployed endpoint, use the `run-remote-sample` target in place of the `run-sample` target. In this situation, make sure that the `endpoint.host`, `endpoint.port`, `http.proxyHost`, `http.proxyPort`, and `service.url` properties are set correctly in the `build.properties` file (as discussed in Setting Build Properties) before running the sample.

If the application runs successfully, you will see a message similar to the following:

```
[echo] Running the client program....
     [java] ==== Sending Message Start ====
     ...
     [java] ==== Sending Message End ====
     [java] ==== Received Message Start ====
     ...
     [java] ==== Received Message End ====
     [java] Hello to Duke!
```

You can view similar messages in the server logs:

```
<SJSAS_HOME>/domains/<domain-name>/logs/server.log
<TOMCAT_HOME>/logs/launcher.server.log
<SJSWS_HOME>/<Virtual-Server-Dir>/logs/errors
```

Dynamic Policy Sample Application

The `dynamic-policy` sample application demonstrates how to specify (or change) the request and/or response security policy at runtime using the XWS-Security APIs. Another sample that demonstrates using the XWS-Security APIs is `api-sample`, which is discussed in XWS-Security APIs Sample Application.

You would want to dynamically set the security policy for an application at runtime when one of these conditions is present:

- Response policy: When you don't know who the requester may be, you want to be able to specify the response security policy after you determine the identity of the requester.
- Request policy: When you don't know what the runtime parameters will be, you want to discover these parameters, such as whether SSL is enabled at the transport layer, before you specify your request policy.

The application prints out both the client and server request and response SOAP messages. The output from the server may be viewed in the appropriate container's log file. The output from the client is sent to `stdout` or to whichever stream is used by the configured log handler. Messages are logged at the `INFO` level.

In this example, server-side code is found in the `/dynamic-policy/server/src/sample/` directory. Client-side code is found in the `/dynamic-policy/client/src/sample/` directory. The `asant` (or `ant`) targets build objects under the `/build/server/` and `/build/client/` directories.

Security Configuration Files for Enabling Dynamic Policy

To specify the request and/or response security policy dynamically at runtime, you need to enable `DynamicPolicyCallback` by setting the `enableDynamicPolicy` flag on the `<xwss:SecurityConfiguration>` element. The application-defined runtime parameters can then be set by the application and passed into the `ProcessingContext`, which is made available to the `CallbackHandler` as a `DynamicApplicationContext`. The `CallbackHandler` can then modify an existing policy or set a completely new policy into the `Callback`.

As you can see, the security configuration files for this example are very simple, because the actual security policy that will be applied at runtime is being decided by `SecurityEnvironmentHandler`. The SecurityEnvironmentHandler is discussed in Setting Security Policies at Runtime. The security configuration file for the client, `dynamic-client.xml`, looks like this:

```
<xwss:JAXRPCSecurity xmlns:xwss="http://java.sun.com/xml/ns/
xwss/config">

    <xwss:Service>
        <!-- the exact policy to apply will be decided by the
               SecurityEnvironmentHandler at runtime -->
       <xwss:SecurityConfiguration dumpMessages="true" enable-
DynamicPolicy="true">
        </xwss:SecurityConfiguration>
   </xwss:Service>

    <xwss:SecurityEnvironmentHandler>
```

```
        com.sun.xml.wss.sample.SecurityEnvironmentHandler
    </xwss:SecurityEnvironmentHandler>

</xwss:JAXRPCSecurity>
```

The security configuration file for the server, `dynamic-server.xml`, looks like this:

```
<xwss:JAXRPCSecurity xmlns:xwss="http://java.sun.com/xml/ns/
xwss/config">

    <xwss:Service>
         <!-- the exact policy to apply will be decided by the
                SecurityEnvironmentHandler at runtime -->
       <xwss:SecurityConfiguration dumpMessages="true" enable-
DynamicPolicy="true">
        </xwss:SecurityConfiguration>
   </xwss:Service>

    <xwss:SecurityEnvironmentHandler>
        com.sun.xml.wss.sample.SecurityEnvironmentHandler
    </xwss:SecurityEnvironmentHandler>

</xwss:JAXRPCSecurity>
```

Setting Security Policies at Runtime

The `dynamic-policy` sample application demonstrates how the request and response security policies can be set at runtime from the `SecurityEnvironmentHandler` callback.

In this sample, the `SecurityEnvironmentHandler` inserts a `SignaturePolicy` at runtime. The `SignaturePolicy` asks for a signature over the body of the message. For the requesting side, this is equivalent to using an `<xwss:Sign>` element in the configuration file. For the receiving side, this is equivalent to using an `<xwss:RequireSignature>` element in the configuration file. Both the request and response contain a signature over the body.

Note: The APIs used in this sample by the `SecurityEnvironmentHandler` callback are evolving and hence are subject to modification prior to the release of XWS Security FCS 2.0.

The full code for the SecurityEnvironmentHandler is located in the /dynamic-policy/src/com/sun/xml/wss/sample directory. The SecurityEnvironmentHandler file is a sample implementation of a CallbackHandler. The following snippet of that file demonstrates how to handle a DynamicPolicyCallback:

```
} else if (callbacks[i] instanceof DynamicPolicyCallback) {
                DynamicPolicyCallback dpCallback =
(DynamicPolicyCallback) callbacks[i];
                SecurityPolicy policy =
dpCallback.getSecurityPolicy();

                if (policy instanceof WSSPolicy) {
                    try {
                         handleWSSPolicy (dpCallback);
                    } catch (PolicyGenerationException pge) {
                     throw new IOException (pge.getMessage());
                    }
            } else if (policy instanceof DynamicSecurityPolicy)
{
                    try {
                     handleDynamicSecurityPolicy (dpCallback);
                    } catch (PolicyGenerationException pge) {
                     throw new IOException (pge.getMessage());
                    }
                } else {
                    throw unsupported;
                }
```

Running the Dynamic Policy Sample Application

To run the dynamic-policy sample application, follow these steps:

1. Complete the tasks defined in the following sections of this addendum:
    - Setting System Properties
    - Configuring a JCE Provider
    - Setting Build Properties
2. Start the selected container and make sure the server is running. To start the Application Server,
    a. From a Unix machine, enter the following command from a terminal window: `asadmin start-domain domain1`

   b. From a Windows machine, choose Start→Programs→Sun Microsystems→J2EE 1.4→Start Default Server.

3. Build and run the application from a terminal window or command prompt.

   - On the Application Server, the command to build and run the application is: `asant run-sample`
   - On the other containers, the command to build and run the application is: `ant run-sample`

Note: To run the sample against a remote server containing the deployed endpoint, use the `run-remote-sample` target in place of the `run-sample` target. In this situation, make sure that the `endpoint.host`, `endpoint.port`, `http.proxyHost`, `http.proxyPort`, and `service.url` properties are set correctly in the `build.properties` file (as discussed in Setting Build Properties) before running the sample.

If the application runs successfully, you will see a message similar to the following:

```
[echo] Running the client program....
[java] ==== Sending Message Start ====
...
[java] ==== Sending Message End ====
[java] ==== Received Message Start ====
...
[java] ==== Received Message End ====
```

You can view similar messages in the server logs:

```
<SJSAS_HOME>/domains/<domain-name>/logs/server.log
<TOMCAT_HOME>/logs/launcher.server.log
<SJSWS_HOME>/<Virtual-Server-Dir>/logs/errors
```

Dynamic Response Sample Application

The `dynamic-response` sample application demonstrates using the certificate that arrived in a signed request to encrypt the response back to the requester using the XWS-Security APIs. To accomplish this task,

- A `CallbackHandler` retrieves the requester `Subject` and obtains its certificate.
- The requester certificate is used to encrypt the response back to the requester.

The application prints out both the client and server request and response SOAP messages. The output from the server may be viewed in the appropriate container's log file. The output from the client is sent to `stdout` or to whichever stream is used by the configured log handler. Messages are logged at the `INFO` level.

In this example, server-side code is found in the `/dynamic-response/server/src/sample/` directory. Client-side code is found in the `/dynamic-response/client/src/sample/` directory. The `asant` (or `ant`) targets build objects under the `/build/server/` and `/build/client/` directories.

Security Configuration Files for Enabling Dynamic Response

For this sample application, the security configuration files are fairly simple. The security configuration files are used to sign the request and encrypt the response, but the work of using the requester certificate to encrypt the response back to the requester is accomplished using the `SecurityEnvironmentHandler`, which is discussed in Using the CallbackHandler to Enable Dynamic Response.

The client security configuration file for this example, `sign-client.xml`, looks like this:

```
<xwss:JAXRPCSecurity xmlns:xwss="http://java.sun.com/xml/ns/
xwss/config">

    <xwss:Service>
        <xwss:SecurityConfiguration dumpMessages="true">
            <!--
```

```
             Note that in the <Sign> operation, a Timestamp is
exported
              in the security header and signed by default.
            -->
            <xwss:Sign>
               <xwss:X509Token certificateAlias="xws-security-
client"/>
            </xwss:Sign>
            <xwss:RequireEncryption/>
        </xwss:SecurityConfiguration>
   </xwss:Service>

    <xwss:SecurityEnvironmentHandler>
        sample.SecurityEnvironmentHandler
    </xwss:SecurityEnvironmentHandler>

</xwss:JAXRPCSecurity>
```

The server security configuration file for this example, `encrypt-server.xml`, looks like this:

```
<xwss:JAXRPCSecurity xmlns:xwss="http://java.sun.com/xml/ns/
xwss/config">

    <xwss:Service>
        <xwss:SecurityConfiguration dumpMessages="true">
            <xwss:RequireSignature/>
            <xwss:Encrypt/>
        </xwss:SecurityConfiguration>
    </xwss:Service>

    <xwss:SecurityEnvironmentHandler>
        sample.SecurityEnvironmentHandler
    </xwss:SecurityEnvironmentHandler>

</xwss:JAXRPCSecurity>
```

Using the CallbackHandler to Enable Dynamic Response

In this sample application, the security configuration files sign the request and encrypt the response, however the work of using the requester certificate to encrypt the response back to the requester is done in the `SecurityEnvironmentHandler`. The full source code for the `SecurityEnvironmentHandler` is located in the directory dynamic-response/src/sample. This snippet from that file demonstrates how to use a CallbackHandler to generate the response dynamically:

```
if (cb.getRequest() instanceof
EncryptionKeyCallback.AliasX509CertificateRequest) {

EncryptionKeyCallback.AliasX509CertificateRequest request =

(EncryptionKeyCallback.AliasX509CertificateRequest)
cb.getRequest();
                    String alias = request.getAlias();
                    if ((alias == null) || "".equals(alias)) {
                        Subject currentSubject =
SubjectAccessor.getRequesterSubject();
                        Set publicCredentials =
currentSubject.getPublicCredentials();
                        for (Iterator it =
publicCredentials.iterator(); it.hasNext();) {
                            Object cred = it.next();
                            if(cred instanceof
java.security.cert.X509Certificate){
                          java.security.cert.X509Certificate
cert =
                                       (java.security.cert.X509
Certificate)cred;
                              request.setX509Certificate(cert)
;
```

Running the Dynamic Response Sample Application

To run the dynamic-response sample application, follow these steps:

1. Complete the tasks defined in the following sections of this addendum:
   - Setting System Properties
   - Configuring a JCE Provider
   - Setting Build Properties
2. Start the selected container and make sure the server is running. To start the Application Server,
   a. From a Unix machine, enter the following command from a terminal window: asadmin start-domain domain1

   b. From a Windows machine, choose Start→Programs→Sun Microsystems→J2EE 1.4→Start Default Server.

3. Build and run the application from a terminal window or command prompt.
   - On the Application Server, the command to build and run the application is: `asant run-sample`
   - On the other containers, the command to build and run the application is: `ant run-sample`

Note: To run the sample against a remote server containing the deployed endpoint, use the `run-remote-sample` target in place of the `run-sample` target. In this situation, make sure that the `endpoint.host`, `endpoint.port`, `http.proxyHost`, `http.proxyPort`, and `service.url` properties are set correctly in the `build.properties` file (as discussed in Setting Build Properties) before running the sample.

If the application runs successfully, you will see a message similar to the following:

```
[echo] Running the client program....
[java] ==== Sending Message Start ====
...
[java] ==== Sending Message End ====
[java] ==== Received Message Start ====
...
[java] ==== Received Message End ====
```

You can view similar messages in the server logs:

```
<SJSAS_HOME>/domains/<domain-name>/logs/server.log
<TOMCAT_HOME>/logs/launcher.server.log
<SJSWS_HOME>/<Virtual-Server-Dir>/logs/errors
```

Further Information

For links to specifications and other documents relevant to XWS-Security, refer to the Further Information section in Introduction to XML and Web Services Security.

6

Java XML Digital Signature API

THE Java XML Digital Signature API is a standard Java API for generating and validating XML Signatures. This API was defined under the Java Community Process as JSR 105 (see `http://jcp.org/en/jsr/detail?id=105`). This JSR is final and this release of Java WSDP contains an FCS access implementation of the Final version of the APIs.

XML Signatures can be applied to data of any type, XML or binary (see `http://www.w3.org/TR/xmldsig-core/`). The resulting signature is represented in XML. An XML Signature can be used to secure your data and provide data integrity, message authentication, and signer authentication.

After providing a brief overview of XML Signatures and the XML Digital Signature API, this chapter presents two examples that demonstrate how to use the API to validate and generate an XML Signature. This chapter assumes that you have a basic knowledge of cryptography and digital signatures.

The API is designed to support all of the required or recommended features of the W3C Recommendation for XML-Signature Syntax and Processing. The API is extensible and pluggable and is based on the Java Cryptography Service Provider Architecture. The API is designed for two types of developers:

- Java programmers who want to use the XML Digital Signature API to generate and validate XML signatures

- Java programmers who want to create a concrete implementation of the XML Digital Signature API and register it as a cryptographic service of a JCA provider (see `http://java.sun.com/j2se/1.4.2/docs/guide/security/CryptoSpec.html#Provider`)

How XWS-Security and XML Digital Signature API Are Related

Before getting into specifics, it is important to see how XWS-Security and XML Digital Signature API are related. In this release of the Java WSDP, XWS-Security is based on non-standard XML Digital Signature APIs.

XML Digital Signature API is an API that should be used by Java applications and middleware that need to create and/or process XML Signatures. It can be used by Web Services Security (the goal for a future release) and by non-Web Services technologies (for example, signing documents stored or transferred in XML). Both JSR 105 and JSR 106 (XML Digital Encryption APIs) are core-XML security components. (See `http://www.jcp.org/en/jsr/detail?id=106` for more information about JSR 106.)

XWS-Security does not currently use the XML Digital Signature API or XML Digital Encryption APIs. XWS-Security uses the Apache libraries for XML-DSig and XML-Enc. The goal of XWS-Security is to move toward using these APIs in future releases.

XML Security Stack

Figure 6–1 shows how XML Digital Signature API (JSR 105) interacts with security components today and how it will interact with other security components, including XML Digital Encryption API (JSR 106), in future releases.

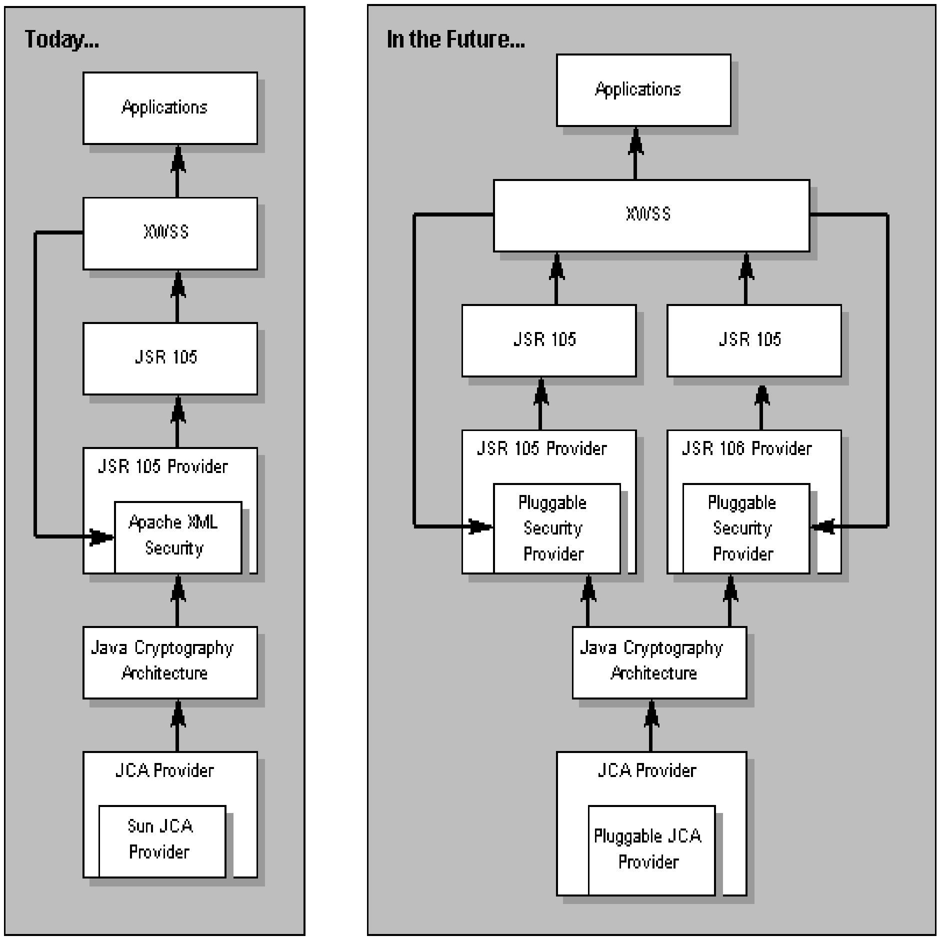

Figure 6–1 Java WSDP Security Components

XWSS calls Apache XML-Security directly today; in future releases, it should be able to call other pluggable security providers. The Apache XML-Security provider and the Sun JCA Provider are both pluggable components. Since JSR 105 is final today, the JSR 105 layer is standard now; the JSR 106 layer will be standard after that JSR becomes final.

Package Hierarchy

The six packages in the XML Digital Signature API are:

- `javax.xml.crypto`
- `javax.xml.crypto.dsig`
- `javax.xml.crypto.dsig.keyinfo`
- `javax.xml.crypto.dsig.spec`
- `javax.xml.crypto.dom`
- `javax.xml.crypto.dsig.dom`

The `javax.xml.crypto` package contains common classes that are used to perform XML cryptographic operations, such as generating an XML signature or encrypting XML data. Two notable classes in this package are the `KeySelector` class, which allows developers to supply implementations that locate and optionally validate keys using the information contained in a `KeyInfo` object, and the `URIDereferencer` class, which allows developers to create and specify their own URI dereferencing implementations.

The `javax.xml.crypto.dsig` package includes interfaces that represent the core elements defined in the W3C XML digital signature specification. Of primary significance is the `XMLSignature` class, which allows you to sign and validate an XML digital signature. Most of the XML signature structures or elements are represented by a corresponding interface (except for the `KeyInfo` structures, which are included in their own package and are discussed in the next paragraph). These interfaces include: `SignedInfo`, `CanonicalizationMethod`, `SignatureMethod`, `Reference`, `Transform`, `DigestMethod`, `XMLObject`, `Manifest`, `SignatureProperty`, and `SignatureProperties`. The `XMLSignatureFactory` class is an abstract factory that is used to create objects that implement these interfaces.

The `javax.xml.crypto.dsig.keyinfo` package contains interfaces that represent most of the `KeyInfo` structures defined in the W3C XML digital signature recommendation, including `KeyInfo`, `KeyName`, `KeyValue`, `X509Data`, `X509IssuerSerial`, `RetrievalMethod`, and `PGPData`. The `KeyInfoFactory` class is an abstract factory that is used to create objects that implement these interfaces.

The `javax.xml.crypto.dsig.spec` package contains interfaces and classes representing input parameters for the digest, signature, transform, or canonicalization algorithms used in the processing of XML signatures.

Finally, the `javax.xml.crypto.dom` and `javax.xml.crypto.dsig.dom` packages contains DOM-specific classes for the `javax.xml.crypto` and javax.xml.crypto.dsig packages, respectively. Only developers and users who are creating or using a DOM-based XMLSignatureFactory or KeyInfo-Factory implementation should need to make direct use of these packages.

Service Providers

A JSR 105 cryptographic service is a concrete implementation of the abstract XMLSignatureFactory and KeyInfoFactory classes and is responsible for creating objects and algorithms that parse, generate and validate XML Signatures and KeyInfo structures. A concrete implementation of XMLSignatureFactory *must* provide support for each of the *required* algorithms as specified by the W3C recommendation for XML Signatures. It *may* support other algorithms as defined by the W3C recommendation or other specifications.

JSR 105 leverages the JCA provider model for registering and loading XMLSignatureFactory and KeyInfoFactory implementations.

Each concrete XMLSignatureFactory or KeyInfoFactory implementation supports a specific XML mechanism type that identifies the XML processing mechanism that an implementation uses internally to parse and generate XML signature and KeyInfo structures. This JSR supports one standard type, DOM. The XML Digital Signature API early access provider implementation that is bundled with Java WSDP supports the DOM mechanism. Support for new standard types, such as JDOM, may be added in the future.

An XML Digital Signature API implementation *should* use underlying JCA engine classes, such as java.security.Signature and java.security.MessageDigest, to perform cryptographic operations.

In addition to the XMLSignatureFactory and KeyInfoFactory classes, JSR 105 supports a service provider interface for transform and canonicalization algorithms. The TransformService class allows you to develop and plug in an implementation of a specific transform or canonicalization algorithm for a particular XML mechanism type. The TransformService class uses the standard JCA provider model for registering and loading implementations. Each JSR 105 implementation *should* use the TransformService class to find a provider that supports transform and canonicalization algorithms in XML Signatures that it is generating or validating.

Introduction to XML Signatures

As mentioned, an XML Signature can be used to sign any arbitrary data, whether it is XML or binary. The data is identified via URIs in one or more Reference elements. XML Signatures are described in one or more of three forms: detached, enveloping, or enveloped. A detached signature is over data that is external, or outside of the signature element itself. Enveloping signatures are signatures over data that is inside the signature element, and an enveloped signature is a signature that is contained inside the data that it is signing.

Example of an XML Signature

The easiest way to describe the contents of an XML Signature is to show an actual sample and describe each component in more detail. The following is an example of an enveloped XML Signature generated over the contents of an XML document. The contents of the document before it is signed are:

```
<Envelope xmlns="urn:envelope">
</Envelope>
```

The resulting enveloped XML Signature, indented and formatted for readability, is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<Envelope xmlns="urn:envelope">
  <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
    <SignedInfo>
      <CanonicalizationMethod
           Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-
20010315#WithComments"/>
      <SignatureMethod Algorithm="http://www.w3.org/2000/09/
xmldsig#dsa-sha1"/>
      <Reference URI="">
        <Transforms>
          <Transform Algorithm="http://www.w3.org/2000/09/
xmldsig#enveloped-signature"/>
        </Transforms>
        <DigestMethod Algorithm="http://www.w3.org/2000/09/
xmldsig#sha1"/>
       <DigestValue>uooqbWYa5VCqcJCbuymBKqm17vY=</DigestValue>
      </Reference>
    </SignedInfo>
<SignatureValue>
```

```
KedJuTob5gtvYx9qM3k3gm7kbLBwVbEQRl26S2tmXjqNND7MRGtoew==
    </SignatureValue>
    <KeyInfo>
      <KeyValue>
        <DSAKeyValue>
          <P>
/KaCzo4Syrom78z3EQ5SbbB4sF7ey80etKII864WF64B81uRpH5t9jQTxe
Eu0ImbzRMqzVDZkVG9xD7nN1kuFw==
          </P>
          <Q>li7dzDacuo67Jg7mtqEm2TRuOMU=</Q>
          <G>Z4Rxsnqc9E7pGknFFH2xqaryRPBaQ01khpMdLRQnG541Awtx/
XPaF5Bpsy4pNWMOHCBiNU0NogpsQW5QvnlMpA==
          </G>
          <Y>qV38IqrWJG0V/
mZQvRVi1OHw9Zj84nDC4jO8P0axi1gb6d+475yhMjSc/
BrIVC58W3ydbkK+Ri4OKbaRZlYeRA==
         </Y>
        </DSAKeyValue>
      </KeyValue>
    </KeyInfo>
  </Signature>
</Envelope>
```

The `Signature` element has been inserted inside the content that it is signing, thereby making it an enveloped signature. The required `SignedInfo` element contains the information that is actually signed:

```
<SignedInfo>
  <CanonicalizationMethod
    Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-
20010315#WithComments"/>
  <SignatureMethod Algorithm="http://www.w3.org/2000/09/
xmldsig#dsa-sha1"/>
  <Reference URI="">
      <Transforms>
        <Transform Algorithm="http://www.w3.org/2000/09/
xmldsig#enveloped-signature"/>
      </Transforms>
      <DigestMethod Algorithm="http://www.w3.org/2000/09/
xmldsig#sha1"/>
      <DigestValue>uooqbWYa5VCqcJCbuymBKqm17vY=</DigestValue>
  </Reference>
</SignedInfo>
```

The required `CanonicalizationMethod` element defines the algorithm used to canonicalize the `SignedInfo` element before it is signed or validated. Canonicalization is the process of converting XML content to a canonical form, to take into account changes that can invalidate a signature over that data. Canonicalization is necessary due to the nature of XML and the way it is parsed by different processors and intermediaries, which can change the data such that the signature is no longer valid but the signed data is still logically equivalent.

The required `SignatureMethod` element defines the digital signature algorithm used to generate the signature, in this case DSA with SHA-1.

One or more `Reference` elements identify the data that is digested. Each `Reference` element identifies the data via a URI. In this example, the value of the URI is the empty String (""), which indicates the root of the document. The optional `Transforms` element contains a list of one or more `Transform` elements, each of which describes a transformation algorithm used to transform the data before it is digested. In this example, there is one `Transform` element for the enveloped transform algorithm. The enveloped transform is required for enveloped signatures so that the signature element itself is removed before calculating the signature value. The required `DigestMethod` element defines the algorithm used to digest the data, in this case SHA1. Finally the required `DigestValue` element contains the actual base64-encoded digested value.

The required `SignatureValue` element contains the base64-encoded signature value of the signature over the `SignedInfo` element.

The optional `KeyInfo` element contains information about the key that is needed to validate the signature:

```
<KeyInfo>
  <KeyValue>
    <DSAKeyValue>
      <P>
/KaCzo4Syrom78z3EQ5SbbB4sF7ey80etKII864WF64B81uRpH5t9jQTxe
Eu0ImbzRMqzVDZkVG9xD7nN1kuFw==
      </P>
      <Q>li7dzDacuo67Jg7mtqEm2TRuOMU=</Q>
      <G>Z4Rxsnqc9E7pGknFFH2xqaryRPBaQ01khpMdLRQnG541Awtx/
XPaF5Bpsy4pNWMOHCBiNU0NogpsQW5QvnlMpA==
      </G>
      <Y>
qV38IqrWJG0V/mZQvRVi1OHw9Zj84nDC4jO8P0axi1gb6d+475yhMjSc/
BrIVC58W3ydbkK+Ri4OKbaRZlYeRA==
      </Y>
    </DSAKeyValue>
  </KeyValue>
</KeyInfo>
```

This `KeyInfo` element contains a `KeyValue` element, which in turn contains a `DSAKeyValue` element consisting of the public key needed to validate the signature. `KeyInfo` can contain various content such as X.509 certificates and PGP key identifiers. See the `KeyInfo section` of the XML Signature Recommendation for more information on the different `KeyInfo` types.

XML Digital Signature API Examples

The following sections describe two examples that show how to use the XML Digital Signature API:

- Validate example
- Signing example

To run the sample applications using the supplied Ant `build.xml` files, issue the following commands after you installed Java WSDP:

For Solaris/Linux:

```
1.% export JWSDP_HOME=<your Java WSDP installation directory>
2.% export ANT_HOME=$JWSDP_HOME/apache-ant
3.  % export PATH=$ANT_HOME/bin:$PATH
4.  % cd $JWSDP_HOME/xmldsig/samples/<sample-name>
```

For Windows 2000/XP:

```
1. > set JWSDP_HOME=<your Java WSDP installation directory>
2.> set ANT_HOME=%JWSDP_HOME%\apache-ant
3.> set PATH=%ANT_HOME%\bin;%PATH%
4.> cd %JWSDP_HOME%\xmldsig\samples\<sample-name>
```

validate Example

You can find the code shown in this section in the `Validate.java` file in the `<JWSDP_HOME>/xmldsig/samples/validate` directory. The file on which it operates, `envelopedSignature.xml`, is in the same directory.

To run the example, execute the following command from the `<JWSDP_HOME>/xmldsig/samples/validate` directory:

```
$ ant
```

The sample program will validate the signature in the file `envelopedSignature.xml` in the current working directory. To validate a different signature, run the following command:

```
$ ant -Dsample.args="signature.xml"
```

where `"signature.xml"` is the pathname of the file.

Validating an XML Signature

This example shows you how to validate an XML Signature using the JSR 105 API. The example uses DOM (the Document Object Model) to parse an XML document containing a Signature element and a JSR 105 DOM implementation to validate the signature.

Instantiating the Document that Contains the Signature

First we use a JAXP `DocumentBuilderFactory` to parse the XML document containing the Signature. An application obtains the default implementation for `DocumentBuilderFactory` by calling the following line of code:

```
DocumentBuilderFactory dbf =
  DocumentBuilderFactory.newInstance();
```

We must also make the factory namespace-aware:

```
dbf.setNamespaceAware(true);
```

Next, we use the factory to get an instance of a `DocumentBuilder`, which is used to parse the document:

```
DocumentBuilder builder = dbf.newDocumentBuilder();
Document doc = builder.parse(new FileInputStream(argv[0]));
```

Specifying the Signature Element to be Validated

We need to specify the `Signature` element that we want to validate, since there could be more than one in the document. We use the DOM method `Docu-`

ment.getElementsByTagNameNS, passing it the XML Signature namespace URI and the tag name of the Signature element, as shown:

```
NodeList nl = doc.getElementsByTagNameNS
  (XMLSignature.XMLNS, "Signature");
if (nl.getLength() == 0) {
  throw new Exception("Cannot find Signature element");
}
```

This returns a list of all Signature elements in the document. In this example, there is only one Signature element.

Creating a Validation Context

We create an XMLValidateContext instance containing input parameters for validating the signature. Since we are using DOM, we instantiate a DOMValidateContext instance (a subclass of XMLValidateContext), and pass it two parameters, a KeyValueKeySelector object and a reference to the Signature element to be validated (which is the first entry of the NodeList we generated earlier):

```
DOMValidateContext valContext = new DOMValidateContext
  (new KeyValueKeySelector(), nl.item(0));
```

The KeyValueKeySelector is explained in greater detail in Using KeySelectors (page 277).

Unmarshaling the XML Signature

We extract the contents of the Signature element into an XMLSignature object. This process is called unmarshalling. The Signature element is unmarshalled using an XMLSignatureFactory object. An application can obtain a DOM implementation of XMLSignatureFactory by calling the following line of code:

```
XMLSignatureFactory factory =
  XMLSignatureFactory.getInstance("DOM");
```

We then invoke the `unmarshalXMLSignature` method of the factory to unmarshal an `XMLSignature` object, and pass it the validation context we created earlier:

```
XMLSignature signature =
  factory.unmarshalXMLSignature(valContext);
```

Validating the XML Signature

Now we are ready to validate the signature. We do this by invoking the `validate` method on the `XMLSignature` object, and pass it the validation context as follows:

```
boolean coreValidity = signature.validate(valContext);
```

The `validate` method returns "true" if the signature validates successfully according to the `core validation rules` in the `W3C XML Signature Recommendation`, and false otherwise.

What If the XML Signature Fails to Validate?

If the `XMLSignature.validate` method returns false, we can try to narrow down the cause of the failure. There are two phases in core XML Signature validation:

- `Signature validation` (the cryptographic verification of the signature)
- `Reference validation` (the verification of the digest of each reference in the signature)

Each phase must be successful for the signature to be valid. To check if the signature failed to cryptographically validate, we can check the status, as follows:

```
boolean sv =
  signature.getSignatureValue().validate(valContext);
System.out.println("signature validation status: " + sv);
```

We can also iterate over the references and check the validation status of each one, as follows:

```
Iterator i =
  signature.getSignedInfo().getReferences().iterator();
for (int j=0; i.hasNext(); j++) {
  boolean refValid = ((Reference)
     i.next()).validate(valContext);
  System.out.println("ref["+j+"] validity status: " +
     refValid);
}
```

Using KeySelectors

`KeySelectors` are used to find and select keys that are needed to validate an XMLSignature. Earlier, when we created a `DOMValidateContext` object, we passed a `KeySelector` object as the first argument:

```
DOMValidateContext valContext = new DOMValidateContext
  (new KeyValueKeySelector(), nl.item(0));
```

Alternatively, we could have passed a `PublicKey` as the first argument if we already knew what key is needed to validate the signature. However, we often don't know.

The `KeyValueKeySelector` is a concrete implementation of the abstract `KeySelector` class. The `KeyValueKeySelector` implementation tries to find an appropriate validation key using the data contained in `KeyValue` elements of the `KeyInfo` element of an `XMLSignature`. It does not determine if the key is trusted. This is a very simple `KeySelector` implementation, designed for illustration rather than real-world usage. A more practical example of a `KeySelector` is one that searches a `KeyStore` for trusted keys that match `X509Data` information (for example, `X509SubjectName`, `X509IssuerSerial`, `X509SKI`, or `X509Certificate` elements) contained in a `KeyInfo`.

The implementation of the `KeyValueKeySelector` is as follows:

```
private static class KeyValueKeySelector extends KeySelector {

  public KeySelectorResult select(KeyInfo keyInfo,
       KeySelector.Purpose purpose,
       AlgorithmMethod method,
       XMLCryptoContext context)
    throws KeySelectorException {

    if (keyInfo == null) {
       throw new KeySelectorException("Null KeyInfo object!");
    }
    SignatureMethod sm = (SignatureMethod) method;
    List list = keyInfo.getContent();

    for (int i = 0; i < list.size(); i++) {
            XMLStructure xmlStructure = (XMLStructure) list.get(i);
            if (xmlStructure instanceof KeyValue) {
               PublicKey pk = null;
               try {
                  pk = ((KeyValue)xmlStructure).getPublicKey();
               } catch (KeyException ke) {
                  throw new KeySelectorException(ke);
               }
               // make sure algorithm is compatible with method
               if (algEquals(sm.getAlgorithm(),
                     pk.getAlgorithm())) {
                  return new SimpleKeySelectorResult(pk);
               }
            }
         }
         throw new KeySelectorException("No KeyValue element 
found!");
      }

      static boolean algEquals(String algURI, String algName) {
         if (algName.equalsIgnoreCase("DSA") &&
               algURI.equalsIgnoreCase(SignatureMethod.DSA_SHA1)) {
            return true;
         } else if (algName.equalsIgnoreCase("RSA") &&
               algURI.equalsIgnoreCase(SignatureMethod.RSA_SHA1)) {
            return true;
         } else {
            return false;
         }
      }
   }
```

genenveloped Example

The code discussed in this section is in the `GenEnveloped.java` file in the `<`*`JWSDP_HOME`*`>/xmldsig/samples/genenveloped` directory. The file on which it operates, `envelope.xml`, is in the same directory. It generates the file `envelopedSignature.xml.`

To compile and run this sample, execute the following command from the `<`*`JWSDP_HOME`*`>/xmldsig/samples/genenveloped` directory:

```
$ ant
```

The sample program will generate an enveloped signature of the document in the file `envelope.xml` and store it in the file `envelopedSignature.xml` in the current working directory.

Generating an XML Signature

This example shows you how to generate an XML Signature using the XML Digital Signature API. More specifically, the example generates an enveloped XML Signature of an XML document. An enveloped signature is a signature that is contained inside the content that it is signing. The example uses DOM (the Document Object Model) to parse the XML document to be signed and a JSR 105 DOM implementation to generate the resulting signature.

A basic knowledge of XML Signatures and their different components is helpful for understanding this section. See `http://www.w3.org/TR/xmldsig-core/` for more information.

Instantiating the Document to be Signed

First, we use a JAXP `DocumentBuilderFactory` to parse the XML document that we want to sign. An application obtains the default implementation for `DocumentBuilderFactory` by calling the following line of code:

```
DocumentBuilderFactory dbf =
  DocumentBuilderFactory.newInstance();
```

We must also make the factory namespace-aware:

```
dbf.setNamespaceAware(true);
```

Next, we use the factory to get an instance of a `DocumentBuilder`, which is used to parse the document:

```
DocumentBuilder builder = dbf.newDocumentBuilder();
Document doc = builder.parse(new FileInputStream(argv[0]));
```

Creating a Public Key Pair

We generate a public key pair. Later in the example, we will use the private key to generate the signature. We create the key pair with a `KeyPairGenerator`. In this example, we will create a DSA `KeyPair` with a length of 512 bytes :

```
KeyPairGenerator kpg = KeyPairGenerator.getInstance("DSA");
kpg.initialize(512);
KeyPair kp = kpg.generateKeyPair();
```

In practice, the private key is usually previously generated and stored in a `Key-Store` file with an associated public key certificate.

Creating a Signing Context

We create an XML Digital Signature `XMLSignContext` containing input parameters for generating the signature. Since we are using DOM, we instantiate a `DOM-SignContext` (a subclass of `XMLSignContext`), and pass it two parameters, the private key that will be used to sign the document and the root of the document to be signed:

```
DOMSignContext dsc = new DOMSignContext
  (kp.getPrivate(), doc.getDocumentElement());
```

Assembling the XML Signature

We assemble the different parts of the `Signature` element into an `XMLSignature` object. These objects are all created and assembled using an `XMLSignatureFac-tory` object. An application obtains a DOM implementation of `XMLSignature-Factory` by calling the following line of code:

```
XMLSignatureFactory fac =
  XMLSignatureFactory.getInstance("DOM");
```

We then invoke various factory methods to create the different parts of the `XML-Signature` object as shown below. We create a `Reference` object, passing to it the following:

- The URI of the object to be signed (We specify a URI of "", which implies the root of the document.)
- The `DigestMethod` (we use SHA1)

- A single Transform, the enveloped Transform, which is required for enveloped signatures so that the signature itself is removed before calculating the signature value

```
Reference ref = fac.newReference
  ("", fac.newDigestMethod(DigestMethod.SHA1, null),
    Collections.singletonList
      (fac.newTransform(Transform.ENVELOPED,
        (TransformParameterSpec) null)), null, null);
```

Next, we create the SignedInfo object, which is the object that is actually signed, as shown below. When creating the SignedInfo, we pass as parameters:

- The CanonicalizationMethod (we use inclusive and preserve comments)
- The SignatureMethod (we use DSA)
- A list of References (in this case, only one)

```
SignedInfo si = fac.newSignedInfo
  (fac.newCanonicalizationMethod
    (CanonicalizationMethod.INCLUSIVE_WITH_COMMENTS,
      (C14NMethodParameterSpec) null),
    fac.newSignatureMethod(SignatureMethod.DSA_SHA1, null),
    Collections.singletonList(ref));
```

Next, we create the optional KeyInfo object, which contains information that enables the recipient to find the key needed to validate the signature. In this example, we add a KeyValue object containing the public key. To create KeyInfo and its various subtypes, we use a KeyInfoFactory object, which can be obtained by invoking the getKeyInfoFactory method of the XMLSignature-Factory, as follows:

```
KeyInfoFactory kif = fac.getKeyInfoFactory();
```

We then use the KeyInfoFactory to create the KeyValue object and add it to a KeyInfo object:

```
KeyValue kv = kif.newKeyValue(kp.getPublic());
KeyInfo ki = kif.newKeyInfo(Collections.singletonList(kv));
```

Finally, we create the XMLSignature object, passing as parameters the Signed-Info and KeyInfo objects that we created earlier:

```
XMLSignature signature = fac.newXMLSignature(si, ki);
```

Notice that we haven’t actually generated the signature yet; we’ll do that in the next step.

Generating the XML Signature

Now we are ready to generate the signature, which we do by invoking the `sign` method on the `XMLSignature` object, and pass it the signing context as follows:

```
signature.sign(dsc);
```

The resulting document now contains a signature, which has been inserted as the last child element of the root element.

Printing or Displaying the Resulting Document

You can use the following code to print the resulting signed document to a file or standard output:

```
OutputStream os;
if (args.length > 1) {
  os = new FileOutputStream(args[1]);
} else {
  os = System.out;
}

TransformerFactory tf = TransformerFactory.newInstance();
Transformer trans = tf.newTransformer();
trans.transform(new DOMSource(doc), new StreamResult(os));
```

7

Using the Service Registry Web Console

THIS chapter describes the Web Console for the Service Registry (“the Registry”). It contains the following sections:

- Getting Started With the Web Console
- Searching the Registry
- Publishing and Managing Registry Objects

Getting Started With the Web Console

The Web Console is a web-based user interface that allows you to search the Registry and to publish content to the Registry and Repository. This section describes the preliminary steps to follow before you can perform these operations.

- Starting the Web Console
- Changing the Default Language

Starting the Web Console

To start the Web Console, type the following URL into a web browser:

```
http://hostname:port/soar/registry/thin/browser.jsp
```

For example:

```
http://localhost:8080/soar/registry/thin/browser.jsp
```

If the Registry is installed on your system, the *hostname* is `localhost`. If the Registry is not installed on your system, use the name of the system where the Registry is installed. The *port* value is usually 8080 unless there is a port conflict.

The Web Console has the following main sections:

- Top banner, where you can reset the locale, end the current session, and set the content language
- Menu area on the left side of the screen
- Registry Objects area to the right of the menu area, which displays found objects
- Detail area below the Registry Objects area, which displays the details for any found object

Note: At this release, the top right corner of the Web Console lists the Current User as Registry Guest. In fact, the user is Registry Operator; that is the user identity under which all objects will be published.

Changing the Default Language

You can change the default language for the display of two kinds of information:

- Web Console labels and messages
- Registry content

Changing the Default Language for Labels and Messages

The Web Console's labels and messages can be displayed in the languages listed in Table 7–1.

Table 7–1 Languages Supported by the Web Console

Language	Code
Simplified Chinese (China)	`zh_CN`
Traditional Chinese (Taiwan)	`zh_TW`
English (United States)	`en_US`
German	`de`
Japanese	`ja`
Korean	`ko`
Spanish	`es`
French (Canada)	`fr_CA`
Finnish	`fi`

To change the language from the default, follow these steps:

1. Add the language to your web browser language preferences by following the instructions for the web browser.

   For most browsers you can find the language settings in the General area of the Internet Options, Options, or Preferences dialog box.

2. Make the language your preferred language by placing it first in the list of languages.
3. Click the Reset Locale button.

   The labels appear in the appropriate language.

Changing the Default Language for Registry Content

You can publish content to the registry in any of the languages shown in the Content Language drop-down list in the top banner area, if the language is supported on your system. The default is the language setting for your web browser.

To change the language from the default, choose the language from the Content Language drop-down list.

Searching the Registry

The Search and Explore links in the menu area allow you to search the Registry.

- Using the Search Menu
- Selecting a Query
- Searching by Object Type
- Searching by Name and Description
- Searching by Classification
- Viewing Search Results
- Using the Explore Menu

Using the Search Menu

Click Search in the menu area. The Search form opens. It contains the following components:

- Select Predefined Query drop-down list
- Name text field
- Description text field
- ClassificationSchemes tree

Click Hide Search Form to close the Search form and clear the results area.

The next few sections describe how to use these components.

Selecting a Query

The Select Predefined Query drop-down list contains the items shown in Table 7–2.

Table 7–2 Predefined Queries

Query Name	**Search Purpose**
Basic Query	The default generic query, which allows you to search by object type, name, description, and classification
Basic Query - Case Sensitive	Case-sensitive version of Basic Query
FindAllMyObjects	Finds all objects owned (published) by the user who makes the query; may take a long time if the user owns many objects
GetCallersUser	Finds the `User` object for the user who makes the query

The default selection is Basic Query. The following sections describe how to perform basic queries:

- Searching by Object Type
- Searching by Name and Description
- Searching by Classification

Use the FindAllMyObjects and GetCallersUser queries to search for all the objects you have published and to view and modify data for the user you created when you registered.

Searching by Object Type

The simplest search is by object type only.

The default choice in the Object Type drop-down list is RegistryObject, which searches all objects in the Registry. To narrow the search, change the object type.

Follow these steps:

1. Choose an object type from the Object Type drop-down list.

2. Click the Search button.

The search returns all objects of the specified type. You can narrow the search by specifying a name, description, or classification.

Searching by Name and Description

To search by the name or description of an object, follow these steps:

1. From the Select Predefined Query drop-down list, select either Basic Query or Basic Query -- Case Sensitive.
2. Type a string in the Name or Description field.
3. Click Search.

By default, the search looks for a name or description that matches the entire string you typed. You can use wildcards to find a range of objects.

The wildcard characters are percent (`%`) and underscore (`_`).

The `%` wildcard matches multiple characters:

- Type `%off%` to return words that contain the string `off`, such as `Coffee`.
- Type `%nor` to return words that start with `Nor` or `nor`, such as `North` and `northern`.
- Type `ica%` to return all words that end with `ica`, such as `America`.

The underscore wildcard matches a single character. For example, the search string `_us_` would match objects named `Aus1` and `Bus3`.

Searching by Classification

Classification objects classify or categorize objects in the registry using unique concepts that define valid values within a classification scheme. The classification scheme is the parent in a tree hierarchy containing generations of child concepts. Table 7–3 describes the classification schemes provided by the Registry specifications. Many of the terms in this table are defined in the Registry specifications.

Table 7–3 Classification Scheme Usage

Classification Scheme Name	Usage	Description or Purpose
AssociationType	Frequently	Defines the types of associations between RegistryObjects. Used as the value of the `associationType` attribute of an `Association` instance to describe the nature of the association.
ContentManagementService	Rarely	Defines the types of content management services. Used in the configuration of a content management service, such as a validation or cataloging service.
DataType	Frequently	Defines the data types for attributes in classes defined by this document. Used as the value of the `slotType` attribute of a `Slot` instance to describe the data type of the `Slot` value.
DeletionScopeType	Occasionally	Defines the values for the `deletionScope` attribute of the `RemoveObjectsRequest` protocol message.
EmailType	Rarely	Defines the types of email addresses.
ErrorHandlingModel	Rarely	Defines the types of error handling models for content management services.
ErrorSeverityType	Rarely	Defines the different error severity types encountered by the Registry while processing protocol messages.
EventType	Occasionally	Defines the types of events that can occur in a registry.
InvocationModel	Rarely	Defines the different ways that a content management service may be invoked by the Registry.

Table 7–3 Classification Scheme Usage (Continued)

Classification Scheme Name	**Usage**	**Description or Purpose**
NodeType	Occasionally	Defines the different ways in which a `ClassificationScheme` may assign the value of the `code` attribute for its `ClassificationNodes`.
NotificationOptionType	Rarely	Defines the different ways in which a client may wish to be notified by the registry of an event within a `Subscription`.
ObjectType	Occasionally	Defines the different types of RegistryObjects a registry may support.
PhoneType	Rarely	Defines the types of telephone numbers.
QueryLanguage	Rarely	Defines the query languages supported by the Registry.
ResponseStatusType	Rarely	Defines the different types of status for a `RegistryResponse`.
StatusType	Occasionally	Defines the different types of status for a `RegistryResponse`.
SubjectGroup	Rarely	Defines the groups that a user can belong to for access control purposes.
SubjectRole	Rarely	Defines the roles that can be assigned to a user for access control purposes.

In the menu area, the root of the ClassificationScheme tree is below the Description field.

To search by classification, follow these steps:

1. Expand the root node to view the full list of classification schemes.

   The number in parentheses after each entry indicates how many concepts the parent contains.

2. Expand the node for the classification scheme you want to use.

3. Expand concept nodes beneath the classification scheme until you find the leaf node (a node with no concepts beneath it) by which you want to search.
4. Select the leaf node.
5. Optionally, restrict the search by choosing an object type or specifying a name or description string.
6. Click the Search button.

Viewing Search Results

Objects found by a search appear in the Registry Objects area.

The Registry Objects area consists of the following:

- Buttons labeled Save, Approve, Deprecate, Undeprecate, Relate, and Delete, which allow you to perform actions on objects. You must be the object's creator to perform any of these actions.
- A found objects display consisting of a search results table with the following columns:
  - Pick checkbox. Select any two objects to activate the Relate button. See Creating Relationships Between Objects (page 303) for details.
  - Details link. Click this link to open the Details area directly below the Registry Objects area (see "Viewing Object Details").
  - Object Type field.
  - Name field.
  - Description field.
  - Version field.
  - VersionComment field.
  - Pin checkbox. Select this checkbox to "pin" this object in place while you perform another search. You can then relate two different objects by selecting both objects.

Viewing Object Details

In the search results table, click the Details link for an object to open the Details area immediately below the Registry Objects area.

This section has a row of buttons and a row of tabs:

- The buttons are Save, Approve, Deprecate, Undeprecate, and Delete. The buttons represent actions you can perform on the object.
- The tabs represent the object's attributes. The tabs you see vary depending on the object type. Table 7–4 describes the tabs and the objects they apply to.

Table 7–4 Attribute Tabs in the Details Area

Tab Name	Applies To
Object Detail	All objects
Classifications	All objects
ExternalIdentifiers	All objects
Associations	All objects
ExternalLinks	All objects
Audit Trail	All objects
PostalAddresses	Organization, User
TelephoneNumbers	Organization, User
EmailAddresses	Organization, User
Users	Organization
Organizations	Organization
ServiceBindings	Service
SpecificationLinks	ServiceBinding
ChildConcepts	ClassificationScheme

Click a tab to find out if the object has any values for the attribute. If it does, click the Details link for the attribute value to open a web browser window with the details for the attribute value.

The Audit Trail tab does not produce a table with a Details link. Instead, it produces a table containing the event type, the date and time of the event, and the name of the User that caused it.

For every object, the Unique Identifier is an active link. Click this link to view the XML for the object in a web browser window. (All registry objects are stored in XML format.)

If the object is an ExternalIdentifier, the details panel has a Display Content link. Click this link to view the object in a web browser window.

If the object is an ExtrinsicObject, the details panel has a View Repository Item Content link. Click this link to view the repository item in a web browser window.

Using the Explore Menu

The Explore menu allows you to navigate through Registry and Repository content using the metaphor of a hierarchy of file folders. The root folder, named `root`, contains all Registry content, and is similar to the UNIX root directory.

To use the Explore menu, follow these steps:

1. Click the Explore link.
2. Expand the folder named “root”. It contains two subfolders: userData, where all user content is placed, and ClassificationSchemes.

To explore the classification schemes, follow these steps:

1. Click the ClassificationSchemes folder (not the node symbol). All the ClassificationScheme objects appear in the Registry Objects area. Follow the instructions in Viewing Search Results (page 291) to view the objects.
2. Expand the ClassificationSchemes node to open the Classification-Schemes tree hierarchy in the menu area.
3. Click any file icon to view that classification scheme in the Registry Objects area.
4. Expand a classification scheme node to see the Concept folders beneath it.

   Not all classification schemes have concepts that are viewable in the Explore menu. The last seven classification schemes have concepts that are not viewable here.
5. Click a Concept folder to view that concept in the Registry Objects area.

To explore the userData folder, follow these steps:

1. Expand the userData node.
2. Expand the RegistryObject node. Do not click the folder unless you want to view all registry objects.

   (The node named “folder1” has no content.)
3. Click a folder to view the registry objects of that type. Expand a node to view the object types at the next level.

When you have finished, click Hide Explorer to close the Explore menu and clear the results area.

Publishing and Managing Registry Objects

- Publishing Objects
- Adding a Classification to an Object
- Adding an External Identifier to an Object
- Adding an External Link to an Object
- Adding Custom Information to an Object Using Slots
- Changing the State of Objects
- Removing Objects
- Creating Relationships Between Objects

Publishing Objects

Publishing objects to the registry is a two-step process:

1. Create the object.
2. Save the object. The object does not appear in the Registry until after you save it.

At this release, ignore the Create User Account menu item. You can publish objects to the registry without performing any authentication steps.

To create and save a new registry object, follow these steps:

1. In the menu area, click Create a New Registry Object.

2. In the Registry Objects area, choose an object type from the drop-down list and click Add.
3. A Details form for the object appears in the Details area.
4. Type a name and description in the fields of the Details form. Type values for other fields that appear in the Details form.
5. Click Save to save the object.

   A status message appears, indicating whether the save was successful.

Either before or after you save the object, you can edit it by adding other objects to it. Table 7–4 lists the objects you can add. The following sections describe how to add these objects.

Adding a Classification to an Object

To create a classification, you use an *internal classification scheme*. An internal classification scheme contains a set of concepts whose values are known to the Registry.

To add a Classification to an object, search for the appropriate classification scheme, then choose a concept within that classification scheme. Follow these steps:

1. In the Details area for the object, click the Classifications button.

   The Classifications table (which may be empty) appears.

2. Click Add.

   A Details Panel window opens.

3. Type a name and description for the classification.
4. Click the Select ClassificationScheme or Concept button.

   A ClassificationScheme/Concept Selector window opens.

5. Expand the ClassificationSchemes node, then expand concept nodes until you have selected the leaf node you want to use.
6. Click OK to close the ClassificationScheme/Concept Selector window.

   The classification scheme and concept appear in the Details Panel window.

7. Click Add to close the Details Panel window.

   The new version of the object appears in the Registry Objects area.

8. Click Save in the Details area for the object.

Adding an External Identifier to an Object

To create an external identifier, you use an *external classification scheme*, one whose values are not known to the Registry because the classification scheme has no concepts.

To add an external identifier to an object, search for the appropriate classification scheme, then enter a value. Follow these steps:

1. In the Details area for the object, click the ExternalIdentifiers tab.

   The ExternalIdentifiers table (which may be empty) appears.

2. Click Add.

   A Details Panel window opens.

3. Type a name and description for the external identifier.
4. Click the Select ClassificationScheme button.

   A ClassificationScheme/Concept Selector window opens.

5. Expand the ClassificationSchemes node, then expand concept nodes until you have selected the leaf node you want to use.
6. Click OK to close the ClassificationScheme/Concept Selector window.

   The classification scheme and concept appear in the Details Panel window.

7. Click Add to close the Details Panel window.

   The new version of the object appears in the Registry Objects area.

8. Click Save in the Details area for the object.

Adding an External Link to an Object

An external link allows you to associate a URI with a registry object.

To add an external link to an object, follow these steps:

1. In the Details area for the object, click the ExternalLinks tab.

   The ExternalLinks table (which may be empty) appears.

2. Click Add.

   A Details Panel window opens.

3. Type a name for the external link.
4. Type the URL for the external link in the External URI field.
5. Optionally, click the Select Concept for Object Type button if you want to specify the type of content to which the URL points.

   Expand the ClassificationSchemes node and locate the content type by expanding the ObjectType, RegistryObject, and ExtrinsicObject nodes. Select the concept, then click OK. If you do not find a suitable type, click Cancel. You can create a new concept for ExtrinsicObjects if you wish.
6. Click Add to close the Details Panel window.

   The new version of the object appears in the Registry Objects area.
7. Click Save in the Details area for the object.

Adding Custom Information to an Object Using Slots

A slot contains extra information that would otherwise not be stored in the Registry. Slots provide a way to add arbitrary attributes to objects.

To add a slot to an object, follow these steps:

1. In the Details area for the object, click the Slots tab.

   The Slots table (which may be empty) appears.
2. Click Add.

   A Details Panel window opens.
3. Type a name for the Slot.
4. Optionally, type a value in the Slot Type field. You may use this field to specify a data type for the slot or to provide a way to group slots together.
5. Type a value in the Values field.
6. Click Add to close the Details Panel window.

   The new version of the object appears in the Registry Objects area.
7. Click Save in the Details area for the object.

Adding a Postal Address to an Organization or User

An Organization or User can have one or more postal addresses. To add a postal address to either an Organization or a User, follow these steps:

1. In the Details area for the Organization or User, click the PostalAddresses tab.

   The PostalAddresses table (which may be empty) appears.

2. Click Add.

   A Details Panel window opens.

3. Type values in the fields. All fields except Street are optional.
   - Street Number
   - Street (required)
   - City
   - State or Province
   - Country
   - Postal Code

4. Click Add to close the Details Panel window.

   The new version of the object appears in the Registry Objects area.

5. Click Save in the Details area for the object.

Adding a Telephone Number to an Organization or User

An Organization or User can have one or more telephone numbers. To add a telephone number to either an Organization or a User, follow these steps:

1. In the Details area for the Organization or User, click the TelephoneNumbers tab.

   The TelephoneNumbers table (which may be empty) appears.

2. Click Add.

   A Details Panel window opens.

3. Select a value from the Type combo box (Beeper, FAX, HomePhone, MobilePhone, or OfficePhone).
4. Type values in the fields. All fields except Phone Number are optional.
   - Country Code
   - Area Code
   - Phone Number (required)
   - Extension
5. Click Add to close the Details Panel window.

   The new version of the object appears in the Registry Objects area.
6. Click Save in the Details area for the object.

Adding an Email Address to an Organization or User

An Organization or User can have one or more email addresses. To add an email address to either an Organization or a User, follow these steps:

1. In the Details area for the Organization or User, click the EmailAddresses tab.

   The EmailAddresses table (which may be empty) appears.
2. Click Add.

   A Details Panel window opens.
3. Select a value from the Type combo box (HomeEmail or OfficeEmail).
4. Type a value in the Email Address field.
5. Click Add to close the Details Panel window.

   The new version of the object appears in the Registry Objects area.
6. Click Save in the Details area for the object.

Adding a User to an Organization

An Organization can have one or more users. One user is the primary contact, which is normally the user that created the organization. To create a new user and add it to an Organization, follow these steps:

1. In the Details area for the Organization, click the Users tab.

The Users table appears.

2. Click Add.

   A Details Panel window opens.

3. In the Name field, type the last name of the user to the left of the comma and (optionally) the first and middle names to the right of the comma.
4. Optionally, type a description of the user in the Description field.
5. In the First Name, Middle Name, and Last Name fields, type the first name, middle name, and surname of the user. (All fields are optional.)
6. Click Add to close the Details Panel window.

   The new version of the organization appears in the Registry Objects area, and the user is created.

7. Click Save in the Details area for the object.

Adding a Child Organization to an Organization

An Organization can have one or more child organizations. To add a child organization to an Organization, follow these steps:

1. In the Details area for the Organization, click the Organizations tab.

   The Organizations table appears.

2. Click Add.

   A Details Panel window opens.

3. In the Name field, type a name for the new organization.
4. Optionally, type a description in the Description field.
5. Type values in the address fields. All fields except Street are optional.
   - Street Number
   - Street (required)
   - City
   - State or Province
   - Country
   - Postal Code
6. Click Add to close the Details Panel window.

The new version of the object appears in the Registry Objects area, and the new Organization is created.

7. Click Save in the Details area for the object.

Adding a Service Binding to a Service

A Service normally has one or more service bindings. To add a service binding to a Service, follow these steps:

1. In the Details area for the Service, click the ServiceBindings tab.

   The ServiceBindings table appears.

2. Click Add.

   A Details Panel window opens.

3. In the Name field, type a name for the service binding.
4. Optionally, type a description of the service binding in the Description field.
5. In the Access URI field, type the URL for the service binding.
6. Click Add to close the Details Panel window.

   The new version of the object appears in the Registry Objects area.

7. Click Save in the Details area for the object.

Adding a Specification Link to a Service Binding

A ServiceBinding normally has a SpecificationLink object. To add a SpecificationLink to a ServiceBinding, follow these steps:

1. In the Details area for the ServiceBinding, click the SpecificationLinks tab.

   The SpecificationLinks table appears.

2. Click Add.

   A Details Panel window opens.

3. In the Name field, type a name for the SpecificationLink.
4. Optionally, type a description of the SpecificationLink in the Description field.

5. In the Usage Description field, type a usage description for the usage parameters, if there are any.
6. In the Usage Parameters field, type the usage parameters, if there are any.
7. Click Add to close the Details Panel window.

   The new version of the object appears in the Registry Objects area.
8. Click Save in the Details area for the object.

Adding a Child Concept to a Classification Scheme or Concept

A ClassificationScheme normally has numerous child concepts (which can in turn have child concepts). To add a child concept to a ClassificationScheme, follow these steps:

1. In the Details area for the ClassificationScheme, click the Concepts tab.

   The Concepts table appears.
2. Click Add.

   A Details Panel window opens.
3. In the Name field, type a name for the concept.
4. Optionally, type a description of the concept in the Description field.
5. In the Value field, type a value for the concept.
6. Click Add to close the Details Panel window.

   The new version of the object appears in the Registry Objects area.
7. Click Save in the Details area for the object.

Changing the State of Objects

In addition to saving, editing, and removing objects, you can perform the following actions on them if you are the owner or are otherwise authorized to do so:

- Approval
- Deprecation
- Undeprecation

These features are useful in a production environment if you want to establish a version control policy for registry objects. For example, you can approve a version of an object for general use and deprecate an obsolete version before you remove it. If you change your mind after deprecating an object, you can undeprecate it.

You perform all these actions in the Search Results area.

- To approve an object, select it and click the Approve button. A message verifying the approval appears, and the event is added to the Audit Trail.
- To deprecate an object, select it and click the Deprecate button. A message verifying the deprecation appears, and the event is added to the Audit Trail.
- To undeprecate an object, select it and click the Undeprecate button. A message verifying the undeprecation appears, and the event is added to the Audit Trail.

Removing Objects

To remove an object you own from the Registry, select the object and click the Delete button.

If the object is an extrinsic object, you have two choices.

- Choose Delete Object and Repository Item (the default) from the Deletion Options menu to delete both the ExtrinsicObject registry object and the repository item to which it refers.
- Choose Delete Repository Item Only to delete the repository item and leave the ExtrinsicObject in the Registry. You can then add another repository item.

The Deletion Options menu is meaningful only for extrinsic objects.

Creating Relationships Between Objects

There are two kinds of relationships between objects: references and associations. They are both *unidirectional*. That is, each has a source object and a target object.

The Registry supports references, called ObjectRefs, between certain types of objects. For example, if you create a Service and a ServiceBinding, you can create a ServiceBinding reference from the Service to the ServiceBinding. However, you cannot create a reference from the ServiceBinding to the Service. A Reference is not a registry object.

An Association is a registry object, and you can create an Association from any registry object to any other. The Registry supports an AssociationType classification scheme that includes a number of predefined association types: OffersService, RelatedTo, HasMember, and so on. You can also create new association types. Associations between registry objects that you own are called *intramural associations*. Associations in which you do not own one or both of the objects are called *extramural associations*. If you create an Organization and add a Service to it, an Association of type OffersService is automatically created from the Organization to the Service.

If no valid reference exists for the source and target objects, you cannot create a reference.

You use the Relate button in the Registry Objects area to relate two objects. This button becomes active when you select two objects in the search results table.

If the two objects are not both visible in the search results table, select the Pin checkbox to hold one object in the search results table while you find the object to which you want to relate it.

Creating References

To create a Reference, follow these steps:

1. In the Registry Objects area, select two objects and click Relate.
2. In the Create Relationship area, select the source object if it is not already selected.

   The other object becomes the target object.
3. If a valid reference exists for the source and target objects, the Reference option is selected by default, and the valid reference attribute appears. If no valid reference exists for the source and target objects, the Reference radio button is grayed out.
4. Click Save to save the Reference.

Creating Associations

To create an Association, follow these steps:

1. In the Registry Objects area, select two objects and click Relate.
2. In the Create Relationship area, select the source object if it is not already selected.

The other object becomes the target object.

3. Select the Association radio button, if it is not already selected.
4. Type a name and description for the Association in the Details area.

   The source and target object ID values are already filled in.
5. Choose a type value from the Association Type menu.
6. Click Save to save the Association.

8

Developing Clients for the Service Registry

THIS chapter describes how to use the Java API for XML Registries (JAXR) to access the Service Registry (“the Registry”).

After providing a brief overview of JAXR and the examples described in this chapter, this chapter describes how to implement a JAXR client to query the Registry and publish content to the Registry and its associated repository.

Overview of JAXR

This section provides a brief overview of JAXR. It covers the following topics:

- About Registries and Repositories
- About JAXR
- JAXR Architecture
- About the Examples

About Registries and Repositories

An XML *registry* is an infrastructure that enables the building, deployment, and discovery of web services. It is a neutral third party that facilitates dynamic and loosely coupled business-to-business (B2B) interactions. A registry is available to organizations as a shared resource, normally in the form of a web-based service.

Currently there are a variety of specifications for XML registries. These include

- The ebXML Registry and Repository standard, which is sponsored by the Organization for the Advancement of Structured Information Standards (OASIS) and the United Nations Centre for the Facilitation of Procedures and Practices in Administration, Commerce and Transport (U.N./CEFACT). *ebXML* stands for Electronic Business using eXtensible Markup Language.
- The Universal Description, Discovery, and Integration (UDDI) project, which is developed by a vendor consortium

A *registry provider* is an implementation of a registry that conforms to a specification for XML registries.

While a UDDI registry stores information about businesses and the services they offer, an ebXML registry has a much wider scope. It is a *repository* as well as a registry. A repository stores arbitrary content as well as information about that content. In other words, a repository stores data as well as metadata. The ebXML Registry standard defines an interoperable Enterprise Content Management (ECM) API for web services.

An ebXML registry and repository is to the web what a relational database is to enterprise applications: it provides a means for web services and web applications to store and share content and metadata.

An ebXML registry can be part of a registry *federation*, an affiliated group of registries. For example, the health ministry of a country in Europe could operate a registry, and that registry could be part of a federation that included the registries of other European health ministries.

About JAXR

JAXR enables Java software programmers to use a single, easy-to-use abstraction API to access a variety of XML registries. A unified JAXR information model describes content and metadata within XML registries.

JAXR gives developers the ability to write registry client programs that are portable across various target registries. JAXR also enables value-added capabilities beyond those of the underlying registries.

The current version of the JAXR specification includes detailed bindings between the JAXR information model and the ebXML Registry specifications. You can find the latest version of the JAXR specification at

```
http://java.sun.com/xml/downloads/jaxr.html
```

The Service Registry includes a JAXR provider that implements the level 1 capability profile, which allows full access to ebXML registries. The ebXML specifications and the JAXR specification are not in perfect alignment, because the ebXML specifications have advanced beyond the JAXR specification. For this reason, the JAXR provider for the Registry includes some additional implementation-specific methods that implement the ebXML specifications and that are likely to be included in the next version of the JAXR specification.

JAXR Architecture

The high-level architecture of JAXR consists of the following parts:

- *A JAXR client*: This is a client program that uses the JAXR API to access a registry through a JAXR provider.
- *A JAXR provider*: This is an implementation of the JAXR API that provides access to a specific registry provider or to a class of registry providers that are based on a common specification. This guide does not describe how to implement a JAXR provider.

A JAXR provider implements two main packages:

- `javax.xml.registry`, which consists of the API interfaces and classes that define the registry access interface.
- `javax.xml.registry.infomodel`, which consists of interfaces that define the information model for JAXR. These interfaces define the types of objects that reside in a registry and how they relate to each other. The basic interface in this package is the `RegistryObject` interface.

The most basic interfaces in the `javax.xml.registry` package are

- `Connection`. The `Connection` interface represents a client session with a registry provider. The client must create a connection with the JAXR provider in order to use a registry.
- `RegistryService`. The client obtains a `RegistryService` object from its connection. The `RegistryService` object in turn enables the client to obtain the interfaces it uses to access the registry.

The primary interfaces, also part of the `javax.xml.registry` package, are

- `QueryManager` and `BusinessQueryManager`, which allow the client to search a registry for information in accordance with the `javax.xml.registry.infomodel` interfaces. An optional interface, `DeclarativeQueryManager`, allows the client to use SQL syntax for queries. The ebXML provider for the Registry implements `DeclarativeQueryManager`.
- `LifeCycleManager` and `BusinessLifeCycleManager`, which allow the client to modify the information in a registry by either saving it (updating it) or deleting it.

When an error occurs, JAXR API methods throw a `JAXRException` or one of its subclasses.

Many methods in the JAXR API use a `Collection` object as an argument or a returned value. Using a `Collection` object allows operations on several registry objects at a time.

Figure 8–1 illustrates the architecture of JAXR. For the Registry, a JAXR client uses the capability level 0 and level 1 interfaces of the JAXR API to access the JAXR provider, which is an ebXML provider. The JAXR provider in turn accesses the Registry, an ebXML registry.

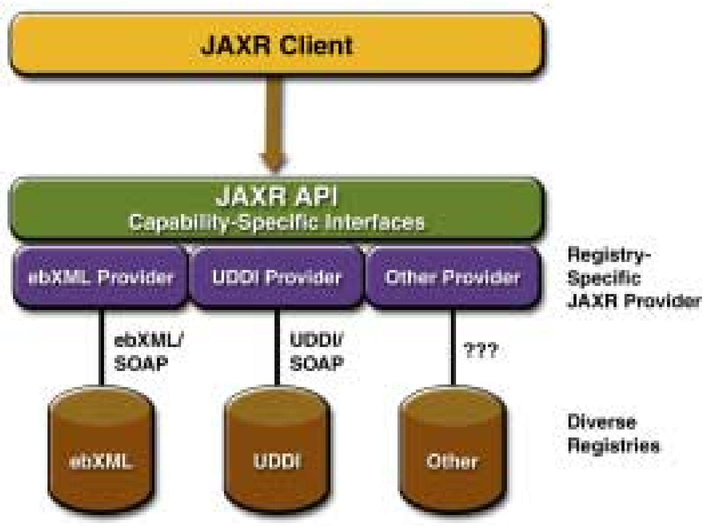

Figure 8–1 JAXR Architecture

About the Examples

Many sample client programs that demonstrate JAXR features are available as part of the Java Web Services Developer Pack (Java WSDP). If you install the Java WSDP, you will find them in the directory `<INSTALL>/registry/samples`. (`<INSTALL>` is the directory where you installed the Java WSDP.)

Each example or group of examples has a `build.xml` file that allows you to compile and run each example using the Ant tool. Each `build.xml` file has a `compile` target and one or more targets that run the example or examples. Some of the run targets take command-line arguments.

Before you run the examples, you must edit two files in the directory `<INSTALL>/registry/samples/common`. The file `build.properties` is used by the Ant targets that run the programs. The file `JAXRExamples.properties` is a resource bundle that is used by the programs themselves.

In addition, a `targets.xml` file in the `<INSTALL>/registry/samples/common` directory defines the classpath for compiling and running the examples. It also contains a `clean` target that deletes the `build` directory created when each example is compiled.

Because Tomcat and the Sun Java System Application Server Platform Edition 8.1 have different file structures, there are two versions of the `build.properties` and `targets.xml` files, with the suffix `tomcat` for Tomcat and the suffix `as` for the Application Server.

Edit the file `build.properties.as` as follows:

1. Set the property `container.home` to the location of Sun Java System Application Server Platform Edition 8.1.
2. Set the property `registry.home` to the directory where you installed the Java WSDP.
3. Set the properties `proxyHost` and `proxyPort` to specify your own proxy settings. The proxy host is the system on your network through which you access the Internet. You usually specify the proxy host in your web browser settings.

   The proxy port has the value 8080, which is the usual one. Change this string if your proxy uses a different port. Your entries usually follow this pattern:

   ```
   proxyHost=proxyhost.mydomain
   proxyPort=8080
   ```

Edit the file `build.properties.tomcat` as follows:

1. Set the property `tomcat.home` to the directory where you installed the Java WSDP.
2. Set the properties `proxyHost` and `proxyPort` to specify your own proxy settings. The proxy host is the system on your network through which you access the Internet. You usually specify the proxy host in your web browser settings.

   The proxy port has the value 8080, which is the usual one. Change this string if your proxy uses a different port. Your entries usually follow this pattern:

   ```
   proxyHost=proxyhost.mydomain
   proxyPort=8080
   ```

Copy the files as follows:

1. Copy the file you edited (`build.properties.tomcat` or `build.properties.as`) to `build.properties`.
2. Copy the corresponding `targets.xml` file (`targets.xml.tomcat` or `targets.xml.as`) to `targets.xml`.

Edit the file `JAXRExamples.properties` as follows:

1. Edit the properties `query.url` and `publish.url` to specify the URL of the Registry. The file provides a default setting of `localhost:8080` for the host and port, but you may need to change this to another host or port if the Registry is installed on a remote server or at a non-default port.
2. Edit the following properties to specify the properties required for logging in to the Registry.

   ```
   security.keystorePath=
   security.storepass=ebxmlrr
   security.alias=
   security.keypass=
   ```

   The `security.keystorePath` property specifies the location of the keystore file. The `security.storepass` property has a default setting of `ebxmlrr`. The `security.alias` and `security.keypass` properties are the alias and password you specify when you use the User Registration Wizard of the Java UI. See Getting Access to the Registry (page 7) for details.

3. Feel free to change any of the data in the remainder of the file as you experiment with the examples. The Ant targets that run the client examples always use the latest version of the file.

Setting Up a JAXR Client

This section describes the first steps to follow to implement a JAXR client that can perform queries and updates to the Service Registry. A JAXR client is a client program that can access registries using the JAXR API. This section covers the following topics:

- Starting the Registry
- Getting Access to the Registry
- Establishing a Connection to the Registry
- Obtaining and Using a RegistryService Object

Starting the Registry

To start the Registry, you start the container into which you installed the Registry: Tomcat or the Sun Java System Application Server.

Getting Access to the Registry

Any user of a JAXR client can perform queries on the Registry for objects that are not restricted by an access control policy. To perform queries for restricted objects, to add data to the Registry, or to update Registry data, however, a user must obtain permission from the Registry to access it. The Registry uses client-certificate authentication for user access.

To create a user that can submit data to the Registry, use the User Registration Wizard of the Web Console that is part of the Registry software. You can also use an existing certificate obtained from a certificate authority.

You will specify your user name and password for some of the JAXR client example programs, along with information about the location of your certificate.

Establishing a Connection to the Registry

The first task a JAXR client must complete is to establish a connection to a registry. Establishing a connection involves the following tasks:

- Creating a Connection Factory
- Creating a Connection

Creating a Connection Factory

A client creates a connection from a connection factory.

To use JAXR in a stand-alone client program, you must obtainan instance of the abstract class `ConnectionFactory`. To do so, call the `getConnectionFactory` method in the JAXR provider's `JAXRUtility` class.

```
import org.freebxml.omar.client.xml.registry.util.JAXRUtility;
...
ConnectionFactory factory = JAXRUtility.getConnectionFactory();
```

Creating a Connection

To create a connection, a client first creates a set of properties that specify the URL or URLs of the registry or registries being accessed. The following code provides the URLs of the query service and publishing service for the Registry if the Registry is deployed on the local system. (There should be no line break in the strings.)

```
Properties props = new Properties();
props.setProperty("javax.xml.registry.queryManagerURL",
  "http://localhost:8080/omar/registry/soap");
props.setProperty("javax.xml.registry.lifeCycleManagerURL",
  "http://localhost:8080/omar/registry/soap");
```

The client then obtains the connection factory as described in Creating a Connection Factory (page 8), sets its properties, and creates the connection. The following code fragment performs these tasks:

```
ConnectionFactory factory =
  JAXRUtility.getConnectionFactory();
factory.setProperties(props);
Connection connection = factory.createConnection();
```

The `makeConnection` method in the sample programs shows the steps used to create a JAXR connection.

Table 8–1 lists and describes the two properties you can set on a connection. These properties are defined in the JAXR specification.

Table 8–1 Standard JAXR Connection Properties

Property Name and Description	Data Type	Default Value
`javax.xml.registry.queryManagerURL` Specifies the URL of the query manager service within the target registry provider.	String	None
`javax.xml.registry.lifeCycleManagerURL` Specifies the URL of the life-cycle manager service within the target registry provider (for registry updates).	String	Same as the specified `queryManagerURL` value

Obtaining and Using a RegistryService Object

After creating the connection, the client uses the connection to obtain a `RegistryService` object and then the interface or interfaces it will use:

```
RegistryService rs = connection.getRegistryService();
BusinessQueryManager bqm = rs.getBusinessQueryManager();
BusinessLifeCycleManager blcm =
  rs.getBusinessLifeCycleManager();
```

Typically, a client obtains a `BusinessQueryManager` object and either a `LifeCycleManager` or a `BusinessLifeCycleManager` object from the `RegistryService` object. If it is using the Registry for simple queries only, it may need to obtain only a `BusinessQueryManager` object.

Querying a Registry

This section describes the interfaces and methods JAXR provides for querying a registry. It covers the following topics:

- Using iterative queries
- Basic Query Methods
- JAXR Information Model Interfaces
- Finding Objects by Name
- Finding Objects by Type
- Finding Objects by Classification
- Finding Objects by External Identifier
- Finding Objects by External Link
- Finding Objects by Unique Identifier
- Finding Objects You Published
- Retrieving Information About an Object
- Using Declarative Queries
- Invoking Stored Queries
- Querying a Registry Federation

Basic Query Methods

The simplest way for a client to use a registry is to query it for information about the objects and data in it. The `QueryManager`, `BusinessQueryManager`, and `RegistryObject` interfaces support a number of finder and getter methods that allow clients to search for data using the JAXR information model. Many of the finder methods return a `BulkResponse` (a collection of objects) that meets a set of criteria specified in the method arguments. The most general of these methods are as follows:

- `getRegistryObject` and `getRegistryObjects`, `QueryManager` methods that return one or more objects based on their type or unique identifier, or return the objects owned by the caller (for information on unique identifiers, see Finding Objects by Unique Identifier, page 22)
- `findObjects`, an implementation-specific `BusinessQueryManager` method that returns a list of all objects of a specified type that meet the specified criteria

Other finder methods allow you to find specific kinds of objects supported by the JAXR information model. While a UDDI registry supports a specific hierarchy of objects (organizations, which contain users, services, and service bindings), an ebXML registry permits the storage of freestanding objects of various types that can be linked to each other in various ways. Other objects are not freestanding but are always attributes of another object.

The `BusinessQueryManager` finder methods are useful primarily for searching UDDI registries. The more general `findObjects` method and the `RegistryObject` getter methods are more appropriate for the Service Registry.

To execute queries, you do not need to log in to the Registry. By default, an unauthenticated user has the identity of the user named “Registry Guest.”

JAXR Information Model Interfaces

Table 8–2 lists the main interfaces supported by the JAXR information model. All these interfaces extend the `RegistryObject` interface.

Table 8–2 JAXR `RegistryObject` Subinterfaces

Object Type	Description
`Association`	Defines a relationship between two objects. Getter/finder methods: `RegistryObject.getAssociations`, `BusinessQueryManager.findAssociations`, `BusinessQueryManager.findCallerAssociations`.
`AuditableEvent`	Provides a record of a change to an object. A collection of `AuditableEvent` objects constitutes an object’s audit trail. Getter method: `RegistryObject.getAuditTrail`.
`Classification`	Classifies an object using a `ClassificationScheme`. Getter method: `RegistryObject.getClassifications`.

Table 8–2 JAXR `RegistryObject` Subinterfaces (Continued)

Object Type	**Description**
`ClassificationScheme`	Represents a taxonomy used to classify objects. An internal `ClassificationScheme` is one in which all taxonomy elements (concepts) are defined in the registry. An external `ClassificationScheme` is one in which the values are not defined in the registry but are represented using an `ExternalIdentifier`. Finder methods: `BusinessQueryManager.findClassificationSchemes`, `BusinessQueryManager.findClassificationSchemeByName`.
`Concept`	Represents a taxonomy element and its structural relationship with other elements in an internal `ClassificationScheme`. Called a `ClassificationNode` in the ebXML specifications. Finder methods: `BusinessQueryManager.findConcepts`, `BusinessQueryManager.findConceptByPath`.
`ExternalIdentifier`	Provides a value for the content of an external `ClassificationScheme`. Getter method: `RegistryObject.getExternalIdentifiers`.
`ExternalLink`	Provides a URI for content that may reside outside the registry. Getter method: `RegistryObject.getExternalLinks`.
`ExtrinsicObject`	Provides metadata that describes submitted content whose type is not intrinsically known to the registry and therefore must be described by means of additional attributes (such as mime type). No specific getter/finder methods.
`Organization`	Provides information about an organization. May have a parent, and may have one or more child organizations. Always has a `User` object as a primary contact, and may offer `Service` objects. Finder method: `BusinessQueryManager.findOrganizations`.
`RegistryPackage`	Represents a logical grouping of registry objects. A `RegistryPackage` may have any number of `RegistryObjects`. Getter/finder methods: `RegistryObject.getRegistryPackages`, `BusinessQueryManager.findRegistryPackages`.
`Service`	Provides information on a service. May have a set of `ServiceBinding` objects. Finder method: `BusinessQueryManager.findServices`.

Table 8–2 JAXR `RegistryObject` Subinterfaces (Continued)

Object Type	**Description**
`ServiceBinding`	Represents technical information on how to access a `Service`. Getter/finder methods: `Service.getServiceBindings`, `BusinessQueryManager.findServiceBindings`.
`Slot`	Provides a dynamic way to add arbitrary attributes to `RegistryObject` instances. Getter methods: `RegistryObject.getSlot`, `RegistryObject.getSlots`.
`SpecificationLink`	Provides the linkage between a `ServiceBinding` and one of its technical specifications that describes how to use the service using the `ServiceBinding`. Getter method: `ServiceBinding.getSpecificationLinks`.
`User`	Provide information about registered users within the registry. `User` objects are affiliated with `Organization` objects. Getter methods: `Organization.getUsers`, `Organization.getPrimaryContact`.

Table 8–3 lists the other interfaces supported by the JAXR information model. These interfaces provide attributes for the main registry objects. They do not themselves extend the `RegistryObject` interface.

Table 8–3 JAXR Object Types Used as Attributes

Object Type	**Description**
`EmailAddress`	Represents an email address. A `User` may have an `EmailAddress`. Getter method: `User.getEmailAddresses`.
`InternationalString`	Represents a `String` that has been internationalized into several locales. Contains a `Collection` of `LocalizedString` objects. The name and description of a `RegistryObject` are `InternationalString` objects. Getter methods: `RegistryObject.getName`, `RegistryObject.getDescription`.
`Key`	Represents a unique key that identifies a `RegistryObject`. Must be a DCE 128 UUID (Universal Unique IDentifier). Getter method: `RegistryObject.getKey`.

Table 8–3 JAXR Object Types Used as Attributes (Continued)

Object Type	Description
LocalizedString	A component of an InternationalString that associates a String with its Locale. Getter method: International-String.getLocalizedStrings.
PersonName	Represents a person's name. A User has a PersonName. Getter method: User.getPersonName.
PostalAddress	Represents a postal address. An Organization or User may have one or more PostalAddress objects. Getter methods: Organization.getPostalAddress, Organization-Impl.getPostalAddresses (implementation-specific), User.getPostalAddresses.
TelephoneNumber	Represents a telephone number. An Organization or a User may have one or more TelephoneNumber objects. Getter methods: Organization.getTelephoneNumbers, User.getTelephoneNumbers.

Finding Objects by Name

To search for objects by name, you normally use a combination of find qualifiers (which affect sorting and pattern matching) and name patterns (which specify the strings to be searched). The BusinessQueryManagerImpl.findObjects method takes a collection of FindQualifier objects as its second argument and takes a collection of name patterns as its third argument. Its method signature is as follows:

```
public BulkResponse findObjects(java.lang.String objectType,
     java.util.Collection findQualifiers,
     java.util.Collection namePatterns,
     java.util.Collection classifications,
     java.util.Collection specifications,
     java.util.Collection externalIdentifiers,
     java.util.Collection externalLinks)
  throws JAXRException
```

You can use wildcards in a name pattern. Use percent signs (%) to specify that the query string occurs at the beginning, end, or middle of the object name.

For example, the following code fragment finds all the organizations in the Registry whose names begin with a specified string, qString, and sorts them in alphabetical order.

```
// Define find qualifiers and name patterns
Collection findQualifiers = new ArrayList();
findQualifiers.add(FindQualifier.SORT_BY_NAME_ASC);
Collection namePatterns = new ArrayList();
namePatterns.add(qString + "%");

// Find organizations with name that starts with qString
BulkResponse response =
  bqm.findObjects("Organization", findQualifiers,
     namePatterns, null, null, null, null);
Collection orgs = response.getCollection();
```

The findObjects method is not case-sensitive, unless you specify FindQualifier.CASE_SENSITIVE_MATCH. In the previous fragment, the first argument could be either "Organization" or "organization", and the name pattern matches names regardless of case.

The following code fragment performs a case-sensitive search for all registry objects whose names contain the string qString and sorts them in alphabetical order.

```
Collection findQualifiers = new ArrayList();
findQualifiers.add(FindQualifier.CASE_SENSITIVE_MATCH);
findQualifiers.add(FindQualifier.SORT_BY_NAME_ASC);
Collection namePatterns = new ArrayList();
namePatterns.add("%" + qString + "%");

// Find objects with name that contains qString
BulkResponse response =
  bqm.findObjects("RegistryObject", findQualifiers,
     namePatterns, null, null, null, null);
Collection orgs = response.getCollection();
```

The percent sign matches any number of characters in the name. To match a single character, use the underscore (_). For example, to match both “Arg1” and “Org2” you would specify a name pattern of _rg_.

Finding Objects by Name: Example

For an example of finding objects by name, see the example `<`*`INSTALL`*`>/registry/samples/query-name/src/JAXRQueryByName.java`. To run the example, follow these steps:

1. Go to the directory `<`*`INSTALL`*`>/registry/samples/query-name`.
2. Type the following command, specifying a string value:

   `ant run -Dname=`*`string`*

The program performs a case-insensitive search, returning all objects whose names contain the specified string. It also displays the object's classifications, external identifiers, external links, slots, and audit trail.

Finding Objects by Type

To find all objects of a specified type, specify only the first argument of the `BusinessQueryManagerImpl.findObjects` method and, optionally, a collection of `FindQualifier` objects. For example, if `typeString` is a string whose value is either `"Service"` or `"service"`, the following code fragment will find all services in the Registry and sort them in alphabetical order.

```
Collection findQualifiers = new ArrayList();
findQualifiers.add(FindQualifier.SORT_BY_NAME_ASC);

BulkResponse response = bqm.findObjects(typeString,
  findQualifiers, null, null, null, null, null);
```

You cannot use wildcards in the first argument to `findObjects`.

Finding Objects by Type: Example

For an example of finding objects by type, see the example `<`*`INSTALL`*`>/registry/samples/query-object-type/src/JAXRQueryByObjectType.java`. To run the example, follow these steps:

1. Go to the directory `<`*`INSTALL`*`>/registry/samples/query-object-type`.
2. Type the following command, specifying a string value:

   `ant run -Dtype=`*`type_name`*

The program performs a case-insensitive search, returning all objects whose type is *type_name* and displaying their names, descriptions, and unique identifiers. Specify the exact name of the type, not a wildcard, as in the following command line:

```
ant run -Dtype=federation
```

Finding Objects by Classification

To find objects by classification, you establish the classification within a particular classification scheme and then specify the classification as an argument to the `BusinessQueryManagerImpl.findObjects` method.

To do this, you first find the classification scheme and then create a `Classification` object to be used as an argument to the `findObjects` method or another finder method.

The following code fragment finds all organizations that correspond to a particular classification within the ISO 3166 country codes classification system maintained by the International Organization for Standardization (ISO). See `http://www.iso.org/iso/en/prods-services/iso3166ma/index.html` for details. (This classification scheme is provided in the sample database included with the Registry.)

```
ClassificationScheme cScheme =
  bqm.findClassificationSchemeByName(null,
    "iso-ch:3166:1999");

Classification classification =
  blcm.createClassification(cScheme, "United States", "US");
Collection classifications = new ArrayList();
classifications.add(classification);
// perform query
BulkResponse response = bqm.findObjects("Organization", null,
  null, classifications, null, null, null);
Collection orgs = response.getCollection();
```

The ebXML Registry Information Model Specification requires a set of canonical classification schemes to be present in an ebXML registry. Each scheme also has a set of required concepts (called ClassificationNodes in the ebXML specifications). The primary purpose of the canonical classification schemes is not to classify objects but to provide enumerated types for object attributes. For example, the `EmailType` classification scheme provides a set of values for the `type` attribute of an `EmailAddress` object.

Table 8–4 lists and describes these canonical classification schemes.

Table 8–4 Canonical Classification Schemes

Classification Scheme	**Description**
`AssociationType`	Defines the types of associations between `RegistryObjects`.
`ContentManagementService`	Defines the types of content management services.
`DataType`	Defines the data types for attributes in classes defined by the specification.
`DeletionScopeType`	Defines the values for the `deletionScope` attribute in the `RemoveObjectsRequest` protocol message.
`EmailType`	Defines the types of email addresses.
`ErrorHandlingModel`	Defines the types of error handling models for content management services.
`ErrorSeverityType`	Defines the different error severity types encountered bythe registry during processing of protocol messages.
`EventType`	Defines the types of events that can occur in a registry.
`InvocationModel`	Defines the different ways that a content management service may be invoked by the registry.
`NodeType`	Defines the different ways in which a `ClassificationScheme` may assign the value of the code attribute for its `ClassificationNodes`.
`NotificationOptionType`	Defines the different ways in which a client may wish to be notified by the registry of an event within a `Subscription`.
`ObjectType`	Defines the different types of `RegistryObjects` a registry may support.
`PhoneType`	Defines the types of telephone numbers.
`QueryLanguage`	Defines the query languages supported by a registry.

Table 8–4 Canonical Classification Schemes (Continued)

Classification Scheme	Description
`ResponseStatusType`	Defines the different types of status for a `RegistryResponse`.
`StatusType`	Defines the different types of status for a `RegistryObject`.
`SubjectGroup`	Defines the groups that a `User` may belong to for access control purposes.
`SubjectRole`	Defines the roles that may be assigned to a `User` for access control purposes.

For a sample program that displays all the canonical classification schemes and their concepts, see `<`*`INSTALL`*`>/registry/samples/classification-schemes/src/JAXRGetCanonicalSchemes.java`. To run this example, follow these steps:

1. Go to the directory `<`*`INSTALL`*`>/registry/samples/classification-schemes`.
2. Type the following command:

```
ant get-schemes
```

Finding Objects by Classification: Examples

For examples of finding objects by classification, see the two examples in `<`*`INSTALL`*`>/registry/samples/query-classification/src`: `JAXRQueryByClassification.java` and `JAXRQueryByCountryClassification.java`. The first example searches for objects that use the canonical classification scheme `InvocationModel`, while the other example searches for organizations that use a geographical classification. To run the examples, follow these steps:

1. Go to the directory `<`*`INSTALL`*`>/registry/samples/query-classification`.
2. Type either of the following commands:

```
ant query-class
ant query-geo
```

These examples are likely to produce results only after you have published an object that uses the specified classification (for example, the one in Adding Classifications: Example, page 48, causes the `query-geo` target to return an object).

Finding Objects by External Identifier

Finding objects by external identifier is similar to finding objects by classification. You first find the classification scheme, then create an `ExternalIdentifier` object to be used as an argument to the `BusinessQueryManagerImpl.findObjects` method or another finder method.

The following code fragment finds all registry objects that contain the Sun Microsystems stock ticker symbol as an external identifier. The sample database included with the Registry does not have any external classification schemes, so you would have to create one named `NASDAQ` for this example to work. See Adding External Identifiers to Objects (page 49) for details on how to do this.

The collection of external identifiers is supplied as the next-to-last argument of the `findObjects` method.

```
ClassificationScheme cScheme = null;
cScheme =
  bqm.findClassificationSchemeByName(null, "NASDAQ");

ExternalIdentifier extId =
  blcm.createExternalIdentifier(cScheme, "%Sun%",
     "SUNW");
Collection extIds = new ArrayList();
extIds.add(extId);
// perform query
BulkResponse response = bqm.findObjects("RegistryObject",
  null, null, null, null, extIds, null);
Collection objects = response.getCollection();
```

Finding Objects by External Identifier: Example

For an example of finding objects by external identifier, see the example `<INSTALL>/registry/samples/query-external-identifier/src/JAXRQueryByExternalIdentifier.java`, which searches for objects that use the `NASDAQ` classification scheme. To run the example, follow these steps:

1. Go to the directory `<INSTALL>/registry/samples/query-external-identifier`.
2. Type the following command:

   ```
   ant run
   ```

This example is not likely to produce results unless you first run the `publish-object` example described in Adding Classifications: Example (page 48).

Finding Objects by External Link

Finding objects by external link does not require the use of a classification scheme, but it does require you to specify a valid URI. The arguments to the `createExternalLink` method are a URI and a description.

If the link you specify is outside your firewall, you also need to specify the system properties `http.proxyHost` and `http.proxyPort` when you run the program so that JAXR can determine the validity of the URI.

The following code fragment finds all organizations that have a specified `ExternalLink` object.

```
ExternalLink extLink =
  blcm.createExternalLink("http://java.sun.com/",
    "Sun Java site");

Collection extLinks = new ArrayList();
extLinks.add(extLink);
BulkResponse response = bqm.findObjects("Organization",
  null, null, null, null, null, extLinks);
Collection objects = response.getCollection();
```

Finding Objects by External Link: Example

For an example of finding objects by external link, see the example `<`*`INSTALL`*`>/registry/samples/query-external-link/src/JAXRQueryByExternalLink.java`, which searches for objects that have a specified external link. The `http.proxyHost` and `http.proxyPort` properties are specified in the `run` target in the `build.xml` file.

To run the example, follow these steps:

1. Go to the directory `<`*`INSTALL`*`>/registry/samples/query-external-link`.
2. Type the following command:

   ```
   ant run
   ```

This example is not likely to produce results unless you first run the publish-object example described in Adding Classifications: Example (page 48).

Finding Objects by Unique Identifier

Every object in the Registry has two identifiers, a unique identifier (also called a `Key`) and a logical identifier. Often the unique and logical identifiers are the same. However, when an object exists in more than one version, the unique identifiers are different for each version, but the logical identifier remains the same. (See Retrieving the Version of an Object, page 34.)

If you know the value of the unique identifier for an object, you can retrieve the object by calling the `QueryManager.getRegistryObject` method with the `String` value as an argument. For example, if `bqm` is your `BusinessQueryManager` instance and `idString` is the `String` value, the following line of code retrieves the object:

```
RegistryObject obj = bqm.getRegistryObject(idString);
```

Once you have the object, you can obtain its type, name, description, and other attributes.

Finding Objects by Unique Identifier: Example

For an example of finding objects by unique identifier, see the example `<`*`INSTALL`*`>/registry/samples/query-id/src/JAXRQueryById.java`, which searches for objects that have a specified unique identifier. To run the example, follow these steps:

1. Go to the directory `<`*`INSTALL`*`>/registry/samples/query-id`.
2. Type the following command:

   `ant run -Did=`*`urn_value`*

For example, if you specify the following ID, you retrieve information on the `ObjectType` classification scheme.

```
urn:oasis:names:tc:ebxml-regrep:classificationScheme:ObjectType
```

Finding Objects You Published

You can retrieve all objects that you published to the Registry, or you can narrow this search to retrieve only the objects you published that are of a particular object type. To retrieve all the objects you have published, use the no-argument version of the `QueryManager.getRegistryObjects` method. The name of this method is misleading, because it returns only objects you have published, not all registry objects.

For example, if `bqm` is your `BusinessQueryManager` instance, use the following line of code:

```
BulkResponse response = bqm.getRegistryObjects();
```

To retrieve all the objects of a particular type that you published, use `QueryManager.getRegistryObjects` with a `String` argument:

```
BulkResponse response = bqm.getRegistryObjects("Service");
```

This method is case-sensitive, so the object type must be capitalized.

The sample programs `JAXRGetMyObjects` and `JAXRGetMyObjectsByType` show how to use these methods.

Finding Objects You Published: Examples

For examples of finding objects by classification, see the two examples in `<`*`INSTALL`*`>/registry/samples/get-objects/src`: `JAXRGetMyObjects.java` and `JAXRGetMyObjectsByType.java`. The first example, `JAXRGetMyObjects.java`, retrieves all objects you have published. The second example, `JAXRGetMyObjectsByType.java`, retrieves all the objects you have published of a specified type. To run the examples, follow these steps:

1. Go to the directory `<`*`INSTALL`*`>/registry/samples/get-objects`.
2. Type the following command to retrieve all objects you published:
   ```
   ant get-obj
   ```
3. Type the following command to retrieve all objects you have published of a specified type, where *`type_name`* is case-sensitive:
   ```
   ant get-obj-type -Dtype=type_name
   ```

Note: At this release, every user has the identity `RegistryOperator`. Because this user owns all the objects in the Registry, the `get-obj` target takes a long time to run, and the `get-obj-type` target may take a long time if the *`type_name`* is one with many objects (`ClassificationNode`, for example).

Retrieving Information About an Object

Once you have retrieved the object or objects you are searching for, you can also retrieve the object's attributes and other objects that belong to it: its name, description, type, ID values, classifications, external identifiers, external links, and slots. For an organization, you can also retrieve the primary contact (a `User` object), postal address, telephone numbers, and the services that the organization offers. For a user, you can retrieve the name, email addresses, postal address, and telephone numbers. For a service, you can retrieve the service bindings.

For an object, you can also retrieve the audit trail, which contains the events that have changed the object's state, and the version. You can also retrieve an object's version number, which is updated whenever a change is made to one of the object's attributes.

This section covers the following topics:

- Retrieving an organization hierarchy
- Retrieving the Name or Description of an Object
- Retrieving the Type of an Object
- Retrieving the ID Values for an Object
- Retrieving the Classifications for an Object
- Retrieving the External Identifiers for an Object
- Retrieving the External Links for an Object
- Retrieving the Slots for an Object
- Retrieving the Attributes of an Organization or User
- Retrieving the Services and Service Bindings for an Organization
- Retrieving an Organization Hierarchy
- Retrieving the Audit Trail of an Object
- Retrieving the Version of an Object

Retrieving the Name or Description of an Object

The name and description of an object are both `InternationalString` objects. An `InternationalString` object contains a set of `LocalizedString` objects. The methods `RegistryObject.getName()` and `RegistryObject.getDescrip-`

tion() return the LocalizedString object for the default locale. You can then retrieve the String value of the LocalizedString object. For example:

```
String name = ro.getName().getValue();
String description = ro.getDescription().getValue();
```

Call the getName or getDescription method with a Locale argument to retrieve the value for a particular locale.

Many of the examples contain private utility methods that retrieve the name, description, and unique identifier for an object. See, for example, JAXRGetMyObjects.java in Finding Objects You Published: Examples (page 23).

Retrieving the Type of an Object

If you have queried the Registry without specifying a particular object type, you can retrieve the type of the objects returned by the query. Use the RegistryObject.getObjectType method, which returns a Concept value. You can then use the Concept.getValue method to obtain the String value of the object type. For example:

```
Concept objType = object.getObjectType();
System.out.println("Object type is " + objType.getValue());
```

The concept will be one of those in the canonical classification scheme ObjectType. For an example of this code, see JAXRQueryByName.java in Finding Objects by Name: Example (page 16).

Retrieving the ID Values for an Object

The unique identifier for an object is contained in a Key object. A Key is a structure that contains the identifier in the form of an id attribute that is a String value. To retrieve the identifier, call the method RegistryObject.getKey().getId().

The JAXR provider also has an implementation-specific method for retrieving the logical identifier, called a lid. The lid is a String attribute of a RegistryObject. To retrieve the lid, call RegistryObjectImpl.getLid(). The method has the following signature:

```
public java.lang.String getLid()
  throws JAXRException
```

For an example of the use of this method, see `JAXRQueryOrg.java` in Retrieving Organization Attributes: Example (page 30).

Retrieving the Classifications for an Object

Use the `RegistryObject.getClassifications` method to retrieve a `Collection` of the object's classifications. For each classification, you can retrieve its name, value, and the classification scheme to which it belongs. The following code fragment retrieves and displays an object's classifications.

```
Collection classifications = object.getClassifications();
Iterator classIter = classifications.iterator();
while (classIter.hasNext()) {
  Classification classification =
    (Classification) classIter.next();
  String name = classification.getName().getValue();
  System.out.println("  Classification name is " + name);
  System.out.println("  Classification value is " +
    classification.getValue());
  ClassificationScheme scheme =
    classification.getClassificationScheme();
  System.out.println("  Classification scheme for " +
    name + " is " + scheme.getName().getValue());
}
```

Some of the examples have a `showClassifications` method that uses this code. See, for example, `JAXRQueryByName.java` in Finding Objects by Name: Example (page 16).

Retrieving the External Identifiers for an Object

Use the `RegistryObject.getExternalIdentifiers` method to retrieve a `Collection` of the object's external identifiers. For each identifier, you can retrieve its name, value, and the classification scheme to which it belongs. For an external identifier, the method that retrieves the classification scheme is `getIdentificationScheme`. The following code fragment retrieves and displays an object's external identifiers.

```
Collection exIds = object.getExternalIdentifiers();
Iterator exIdIter = exIds.iterator();
while (exIdIter.hasNext()) {
  ExternalIdentifier exId =
        (ExternalIdentifier) exIdIter.next();
    String name = exId.getName().getValue();
    System.out.println("  External identifier name is " +
        name);
    String exIdValue = exId.getValue();
    System.out.println("  External identifier value is " +
        exIdValue);
    ClassificationScheme scheme =
        exId.getIdentificationScheme();
    System.out.println("  External identifier " +
        "classification scheme is " +
        scheme.getName().getValue());
}
```

Some of the examples have a `showExternalIdentifiers` method that uses this code. See, for example, `JAXRQueryByName.java` in Finding Objects by Name: Example (page 16).

Retrieving the External Links for an Object

Use the `RegistryObject.getExternalLinks` method to retrieve a `Collection` of the object's external links. For each external link, you can retrieve its name, description, and value. For an external link, the name is optional. The following code fragment retrieves and displays an object's external links.

```
Collection exLinks = obj.getExternalLinks();
Iterator exLinkIter = exLinks.iterator();
while (exLinkIter.hasNext()) {
    ExternalLink exLink = (ExternalLink) exLinkIter.next();
    String name = exLink.getName().getValue();
    if (name != null) {
        System.out.println("  External link name is " + name);
    }
    String description = exLink.getDescription().getValue();
    System.out.println("  External link description is " +
        description);
    String externalURI = exLink.getExternalURI();
    System.out.println("  External link URI is " +
        externalURI);
}
```

Some of the examples have a `showExternalLinks` method that uses this code. See, for example, `JAXRQueryByName.java` in Finding Objects by Name: Example (page 16).

Retrieving the Slots for an Object

Slots are arbitrary attributes that you can create for an object. Use the `RegistryObject.getSlots` method to retrieve a `Collection` of the object's slots. For each slot, you can retrieve its name, values, and type. The name of a `Slot` object is a `String`, not an `InternationalString`, and a slot has a `Collection` of values. The following fragment retrieves and displays an object's slots:

```
Collection slots = object.getSlots();
Iterator slotIter = slots.iterator();
while (slotIter.hasNext()) {
  Slot slot = (Slot) slotIter.next();
  String name = slot.getName();
  System.out.println("  Slot name is " + name);
  Collection values = slot.getValues();
  Iterator valIter = values.iterator();
  int count = 1;
  while (valIter.hasNext()) {
    String value = (String) valIter.next();
    System.out.println("  Slot value " + count++ +
      ": " + value);
  }
  String type = slot.getSlotType();
  if (type != null) {
    System.out.println("  Slot type is " + type);
}
```

Some of the examples have a `showSlots` method that uses this code. See, for example, `JAXRQueryByName.java` in Finding Objects by Name: Example (page 16).

Retrieving the Attributes of an Organization or User

Every `Organization` object can have one postal address and multiple telephone numbers in addition to the attributes available to all other objects. Every organization also has a `User` object as a primary contact, and it may have additional affiliated `User` objects.

The attributes for a `User` object include a `PersonName` object, which has a different format from the name of an object. A user can have multiple postal addresses as well as multiple telephone numbers. A user can also have multiple email addresses.

To retrieve the postal address for an organization, call the `Organization.getPostalAddress` method as follows (`org` is the organization):

```
PostalAddress pAd = org.getPostalAddress();
```

Once you have the address, you can retrieve the address attributes as follows:

```
System.out.println(" Postal Address:\n  " +
  pAd.getStreetNumber() + " " + pAd.getStreet() +
  "\n  " + pAd.getCity() + ", " +
  pAd.getStateOrProvince() + " " +
  pAd.getPostalCode() + "\n  " + pAd.getCountry() +
  "(" + pAd.getType() + ")");
```

To retrieve the primary contact for an organization, call the `Organization.getPrimaryContact` method as follows (`org` is the organization):

```
User pc = org.getPrimaryContact();
```

To retrieve the postal addresses for a user, call the `User.getPostalAddresses` method and extract the `Collection` values as follows (`pc` is the primary contact):

```
Collection pcpAddrs = pc.getPostalAddresses();
Iterator pcaddIter = pcpAddrs.iterator();
while (pcaddIter.hasNext()) {
  PostalAddress pAd = (PostalAddress) pcaddIter.next();
  /* retrieve attributes */
}
```

To retrieve the telephone numbers for either an organization or a user, call the `getTelephoneNumbers` method. In the following code fragment, `org` is the organization. The code retrieves the country code, area code, main number, and type of the telephone number.

```
Collection orgphNums = org.getTelephoneNumbers(null);
Iterator orgphIter = orgphNums.iterator();
while (orgphIter.hasNext()) {
  TelephoneNumber num = (TelephoneNumber) orgphIter.next();
  System.out.println(" Phone number: " +
     "+" + num.getCountryCode() + " " +
     "(" + num.getAreaCode() + ") " +
     num.getNumber() + " (" + num.getType() + ")");
}
```

A `TelephoneNumber` can also have an extension, retrievable through the `getExtension` method. If the number can be dialed electronically, it can have a `url` attribute, retrievable through the `getUrl` method.

To retrieve the name of a user, call the `User.getPersonName` method. A `PersonName` has three attributes that correspond to the given name, middle name(s), and surname of a user. In the following code fragment, `pc` is the primary contact.

```
PersonName pcName = pc.getPersonName();
System.out.println(" Contact name: " +
  pcName.getFirstName() + " " +
  pcName.getMiddleName() + " " +
  pcName.getLastName());
```

To retrieve the email addresses for a user, call the `User.getEmailAddresses` method. An `EmailAddress` has two attributes, the address and its type. In the following code fragment, `pc` is the primary contact.

```
Collection eAddrs = pc.getEmailAddresses();
Iterator eaIter = eAddrs.iterator();
while (eaIter.hasNext()) {
  EmailAddress eAd = (EmailAddress) eaIter.next();
  System.out.println("  Email address: " +
    eAd.getAddress() + " (" + eAd.getType() + ")");
}
```

The attributes for `PostalAddress`, `TelephoneNumber`, `PersonName`, and `EmailAddress` objects are all `String` values. As noted in JAXR Information Model Interfaces (page 11), these objects do not extend the `RegistryObject` interface, so they do not have the attributes of other registry objects.

Retrieving Organization Attributes: Example

For an example of retrieving the attributes of an organization and the `User` that is its primary contact, see the example *<INSTALL>*`/registry/samples/organizations/src/JAXRQueryOrg.java`, which displays information about an organization whose name contains a specified string. To run the example, follow these steps:

1. Go to the directory *<INSTALL>*`/registry/samples/organizations`.
2. Type the following command:

   `ant query-org -Dorg=`*`string`*

Retrieving the Services and Service Bindings for an Organization

Most organizations offer services. JAXR has methods that retrieve the services and service bindings for an organization.

A Service object has all the attributes of other registry objects. In addition, it normally has *service bindings*, which provide information about how to access the service. A ServiceBinding object, along with its other attributes, normally has an access URI and a specification link, which provides the linkage between a service binding and a technical specification that describes how to use the service through the service binding.

In addition to these attributes, you can use the Service.getProvidingOrganization method to retrieve the organization that provides a service, and you can use the ServiceBinding.getService method to retrieve the service for a service binding. The following code fragment retrieves the services for the organization org. Then it retrieves the service bindings for each service and, for each service binding, its access URI and specification links. A specification link in turn has a specification object (typically an ExtrinsicObject), usage description (an InternationalString object), and a Collection of usage parameters that are String values.

```
Collection services = org.getServices();
Iterator svcIter = services.iterator();
while (svcIter.hasNext()) {
  Service svc = (Service) svcIter.next();
  System.out.println(" Service name: " + getName(svc));
  System.out.println(" Service description: " +
     getDescription(svc));

  Collection serviceBindings = svc.getServiceBindings();
  Iterator sbIter = serviceBindings.iterator();
  while (sbIter.hasNext()) {
     ServiceBinding sb = (ServiceBinding) sbIter.next();
     System.out.println("  Binding name: " +
        getName(sb));
     System.out.println("  Binding description: " +
        getDescription(sb));
     System.out.println("  Access URI: " +
        sb.getAccessURI());

     Collection specLinks = sb.getSpecificationLinks();
     Iterator slIter = specLinks.iterator();
     while (slIter.hasNext()) {
        SpecificationLink sl =
          (SpecificationLink) slIter.next();
        RegistryObject ro = sl.getSpecificationObject();
        System.out.println("Specification link " +
          "object of type " + ro.getObjectType());
        System.out.println("Usage description: " +
          sl.getUsageDescription().getValue());
        Collection ups = sl.getUsageParameters();
        Iterator upIter = ups.iterator();
        while (upIter.hasNext()) {
          String up = (String) upIter.next();
          System.out.println("Usage parameter: " +
            up);
        }
     }
  }
}
```

The example Retrieving Organization Attributes: Example (page 30) also displays the services and service bindings for the organizations it finds.

Retrieving an Organization Hierarchy

JAXR allows you to group organizations into families. One organization can have other organizations as its children, and these can in turn have children. Therefore, any given organization may have a parent, children, and descendants.

The `Organization.getParentOrganization` method retrieves an organization's parent. In the following fragment, `chorg` is a child organization.

```
Organization porg = chorg.getParentOrganization();
```

The `Organization.getChildOrganizations` method retrieves a Collection of the organization's children. In the following fragment, `org` is a parent organization.

```
Collection children = org.getChildOrganizations();
```

The `Organization.getDescendantOrganizations` method retrieves multiple generations of descendants, while the `Organization.getRootOrganization` method retrieves the parentless ancestor of any descendant.

For an example of retrieving an organization hierarchy, see Creating and Retrieving an Organization Hierarchy: Example (page 53).

Retrieving the Audit Trail of an Object

Whenever an object is published to the Registry, and whenever it is modified in any way, JAXR creates another object, called an `AuditableEvent`, and adds the event to the audit trail for the published object. The audit trail contains a list of all the events for that object. To retrieve the audit trail, call `RegistryObject.getAuditTrail`. You can also retrieve the individual events in the audit trail and find out their event types. JAXR supports the event types listed in Table 8–5.

Table 8–5 `AuditableEvent` Types

Event Type	**Description**
`EVENT_TYPE_CREATED`	Object was created and published to the registry.
`EVENT_TYPE_DELETED`	Object was deleted using one of the `LifeCycleManager` or `BusinessLifeCycleManager` deletion methods.
`EVENT_TYPE_DEPRECATED`	Object was deprecated using the `LifeCycleManager.deprecateObjects` method.
`EVENT_TYPE_UNDEPRECATED`	Object was undeprecated using the `LifeCycleManager.unDeprecateObjects` method.
`EVENT_TYPE_VERSIONED`	A new version of the object was created. This event typically happens when any of the object’s attributes changes.
`EVENT_TYPE_UPDATED`	Object was updated.
`EVENT_TYPE_APPROVED`	Object was approved using the `LifeCycleManagerImpl.approveObjects` method (implementation-specific).
`EVENT_TYPE_DOWNLOADED`	Object was downloaded (implementation-specific).
`EVENT_TYPE_RELOCATED`	Object was relocated (implementation-specific).

The following code fragment retrieves the audit trail for a registry object, displaying the type and timestamp of each event:

```
Collection events = obj.getAuditTrail();
String objName = obj.getName().getValue();
Iterator eventIter = events.iterator();
while (eventIter.hasNext()) {
  AuditableEventImpl ae = (AuditableEventImpl) eventIter.next();
  int eType = ae.getEventType();
  if (eType == AuditableEvent.EVENT_TYPE_CREATED) {
    System.out.print(objName + " created ");
  } else if (eType == AuditableEvent.EVENT_TYPE_DELETED) {
    System.out.print(objName + " deleted ");
  } else if (eType == AuditableEvent.EVENT_TYPE_DEPRECATED) {
    System.out.print(objName + " deprecated ");
  } else if (eType == AuditableEvent.EVENT_TYPE_UNDEPRECATED) {
    System.out.print(objName + " undeprecated ");
  } else if (eType == AuditableEvent.EVENT_TYPE_UPDATED) {
    System.out.print(objName + " updated ");
  } else if (eType == AuditableEvent.EVENT_TYPE_VERSIONED) {
    System.out.print(objName + " versioned ");
  } else if (eType == AuditableEventImpl.EVENT_TYPE_APPROVED) {
    System.out.print(objName + " approved ");
  } else if (eType == AuditableEventImpl.EVENT_TYPE_DOWNLOADED) {
    System.out.print(objName + " downloaded ");
  } else if (eType == AuditableEventImpl.EVENT_TYPE_RELOCATED) {
    System.out.print(objName + " relocated ");
  } else {
    System.out.print("Unknown event for " + objName + " ");
  }
System.out.println(ae.getTimestamp().toString());
}
```

Some of the examples have a `showAuditTrail` method that uses this code. See, for example, `JAXRQueryByName.java` in Finding Objects by Name: Example (page 16).

See Changing the State of Objects in the Registry (page 65) for information on how to change the state of registry objects.

Retrieving the Version of an Object

If you modify the attributes of a registry object, the Registry creates a new version of the object. For details on how this happens, see Changing the State of

Objects in the Registry (page 65). When you first create an object, it has a version of 1.1.

Note: At this release, versioning of objects is disabled by default. All objects have a version of 1.1 even after modification. For details on how to turn versioning on, see the Release Notes.

To retrieve the version of an object, use the implementation-specific `getVersionInfo` method for a registry object, which returns a `VersionInfoType` object. The method has the following signature:

```
public VersionInfoType getVersionInfo()
  throws JAXRException
```

For example, to retrieve the version number for the organization `org`, cast `org` to a `RegistryObjectImpl` when you call the method. Then call the `VersionInfoType.getVersionName` method, which returns a `String`.

```
import org.oasis.ebxml.registry.bindings.rim.VersionInfoType;
...
VersionInfoType vInfo =
  ((RegistryObjectImpl)org).getVersionInfo();
if (vInfo != null) {
  System.out.println("Org version: " +
     vInfo.getVersionName());
}
```

Some of the examples use code similar to this. See, for example, `JAXRQueryByName.java` in Finding Objects by Name: Example (page 16).

Using Declarative Queries

Instead of the `BusinessQueryManager` interface, you can use the `DeclarativeQueryManager` interface to create and execute queries to the Registry. If you are familiar with SQL, you may prefer to use declarative queries. The `DeclarativeQueryManager` interface depends on another interface, `Query`.

The `DeclarativeQueryManager` interface has two methods, `createQuery` and `executeQuery`. The `createQuery` method takes two arguments, a query type and a string containing the query. The following code fragment creates an SQL query that asks for a list of all `Service` objects in the Registry. Here, `rs` is a `RegistryService` object.

```
DeclarativeQueryManager qm = rs.getDeclarativeQueryManager();
String qString = "select s.* from Service s";
Query query = qm.createQuery(Query.QUERY_TYPE_SQL, qString);
```

After you create the query, you execute it as follows:

```
BulkResponse response = qm.executeQuery(query);
Collection objects = response.getCollection();
```

You then extract the objects from the response just as you do with ordinary queries.

Using Declarative Queries: Example

For an example of the use of declarative queries, see `<INSTALL>/registry/samples/query-declarative/src/JAXRQueryDeclarative.java`, which creates and executes a SQL query.

The SQL query string, which is defined in the `JAXRExamples.properties` file, looks like this (all on one line):

```
declarative.query=SELECT ro.* from RegistryObject ro, Name nm,
Description d WHERE upper(nm.value) LIKE upper('%free%') AND
upper(d.value) LIKE upper('%free%') AND (ro.id = nm.parent AND
ro.id = d.parent)
```

This query finds all objects that have the string `"free"` in both the name and the description attributes.

To run the example, follow these steps:

1. Go to the directory `<INSTALL>/registry/samples/query-declarative`.
2. Type the following command:
   `ant run`

Using Iterative Queries

If you expect a declarative query to return a very large result set, you can use the implementation-specific iterative query feature. The `DeclarativeQueryMan-`

agerImpl.executeQuery method can take an argument that specifies a set of parameters. This method has the following signature:

```
public BulkResponse executeQuery(Query query,
    java.util.Map queryParams,
    IterativeQueryParams iterativeParams)
  throws JAXRException
```

You can specify parameters that cause each query to request a different subset of results within the result set. Instead of making one query return the entire result set, you can make each individual query return a manageable set of results.

Suppose you have a query string that you expect to return up to 100 results. You can create a set of parameters that causes the query to return 10 results at a time. First, you create an instance of the class IterativeQueryParams, which is defined in the package org.freebxml.omar.common. The two fields of the class are startIndex, the starting index of the array, and maxResults, the maximum number of results to return. You specify the initial values for these fields in the constructor.

```
int maxResults = 10;
int startIndex = 0;
IterativeQueryParams iterativeQueryParams =
  new IterativeQueryParams(startIndex, maxResults);
```

Execute the queries within a for loop that terminates with the highest number of expected results and advances by the maxResults value for the individual queries. Increment the startIndex field at each loop iteration.

```
for (int i = 0; i < 100; i += maxResults) {
  // Execute query with iterative query params
  Query query = dqm.createQuery(Query.QUERY_TYPE_SQL,
    queryStr);
  iterativeQueryParams.startIndex = i;
  BulkResponse br = dqm.executeQuery(query, null,
    iterativeQueryParams);
  Collection objects = br.getCollection();
  // retrieve individual objects ...
}
```

The Registry is not required to maintain transactional consistency or state between iterations of a query. Thus it is possible for new objects to be added or existing objects to be removed from the complete result set between iterations.

Therefore, you may notice that a result set element is skipped or duplicated between iterations.

Using Iterative Queries: Example

For an example of the use of an iterative query, see `<INSTALL>/registry/samples/query-iterative/src/JAXRQueryIterative.java`. This program finds all registry objects whose names match a given string and then iterates through the first 100 of them. To run the example, follow these steps:

1. Go to the directory `<INSTALL>/registry/samples/query-iterative`.
2. Type the following command, specifying a string value:
   `ant run -Dname=string`

Invoking Stored Queries

The implementation-specific `AdhocQueryImpl` class, which extends `RegistryObjectImpl`, allows you to invoke queries that are stored in the Registry. The Registry has several default `AdhocQueryImpl` objects that you can invoke. The most useful are named `FindAllMyObjects` and `GetCallersUser`:

- `FindAllMyObjects` is equivalent to the `QueryManager.getRegistryObjects()` method, described in Finding Objects You Published (page 22).
- `GetCallersUser` is equivalent to the question “Who am I?” It returns the `User` object associated with the client that executed the query. If the caller is not logged in to the Registry, this query returns the user named “Registry Guest.”

To invoke a stored query, begin by using the `BusinessQueryManagerImpl.findObjects` method to locate the query. The following code searches for the `GetCallersUser` query.

```
Collection namePatterns = new ArrayList();
namePatterns.add("GetCallersUser");

// Find objects with name GetCallersUser
BulkResponse response =
  bqm.findObjects("AdhocQuery", null, namePatterns,
    null, null, null, null);
Collection queries = response.getCollection();
```

Then find the query string associated with the AdhocQuery and use it to create and execute a query, this time using DeclarativeQueryManager methods.

```
// get the first (only) query and invoke it
Iterator qIter = queries.iterator();
if (!(qIter.hasNext())) {
  System.out.println("No objects found");
} else {
  AdhocQueryImpl aq = (AdhocQueryImpl) qIter.next();
  String qString = aq.toString();
  Query query = dqm.createQuery(qType, qString);

  BulkResponse br = dqm.executeQuery(query);
  Collection objects = br.getCollection();
  ...
```

Invoking Stored Queries: Example

For an example of the use of a stored query, see <*INSTALL*>/registry/samples/query-stored/src/JAXRQueryStored.java. This example returns the user's registry login name. To run the example, follow these steps:

1. Go to the directory <*INSTALL*>/registry/samples/query-stored.
2. Type the following command:
   ```
   ant run
   ```

Querying a Registry Federation

If the registry you are querying is part of one or more registry federations (see About Registries and Repositories, page 1), you can perform declarative queries on all registries in all federations of which your registry is a member, or on all the registries in one federation.

To perform a query on all registries in all federations of which your registry is a member, you call the implementation-specific setFederated method on a QueryImpl object. The method has the following signature:

```
public void setFederated(boolean federated)
  throws JAXRException
```

You call the method as follows:

```
QueryImpl query = (QueryImpl)
  dqm.createQuery(Query.QUERY_TYPE_SQL, qString);
query.setFederated(true);
```

If you know that your registry is a member of only one federation, this method is the only one you need to call before you execute the query.

To limit your query to the registries in one federation, you need to call an additional implementation-specific method, `setFederation`. This method takes as its argument the unique identifier of the federation you want to query:

```
public void setFederation(java.lang.String federationId)
  throws JAXRException
```

Therefore, before you can call this method, you must obtain the unique identifier value. To do so, first call `BusinessQueryManagerImpl.findObjects` to locate the federation by name. In this code, you would substitute the actual name of the federation for the string `"NameOfFederation"`.

```
Collection namePatterns = new ArrayList();
namePatterns.add("NameOfFederation");

// Find objects with name NameOfFederation
BulkResponse response =
  bqm.findObjects("Federation", null, namePatterns,
     null, null, null, null);
```

Then, iterate through the collection (which should have only one member) and retrieve the key value:

```
String fedId = federation.getKey().getId();
```

Finally, create the query, call `setFederated` and `setFederation`, and execute the query:

```
QueryImpl query = (QueryImpl)
  dqm.createQuery(Query.QUERY_TYPE_SQL, qString);
query.setFederated(true);
query.setFederation(fedId);
response = dqm.executeQuery(query);
```

Using Federated Queries: Example

For an example of the use of a federated query, see `<INSTALL>/registry/samples/query-federation/src/JAXRQueryFederation.java`. This example performs two queries, a declarative query and a stored query, on every federation it finds (the database provided with the Registry contains only one).

The declarative query is the same query performed in Using Declarative Queries: Example (page 36). The stored query is the `GetCallersUser` query, as in the previous example.

To run the example, follow these steps:

1. Go to the directory `<INSTALL>/registry/samples/query-federation`.
2. Type the following command:

   ```
   ant run
   ```

Publishing Objects to the Registry

If a client has authorization to do so, it can submit objects to the Service Registry, modify them, and remove them. A client uses the `BusinessLifeCycleManager` interface to perform these tasks.

Registries usually allow a client to modify or remove objects only if the objects are being modified or removed by the same user who first submitted them. Access policies can control who is authorized to publish objects and perform actions on them.

Publishing registry objects involves the following tasks:

- Creating Objects
- Saving Objects in the Registry

It is important to remember that submitting objects is a multi-step task: you create the objects and populate them by setting their attributes, then you save them. The objects appear in the registry only after you save them.

You may remember that when you search for objects by classification, external identifier, and the like, you create the classification or other object that you are using in the search. (For an example, see Finding Objects by Classification, page 17.) However, you do not save this object. You create the object only for the purposes of the search, after which it disappears. You do not need authorization from the Registry to create an object, but you must have authorization to save it.

Note: At this release, you do not need authorization from the Registry to save objects.

Creating Objects

A client creates an object and populates it with data before publishing it. You can create and publish any of the following types of `RegistryObject`:

- `AdhocQuery`
- `Association`
- `ClassificationScheme`
- `Concept`
- `ExternalLink`
- `ExtrinsicObject`
- `Federation`
- `Organization`
- `RegistryPackage`
- `Service`
- `Subscription`
- `User`

The following types of `RegistryObject` cannot be published separately, but you can create and save them as part of another object:

- `Classification` (any `RegistryObject`)
- `ExternalIdentifier` (any `RegistryObject`)
- `ServiceBinding` (`Service`)
- `Slot` (any `RegistryObject`)
- `SpecificationLink` (`ServiceBinding`)

Some objects fall into special categories:

- An `AuditableEvent` is published by the Registry when an object has a change in state.
- A `Notification` is published by the Registry when an `AuditableEvent` that matches a `Subscription` occurs.
- A `Registry` can be published only by a user with the role `Registry-Administrator`.

The subsections that follow describe first the tasks common to creating and saving all registry objects. They then describe some tasks specific to particular object types.

- Adding names and descriptions to objects
- Identifying objects
- Adding classifications to objects
- Adding external identifiers to objects
- Adding external links to objects
- Adding slots to objects
- Creating organizations
- Creating users
- Creating services and service bindings

- Using Create Methods for Objects
- Adding Names and Descriptions to Objects
- Identifying Objects
- Creating and Using Classification Schemes and Concepts
- Adding Classifications to Objects
- Adding External Identifiers to Objects
- Adding External Links to Objects
- Adding Slots to Objects
- Creating Organizations
- Creating Users
- Creating Services and Service Bindings

Using Create Methods for Objects

The `LifeCycleManager` interface supports create methods for all types of `RegistryObject` (except `AuditableEvent` and `Notification`, which can be created only by the Registry itself).

In addition, you can use the `LifeCycleManager.createObject` factory method to create an object of a particular type. This method takes a String argument consisting of one of the static fields supported by the `LifeCycleManager` interface. In the following code fragment, `blcm` is the `BusinessLifeCycleManager` object:

```
Organization org = (Organization)
  blcm.createObject(blcm.ORGANIZATION);
```

The object-specific create methods usually take one or more parameters that set some of the attributes of the object. For example, the `createOrganization` method sets the name of the organization:

```
Organization org = blcm.createOrganization("MyOrgName");
```

On the other hand, the `createExtrinsicObject` method takes a `DataHandler` argument that sets the repository item for the extrinsic object.

Adding Names and Descriptions to Objects

For all objects, you can set the `name` and `description` attributes by calling setter methods. These attributes are of type `InternationalString`. An `InternationalString` includes a set of `LocalizedString` objects that allow users to display the name and description in one or more locales. By default, the `InternationalString` value uses the default locale.

For example, the following fragment creates a description for an organization that uses two localized strings, one in the language of the default locale and one in French (Canada).

```
InternationalString is =
  blcm.createInternationalString("What We Do"));
Locale loc = new Locale("fr", "CA");
LocalizedString ls = blcm.createLocalizedString(loc,
  "ce que nous faisons");
is.addLocalizedString(ls);
org.setDescription(is);
```

Identifying Objects

As stated in Finding Objects by Unique Identifier (page 22), every object in the Registry has two identifiers, a unique identifier and a logical identifier. If you do not set these identifiers when you create the object, the Registry generates a unique value and assigns that value to both the unique and the logical identifiers.

Whenever a new version of an object is created (see Retrieving the Version of an Object, page 34, and Changing the State of Objects in the Registry, page 65), the logical identifier remains the same as the original one, but the Registry generates a new unique identifier by adding a colon and the version number to the unique identifier.

Note: At this release, versioning is disabled by default. The logical and unique identifiers remain the same after the object is modified.

If you plan to use your own identification scheme, you can use API methods to set object identifiers. In the JAXR API, the unique identifier is called a `Key` object. You can use the `LifeCycleManager.createKey` method to create a unique identifier from a `String` object, and you can use the `RegistryObject.setKey` method to set it. The logical identifier is called a `lid`, and the JAXR provider for the Registry has an implementation-specific method, `RegistryObjectImpl.setLid`, which also takes a `String` argument, for setting this identifier. The method has the following signature:

```
public void setLid(java.lang.String lid)
  throws JAXRException
```

Any identifier you specify must be a valid, globally unique URN (Uniform Resource Name). When the JAXR API generates a key for an object, the key is in the form of a DCE 128 UUID (Universal Unique IDentifier).

Creating and Using Classification Schemes and Concepts

You can create your own classification schemes and concept hierarchies for classifying registry objects. To do so, follow these steps:

1. Use the `LifeCycleManager.createClassificationScheme` method to create the classification scheme.

2. Use the `LifeCycleManager.createConcept` method to create concepts.
3. Use the `ClassificationScheme.addChildConcept` method to add the concepts to the classification scheme.
4. For a deeper hierarchy, use the `Concept.addChildConcept` method to add child concepts to the concepts.
5. Save the classification scheme.

The `LifeCycleManager.createClassificationScheme` method has several forms. You can specify two arguments, a name and description, as either `String` or `InternationalString` values. For example, to create a classification scheme to describe how books are shelved in a library, you could use the following code fragment:

```
ClassificationScheme cs =
  blcm.createClassificationScheme("LibraryFloors",
    "Scheme for Shelving Books");
```

An alternate form of the `createClassificationScheme` method takes one argument, a `Concept`, and converts it to a `ClassificationScheme`.

The `createConcept` method takes three arguments: a parent, a name, and a value. The parent can be either a `ClassificationScheme` or another `Concept`. It is acceptable to specify a value but no name.

The following code fragment uses a static `String` array containing the names of the floors of the library to create a concept for each floor of the library, and then adds the concept to the classification scheme.

```
for (int i = 0; i < floors.length; i++) {
  Concept con = blcm.createConcept(cs, floors[i], floors[i]);
  cs.addChildConcept(con);
  ...
```

For each concept, you can create more new concepts and call `Concept.addChildConcept` to create another level of the hierarchy. When you save the classification scheme, the entire concept hierarchy is also saved.

Creating Classification Schemes: Example

For an example of creating a classification scheme, see `<INSTALL>/registry/samples/classification-schemes/src/JAXRPublishScheme.java`. This example creates a classification scheme named `LibraryFloors` and a concept hierarchy that includes each floor of the library and the subject areas that can be found there. To run the example, follow these steps:

1. Go to the directory `<INSTALL>/registry/samples/classification-schemes`.
2. Type the following command:

```
ant pub-scheme
```

To display the concept hierarchy, use the program `<INSTALL>/registry/samples/classification-schemes/src/JAXRQueryScheme.java`. This example displays the concept hierarchy for any classification scheme you specify. To run the example, follow these steps:

1. Go to the directory `<INSTALL>/registry/samples/classification-schemes`.
2. Type the following command:

```
ant query-scheme -Dname=LibraryFloors
```

To delete this classification scheme, use the program `<INSTALL>/registry/samples/classification-schemes/src/JAXRQueryScheme.java`. To run the example, follow these steps:

1. Go to the directory `<INSTALL>/registry/samples/classification-schemes`.
2. Type the following command:

```
ant del-scheme -Dname=LibraryFloors
```

Adding Classifications to Objects

Objects can have one or more classifications based on one or more classification schemes (taxonomies). To establish a classification for an object, the client first locates the taxonomy it wants to use. The client then creates a classification using the classification scheme and a concept (a taxonomy element) within the classification scheme.

For information on creating a new classification scheme with a hierarchy of concepts, see Creating Relationships Between Objects: Associations (page 57). A classification scheme with a concept hierarchy is called an *internal classification scheme*.

To add a classification that uses an existing classification scheme, you usually call the `BusinessQueryManager.findClassificationSchemeByName` method. This method takes two arguments, a `Collection` of `FindQualifier` objects and a `String` that specifies a name pattern. It is an error for this method to return more than one result, so you must define the search very precisely. For example, the following code fragment searches for the classification scheme named `AssociationType`:

```
String schemeName = "AssociationType";
ClassificationScheme cScheme =
  bqm.findClassificationSchemeByName(null, schemeName);
```

After you locate the classification scheme, you call the `LifeCycleManager.createClassification` method, specifying three arguments: the classification scheme and the name and value of the concept.

```
Classification classification =
  blcm.createClassification(cScheme, "Extends", "Extends");
```

An alternative method is to call `BusinessQueryManager.findConcepts` (or `BusinessQueryManagerImpl.findObjects` with a `"Concept"` argument), locate the concept you wish to use, and call another form of `createClassification`, with the concept as the only argument:

```
Classification classification =
  blcm.createClassification(concept);
```

After creating the classification, you call `RegistryObject.addClassification` to add the classification to the object.

```
object.addClassification(classification);
```

To add multiple classifications, you can create a `Collection`, add the classification to the `Collection`, and call `RegistryObject.addClassifications` to add the `Collection` to the object.

Adding Classifications: Example

For an example of adding classifications to an object, see `<INSTALL>/registry/samples/publish-object/src/JAXRPublishObject.java`. This example creates an organization and adds a number of objects to it. To run the example, follow these steps:

1. Go to the directory `<INSTALL>/registry/samples/publish-object`.
2. Type the following command:

   ```
   ant run
   ```

Adding External Identifiers to Objects

To add an external identifier to an object, follow these steps:

1. Find or create the classification scheme to be used.
2. Create an external identifier using the classification scheme.

To create external identifiers, you use an *external classification scheme,* which is a classification scheme without a concept hierarchy. You specify a name and value for the external identifier.

The database supplied with the Registry does not include any external classification schemes, so before you can use one you must create it, using code like the following:

```
ClassificationScheme extScheme =
  blcm.createClassificationScheme("NASDAQ",
     "OTC Stock Exchange");
```

To find an existing classification scheme, you typically call the `BusinessQuery-Manager.findClassificationSchemeByName` method, as described in Adding Classifications to Objects (page 47).

For example, the following code fragment finds the external classification scheme you just created:

```
ClassificationScheme extScheme =
  bqm.findClassificationSchemeByName(null,
     "NASDAQ");
```

To add the external identifier, you call the `LifeCycleManager.createExter-nalIdentifier` method, which takes three arguments: the classification scheme and the name and value of the external identifier. Then you add the external identifier to the object.

```
ExternalIdentifier extId =
  blcm.createExternalIdentifier(extScheme, "Sun",
     "SUNW);
object.addExternalIdentifier(extId);
```

The example `<INSTALL>/registry/samples/publish-object/src/JAXRPub-lishObject.java`, described in Adding Classifications: Example (page 48), also adds an external identifier to an object.

Adding External Links to Objects

To add an external link to an object, you call the `LifeCycleManager.createExternalLink` method, which takes two arguments: the URI of the link, and a description of the link. Then you add the external link to the object.

```
String eiURI = "http://java.sun.com/";
String eiDescription = "Java Technology";
ExternalLink extLink =
  blcm.createExternalLink(eiURI, eiDescription);
object.addExternalLink(extLink);
```

The URI must be a valid URI, and the JAXR provider checks its validity. If the link you specify is outside your firewall, you need to specify the system properties `http.proxyHost` and `http.proxyPort` when you run the program so that JAXR can determine the validity of the URI.

To disable URI validation (for example, if you want to specify a link that is not currently active), call the `ExternalLink.setValidateURI` method before you create the link.

```
extLink.setValidateURI(false);
```

The example `<INSTALL>/registry/samples/publish-object/src/JAXRPublishObject.java`, described in Adding Classifications: Example (page 48), also adds an external link to an object.

Adding Slots to Objects

Slots are arbitrary attributes, so the API provides maximum flexibility for you to create them. You can provide a name, one or more values, and a type. The name and type are `String` objects. The value or values are stored as a `Collection` of `String` objects, but the `LifeCycleManager.createSlot` method has a form that allows you to specify a single `String` value. For example, the following code fragment creates a slot using a `String` value, then adds the slot to the object.

```
String slotName = "Branch";
String slotValue = "Paris";
String slotType = "City";
Slot slot = blcm.createSlot(slotName, slotValue, slotType);
org.addSlot(slot);
```

The example *<INSTALL>*/registry/samples/publish-object/src/JAXRPublishObject.java, described in Adding Classifications: Example (page 48), also adds a slot to an object.

Creating Organizations

An Organization object is probably the most complex registry object. It normally includes the following attributes, in addition to those common to all objects:

- One or more PostalAddress objects.
- One or more TelephoneNumber objects.
- A PrimaryContact object, which is a User object. A User object normally includes a PersonName object and collections of TelephoneNumber, EmailAddress, and PostalAddress objects.
- One or more Service objects and their associated ServiceBinding objects.

An organization can also have one or more child organizations, which can in turn have children, to form a hierarchy of organizations.

The following code fragment creates an organization and specifies its name, description, postal address, and telephone number.

```
// Create organization name and description
Organization org =
  blcm.createOrganization("The ebXML Coffee Break");
InternationalString is =
  blcm.createInternationalString("Purveyor of " +
     "the finest coffees. Established 1905");
org.setDescription(is);

// create postal address for organization
String streetNumber = "99";
String street = "Imaginary Ave. Suite 33";
String city = "Imaginary City";
String state = "NY");
String country = "USA");
String postalCode = "00000";
String type = "Type US";
PostalAddress postAddr =
  blcm.createPostalAddress(streetNumber, street, city, state,
     country, postalCode, type);
org.setPostalAddress(postAddr);
```

```
// create telephone number for organization
TelephoneNumber tNum = blcm.createTelephoneNumber();
tNum.setCountryCode("1");
tNum.setAreaCode("100");
tNum.setNumber("100-1000");
tNum.setType("OfficePhone");
Collection tNums = new ArrayList();
tNums.add(tNum);
org.setTelephoneNumbers(tNums);
```

The telephone number type is the value of a concept in the `PhoneType` classification scheme: `"OfficePhone"`, `"MobilePhone"`, `"HomePhone"`, `"FAX"`, or `"Beeper"`.

To create a hierarchy of organizations, use the `Organization.addChildOrganization` method to add one organization to another, or use the `Organization.addChildOrganizations` method to add a `Collection` of organizations to another.

Creating an Organization: Examples

For examples of creating an organization, see `JAXRPublishOrg.java` and `JAXRPublishOrgNoPC.java` in the directory `<INSTALL>/registry/samples/organizations/src`.

The `JAXRPublishOrg` example creates an organization, its primary contact, and a service and service binding. It displays the unique identifiers for the organization, user, and service so that you can use them later when you delete the objects. This example creates a fictitious `User` as the primary contact for the organization.

The other example, `JAXRPublishOrgNoPC`, does not set a primary contact for the organization. In this case, the primary contact by default is the `User` who is authenticated when you run the program.

To run the examples, follow these steps:

1. Go to the directory `<INSTALL>/registry/samples/organizations`.
2. Type the following commands:

```
ant pub-org
ant pub-org-nopc
```

Creating and Retrieving an Organization Hierarchy: Example

For examples of publishing and retrieving an organization hierarchy, see the examples `<INSTALL>/registry/samples/organizations/src/JAXRPublishOrgFamily.java` and `<INSTALL>/registry/samples/organizations/src/JAXRQueryOrgFamily.java`. To run the examples, follow these steps:

1. Go to the directory `<INSTALL>/registry/samples/organizations`.
2. Type the following command to publish the organizations:
   `ant pub-fam`
3. Type the following command to retrieve the organizations you published:
   `ant query-fam`

Creating Users

If you create an organization without specifying a primary contact, the default primary contact is the `User` object that created the organization (that is, the user whose credentials you set when you created the connection to the Registry). However, you can specify a different user as the primary contact. A `User` is also a complex type of registry object. It normally includes the following attributes, in addition to those common to all objects:

- A `PersonName` object
- One or more `PostalAddress` objects
- One or more `TelephoneNumber` objects
- One or more `EmailAddress` objects
- One or more `URL` objects representing the user's home page

The following code fragment creates a `User` and then sets that `User` as the primary contact for the organization. This `User` has a telephone number and email address but no postal address.

```
// Create primary contact, set name
User primaryContact = blcm.createUser();
String userId = primaryContact.getKey().getId();
System.out.println("User URN is " + userId);
PersonName pName =
  blcm.createPersonName("Jane", "M.", "Doe");
primaryContact.setPersonName(pName);

// Set primary contact phone number
```

```
    TelephoneNumber pctNum = blcm.createTelephoneNumber();
    pctNum.setCountryCode("1");
    pctNum.setAreaCode("100");
    pctNum.setNumber("100-1001");
    pctNum.setType("MobilePhone");
    Collection phoneNums = new ArrayList();
    phoneNums.add(pctNum);
    primaryContact.setTelephoneNumbers(phoneNums);

    // Set primary contact email address
    EmailAddress emailAddress =
    blcm.createEmailAddress("jane.doe@TheCoffeeBreak.com");
    emailAddress.setType("OfficeEmail"));
    Collection emailAddresses = new ArrayList();
    emailAddresses.add(emailAddress);
    primaryContact.setEmailAddresses(emailAddresses);

    URL pcUrl = new URL((bundle.getString("person.url"));
    primaryContact.setUrl(pcUrl);

    // Set primary contact for organization
    org.setPrimaryContact(primaryContact);
```

The telephone number type for the primary contact is the value of a concept in the `PhoneType` classification scheme: `"OfficePhone"`, `"MobilePhone"`, `"HomePhone"`, `"FAX"`, or `"Beeper"`. The email address type for the primary contact is the value of a concept in the `EmailType` classification scheme: either `"OfficeEmail"` or `"HomeEmail"`.

Creating Services and Service Bindings

Most organizations publish themselves to a registry to offer services, so JAXR has facilities to add services and service bindings to an organization.

You can also create services that are not attached to any organization.

Like an `Organization` object, a `Service` object has a name, a description, and a unique key that is generated by the registry when the service is registered. It may also have classifications associated with it.

In addition to the attributes common to all objects, a service also commonly has *service bindings*, which provide information about how to access the service. A `ServiceBinding` object normally has a description, an access URI, and a specification link, which provides the linkage between a service binding and a technical specification that describes how to use the service by using the service binding.

The following code fragment shows how to create a collection of services, add service bindings to a service, and then add the services to the organization. It specifies an access URI but not a specification link. Because the access URI is not real and because JAXR by default checks for the validity of any published URI, the binding sets its `validateURI` attribute to false.

```
// Create services and service
Collection services = new ArrayList();
Service service = blcm.createService("My Service Name");
InternationalString is =
  blcm.createInternationalString("My Service Description");
service.setDescription(is);

// Create service bindings
Collection serviceBindings = new ArrayList();
ServiceBinding binding =
  blcm.createServiceBinding();
is = blcm.createInternationalString("My Service Binding " +
  "Name"));
binding.setName(is);
is = blcm.createInternationalString("My Service Binding " +
  "Description");
binding.setDescription(is);
// allow us to publish a fictitious URI without an error
binding.setValidateURI(false);
binding.setAccessURI("http://TheCoffeeBreak.com:8080/sb/");
...
serviceBindings.add(binding);

// Add service bindings to service
service.addServiceBindings(serviceBindings);

// Add service to services, then add services to organization
services.add(service);
org.addServices(services);
```

A service binding normally has a technical specification that describes how to access the service. An example of such a specification is a WSDL document. To publish the location of a service's specification (if the specification is a WSDL document), you create a `SpecificationLink` object that refers to an `ExtrinsicObject`. For details, see Storing Items in the Repository (page 60).

(This mechanism is different from the way you publish a specification's location to a UDDI registry: for a UDDI registry you create a `Concept` object and then add the URL of the WSDL document to the `Concept` object as an `ExternalLink` object.)

Saving Objects in the Registry

Once you have created an object and set its attributes, you publish it to the Registry by calling the `LifeCycleManager.saveObjects` method or an object-specific save method like `BusinessLifeCycleManager.saveOrganizations` or `BusinessLifeCycleManager.saveServices`. You always publish a collection of objects, not a single object. The save methods return a `BulkResponse` object that contains the keys (that is, the unique identifiers) for the saved objects. The following code fragment saves an organization and retrieves its key:

```
// Add organization and submit to registry
// Retrieve key if successful
Collection orgs = new ArrayList();
orgs.add(org);
BulkResponse response = blcm.saveOrganizations(orgs);
Collection exceptions = response.getExceptions();
if (exceptions == null) {
  System.out.println("Organization saved");

  Collection keys = response.getCollection();
  Iterator keyIter = keys.iterator();
  if (keyIter.hasNext()) {
    javax.xml.registry.infomodel.Key orgKey =
      (javax.xml.registry.infomodel.Key) keyIter.next();
    String id = orgKey.getId();
    System.out.println("Organization key is " + id);
  }
}
```

If one of the objects exists but some of the data have changed, the save methods update and replace the data. This normally results in the creation of a new version of the object (see Changing the State of Objects in the Registry, page 65).

Managing Objects in the Registry

- Once you have published objects to the Registry, you can perform operations on them. This chapter describes these operations.Creating Relationships Between Objects: Associations
- Storing Items in the Repository
- Organizing Objects Within Registry Packages
- Changing the State of Objects in the Registry
- Removing Objects From the Registry and Repository

Creating Relationships Between Objects: Associations

You can create an `Association` object and use it to specify a relationship between any two objects. The ebXML specification specifies an `AssociationType` classification scheme that contains a number of canonical concepts you can use when you create an `Association`. You can also create your own concepts within the `AssociationType` classification scheme, if none of the canonical ones are suitable.

The canonical association types are as follows:

- `AccessControlPolicyFor`
- `AffiliatedWith` (which has the subconcepts `EmployeeOf` and `MemberOf`)
- `Contains`
- `ContentManagementServiceFor`
- `EquivalentTo`
- `Extends`
- `ExternallyLinks`
- `HasFederationMember`
- `HasMember`
- `Implements`
- `InstanceOf`
- `InvocationControlFileFor` (which has the subconcepts `Cataloging-ControlFileFor` and `ValidationControlFileFor`)
- `OffersService`
- `OwnerOf`
- `RelatedTo`
- `Replaces`
- `ResponsibleFor`
- `SubmitterOf`
- `Supersedes`
- `Uses`

The Registry uses some of these association types automatically. For example, when you add a `Service` to an `Organization`, the Registry creates an `OffersService` association with the `Organization` as the source and the `Ser-vice` as the target.

Associations are directional: each `Association` has a source object and a target object. Establishing an association between two objects is a three-step process:

1. Find the `AssociationType` concept you wish to use (or create one).
2. Use the `LifeCycleManager.createAssociation` method to create the association. This method takes two arguments, the target object and the concept that identifies the relationship.
3. Use the `RegistryObject.addAssociation` method to add the association to the source object.

For example, suppose you have two objects, `obj1` and `obj2`, and you want to establish a `RelatedTo` relationship between them. (In this relationship, which object is the source and which is the target is arbitrary.) First, locate the concept named `RelatedTo`:

```
// Find RelatedTo concept for Association
Collection namePatterns = new ArrayList();
namePatterns.add("RelatedTo");
BulkResponse br = bqm.findObjects("Concept", null,
  namePatterns, null, null, null, null);
Collection concepts = br.getCollection();
```

Iterate through the concepts (there should only be one) to find the right one.

```
Concept relConcept = (Concept) concIter.next();
```

Create the association, specifying `obj2` as the target:

```
Association relAssoc =
  blcm.createAssociation(obj2, relConcept);
```

Add the association to the source object, `obj1`:

```
obj1.addAssociation(relAssoc);
```

Finally, save the association:

```
Collection associations = new ArrayList();
associations.add(relAssoc1);
BulkResponse response = blcm.saveObjects(associations);
```

Associations can be of two types, intramural and extramural. You create an *intramural association* when both the source and target object are owned by you. You create an *extramural association* when at least one of these objects is not owned by you. The owner of an object can use an access control policy to restrict the right to create an extramural association with that object as a source or target.

Creating Associations: Example

For an example of creating an association, see `<INSTALL>/registry/samples/publish-association/src/JAXRPublishAssociation.java`. This example creates a `RelatedTo` association between any two objects whose unique identifiers you specify. For example, you could specify the identifiers of the two child organizations created in Creating and Retrieving an Organization Hierarchy: Example (page 53). To run the example, follow these steps:

1. Go to the directory `<INSTALL>/registry/samples/publish-association`.
2. Type the following command:

   `ant run -Did1=string1 -Did2=string2`

Whether the association is intramural or extramural depends upon who owns the two objects.

Storing Items in the Repository

As About Registries and Repositories (page 1) explains, the Registry includes a repository in which you can store electronic content. For every item you store in the repository, you must first create a type of `RegistryObject` called an `ExtrinsicObject`. When you save the `ExtrinsicObject` to the Registry, the associated repository item is also saved.

Creating an Extrinsic Object

To create an `ExtrinsicObject`, you first need to create a `javax.activation.DataHandler` object for the repository item. The `LifeCycleManager.createExtrinsicObject` method takes a `DataHandler` argument.

To store a file in the repository, for example, first create a `java.io.File` object. From the `File` object, create a `javax.activation.FileDataSource` object, which you use to instantiate the `DataHandler` object.

```
String filename = "./MyFile.xml";
File repositoryItemFile = new File(filename);
DataHandler repositoryItem =
  new DataHandler(new FileDataSource(repositoryItemFile));
```

Next, call `createExtrinsicObject` with the `DataHandler` as argument:

```
ExtrinsicObject eo =
  blcm.createExtrinsicObject(repositoryItem);
eo.setName("My File");
```

Set the MIME type of the object to make it accessible. The default MIME type is `application/octet-stream`. If the file is an XML file, set it as follows:

```
eo.setMimeType("text/xml");
```

Finally, call the implementation-specific `ExtrinsicObjectImpl.setObjectType` method to store the `ExtrinsicObject` in an appropriate area of the Registry. This method has the following signature:

```
public void setObjectType(Concept objectType)
  throws JAXRException
```

The easiest way to find the appropriate concept for a file is to use the Explore feature of the Web Console. Look under the `ObjectType` classification scheme for the various types of `ExtrinsicObject` concepts. Specify the ID for the concept as the argument to `getRegistryObject`, then specify the concept as the argument to `setObjectType`.

```
String conceptId =
  "urn:oasis:names:tc:ebxml-
regrep:ObjectType:RegistryObject:ExtrinsicObject:XML";
Concept objectTypeConcept =
  (Concept) bqm.getRegistryObject(conceptId);
((ExtrinsicObjectImpl)eo).setObjectType(objectTypeConcept);
```

Finally, you save the `ExtrinsicObject` to the Registry.

```
Collection extobjs = new ArrayList();
extobjs.add(eo);
BulkResponse response = blcm.saveObjects(extobjs);
```

The `ExtrinsicObject` contains the metadata, and a copy of the file is stored in the repository.

If the Registry does not have a concept for the kind of file you want to store there, you can create and save the concept yourself.

Creating an Extrinsic Object: Example

For an example of creating an extrinsic object, see `<INSTALL>/registry/samples/publish-extrinsic/src/JAXRPublishExtrinsicObject.java`. This example publishes an XML file to the Registry. To run the example, follow these steps:

1. Go to the directory `<INSTALL>/registry/samples/publish-extrinsic`.
2. Type the following command:
   `ant run`

Using an Extrinsic Object as a Specification Link

You can publish an `ExtrinsicObject` by itself, but it is also very common to create an `ExtrinsicObject` to use as the `SpecificationLink` object for a `ServiceBinding` object (see Creating Services and Service Bindings, page 54). The `ExtrinsicObject` typically refers to a WSDL file.

1. Create a `SpecificationLink` object.
2. Store the WSDL document in the repository and create an `ExtrinsicObject` that refers to it. Set the extrinsic object's type to `WSDL` and its mime type to `text/xml`.
3. Specify the extrinsic object as the `specificationObject` attribute of the `SpecificationLink` object.
4. Add the `SpecificationLink` object to the `ServiceBinding` object.
5. Add the `ServiceBinding` object to the `Service` object.
6. Save the `Service` object.

After you create a `Service` and `ServiceBinding`, create a `SpecificationLink`:

```
SpecificationLink specLink = blcm.createSpecificationLink();
specLink.setName("Spec Link Name");
specLink.setDescription("Spec Link Description");
```

Create an ExtrinsicObject as described in Creating an Extrinsic Object (page 60). Use the ID for the WSDL concept and the text/xml MIME type.

```
String conceptId =
  "urn:oasis:names:tc:ebxml-
regrep:ObjectType:RegistryObject:ExtrinsicObject:WSDL";
Concept objectTypeConcept =
  (Concept) bqm.getRegistryObject(conceptId);
((ExtrinsicObjectImpl)eo).setObjectType(objectTypeConcept);
eo.setMimeType("text/xml");
```

Set the ExtrinsicObject as the specification object for the SpecificationLink:

```
specLink.setSpecificationObject(eo);
```

Add the SpecificationLink to the ServiceBinding, then add the objects to their collections and save the services.

```
binding.addSpecificationLink(specLink);
serviceBindings.add(binding);
...
```

When you remove a service from the Registry, the service bindings and specification links are also removed. However, the extrinsic objects associated with the specification links are not removed.

Creating an Extrinsic Object as a Specification Link: Example

For an example of creating an extrinsic object as a specification link, see *<INSTALL>*/registry/samples/publish-service/src/JAXRPublishService.java. This example publishes a WSDL file to the Registry. To run the example, follow these steps:

1. Go to the directory *<INSTALL>*/registry/samples/publish-service.
2. Type the following command:
   ```
   ant run
   ```

Organizing Objects Within Registry Packages

Registry packages allow you to group a number of logically related registry objects, even if the individual member objects belong to different owners. For example, you could create a `RegistryPackage` object and add to it all objects in the Registry whose names shared a particular unique string or that all contained a `Slot` with the same name and value.

To create a `RegistryPackage` object, call the `LifeCycleManager.createRegistryPackage` method, which takes a `String` or `InternationalString` argument. Then call the `RegistryPackage.addRegistryObject` or `RegistryPackage.addRegistryObjects` method to add objects to the package.

For example, you could create a `RegistryPackage` object named “SunPackage”:

```
RegistryPackage pkg =
  blcm.createRegistryPackage("SunPackage");
```

Then, after finding all objects with the string `"Sun"` in their names, you could iterate through the results and add each object to the package:

```
pkg.addRegistryObject(object);
```

A common use of packages is to organize a set of extrinsic objects. A registry administrator can load a file system into the Registry, storing the directories as registry packages and the files as the package contents. See the *Administration Guide* for more information.

Organizing Objects Within Registry Packages: Examples

For examples of using registry packages, see the two examples in `<INSTALL>/registry/samples/packages/src`: `JAXRPublishPackage.java` and `JAXRQueryPackage.java`. The first example publishes a `RegistryPackage` object that includes all objects in the Registry whose names contain the string `"free"`. The second example searches for this package and displays its contents. To run the examples, follow these steps:

1. Go to the directory `<INSTALL>/registry/samples/packages`.
2. Type the following command:

```
ant pub-pkg
```

3. Type the following command:

```
ant query-pkg
```

Changing the State of Objects in the Registry

You add an `AuditableEvent` object to the audit trail of an object when you publish it to the Registry or when you modify it in any way. See Retrieving the Audit Trail of an Object (page 33) for details on these events and on how to obtain information about them. Table 8–5 on page 33 describes the events and how they are created.

Many events are created as a side effect of some other action:

- Saving an object to the Registry creates an `EVENT_TYPE_CREATED` event.
- The following actions create an `EVENT_TYPE_VERSIONED` event:
    - Changing an object's name or description
    - Adding, modifying, or removing a `Classification`, `ExternalIdentifier`, `ExternalLink`, or `Slot`
    - For an `Organization` or `User`, adding, modifying, or removing a `PostalAddress` or `TelephoneNumber`

  You can retrieve version information for an object. See Retrieving the Version of an Object (page 34) for details.

Note: At this release, versioning of objects is disabled. All objects have a version of 1.1.

You can also change the state of objects explicitly. This feature may be useful in a production environment where different versions of objects exist and you wish to use some form of version control. For example, you can approve a version of an object for general use and deprecate an obsolete version before you remove it.

If you change your mind after deprecating an object, you can undeprecate it. You can perform these actions only on objects you own.

- You can approve objects by using the `LifeCycleManagerImpl.approveObjects` method. This feature is implementation-specific.
- You can deprecate objects by using the `LifeCycleManager.deprecateObjects` method.
- You can undeprecate objects by using the `LifeCycleManager.unDeprecateObjects` method.

The `LifeCycleManagerImpl.approveObjects` method has the following signature:

```
public BulkResponse approveObjects(java.util.Collection keys)
  throws JAXRException
```

It is possible to restrict access to these actions to specific users, such as registry administrators.

No `AuditableEvent` is created for actions that do not alter the state of a `RegistryObject`. For example, queries do not generate an `AuditableEvent`, and no `AuditableEvent` is generated for a `RegistryObject` when it is added to a `RegistryPackage` or when you create an `Association` with the object as the source or target.

Changing the State of Objects in the Registry: Examples

For examples of approving, deprecating, undeprecating objects, see the examples in `<INSTALL>/registry/samples/auditable-events/src`: `JAXRApproveObject.java`, `JAXRDeprecateObject.java`, and `JAXRUndeprecateObject.java`. Each example performs an action on an object whose unique identifier you specify, then displays the object's audit trail so that you can see the effect of the example. To run the examples, follow these steps:

1. Go to the directory `<INSTALL>/registry/samples/packages`.
2. Type the following command:
   `ant approve-obj -Did=`*`id_string`*
3. Type the following command:
   `ant deprecate-obj -Did=`*`id_string`*
4. Type the following command:

```
ant undeprecate-obj -Did=id_string
```

The object you specify should be one that you created.

Removing Objects From the Registry and Repository

A registry allows you to remove from it any objects that you have submitted to it. You use the object's ID as an argument to the `LifeCycleManager.deleteObjects` method.

The following code fragment deletes the object that corresponds to a specified key string and then displays the key again so that the user can confirm that it has deleted the correct one.

```
String id = key.getId();
Collection keys = new ArrayList();
keys.add(key);
BulkResponse response = blcm.deleteObjects(keys);
Collection exceptions = response.getException();
if (exceptions == null) {
  System.out.println("Objects deleted");
  Collection retKeys = response.getCollection();
  Iterator keyIter = retKeys.iterator();
  javax.xml.registry.infomodel.Key orgKey = null;
  if (keyIter.hasNext()) {
    orgKey =
      (javax.xml.registry.infomodel.Key) keyIter.next();
    id = orgKey.getId();
    System.out.println("Object key was " + id);
  }
}
```

Deleting an `Organization` does not delete the `Service` and `User` objects that belong to the `Organization`. You must delete them separately.

Deleting a `Service` object deletes the `ServiceBinding` objects that belong to it, and also the `SpecificationLink` objects that belong to the `ServiceBinding` objects. Deleting the `SpecificationLink` objects, however, does not delete the associated `ExtrinsicObject` instances and their associated repository items. You must delete the extrinsic objects separately.

`AuditableEvent` and `Association` objects are not always deleted when the objects associated with them are deleted. You may find that as you use the Registry, a large number of these objects accumulates.

Removing Objects from the Registry: Example

For an example of deleting an object from the Registry, see `<INSTALL>/registry/samples/delete-object/src/JAXRDelete.java`. This example deletes the object whose unique identifier you specify. To run the example, follow these steps:

1. Go to the directory `<INSTALL>/registry/samples/delete-object`.
2. Type the following command:
   ```
   ant run -Did=id_string
   ```

Further Information

For more information about JAXR, registries, and web services, see the following:

- Java Specification Request (JSR) 93: JAXR 1.0:
  `http://jcp.org/jsr/detail/093.jsp`
- JAXR home page:
  `http://java.sun.com/xml/jaxr/`
- ebXML:
  `http://www.ebxml.org/`
- Java 2 Platform, Enterprise Edition:
  `http://java.sun.com/j2ee/`
- Java Technology and XML:
  `http://java.sun.com/xml/`
- Java Technology and Web Services:
  `http://java.sun.com/webservices/`

9 Administering the Service Registry

THIS chapter describes how to use the Administration Tool (“the Admin Tool”) for the Service Registry.

This chapter contains the following sections:

- About the Admin Tool
- Starting the Admin Tool
- Using the Admin Tool
- Using Admin Tool Commands
- Other Administration Tasks

About the Admin Tool

The Service Registry Administration Tool provides a simple command-line interface for common administration tasks, such as adding associations to the Registry and removing objects from the Registry.

The tool can operate in either of two modes:

- In batch mode, you specify one or more commands on the tool’s command line.
- In interactive mode, you enter commands in the tool’s interactive shell.

In keeping with the “files and folders” metaphor used for `RegistryObject` objects in `RegistryPackage` objects, several commands, such as `ls` and `rm`, mimic both the name and the behavior of well-known UNIX® commands that operate on files and folders. Other commands have no corresponding UNIX equivalent.

Starting the Admin Tool

To start the Admin Tool, go to the `bin` directory of the registry and run the `registry-admin` script.

On a Windows system:

```
cd <JWSDP_HOME>\registry\bin
admin-tool [options]...
```

On a UNIX system:

```
cd <JWSDP_HOME>/registry/bin
admin-tool.sh [options]...
```

The <*JWSDP_HOME*> location is the directory where you installed the Java WSDP.

To exit the Admin Tool, use the `quit` command.

Batch Mode

To run the Admin Tool in batch mode, specify the `-command` option on the command line when you start the Admin Tool.

For example, the following command executes the `ls` command:

```
java -jar ~/jwsdp-1.6/registry/lib/admin-tool.jar -command "ls
*.html"
```

The Admin Tool echoes your commands and the tool’s responses to the screen and then exits after your commands have been executed.

Make sure that you properly escape any characters that are significant to your shell.

Interactive Mode

To run the Admin Tool in interactive mode, start the Admin Tool shell by specifying no options on the command line:

```
java -jar <JWSDP_HOME>/registry/lib/admin-tool.jar
```

The Admin Tool displays the following prompt and waits for your input:

```
admin>
```

Admin Tool Command-line Options

The Admin Tool recognizes the command-line options listed in Synopsis and described in Options.

Synopsis

```
[-alias alias] [-command commands] [-debug] [-help]
-keypass keypass [-localdir localdir] [-locale locale]
[-registry url] [-root locator [-create]]
[-sqlselect SQL_statement] [-v | -verbose]
```

Options

`-alias`

The alias to use when accessing the user's certificate in the keystore. At this release, this option is not meaningful.

`-command`

The Admin Tool command sequence to run instead of getting commands from the interactive shell. Use a semicolon (`;`) to separate multiple commands. It is not necessary to include a `quit` command in *`commands`*. If you need to use a semicolon that is not a command separator, precede it by a backslash:

```
\;
```

The shell in which you run the Admin Tool may require you to escape the backslash with a second backslash:

```
\\;
```

If any command contains spaces, enclose the entire command sequence in single or double quotes so that the tool will treat the sequence as one command-line parameter instead of several. If your shell also interprets a semicolon as separating shell commands, you always have to put sequences of multiple Admin Tool commands in quotation marks.

`-create`

If necessary, create the `RegistryPackage` specified by the `-root` option as well as any parent `RegistryPackage` objects as needed. This option is valid only if the user running the Admin Tool is authorized to create objects.

`-debug`

Outputs extra information that is useful when debugging.

`-help`

Provides a list of these options.

`-keypass`

The password to use when accessing a user's certificate in the keystore. At this release, this option is not meaningful.

`-localdir`

The base directory in the local file system for commands that relate to files in the local file system.

`-locale`

The locale (for example, `en` or `fr_CA`) to use for selecting the resource bundle to use for error and status messages. The default is determined by the Java^{TM} Virtual Machine.

`-registry`

The URL of the ebXML registry to which to connect. The default is `http://localhost:8080/soar/registry/soap`.

`-root`

The locator (for example, `/registry/userData`) of the `RegistryPackage` to use as the base for those commands that treat the repository as a tree of `RegistryPackage` objects that each contain other `RegistryObject` and `RegistryPackage` objects. The default is the `RegistryPackage` that is defined for all users' data: `/registry/userData`.

`-sqlselect`

Execute *`SQL_statement`* to select registry objects. This should be a complete SQL statement that starts with `select`. The SQL statement must be enclosed in quotation marks, but it does not have to be terminated by a semicolon.

`-v | -verbose`

Specifies the verbose output of status messages.

Using the Admin Tool

This section covers the following topics:

- Permissions
- Displaying Exceptions
- Identifying Registry Objects
- The Effect of Locale on Specifying Names
- Case Sensitivity

Permissions

When you use the Admin Tool, you can perform only those actions that are allowed for the user whose alias and password you specified when you started the tool. Only a user with the role of administrator can perform certain commands (`chown`, for example).

Displaying Exceptions

The Admin Tool enables you to avoid viewing long stack traces when a command fails.

When a command fails, the Admin Tool prints the first line of the stack trace and the following message:

```
An error occurred when executing the function.  Use the show
exception command to view messages.
```

If you need more information, execute the `show exception` command next to see the full stack trace.

The `show exception` command always displays the stack trace of the immediately preceding command.

Identifying Registry Objects

The primary way to identify registry objects is by name. This extends to identifying `RegistryPackage` objects by the path from the registry root to the `RegistryPackage`. For example, `/registry/userData` is the path to the `userData RegistryPackage`.

Some matches for names support wildcards. Use a question mark (`?`) to match a single character, and use an asterisk (`*`) to match zero or more characters.

Some commands (for example, `cd` and `chown`) support identifying objects by their Uniform Resource Name (URN), which must include a leading `urn:`. For example, `urn:uuid:2702f889-3ced-4d49-82d1-e4cd846cb9e4`.

The `chown` command also supports using `%`*`number`* to refer to a `User` listed by a previous `users` command.

For some commands, you can enter names that contain spaces by enclosing the entire name in double quotes or by preceding each space in the name by a backslash.

The Effect of Locale on Specifying Names

A `RegistryObject` (or a `RegistryPackage`) may have multiple names, each of which is associated with a different locale.

The paths and object names that you specify are evaluated with respect to the current locale only. When you attempt to select by name a registry object that has multiple names, the Registry attempts to match the name that you provide against only one alternative for the registry object's name (the choice whose locale most closely matches the current locale), not against all the multiple names for the registry object.

For example, suppose the current `RegistryPackage` has a member object that has two names, each associated with a different locale: `red` in the `en` (English) locale and `rouge` in the `fr` (French) locale. When the current locale is `en`, the command `ls rouge` does not display that member object, but when the locale is `fr` (or one of its variants), then it does.

Case Sensitivity

Command names and literal parameters that are recognized by the Admin Tool are not case sensitive. For example, `ls`, `Ls`, and `LS` are equivalent.

Options to which you provide the value are passed literally to the code that uses the option.

Using Admin Tool Commands

The following sections describe the available commands. For each command, the synopsis and the descriptions of the options and operands observe the following typographical conventions:

- *`Italics`* indicate an option argument or operand that should be replaced by an actual value when you run the command.
- Curly braces (`{ }`) delimit a choice of options or operands where you must include one of the options or operands. The options or operands are separated by a vertical bar (`|`).
- Square brackets (`[ ]`) delimit an option or operand, or a choice of options or operands, that may be omitted.

- Ellipses (`...`) after an option or operand indicates that you may repeat the option or operand.

Anything else is literal text that you must include when running the command.

add association

Adds an `Association` object to the Registry.

Synopsis

```
add association -type association-type sourceURN targetURN
```

Description

The `add association` command adds an `Association` object of the specified type to the Registry. You can use any of the following types:

- `AccessControlPolicyFor`
- `AffiliatedWith` (which has the subconcepts `EmployeeOf` and `MemberOf`)
- `Contains`
- `ContentManagementServiceFor`
- `EquivalentTo`
- `Extends`
- `ExternallyLinks`
- `HasFederationMember`
- `HasMember`
- `Implements`
- `InstanceOf`
- `InvocationControlFileFor` (which has the subconcepts `Cataloging-ControlFileFor` and `ValidationControlFileFor`)
- `OffersService`
- `OwnerOf`
- `RelatedTo`
- `Replaces`
- `ResponsibleFor`
- `SubmitterOf`
- `Supersedes`
- `Uses`

Options

`-type`

The type of the `Association` object.

Operands

`sourceURN`

The URN of the source object.

targetURN

The URN of the target object.

Example

The following command (all on one line) creates a `RelatedTo` relationship between the objects with the two specified URNs.

```
admin> add association -type RelatedTo \
urn:uuid:ab80d8f7-3bea-4467-ad26-d04a40045446 \
urn:uuid:7a54bbca-2131-4a49-8ecc-e7b4ac86c4fd
```

add user

Adds a user to the Registry.

Synopsis

```
add user [-edit] [-load file] [-firstName string] [-lastName
string] [-middleName string] [-alias string] [-keypass string]
[-post1.type string] [-post1.city string] [-post1.country
string] [-post1.postalcode string] [-post1.stateOrProvince
string] [-post1.street string] [-post1.streetNumber string]
[-post2.type string] [-post2.city string] [-post2.country
string] [-post2.postalcode string] [-post2.stateOrProvince
string] [-post2.street string] [-post2.streetNumber string]
[-post3.type string] [-post3.city string] [-post3.country
string] [-post3.postalcode string] [-post3.stateOrProvince
string] [-post3.street string] [-post3.streetNumber string]
[-telephone1.type string] [-telephone1.areaCode string]
[-telephone1.countryCode string] [-telephone1.extension
string] [-telephone1.number string] [-telephone1.URL string]
[-telephone2.type string] [-telephone2.areaCode string]
[-telephone2.countryCode string] [-telephone2.extension
string] [-telephone2.number string] [-telephone2.URL string]
[-telephone3.type string] [-telephone3.areaCode string]
[-telephone3.countryCode string] [-telephone3.extension
string] [-telephone3.number string] [-telephone3.URL string]
[-email1.type string] [-email1.address string] [-email2.type
string] [-email2.address string] [-email3.type string]
[-email3.address string]
```

Description

The `add user` command adds a `User` object. A `User` object contains at least one `PostalAddress`, `TelephoneNumber`, and `EmailAddress` object. The information that you provide is checked for validity using the same criteria as when you add a new user using the Web Console or the JAXR API.

Specify the information about the user either on the command line itself or by using the `-load` option to specify a Java properties file with the information. The information options and the `-load` option are evaluated in the order they appear on the command line. For example, you can specify some properties on the command line, load others from a properties file, and then override information in the properties file with later command-line options.

Specify at least one postal address, telephone number, and email address for the new user. You can specify up to three of each type. If you need more, you can add them later using the Web Console or JAXR.

When you specify a postal address, telephone number, or email address, you must provide a value for its type: for example, `-emailType OfficeEmail`.

You can use shorthand options (such as `-fn`) on the command line for some of the common information that is required for every user, but you must use the longer form when providing the information in a properties file. For example, you can specify the user's first email address on the command line using either `-email1.address`, `-emailAddress`, or `-email`, but when you specify it in a properties file, you must use `email1.address=`. Because there is only one option for the user's second email address, you must use `-email2.address` on the command line and `email2.address=` in a properties file.

If you specify the `-edit` option, the Admin Tool launches an editor so that you can edit the new user's information. See the option description for details.

The properties files that you load with `-load` or edit with `-edit` use the IS0-8859-1 charset, as do all Java properties files. See the documentation for `java.util.Properties.load(InputStream)` for details on how to represent other characters not in ISO-8859-1 in properties files.

Options

`-edit`

Causes the Admin Tool to launch an editor so that you can edit the new user’s information. The tool launches the editor after evaluating the other command-line parameters, so editing starts with the result of evaluating any information specified on the command line or with a properties file. The editing program must terminate without error before the command can continue. (At the time of this writing, -edit currently works with emacsclient and the NetBeans command bin/runide.sh --open (but not very well), has not been shown to work with vi, and has not been tested on Windows.)

`-load`

Specifies a Java properties file whose contents specify properties for the user. The property names are the same as those of the long form of the `add user` command options (for example, `lastName` and `post1.type`).

`-fn | -firstName`

Specifies the first name of a user.

`-ln | -lastName`

Specifies the last name of a user. A last name is required; it must be specified either on the command line or in a properties file.

`-mn | -middleName`

Specifies the middle name of a user.

`-alias`

The alias to use when accessing the user’s certificate in the keystore.

`-keypass`

The password to use when accessing a user’s certificate in the keystore.

`-postalType | -post1.type`

The type of the first `PostalAddress`. The type is required; it must be specified either on the command line or in a properties file. It is an arbitrary string (for example, `Office` or `Home`).

`-city | -post1.city`

The city of the first `PostalAddress`.

`-country | -post1.country`

The country of the first `PostalAddress`.

`-postalcode | -postcode | -zip | -post1.postalcode`

The postal code of the first `PostalAddress`.

`-stateOrProvince | -state | -province | -post1.stateOrProvince`

The state or province of the first `PostalAddress`.

`-street | -post1.street`

The street name of the first `PostalAddress`. The street is required; it must be specified either on the command line or in a properties file.

`-streetNumber | -number | -post1.streetNumber`

The street number of the first `PostalAddress`.

`-post2.type`

The type of the second `PostalAddress`. If a second `PostalAddress` is specified, the type is required; it must be specified either on the command line or in a properties file. It is an arbitrary string (for example, `Office` or `Home`).

`-post2.city`

The city of the second `PostalAddress`.

`-post2.country`

The country of the second `PostalAddress`.

`-post2.postalcode`

The postal code of the second `PostalAddress`.

`-post2.stateOrProvince`

The state or province of the second `PostalAddress`.

`-post2.street`

The street name of the second `PostalAddress`. If a second `PostalAddress` is specified, the street is required; it must be specified either on the command line or in a properties file.

`-post2.streetNumber`

The street number of the second `PostalAddress`.

`-post3.type`

The type of the third `PostalAddress`. If a third `PostalAddress` is specified, the type is required; it must be specified either on the command line or in a properties file. It is an arbitrary string (for example, `Office` or `Home`).

`-post3.city`

The city of the third `PostalAddress`.

`-post3.country`

The country of the third `PostalAddress`.

`-post3.postalcode`

The postal code of the third `PostalAddress`.

`-post3.stateOrProvince`

The state or province of the third `PostalAddress`.

`-post3.street`

The street name of the third `PostalAddress`. If a third `PostalAddress` is specified, the street is required; it must be specified either on the command line or in a properties file.

`-post3.streetNumber`

The street number of the third `PostalAddress`.

`-phoneType | -telephone1.type`

The type of the first `TelephoneNumber`. The type is required; it must be specified either on the command line or in a properties file. It can have any of the following values: `Beeper`, `FAX`, `HomePhone`, `MobilePhone`, or `OfficePhone`.

`-areaCode | -telephone1.areaCode`

The area code of the first `TelephoneNumber`.

`-countryCode | -telephone1.countryCode`

The country code of the first `TelephoneNumber`.

`-extension | -telephone1.extension`

The extension of the first `TelephoneNumber`.

`-number | -telephone1.number`

The telephone number suffix, not including the country or area code, of the first `TelephoneNumber`. The number is required; it must be specified either on the command line or in a properties file.

`-URL | -telephone1.URL`

The URL of the first `TelephoneNumber` (the URL that can dial this number electronically).

`-telephone2.type`

The type of the second `TelephoneNumber`. If a second `TelephoneNumber` is specified, the type is required; it must be specified either on the command line or in a properties file. It can have any of the following values: `Beeper`, `FAX`, `HomePhone`, `MobilePhone`, or `OfficePhone`.

`-telephone2.areaCode`

The area code of the second `TelephoneNumber`.

`-telephone2.countryCode`

The country code of the second `TelephoneNumber`.

`-telephone2.extension`

The extension of the second `TelephoneNumber`.

`-telephone2.number`

The telephone number suffix, not including the country or area code, of the second `TelephoneNumber`. If a second `TelephoneNumber` is specified, the number is required; it must be specified either on the command line or in a properties file.

`-telephone2.URL`

The URL of the second `TelephoneNumber` (the URL that can dial this number electronically).

`-telephone3.type`

The type of the third `TelephoneNumber`. If a third `TelephoneNumber` is specified, the type is required; it must be specified either on the command line or in a properties file. It can have any of the following values: `Beeper`, `FAX`, `HomePhone`, `MobilePhone`, or `OfficePhone`.

`-telephone3.areaCode`

The area code of the third `TelephoneNumber`.

`-telephone3.countryCode`

The country code of the third `TelephoneNumber`.

`-telephone3.extension`

The extension of the third `TelephoneNumber`.

`-telephone3.number`

The telephone number suffix, not including the country or area code, of the third `TelephoneNumber`. If a third `TelephoneNumber` is specified, the number is required; it must be specified either on the command line or in a properties file.

`-telephone3.URL`

The URL of the third `TelephoneNumber` (the URL that can dial this number electronically).

`-emailType | -email1.type`

The type of the first `EmailAddress`. The type is required; it must be specified either on the command line or in a properties file. It can have either of the following values: `HomeEmail` or `OfficeEmail`.

`-emailAddress | -email | -email1.address`

The first email address. The first email address is required.

`-email2.type`

The type of the second `EmailAddress`. If a second `EmailAddress` is specified, the type is required; it must be specified either on the command line or in a properties file. It can have either of the following values: `HomeEmail` or `OfficeEmail`.

`-email2.address`

The second email address.

`-email3.type`

The type of the second `EmailAddress`. If a third `EmailAddress` is specified, the type is required; it must be specified either on the command line or in a properties file. It can have either of the following values: `HomeEmail` or `OfficeEmail`.

`-email3.address`

The third email address.

Examples

The following command loads the User properties from the file `JaneSmith.properties` in the user's home directory.

```
admin> add user -load ~/JaneSmith.properties
```

The following command (all on one line) specifies the minimum properties required to create a User.

```
admin> add user -ln Smith -postaltype Office \
-street “Smith Street” -phonetype Office \
-number 333-3333 -emailtype OfficeEmail \
-emailaddress JaneSmith@JaneSmith.com
```

cd

Changes the `RegistryPackage` location.

Synopsis

```
cd {locator | URN}
```

Description

Change directory (metaphorically) to the `RegistryPackage` at the specified path or with the specified URN.

Change to a specified URN when there are multiple `RegistryPackage` objects with the same path (for the current locale).

Operands

`locator`

The path of names of registry objects from the root of the repository to an object in the repository, with each name preceded by a forward slash (`/`).

For example, the locator for the `userData` `RegistryPackage` that is a member of the `registry` `RegistryPackage` (which is not itself a member of any `RegistryPackage`) is `/registry/userData`, and the locator for the `folder1` `RegistryPackage` that is a member of the `userData` `RegistryPackage` is `/registry/userData/folder1`.

If you used the `-root` option to specify the `RegistryPackage` locator when you started the Admin Tool, the locator value is relative to that root.

`URN`

The URN of the `RegistryPackage`, which must be a URN starting with `urn:`.

Examples

The following command changes the directory to the `RegistryPackage` with the URN `urn:uuid:92d3fd01-a929-4eba-a5b4-a3f036733017`.

```
admin> cd urn:uuid:92d3fd01-a929-4eba-a5b4-a3f036733017
```

The following command changes the directory to the location `/registry/userData/myData`.

```
admin> cd /registry/userData/myData
```

chown

Changes the owner of a `RegistryObject`.

Synopsis

```
chown {URN | %index}
```

Description

The `chown` command changes the ownership of the objects selected with a preceding `select` command to the user specified by either the URN or the reference to the user’s URN when listed by a preceding `users` command.

Only a user with the role of administrator can execute this command successfully.

Operands

URN

The User specified by the URN.

%index

A numerical reference to a URN for a user listed in a preceding `users` command.

Examples

The following command changes the ownership of the selected objects to the user specified by the URN `urn:uuid:26aa17e6-d669-4775-bfe8-a3a484d3e079`.

```
admin> chown urn:uuid:26aa17e6-d669-4775-bfe8-a3a484d3e079
```

The following command changes the ownership of the selected objects to the user with the number 2 in a preceding `users` command.

```
admin> chown %2
```

cp

Copies files and folders into the Registry.

Synopsis

```
cp [-owner {URN | %index}] [-exclude pattern]... pattern...
```

Description

The `cp` command copies files and folders into the Registry as `RegistryPackage` and `ExtrinsicObject` objects, respectively.

The local directory on the local file system from which to copy files and folders defaults to the current directory from which you started the Admin Tool. You can use the `-localdir` option to change the local directory when you start the Admin Tool, or you can use the `lcd` command to change it after the Admin Tool has started. You can get the absolute path of the current local directory using the `show localdir` command.

The command is recursive. That is, if you specify a directory, the command copies all the files and folders under the directory.

Options

`-owner`

Sets the owner of the copied registry objects to the user specified by the *`URN`* or *`%index`* argument. See the description of the `chown` command for a description of these arguments. You must have the role of administrator to specify an owner other than yourself.

`-exclude`

Copies all files except those whose names contain the specified pattern, where *`pattern`* is a pattern comprising literal characters and the special characters asterisk (`*`) (representing zero or more characters) and question mark (`?`) (representing one and only one character).

You can specify this option more than once.

Operands

`pattern`

The files or folders to be copied, specified by a pattern comprising literal characters and the special characters asterisk (`*`) (representing zero or more characters) and question mark (`?`) (representing one and only one character). You can specify more than one *`pattern`*.

Examples

The following command copies the directory `mydir` to the Registry, to be owned by the user with the number 4 in a preceding `users` command.

```
admin> cp -owner %4 mydir
```

The following command copies the directory `mydir` to the Registry, excluding files and directories that end with the string `.z` or `.c`.

```
admin> cp mydir -exclude \.z -exclude \.c
```

echo

Echoes a string.

Synopsis

```
echo string
```

Description

The `echo` command echoes the specified *`string`* to the output. It is most useful when you specify it in the `-command` option when you run the Admin Tool in batch mode.

Operand

`string`

A sequence of characters.

Example

The following command prints the date and the result of the `ls` command into a log file.

```
registry-admin.sh -command "echo ‘date‘; ls" > admin.log
```

help

Displays information about commands.

Synopsis

```
help [command_name]
```

Description

The `help` command displays information about the available commands or a specified command.

For commands with subcommands, such as `add` and `show`, the `help` command displays information about the subcommands.

If you do not specify an argument, the `help` command displays usage information for all commands.

Operand

`command_name`

The name of an Admin Tool command.

Examples

The following command displays usage information for all commands.

```
admin> help
```

The following command displays usage information for the `lcd` command.

```
admin> help lcd
```

The following command displays usage information for the `add` subcommands.

```
admin> help add
```

lcd

Changes the current directory on the local file system.

Synopsis

```
lcd [path_name]
```

Description

The `lcd` command changes the current local directory on the local file system.

If you do not specify an argument, the `lcd` command changes the current directory to your default home directory.

Operand

path_name

A directory name, which may be absolute or relative.

Examples

The following command changes the current local directory to the `/usr/share` directory.

```
admin> cd /usr/share
```

The following command changes the current local directory to your default home directory on the local file system.

```
admin> lcd
```

ls

Lists the objects in the current `RegistryPackage`.

Synopsis

```
ls [{pattern | URN}...]
```

Description

The `ls` command lists the objects in the current `RegistryPackage` or, when a *`pattern`* or *`URN`* is provided, list the objects in the current `RegistryPackage` whose names (in the current locale) or unique identifiers match *`pattern`* or *`URN`*.

Operands

`pattern`

A pattern comprising literal characters and the special characters asterisk (`*`) (representing zero or more characters) and question mark (`?`) (representing one and only one character). You can specify more than one *`pattern`*.

`URN`

A URN starting with `urn:`, for example, `urn:uuid:4a6741e7-4be1-4cfb-960a-e5520356c4fd`. You can specify more than one *`URN`*.

Examples

The following command lists all the objects in the current `RegistryPackage`.

```
admin> ls
```

The following command lists all the objects whose name matches the pattern `urn:bird:poultry:chicken` or whose ID is `urn:bird:poultry:chicken`.

```
admin> ls urn:bird:poultry:chicken
```

The following command lists all the objects whose name matches the pattern `*bird*`. (It would also list the objects whose ID is `*bird*`, if `*bird*` were a valid ID.)

```
admin> ls *bird*
```

The following command lists all the objects whose name matches the pattern `*bird*` or whose name matches the pattern `urn:bird:poultry:chicken` or whose ID is `urn:bird:poultry:chicken`.

```
admin> ls *bird* urn:bird:poultry:chicken
```

pwd

Displays the path to the current `RegistryPackage`.

Synopsis

```
pwd
```

Description

The `pwd` command displays the path (or paths) to the current `RegistryPackage` using the best-matching names for the current locale. Also displays the locale for the path.

Example

```
admin> pwd
(en_US) /registry/userData
```

quit

Exits the Admin Tool.

Synopsis

```
quit
```

Description

The `quit` command exits the Admin Tool.

Example

```
admin> quit
```

rm

Removes objects from a `RegistryPackage`.

Synopsis

```
rm [-d] [-r] {pattern | URN}...
```

Description

The `rm` command removes the member objects of the current `RegistryPackage` whose names (in the current locale) match the patterns specified by a *`pattern`* or *`URN`*.

When a matching `RegistryObject` is a member of multiple `RegistryPackage` objects, this command removes only the association between the current `RegistryPackage` and the object. The object is removed from the Registry only when the removal of the association leaves the object with no association with any other `RegistryObject`.

When a matching member object is itself a `RegistryPackage` that contains other objects, neither the object nor the association between the current `RegistryPackage` and the member `RegistryPackage` is removed unless either the `-r` or the `-d` option is specified.

When both the `-d` and `-r` options are specified, the `-d` option is applied recursively, so all objects that would be selected by `-r` (and their associations) are removed whether or not they have other associations.

Options

`-d`

Removes the association between the current `RegistryPackage` and the specified `RegistryPackage`. Removes the specified `RegistryPackage` only if its only remaining associations are to its member objects. Member objects of the now-removed `RegistryPackage` that are not anchored by being the target of other `HasMember` associations are now accessible as members of the root of the Registry.

`-r`

Removes the specified `RegistryPackage` object and all its descendant objects (except when an object has other associations).

Operands

`pattern`

A pattern comprising literal characters and the special characters asterisk (`*`) (representing zero or more characters) and question mark (`?`) (representing one and only one character). You can specify more than one *`pattern`*.

`URN`

A URN starting with `urn:`, for example, `urn:uuid:4a6741e7-4be1-4cfb-960a-e5520356c4fd`. You can specify more than one *`URN`*.

Examples

The following command removes all `RegistryPackage` objects containing the string “stat” and all their descendants.

```
admin> rm -r *stat*
```

select

Executes an SQL `select` statement.

Synopsis

```
select [SQL]
```

Description

The `select` command selects and lists the objects specified by evaluating the entire command as an SQL query. If no argument is specified, the command lists any objects selected by a preceding `select` command.

Operand

`SQL`

An SQL `select` statement (without the leading `select` because that is already present as the name of the command).

Examples

The following command lists all `ClassificationScheme` objects in the Registry:

```
admin> select s.* from ClassificationScheme s
```

set

Sets a property value.

Synopsis

```
set property value
```

Description

The `set` command sets the value of a property of the Admin Tool shell.

The tool supports the following properties and values.

```
set debug {true | on | yes | false | off | no}
```

Enables or disables output of debugging messages.

```
set editor string
```

Sets the command to use when the Admin Tool launches an interactive editor. The default value is `/bin/vi` on UNIX and Linux systems, and is `notepad.exe` on Windows systems.

```
set verbose {true | on | yes | false | off | no}
```

Enables or disables output of more verbose messages when executing commands.

Operands

property

One of the following properties: `debug`, `editor`, `verbose`.

value

A supported value of the specified property. See the Description section for details.

Examples

The following command sets the editor to `/usr/bin/vi` instead of the default `/bin/vi`.

```
admin> set editor /usr/bin/vi
```

show

Displays a property value.

Synopsis

```
show [property]
```

Description

The `show` command displays the value of a property of the Admin Tool shell.

If no argument is specified, the command displays the values of all properties.

The command supports the following properties:

`debug`

Whether or not debugging output is enabled.

`editor`

The editor to use when the Admin Tool launches an interactive editor.

`exception`

The exception stack trace, if any, from the immediately preceding executed command.

`locale`

The current locale.

`verbose`

Whether or not verbose output is enabled.

Operands

property

The property whose current value is to be displayed. The properties `exception` and `locale` can be displayed, but you cannot use the `set` command to set them.

Example

The following command displays the exceptions from the previous command.

```
admin> show exception
```

users

Lists the current `User` objects.

Synopsis

```
users
```

Description

The `users` command lists the `User` objects currently in the Registry.

The output has the following format:

```
%index: URN lastname, firstname
```

In the output, the *index* is a numeric value that you can use, including the percent sign (`%`), to refer to a user when you run the `chown` or `cp` command. The *`lastname`* and *`firstname`* are the first and last names of the user.

Examples

The following command displays the current users:

```
admin> users
%0:  urn:uuid:2702f889-3ced-4d49-82d1-e4cd846cb9e4  user, test
%1:  urn:uuid:85428d8e-1bd5-473b-a8c8-b9d595f82728  Parker, Miles
%2: urn:uuid:921284f0-bbed-4a4c-9342-ecaf0625f9d7  Operator, Registry
%3:  urn:uuid:977d9380-00e2-4ce8-9cdc-d8bf6a4157be  Brown, Arthur
%4:  urn:uuid:abfa78d5-605e-4dbc-b9ee-a42e99d5f7cf  Guest, Registry
```

Other Administration Tasks

This section describes other tasks you may need to perform for the Registry:

- Backing Up and Restoring the Database

Backing Up and Restoring the Database

The Registry uses the Apache Derby database. By default, the database is located in the following directory:

```
$HOME/soar/platform/3.0/data/registry/soar/
```

where *`platform`* is either `tomcat` or `as8.1`.

To learn how to back up and restore the database, consult the Apache Derby documentation. To locate the documentation, follow these steps:

1. In a web browser, go to the URL `http://incubator.apache.org/derby/`.
2. Click the Manuals tab.
3. Locate the Server & Admin Guide.
4. Locate the sections on backing up and restoring databases.

A

XWS-Security Formal Schema Definition

Formal Schema Definition

This chapter shows the formal schema definition for security configuration files for XWS-Security EA 2.0. More information on using security configuration files is described in Introduction to XML and Web Services Security. More information on each of the schema elements is described in XWS-Security Configuration File Schema. Sample applications that use these elements are described in Understanding and Running the XWS-Security Sample Applications.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns="http://java.sun.com/xml/ns/xwss/config"
targetNamespace="http://java.sun.com/xml/ns/xwss/config"
elementFormDefault="qualified">
  <xs:element name="JAXRPCSecurity">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Service" type="Service_T" minOccurs="0"
        maxOccurs="unbounded"/>
        <xs:element name="SecurityEnvironmentHandler"
        type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:complexType name="Service_T">
    <xs:sequence>
        <xs:element ref="SecurityConfiguration" minOccurs="0"/>
        <xs:element name="Port" type="Port_T" minOccurs="0"
        maxOccurs="unbounded"/>
        <xs:element name="SecurityEnvironmentHandler"
        type="xs:string" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="optional"/>
    <xs:attribute name="id" type="id_T" use="optional"/>
    <xs:attribute name="conformance" use="optional">
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:enumeration value="bsp"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="useCache" type="xs:boolean" use="optional"
    default="false"/>
</xs:complexType>
<xs:complexType name="Port_T" mixed="true">
    <xs:sequence>
        <xs:element ref="SecurityConfiguration" minOccurs="0"/>
        <xs:element name="Operation" type="Operation_T"
        minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="name" use="required">
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:minLength value="1"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="conformance" use="optional">
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:enumeration value="bsp"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
</xs:complexType>
<xs:complexType name="Operation_T">
    <xs:sequence>
        <xs:element ref="SecurityConfiguration" minOccurs="0"
        maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="name" use="required">
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:minLength value="1"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
</xs:complexType>
<xs:element name="SecurityConfiguration"
type="SecurityConfiguration_T"/>
<xs:complexType name="SecurityConfiguration_T">
    <xs:sequence>
        <xs:group ref="SecurityConfigurationElements" minOccurs="0"
        maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="dumpMessages" type="xs:boolean"
    use="optional" default="false"/>
    <xs:attribute name="enableDynamicPolicy" type="xs:boolean"
    use="optional" default="false"/>
</xs:complexType>
<xs:group name="SecurityConfigurationElements">
    <xs:choice>
        <xs:element name="Timestamp" type="Timestamp_T"
        minOccurs="0"/>
        <xs:element name="RequireTimestamp"
        type="RequireTimestamp_T" minOccurs="0"/>
        <xs:element name="UsernameToken" type="UsernameToken_T"
        minOccurs="0"/>
        <xs:element name="RequireUsernameToken"
        type="RequireUsernameToken_T" minOccurs="0"/>
        <xs:element name="SAMLAssertion" type="SAMLAssertion_T"
        minOccurs="0"/>
        <xs:element name="RequireSAMLAssertion"
        type="RequireSAMLAssertion_T" minOccurs="0"/>
        <xs:element name="OptionalTargets" type="OptionalTargets_T"
        minOccurs="0"/>
        <xs:element name="Sign" type="Sign_T"/>
        <xs:element name="Encrypt" type="Encrypt_T"/>
        <xs:element name="RequireSignature"
        type="RequireSignature_T"/>
        <xs:element name="RequireEncryption"
        type="RequireEncryption_T"/>
    </xs:choice>
</xs:group>
<xs:complexType name="Timestamp_T">
    <xs:attribute name="id" type="id_T" use="optional"/>
    <xs:attribute name="timeout" type="xs:decimal" use="optional"
        default="300"/>
</xs:complexType>
<xs:complexType name="RequireTimestamp_T">
    <xs:attribute name="id" type="id_T" use="optional"/>
    <xs:attribute name="maxClockSkew" type="xs:decimal"
    use="optional" default="60"/>
    <xs:attribute name="timestampFreshnessLimit" type="xs:decimal"
    use="optional" default="300"/>
</xs:complexType>
<xs:complexType name="UsernameToken_T">
    <xs:attribute name="id" type="id_T" use="optional"/>
    <xs:attribute name="name" type="xs:string" use="optional"/>
    <xs:attribute name="password" type="xs:string" use="optional"/>
    <xs:attribute name="useNonce" type="xs:boolean" use="optional"
    default="true"/>
    <xs:attribute name="digestPassword" type="xs:boolean"
    use="optional" default="true"/>
</xs:complexType>
<xs:complexType name="RequireUsernameToken_T">
    <xs:attribute name="id" type="id_T" use="optional"/>
    <xs:attribute name="nonceRequired" type="xs:boolean"
    use="optional" default="true"/>
    <xs:attribute name="passwordDigestRequired" type="xs:boolean"
    use="optional" default="true"/>
    <xs:attribute name="maxClockSkew" type="xs:decimal"
    use="optional" default="60"/>
    <xs:attribute name="timestampFreshnessLimit" type="xs:decimal"
    use="optional" default="300"/>
    <xs:attribute name="maxNonceAge" type="xs:decimal"
    use="optional" default="900"/>
</xs:complexType>
<xs:complexType name="Encrypt_T">
    <xs:sequence minOccurs="0">
        <xs:choice minOccurs="0" maxOccurs="1">
            <xs:element name="X509Token" type="X509Token_T"/>
            <xs:element name="SAMLAssertion"
            type="SAMLAssertion_T"/>
            <xs:element name="SymmetricKey"
            type="SymmetricKey_T"/>
        </xs:choice>
        <xs:element name="KeyEncryptionMethod"
        type="KeyEncryptionMethod_T" minOccurs="0"
        maxOccurs="1"/>
        <xs:element name="DataEncryptionMethod"
        type="DataEncryptionMethod_T" minOccurs="0"
        maxOccurs="1"/>
        <xs:choice minOccurs="0" maxOccurs="unbounded">
            <xs:element name="Target" type="Target_T" minOccurs="0"
            maxOccurs="unbounded"/>
            <xs:element name="EncryptionTarget"
            type="EncryptionTarget_T" minOccurs="0"
            maxOccurs="unbounded"/>
        </xs:choice>
    </xs:sequence>
    <xs:attribute name="id" type="id_T" use="optional"/>
</xs:complexType>
<xs:complexType name="KeyEncryptionMethod_T">
    <xs:attribute name="algorithm" use="optional"
    default="http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p">
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:enumeration value=
                "http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p"/>
                <xs:enumeration value=
                "http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
                <xs:enumeration value=
                "http://www.w3.org/2001/04/xmlenc#kw-tripledes"/>
                <xs:enumeration value=
                "http://www.w3.org/2001/04/xmlenc#kw-aes128"/>
                <xs:enumeration value=
                "http://www.w3.org/2001/04/xmlenc#kw-aes256"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
</xs:complexType>
<xs:complexType name="DataEncryptionMethod_T">
    <xs:attribute name="algorithm" use="optional"
    default="http://www.w3.org/2001/04/xmlenc#aes128-cbc">
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:enumeration value=
                "http://www.w3.org/2001/04/xmlenc#aes128-cbc"/>
                <xs:enumeration value=
                "http://www.w3.org/2001/04/xmlenc#aes256-cbc"/>
                <xs:enumeration value=
                "http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
</xs:complexType>
<xs:complexType name="EncryptionTarget_T">
    <xs:sequence>
        <xs:element name="Transform" type="Transform_T"
        minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="type" type="xs:string" use="optional"
    default="qname"/>
    <xs:attribute name="contentOnly" type="xs:boolean" use="optional"
    default="true"/>
    <xs:attribute name="enforce" type="xs:boolean" use="optional"
    default="true"/>
    <xs:attribute name="value" type="xs:string" use="required"/>
</xs:complexType>
<xs:complexType name="SymmetricKey_T">
    <xs:attribute name="keyAlias" use="required">
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:minLength value="1"/>
            </xs:restriction>
            </xs:simpleType>
        </xs:attribute>
    </xs:complexType>
    <xs:complexType name="Sign_T">
        <xs:sequence>
            <xs:choice minOccurs="0" maxOccurs="1">
                <xs:element name="X509Token" type="X509Token_T"/>
                <xs:element name="SAMLAssertion"
                type="SAMLAssertion_T"/>
                <xs:element name="SymmetricKey"
                type="SymmetricKey_T"/>
            </xs:choice>
            <xs:element name="CanonicalizationMethod"
            type="CanonicalizationMethod_T" minOccurs="0"/>
            <xs:element name="SignatureMethod"
type="SignatureMethod_T"
            minOccurs="0"/>
            <xs:sequence minOccurs="0" maxOccurs="unbounded">
                <xs:element name="Target" type="Target_T" minOccurs="0"
                maxOccurs="unbounded"/>
                <xs:element name="SignatureTarget"
                type="SignatureTarget_T" minOccurs="0"
                maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:sequence>
        <xs:attribute name="id" type="id_T" use="optional"/>
        <xs:attribute name="includeTimestamp" type="xs:boolean"
        use="optional" default="true"/>
    </xs:complexType>
    <xs:complexType name="CanonicalizationMethod_T">
        <xs:attribute name="algorithm" type="xs:string" use="optional"
        default="http://www.w3.org/2001/10/xml-exc-c14n#"/>
    </xs:complexType>
    <xs:complexType name="SignatureMethod_T">
        <xs:attribute name="algorithm" type="xs:string" use="optional"
        default="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
    </xs:complexType>
    <xs:complexType name="RequireSignature_T">
        <xs:sequence minOccurs="0" maxOccurs="1">
            <xs:choice minOccurs="0" maxOccurs="1">
                <xs:element name="X509Token" type="X509Token_T"/>
                <xs:element name="SAMLAssertion"
                type="SAMLAssertion_T"/>
                <xs:element name="SymmetricKey"
                type="SymmetricKey_T"/>
            </xs:choice>
            <xs:element name="CanonicalizationMethod"
            type="CanonicalizationMethod_T" minOccurs="0"
            maxOccurs="1"/>
            <xs:element name="SignatureMethod"
type="SignatureMethod_T"
            minOccurs="0" maxOccurs="1"/>
            <xs:choice minOccurs="0"  maxOccurs="unbounded">
                <xs:element name="Target" type="Target_T" minOccurs="0"
                maxOccurs="unbounded"/>
                <xs:element name="SignatureTarget"
                type="SignatureTarget_T" minOccurs="0"
                maxOccurs="unbounded"/>
            </xs:choice>
        </xs:sequence>
        <xs:attribute name="id" type="id_T" use="optional"/>
        <xs:attribute name="requireTimestamp" type="xs:boolean"
        use="optional" default="true"/>
    </xs:complexType>
    <xs:complexType name="RequireEncryption_T">
        <xs:sequence>
            <xs:choice minOccurs="0" maxOccurs="1">
                <xs:element name="X509Token" type="X509Token_T"/>
                <xs:element name="SAMLAssertion"
                type="SAMLAssertion_T"/>
                <xs:element name="SymmetricKey"
                type="SymmetricKey_T"/>
            </xs:choice>
            <xs:element name="KeyEncryptionMethod"
            type="KeyEncryptionMethod_T" minOccurs="0"
            maxOccurs="1"/>
            <xs:element name="DataEncryptionMethod"
            type="DataEncryptionMethod_T" minOccurs="0"
            maxOccurs="1"/>
            <xs:choice minOccurs="0" maxOccurs="unbounded">
                <xs:element name="Target" type="Target_T"/>
                <xs:element name="EncryptionTarget"
                type="EncryptionTarget_T"/>
            </xs:choice>
        </xs:sequence>
        <xs:attribute name="id" type="id_T" use="optional"/>
    </xs:complexType>
    <xs:complexType name="OptionalTargets_T">
        <xs:choice minOccurs="0" maxOccurs="unbounded">
            <xs:element name="Target" type="Target_T"/>
        </xs:choice>
    </xs:complexType>
    <xs:complexType name="X509Token_T">
        <xs:attribute name="id" type="id_T" use="optional"/>
        <xs:attribute name="strId" type="id_T" use="optional"/>
        <xs:attribute name="certificateAlias" type="xs:string"
        use="optional"/>
        <xs:attribute name="keyReferenceType" use="optional"
        default="Direct">
            <xs:simpleType>
                <xs:restriction base="xs:string">
                    <xs:enumeration value="Direct"/>
                    <xs:enumeration value="Identifier"/>
                    <xs:enumeration value="IssuerSerialNumber"/>
                </xs:restriction>
            </xs:simpleType>
        </xs:attribute>
        <xs:attribute name="encodingType" use="optional">
            <xs:simpleType>
                <xs:restriction base="xs:string">
                    <xs:enumeration value="http://docs.oasis-
                    open.org/wss/2004/01/oasis-200401-wss-soap-message-
                    security-1.0#Base64Binary"/>
                </xs:restriction>
            </xs:simpleType>
        </xs:attribute>
        <xs:attribute name="valueType" type="xs:string" use="optional"/>
    </xs:complexType>
    <xs:complexType name="SAMLAssertion_T">
        <xs:attribute name="id" type="id_T" use="optional"/>
        <xs:attribute name="authorityId" type="id_T" use="optional"/>
        <xs:attribute name="strId" type="id_T" use="optional"/>
        <xs:attribute name="keyIdentifier" type="id_T" use="optional"/>
        <xs:attribute name="encodingType" use="prohibited"/>
        <xs:attribute name="keyReferenceType" use="optional"
        default="Identifier">
            <xs:simpleType>
                <xs:restriction base="xs:string">
                    <xs:enumeration value="Identifier"/>
                    <xs:enumeration value="Embedded"/>
                </xs:restriction>
            </xs:simpleType>
        </xs:attribute>
        <xs:attribute name="type" use="required">
            <xs:simpleType>
                <xs:restriction base="xs:string">
                    <xs:enumeration value="HOK"/>
                    <xs:enumeration value="SV"/>
                </xs:restriction>
            </xs:simpleType>
        </xs:attribute>
    </xs:complexType>
    <xs:complexType name="RequireSAMLAssertion_T">
        <xs:attribute name="id" type="id_T" use="optional"/>
        <xs:attribute name="authorityId" type="id_T" use="optional"/>
        <xs:attribute name="strId" type="id_T" use="optional"/>
        <xs:attribute name="type" type="xs:string" use="required"
        fixed="SV"/>
        <xs:attribute name="encodingType" use="prohibited"/>
        <xs:attribute name="keyReferenceType" use="optional"
        default="Identifier">
            <xs:simpleType>
                <xs:restriction base="xs:string">
                    <xs:enumeration value="Direct"/>
                    <xs:enumeration value="Identifier"/>
                    <xs:enumeration value="Embedded"/>
                    <xs:enumeration value="IssuerSerialNumber"/>
                </xs:restriction>
            </xs:simpleType>
        </xs:attribute>
    </xs:complexType>
    <xs:complexType name="Target_T">
            <xs:simpleContent>
            <xs:extension base="xs:string">
```

```
            <xs:attribute name="type" use="optional" default="qname">
                <xs:simpleType>
                    <xs:restriction base="xs:string">
                        <xs:enumeration value="qname"/>
                        <xs:enumeration value="uri"/>
                        <xs:enumeration value="xpath"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:attribute>
            <xs:attribute name="contentOnly" type="xs:boolean"
            use="optional" default="true"/>
            <xs:attribute name="enforce" type="xs:boolean" use="optional"
            default="true"/>
                            </xs:extension>
            </xs:simpleContent>
</xs:complexType>
<xs:complexType name="SignatureTarget_T">
    <xs:sequence minOccurs="0" maxOccurs="1">
        <xs:element name="DigestMethod" type="DigestMethod_T"
        minOccurs="0" maxOccurs="1"/>
        <xs:element name="Transform" type="Transform_T"
        minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="type" use="optional" default="qname">
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:enumeration value="qname"/>
                <xs:enumeration value="uri"/>
                <xs:enumeration value="xpath"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="enforce" type="xs:boolean" use="optional"
    default="true"/>
    <xs:attribute name="value" type="xs:string" use="optional"
    default="true"/>
</xs:complexType>
<xs:complexType name="DigestMethod_T">
    <xs:attribute name="algorithm" type="xs:string" use="optional"
    default="http://www.w3.org/2000/09/xmldsig#sha1"/>
</xs:complexType>
```

```
    <xs:complexType name="Transform_T">
        <xs:sequence>
            <xs:element name="AlgorithmParameter"
            type="AlgorithmParameter_T" minOccurs="0"
            maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="algorithm" type="xs:string" use="required"/>
    </xs:complexType>
    <xs:complexType name="AlgorithmParameter_T">
        <xs:attribute name="name" type="xs:string" use="required"/>
        <xs:attribute name="value" type="xs:string" use="required"/>
    </xs:complexType>
    <xs:simpleType name="id_T">
        <xs:restriction base="xs:string">
            <xs:minLength value="1"/>
        </xs:restriction>
    </xs:simpleType>
</xs:schema>
```

B

SJSXP JAR Files

There are two JAR files in the Sun Java System XML Streaming Parser (SJSXP) implementation of JSR 173, Streaming API for XML (StAX). Both of these JARs are located in the `<JWSDP_HOME>/sjsxp/lib` directory:

- **sjsxp.jar** – Sun implementation JAR for SJSXP
- **jsr173_api.jar** – Standard API JAR for JSR 173

The remainder of this appendix lists the contents of these JAR files. Refer to Chapter 3, "Streaming API for XML," for detailed information about StAX and Sun's SJSXP implementation.

sjsxp.jar

The `sjsxp.jar` file contains the following files:

```
META-INF/services/javax.xml.stream.XMLEventFactory
META-INF/services/javax.xml.stream.XMLInputFactory
META-INF/services/javax.xml.stream.XMLOutputFactory
META-INF/pack.properties
com/sun/xml/stream/xerces/impl/msg/DOMMessages.properties
com/sun/xml/stream/xerces/impl/msg/XMLMessages.properties
com/sun/xml/stream/xerces/impl/msg/XMLSchemaMessages.propertie
s
com/sun/xml/stream/xerces/impl/msg/XMLSerializerMessages.prope
rties
com/sun/xml/stream/BufferManager.class
com/sun/xml/stream/Constants.class
```

```
com/sun/xml/stream/Constants$ArrayEnumeration.class
com/sun/xml/stream/Entity.class
com/sun/xml/stream/Entity$ExternalEntity.class
com/sun/xml/stream/Entity$InternalEntity.class
com/sun/xml/stream/Entity$ScannedEntity.class
com/sun/xml/stream/EventFilterSupport.class
com/sun/xml/stream/FileBufferManager.class
com/sun/xml/stream/PropertyManager.class
com/sun/xml/stream/StaxEntityResolverWrapper.class
com/sun/xml/stream/StaxErrorReporter.class
com/sun/xml/stream/StaxErrorReporter$1.class
com/sun/xml/stream/StaxXMLInputSource.class
com/sun/xml/stream/StreamBufferManager.class
com/sun/xml/stream/StreamBufferManager$RewindableInputStream.c
lass
com/sun/xml/stream/XMLBufferListener.class
com/sun/xml/stream/XMLDTDScannerImpl.class
com/sun/xml/stream/XMLDocumentFragmentScannerImpl.class
com/sun/xml/stream/XMLDocumentFragmentScannerImpl$Driver.class
com/sun/xml/stream/XMLDocumentFragmentScannerImpl$Element.clas
s
com/sun/xml/stream/XMLDocumentFragmentScannerImpl$ElementStack
.class
com/sun/xml/stream/XMLDocumentFragmentScannerImpl$ElementStack
2.class
com/sun/xml/stream/XMLDocumentFragmentScannerImpl$FragmentCont
entDriver.class
com/sun/xml/stream/XMLDocumentScannerImpl.class
com/sun/xml/stream/XMLDocumentScannerImpl$ContentDriver.class
com/sun/xml/stream/XMLDocumentScannerImpl$DTDDriver.class
com/sun/xml/stream/XMLDocumentScannerImpl$PrologDriver.class
com/sun/xml/stream/XMLDocumentScannerImpl$TrailingMiscDriver.c
lass
com/sun/xml/stream/XMLDocumentScannerImpl$XMLBufferListenerImp
l.class
com/sun/xml/stream/XMLDocumentScannerImpl$XMLDeclDriver.class
com/sun/xml/stream/XMLEntityHandler.class
com/sun/xml/stream/XMLEntityManager.class
com/sun/xml/stream/XMLEntityManager$RewindableInputStream.clas
s
com/sun/xml/stream/XMLEntityReader.class
com/sun/xml/stream/XMLEntityReaderImpl.class
com/sun/xml/stream/XMLEntityStorage.class
com/sun/xml/stream/XMLErrorReporter.class
com/sun/xml/stream/XMLEventReaderImpl.class
com/sun/xml/stream/XMLNSDocumentScannerImpl.class
com/sun/xml/stream/XMLNSDocumentScannerImpl$NSContentDriver.cl
ass
```

```
com/sun/xml/stream/XMLNamespaceBinder.class
com/sun/xml/stream/XMLReaderImpl.class
com/sun/xml/stream/XMLReaderImpl$1.class
com/sun/xml/stream/XMLScanner.class
com/sun/xml/stream/XMLStreamFilterImpl.class
com/sun/xml/stream/ZephyrParserFactory.class
com/sun/xml/stream/ZephyrWriterFactory.class
com/sun/xml/stream/dtd/DTDGrammarUtil.class
com/sun/xml/stream/dtd/nonvalidating/DTDGrammar.class
com/sun/xml/stream/dtd/nonvalidating/DTDGrammar$QNameHashtable
.class
com/sun/xml/stream/dtd/nonvalidating/XMLAttributeDecl.class
com/sun/xml/stream/dtd/nonvalidating/XMLElementDecl.class
com/sun/xml/stream/dtd/nonvalidating/XMLNotationDecl.class
com/sun/xml/stream/dtd/nonvalidating/XMLSimpleType.class
com/sun/xml/stream/events/AttributeImpl.class
com/sun/xml/stream/events/CharacterEvent.class
com/sun/xml/stream/events/CommentEvent.class
com/sun/xml/stream/events/DTDEvent.class
com/sun/xml/stream/events/DummyEvent.class
com/sun/xml/stream/events/EndDocumentEvent.class
com/sun/xml/stream/events/EndElementEvent.class
com/sun/xml/stream/events/EntityDeclarationImpl.class
com/sun/xml/stream/events/EntityReferenceEvent.class
com/sun/xml/stream/events/LocationImpl.class
com/sun/xml/stream/events/NamedEvent.class
com/sun/xml/stream/events/NamespaceImpl.class
com/sun/xml/stream/events/NotationDeclarationImpl.class
com/sun/xml/stream/events/ProcessingInstructionEvent.class
com/sun/xml/stream/events/StartDocumentEvent.class
com/sun/xml/stream/events/StartElementEvent.class
com/sun/xml/stream/events/XMLEventAllocatorImpl.class
com/sun/xml/stream/events/ZephyrEventFactory.class
com/sun/xml/stream/util/ReadOnlyIterator.class
com/sun/xml/stream/writers/WriterUtility.class
com/sun/xml/stream/writers/XMLEventWriterImpl.class
com/sun/xml/stream/writers/XMLStreamWriterImpl.class
com/sun/xml/stream/writers/XMLStreamWriterImpl$Attribute.class
com/sun/xml/stream/writers/XMLStreamWriterImpl$ElementStack.cl
ass
com/sun/xml/stream/writers/XMLStreamWriterImpl$ElementState.cl
ass
com/sun/xml/stream/writers/XMLStreamWriterImpl$NamespaceContex
tImpl.class
com/sun/xml/stream/xerces/impl/io/ASCIIReader.class
com/sun/xml/stream/xerces/impl/io/UCSReader.class
com/sun/xml/stream/xerces/impl/io/UTF8Reader.class
com/sun/xml/stream/xerces/impl/msg/XMLMessageFormatter.class
```

```
com/sun/xml/stream/xerces/util/AugmentationsImpl.class
com/sun/xml/stream/xerces/util/AugmentationsImpl$Augmentations
ItemsContainer.class
com/sun/xml/stream/xerces/util/AugmentationsImpl$LargeContaine
r.class
com/sun/xml/stream/xerces/util/AugmentationsImpl$SmallContaine
r.class
com/sun/xml/stream/xerces/util/AugmentationsImpl$SmallContaine
r$SmallContainerKeyEnumeration.class
com/sun/xml/stream/xerces/util/DefaultErrorHandler.class
com/sun/xml/stream/xerces/util/EncodingMap.class
com/sun/xml/stream/xerces/util/IntStack.class
com/sun/xml/stream/xerces/util/MessageFormatter.class
com/sun/xml/stream/xerces/util/NamespaceContextWrapper.class
com/sun/xml/stream/xerces/util/NamespaceSupport.class
com/sun/xml/stream/xerces/util/NamespaceSupport$IteratorPrefix
es.class
com/sun/xml/stream/xerces/util/NamespaceSupport$Prefixes.class
com/sun/xml/stream/xerces/util/ObjectFactory.class
com/sun/xml/stream/xerces/util/ObjectFactory$ConfigurationErro
r.class
com/sun/xml/stream/xerces/util/ParserConfigurationSettings.cla
ss
com/sun/xml/stream/xerces/util/STAXAttributesImpl.class
com/sun/xml/stream/xerces/util/STAXAttributesImpl$Attribute.cl
ass
com/sun/xml/stream/xerces/util/SecuritySupport.class
com/sun/xml/stream/xerces/util/SecuritySupport12.class
com/sun/xml/stream/xerces/util/SecuritySupport12$1.class
com/sun/xml/stream/xerces/util/SecuritySupport12$2.class
com/sun/xml/stream/xerces/util/SecuritySupport12$3.class
com/sun/xml/stream/xerces/util/SecuritySupport12$4.class
com/sun/xml/stream/xerces/util/ShadowedSymbolTable.class
com/sun/xml/stream/xerces/util/SymbolHash.class
com/sun/xml/stream/xerces/util/SymbolHash$Entry.class
com/sun/xml/stream/xerces/util/SymbolTable.class
com/sun/xml/stream/xerces/util/SymbolTable$Entry.class
com/sun/xml/stream/xerces/util/SynchronizedSymbolTable.class
com/sun/xml/stream/xerces/util/URI.class
com/sun/xml/stream/xerces/util/URI$MalformedURIException.class
com/sun/xml/stream/xerces/util/XMLAttributesImpl.class
com/sun/xml/stream/xerces/util/XMLAttributesImpl$Attribute.cla
ss
com/sun/xml/stream/xerces/util/XMLAttributesIteratorImpl.class
com/sun/xml/stream/xerces/util/XMLChar.class
com/sun/xml/stream/xerces/util/XMLResourceIdentifierImpl.class
com/sun/xml/stream/xerces/util/XMLStringBuffer.class
com/sun/xml/stream/xerces/util/XMLSymbols.class
```

```
com/sun/xml/stream/xerces/xni/Augmentations.class
com/sun/xml/stream/xerces/xni/NamespaceContext.class
com/sun/xml/stream/xerces/xni/QName.class
com/sun/xml/stream/xerces/xni/XMLAttributes.class
com/sun/xml/stream/xerces/xni/XMLDTDContentModelHandler.class
com/sun/xml/stream/xerces/xni/XMLDTDHandler.class
com/sun/xml/stream/xerces/xni/XMLDocumentFragmentHandler.class
com/sun/xml/stream/xerces/xni/XMLDocumentHandler.class
com/sun/xml/stream/xerces/xni/XMLLocator.class
com/sun/xml/stream/xerces/xni/XMLResourceIdentifier.class
com/sun/xml/stream/xerces/xni/XMLString.class
com/sun/xml/stream/xerces/xni/XNIException.class
com/sun/xml/stream/xerces/xni/parser/XMLComponent.class
com/sun/xml/stream/xerces/xni/parser/XMLComponentManager.class
com/sun/xml/stream/xerces/xni/parser/XMLConfigurationException
.class
com/sun/xml/stream/xerces/xni/parser/XMLDTDContentModelFilter.
class
com/sun/xml/stream/xerces/xni/parser/XMLDTDContentModelSource.
class
com/sun/xml/stream/xerces/xni/parser/XMLDTDFilter.class
com/sun/xml/stream/xerces/xni/parser/XMLDTDScanner.class
com/sun/xml/stream/xerces/xni/parser/XMLDTDSource.class
com/sun/xml/stream/xerces/xni/parser/XMLDocumentFilter.class
com/sun/xml/stream/xerces/xni/parser/XMLDocumentScanner.class
com/sun/xml/stream/xerces/xni/parser/XMLDocumentSource.class
com/sun/xml/stream/xerces/xni/parser/XMLEntityResolver.class
com/sun/xml/stream/xerces/xni/parser/XMLErrorHandler.class
com/sun/xml/stream/xerces/xni/parser/XMLInputSource.class
com/sun/xml/stream/xerces/xni/parser/XMLParseException.class
com/sun/xml/stream/xerces/xni/parser/XMLParserConfiguration.cl
ass
com/sun/xml/stream/xerces/xni/parser/XMLPullParserConfiguratio
n.class
```

jsr173_api.jar

The `jsr173_api.jar` file contains the following files:

```
META-INF/pack.properties
javax/xml/XMLConstants.class
javax/xml/namespace/NamespaceContext.class
javax/xml/namespace/QName.class
javax/xml/stream/EventFilter.class
javax/xml/stream/FactoryConfigurationError.class
javax/xml/stream/FactoryFinder.class
```

```
javax/xml/stream/FactoryFinder$1.class
javax/xml/stream/FactoryFinder$ClassLoaderFinder.class
javax/xml/stream/FactoryFinder$ClassLoaderFinderConcrete.class
javax/xml/stream/Location.class
javax/xml/stream/StreamFilter.class
javax/xml/stream/XMLEventFactory.class
javax/xml/stream/XMLEventReader.class
javax/xml/stream/XMLEventWriter.class
javax/xml/stream/XMLInputFactory.class
javax/xml/stream/XMLOutputFactory.class
javax/xml/stream/XMLReporter.class
javax/xml/stream/XMLResolver.class
javax/xml/stream/XMLStreamConstants.class
javax/xml/stream/XMLStreamException.class
javax/xml/stream/XMLStreamReader.class
javax/xml/stream/XMLStreamWriter.class
javax/xml/stream/events/Attribute.class
javax/xml/stream/events/Characters.class
javax/xml/stream/events/Comment.class
javax/xml/stream/events/DTD.class
javax/xml/stream/events/EndDocument.class
javax/xml/stream/events/EndElement.class
javax/xml/stream/events/EntityDeclaration.class
javax/xml/stream/events/EntityReference.class
javax/xml/stream/events/Namespace.class
javax/xml/stream/events/NotationDeclaration.class
javax/xml/stream/events/ProcessingInstruction.class
javax/xml/stream/events/StartDocument.class
javax/xml/stream/events/StartElement.class
javax/xml/stream/events/XMLEvent.class
javax/xml/stream/util/EventReaderDelegate.class
javax/xml/stream/util/StreamReaderDelegate.class
javax/xml/stream/util/XMLEventAllocator.class
javax/xml/stream/util/XMLEventConsumer.class
```

Index