Java™ API for XML Messaging (JAXM) v1.1

Please send technical comments to: jaxm-final@sun.com

Maintenance Release – June 2002

Nicholas Kassem <Nick.Kassem@sun.com>
Anil Vijendran <Anil.Vijendran@sun.com>
Rajiv.Mordani <Rajiv.Mordani@sun.com>

Java™ API for XML Messaging (JAXM) Specification ("Specification")
Version: 1.1
Status: FCS
Release: June 11, 2002

NOTICE; LIMITED LICENSE GRANTS

Sun hereby grants you a fully-paid, non-exclusive, non-transferable, worldwide, limited license (without the right to sublicense), under the Specification Lead's applicable intellectual property rights to view, download, use and reproduce the Specification only for the purpose of internal evaluation, which shall be understood to include developing applications intended to run on an implementation of the Specification provided that such applications do not themselves implement any portion(s) of the Specification.

Sun also grants you a perpetual, non-exclusive, worldwide, fully paid-up, royalty free, limited license (without the right to sublicense) under any applicable copyrights or patent rights it may have in the Specification to create and/or distribute an Independent Implementation of the Specification that:(i) fully implements the Spec(s) including all its required interfaces and functionality; (ii) does not modify, subset, superset or otherwise extend the Licensor Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the Licensor Name Space other than those required/authorized by the Specification or Specifications being implemented; and (iii) passes the TCK (including satisfying the requirements of the applicable TCK Users Guide) for such Specification. The foregoing license is expressly conditioned on your not acting outside its scope. No license is granted hereunder for any other purpose.

You need not include limitations (i)-(iii) from the previous paragraph or any other particular "pass through" requirements in any license You grant concerning the use of your Independent Implementation or products derived from it. However, except with respect to implementations of the Specification (and products derived from them) by the your licensee that satisfy limitations (i)-(iii) from the previous paragraph, You may neither:(a) grant or otherwise pass through to your licensees any licenses under Sun's applicable intellectual property rights; nor (b) authorize your licensees to make any claims concerning their implementation's compliance with the Spec in question.

For the purposes of this Agreement:"*Independent Implementation*" shall mean an implementation of the Specification that neither derives from any of Sun's source code or binary code materials nor, except with an appropriate and separate license from Sun, includes any of Sun's source code or binary code materials; and "*Licensor Name Space*" shall mean the public class or interface declarations whose names begin with "java", "javax", "com.sun" or their equivalents in any subsequent naming convention adopted by Sun through the Java Community Process, or any recognized successors or replacements thereof.

This Agreement will terminate immediately without notice from Sun if you fail to comply with any material provision of or act outside the scope of the licenses granted above.

TRADEMARKS

No right, title, or interest in or to any trademarks, service marks, or trade names of Sun or Sun's licensors is granted hereunder. Sun, Sun Microsystems, the Sun logo, Java, and the Java Coffee Cup logo are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

DISCLAIMER OF WARRANTIES

THE SPECIFICATION IS PROVIDED "AS IS". SUN MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT, THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE OR THAT ANY PRACTICE OR IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADE SECRETS OR OTHER RIGHTS. This document does not represent any commitment to release or implement any portion of the Specification in any product.

THE SPECIFICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION THEREIN; THESE CHANGES WILL BE INCORPORATED INTO NEW VERSIONS OF THE SPECIFICATION, IF ANY. SUN MAY MAKE IMPROVEMENTS AND/OR CHANGES TO THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THE SPECIFICATION AT ANY TIME. Any use of such changes in the Specification will be governed by the then-current license for the applicable version of the Specification.

LIMITATION OF LIABILITY

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE, PROFITS OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED TO ANY FURNISHING, PRACTICING, MODIFYING OR ANY USE OF THE SPECIFICATION, EVEN IF SUN AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You will indemnify, hold harmless, and defend Sun and its licensors from any claims arising or resulting from:(i) your use of the Specification; (ii) the use or distribution of your Java application, applet and/or clean room implementation; and/or (iii) any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.

RESTRICTED RIGHTS LEGEND

U.S. Government:If this Specification is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in the Software and accompanying documentation shall be only as set forth in this license; this is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions).

REPORT

You may wish to report any ambiguities, inconsistencies or inaccuracies you may find in connection with your use of the Specification ("Feedback"). To the extent that you provide Sun with any Feedback, you hereby:(i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant Sun a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose related to the Specification and future versions, implementations, and test suites thereof.

(LFI#113415/Form ID#011801)

Contents

Status

JAXM.S.1 Status of This Document

This specification is being developed following the Java™ Community Process (JCP2.1). Comments from experts, participants, and the broader developer community were reviewed and incorporated into this specification.

This document is the JAXM Specification, version 1.1 and is a manitenance release of the Java™API for XML Messaging (JAXM) 1.0 specification. JAXM 1.0 was the final deliverable of JSR067 Expert Group (EG). The proposed changes specified in the JSR067 changelog and accepted on 15 April 2002, have been incorporated in this document.

JAXM.S.2 Acknowledgements

This maintenance release is the product of collaborative work within the Java Community.

JAXM.S.3 Terminology

The keywords MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL, when they appear in this document, are to be interpreted as described in RFC 2119 as quoted here:

> MUST: This word, or the terms “REQUIRED” or “SHALL”, mean that the definition is an absolute requirement of the specification.

MUST NOT: This phrase, or the phrase “SHALL NOT”, mean that the definition is an absolute prohibition of the specification.

SHOULD: This word, or the adjective “RECOMMENDED”, mean that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.

SHOULD NOT: This phrase, or the phrase “NOT RECOMMENDED”, mean that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label.

MAY: This word, or the adjective “OPTIONAL”, mean that an item is truly optional. One vendor may choose to include the item because a particular marketplace requires it or because the vendor feels that it enhances the product while another vendor may omit the same item. An implementation which does not include a particular option MUST be prepared to interoperate with another implementation which does include the option, though perhaps with reduced functionality. In the same vein an implementation which does include a particular option MUST be prepared to interoperate with another implementation which does not include the option (except, of course, for the feature the option provides.)

Preface

JAXM.P.1 Audience

This document is intended for developers using the Java programming language, e-commerce architects, and application developers focused on the development of business-to-business messaging applications. Such developers typically build-on and extend the SOAP 1.1 messaging specifcations.

Familiarity with the SOAP specifications (including the associated processing model), MIME standards, and XML is assumed.

JAXM.P.2 Abstract

The Java™ API for XML Messaging (JAXM v1.1) enables developers to write business applications that support messaging standards based on the SOAP1.1 and SOAP with Attachments specifications. Because the XML messaging standards are being developed outside of the Java™ Community Process and are evolving at different rates (driven by a diverse set of business and technical requirements), the JAXM 1.1 specification does not mandate the use of any specific XML messaging standard. The term *standard* is being used here to denote a specific usage of SOAP messaging. Within the context of this document, the term *Profile* is used to refer to a specific protocol based on SOAP1.1 (with Attachments).

JAXM compatible implementations must support SOAP1.1 [See “SOAP” on page 55] and SOAP with Attachments [See “SOAP Messages with Attachments” on page 55]. They may additionally support one or more SOAP message Profiles [See “Messaging Profiles” on page 16]. This specification makes no assumption about the number of such Profiles but assumes that they must be named in a consistent and standard way.

JAXM.P.3 Change History

This document is based on the JAXM 1.0 specification and includes the "accepted changes", as specified in the JSR067 changelog. The key changes are as follows:

- `javax.xml.soap` package was moved from the JAXM specification to a new document designates as the SOAP with Attachments API for Java™ (SAAJ) version 1.1.
- the JAXM 1.1 specification require that `SOAPCOnnectionFactory` objects as specified in the SAAJ 1.1 specification produce `SOAPConnection` objects that support `URLEndpoint` objects.

CHAPTER JAXM.1

Background

JAXM.1.1 Conceptual Model

The following figure presents a conceptual relationship between JAXM and other architectural elements required in web-based, business-to-business messaging.

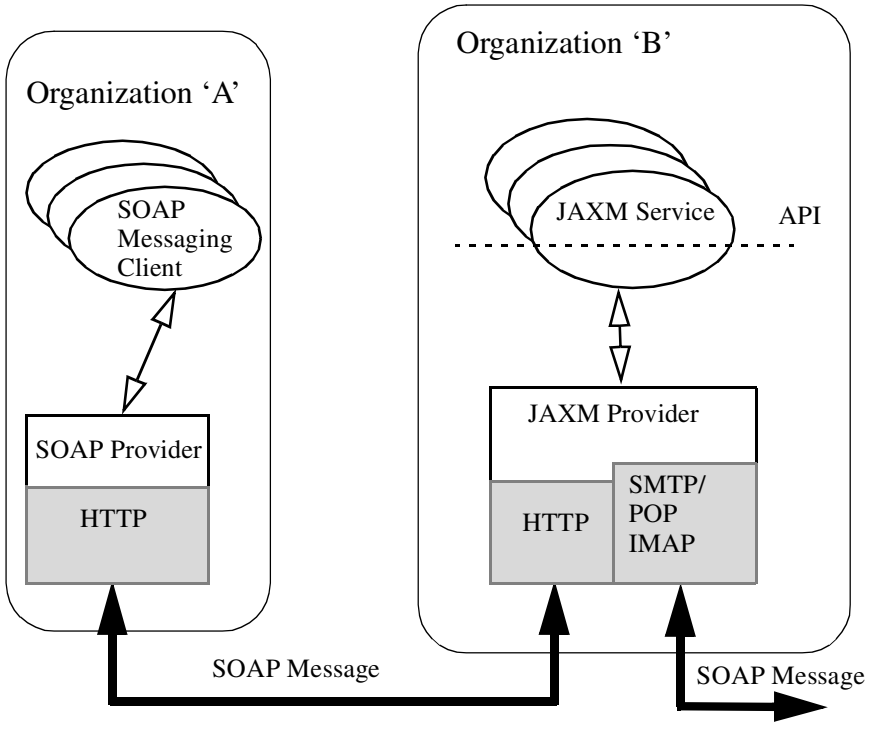


Figure 1. B2B Messaging Conceptual Model

JAXM is intended to be a lightweight messaging API for the development of XML-based business messaging applications. These applications appear primarily, though not exclusively, at the edge of organizations. The term *edge* is being used loosely to denote the set of applications that, collectively, deal with the production and consumption of **standard** business messages. The requirement for processing such messages is being fueled by the increasing need of organizations, irrespective of their size, to exchange business documents electronically. An application designed to consume specific business documents in an agreed-upon manner, and in response, to produce appropriate business documents, is informally referred to as a business service. Such services, when deployed in a Web Container are typically called Web Services. The formal specification of such services is outside the scope of this document.

Figure 1. makes a logical distinction between an application that uses the JAXM API for messaging ("JAXM client") and a messaging provider that is implemented to support the JAXM API ("JAXM provider"). The latter is responsible for the actual transmission and reception of SOAP messages.

JAXM.1.2 Scope

Message exchange scenarios based on JAXM are always document-centric, that is, they involve the exchange of XML documents. These exchanges, which are generally between business partners, fall into five broad categories. One type of message exchange is an update whose response is simply an acknowledgement that the update was received. Another type of exchange is an inquiry whose response is information of some sort, typically data requested in the original message. Both of these exchange scenarios may be either synchronous or asynchronous. When the exchange is synchronous, the sender of the update or inquiry waits until the response is received. When the exchange is asynchronous in nature, the response is sent in a separate operation at a later time. A fifth possible exchange scenario is the one where the sender simply sends a message and does not expect a reply.

Note that when a messaging provider is being used, all messages go through it. That is, when a JAXM client sends a message, the message first goes to the JAXM provider, which then handles the actual transmission of the message to its destination. When a JAXM client receives a message, the provider has actually received the message on the client's behalf and then forwarded the message to the client. For simplicity, the term *sender* is used here to refer collectively to a JAXM client/JAXM provider pairing in a message production role. Similarly, the term

receiver refers to a JAXM client/JAXM provider coupling in a message consumer role.

Note that it is also possible for a JAXM client to send messages without using a JAXM provider. In this case, the client is limited to sending synchronous messages to a specified URL. The advantage of not using a provider is simplicity and ease of use; the disadvantage is that the client does not get the flexibility or quality of service that a messaging provider can offer.

All implementations of JAXM must support the five message exchange scenarios described at the beginning of this section. These interaction styles are illustrated in Figure 2. through Figure 6.

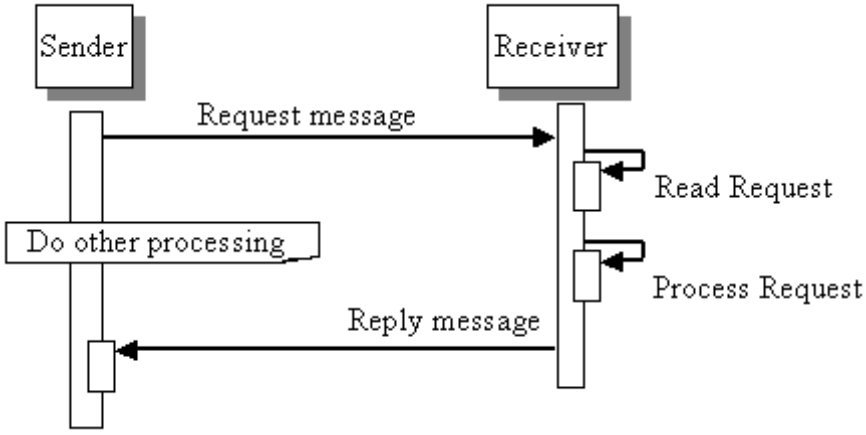


Figure 2. Asynchronous Inquiry

In the asynchronous inquiry scenario, the sender is assumed to send a message without needing to wait for a response. The receiver is required to read and process the request and generate an appropriate reply to the original request. Sending the reply is a totally separate operation, and the time interval between a request and a reply may be measured in days. JAXM implementations must therefore be able to support long-lived transactions.

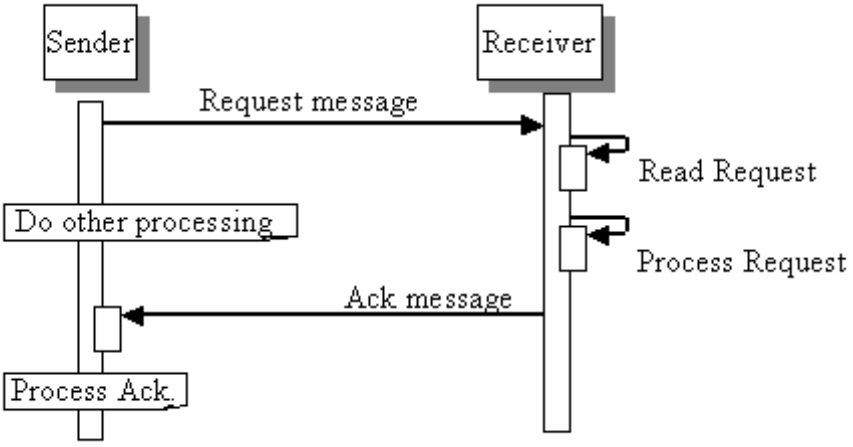


Figure 3. Asynchronous Update with Acknowledgement

Figure 3. depicts a scenario in which the reception of an acknowledgement message denotes the successful completion of an earlier request. An acknowledgement message must be correlated to the request message to which it refers. Note that JAXM does not specify how this is done.

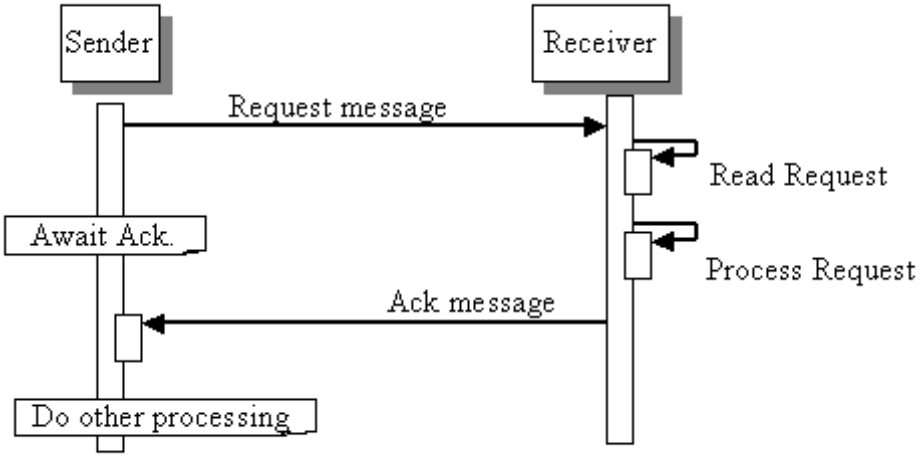


Figure 4. Synchronous Update

Figure 4. reflects a synchronous scenario, in which the sender either cannot or must not proceed until a response to the request being sent is received. A typical

response to an update is an acknowledgement message, which implies the successful completion of the request that was sent.

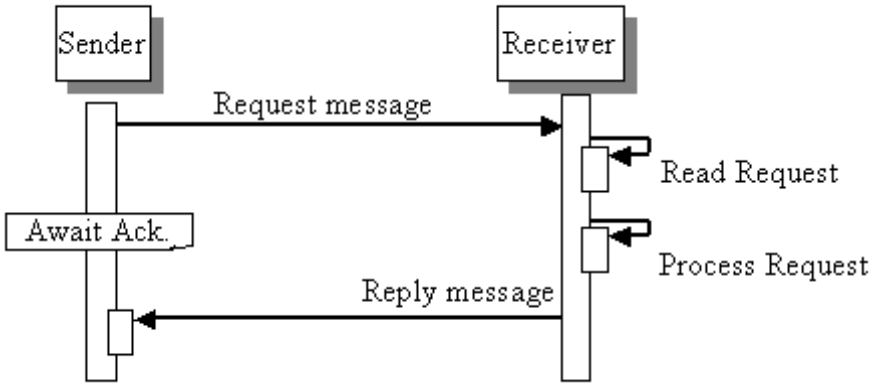


Figure 5. Synchronous Inquiry

The scenario in Figure 5. is a simple variation on the previous case. The sender waits for a reply message to the request that was sent. The distinction is that in this case, the reply message does not relate to the request but is instead a message whose only function is to unblock the calling application. This is in contrast to an acknowledgement message, which must identify the earlier message whose receipt it is acknowledging.

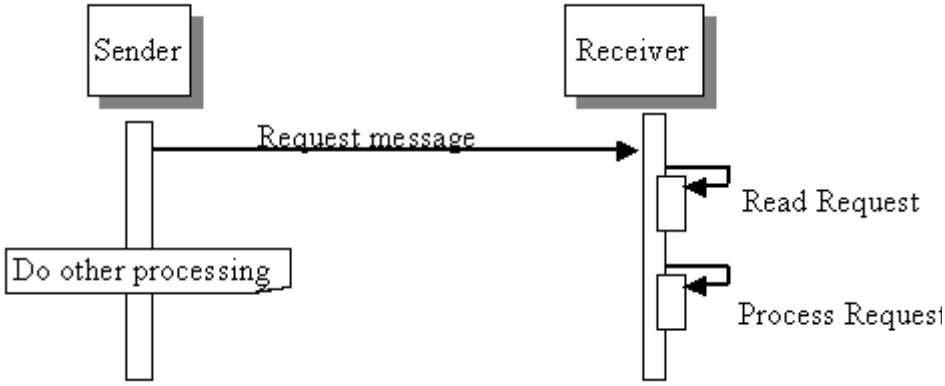


Figure 6. Fire and Forget

The case shown in Figure 6. implies that the sender is not expecting a response to the request message being sent.

JAXM may facilitate the automation of portions of an overall business process, but its use does not necessarily apply to the entire business process. The applicability of JAXM to the larger business system is a function of an overall business process model that is specific to a particular group of trading partners or vertical industries. This specification does not address the ways in which business objects are expressed in XML.

JAXM.1.3 Interoperability

An important notion presented in the "B2B Messaging Conceptual Model" on page 1 is that a JAXM client must be capable of interoperating with a peer business application, whether or not the other application uses JAXM. One of the key ingredients enabling standards-based interoperability is the widespread adoption of the following:

- a transport-neutral packaging model
- agreements on message header structures, manifests, and so on.

Although JAXM is heavily biased towards using industry standards, the only requirement placed on JAXM 1.1 providers is that they must support the SOAP1.1 and SOAP with Attachments specifications. In addition, JAXM providers may optionally choose to support higher level industry messaging protocols built on top of the SOAP standard. As stated earlier, a specific industry usage of SOAP is referred to here as a Profile. A JAXM Profile, therefore, represents a given industry or standards group's particular use of SOAP.

JAXM 1.1 providers must implement their transport bindings in accordance with the SOAP 1.1 specification and must therefore support SOAP 1.1 bindings for the HTTP protocol but may, in addition, choose to implement other standard networking protocols, such as FTP and SMTP(IMAP, POP). The points (within network topologies) at which JAXM providers produce and consume SOAP1.1 messages bound to HTTP are referred to as "JAXM interoperability points". The existence of such "points" establishes a basis or minimum standard for interoperasbility between JAXM and non-JAXM implementations of SOAP. In all cases, JAXM applications assume that SOAP messages are being transported by the communications infrastructure. The assured level of interoperability is therefore the SOAP Message and not specific transport bindings. In order for a JAXM Client (or Service) to interoperate with a JAXM or non-JAXM Service (or Client),

the parties must first agree on SOAP Transport bindings as well as messaging Profiles.

JAXM.1.4 SOAP Packaging Model

There are two packaging models for SOAP messages, one that includes attachments and one that does not. JAXM provides a standard way of both producing and consuming SOAP messages with or without attachments.

JAXM.1.4.1 SOAP Message with Attachments

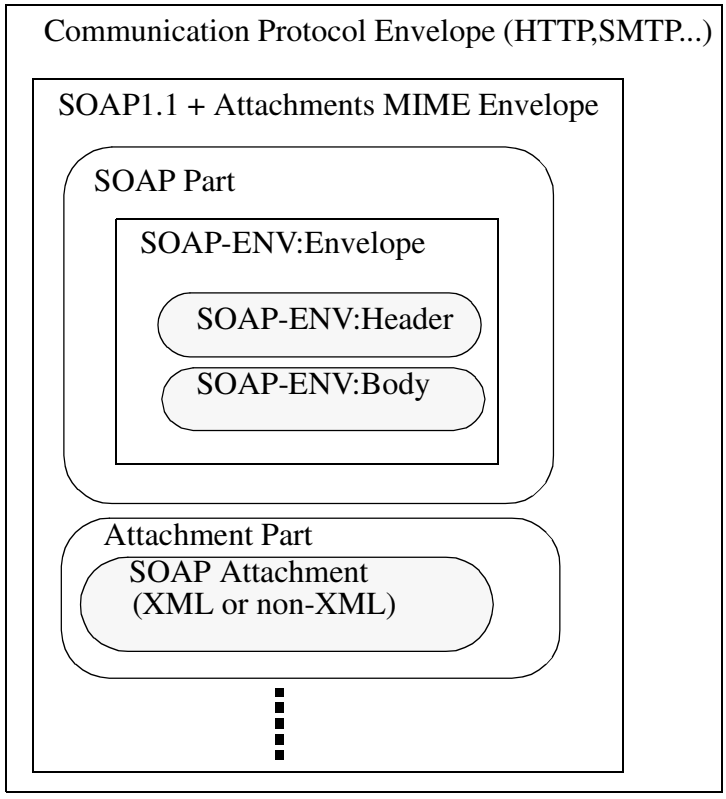


Figure 7. SOAP1.1 with Attachments

Figure 7. depicts the conceptual model of a JAXM message that includes one or more attachments. This message is aligned with the SOAP1.1 and SOAP with Attachments specifications, which all JAXM implementations must support.

A JAXM client may choose whether to create and/or consume SOAP attachments based on application-specific requirements. For instance, an acknowledgement message need not have an attachment part, but a message with content that is not in XML format must have an attachment part to contain the non-XML data. This is true because the attachment part of a message can contain any kind of content, from image files to plain text, whereas the SOAP part can contain only data in XML format.

Whether a JAXM client sends a message that contains an attachment part is up to the application developer; however, a JAXM client that receives a message is required to recognize the presence of any attachment part(s) and also to process them. How this processing is done is up to the application developer.

A message that contains one or more attachments must have a MIME envelope, which contains the SOAP part of a message and also the attachment part. A JAXM client does not have to do anything about this, however. When a client creates an `AttachmentPart` object, the MIME envelope is automatically created.

JAXM uses the JavaBeans™ Activation Framework (JAF) to support a flexible way of handling SOAP attachments based on the MIME types. Refer to “JavaBeanstm Activation Framework Version 1.0a” [see page 55] .

JAXM.1.4.2 SOAP Message without Attachments

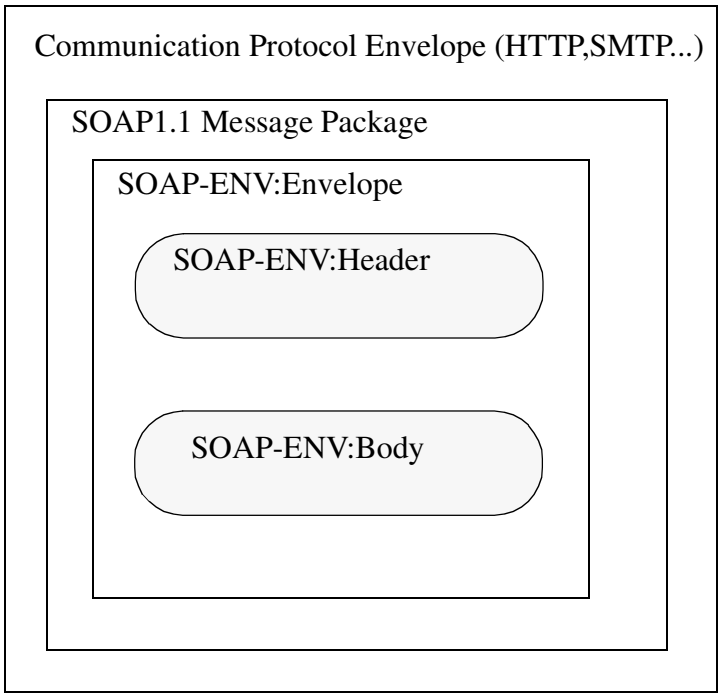


Figure 8. SOAP1.1 Packaging Model without Attachments

Figure 8. shows the packaging model for a message with no attachments. In addition to not having an attachment part, it also has no MIME envelope. Without attachments, the MIME Multipart/Related outer wrapper is redundant, and a JAXM implementation must not produce one.

JAXM.1.5 JAXM, JMS & JavaMail

The Java™ 2 Platform provides APIs for three distinct messaging infrastructures:

- Web Services (SOAP1.1 based messaging)
- Message Oriented Middleware (MOM)
- Mail

Although these infrastructures share similarities at the conceptual level, in reality they are sufficiently different that developers have required and gravitated towards APIs optimized for each infrastructure.

The underlying similarities between messaging infrastructures have, over time, resulted in hybrid use cases. For example, there are cases where mail messages are transported over MOM infrastructures as opposed to conventional e-mail backbones. These variations and cross-infrastructure use cases are typically not visible to individual developers and rarely influence their architectural choices. For example, a developer using the Java™ Message Service (JMS) typically requires a MOM infrastructure at both ends of a particular application. The end-to-end applications developed are typically conceived of as MOM applications and not as (say) Mail applications. The fact that a given MOM infrastructure may be transporting the messages over an e-mail backbone has no bearing on the application model developers choose to adopt.

Similarly, JAXM is an abstraction on an emerging Web-centric messaging infrastructure. In essence, JAXM is promoting a programming model for an infrastructure that fits somewhere between e-mail and MOM. The key notion is that a JAXM application may be written such that it is a SOAP-based Web service, is a client of a SOAP service, or is both. An application based on the JAXM API is therefore quite different from an application based on the JMS API, which in turn is distinct from an application based on the JavaMail™ API. Each API is equally valid and relevant with respect to, and in the context of, its associated infrastructure. Beyond some superficial similarities between these APIs, they differ primarily in the communication concepts represented by the semantics of their message headers. SOAP 1.1 is silent on the contents of the message header, but JAXM Profiles (for example, the Profile for ebXML Message Service) introduce the concept of 'separate' SOAP infrastructures.

Application developers are likely to choose JAXM in cases where they wish to communicate over SOAP infrastructures. The fact that SOAP has specified more than one transport binding (and indeed has not precluded bindings for MOM infrastructures) does not undermine the notion that the conceptual model for JAXM developers is one in which the application endpoints are in fact SOAP endpoints. Having said this, a given JAXM provider may choose to transport SOAP messages over a MOM infrastructure. This is an implementation detail that is completely invisible to the application, and it does not in any way transform a JAXM application into a MOM application.

In summary, JAXM is not intended to be a ‘grand unification’ API, and its role and relationship to a SOAP infrastructure is equivalent to the relationship that JMS has with MOM infrastructures. JAXM, JMS, and JavaMail are APIs focused on meeting the needs of their respective constituencies by utilizing the distinct characteristics of SOAP, MOM and Mail infrastructures, respectively.

CHAPTER JAXM.2

Infrastructure

As used here, the term *infrastructure* refers to the implementation-specific functionality required to support a JAXM implementation.

JAXM.2.1 JAXM Client

A JAXM client, which is essentially synonymous with an application, falls into two broad categories, one that uses a JAXM messaging provider and one that does not.

JAXM.2.1.1 JAXM Client Using a JAXM Provider

It is expected that in the majority of web messaging scenarios, a JAXM client will use a JAXM provider. In such a scenario, a JAXM client is deployed in a container, which means that it can send messages both synchronously and asynchronously. The container may be either a Java™ 2 Platform Enterprise Edition (J2EE™) Web Container or a J2EE Enterprise JavaBeans™ (EJB™) Container. The client maintains a connection with its JAXM provider, and all messages that the client sends or receives go through that messaging provider.

A JAXM messaging provider works behind the scenes, offering functionality such as message routing and reliable messaging. Different providers may vary widely in what they offer, and their specific capabilities fall outside the scope of this document. The services of a JAXM provider are part of the communications infrastructure, as opposed to being specific to an application. As a consequence, they are completely transparent to the JAXM client, which is unaware of the provider except when it establishes a connection with its provider. A JAXM client can get information about its JAXM provider via the `javax.xml.messaging.ProviderMetaData` interface.

JAXM.2.1.2 Standalone JAXM Client

A JAXM client may be a standalone Java™ 2 Platform Standard Edition (J2SE™) application, in which case it does not use a JAXM provider. A standalone client implements only the client view of the JAXM API and is considered to be a special case because, for whatever reason, it does not need to use the services of a JAXM messaging provider. A standalone client is assumed to be a client of a point-to-point synchronous web service offering only a request-response style of interaction. Such a client establishes a connection directly with the service (using a URL), sends its request, and is blocked until it gets the response.

JAXM.2.1.3 The Relationship between JAXM Clients

The relationship between JAXM clients (applications) is fundamentally peer-to-peer, which means that from a programming model perspective, a JAXM client may choose to play a client (application) role or a server (service) role. In addition, depending on a specific context or messaging choreography, a JAXM client is free to switch roles.

By way of example, suppose that three businesses, A, B, and C, are all JAXM clients. Business A sends a purchase order to B, and B sends a batch of purchase orders to C, which is a purchase order consolidation service. In this scenario, A is acting as a client when it sends its purchase order to B. B is acting in a server role with respect to A because it will fulfill the orders. B takes on a client role with respect to C when it sends its purchase orders to C to be consolidated. In other scenarios, A could be implementing a service requested by other businesses, thus putting it in the server role. And C could be functioning in a client role when it requests a service of another business. In general, then, a JAXM application assumes a client role when it requests a service, and assumes a server role when it performs a service.

JAXM.2.1.4 Client and Service Implementations

Another consideration for the developer of JAXM applications, in addition to whether they are acting in a client or server role, is whether messages are sent as one-way messages (asynchronously) or as request-response messages (synchronously).

The simplest case is that of the standalone JAXM application because of the limitations placed on it. Because asynchronous messaging requires a JAXM provider, a standalone JAXM application is limited to sending messages synchronously. Further, it is limited to acting in a client role. That is, a JAXM application

that has no JAXM provider and is not operating from within a container can act only as a client and can send its requests using only the request-response style of messaging.

An application that uses a JAXM provider and is deployed in a J2EE container, on the other hand, has much more flexibility with regard to the style of messaging it can use and the roles it can assume. It can use either or both kinds of messaging, and it can assume both client and server roles.

There are some requirements placed on a JAXM application operating in a service role, however. One requirement is that it must be capable of consuming all messages that conform to the SOAP 1.1 and SOAP with Attachments specifications. Thus, the service must be able to consume all standard SOAP messages, including those sent by non-JAXM clients. Another requirement is that a service must be implemented to handle one-way or request-response styles of messaging but not both. The implementation of point-to-point request-response messaging, which does not require a provider, ensures that a JAXM application in a service role is capable of receiving requests from a standalone JAXM application.

When a JAXM application is deployed in a J2EE Web Container, the SOAP1.1 protocol must be bound to HTTP. However, the way in which a JAXM application communicates with a JAXM provider is considered to be a private implementation detail. A JAXM application is not required to be co-located with its provider, nor is it required to be in a different virtual machine.

JAXM.2.2 Error Messages

JAXM implementations must adopt a best effort strategy for ensuring the validity of messages produced and sent to peer entities. JAXM providers, on receiving a malformed message, are responsible for producing an appropriate error message and sending it to the offending peer entity. The structure and addressing of inter-provider error messages is Profile-specific. JAXM does not make a distinction between error messages and any other Profile-specific message. Given that JAXM providers are “Profile-aware”, they may choose to map an error condition onto a Profile-specific error message. Such messages would be delivered to a client in the same manner as any other message. It is the responsibility of a JAXM application to consume error messages and take application-specific corrective action.

JAXM.2.3 Messaging Profiles

As stated previously, JAXM implementations must support the SOAP1.1 and SOAP with Attachments specifications. However, these specifications provide only a very basic packaging model and offer no specific addressing scheme or message structure for the routing of messages between peer entities. These may be supplied by a Profile that operates on top of SOAP.

By way of example, a specific Profile may stipulate a specific usage of a SOAP header. JAXM does not specify what specific XML content must be placed in the SOAP header, body, or attachments. Most enterprise-grade usages of SOAP messaging will typically specify critical information regarding the sender, recipient, message ID, and correlation information. (Correlation information identifies the previously sent request for which the current message is a response.)

JAXM implementations may choose to support a number of industry standard messaging Profiles. Profiles are identified by a name that uniquely identifies a particular industry/standards body's usage of SOAP messaging. A JAXM client must use the URI of the schema associated with a given Profile as the Profile name. For example, the Profile for ebXML Message Service Specification (MS) could be identified by the following URI:

```
http://www.ebxml.org/project_teams/transport/messageHeader.xsd
```

Developers are required to specify, either at run time or at deployment time, critical system-level information necessary to correctly route, deliver and correlate messages. The way in which this information is mapped on to a given message depends on the Profile being used. JAXM makes no assumptions about where this information, if present, is stored within a message. An explicit contract must therefore exist between a JAXM client and its JAXM provider. A Profile string is used to establish this contract at run time. In order for JAXM applications to be able to exchange business messages with peer entities, they must have an agreement to use the same Profile. The way in which such agreements can be established is outside the scope of this document.

By way of example, an ebXML MS Profile clearly specifies how a SOAP header should be populated with necessary addressing and message identification information. A JAXM application, when using such a Profile, is responsible for constructing a SOAP header as per the specifications associated with this Profile. All providers supporting the same Profile (identified by a Profile name) will therefore have a common understanding of the message structure and message semantics. Note that a JAXM application may specify a Profile name when it creates a `MessageFactory` object. `Message` objects produced by such a `MessageFactory`

object will be specific to the named Profile. In addition, JAXM implementations may choose to pre-populate a `Message` object with critical information, such as the sender and recipient, in a Profile-specific manner.

If an application chooses not to specify a standard Profile, the JAXM provider must default to using an application-specific (that is, private) Profile. In such cases, developers cannot be assured of any level of interoperability based on public standards. It is conceivable that a given provider may support multiple application-specific Profiles.

JAXM.2.4 JAXM Deployment

JAXM applications may be deployed in Servlet 2.2 and/or J2EE™1.3 containers. It is anticipated that future versions of the J2EE specification will include JAXM-specific deployment information.

Standalone JAXM applications, which implement only a request-response style of messaging, are considered to be J2SE applications. No new deployment requirements will be introduced for such applications.

JAXM.2.5 OnewayListener

JAXM promotes a standard way of delivering messages asynchronously to JAXM clients, namely through a message listener interface. In the case of EJB containers, the `OnewayListener` interface may be implemented using J2EE Message Driven Beans (MDB). In the case of Servlet containers, JAXM applications may extend the `JAXMServlet` interface and implement the `onMessage` method.

As mentioned previously, the provider-to-client contract is based on the SOAP1.1 and SOAP with Attachments specifications. In other words, irrespective of where a JAXM application is deployed, a SOAP message enveloping scheme must be used. Furthermore, there is a clear assumption that when a JAXM client is deployed in a Servlet container, the asynchronous activation model is built on SOAP1.1 (with attachments) bound to HTTP.

JAXM.2.6 ReqRespListener

The `ReqRespListener` interface is intended to enable the development of request-response style JAXM services. JAXM services implementing this interface typically do not require a provider. Their SOAP messages must be bound to HTTP.

JAXM.2.7 Message Security

JAXM introduces no new security requirements. Messages are assumed to have both a transitory as well as a persistent confidentiality requirement. Support for security features and capabilities assuring confidentiality while messages are in transit are implementation details of the JAXM provider. Although HTTP is the transport of choice, support for protocols such as SSL may be appropriate and adequate. In the case of SMTP infrastructures, JAXM providers may choose to use PGP and or S/MIME.

JAXM provides no specific interfaces to digital signatures that span an entire message. The assumption is that developers will have access to "user" level portions of a message—where "user" level is defined as the application-specific parts of a message. Note that the signing or encrypting of the SOAP header in a manner that would prevent the message from being interpreted and therefore correctly routed will raise a JAXM exception. A developer may require some application-specific (and therefore potentially nonstandard) encryption algorithms and/or security functions to be applied to predefined portions of a message. In such circumstances, developers must select an appropriate Profile known to the JAXM provider.

Developers may choose to use digital signature technologies to sign application-level XML fragments as they see fit. As mentioned earlier, the application of specific signing technology must not interfere with the routing of messages by the JAXM infrastructure.

The authentication of JAXM clients to JAXM providers is considered to be an implementation detail and beyond the scope of this specification.

CHAPTER JAXM.3

Package Overview

This chapter presents an overview of the JAXM API, which consists of a single package:

- `javax.xml.messaging`

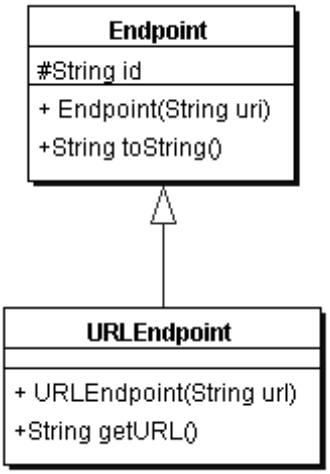
Note that this specification depends on the `javax.xml.soap` package and hence requires support for the SAAJ1.1 specifcation.

The following sections give an overview of the functionality in this package and also present some code samples to illustrate how the packages are used. Note that the examples are not exhaustive and should not be considered a normative part of this specification. The intent is to provide an overview of the packages, the details of which are provided in the API documentation included later in this document and the SAAJ1.1 specification.

JAXM.3.1 javax.xml.messaging Package

The `javax.xml.messaging` package provides a higher level abstraction than the `javax.xml.soap` package offers on its own. JAXM clients intending to support asynchronous one-way messaging must do so by using an implementation of the `messaging` package, which includes the appropriate connection, listener, and endpoint objects.

JAXM.3.1.1 Endpoint & URLEndpoint

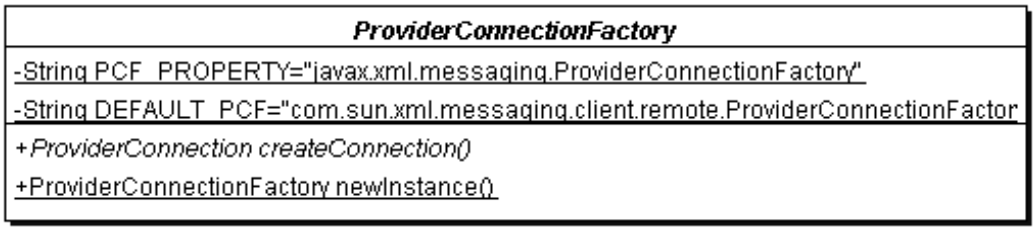


In most messaging applications, messages are typically addressed such that the recipient and originator of the message are conveyed as part of the message itself. The notion of self-addressed messages is fundamental to true messaging since messages can be routed in a manner that is independent of the underlying communication infrastructure and network topology. In this sense, the notion of SOAP messaging is somewhat of a misnomer. The SOAP1.1 specifications allow for messages to contain the necessary addressing information but do not define a standard way for this information to be represented within a message. JAXM therefore relies on Profiles, that is, industry standard usages of SOAP, for interoperable addressing conventions. Given that JAXM supports both one-way and request-response forms of messaging, the `Endpoint` object has been modeled such that it must be specified explicitly on the `call` method of the `SOAPConnection` class. By contrast, the `send` method of the `ProviderConnection` class does not allow for the specification of an `Endpoint` object because there is an assumption that one-way messaging is inherently asynchronous, thus requiring self-addressed (that is, Profiled) messages.

The `URLEndpoint` object is a direct subclass of an `Endpoint` class and is intended to support a special but common use-case. JAXM clients wishing to contact a SOAP-based service in a point-to-point, request-response (that is, synchronous) manner, may choose to do so without utilizing the services of a JAXM provider. A `URLEndpoint` object contains a URL that is passed to the `call` method to send a message to a SOAP service and block while waiting for a response from the service.

Note that JAXM 1.1 requires that SAAJ1.1 implementations fully support the `call` method of the `SOAPConnection` class and hence the `newInstance` method of the `SOAPConnectionFactory` must not throw an `UnsupportedOperationException`.

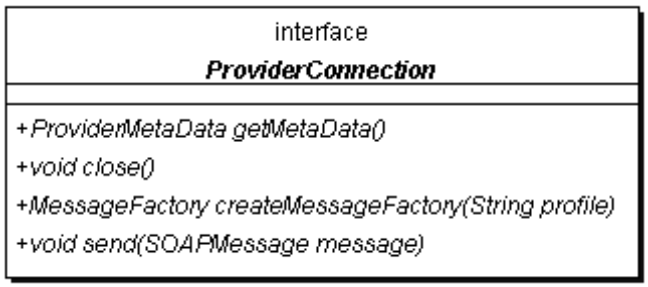
JAXM.3.1.2 ProviderConnection & Factory



The `ProviderConnection` and associated `ProviderConnectionFactory` objects must be used when a JAXM application requires one-way (asynchronous) messaging semantics.

The `ProviderConnectionFactory` object is an administered object created by the container (a Servlet or Enterprise JavaBeans™ container) on which the application has been deployed. The information necessary to set up a `ProviderConnectionFactory` is specified at deployment time. The connections created by the factory will be to a particular messaging provider. Factory objects are registered with a naming service such as one based on Java Naming and Directory Interface™ (JNDI) technology and are retrieved from that naming service in order to create a `ProviderConnection` object.

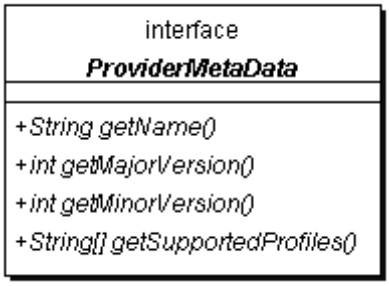
JAXM clients that choose to use the services of a messaging provider must create a `ProviderConnection` object that connects the application to the provider. In order to do this, the application must look up the relevant `ProviderConnectionFactory` object using a pre configured logical name. The returned factory object can subsequently be used to create the connection to the specific messaging provider. In cases where JNDI support is not available, JAXM applications may create a default `ProviderConnectionFactory` object by using the `newInstance()` method. This latter model is intended for a relatively small set of use-cases and is the less preferred approach as compared with the JNDI look-up pattern.



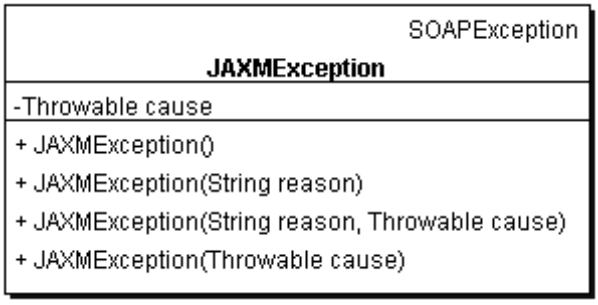
Once a JAXM client has established a connection to its messaging provider, it must then create profile-specific `MessageFactory` objects. These objects can then be used to cre-

ate SOAPMessage objects. Note that SOAP messages must be "populated" in a profile-specific and application-specific manner. Once a SOAPMessage object has been created and populated, it can be sent asynchronously by calling the javax.xml.messaging.ProviderConnection.send method. The message can also be sent synchronously by calling the method javax.xml.soap.SOAPConnection.call.

JAXM.3.1.3 ProviderMetaData & JAXMException

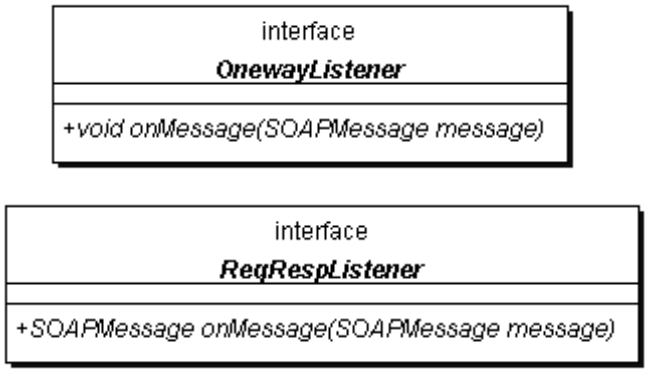


Provider-specific information can be obtained using an instance of a ProviderMetaData object created using the getMetaData method of the ProviderConnection object.



A JAXMException object is a subclass of SOAPException and may contain a String denoting the reason for the exception, an embedded Throwable object, or both.

JAXM.3.1.4 Oneway and Request-Response Listeners



A JAXM service must implement either the `OnewayListener` or the `ReqRespListener` marker interface. The choice of which to implement is application-specific. The `OnewayListener` interface is intended for services that are implemented to use a one-way, that is, asynchronous, style of messaging. Similarly, `ReqRespListener` is intended to be used by developers wishing to implement a service that uses a request-response messaging style, that is, one that is synchronous. Both interfaces contain an `onMessage` method that must be implemented to define how messages that are received are to be handled. Irrespective of the type of service being implemented, the `onMessage` method must be called by code that is specific to the container hosting the service. Thus, although the service supplies the implementation of the `onMessage` method, it is container code that invokes it.

The `JAXMServlet` class is a utility class, and there is no requirement that it be implemented or extended. However, an implementation of a JAXM service that is deployed in a Servlet container will need to implement similar functionality in order to support the required behavior for the two types of listener interfaces. Developers who choose simply to extend `JAXMServlet` must avoid overriding the `doPost` method. If they choose to do so, they must faithfully implement the JAXM client contract.

JAXM.3.2 A simple Message Producer example

```
/*
* $Id: SendingServlet.java,v 1.3 2001/09/17 21:01:27 akv Exp $
* $Revision: 1.3 $
* $Date: 2001/09/17 21:01:27 $
*
* Copyright 2000-2001 by Sun Microsystems, Inc.,
* 901 San Antonio Road, Palo Alto, California, 94303, U.S.A.
* All rights reserved.
*
* This software is the confidential and proprietary information
* of Sun Microsystems, Inc. ("Confidential Information"). You
* shall not disclose such Confidential Information and shall use
* it only in accordance with the terms of the license agreement
* you entered into with Sun.
*/
package simple.sender;
import java.net.*;
import java.io.*;
import java.util.*;
import javax.servlet.http.*;
import javax.servlet.*;
import javax.xml.messaging.*;
import javax.xml.soap.*;
import javax.activation.*;
import javax.naming.*;


/**
* Sample servlet that is used for sending the message.
*
* @author Rajiv Mordani (rajiv.mordani@sun.com)
* @author Anil Vijendran (anil@sun.com)
*/
public class SendingServlet extends HttpServlet {

    String to = null;
    String data = null;
    ServletContext servletContext;

    // Connection to send messages.
    private SOAPConnection con;
```

```
public void init(ServletConfig servletConfig) throws
                                    ServletException {
    super.init(servletConfig);
    servletContext = servletConfig.getServletContext();

    try {
        SOAPConnectionFactory scf =
                            SOAPConnectionFactory.newInstance();
        con = scf.createConnection();
    } catch(Exception e) {
        System.err.println("Unable to open a SOAPConnection", e);
    }
    InputStream in
    = servletContext.getResourceAsStream("/WEB-INF/
                  address.properties");
    if (in != null) {
        Properties props = new Properties();
        try {
            props.load(in);
            to = props.getProperty("to");
            data = props.getProperty("data");
        } catch (IOException ex) {
            // Ignore
        }
    }
}

public void doGet(HttpServletRequest req,
                  HttpServletResponse resp)
                                      throws ServletException {
    String retval ="<html> <H4>";
    try {
        // Create a message factory.
        MessageFactory mf = MessageFactory.newInstance();

        // Create a message from the message factory.
        SOAPMessage msg = mf.createMessage();

        // Message creation takes care of creating the SOAPPart- a
        // required part of the message as per the SOAP 1.1
        // specification.
        SOAPPart sp = msg.getSOAPPart();

        // Retrieve the envelope from the soap part to start
```

```
// building the soap message.
SOAPEnvelope envelope = sp.getEnvelope();

// Create a soap header from the envelope.
SOAPHeader hdr = envelope.getHeader();

// Create a soap body from the envelope.
SOAPBody bdy = envelope.getBody();

// Add a soap body element to the soap body
SOAPBodyElement gltp = bdy.addBodyElement(
        envelope.createName("GetLastTradePrice",
                  "ztrade",
                  "http://wombat.ztrade.com"));
gltp.addChildElement(
    envelope.createName("symbol",
    "ztrade",
    "http://wombat.ztrade.com")).addTextNode("SUNW");
StringBuffer urlSB=new StringBuffer();
urlSB.append(req.getScheme()).append("://")
                  .append(req.getServerName());
urlSB.append( ":" ).append(req.getServerPort()).append(
    req.getContextPath());
String reqBase=urlSB.toString();
if(data==null) {
    data=reqBase + "/index.html";
}
// Want to set an attachment from the following url.
// Get context
URL url = new URL(data);
AttachmentPart ap =
    msg.createAttachmentPart(new DataHandler(url));
ap.setContentType("text/html");

// Add the attachment part to the message.
msg.addAttachmentPart(ap);

// Create an endpoint for the recipient of the message.
if(to==null) {
    to=reqBase + "/receiver";
}
URLEndpoint urlEndpoint = new URLEndpoint(to);
System.err.println("Sending message to URL:"
    +urlEndpoint.getURL());
```

```
            System.err.println(
                "Sent message is logged in\"sent.msg\"");
            retval += " Sent message (check \"sent.msg\") and ";
            FileOutputStream sentFile = new
                FileOutputStream("sent.msg");
            msg.writeTo(sentFile);
            sentFile.close();
            // Send the message to the provider using the connection.
            SOAPMessage reply = con.call(msg, urlEndpoint);
            if (reply != null) {
                FileOutputStream replyFile = new
                    FileOutputStream("reply.msg");
                reply.writeTo(replyFile);
                replyFile.close();
                System.err.println(
                    "Reply logged in \"reply.msg\"");
                retval +=
                " received reply (check \"reply.msg\"). </H4> </
html>";
            } else {
                System.err.println("No reply");
                retval += " no reply was received. </H4> </html>";
            }
        } catch(Throwable e) {
            e.printStackTrace();
            System.err.println("Error in constructing or sending
message "
                        +e.getMessage());
            retval += " There was an error " +
            "in constructing or sending message. </H4> </html>";
        }
        try {
            OutputStream os = resp.getOutputStream();
            os.write(retval.getBytes());
            os.flush();
            os.close();
        } catch (IOException e) {
            e.printStackTrace();
            System.err.println( "Error in outputting servlet
            response"+ e.getMessage());
        }
    }
}
```

JAXM.3.3 A simple Message Consumer example

```
/*
* $Id: ReceivingServlet.java,v 1.2 2001/09/17 21:01:27 akv Exp $
* $Revision: 1.2 $
* $Date: 2001/09/17 21:01:27 $
*
* Copyright 2000-2001 by Sun Microsystems, Inc.,
* 901 San Antonio Road, Palo Alto, California, 94303, U.S.A.
* All rights reserved.
*
* This software is the confidential and proprietary information
* of Sun Microsystems, Inc. ("Confidential Information"). You
* shall not disclose such Confidential Information and shall use
* it only in accordance with the terms of the license agreement
* you entered into with Sun.
*/
package simple.receiver;
import javax.xml.messaging.*;
import javax.xml.soap.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.xml.transform.*;
import javax.naming.*;

/**
* Sample servlet that receives messages.
*
* @author Rajiv Mordani (mode@eng.sun.com)
* @author Anil Vijendran (anil@sun.com)
*/
public class ReceivingServlet
                 extends JAXMServlet
                 implements ReqRespListener {

    static MessageFactory fac = null;
    static {
        try {
            fac = MessageFactory.newInstance();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    };
```

```
    public void init(ServletConfig servletConfig)
    throws ServletException {
        super.init(servletConfig);
        // Not much there to do here.
    }
    // This is the application code for handling the message..
    // Once the message is received the application can retrieve
    // the soap part, the attachment part if there are any, or
    // any other information from the message.

    public SOAPMessage onMessage(SOAPMessage message) {
        System.out.println("On message called in receiving servlet");
        try {
            System.out.println("Here's the message: ");
            message.writeTo(System.out);
            SOAPMessage msg = fac.createMessage();
            SOAPEnvelope env = msg.getSOAPPart().getEnvelope();
            env.getBody().addChildElement(
                env.createName("Response"))
                .addTextNode("This is a response");
            return msg;
        } catch(Exception e) {
            System.err.println(
                "Error in processing or replying to a message", e);
                return null;
        }
    }
}
```

CHAPTER JAXM.4

Package javax.xml.messaging

This package specifies the API for using a messaging provider to send and receive SOAP1.1 messages. A client using Java™ API for XML Messaging technology ("JAXM client") makes its connections to a messaging provider, which means that all messages it sends or receives go through the provider. The messaging provider is responsible for the delivery of messages and performing many functions behind the scenes.

Class Summary

Interfaces

`OnewayListener` 41	A marker interface for components (for example, servlets) that are intended to be consumers of one-way (asynchronous) JAXM messages.
`ProviderConnection` 42	A client's active connection to its messaging provider.
`ProviderMetaData` 48	Information about the messaging provider to which a client has a connection.
`ReqRespListener` 50	A marker interface for components that are intended to be consumers of request-response messages.

Classes

`Endpoint` 33	An opaque representation of an application endpoint.
`JAXMServlet` 37	The superclass for components that live in a servlet container that receives JAXM messages.
`ProviderConnectionFactory` 45	A factory for creating connections to a particular messaging provider.
`URLEndpoint` 52	A special case of the `Endpoint` class used for simple applications that want to communicate directly with another SOAP-based application in a point-to-point fashion instead of going through a messaging provider.

Exceptions

`JAXMException` 35	An exception that signals that a JAXM exception has occurred.

With a messaging provider, it is possible to send a message to multiple destinations, and a messaging provider can be configured to do the following:

- maintain a list of endpoints to which messages will be sent
- send, and if necessary, resend a message until the message is delivered successfully or until the specified

limit for retries is reached
- log messages in a specified directory

In addition, a messaging provider can make it possible for a protocol such as ebXML or SOAP RP to operate on top of SOAP, extending the Quality of Service available to JAXM messages.

The API in the `javax.xml.messaging` package makes it possible to do one-way messaging. One-way messaging allows the client to send a message and immediately go on to other work because the response, if there is one, will be sent as a separate operation at some time in the future.

The `javax.xml.messaging` package requires the `javax.xml.soap` package, which provides the API for constructing SOAP messages and retreiving their content. The `javax.xml.soap` package is defined in the *SOAP with Attachments API for Java*™ *(SAAJ) 1.1* specification.

javax.xml.messaging
JAXM.4.1 Endpoint

Declaration

public class **Endpoint**

```
java.lang.Object
  |
  +--javax.xml.messaging.Endpoint
```

Direct Known Subclasses: URLEndpoint[52]

Description

An opaque representation of an application endpoint. Typically, an Endpoint object represents a business entity, but it may represent a party of any sort. Conceptually, an Endpoint object is the mapping of a logical name (example, a URI) to a physical location, such as a URL.

For messaging using a provider that supports profiles, an application does not need to specify an endpoint when it sends a message because destination information will be contained in the profile-specific header. However, for point-to-point plain SOAP messaging, an application must supply an Endpoint object to the SOAPConnection method call to indicate the intended destination for the message. The subclass URLEndpoint[52] can be used when an application wants to send a message directly to a remote party without using a messaging provider.

The default identification for an Endpoint object is a URI. This defines what JAXM messaging providers need to support at minimum for identification of destinations. A messaging provider needs to be configured using a deployment-specific mechanism with mappings from an endpoint to the physical details of that endpoint.

Endpoint objects can be created using the constructor, or they can be looked up in a naming service. The latter is more flexible because logical identifiers or even other naming schemes (such as DUNS numbers) can be bound and rebound to specific URIs.

Member Summary

Fields

protected	id[34] A string that identifies the party that this Endpoint object represents; a URI is the default.

Constructors

public	Endpoint(String)[34] Constructs an Endpoint object using the given string identifier.

Methods

public java.lang.String	toString()[34] Retrieves a string representation of this Endpoint object.

Inherited Member Summary
Methods inherited from class java.lang.Object `clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait`

Fields

4.1.1 id

protected java.lang.String id

A string that identifies the party that this `Endpoint` object represents; a URI is the default.

Constructors

4.1.2 Endpoint(String)

public Endpoint(java.lang.String uri)

Constructs an `Endpoint` object using the given string identifier.

Parameters:

`uri` - a string that identifies the party that this `Endpoint` object represents; the default is a URI

Methods

4.1.3 toString()

public java.lang.String toString()

Retrieves a string representation of this `Endpoint` object. This string is likely to be provider-specific, and programmers are discouraged from parsing and programmatically interpreting the contents of this string.

Overrides: java.lang.Object.toString() in class java.lang.Object

Returns: a `String` with a provider-specific representation of this `Endpoint` object

javax.xml.messaging

JAXM.4.2 JAXMException

Declaration

public class **JAXMException** extends **SOAPException**[114]

```
java.lang.Object
  |
  +--java.lang.Throwable
        |
        +--java.lang.Exception
              |
              +--javax.xml.soap.SOAPException[114]
                    |
                    +--javax.xml.messaging.JAXMException
```

All Implemented Interfaces: java.io.Serializable

Description

An exception that signals that a JAXM exception has occurred. A `JAXMException` object may contain a `String` that gives the reason for the exception, an embedded `Throwable` object, or both. This class provides methods for retrieving reason messages and for retrieving the embedded `Throwable` object.

Typical reasons for throwing a `JAXMException` object are problems such as not being able to send a message and not being able to get a connection with the provider. Reasons for embedding a `Throwable` object include problems such as an input/output errors or a parsing problem, such as an error parsing a header.

Member Summary

Constructors

public	`JAXMException()`[36] Constructs a `JAXMException` object with no reason or embedded `Throwable` object.
public	`JAXMException(String)`[36] Constructs a `JAXMException` object with the given `String` as the reason for the exception being thrown.
public	`JAXMException(String, Throwable)`[36] Constructs a `JAXMException` object with the given `String` as the reason for the exception being thrown and the given `Throwable` object as an embedded exception.
public	`JAXMException(Throwable)`[36] Constructs a `JAXMException` object initialized with the given `Throwable` object.

Inherited Member Summary

Methods inherited from class java.lang.Object

Inherited Member Summary
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait
Methods inherited from interface SOAPException[114]
getCause()[116], getMessage()[116], initCause(Throwable)[116]
Methods inherited from class java.lang.Throwable
fillInStackTrace, getLocalizedMessage, printStackTrace, printStackTrace, printStackTrace, toString

Constructors

4.2.1 JAXMException()

public JAXMException()

Constructs a JAXMException object with no reason or embedded Throwable object.

4.2.2 JAXMException(String)

public JAXMException(java.lang.String reason)

Constructs a JAXMException object with the given String as the reason for the exception being thrown.

Parameters:

reason - a String giving a description of what caused this exception

4.2.3 JAXMException(String, Throwable)

public JAXMException(java.lang.String reason, java.lang.Throwable cause)

Constructs a JAXMException object with the given String as the reason for the exception being thrown and the given Throwable object as an embedded exception.

Parameters:

reason - a String giving a description of what caused this exception

cause - a Throwable object that is to be embedded in this JAXMException object

4.2.4 JAXMException(Throwable)

public JAXMException(java.lang.Throwable cause)

Constructs a JAXMException object initialized with the given Throwable object.

Parameters:

cause - a Throwable object that is to be embedded in this JAXMException object

javax.xml.messaging
JAXM.4.3 JAXMServlet

Declaration

public abstract class `JAXMServlet` extends javax.servlet.http.HttpServlet

```
java.lang.Object
  |
  +--javax.servlet.GenericServlet
        |
        +--javax.servlet.http.HttpServlet
              |
              +--javax.xml.messaging.JAXMServlet
```

All Implemented Interfaces: java.io.Serializable, javax.servlet.Servlet, javax.servlet.ServletConfig

Description

The superclass for components that live in a servlet container that receives JAXM messages. A `JAXMServlet` object is notified of a message's arrival using the HTTP-SOAP binding.

The `JAXMServlet` class is a support/utility class and is provided purely as a convenience. It is not a mandatory component, and there is no requirement that it be implemented or extended.

Note that when a component that receives messages extends `JAXMServlet`, it also needs to implement either a `ReqRespListener` object or a `OnewayListener` object, depending on whether the component is written for a request-response style of interaction or for a one-way (asynchronous) style of interaction.

Member Summary

Fields

protected	`msgFactory`[38] The `MessageFactory` object that will be used internally to create the `SOAPMessage` object to be passed to the method `onMessage`.

Constructors

public	`JAXMServlet()`[39]

Methods

public void	`doPost(HttpServletRequest, HttpServletResponse)`[39] Internalizes the given `HttpServletRequest` object and writes the reply to the given `HttpServletResponse` object.
protected static javax.xml.soap.MimeHeaders	`getHeaders(HttpServletRequest)`[39] Returns a `MimeHeaders` object that contains the headers in the given `HttpServletRequest` object.
public void	`init(ServletConfig)`[39] Initializes this `JAXMServlet` object using the given `ServletConfig` object and initializing the `msgFactory` field with a default `MessageFactory` object.
protected static void	`putHeaders(MimeHeaders, HttpServletResponse)`[40] Sets the given `HttpServletResponse` object with the headers in the given `MimeHeaders` object.

Member Summary	
public void	setMessageFactory(MessageFactory)[40] Sets this JAXMServlet object's msgFactory field with the given MessageFactory object.

Inherited Member Summary
Methods inherited from class javax.servlet.GenericServlet destroy, getInitParameter, getInitParameterNames, getServletConfig, getServletContext, getServletInfo, getServletName, init, log, log
Methods inherited from class javax.servlet.http.HttpServlet doDelete, doGet, doOptions, doPut, doTrace, getLastModified, service, service
Methods inherited from class java.lang.Object clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Fields

4.3.1 msgFactory

protected MessageFactory[78] msgFactory

The MessageFactory object that will be used internally to create the SOAPMessage object to be passed to the method onMessage. This new message will contain the data from the message that was posted to the servlet. Using the MessageFactory object that is the value for this field to create the new message ensures that the correct profile is used.

Constructors

4.3.2 JAXMServlet()

public JAXMServlet()

Methods

4.3.3 doPost(HttpServletRequest, HttpServletResponse)

public void doPost(javax.servlet.http.HttpServletRequest req,
javax.servlet.http.HttpServletResponse resp)
throws ServletException, IOException

Internalizes the given `HttpServletRequest` object and writes the reply to the given `HttpServletResponse` object.

Note that the value for the `msgFactory` field will be used to internalize the message. This ensures that the message factory for the correct profile is used.

Overrides: javax.servlet.http.HttpServlet.doPost(javax.servlet.http.HttpServletRequest, javax.servlet.http.HttpServletResponse) in class javax.servlet.http.HttpServlet

Parameters:

`req` - the `HttpServletRequest` object containing the message that was sent to the servlet

`resp` - the `HttpServletResponse` object to which the response to the message will be written

Throws:

`ServletException` - if there is a servlet error

`IOException` - if there is an input or output error

4.3.4 getHeaders(HttpServletRequest)

protected static `MimeHeaders`[83] getHeaders(javax.servlet.http.HttpServletRequest req)

Returns a `MimeHeaders` object that contains the headers in the given `HttpServletRequest` object.

Parameters:

`req` - the `HttpServletRequest` object that a messaging provider sent to the servlet

Returns: a new `MimeHeaders` object containing the headers in the message sent to the servlet

4.3.5 init(ServletConfig)

public void init(javax.servlet.ServletConfig servletConfig)
throws ServletException

Initializes this `JAXMServlet` object using the given `ServletConfig` object and initializing the `msgFactory` field with a default `MessageFactory` object.

Overrides: javax.servlet.GenericServlet.init(javax.servlet.ServletConfig) in class javax.servlet.GenericServlet

Parameters:

`servletConfig` - the `ServletConfig` object to be used in initializing this `JAXMServlet` object

Throws:

`ServletException`

4.3.6 putHeaders(MimeHeaders, HttpServletResponse)

protected static void putHeaders(`MimeHeaders`[83] headers, javax.servlet.http.HttpServletResponse res)

Sets the given `HttpServletResponse` object with the headers in the given `MimeHeaders` object.

Parameters:

`headers` - the `MimeHeaders` object containing the the headers in the message sent to the servlet

`res` - the `HttpServletResponse` object to which the headers are to be written

See Also: `getHeaders(HttpServletRequest)`[39]

4.3.7 setMessageFactory(MessageFactory)

public void setMessageFactory(`MessageFactory`[78] msgFactory)

Sets this `JAXMServlet` object's `msgFactory` field with the given `MessageFactory` object. A `MessageFactory` object for a particular profile needs to be set before a message is received in order for the message to be successfully internalized.

Parameters:

`msgFactory` - the `MessageFactory` object that will be used to create the `SOAPMessage` object that will be used to internalize the message that was posted to the servlet

javax.xml.messaging

JAXM.4.4 OnewayListener

Declaration

public interface `OnewayListener`

Description

A marker interface for components (for example, servlets) that are intended to be consumers of one-way (asynchronous) JAXM messages. The receiver of a one-way message is sent the message in one operation, and it sends the response in another separate operation. The time interval between the receipt of a one-way message and the sending of the response may be measured in fractions of seconds or days.

The implementation of the `onMessage` method defines how the receiver responds to the `SOAPMessage` object that was passed to the `onMessage` method.

See Also: `JAXMServlet`[37], `ReqRespListener`[50]

Member Summary	
Methods	
`public void`	`onMessage(SOAPMessage)`[41] Passes the given `SOAPMessage` object to this `OnewayListener` object.

Methods

4.4.1 onMessage(SOAPMessage)

public void onMessage(`SOAPMessage`[128] message)

Passes the given `SOAPMessage` object to this `OnewayListener` object. This method is invoked behind the scenes, typically by the container (servlet or EJB container) after the messaging provider delivers the message to the container. It is expected that EJB Containers will deliver JAXM messages to EJB components using message driven Beans that implement the `javax.xml.messaging.OnewayListener` interface.

Parameters:

`message` - the `SOAPMessage` object to be passed to this `OnewayListener` object

javax.xml.messaging
JAXM.4.5 ProviderConnection

Declaration

public interface `ProviderConnection`

Description

A client's active connection to its messaging provider.

A `ProviderConnection` object is created using a `ProviderConnectionFactory` object, which is configured so that the connections it creates will be to a particular messaging provider. To create a connection, a client first needs to obtain an instance of the `ProviderConnectionFactory` class that creates connections to the desired messaging provider. The client then calls the `createConnection` method on it.

The information necessary to set up a `ProviderConnectionFactory` object that creates connections to a particular messaging provider is supplied at deployment time. Typically an instance of `ProviderConnectionFactory` will be bound to a logical name in a naming service. Later the client can do a lookup on the logical name to retrieve an instance of the `ProviderConnectionFactory` class that produces connections to its messaging provider.

The following code fragment is an example of a client doing a lookup of a `ProviderConnectionFactory` object and then using it to create a connection. The first two lines in this example use the Java™ Naming and Directory Interface (JNDI) to create a context, which is then used to do the lookup. The argument provided to the `lookup` method is the logical name that was previously associated with the desired messaging provider. The `lookup` method returns a Java `Object`, which needs to be cast to a `ProviderConnectionFactory` object before it can be used to create a connection. In the following code fragment, the resulting `ProviderConnection` object is a connection to the messaging provider that is associated with the logical name "ProviderXYZ".

```
        Context ctx = new InitialContext();
        ProviderConnectionFactory pcf = (ProviderConnectionFactory)ctx.lookup("ProviderXYZ")
    ;
        ProviderConnection con = pcf.createConnection();
```

After the client has obtained a connection to its messaging provider, it can use that connection to create one or more `MessageFactory` objects, which can then be used to create `SOAPMessage` objects. Messages are delivered to an endpoint using the `ProviderConnection` method `send`.

The messaging provider maintains a list of `Endpoint` objects, which is established at deployment time as part of configuring the messaging provider. When a client uses a messaging provider to send messages, it can send messages only to those parties represented by `Endpoint` objects in its messaging provider's list. This is true because the messaging provider maps the URI for each `Endpoint` object to a URL.

Note that it is possible for a client to send a message without using a messaging provider. In this case, the client uses a `SOAPConnection` object to send point-to-point messages via the method `call`. This method takes an `Endpoint` object (actually a `URLEndpoint` object) that specifies the URL where the message is to be sent. See `SOAPConnection`$_{95}$ and `URLEndpoint`$_{52}$ for more information.

Typically, because clients have one messaging provider, they will do all their messaging with a single `ProviderConnection` object. It is possible, however, for a sophisticated application to use multiple connections.

Generally, a container is configured with a listener component at deployment time using an implementation-specific mechanism. A client running in such a container uses a `OnewayListener` object to receive messages asynchronously. In this scenario, messages are sent via the `ProviderConnection` method `send`. A client running in a container that wants to receive synchronous messages uses a `ReqRespListener` object. A `ReqRespListener` object receives messages sent via the `SOAPConnection` method `call`.

Due to the authentication and communication setup done when a `ProviderConnection` object is created, it is a relatively heavy-weight object. Therefore, a client should close its connection as soon as it is done using it.

JAXM objects created using one `ProviderConnection` object cannot be used with a different `ProviderConnection` object.

Member Summary	
Methods	
public void	`close()`[43] Closes this `ProviderConnection` object, freeing its resources and making it immediately available for garbage collection.
public javax.xml.soap.MessageFactory	`createMessageFactory(String)`[43] Creates a `MessageFactory` object that will produce `SOAPMessage` objects for the given profile.
public javax.xml.messaging.ProviderMetaData	`getMetaData()`[44] Retrieves the `ProviderMetaData` object that contains information about the messaging provider to which this `ProviderConnection` object is connected.
public void	`send(SOAPMessage)`[44] Sends the given `SOAPMessage` object and returns immediately after handing the message over to the messaging provider.

Methods

4.5.1 close()

public void close()
throws JAXMException

Closes this `ProviderConnection` object, freeing its resources and making it immediately available for garbage collection. Since a provider typically allocates significant resources outside the JVM on behalf of a connection, clients should close connections when they are not needed. Relying on garbage collection to eventually reclaim these resources may not be timely enough.

Throws:

`JAXMException`[35] - if a JAXM error occurs while closing the connection.

4.5.2 createMessageFactory(String)

public `MessageFactory`[78] createMessageFactory(java.lang.String profile)

throws JAXMException

Creates a `MessageFactory` object that will produce `SOAPMessage` objects for the given profile. The `MessageFactory` object that is returned can create instances of `SOAPMessage` subclasses as appropriate for the given profile.

Parameters:

`profile` - a string that represents a particular JAXM profile in use. An example of a JAXM profile is: "ebxml".

Returns: a new `MessageFactory` object that will create `SOAPMessage` objects for the given profile

Throws:

`JAXMException`[35] - if the JAXM infrastructure encounters an error, for example, if the endpoint that is being used is not compatible with the specified profile

4.5.3 getMetaData()

public `ProviderMetaData`[48] getMetaData()
throws JAXMException

Retrieves the `ProviderMetaData` object that contains information about the messaging provider to which this `ProviderConnection` object is connected.

Returns: the `ProviderMetaData` object with information about the messaging provider

Throws:

`JAXMException`[35] - if there is a problem getting the `ProviderMetaData` object

See Also: `ProviderMetaData`[48]

4.5.4 send(SOAPMessage)

public void send(`SOAPMessage`[128] message)
throws JAXMException

Sends the given `SOAPMessage` object and returns immediately after handing the message over to the messaging provider. No assumptions can be made regarding the ultimate success or failure of message delivery at the time this method returns.

Parameters:

`message` - the `SOAPMessage` object that is to be sent asynchronously over this `ProviderConnection` object

Throws:

`JAXMException`[35] - if a JAXM transmission error occurs

javax.xml.messaging

JAXM.4.6 ProviderConnectionFactory

Declaration

public abstract class `ProviderConnectionFactory`

```
java.lang.Object
  |
  +--javax.xml.messaging.ProviderConnectionFactory
```

Description

A factory for creating connections to a particular messaging provider. A `ProviderConnectionFactory` object can be obtained in two different ways.

- Call `ProviderConnectionFactory.newInstance` method to get an instance of the default `ProviderConnectionFactory` object. This instance can be used to create a `ProviderConnection` object that connects to the default provider implementation.

```
ProviderConnectionFactory pcf = ProviderConnectionFactory.newInstance();
ProviderConnection con = pcf.createConnection();
```

- Retrieve a `ProviderConnectionFactory` object that has been registered with a naming service based on Java Naming and Directory Interface™ (JNDI) technology.
  In this case, the `ProviderConnectionFactory` object is an administered object that was created by a container (a servlet or Enterprise JavaBeans™ container). The `ProviderConnectionFactory` object was configured in an implementation- specific way, and the connections it creates will be to the specified messaging provider. Registering a `ProviderConnectionFactory` object with a JNDI naming service associates it with a logical name. When an application wants to establish a connection with the provider associated with that `ProviderConnectionFactory` object, it does a lookup, providing the logical name. The application can then use the `ProviderConnectionFactory` object that is returned to create a connection to the messaging provider. The first two lines of the following code fragment uses JNDI methods to retrieve a `ProviderConnectionFactory` object. The third line uses the returned object to create a connection to the JAXM provider that was registered with `"ProviderXYZ"` as its logical name.

```
Context ctx = new InitialContext();
ProviderConnectionFactory pcf = (ProviderConnectionFactory)ctx.lookup(
                                                              "ProviderXYZ");
ProviderConnection con = pcf.createConnection();
```

Member Summary

Constructors

public	`ProviderConnectionFactory()` *46*

Methods

public abstract javax.xml.messaging.ProviderConnection	`createConnection()` *46* Creates a `ProviderConnection` object to the messaging provider that is the provider associated with this `ProviderConnectionFactory` object.

Member Summary	
public static javax.xml.messaging.ProviderConnectionFactory	newInstance()[46] Creates an instance of the default ProviderConnectionFactory object.

Inherited Member Summary
Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructors

4.6.1 ProviderConnectionFactory()

public ProviderConnectionFactory()

Methods

4.6.2 createConnection()

public abstract ProviderConnection[42] createConnection()
throws JAXMException

Creates a ProviderConnection object to the messaging provider that is the provider associated with this ProviderConnectionFactory object.

Returns: a ProviderConnection object that represents a connection to the provider associated with this ProviderConnectionFactory object

Throws:
JAXMException[35] - if there is an error in creating the connection

4.6.3 newInstance()

public static ProviderConnectionFactory[45] newInstance()
throws JAXMException

Creates an instance of the default ProviderConnectionFactory object.

Returns: a new instance of a ProviderConnectionFactory

Throws:

`JAXMException`[35] - if there was an error creating the default `ProviderConnectionFactory`

javax.xml.messaging
JAXM.4.7 ProviderMetaData

Declaration

public interface `ProviderMetaData`

Description

Information about the messaging provider to which a client has a connection.

After obtaining a connection to its messaging provider, a client can get information about that provider. The following code fragment demonstrates how the `ProviderConnection` object `con` can be used to retrieve its `ProviderMetaData` object and then to get the name and version number of the messaging provider.

```
ProviderMetaData pmd = con.getMetaData();
String name = pmd.getName();
int majorVersion = pmd.getProviderMajorVersion();
int minorVersion = pmd.getProviderMinorVersion();
```

The `ProviderMetaData` interface also makes it possible to find out which profiles a JAXM provider supports. The following line of code uses the method `getSupportedProfiles` to retrieve an array of `String` objects naming the profile(s) that the JAXM provider supports.

```
String [] profiles = pmd.getSupportedProfiles();
```

When a JAXM implementation supports a profile, it supports the functionality supplied by a particular messaging specification. A profile is built on top of the SOAP 1.1 and SOAP with Attachments specifications and adds more capabilities. For example, a JAXM provider may support an ebXML profile, which means that it supports headers that specify functionality defined in the ebXML specification “Message Service Specification: ebXML Routing, Transport, & Packaging, Version 1.0”.

Support for profiles, which typically add enhanced security and quality of service features, is required for the implementation of end-to-end asynchronous messaging.

Member Summary

Methods

`public int`	`getMajorVersion()`[49] Retrieves an `int` indicating the major version number of the messaging provider to which the `ProviderConnection` object described by this `ProviderMetaData` object is connected.
`public int`	`getMinorVersion()`[49] Retrieves an `int` indicating the minor version number of the messaging provider to which the `ProviderConnection` object described by this `ProviderMetaData` object is connected.
`public java.lang.String`	`getName()`[49] Retrieves a `String` containing the name of the messaging provider to which the `ProviderConnection` object described by this `ProviderMetaData` object is connected.

Member Summary	
public java.lang.String[]	getSupportedProfiles()[49] Retrieves a list of the messaging profiles that are supported by the messaging provider to which the ProviderConnection object described by this ProviderMetaData object is connected.

Methods

4.7.1 getMajorVersion()

public int getMajorVersion()

Retrieves an int indicating the major version number of the messaging provider to which the ProviderConnection object described by this ProviderMetaData object is connected.

Returns: the messaging provider's major version number as an int

4.7.2 getMinorVersion()

public int getMinorVersion()

Retrieves an int indicating the minor version number of the messaging provider to which the ProviderConnection object described by this ProviderMetaData object is connected.

Returns: the messaging provider's minor version number as an int

4.7.3 getName()

public java.lang.String getName()

Retrieves a String containing the name of the messaging provider to which the ProviderConnection object described by this ProviderMetaData object is connected. This string is provider implementation-dependent. It can either describe a particular instance of the provider or just give the name of the provider.

Returns: the messaging provider's name as a String

4.7.4 getSupportedProfiles()

public java.lang.String[] getSupportedProfiles()

Retrieves a list of the messaging profiles that are supported by the messaging provider to which the ProviderConnection object described by this ProviderMetaData object is connected.

Returns: a String array in which each element is a messaging profile supported by the messaging provider

javax.xml.messaging
JAXM.4.8 ReqRespListener

Declaration

public interface `ReqRespListener`

Description

A marker interface for components that are intended to be consumers of request-response messages. In the request-response style of messaging, sending a request and receiving the response are both done in a single operation. This means that the client sending the request cannot do anything else until after it has received the response.

From the standpoint of the sender, a message is sent via the `SOAPConnection` method `call` in a point-to-point fashion. The method `call` blocks, waiting until it gets a response message that it can return. The sender may be a standalone client, or it may be deployed in a container.

The receiver, typically a service operating in a servlet, implements the `ReqRespListener` method `onMessage` to specify how to respond to the requests it receives.

It is possible that a standalone client might use the method `call` to send a message that does not require a response. For such cases, the receiver must implement the method `onMessage` such that it returns a message whose only purpose is to unblock the `call` method.

See Also: `JAXMServlet`[37], `OnewayListener`[41], `call(SOAPMessage, Endpoint)`[96]

Member Summary	
Methods	
public javax.xml.soap.SOAPMessage	`onMessage(SOAPMessage)`[50] Passes the given `SOAPMessage` object to this `ReqRespListener` object and returns the response.

Methods

4.8.1 onMessage(SOAPMessage)

public `SOAPMessage`[128] onMessage(`SOAPMessage`[128] message)

Passes the given `SOAPMessage` object to this `ReqRespListener` object and returns the response. This method is invoked behind the scenes, typically by the container (servlet or EJB container) after the messaging provider delivers the message to the container. It is expected that EJB Containers will deliver JAXM messages to EJB components using message driven Beans that implement the `javax.xml.messaging.ReqRespListener` interface.

Parameters:

`message` - the `SOAPMessage` object to be passed to this `ReqRespListener` object

Returns: the response. If this is `null`, then the original message is treated as a "oneway" message.

javax.xml.messaging
JAXM.4.9 URLEndpoint

Declaration

public class **URLEndpoint** extends **Endpoint**[33]

```
java.lang.Object
  |
  +--javax.xml.messaging.Endpoint[33]
        |
        +--javax.xml.messaging.URLEndpoint
```

Description

A special case of the `Endpoint` class used for simple applications that want to communicate directly with another SOAP-based application in a point-to-point fashion instead of going through a messaging provider.

A `URLEndpoint` object contains a URL, which is used to make connections to the remote party. A standalone client can pass a `URLEndpoint` object to the `SOAPConnection` method `call` to send a message synchronously.

Member Summary

Constructors

public	`URLEndpoint(String)`[53]
	Constructs a new `URLEndpoint` object using the given URL.

Methods

public	`getURL()`[53]
`java.lang.String`	Gets the URL associated with this `URLEndpoint` object.

Inherited Member Summary

Fields inherited from class Endpoint[33]

`id`[34]

Methods inherited from class Endpoint[33]

`toString()`[34]

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait`

Constructors

4.9.1 URLEndpoint(String)

public URLEndpoint(java.lang.String url)

Constructs a new `URLEndpoint` object using the given URL.

Parameters:
`url` - a `String` giving the URL to use in constructing the new `URLEndpoint` object

Methods

4.9.2 getURL()

public java.lang.String getURL()

Gets the URL associated with this `URLEndpoint` object.

Returns: a `String` giving the URL associated with this `URLEndpoint` object

CHAPTER JAXM.5

References

- ebXML Transport, Routing & Packaging V1.0 - Message Service Specification
  - http://www.ebxml.org/specs/ebMS.pdf
- XML Messaging Requirements specification
  - http://search.ietf.org/internet-drafts/draft-ietf-trade-iotp2-req-00.txt
- MIME-based Secure EDI
  - http://search.ietf.org/internet-drafts/draft-ietf-ediint-as1-13.txt
- SOAP
  - http://www.w3.org/TR/SOAP
- SOAP Messages with Attachments
  - http://www.w3.org/TR/SOAP-attachments
- JavaBeans™ Activation Framework Version 1.0a
  - http://java.sun.com/products/javabeans/glasgow/jaf.html
- Java API for XML Processing - Version 1.1 Final Release
  - http://java.sun.com/xml/xml_jaxp.html

- SOAP with Attachments API for Java™ (SAAJ) - Version 1.1 Maintenance Release

  - http://java.sun.com/xml/saaj/index.html