

Java™ Web Services Developer Pack

Part 2: RPC Calls, Messaging, and the JAX-RPC and JAXM API

by Ed Ort and Ramesh Mandava, with contributions from Bhakti Mehta

October 2002

This article is the second in a series that describes the Java Web Services Developer Pack. If you're unfamiliar with Web services, the Web services model, or the basics of the Extensible Markup Language (XML) that underlies the Web services model, see [Part 1 of the series](#).

You will find the following topics covered in this article:

- [What's This About?](#)
- [XML Technologies for RPC Calls and Messaging](#)
- [A SOAP Refresher](#)
 - [SOAP for RPC](#)
 - [SOAP for Messaging](#)
 - [SOAP Messages With Attachments](#)
- [WSDL](#)
 - [Describing the Abstract](#)
 - [WSDL Document](#)
- [JAX-RPC](#)
 - [Service Endpoints](#)
 - [Artifacts](#)
 - [Java-WSDL/XML Mappings](#)
 - [Bindings](#)
 - [Stubs, Dynamic Proxies, and Dynamic Invocation](#)
 - [JAX-RPC Packages](#)
 - [The wscompile Tool](#)
 - [The wsdeploy Tool](#)
 - [A JAX-RPC Example](#)
- [JAXM](#)
 - [Clients and Providers](#)
 - [Message Exchanges](#)
 - [Profiles](#)
- [Connection and Message Objects](#)
- [JAXM Packages](#)
- [Exchanging Messages Without a JAXM Provider](#)
- [Exchanging Messages With a JAXM Provider](#)
- [A JAXM Example](#)
- [Build and Run the Sample Application](#)

What's This About?

The [Java™ Web Services Developer Pack \(Java™ WSDP\)](#) is a package of technologies and tools for building Web services using the Java programming language, and for building Java applications that access Web services.

This article is the second in a series that describes the Java Web Services Developer Pack. As its name implies, the first article in the series, [Java™ Web Services Developer Pack Part 1: Registration and the JAXR API](#), focused on Web services registration and the Java™ API for XML Registries (JAXR), an API in the Java WSDP that you can use to register a Web service or to discover a registered Web service. The first article also examined some fundamental technologies that drive the Web services model -- especially those that are specifically pertinent to registration, such as [UDDI](#) and [ebXML](#). Finally, the article presented an [example](#) that illustrated how a fictitious company named BooksToGo used JAXR to register a Web service, and how another fictitious company, BoomingBusiness.com, used JAXR to discover the service.

In this second article in the series, the focus is on client-to-Web service communication through remote procedure calls (RPC calls) and through document-oriented messages (commonly called "messaging"). The article highlights two APIs in the Java WSDP: Java API for XML-based RPC (JAX-RPC), and Java API for XML Messaging (JAXM). Their names describe what the APIs do. JAX-RPC is a Java API for making XML-based RPC calls. JAXM is a Java API for XML-based messaging. The article also examines some Web services technologies that are specifically pertinent to making RPC calls and communicating document-oriented messages in the Web services context. Finally, the article extends the example introduced in the first article to show how BoomingBusiness.com uses JAX-RPC and JAXM in its applications to communicate with Web services.

At the time the first article in the series was written, Early Access Release 1 of Java WSDP was available, and so the examples in that article were based on that (Java WSDP EA1). Since then the first customer release of Java WSDP became available and was followed by an update release (Java WSDP 1.0_01). The examples in this second article of the series are based on the update release.

If you're familiar with XML and the XML-based technologies that underlie RPC calls and messaging in the Web services model, skip to the discussion of [JAX-RPC](#) or [JAXM](#). Otherwise see [XML Technologies for RPC Calls and Messaging](#).

For More Information

[Java Web Services Developer Pack](#)
[The Java Web Services Tutorial](#)
[Java™ 2 Platform, Enterprise Edition \(J2EE™\)](#)
[Web Services and Java Technology](#)
[Web Services Technical Articles on java.sun.com](#)
[Java API for XML-Based RPC \(JAX-RPC\)](#)
[Java API for XML Messaging \(JAXM\)](#)
[SOAP with Attachments API for Java \(SAAJ\)](#)
[Simple Object Access Protocol \(SOAP\) 1.1](#)
[SOAP Messages With Attachments](#)
[Web Services Description Language \(WSDL\) 1.1](#)
[ebXML Message Service Specification V1.0](#)
[ebXML Transport, Routing & Packaging Version 1.0](#)

In addition, source code for Java WSDP components such as JAX-RPC, SAAJ, JAXM, and JAXP can be obtained from

the [Sun Community Source Licensing page](#)

Also, explore the following Sun products that incorporate Java WSDP technologies:

[Sun ONE Studio 4 update 1](#)

[Sun ONE Application Server 7](#)

About the Author

Ed Ort is a staff member of the Java Developer Connection. He has written extensively about relational database technology and programming languages.

Ramesh Mandava is a staff engineer at Sun Microsystems. Currently, he is a member of the Core XML Technologies and Standards team which is part of the Web Services Technologies and Standards group. Previously, Ramesh was the lead for the WebServices Interoperability and Quality team, which ensures that Web services deliverables from Sun meet customer quality expectations and needs.

Bhakti Mehta is a Member of Technical Staff at Sun Microsystems Inc. She is in the Web Technologies and Standards Interoperability and Quality team, and has worked with JAXP, JAXB, JAXR and JAXM.

Java™ Web Services Developer Pack

Part 2: RPC Calls, Messaging, and the JAX-RPC and JAXM API

XML Technologies for RPC Calls and Messaging

One of the things that makes the Web services model so attractive is that it's built on standard technologies, especially XML technologies, that have wide industry acceptance. The first article in the series introduced one of these XML technologies, [SOAP](#), a major underpinning of the communication mechanism for Web services. It showed how the SOAP protocol is used in the exchange of messages in a distributed environment. It also described the structure of a SOAP message. Both JAX-RPC and JAXM work in conformance with the [SOAP 1.1 specification](#) and the [SOAP 1.1 with Attachments specification](#). The RPC calls made through JAX-RPC conform to these specifications, and so too do the messages exchanged through JAXM. This section highlights some of the important features of SOAP that were introduced in the first article in the series, and shows how you can use SOAP for RPC calls as well as for messaging. The section also covers another XML-based technology, Web Services Description Language (WSDL), that plays an important role in RPC calls.

A SOAP Refresher

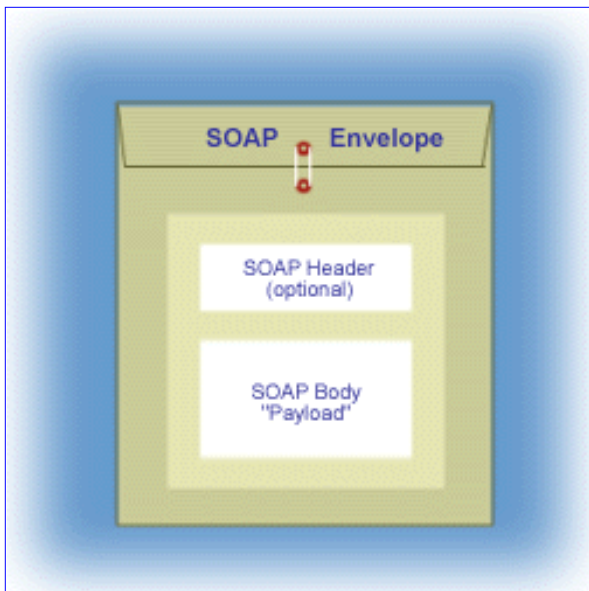
Here are some of the important characteristics of SOAP that were presented in the first article in the series:

- SOAP (which is defined by the [SOAP 1.1 specification](#)) is an XML-based protocol for exchanging information in a distributed environment.
- SOAP is a generally accepted standard for Web services.
- The basic item of transmission in SOAP is the SOAP message.
- A SOAP message consists of a mandatory [SOAP envelope](#), an optional [SOAP header](#), and a mandatory [SOAP body](#).

SOAP is an XML-based protocol for exchanging information in a distributed environment.

What those items mean is that if a Web services client exchanges some information with a Web service, what gets exchanged is a SOAP message that conceptually looks like this:

A SOAP message is an XML document. The envelope is the top element of the XML document, and is represented by an `Envelope` element. The envelope can contain namespace declarations as well as additional attributes such as an encoding style. An XML namespace defines a collection of names that can be used in XML elements and attributes. The



Click image to enlarge

namespace identifies the scope for a name, so that "name clashes" can be avoided. In particular, namespace declarations are designed to distinguish the names used in the XML document from the names used in an application that might process the XML document. They also distinguish names within the XML document, so that two elements with the same name, but associated with different namespaces, can be correctly understood as being different. An encoding style identifies the data types recognized by SOAP messages, and specifies rules for how these data types are serialized, that is, transformed for transport across the Web.

The header is represented by a `Header` element. As mentioned before, the header is optional. However, if it's included in a SOAP message, it must be the first child of the `Envelope` element. The header, through attributes, extends the SOAP message. In other words, it adds information beyond what is in the body of the SOAP message. It's important to understand that as a SOAP message travels from an originator (a client application) to a final destination (for example, an application providing a requested service), it potentially passes through a set of intermediate nodes along the path. Each node is an application that can receive and forward SOAP messages. The SOAP header can be used to indicate some

additional processing at a node, that is, processing independent of the processing done at the final destination. For example, the header could be used to request that each message be logged at a particular node.

The body, represented by a `Body` element, contains the "payload", that is, the information intended for the final destination. The `Body` element can optionally contain a `Fault` element that is used to hold error and status information returned by a processing node. The `Body` element must be an immediate child element of a `SOAP Envelope` element. If the envelope contains a header, the `Body` element must immediately follow the `Header` element.

Here's an example of a SOAP message:

```
<SOAP-ENV: Envelope
  xmlns:SOAP-ENV=
    "http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:
    encodingStyle=
      "http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Header>
    <t:Transaction xmlns:t="some-URI">
      SOAP-ENV:mustUnderstand="1">
        5
      </t:Transaction>
    </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <m:GetLastTradePrice xmlns:m="some-URI">
      <symbol>DEF</Symbol>
    </m: GetLastTradePrice>
  </SOAP-ENV:Body>
</SOAP-Envelope>
```

The `Envelope` element specifies:

- A namespace. The URI `http://schemas.xmlsoap.org/soap/envelope/` identifies the namespace used for this SOAP message. This is the standard namespace for all SOAP messages. Notice the prefix

SOAP-ENV. This prefix maps to the namespace, so that any additional references to the prefix in the SOAP message, are references to that namespace.

- An encoding style. The URI `http://schemas.xmlsoap.org/soap/encoding/` identifies the encoding style for "Section 5" encodings, that is, the SOAP encoding described in Section 5 of the SOAP specification. Section 5 define rules for encoding data types in XML, and rules for serialization.

The Header element specifies:

- A Transaction element. The URI represented by "some-URI" identifies the namespace for the element.
- A demand to process the header. The attribute value `mustUnderstand=1` means that the receiver of the header must process it. SOAP provides an attribute, called the "SOAP actor", whose value is a URI. The URI identifies the node that receives the SOAP header (remember that a SOAP message can potentially travel through multiple nodes before arriving at a final destination). Because the SOAP actor attribute is not specified in this example, it means that header should be received at the final destination. And because `mustUnderstand=1` is specified, that node must process the header.
- A value of 5. Presumably, the receiver knows how to process this value.

The Body element specifies a `GetLastTradePrice` element (and its namespace), and a subordinate element `DEF`.

If you've guessed that this example has something to do with stock quotes, you're right. The SOAP message is designed to retrieve the current price of a stock.

SOAP can be used for RPC calls as well as for messaging.

Specifically, the SOAP message includes a request for the current stock price (`GetLastTradePrice`), and passes a symbol (`DEF`) that identifies the stock. This looks similar to a Remote Procedure Call, where `GetLastTradePrice` represents a method, and `DEF` represents a parameter to the RPC call. In fact, that's what's happening here. SOAP is being used to transmit an RPC call. Using [SOAP for RPC](#) is one of two approaches to using SOAP. The other is using [SOAP for messaging](#).

SOAP for RPC

The SOAP specification states:

One of the design goals of SOAP is to encapsulate and exchange RPC calls using the extensibility and flexibility of XML.

To meet that goal, the SOAP specification includes a section that describes how to represent RPC calls and responses. The section identifies the following as information needed in an RPC call:

- The URI of the target object
- A method name
- An optional method signature
- The parameters to the method
- Optional header data

All of this information, with the exception of the URI of the target object, is carried in a SOAP message. The URI is provided by the [transport protocol](#), such as HTTP, that is used to communicate the SOAP message. The section then specifies how the other information needed in the call is represented. Here are some of the significant items in the specification:

RPC method calls and responses are carried in the SOAP Body element. In the SOAP message example above, the Body element contains the following:

```
<SOAP-ENV:Body>
  <m:GetLastTradePrice xmlns:m="some-URI">
    <symbol>DEF</Symbol>
  </m: GetLastTradePrice>
</SOAP-ENV:Body>
```

This represents a method call. Here `GetLastTradePrice` is the method called, and `DEF` is the method parameter. This indicates something important about using SOAP for RPC (as opposed to using SOAP for messaging). When using SOAP for RPC, the client knows which procedure it wants to call, and knows what parameters that procedure requires. (By comparison, when using SOAP for messaging, the client doesn't necessarily know what application will process the request.)

The response to the method call is also contained in a SOAP message `Body` element. Assume, for example, that the `Body` element of the response looks like this:

```
<SOAP-ENV:Body>
  <m:GetLastTradePriceResponse xmlns:m="some-URI">
    <price>42.40</price>
  </m: GetLastTradePriceResponse>
</SOAP-ENV:Body>
```

Notice the response element `GetLastTradePriceResponse`. Although SOAP does not require it, the convention is to give the response element the same name as the method name, with the string "Response" appended to it.

Although a response is shown here, SOAP does not require that the receiver of a request return a response. In fact, "one-way requests", where a SOAP client asks for a service without expecting a response, are perfectly valid.

SOAP can be used for one-way (asynchronous) or two-way (synchronous) requests.

Another term used for one-way is "asynchronous". By comparison, a "two-way request" is one where a SOAP client asks for a service and waits for a response before proceeding. Another term for two-way is "synchronous".

RPC method calls and responses are modeled as structures. SOAP specifies that:

- A method call and its response are both modeled as structures called structs, and that the structs have the same name and type as the method call or response.
- A struct is a compound value, each of whose member values, called accessors, have a distinct name.
- The struct for the method call must contain an accessor for each method parameter.
- The accessors must have the same name and type as the method parameters, and must appear in the struct in the same order as in the method signature.

So in the example above, the method call `GetLastTradePrice` is represented by a struct with the same name and type as the method. That struct contains one accessor, `DEF`, that has the same name and type as the parameter to the method.

The same sort of rules apply to the response:

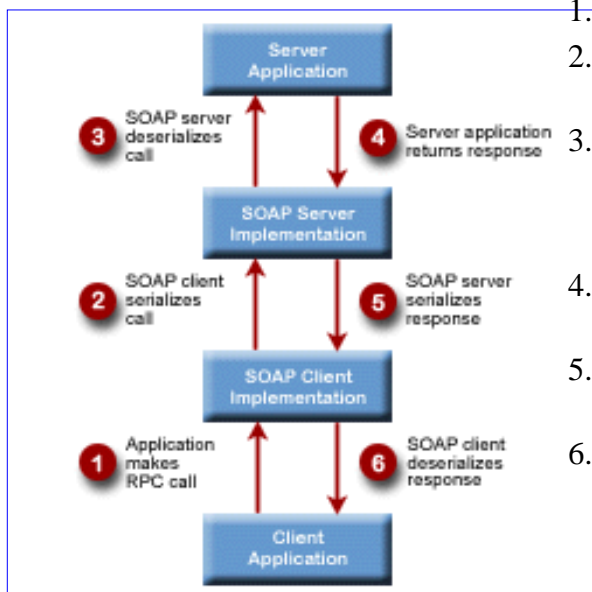
- The struct for the method response must contain an accessor for each output parameter.
- Each of these accessors must have the same name and type as its respective output parameter.
- Each accessor must appear in the struct in the same order as in the method signature.

In the example above, the method response `GetLastTradePriceResponse` is represented by a struct that contains one accessor, `price`, that has the same name and type as the output parameter.

RPC method calls and responses can be encoded according to specified encoding rules. Notice again that the `Envelope` element in the SOAP example identifies an encoding style. Specifically, the URI `http://schemas.xmlsoap.org/soap/encoding/` identifies the encoding style for Section 5 encodings. The

encoding rules essentially identify the types recognized by SOAP. For example, the Section 5 encoding rules identify simple types such as int and float, and compound types such as structs and arrays. The serialization rules specify how objects are represented in XML so that they can be exchanged between a client and a server. For example, the Section 5 serialization rules state that a simple value is represented as character data, that is, without any subelements, and a compound value is encoded as a sequence of elements.

Together, the Section 5 encoding style and the SOAP for RPC rules provide a framework for using SOAP for RPC. The framework allows an application to use RPC call semantics in an XML based, client-server scenario as follows:



1. An application makes RPC calls following the SOAP for RPC rules.
2. A SOAP client implementation serializes the calls to XML following Section 5 encodings.
3. A SOAP server implementation deserializes the calls from XML following Section 5 encodings, for use by a server application.

Then the process is reversed:

4. The server application returns a response following the SOAP for RPC rules.
5. The SOAP server implementation serializes the response to XML following Section 5 encodings.
6. The SOAP client implementation deserializes the response following Section 5 encodings, for use by the client application.

[Click image to enlarge](#)

Although Section 5 encoding style appears here to be mandatory, it's actually not. Any encoding style is allowed by SOAP. Of course, to make the information exchange sensible between the client and the server, both need to agree on the encoding style. The advantage of using Section 5 encoding is that it's a standardized encoding style, something that aligns well with the standardized protocols that underlie Web services.

Transport Protocols. One question you might ask at this point is "How does a SOAP message get transported from the client to the server and back?" The answer is that it's bound to a transport protocol such as HTTP or SMTP. SOAP does not require a particular transport binding, although HTTP is typically used to transport SOAP messages and responses. In fact HTTP is the only protocol binding that's covered in the SOAP 1.1 specification. Here, for example, is a SOAP message embedded in an HTTP 1.1 POST request. The POST action transports the SOAP message to an HTTP server.

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"
```

```
<SOAP-ENV: Envelope
  xmlns:SOAP-ENV=
    "http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:
    encodingStyle=
      "http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
```

```

    <m:GetLastTradePrice xmlns:m="some-URI">
      <symbol>DEF</Symbol>
    </m: GetLastTradePrice>
  </SOAP-ENV:Body>
</SOAP-Envelope>

```

SOAP for Messaging

Although the focus so far has been on SOAP for RPC, SOAP can also be used for messaging. In messaging, the client sends a document (rather than an RPC call) for the server to process. The document contains information in a format that both the client and server can understand. For example, the document could be something that both the client and server recognize as a purchase order. The server then processes the purchase order, and could return a response (perhaps confirming that the purchase order was received).

It's important to understand that the messaging client does not call a specific method -- in the purchase order example, the messaging client does not call a purchase order method. In fact, the client might not know what program on the server actually processes the request. Instead, the client relies on the server to invoke the appropriate application to process the request because the server understands the format of the message it receives. Similarly, what the server returns is not a method response, but rather a response generated by the processing application.

This example indicates that the server could return a response, but it doesn't have to. As is the case when using SOAP for RPC, SOAP for messaging does not require that the receiver of a request return a response. In fact, one-way (asynchronous) requests are probably more typical in messaging than two-way (synchronous) requests.

The architecture of a SOAP message is the same for messaging as it is for RPC. In both cases, the message contains an envelope, an optional header, and a body. For messaging, the payload in the body is the document passed by the client for processing by the server. Here's an example of a SOAP message used for messaging:

```

<SOAP-ENV: Envelope
  xmlns:SOAP-ENV=
    "http://schemas.xmlsoap.org/soap/envelope/"
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <bp:GetBookDetails xmlns:bp="http://bookprovider.com">
      <searchCriteria>ISBN</searchCriteria>
      <searchValue>0123454321</searchCriteria>
    </m: GetBookDetails>
  </SOAP-ENV:Body>
</SOAP-Envelope>

```

Notice that the body contains an element `GetBookDetails` that contains two subelements `searchCriteria` and `searchValue`. Assume that `GetBookDetails` is recognized as a structure that contains information about books, in particular, the International Standard Book Number (ISBN). Assume too that the server processes the message by searching a database of book information, using the ISBN value as search criteria. Finally, assume that the server returns detailed information about the book that has the specified ISBN value.

In other respects, SOAP for messaging is similar to SOAP for RPC. In particular, when used for messaging:

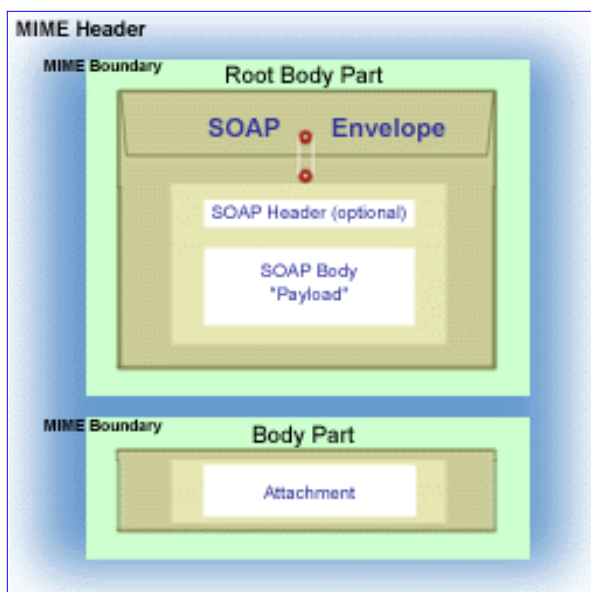
- A SOAP message can potentially pass through a set of intermediate nodes (applications) along the path from the client to the final destination. Any of these nodes can perform additional processing on the message, such as security checking, as directed by the SOAP header.
- A SOAP message can identify one or more namespaces that identify the scope of names used in the message. Notice in the messaging example, the `Envelope` element specifies the standard namespace for SOAP messages (`http://schemas.xmlsoap.org/soap/envelope/`), and the `Body` element specifies a namespace (`http://bookprovider.com`) that defines the scope of names for the body.

- A SOAP message can identify an encoding style, although an encoding style is more typically specified in SOAP for RPC. There is no default encoding style for a SOAP message.

SOAP Messages With Attachments

The SOAP 1.1 specification does not cover attachments to SOAP messages. For example, if you send a SOAP message that conforms to the SOAP 1.1 specification, you send a payload that either represents an RPC call or that contains a document for messaging. But what if you want to attach something to the SOAP message, say an image? A standard way of doing this is described in the [SOAP Messages With Attachments specification](#). This specification describes how to attach one or more items to a SOAP message, no matter what content these items hold. For example, an attachment can be an image, audio, text, even an XML document.

A SOAP message with attachments, which the specification calls a "SOAP message package", conceptually looks like this.



Click image to enlarge

The SOAP Messages With Attachments specification takes a MIME approach in describing how to build a SOAP message package. MIME, which stands for Multipurpose Internet Mail Extensions, is a standard that extends the format of Internet mail to allow for things like multipart message bodies.

A message that conforms to the MIME standard consists of a header and a body. The header contains various fields that contain information about the message. For example, the `MIME-Version` header field specifies which version of the MIME standard applies to the message. Another header field, named `CONTENT-TYPE` specifies the type of data in the body of the message. For example, the `CONTENT-TYPE` field value `Text` specifies that the body contains text, and a value of `Audio` specifies audio data. One `CONTENT-TYPE` field value has particular relevance for SOAP message packages. It's the `CONTENT-TYPE` field value `Multipart/Related`. This value specifies that the body contains a compound object consisting of several interrelated body parts. This is the MIME data type that is used to construct a SOAP message package.

A `Multipart/Related` type of compound object has a root body part (by default, this is the first body part in the compound object) and one or more related body parts. Complicating matters is that fact that each body part has its own header and body. The MIME standard further specifies that each body part in a MIME message is separated from another body part at a boundary, and that a boundary is identified in a parameter of the `CONTENT-TYPE` field. So putting all this together, a SOAP message package consists of a SOAP message that is contained in the root body part, and one or more attachments that are other body parts separated from each other at a boundary.

Here's an example of a SOAP message package. The SOAP message is in the root body part. The value of the `start` parameter in the `CONTENT-TYPE` field of the header points to content of the SOAP message (`claim061400a.xml@claiming-it.com`). The package contains an attachment that's an image (`claim061400a.tiff@claiming-it.com`).

```
MIME-Version: 1.0
Content-Type: Multipart/Related;
  boundary=MIME_boundary; type=text/xml;
  start="<claim061400a.xml@claiming-it.com>"
Content-Description:
  This is the optional message description.
```

```

--MIME_boundary
Content-Type: text/xml; charset=UTF-8
Content-Transfer-Encoding: 8bit
Content-ID: <claim061400a.xml@claiming-it.com>

<?xml version='1.0' ?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Body>
..
<theSignedForm href="cid:claim061400a.tiff@claiming-it.com"/>
..
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

--MIME_boundary
Content-Type: image/tiff
Content-Transfer-Encoding: binary
Content-ID: <claim061400a.tiff@claiming-it.com>

...binary TIFF image...
--MIME_boundary--

```

Notice that the SOAP message refers to the attached image. SOAP 1.1 allows for an accessor to point to a URI. The URI can provide any type of resource, including a non-XML resource such as an image.

WSDL

How does a Web services client know what format to use in making a request to a server? For that matter, how do the client and the server know what the request means? In SOAP for messaging, the answer is that there's a previous agreement about the format and meaning of SOAP messages exchanged between the client and the server. The client and server know what a document representing a purchase order looks like because there's previous agreement about its format. But what about SOAP for RPC? How does a Web services client know what method to call, what arguments to pass, and what to expect in response? In fact, how does the client know to expect a response? The answers to all these questions are provided by information in an XML document, called the [WSDL document](#), that contains a description of the Web service's interface and semantics.

WSDL is an XML-based language for describing a Web service.

A WSDL document contains information specified in Web Service Description Language (WSDL), as defined in the [WSDL specification](#). WSDL defines an XML schema for describing a Web service. To uncover the description for a Web service, a Web services client needs to find the service's WSDL document. One way, perhaps the most typical way, to do this is for the client to find a pointer to the WSDL document in the Web service's UDDI registration. (However the UDDI specification doesn't mandate such a link.) A typical scenario is that a business registers its service in a UDDI registry. The registry entry includes a pointer to a WSDL file that contains the WSDL document for the service. Another business searches the registry and finds the service. A programmer uses the interface and semantic information in the WSDL document to construct the appropriate calls to the service.

Describing the Abstract

WSDL deals in the abstract. A WSDL document describes a Web service as a collection of abstract items called "ports" or "endpoints". A WSDL document also defines the actions performed by a Web service and the data transmitted to these

actions in an abstract way. Actions are represented by "operations", and data is represented by "messages". A collection of related operations is known as a "port type". A port type constitutes the collection of actions offered by a Web service. What turns a WSDL description from abstract to concrete is a "binding". A binding specifies the network protocol and message format specifications for a particular port type. A port is defined by associating a network address with a binding. If a requester locates a WSDL document, and finds the binding and network address for each port, it can call the service's operations according to the specified protocol and message format.

WSDL Document

What's in a WSDL document? [Here](#) is an example. The example illustrates a WSDL document for a service that provides stock quotes. This is the service that was called in the example illustrated in [A SOAP Refresher](#). Recall that the application that requests this service supplies a symbol for a specific stock. The service responds with the current price of the stock. Notice that the document contains the following elements:

definitions. In WSDL terms, the description of a Web service's interface is its "definition", marked by the `definitions` element. This is the root element in a WSDL document. A `definitions` element specifies a name for the definition (in this example, `StockQuote`) and points (using URIs) to pertinent namespaces. All WSDL files need to point to a WSDL namespace, a SOAP namespace, and an XML Schema Definition (XSD) namespace. A WSDL file can also point to a target namespace for the associated Web service. In this example, a target namespace for the Web service is located at the URI `http://example.com/stockquote.wsdl`. Here is part of the complete definitions structure for the stock quote service:

```
<definitions name="StockQuote"
  targetNamespace="http://example.com/stockquote.wsdl"
  xmlns:tns="http://example.com/stockquote.wsdl"
  xmlns:xsd="http://example.com/stockquote.xsd"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
```

types. This structure, marked by the `types` element, contains data type definitions using some type system (such as XSD). WSDL uses a small set of primitive data types that are defined in XSD. Using the `types` structure in a WSDL document, you can define complex types that build on these basic types. These types can then be used in the operations defined for the Web service. Notice that the example defines two complex types, one that contains an element named `tickerSymbol` that is based on the primitive data type `string`, and another complex type that contains the element `price` that is based on the primitive data type `float`. Here is the `types` structure for the stock quote service:

```
<types>
  <schema targetNamespace="http://example.com/stockquote.xsd"
    xmlns="http://www.w3.org/2000/10/XMLSchema">
    <element name="TradePriceRequest">
      <complexType>
        <all>
          <element name="tickerSymbol"
            type="string"/>
        </all>
      </complexType>
    </element>
    <element name="TradePrice">
      <complexType>
        <all>
          <element name="price" type="float"/>
        </all>
      </complexType>
```

```

    </element>
</schema>
</types>

```

message. The message element describes data that is passed to and from the operations defined for the Web service. An operation identifies an action or set of actions offered by the Web service. In essence, a message describes the parameters that are passed to and from an operation. Messages are made up of parts (marked by the part element). This allows for multi-part data to be passed to an operation, for example, a purchase order with an accompanying invoice can be passed to an auditing service. In the StockQuote example, two messages are defined: GetLastTradePriceInput and GetLastTradePriceOutput. Each is a one-part message. Notice that TradePriceRequest is the part defined in GetLastTradePriceInput. Recall that it was defined in the type structure as containing the type tickerSymbol. TradePrice, the part defined in GetLastTradePriceOutput, was defined in the type structure as containing the type price. Here are the messages defined for the stock quote service:

```

<message name="GetLastTradePriceInput">
  <part name="body" element="xsd1:TradePriceRequest" />
</message>

<message name="GetLastTradePriceOutput">
  <part name="body" element="xsd1:TradePrice" />
</message>

```

port type. The portType structure defines the operations and messages for the Web service. In other words, it identifies the distinct actions provided by the Web service, and the data passed to each one. Each operation in the portType structure is marked with the operation element. The port type structure has a grammar that identifies the message-passing protocol for the operation, for example, one-way or request-response. Notice the one operation defined in the StockQuote example. The port type in the example has a grammar for a request-response protocol. The operation named GetLastTradePrice specifies an input message (GetLastTradePriceInput) and a output (response) message (GetLastTradePriceOutput). Remember that GetLastTradePriceInput has the type tickerSymbol, and TradePrice has the type price. This means that the GetLastTradePrice operation takes a tickerSymbol as input, and returns a price. Here's the port type defined for the stock quote service:

```

<portType name="StockQuotePortType">
  <operation name="GetLastTradePrice">
    <input message="tns:GetLastTradePriceInput" />
    <output message="tns:GetLastTradePriceOutput" />
  </operation>
</portType>

```

binding. The binding structure defines the message format and protocol details for operations and messages associated with a particular portType. Typically the binding structure identifies SOAP as the protocol for the binding. This is the case in the StockQuote example. Notice that the binding structure in the example specifies additional SOAP-specific information. The soap:operation element specifies a SOAPAction header URI of http://example.com/GetLastTradePrice. The URI indicates the intent of the message (in this case, a SOAP message). This allows it to perform conditional processing based on the presence of the SOAPAction header. The input and output elements in the binding indicates that the input and output messages to and from GetLastTradePrice reference a concrete schema definition using type attribute. Here's the binding defined for the stock quote service:

```

<binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
  <soap:binding style="document" transport=
    "http://schemas.xmlsoap.org/soap/http" />

```

```

<operation name="GetLastTradePrice">
  <soap:operation soapAction=
    "http://example.com/GetLastTradePrice"/>
  <input>
    <soap:body use="literal"/>
  </input>
  <output>
    <soap:body use="literal"/>
  </output>
</operation>
</binding>

```

service. The service structure defines a collection of related ports that comprise a Web service. Each port identifies a binding and an address for the binding. In the StockQuote example, a service named StockQuoteService is defined to have one port StockQuotePort. The binding for the port is StockQuoteBinding and the address for the binding is `http://example.com/stockquote`. Here's the service structure defined for the stock quote service:

```

<service name="StockQuoteService">
  <documentation>My first service</documentation>
  <port name="StockQuotePort" binding="tns:StockQuoteBinding">
    <soap:address location="http://example.com/stockquote"/>
  </port>
</service>

```

Java™ Web Services Developer Pack

Part 2: RPC Calls, Messaging, and the JAX-RPC and JAXM API

JAX-RPC

JAX-RPC is a Java API for accessing Web services through XML (SOAP-based) RPC calls. It allows a Java-based client to call Web service methods in a distributed environment, for example, where the client and the Web service are on different systems. From an application developer's point of view, JAX-RPC provides a way to call a Web service. From a Web service developer's point of view, it provides a way to make a Web service available so that it can be called from an application.

JAX-RPC is a Java API for making RPC calls that conform to SOAP specifications.

Although JAX-RPC is a Java API, it doesn't limit the client and the Web service to both be deployed on a Java platform. A Java-based client can use JAX-RPC to make SOAP-based RPC calls to Web service methods on a non-Java platform. A client on a non-Java platform can access methods in a JAX-RPC enabled Web service on a Java platform.

Also, though JAX-RPC offers a way to make SOAP-based RPC calls, it's designed to hide the complexity of SOAP. When you use JAX-RPC to make an RPC call, you don't explicitly code a SOAP message. Instead you code the call in the Java programming language, using the Java API. JAX-RPC converts the RPC call to a SOAP message and then transports the SOAP message to the server. The server converts the SOAP message and then processes it. Then the sequence is reversed. The server converts the response to a SOAP message and transports it back to the client.

In order to understand and use JAX-RPC, it's important to understand some basic concepts. These concepts include:

- [Service endpoints](#)
- [Artifacts](#)
- [Java-WSDL/XML mappings](#)

- [Bindings](#)
- [Stubs, dynamic proxies, and dynamic invocation](#)

After you're familiar with these concepts, see the [JAX-RPC Example](#).

Service Endpoints

JAX-RPC relies on [WSDL](#) for a description of Web services. WSDL describes a Web service as a collection of ports, also called endpoints, that operate on messages. Each of these endpoints identifies the distinct actions provided by the Web service, and the data passed to each action. In JAX-RPC, requests are directed to endpoints.

JAX-RPC's reliance on WSDL is important for interoperability. WSDL defines an XML schema for describing a Web service, not a Java schema. Because JAX-RPC doesn't limit the client and the Web service to both be on a Java platform, it needs a way for a Web service to be defined such that the definition is recognized on multiple platforms. WSDL provides for this platform-independent definition.

In JAX-RPC, requests are directed to endpoints. To make a Web service available to clients through JAX-RPC, a Web service developer needs to provide a JAX-RPC service endpoint definition.

To make a Web service available to clients through JAX-RPC, a Web service developer needs to provide a JAX-RPC service endpoint definition. This involves defining two Java classes for each endpoint: one that defines the JAX-RPC service endpoint interface, and the other that implements the interface. At this point you might ask "what about a Web service that's not on a Java platform?" For these services, a Web service developer can use a mapping tool to generate the JAX-RPC service endpoint definition from a WSDL document. See [Java-WSDL/XML Mappings](#) for more details.

The service endpoint interface describes the remote interface to the client. In other words, it identifies the remote methods that can be called by the client and the method signatures. The implementation class provides the code to be executed for each method. Here, for example, is a Java class that defines a service endpoint interface, `StockQuoteProvider`, for a stock quote service. Only one method is defined in the interface: `getLastTradePrice`. The method takes as input a string that represents a ticker symbol, `tickerSymbol`, and returns a float value (the last traded price for the stock represented by the ticker symbol).

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface StockQuoteProvider extends Remote {

    public float getLastTradePrice(String tickerSymbol)
        throws RemoteException;

}
```

Notice that the service endpoint definition extends the `java.rmi.Remote` interface, and the method declares that it throws a `java.rmi.RemoteException` exception. These are JAX-RPC requirements. All JAX-RPC service endpoint definitions must extend the `java.rmi.Remote` interface, and their methods must

All JAX-RPC service endpoint definitions must extend the `java.rmi.Remote` interface, and their methods must declare that they throw a `java.rmi.RemoteException` exception.

declare that they throw a `java.rmi.RemoteException` exception. There are other rules that govern JAX-RPC service endpoint interfaces. For example, method parameters and return types must be JAX-RPC supported types.

The service developer would then need to provide the implementation class for the `getLastTradePrice` method. It would look something like this:

```
import java.xml.rpc.server.ServiceLifecycle;
```

```

public class StockQuoteService implements
    StockQuoteProvider, ServiceLifecycle {

public float getLastTradePrice(String tickerSymbol)
    {
    // Code for the method
    ...
    }

}

```

Notice that the class implements the `ServiceLifecycle` interface as well as the `StockQuoteProvider` interface. The JAX-RPC runtime system uses the implementation of the `ServiceLifecycle` interface to manage the lifecycle of the service endpoint class. For example, the runtime system uses the implementation of methods in the `ServiceLifecycle` interface to initialize and eliminate instances of the service endpoint class.

After a service endpoint is defined, it's deployed in a container that implements the JAX-RPC runtime system on the server. For example, the service endpoint can be deployed as a servlet in a servlet container, or a stateless session bean in an EJB container. The JAX-RPC specification defines a non-normative deployment descriptor for an endpoint deployed in a servlet container. The JAX-RPC Reference Implementation, which is part of Java WSDP 1.0_01 FCS, supports only deployment in a servlet container.

Artifacts

In order to handle communication between a client and a service endpoint, JAX-RPC needs various classes, interfaces, and other files on both the client and server side of the communication. These files are collectively called artifacts. An implementation of JAX-RPC must provide a tool to generate these artifacts. The specification does not require any specific tool to do this -- it's implementation dependent. In the Java WSDP, the tool is [wscompile](#).

Among the required artifacts for client-server communication, are stubs, ties, serializers, and deserializers. Stubs are classes that represent a service endpoint on the client. This allows a JAX-RPC client to invoke a remote method on a service endpoint as though the method were local. You can learn more about stubs in [Stubs](#). A tie is the server-side analog to a stub. It represents the service endpoint on the server. Serializers and deserializers are classes that are used to serialize a Java type to XML, or XML to Java, respectively.

Use a mapping tool, such as the `wscompile` tool in the Java WSDP, to generate artifacts such as stubs, ties, serializers, and deserializers. You can also use the `wscompile` tool to produce a WSDL document from a JAX-RPC service endpoint definition, or produce a JAX-RPC service endpoint definition from a WSDL document.

Java-WSDL/XML Mappings

Recall that JAX-RPC relies on WSDL for the description of Web services. In fact, being able to access a WSDL description of a Web service is a requirement for JAX-RPC interoperability -- remember JAX-RPC does not require both the client and server to be on a Java platform. To meet this WSDL requirement, all JAX-RPC implementations must be able to produce a [WSDL document](#) from a service endpoint definition. The JAX-RPC specification defines the mapping between the definition of a JAX-RPC service endpoint and a WSDL service description. For example, it specifies that a service endpoint interface is mapped to a WSDL `portType` structure, and the methods defined in the service endpoint interface are mapped to `operation` elements in the `portType` structure. A JAX-RPC implementation must be able to produce a Web service description according to the mappings defined in the JAX-RPC specification. As is the case for generating artifacts, the specification does not require any specific tool to do this. In the Java WSDP, the tool is `wscompile`. In other words, you can use the `wscompile` tool to generate artifacts such as stubs and ties, and also use it to produce a WSDL document from a JAX-RPC service endpoint definition. The tool also works in reverse -- you can use

it to produce a JAX-RPC service endpoint definition from a WSDL document. [Here](#), for example, is a WSDL document that the wscompile tool generates for the `StockQuoteService` service endpoint.

The JAX-RPC specification also lists the Java data types that a JAX-RPC implementation must support. For example, it requires support for Java primitive data types such as `boolean`, `byte`, `int`, and `double`. In addition, it specifies the mapping of each supported Java data type to an XML data type, that is, a data type defined in XSD (XML Schema Definition language). For example, the Java data type `boolean` maps to the XML data type `xsd:boolean`. Complying with these specifications, enables JAX-RPC implementations on the client and server to interchange method parameters and return data in an intelligible way.

Bindings

Notice the `binding` element in the WSDL document that the wscompile tool generated for `StockQuoteService`. In generating a WSDL document, a mapping tool configures one or more protocol bindings for each service endpoint. The binding ties an abstract service endpoint definition to a specific protocol and transport. The binding in the `StockQuoteService` example is SOAP 1.1 over HTTP. It's important to note that the JAX-RPC specification does not mandate any specific XML-based protocol for exchanging and transporting information. However, the specification does state that "An interoperable JAX-RPC system is required to support the SOAP 1.1 with attachment protocol." What this means is that for interoperability, a JAX-RPC implementation must support SOAP 1.1 with attachments, but additional protocols can be supported. Similarly, the JAX-RPC specification requires an implementation to support HTTP 1.1 network transport protocol. However an implementation can support additional transport protocols. The JAX-RPC 1.0 reference implementation supports SOAP 1.1 and SOAP 1.1 with Attachments as the XML-based protocols for information exchange, and HTTP 1.1 as the network transport protocol.

The JAX-RPC specification does not mandate any specific XML-based protocol for exchanging and transporting information. But it does require an interoperable JAX-RPC system to support the SOAP 1.1 with attachment protocol and the HTTP 1.1 network transport protocol.

Stubs, Dynamic Proxies, and Dynamic Invocation

A JAX-RPC client can invoke a remote method on a service endpoint in various ways:

- It can invoke the remote method through a local object called a [stub](#).
- It can use a [dynamic proxy](#) to invoke the method.
- It can dynamically invoke the method using the JAX-RPC [Dynamic Invocation Interface \(DII\)](#).

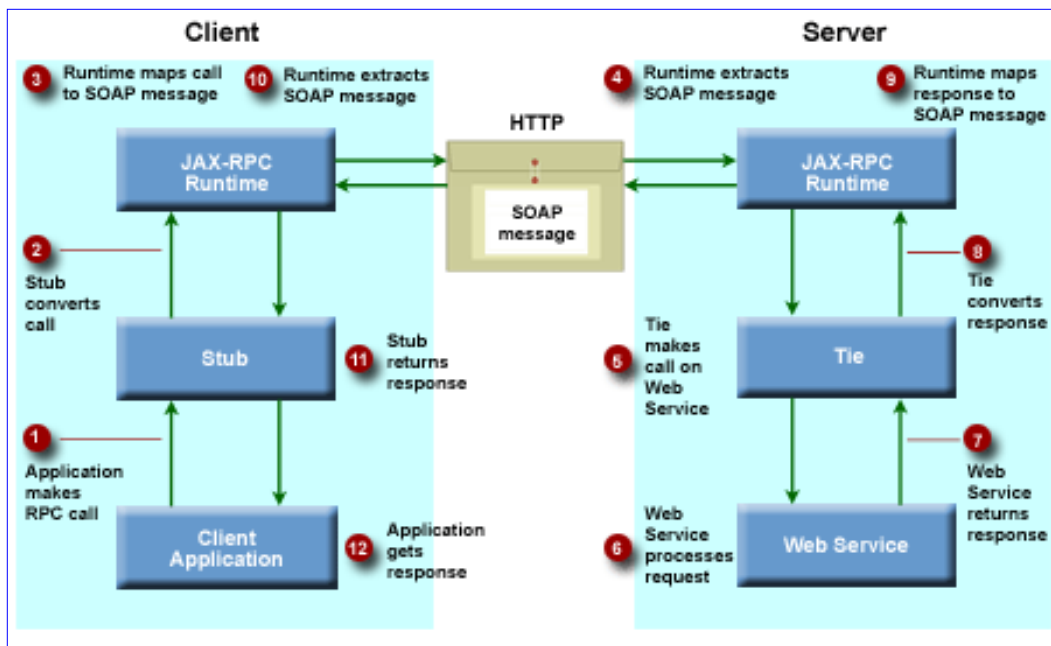
Stubs

Stubs are used when a JAX-RPC client knows what method to call and how to call it (for example, what parameters to pass). Invoking a remote method through a stub is like invoking a remote method using the [Java](#)

A stub is designed to simplify remote method calls, that is, by making them appear like local method calls.

[Remote Method Invocation \(RMI\) system](#). As is the case for RMI, in JAX-RPC, a stub is designed to simplify remote method calls, that is, by making them appear like local method calls. A local stub object is used to represent a remote object. To make a remote method call, all a JAX-RPC client needs to do is make the method call on the local stub. The stub (using the underlying runtime environment) then formats the method call and directs it to the server -- this process is called marshalling. On the server, a class called a tie (also called a skeleton) unmarshals this information and makes the call on the remote object. The process is then reversed for returning information to the client.

The following figure illustrates what happens when a JAX-RPC client invokes a remote method through a stub.



Click image to enlarge

RPC Call on the client:

1. A client application makes an RPC call on a local object called a stub.
2. The stub converts the RPC call to JAX-RPC runtime system requests.
3. The JAX-RPC runtime system maps the requests to a SOAP message and transmits it as part of an HTTP request.

RPC response on the client:

10. The JAX-RPC runtime system extracts the SOAP message and maps it to a response on the stub.
11. The stub returns the response to the client application.
12. The client application gets (and processes) the response.

RPC Call on the the server:

4. The JAX-RPC runtime system extracts the SOAP message from the HTTP request, and maps it to a method call on a local object called a tie.
5. The tie invokes the method call on the Web service.
6. The Web service processes the request.

RPC response on the server:

7. The Web service returns the response.
8. The tie converts the response to JAX-RPC runtime system requests.
9. The JAX-RPC runtime system maps the response to a SOAP message and transmits it as part of an HTTP request.

As mentioned in [Artifacts](#), an application developer uses a mapping tool, such as the wscompile tool, to generate the stub (as well as other artifacts). There are actually two ways to generate the stub. The stub can be generated from the service endpoint definition or from a WSDL document. An application developer can use the wscompile tool to generate the

stub using either approach.

In order to use a stub, it has to be configured with information such as the service endpoint address. If a stub is generated from a WSDL document, the mapping tool configures the stub using information in the WSDL document. However if the stub is generated from a service endpoint interface, the developer needs to provide the configuration information. JAX-RPC provides a client-side API, `javax.xml.rpc.Stub`, to specify this information.

Here, for example, is part of what the code might look like for a client class that uses a stub to invoke the `getLastTradePrice` method. In this example, the stub was generated from a service endpoint definition:

```
import javax.xml.rpc.Stub;
import javax.xml.rpc.JAXRPCException;

public class SqpClient {
    public static void main(String[] args) {
        try {
            StockQuoteProvider_Stub sqp =
                (StockQuoteProvider_Stub)(
new StockQuoteService_Impl().getStockQuoteProviderPort());
            sqp._setProperty(
                Stub.ENDPOINT_ADDRESS_PROPERTY,
                "http:// ...")
            float quote = sqp.getLastTradePrice(
                "ACME");

        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Notice how the client class gets an instance of the stub. The JAX-RPC specification does not specify a standard way to do this. The approach used here assumes that the stub was generated using the `wscompile` tool. The `StockQuoteService_Impl` class is a client-side implementation class that is generated by the `wscompile` tool for the `StockQuoteService` service endpoint. The `StockQuoteService_Impl` class provides a method, `getStockQuoteProviderPort`, to get a reference to the stub for the endpoint. Also notice how the endpoint address is set on the stub using the `_setProperty` method. Finally, notice the `JAXRPCException` class. `JAXRPCException` is thrown from the core JAX-RPC APIs to indicate an exception related to the JAX-RPC runtime system.

Dynamic Proxies

A dynamic proxy is a class that dynamically supports service endpoints at runtime, without the need to generate stubs. A client creates a dynamic proxy by calling the `getPort` method of the interface `javax.xml.rpc.Service`. In making the call, the client specifies the port for a service endpoint and the service endpoint interface. The method returns a dynamically built implementation of the service endpoint. The client can then invoke a method on the dynamic proxy. For example, here is what the code might look like for a client class that uses a dynamic proxy to invoke the `getLastTradePrice` method:

```
import java.net.URL;
import javax.xml.rpc.Service;
import javax.xml.rpc.JAXRPCException;
import javax.xml.namespace.QName;
```

```
import javax.xml.rpc.ServiceFactory;

public class SqpClient {

    public static void main(String[] args) {
        try {
            String urlString = "http://...";
            String namespaceUri = "http://proxy.org/wsdl";
            String serviceName = "StockQuoteService";
            String portName = "StockQuoteProviderPort";

            URL sqpWsdUrl = new URL(urlString);

            ServiceFactory serviceFactory =
                ServiceFactory.newInstance();

            Service sqpService =
                serviceFactory.createService(sqpWsdUrl,
                    new QName(namespaceUri, serviceName));

            StockQuoteProvider sqp =
                (StockQuoteProvider) sqpService.getPort(
                    new QName(namespaceUri, portName),
                    StockQuoteProvider.class);

            float quote = sqp.getLastTradePrice("ACME");

        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Notice how the client class uses the `createService` method of the `ServiceFactory` class to create an instance of the service. The method takes as input the URL of the WSDL document location for the service, and a `QName` object that provides the qualified name for the service.

Dynamic Invocation Interface (DII)

Sometimes a JAX-RPC client needs to invoke a remote method dynamically. For example, consider a scenario where the client doesn't know the remote method name or its signature until run time. In cases like these, the client can use the JAX-RPC Dynamic Invocation Interface (DII). As is the case for dynamic proxies, dynamic invocation does not involve the use of stubs.

To use the DII, a client:

- Creates a `Call` object. The object provides an in-memory model of the WSDL description of a service. The `JAX-RPC Service` class acts as a factory for these objects.
- Creates the call using one of the `createCall` methods on the `Service` object. The configuration of a `Call` object includes the following properties:
 - The name of a specific operation
 - The port type for the service endpoint
 - Binding properties such as the `SOAPAction` header URI for the SOAP binding to HTTP

- The name, type, and mode of input and output parameters
- The return type

If the service endpoint implementation was generated from a WSDL document, the WSDL description can provide most of the configuration information. Otherwise, a client uses setter methods to specify the configuration information.

- Invokes the remote method on the `Call` object.

DII supports two types of invocation: synchronous request-response mode and one-way mode. In synchronous request-response mode invocation, the client uses the `invoke` method of the `Call` object to invoke a remote method. The client then waits (specifically, the client thread blocks) until the operation is complete, that is, until a response (or exception) is returned. In one-way invocation mode, the client uses the `invokeOneWay` method of the `Call` object to invoke a remote method. In this case, the client doesn't block -- in other words, the client continues processing without waiting for the operation to complete.

Here is an example of a client that uses DII:

```
import javax.xml.rpc.Call;
import javax.xml.rpc.Service;
import javax.xml.rpc.ParameterMode;
import javax.xml.rpc.JAXRPCException;
import javax.xml.namespace.QName;
import javax.xml.rpc.ServiceFactory;

public class SqpClient {

    private static String qnameService =
        "StockQuoteService";
    private static String qnamePort =
        "StockQuoteProviderPort";
    private static String ENCODING_STYLE_PROPERTY =
        "javax.xml.rpc.encodingstyle.namespace.uri";
    private static String NS_XSD =
        "http://www.w3.org/2001/XMLSchema";

    public static void main(String[] args) {
        try {
            factory = ServiceFactory.newInstance();
            Service service =
                factory.createService(new QName(
                    qnameService));
            QName port = new QName(qnamePort);
            Call call = service.createCall(port);

            call.setTargetEndpointAddress(
                endpointAddress);

            call.setProperty(
                Call.SOAPACTION_USE_PROPERTY,
                new Boolean(true));
            call.setProperty(
                Call.SOAPACTION_URI_PROPERTY,
                "");
        }
    }
}
```

```

        call.setProperty(ENCODING_STYLE_PROPERTY,
                        URI_ENCODING);

        call.addParameter("String_1", QNameType.STRING,
                        ParameterMode.IN);
        call.addParameter("String_2", QNameType.STRING,
                        ParameterMode.OUT);

        call.setReturnType(QNameType.INT);

        call.setOperationName(
            new QName(BODY_NAMESPACE_VALUE,
                    "getLastTradePrice"));

        Object[] inParams = new Object[] {"ACME"};
        Integer ret = (Integer) call.invoke(inParams);
        Map outParams = call.getOutputParams();
        String OutValue = (String)outParams.get("param2");

    } catch (Exception ex) {
        ex.printStackTrace();
    }
}
}
}

```

In this example:

- The `newInstance` method of the `ServiceFactory` class creates a `ServiceFactory` object.
- The `createService` method of the `ServiceFactory` object creates a `Service` object. The method `call` specifies a qualified name for the service.
- The `createCall` method of the `Service` object creates a `Call` object. The method `call` specifies a qualified name for the service endpoint.
- Setter methods such as `setTargetEndpointAddress` and `setOperationName` are used to configure the `Call` object.
- The `setProperty` method is used to set standard properties that are listed in the JAX-RPC specification, such as `SOAPACTION_USE_PROPERTY` which indicates whether or not `SOAPAction` is used. In this example, `SOAPACTION_USE_PROPERTY` is set to `true`, so `SOAPAction` is used.
- The `addParameter` method is used to add a parameter and type for the operation specified in the `setOperationName` method. Note that the values of these parameters are obtained from the WSDL document for the service.
- The `setReturnType` method is used to set the return type for the operation specified in the `setOperationName` method.
- The `invoke` method invokes the operation specified in the `setOperationName` method, using a synchronous request-response interaction mode. The method `call` specifies the input parameters for the invocation.
- The `getOutputParams` method returns a `Map` of {name, value} for the output parameters of the invoked operation.

For a more complete DII example, see [A JAX-RPC Example](#).

JAX-RPC Packages

The JAX-RPC API comprises a number of packages. Two of the packages are used in the examples that illustrate how to

use [stubs](#), [dynamic proxies](#), and the [Dynamic Invocation Interface](#). Those packages are:

- `javax.xml.rpc`. This package contains the core JAX-RPC APIs for the client programming model. This includes interfaces and classes that are used by a JAX-RPC client.

The interfaces in the package are:

Stub	This is the common base interface for stub classes. All generated stub classes are required to implement the <code>Stub</code> interface.
Service	This interface provides support for creating a dynamic proxy and for creating a <code>Call</code> object.
Call	This interface provides support for the dynamic invocation of an operation on a service endpoint.

An important class in the package is:

ServiceFactory	This is an abstract class that provides a factory for creating <code>Service</code> objects.
----------------	--

- `javax.xml.namespace`. This package contains a class that provides a qualified name. The class is:

QName	This class represents the value of an XML qualified name as specified in XML Schema Part2: Datatypes specification.
-------	---

Other packages in the JAX-RPC API are intended primarily for JAX-RPC implementations. For example, they provide interfaces and classes for Java-XML serialization and deserialization, for handling SOAP messages, and for data type mapping.

Note that the JAX-RPC API also depends on another package, `javax.xml.soap`, that is defined by the [SOAP with Attachments API for Java \(SAAJ\) 1.1 specification](#). As its name implies, SAAJ is an API that is used to represent a SOAP message with attachments. The JAX-RPC API has a number of dependencies on the SAAJ API. For example, it uses elements of the `javax.xml.soap` package to represent the mapping of literal fragments in a SOAP message.

The wscompile Tool

`wscompile` is a mapping tool provided in the Java WSDP to generate stubs, ties, and other artifacts. You can also use the `wscompile` tool to produce a WSDL document from a JAX-RPC service endpoint definition, or produce a JAX-RPC service endpoint definition from a WSDL document.

`wscompile` and a companion tool, [wsdeploy](#), replace the `xrcc` tool that was provided in earlier releases of the Java WSDP. The `xrcc` tool is still provided in Java WSDP 1.0_01, however its use is deprecated.

You have the option of running the `wscompile` tool to produce only client-side artifacts such as stubs, server side artifacts such as ties, or both client and server-side artifacts. In any case, you need to provide a configuration file as input to the tool. The configuration file is an XML file that contains information needed by the tool, such as what artifacts to generate, data for the generated WSDL document (if a WSDL document is generated from a JAX-RPC service endpoint

definition), or data for the generated service endpoint definition (if the service endpoint definition is generated from a WSDL document).

Here is an example of a configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration xmlns="http://java.sun.com/xml/ns/jax-
    rpc/ri/config">
  <service name="StockQuote"
    targetNamespace="http://example.com/stockquote.wsdl"
    typeNamespace="http://example.com/stockquote/types"
    packageName="stockqt">
    <interface name="stockqt.StockQuoteProvider"
      servantName="stockqt.StockQuoteServiceImpl"/>
  </service>
</configuration>
```

Notice that the file starts with a `configuration` element. This element identifies the namespace for the configuration file, which always must be `http://java.sun.com/xml/ns/jax-rpc/ri/config`. Next, the `service` element identifies the input as coming from the service endpoint definition. This information will be used by the `wscompile` tool to generate a WSDL document as well as artifacts. If the input was a WSDL document, the configuration file would specify a `wsdl` element instead of a `service` element. In that case, the tool would use that information to generate a service endpoint definition.

The `service` element identifies:

- The name of the service: `StockQuote`
- The namespace for the service: `http://example.com/stockquote.wsdl`
- The type namespace for the service: `http://example.com/stockquote/types`
- The package name for artifacts generated by the `wscompile` tool: `stockqt`

The `interface` element within the `service` element structure identifies the service endpoint interface (`StockQuoteProvider`) and the interface implementation (the "servant name"): `StockQuoteServiceImpl`.

In addition to the stubs and ties that it generates, the `wscompile` tool also generates a model file. This model file is designed to be used by the [wsdeploy tool](#) in generating a deployable J2EE WAR (web archive) file for a Web service. The model file is an XML file that contains information about the service, such as dependency relationships.

To run the `wscompile` tool, you run a script (in UNIX) or a batch file (in Windows). The UNIX syntax is:

```
wscompile.sh options config-filename
```

The Windows syntax is:

```
wscompile.bat options config-filename
```

where *options* are one or more control options for the `wscompile` tool, and *config-filename* is the name of a configuration file.

Some of the `wscompile` options are:

<code>-gen:both</code>	Generates client-side and server-side artifacts (and depending on the contents of the configuration file, a WSDL document or a service endpoint definition)
------------------------	---

-gen:client	Generates client-side artifacts (and depending on the contents of the configuration file, a WSDL document or a service endpoint definition)
-gen:server	Generates server-side artifacts (and depending on the contents of the configuration file, a WSDL document or a service endpoint definition)
-keep	Keeps the generated source files after they are compiled
-classpath	Specifies where to find the input class files
-d	Specifies a directory for generated output
-model	Writes the generated model to the specified file

If you invoke the `wscmcompile` tool without specifying an option, it will display the tool invocation syntax and a description of all the options.

Here's an example of an `wscmcompile` tool invocation in UNIX. (Although shown on two lines, the command is specified on one line.)

```
wscmcompile.sh -gen:both -keep -d classout
               -model wscmodel.xml.gz config.xml
```

Assume that the contents of the configuration file (`config.xml`) are the same as shown in the earlier configuration file example. As a result, the `wscmcompile` tool will generate client-side and server-side artifacts. It will also generate a WSDL document from the service endpoint definition identified in the configuration file. Source files that are generated by the `wscmcompile` tool will be kept, the generated classes will be placed in the `classout` directory, and the model will be written to file `wscmodel.xml.gz`.

The `wscmdeploy` Tool

`wscmdeploy` is a tool provided in the Java WSDP to generate a deployable WAR file for a service. The tool takes as input a "raw" (that is, not yet deployable) WAR file for the service, and generates a deployable WAR file. A raw WAR file contains the following components:

- META-INF
 - MANIFEST.MF Contains information about the file packaged in the WAR file.
- WEB-INF
 - web.xml Contains information about the service, such as its display name and description.
 - model.xml.gz A compressed version of the model.
 - jaxrpc-ri.xml The deployment descriptor for the service.
 - classes A directory that contains the bulk of the components.

Typically, you create a raw WAR file with a GUI development tool or through a build tool such as the Ant Build Tool packaged in the Java WSDP. A deployable WAR file contains an updated version of the `web.xml` component. The

updated version contains additional information about the service, such as the servlet class for the service endpoint. The deployable WAR file also contains a runtime deployment descriptor, `jaxrpc-ri-runtime.xml`. The descriptor contains runtime information, such as the class for the tie.

In generating the file, `wsdeploy` examines the deployment descriptor, `jaxrpc-ri.xml`. If the deployment descriptor identifies a model file, the information in the model file is used in generating the deployable WAR file. If the deployment descriptor does not identify a model file, the `wsdeploy` tool generates the model.

Also, note that in generating the file, `wsdeploy` runs `wscompile` with the `-gen:server` option. In other words, `wsdeploy` generates server-side artifacts (such as a tie), and can also generate a WSDL document or a service endpoint definition. So rather than running `wscompile` explicitly to generate server-side artifacts (and possible a WSDL document or a service endpoint definition), you can run `wsdeploy` to generate the same artifacts, classes, and WSDL file.

To run the `wsdeploy` tool, you run a script (in UNIX) or a batch file (in Windows). The UNIX syntax is:

```
wsdeploy.sh options war-filename
```

The Windows syntax is:

```
wscompile.bat options war-filename
```

where *options* are one or more control options for the `wsdeploy` tool, and *war-filename* is the name of a raw WAR file.

Some of the `wsdeploy` options are:

<code>-o</code>	Specifies where to place the generated WAR file (this is a required option)
<code>-keep</code>	Keeps temporary files generated during the process
<code>-classpath</code>	Specifies where to find the input class files
<code>-tmpdir</code>	Specifies a directory for temporary files

If you invoke the `wsdeploy` tool without specifying an option, it will display the tool invocation syntax and a description of all the options.

Here's an example of an `wsdeploy` tool invocation in UNIX:

```
wsdeploy.sh -o target.war raw.war
```

In this example, the `wsdeploy` tool will take the raw WAR file, `raw.war`, and generate a deployable version, named `target.war`.

A JAX-RPC Example

This section presents an example of JAX-RPC in use. The example is based on a sample application that uses JAX-RPC (as well as JAXM) to access Web services. The example also uses tools, such as the `wscompile` tool, that are provided in the Java WSDP. For instructions on building and running the sample application, see [Build and Run the Sample Application](#).

[Part 1 of this series](#) presented an example that illustrated how a fictitious company named BooksToGo used JAXR to register a Web service, and how another fictitious company, BoomingBusiness.com, used JAXR to discover the service. Recall that BoomingBusiness.com provides a Web site to its employees. The Web site is a portal to a variety of employee services. One of the services that BoomingBusiness.com plans to make available through its Web site is an online service for ordering books. BooksToGo is a provider of that service. (The BoomingBusiness.com employee portal will access the book ordering service using JAXM. This is explained further in [A JAXM Example](#).)

Let's extend the example. Assume that BoomingBusiness.com decides to add another employee service to its portal: a retirement planning service. Using this service, an employee could search for available retirement funds, identify which companies provide those funds, and then get quotes from prospective providers for their services. The employee could then select a provider, and invest money in the funds of choice.

As was the case for the book ordering service, BoomingBusiness.com wants to implement this extension as a Web services-based solution. In particular, BoomingBusiness.com plans to extend the portal so that when an employee requests the retirement planning service, an underlying program dynamically searches a business registry for retirement fund providers. BoomingBusiness.com's IT staff further decides to use JAX-RPC for client-to-provider interactions such as getting a list of funds that the providers offer, and getting quotes from providers.

Pan American Services and Retirements Specialists are two companies that specialize in providing retirement services. They are both retirement fund providers. Both companies want to make their retirement services available to clients through JAX-RPC. Let's examine what the retirement fund providers do to make their retirement services available to clients through JAX-RPC. Then let's examine what BoomingBusiness.com does to access the retirement services through JAX-RPC.

The retirement fund providers take the following actions:

- [Provide the Service Endpoint Interface](#)
- [Register the Retirement Service](#)
- [Generate the Artifacts](#)
- [Deploy the Service](#)

BoomingBusiness.com takes the following actions:

- [Create the Client](#)

Provide the Service Endpoint Interface

For the purposes of this example, assume that the retirement service providers have agreed to support a service endpoint interface named RetirementServiceIF. Here is part of the source code for the RetirementServiceIF class:

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RetirementServiceIF extends Remote {

    public String[] getAvailableFunds( )
                                   throws RemoteException;

    public double getQuote(
        FundInfo[] reqFunds, double monthlyInvestment )
        throws RemoteException;

    public String confirmQuote(
```

```
FundInfo[] reqFunds, double monthlyInvestment,
    EmployeeInfo ei) throws RemoteException;
```

```
}
```

The source code shown for the service endpoint interface defines three methods:

- `getAvailableFunds`. This method returns the retirement funds offered by a provider.
- `getQuote`. This method returns a quote from each provider that offers all the funds that the client requests. The input parameters to the method are an array (`reqFunds`) that contains the names of the requested retirement funds, and a monthly investment amount (`monthlyInvestment`) for each fund.
- `confirmQuote`. This method confirms that a client has accepted a provider's quote. Like the `getQuote` method, the input parameters to the `confirmQuote` method are an array of requested funds, and the monthly investment amount. The method also takes as input an `EmployeeInfo` object `ei` that contains information about the employee who is investing in the retirement funds.

To see the complete source code for the `RetirementServiceIF` class used in the sample application, look [here](#).

Each retirement fund provider implements the `RetirementServiceIF` service endpoint interface. For example, here is part of the source code for the implementation class provided by Pan American Services. To make things simpler for this example, the funds provided by Pan American Services are specified in the class. In a real-life example, the class would probably access a database to get the names of the funds.

```
public class PanAmericanRSImpl implements
    RetirementServiceIF {
    public String providerName="PanAmerican";
    private Vector availableFunds = null;

    public PanAmericanRSImpl( )
    {
        availableFunds = new Vector();
        availableFunds.addElement("Old age funds");
        availableFunds.addElement(
            "Happy old days funds");
        availableFunds.addElement("Sunrise funds");
    }

    public String[] getAvailableFunds ( ) {
        if ( availableFunds == null ) {
            return null;
        }

        String[] aFunds =
            new String[ availableFunds.size() ];
        availableFunds.copyInto( (Object[])aFunds );
        return aFunds;
    }

    public double getQuote(
        FundInfo[] rFunds, double monthlyInvestment ) {

        for ( int i=0; i<rFunds.length; i++ )
```

```

    {
        FundInfo fi = rFunds[i];
        System.out.println("Requested :"
            + fi.getFundName() + "="
            + fi.getFundPercent() );
        if ( !availableFunds.contains(
            fi.getFundName() ) ) {
            return 0;
        }
    }

    return 3.0;
}

public String confirmQuote( FundInfo[] rFunds,
    double monthlyInvestment ,
    EmployeeInfo ei ) {

    for ( int i=0; i<rFunds.length; i++ )
    {
        FundInfo fi = rFunds[i];
        System.out.println("Requested :"
            + fi.getFundName() + "="
            + fi.getFundPercent() );
    }

    return "Fund Request confirmed";
}
}

```

To see the complete source code for the `PanAmericanRSImpl` implementation class used in the sample application, look [here](#).

After coding the classes for the service endpoint definition, the service providers compile the classes.

Register the Retirement Service

Pan American Services and Retirements Specialists use the JAXR API in the Java WSDP package to register their retirement services. They choose to register the services in a UDDI registry. (In the sample application, the UDDI registry used is the Java WSDP Registry Server v1.0_02 provided in the Java WSDP v1.0_01.) For a description of the steps involved in registering a Web service, see the [JAXR example](#) in Part 1 of this series.

Each provider publishes information about its retirement service. For example, Pan American Retirement Service publishes the following information:

Business Name	Pan American Services
---------------	-----------------------

Contact Information	Primary Contact: Bhakti Mehta Phone number: (408)1234567 Email Address: bhakti.mehta@panamerican.com
Classification Scheme (classification, code)	NAICS (Pension Funds, 52511)
Service	Retirement Service
Service Binding	Description: JAXRPC-FCS (SOAP/HTTP) based binding Access Point: http://localhost:8080/PanAmericanRS/jaxrpc/RetirementServiceIF

Notice the classification code, 52511. This is the [North American Industry Classification System \(NAICS\) code](#) for pension funds (in other words, retirement funds). To find retirement fund providers, BoomingBusiness.com's application program will query a registry for entries that have the pension fund code in their classification.

Notice too the binding information: JAXRPC-FCS (SOAP/HTTP) based binding. This identifies the service as accessible through JAX-RPC. In looking for retirement fund providers, BoomingBusiness.com's application program will look for entries that meet this criteria. The access point identifies the URL for the service endpoint. JAX-RPC clients will specify this URL to contact the service. In this example, the URL is a localhost URL, so that the service endpoint is actually on your machine. In a real Web service deployment, the service endpoint would probably be on a different machine.

The sample application includes a class that registers the retirement services. To see the source code for the class, look [here](#).

Generate the Artifacts

Each retirement service provider uses the `wscmpile` tool to generate the artifacts needed for a client to access and use the service. For example, here is a command that Retirement Specialists executes in the UNIX environment to generate the artifacts (although shown on two lines, the command is entered on one line):

```
wscmpile.sh -classpath toolclass -gen:both -keep
  -d classout -model modelfile config.xml
```

`toolclass`, `classout`, and `modelfile` are representative. Retirement Specialists specifies actual class paths for `toolclass` and `classout`. They specify a file name (and path) for `modelfile`. Based on the options specified in the command, the `wscmpile` tool generates both client-side and server-side artifacts, keeps source files that it generates, and puts the generated classes in the `classout` directory. It also creates and stores the model file in the location identified by `modelfile`.

In the sample application, the `wscmpile` tool is invoked from an XML file. [Here](#) are the contents of the file. Notice the part of the file that invokes the `wscmpile` tool. For example, here's the XML code that invokes the `wscmpile` tool in UNIX for Retirement Specialists:

```
<target name="wscmpile-retirementspecialists"
  if="isNotWindows" >
  <echo message="wscmpiling retirement specialists..." />
  <exec executable="{JWSDP_HOME}/bin/wscmpile.sh">
    <arg line="-classpath ${build}/jaxrpc-providers"/>
```

```

    <arg line="-gen:both"/>
    <arg line="-keep"/>
    <arg line="-d ${build}/jaxrpc-providers"/>
    <arg line="-model"/>
    <arg line="-d ${build}/jaxrpc-providers/rs-model.Z"/>
    <arg line="jaxrpc-providers/config/rs-config-rmi.xml"/>
  </exec>
</target>

```

Here are the contents of the configuration file, `rs-config-rmi.xml`:

```

<?xml version="1.0" encoding="UTF-8"?>
<configuration
  xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
  <service name="RetirementSpecialistsRS"
    targetNamespace="http://retirementsevice.org/wsdl"
    typeNamespace="http://retirementsevice.org/types"
    packageName="com.sun.eportal.retirement"
    <interface name="com.sun.eportal.RetirementServiceIF"
      servantName="com.sun.eportal.retirement.RetirementSpecialistsImpl"/>
  </service>
</configuration>

```

Notice that it includes a `service` element. As a result, the `wscompile` tool will generate a WSDL file from the service endpoint definition identified in the configuration file. [Here](#) is the WSDL file for the Retirement Specialists service.

[Here](#) is the model file created by the `wscompile` tool for Retirement Specialists.

Deploy the Service

Both retirement service providers want to deploy their services in a J2EE-compliant container. (In the sample application, the container is Tomcat.) They also decide to implement the service endpoints for their services as servlets. To deploy Web server-based components such as servlets in a J2EE-compliant container, each service provider needs to create a WAR file. The service providers create their respective WAR files using J2EE tools. However the WAR files are still raw. To make their WAR files deployable, each service provider uses the `wsdeploy` tool. For example, here is a command that Retirement Specialists executes in the UNIX environment to run the `wsdeploy` tool (although shown on three lines, the command is entered on one line):

```

wsdeploy.sh -keep tmpdir tmpdir
-o RetirementSpecialistsRS.war
RetirementSpecialistsRSraw.war

```

`tmpdir` is representative. Retirement Specialists specifies an actual class path for temporary files generated by the `wsdeploy` tool. The `wsdeploy` tool takes the WAR file, `RetirementSpecialistsRSraw.war`, as input. Based on the options specified in the command, the tool generates a deployable WAR file, `RetirementSpecialistsRSraw.war`. Temporary files generated during the process are stored in the specified directory.

Create the Client

BoomingBusiness.com envisions the following flow of events for their retirement planning service. After an employee clicks a "Retirement Services" link in the employee portal:

Click to enlarge each image

1. The employee portal displays a list of available retirement funds

Please look over the list of available funds below.
 Select one or more funds.
 Specify the percentage of investment for each fund you select.
 Specify a monthly investment amount.
 Click Go to confirm.

Funds	Investment Percentage
Sunrise funds	<input type="text"/>
Old age funds	<input type="text"/>
Happy old days funds	<input type="text"/>
Green olive funds	<input type="text"/>
Old orchards funds	<input type="text"/>
Investing Money	\$ <input type="text"/> /Month

2. The employee selects one or more retirement funds, specifies a percentage of money to be invested in each fund, and the total monthly amount of money to be invested.

Please look over the list of available funds below
 Then click go to confirm that this is what you would like to invest in.

Funds	Investment Percentage
Sunrise funds	<input type="text"/>
Old age funds	<input type="text"/>
Happy old days funds	30
Green olive funds	<input type="text"/>
Old orchards funds	70
Investing Money	\$100 /Month

3. Providers that offer all of the selected funds respond with a quote for their services.

Requested Funds

Fund Name	Fund Percent
Old orchards funds	70
Happy old days funds	30
Monthly Investment	100.0

Quotes Received

	Provider	Service Fee(\$)
<input type="radio"/>	Universal Retirement Services	5.0
<input type="radio"/>	International Pension Specialists	6.0

4. The employee selects one of the providers that offers a quote.

Requested Funds

Fund Name	Fund Percent
Old orchards funds	70
Happy old days funds	30
Monthly Investment	100.0

Quotes Received

	Provider	Service Fee(\$)
<input checked="" type="radio"/>	Universal Retirement Services	5.0
<input type="radio"/>	International Pension Specialists	6.0

The selected provider then sends a note to confirm the transaction.

Here's what BoomingBusiness.com's IT staff plans to implement in support of this interaction:

- A link for "Retirement Service" in the home page of their employee portal.
- An ["available funds" JavaServer Page](#) (JSP™) that displays the table of available retirement funds.
- A ["401kbidding" JSP](#) that displays bids from retirement fund providers. When an employee selects one or more retirement funds in the table of available funds, it opens the 401kbidding JSP.
- A ["fund lister" JavaBean](#). The available funds JSP uses the fund lister JavaBean to get an aggregated list of available retirement funds. The 401kbidding JSP uses the fund lister JavaBean to get a quote from each provider that offers all of the selected funds.
- A [DII client class](#). The fund lister JavaBean uses this class to get a list of available funds, and to get quotes from retirement fund providers.

- A "[confirmation](#)" JSP that confirms the transaction. When an employee selects one of the retirement providers that offers a quote, the 401kbidding JSP opens the confirmation JSP.

Let's look at some of these components in more detail.

Available Funds JSP: [Here](#) is the source code for the available funds JSP. Notice the use of the [fund lister JavaBean](#). The JSP invokes the `populateAvailableFunds` method in the fund lister JavaBean to get a list of available funds, and then invokes the `getFundHash` method in the fund lister to create a hashtable (that is, a table that maps keys to values) for the retrieved funds.

```
<jsp:useBean id="fundLister"
  class="com.sun.eportal.FundLister" scope="session" />

  ...

  fundLister.populateAvailableFunds();
  Hashtable fundHash = fundLister.getFundHash();
  ...
```

The JSP then displays the retirement funds in a table. The user selects one or more funds, specifies an investment percentage for each, and a monthly investment amount. After the user makes these selections and specifications, the subsequent action is to invoke the [401kbidding JSP](#):

```
<form action="401kbidding.jsp">
<TABLE border=1>
<TR> <TH> Funds </TH> <TH> Investment Percentage</TH> </TR>

<%

Enumeration keys = fundHash.keys();
while ( keys.hasMoreElements( ) )
{
    String fundName= (String)keys.nextElement( );
    String fundNamePercent = fundName + "Percent";
%>
<TR> <TD>
  ... <%= fundName %> ...</TD>
  <TD> <INPUT TYPE=text name='<%= fundNamePercent %> ...
  </TD> </TR>
<%
}
%>
<TR> <TD ...> <...>Investing Money :</FONT> </TD>
<TD> $<INPUT type="text" name="monthlyInvestment" ...>/Month
</TD> </TR>
...

</TABLE>
```

401kbidding JSP: [Here](#) is the source code for the 401kbidding JSP. Notice that it invokes the `getAllQuotes` method in the fund lister JavaBean, passing the method an array of the selected funds, and a monthly investment amount. The `getAllQuotes` method gets quotes from providers that offer all of the selected funds.

```

<jsp:useBean id="fundLister"
  class="com.sun.eportal.FundLister" scope="session" />
...
Hashtable quoteHash = fundLister.getAllQuotes(
    fundInfoArray, monthlyInvestment );
...

```

The 401kbidding JSP then displays two tables, one that lists the selected funds, and another that lists the providers and their quotes. The user selects a provider and submits it. This invokes the [confirmation JSP](#).

```

<FORM method=post action="confirmation.jsp" >

<FONT ...>Requested Funds </FONT>
<TABLE ...>
<TR><TH>Fund Name</TH> <TH>Fund Percent</TH> </TR>
...
</TABLE>

<TABLE ...>
<TR><TH></TH><TH> End Point </TH>
<TH> Service Fee($)</TH> </TR>
<%
Enumeration keys = quoteHash.keys();
while ( keys.hasMoreElements( ) )
{
  String endpoint = (String)keys.nextElement();
  Double quotedD = (Double)quoteHash.get( endpoint);
  %>
<TR> <TD ...><INPUT TYPE=RADIO
  NAME="endpointRadio" value="<%=endpoint%>"></TD>
  <TD ...> <%= endpoint %> </TD> <TD ...>
  <%= quotedD %>
</TD> </TR>
<%
}
%>
<TR> <TD ...> <INPUT TYPE="Submit" value="Confirm"/>
</TD> </TR>
</TABLE>
</FORM>

```

Confirmation JSP: [Here](#) is the source code for the confirmation JSP. The JSP obtains information about the employee and the employee's retirement fund provider selection. It then calls the `getQuote` method in the [DII client](#) to confirm the transaction.

```

RSClient_DII rsClient = new RSClient_DII();
String resultString = rsClient.confirmQuote(
    endPoint, fundInfoArray, monthlyInvest, ei );

```

After the transaction is confirmed, the confirmation JSP uses a bean to store employee information in persistent storage (typically this would be a database):

```

if (resultString != null )
{
int numRows = dbBean.executeUpdate( "Update ServiceInfo
SET retirementprovider='" + endPoint + "'
WHERE employeeid='" + employeeId + "'" );
    if ( numRows > 0){
        System.out.println("Service Info created successfully");
    }
}

```

The confirmation JSP then sends a confirmation message:

```

Hi <%= firstName %> <%= lastName %> ...
...
You will get confirmation email from <%= endPoint %>
with all the details.
...

Thank you for using EmployeePortal
...

```

Fund Lister JavaBean: [Here](#) is the source code for the fund lister JavaBean. Notice the `populateAvailableFunds` method. This is the method that the fund lister JavaBean uses to search for appropriate providers. The `populateAvailableFunds` method uses a query method in a [client query object](#) to search a Web services registry for providers classified as pension fund (in other words, retirement fund) providers. The parameters passed to the query method comprise the NAICS classification scheme for pension funds.

```

public void populateAvailableFunds ( ) {
    try {
        String cScheme="ntis-gov:naics";
        String keyName="Pension Funds";
        String keyValue="52511";
        String serviceName="Retirement Service";

        ...

        ClientQuery clientQuery = new ClientQuery( );
        Vector serviceProviderInfoVector =
            clientQuery.query( cScheme, keyName, keyValue);

        ...
    }
}

```

After the the query returns information from the registry, the fund lister JavaBean examines the binding information exposed by the candidate providers. If the binding is JAX-RPC, the fund lister JavaBean adds the provider's service endpoint to an array:

```

Vector jaxrpcEndpointVector= new Vector();

for ( int i=0;
      i< serviceProviderInfoVector.size(); i++ ) {
    ServiceProviderInfo spi = (ServiceProviderInfo)
        serviceProviderInfoVector.elementAt(i);
}

```

```

if ( spi.getCommunicationType().equals(
    "JAXRPC" ) ) {
    jaxrpcEndpointVector.addElement(
        spi.getEndpoint() );
}

```

```

}
jaxrpcEndpointArray =
    new String[ jaxrpcEndpointVector.size() ];
jaxrpcEndpointVector.copyInto (
    (Object[])jaxrpcEndpointArray );

```

The fund lister JavaBean then uses a [DII client](#) to get a list of available retirement funds offered by the subset of providers that expose a JAX-RPC binding. It then aggregates the list:

```

RSClient_DII rsclient_DII = new RSClient_DII();
fundHash =rsclient_DII.getAllAvailableFunds (
    jaxrpcEndpointArray );

```

...

The fund lister JavaBean then returns the aggregated list to the available funds JSP for display in a table.

After a user selects one or more funds and specifies investment amounts, the available funds JSP invokes the 401kbidding JSP, which, in turn, invokes the `getAllQuotes` method in the fund lister JavaBean. Here is the source code for the `getAllQuotes` method. It calls the `getAllQuotes` method in the [DII client](#) to get a list of quotes.

```

public Hashtable getAllQuotes (
    FundInfo[] reqFundsInfos, double mi ) {

    RSClient_DII rsclient_DII = new RSClient_DII();
    Hashtable quoteHash =
        rsclient_DII.getAllQuotes (jaxrpcEndpointArray,
            reqFundsInfos, mi );

    return quoteHash;

}

```

Client Query Class: [Here](#) is the source code for the client query class. The class uses JAXR to search a Web services registry. Notice how the query method in the class searches the registry for organizations that meet the supplied classification scheme, and retrieves information about service providers that meet the classification criteria:

```

ClassificationScheme classificationScheme =
    queryManager.findClassificationSchemeByName(
        findQualifiers, cName );
Classification classification =
    lifeCycleManager.createClassification(
        classificationScheme, keyName, keyValue );

```

...

```

BulkResponse response =

```

```

queryManager.findOrganizations(findQualifiers,
    null, classifications, null, null, null);

```

```

Collection orgs = response.getCollection();

```

For more information about using JAXR to search a Web services registry, see [Part 1 of the series](#).

DII Client: [Here](#) is the source code for the DII client class. The fund lister JavaBean calls the `getAllAvailableFunds` method in the class to get an aggregated list of available funds. The method call specifies one argument: the array of endpoint addresses for the providers that meet the selection criteria (that is, pension funds) and that expose a JAX-RPC binding. The `getAllAvailableFunds` method uses the `getAvailableFunds` method to build the list of funds. Notice how the `getAvailableFunds` method creates a serializer and deserializer for serializing Java data types to XML, and XML to Java, respectively:

```

SerializerFactory stringArraySerializerFactory =
    new SingletonSerializerFactory(stringArraySerializer);
DeserializerFactory stringArrayDeserializerFactory =
    new SingletonDeserializerFactory(stringArraySerializer);

```

It then creates the `Service` and `Call` objects for the DII call. (In this example, the local part of the qualified name for the `Service` is explicitly specified, that is, "PanAmericanRS". In general, you would obtain the qualified name from a registry.)

```

ServiceFactory factory = ServiceFactory.newInstance();
Service service = factory.createService(
    new QName("PanAmericanRS"));

```

```

...
Call call = service.createCall();
...

```

The `getAvailableFunds` method then configures the `Call` object.

```

call.setPortTypeName(port);
call.setTargetEndpointAddress(endpointAddress);

call.setProperty(Call.SOAPACTION_USE_PROPERTY, new Boolean(true));
call.setProperty(Call.SOAPACTION_URI_PROPERTY, "");
call.setProperty(ENCODING_STYLE_PROPERTY,
    SOAPConstants.URI_ENCODING);

call.setReturnType(stringArrayTypeQname);
    call.setOperationName(new QName(bodyNamespaceValue,
        "getAvailableFunds"));

```

The `getAvailableFunds` method then gets the available funds. Notice the use of the `invoke` method of the `Call` object. This invokes the specified operation ("getAvailableFunds") in synchronous request-response mode. The DII client will wait until the operation completes before continuing.

```

call.setReturnType(stringArrayTypeQname);
call.setOperationName(new QName(bodyNamespaceValue,
    "getAvailableFunds"));
System.out.println("Invoking available Funds");

```

```
String[] avFunds = (String[])call.invoke(null);
if ( avFunds != null ) {
    for ( int i=0; i< avFunds.length; i++ ) {
        System.out.println(
            " DII -Available Fund[" + i +
            " ]-> " + avFunds[i] );
    }
}
```

Another method of interest in the DII client is `getAllQuotes`. This method is called by the fund lister JavaBean to get quotes from providers that offer all of the funds that the user selected. The fund lister JavaBean invokes the method with three arguments: the array of endpoints for retirement fund providers, the selected funds, and the monthly investment. The `getAllQuotes` method uses the `getQuote` method to get the quotes. The `getQuote` method uses the `newBeanCall` object to create the `Service` and `Call` objects for the DII call, and the `newCall` object to set the port type and endpoint address for the `Call` object. The `getQuote` method then specifies the operation for getting the quotes (`getQuote`), and parameters for the operation. It then invokes the operation:

```
Call call = newBeanCall(endpointAddress,  RETIREMENT_SERVICE,
                        RETIREMENT_BODY_NAMESPACE, RETIREMENT_PORT );
...

call.setOperationName(new QName(RETIREMENT_BODY_NAMESPACE,
                                "getQuote"));
call.addParameter( "arrayOfFundInfo_1",  fundInfoArrayTypeQname,
                  ParameterMode.IN );
call.addParameter( ""double_2",  QNAME_TYPE_DOUBLE,
                  ParameterMode.IN );

Object[] params = new Object[2];
params[0] = reqFundInfos;
params[1] = new Double( mi );

quoteObject = (Double)call.invoke( params );
...
```

Notice also the `confirmQuote` method. This method is called by the confirmation JSP to confirm the transaction between the employee and the retirement fund provider. The confirmation JSP invokes the `confirmQuote` method with four arguments: the endpoint of the selected retirement fund provider, an array that contains information about the selected funds, the monthly investment amount that the employee specified, and information about the employee. As is the case for the `getQuote` method, the `confirmQuote` method uses the `newBeanCall` object to create the `Service` and `Call` objects for the DII call, and the `newCall` object to set the port type and endpoint address for the `Call` object. The `confirmQuote` method then specifies the operation for confirming the transaction (`confirmQuote`), and parameters for the operation. It then invokes the operation:

```
Call call = newBeanCall(endpointAddress,  RETIREMENT_SERVICE,
                        RETIREMENT_BODY_NAMESPACE, RETIREMENT_PORT );
...

call.setOperationName(new QName(RETIREMENT_BODY_NAMESPACE,
                                "confirmQuote"));
call.addParameter( "arrayOfFundInfo_1",  fundInfoArrayTypeQname,
                  ParameterMode.IN );
call.addParameter( ""double_2",  QNAME_TYPE_DOUBLE,
                  ParameterMode.IN );
```

```
call.addParameter( "EmployeeInfo_3", employeeInfoQname,
                  ParameterMode.IN );
```

```
Object[] params = new Object[3];
params[0] = reqFundInfos;
params[1] = new Double( mi );
params[2] = ei;
```

```
confirmationMessage = (String)call.invoke( params );
...
```

Java™ Web Services Developer Pack

Part 2: RPC Calls, Messaging, and the JAX-RPC and JAXM API

JAXM

JAXM is a Java API for XML-based messaging. In XML-based messaging, a client sends an XML document to the server for processing. What's important in this context is that both the client and server recognize the

JAXM is a Java API for SOAP-based XML (document-oriented) messaging.

format and meaning of the document. For example, the document could be something that both the client and server

recognize as a purchase order. The server processes the purchase order, and could return a response (perhaps confirming that the purchase order was received). JAXM is focused on business-to-business messaging. In other words, it's designed for the exchange of documents, such as purchase orders, that are typically used in business operations.

If you read the section on [JAX-RPC](#), you know that a JAX-RPC client calls a specific method on the server. In contrast to a JAX-RPC client, a messaging client does not call a specific method -- in the purchase order example, the messaging client does not call a specific purchase order method. In fact, the client might not know what program on the server actually processes the request. Instead, the client relies on the server to invoke the appropriate application to process the request because the server understands the document it receives. Similarly, what the server returns is not a method response, but rather a response generated by the processing application.

JAXM works in conformance with [SOAP](#), specifically the [SOAP 1.1 specification](#) and the [SOAP 1.1 with Attachments specification](#). What this means is that in a JAXM scenario, documents and responses are [SOAP messages](#) or [SOAP messages with attachments](#). What it also means is that JAXM implementations must support the SOAP 1.1 and SOAP 1.1 with Attachments specifications. However a [JAXM provider](#) can also support messaging protocols (usually industry-based) that are built on top of SOAP 1.1 and SOAP 1.1 with Attachments, for example, the [ebXML Message Service Specification V1.0](#).

Like JAX-RPC, JAXM is designed to hide the complexity of SOAP. When you use JAXM, you don't explicitly code a SOAP request. Instead you code the request in the Java programming language, using a Java API. JAXM converts the request to a SOAP message (or SOAP message with attachments) and then transports it to the server. The server converts the SOAP message (with possible attachments) and then processes it. Then the sequence is reversed. The server converts the response to a SOAP message (possibly with attachments) and transports it back to the client.

In order to understand and use JAXM, it's important to understand some basic concepts. These concepts include:

- [Clients and Providers](#)
- [Message Exchanges](#)
- [Connection and Message Objects](#)

After you're familiar with these concepts, see the [JAXM Example](#).

Clients and Providers

There are two roles that can take part in JAXM messaging: clients and providers. A JAXM client is necessary for messaging, a JAXM provider is optional. A JAXM client is an application that sends messages using the JAXM API. If a provider isn't used, the client sends the message to a recipient (typically a service) on a server, identified by a URL.

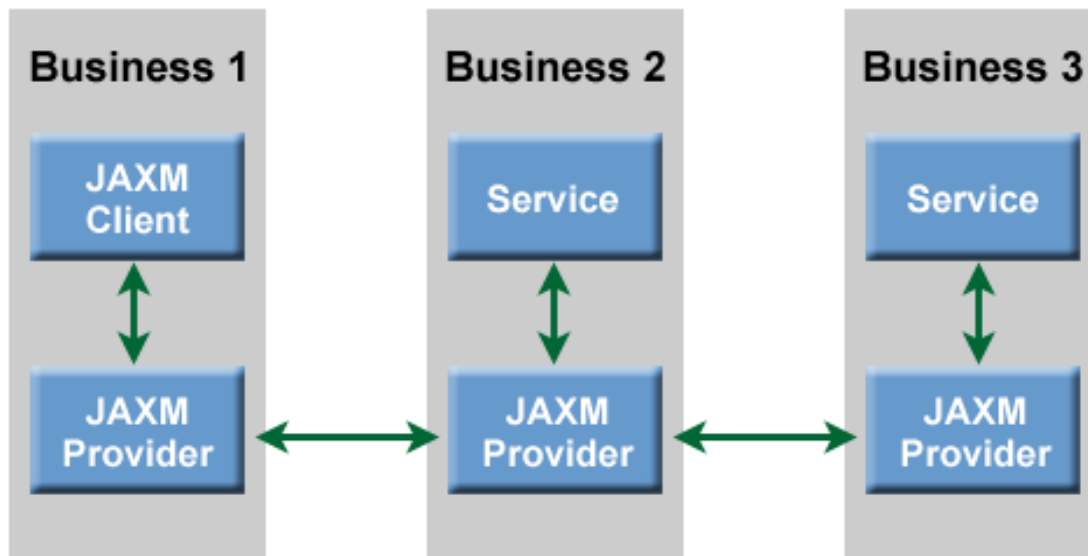
A JAXM client is necessary for messaging, a JAXM provider is optional.



A JAXM provider (also called a messaging provider) is an intermediate service that acts on behalf of a client. If a JAXM provider is used, the client sends the message to the provider, which then transmits and routes the message to the service. To do this, a JAXM provider implements and supports the JAXM API.



A client or client-provider pair can also receive messages, in other words, it can be a target service of a message. For example, suppose three businesses use JAXM to exchange purchase orders. One business, Business 1, sends purchase orders to a service provided by a second business, Business 2, which in turn, sends the purchase order to a service provided by a third business, Business 3. Business 2's service acts as a service when it receives the purchase order from Business 1, and acts as a client when it sends the purchase order to Business 3. If a JAXM provider is used in this scenario, Business 1's JAXM client sends the purchase order to its JAXM provider, which transmits the purchase order to Business 2's JAXM provider. Business 2's JAXM provider sends the purchase order to its service. The service then uses Business 2's JAXM provider to transmit the purchase order to Business 3.



The choice of whether or not to use a provider has implications for deployment, level of service, and routing:

- Where you deploy a client.** If you use a provider, you must deploy the client in a J2EE Web container or in a J2EE Enterprise JavaBeans™ (EJB™) container. If you don't use a provider you don't have to deploy the client in a J2EE container. You could deploy it, for example, as an application on the J2SE™ platform. A JAXM client that isn't deployed in a J2EE container is called a standalone client. A standalone client can act only as a client -- it can't also act as a service.
- The level of service provided to messages.** As mentioned earlier, JAXM supports additional messaging protocols that are built on top of SOAP 1.1 and SOAP 1.1 with Attachments. For example, JAXM supports the ebXML Message Service Specification V1.0. In JAXM terminology, these additionally-supported protocols are called profiles, and the support is provided by a JAXM provider (but not a JAXM client). A JAXM provider can support one or more profiles (or no profiles). In supporting a profile, the JAXM provider makes available the additional levels of service offered by the associated protocol. Note that a JAXM client can only take advantage of one profile at a time when it sends a message to a JAXM provider -- even if the provider supports more than one profile. In fact, the client and provider must agree on the profile before the client sends the message.
- Additional routing.** If a JAXM provider isn't used, a JAXM client sends a message directly to a service. If a JAXM provider is used, the message is routed through the provider to the service. However a JAXM provider allows for even more intricate routing. For example, a message can be routed to a number of intermediate destinations (for perhaps some intermediate level of processing) before routing to a final destination.

If you use a provider, you must deploy the client in a J2EE Web container or in a J2EE Enterprise JavaBeans container. If you don't use a provider you can deploy a standalone client.

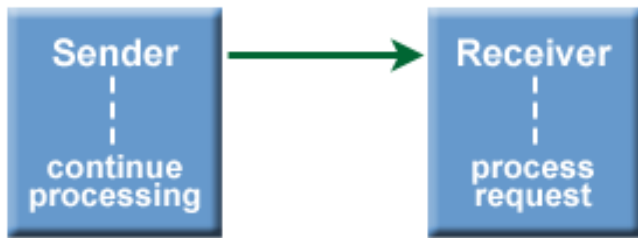
Message Exchanges

JAXM supports various types of exchanges between a JAXM client and a service, or between an JAXM client-JAXM provider pair that sends a message (the JAXM specification calls a client-provider pair that takes this role a sender) and a JAXM client-JAXM provider pair that receives a message (the JAXM specification calls a client-provider pair that takes this role a receiver). First, it supports two categories of exchanges: asynchronous and synchronous.

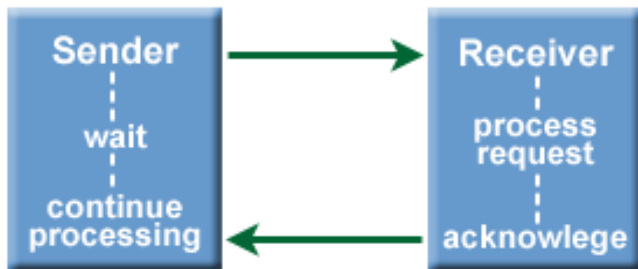
JAXM supports asynchronous (that is, one-way) and synchronous (that is, two-way) exchanges between a sender and a receiver.

An asynchronous exchange is a one-way exchange between a sender and receiver. (Notice that client and service is not

mentioned here. If a JAXM provider is not used, a JAXM client can only have a synchronous exchange with a service.) In an asynchronous exchange, the sender sends a message to the receiver without waiting for a response. The sender can then continue processing. When the receiver receives the message, it must process it.



A synchronous exchange is a two-way exchange (also called a request-response exchange) between a client and a service or a sender and receiver. In a synchronous exchange, the client or sender sends a message to the receiver and then waits for a response. In other words the client or sender is blocked, it can't continue processing until the service or receiver replies.



In addition, JAXM supports a number of request types within these categories:

- **Asynchronous inquiry.** In this exchange, a sender uses JAXM to ask a receiver for information. Because it's an asynchronous request, the sender can continue processing -- it's not blocked. The receiver reads and processes the request and returns a reply message. Sending the reply message is a totally separate operation and can happen long after the request is sent.
- **Asynchronous update with acknowledgment.** In this exchange, a sender uses JAXM to ask a receiver to update information. Again, because this is an asynchronous request, the sender can continue processing -- it's not blocked. The receiver reads and processes the request. When it successfully completes the processing, the receiver sends the client an acknowledgment that it made the update.
- **Synchronous update.** Unlike an asynchronous update, a synchronous update forces the client or sender to wait for an acknowledgment. The acknowledgment, which is correlated to the request, indicates that the update was successfully made. The acknowledgment unblocks the client or sender.
- **Synchronous inquiry.** This exchange is a variation of the synchronous update exchange. The one difference is that in this exchange the service or receiver returns a reply message (rather than acknowledgment message). The reply message has no correlation to the request. Its only purpose is to unblock the client or sender.
- **Fire and forget.** This exchange does not involve an acknowledgment or reply. The sender makes the request and

continues processing.

Profiles

A JAXM provider can offer messaging functionality that extends what's offered in support of the SOAP 1.1 and SOAP 1.1 with Attachments specifications. For example, a JAXM provider might support a specification such as the ebXML Message Service Specification, [ebXML Transport, Routing & Packaging Version 1.0](#), which adds transport, routing, and packaging features beyond the basic messaging features specified in the SOAP 1.1 and SOAP 1.1 with Attachments specifications. In this case, the specification documents specifics regarding the content of the SOAP header and the content of other SOAP elements that is not covered in the basic SOAP model. It's important to note that although this ebXML specification extends the SOAP specifications, it is also based on the SOAP specifications. In fact, any additional functionality that a JAXM provider offers must be based on the SOAP specifications. In JAXM, the extended functions that a JAXM provider offers are represented by a profile. In other words, a JAXM provider that supports the ebXML Transport, Routing & Packaging Version 1.0 specification, supports a profile for that specification. Support for a profile is required for asynchronous messaging.

A JAXM provider can support multiple profiles. Each profile can represent a particular set of extensions to basic SOAP messaging that are agreed on by a particular industry or standards body. Typically, these extensions cover security and quality-of-service features not covered in the SOAP 1.1 and SOAP 1.1 with Attachments specifications. However, a profile does not have to be industry-based or standards body-based. It can be application-specific.

Profiles play a part in creating a message for exchange with a JAXM provider. Before a JAXM client creates a message, it needs to identify a profile for the exchange. This makes the extended functions associated with the profile available to the exchange. JAXM provides a mechanism that a client can use to determine which profiles a provider supports. After determining what profiles are supported, the client then specifies one of the supported profiles when it creates a message for transmission to the provider.

Connection and Message Objects

In JAXM, SOAP messages are sent between a client (or sender) and a service (or receiver) over a connection. A connection is either point-to-point, that is, directly from a client to a destination such as a URL or a service, or client-to-JAXM provider. A point-to-point connection is represented by a `SOAPConnection` object. A connection to a JAXM provider is represented by a `ProviderConnection` object. In other words, you use a `SOAPConnection` object in a synchronous, request-response exchange between standalone JAXM client and a service. You use a `ProviderConnection` object in an asynchronous exchange between a sender and receiver.

A SOAP message is represented by a `SOAPMessage` object. The object models the [structure of a SOAP message with attachments](#). The `SOAPMessage` object is a container that holds other objects that model the parts of a SOAP message with attachments, that is, the root body part and attachment parts (if there are attachments in the message). In particular, a `SOAPMessage` object contains the following object hierarchy:

- `SOAPPart`
 - `SOAPEnvelope`
 - `SOAPHeader`
 - `SOAPBody`
- `AttachmentPart`
 - `MimeHeaders`
 - `MimeHeader`

JAXM Packages

The JAXM 1.1 API consists of a single package, `javax.xml.messaging`. However the JAXM 1.1 specification also depends on another package, `javax.xml.soap`, that is defined by the [SOAP with Attachments API for Java \(SAAJ\)](#)

[1.1 specification](#). Originally, both packages were part of JAXM API, however the `javax.xml.soap` package was decoupled from the JAXM API so that other specifications, such as JAX-RPC 1.0, could depend on this package without having to depend on the rest of the JAXM specification.

javax.xml.soap. This package provides the basic building blocks for creating a SOAP message, and for sending it synchronously, that is, in a request-response exchange. For example, if you use a standalone JAXM client to request a synchronous update, you use the `javax.xml.soap` package. Some of the important classes in the package are:

SOAPConnectionFactory

This is an abstract base class for factory classes that create a synchronous, point-to-point connection. Two important methods in the class are `newInstance`, which is used to create an instance of a default `SOAPConnectionFactory` object, and `createConnection`, which creates a `SOAPConnection` object.

MessageFactory

This is an abstract base class for factory classes that create a `SOAPMessage` object. Two important methods in the class are `newInstance`, which is used to create an instance of a default `MessageFactory` object, and `createMessage`, which creates a `SOAPMessage` object.

SOAPPart

This is a container class for the `SOAPEnvelope` object. An important method in the class is `getEnvelope`, which is used to get the `SOAPEnvelope` object contained in the `SOAPPart` object.

SOAPEnvelope

This is a container class for the `SOAPHeader` and `SOAPBody` objects. Some of the important methods in the class are `getHeader`, which is used to get the `SOAPHeader` object, `getBody`, which is used to get the `SOAPBody` object, and `createName`, which is used to create a `Name` object needed for qualifying and localizing names in the SOAP message.

SOAPHeader

This class represents a SOAP Header element, in other words, it adds information beyond what is in the body of the SOAP message. For example, it can indicate additional processing that needs to be done at an intermediate node along the path to the final destination. A SOAPEnvelope object contains an empty SOAPHeader object by default. However, a SOAPHeader object is optional. If the SOAPHeader object is not needed, the method detachNode can be used to remove the object.

SOAPBody

This class represents a SOAP Body element. The class can contain SOAPBodyElement objects that model the content of the body of a SOAP message. An important method in the class is addBodyElement, which is used to create a SOAPBodyElement object, and add it to a SOAPBody object

SOAPBodyElement

This class represents content within a SoapBody object. The set of SOAPBodyElement objects added to a SoapBody object models the full content in the body of a SOAP message. An important method in the class is addChildElement, which adds a child element to a SOAPBody object

AttachmentPart

This class represents an attachment to a SoapMessage object. The content of the attachment can be XML or some other content type such as an image. A SoapMessage object can contain multiple AttachmentPart objects. Each AttachmentPart object contains the content of the attachment, and MIME headers that identify the type of content. Two important methods in the class are addMIMEHeader, which adds a MIME header to an AttachmentPart object, and setContent, which sets the content of the AttachmentPart object.

javax.xml.messaging. This package provides the API for using a JAXM provider to send and receive SOAP messages. The API is also used by a recipient to process a SOAP message and return a reply -- even if a JAXM provider is not used in the exchange with a client. When you send a SOAP message using a JAXM provider, you still need to create the SOAP message, so you need to use the javax.xml.soap package in addition to the javax.xml.messaging package. For example, if you use a JAXM provider to make an asynchronous inquiry, you need the javax.xml.soap package to create the SOAP message and the javax.xml.messaging package to use

the JAXM provider. Some of the important classes and interfaces in the package are:

ProviderConnectionFactory	This is an abstract base class for factory classes that create a connection to a JAXM provider. Two important methods in the class are <code>newInstance</code> , which is used to create an instance of a default <code>ProviderConnectionFactory</code> object, and <code>createConnection</code> , which creates a <code>ProviderConnection</code> object.
ProviderConnection	This class represents an active connection to a JAXM provider. Two important methods in the class are <code>getMetaData</code> , which is used to get a <code>ProviderMetaData</code> object that contains information about a JAXM provider, and <code>createMessageFactory</code> , which creates a <code>MessageFactory</code> object for sending <code>SOAPMessage</code> objects for a specified provider profile.
ProviderMetaData	This class contains information about a JAXM provider. An important method in the interface is <code>getSupportedProfiles</code> , which is used to retrieve a list of the profiles supported by the provider.
ReqRespListener	This is an interface that is implemented by a recipient in a synchronous message exchange with a client. The interface defines one method, <code>onMessage</code> , that is used to pass a <code>SOAPMessage</code> to the <code>ReqRespListener</code> implementation, and to return a response.
OnewayListener	This is an interface that is implemented by a recipient in an asynchronous message exchange with a client. The interface defines one method, <code>onMessage</code> , that is used to pass a <code>SOAPMessage</code> to the <code>OnewayListener</code> implementation, and to return a response.

Let's see how these classes are used to exchange messages. First let's look at how the classes are used by a JAXM client that doesn't use a JAXM provider, then let's look at how the classes are used by a JAXM client that does use a JAXM

provider.

Exchanging Messages Without a JAXM Provider

A standalone JAXM client, that is one that doesn't use a JAXM provider, can only exchange messages synchronously in a request-response exchange with a recipient such as a service. Here are the steps in this kind of exchange.

The JAXM client:

- [Gets a connection](#)
- [Creates a message](#)
- [Adds content to the message](#)
- [Adds Attachments \(if any\)](#)
- [Sends the message to the service \(and waits\)](#)

The service then [processes the message](#) and returns a reply.

The client then [processes the reply](#).

Get a Connection

A standalone JAXM client gets a connection by creating a `SOAPConnection` object. This object represents a connection between the client and the service. The client uses the `newInstance` method of the static `SOAPConnectionFactory` class to create an instance of the class. (The SAAJ API provides a default implementation of this class.) The client then uses the `createConnection` method of that instance to create a `SOAPConnection` object.

```
import javax.xml.soap.*;

SOAPConnection conn;
SOAPConnectionFactory scf =
    SOAPConnectionFactory.newInstance();
conn = scf.createConnection();
```

Create a Message

A standalone JAXM client creates a message by creating a `SOAPMessage` object. The client uses the `newInstance` method of the static `MessageFactory` class to create an instance of the class. (The SAAJ API provides a default implementation of this class.) The client then uses the `createMessage` method of that instance to create the `SOAPMessage` object.

```
MessageFactory mf = MessageFactory.newInstance();
SOAPMessage msg = mf.createMessage();
```

Add Content to the Message

A `SOAPMessage` object represents all the elements of a SOAP message, but the elements initially have no content. To add content to the SOAP message, a JAXM client needs to access the appropriate element in the message (that is, the body and optionally the header), and then add content to that element. To access the header and body elements, the client first uses the `getSOAPPart` method of the `SOAPMessage` class to get a `SOAPPart` object. The `SOAPPart` object contains a hierarchy of objects that represents the envelope, header, and body of the message. After getting the `SOAPPart` object, the client successively gets the objects in hierarchy using the appropriate get methods (`getEnvelope`, `getHeader`, `getBody`).

```
SOAPPart sp = msg.getSOAPPart() ;
SOAPEnvelope envelope = sp.getEnvelope();
SOAPHeader hdr = envelope.getHeader();
SOAPBody bdy = envelope.getBody();
```

To add content to the header, a client:

- Uses the `createName` method of the `SOAPEnvelope` object to name a `SOAPHeaderElement` object
- Uses the `addHeaderElement` method of the `SOAPHeader` object to add the `SOAPHeaderElement` object to the `SOAPHeader` object
- Uses the appropriate `add` method of the `SOAPHeaderElement` object to add content to the header element

For example, the following code adds a `SOAPHeaderElement` object named `OnlineBooks` to a `SOAPHeader` object, and adds text to the `SOAPHeaderElement` object. The text is "Online Book Orders":

```
hdr.addHeaderElement( envelope.createName(
    "OnlineBooks", "ob",
    "http://www.bookprovider.com" ) ).addTextNode(
    "Online Book Orders");
```

To add content to the body of the message, a client adds one or more child elements to the body, and then adds content to each child element. To do this, the client:

- Uses the `addBodyElement` method of the `SOAPBody` object to create a `SOAPBodyElement` object
- Uses the `createName` method of the `SOAPEnvelope` object to name the `SOAPBodyElement` object
- Uses the `createName` method of the `SOAPEnvelope` object to name a `SOAPElement` object that represents a child element
- Uses the `addChildElement` method of the `SOAPBodyElement` object to add the child element to the `SOAPBodyElement` object
- Uses the appropriate `add` method of the `SOAPBodyElement` object to add content to the child element

For example, the following code creates a `SOAPBodyElement` object named `GetBookDetails`, adds two child elements to it named `searchCriteria` and `searchValue`, and adds text to each child element. The text for the `searchCriteria` child element is "author", and the text for the `searchValue` child element is "Hemmingway":

```
SOAPBodyElement sbe = bdy.addBodyElement
( envelope.createName( "GetBookDetails", "bp",
    "http://www.bookprovider.com" ) );
sbe.addChildElement( envelope.createName(
    "searchCriteria", "bp",
    "http://www.bookprovider.com" ) ).addTextNode( "author" );
sbe.addChildElement( envelope.createName(
    "searchValue", "bp",
    "http://www.bookprovider.com" ) ).addTextNode( "Hemmingway" );
```

The name specified for header and body elements must be a local name. The `SOAPEnvelope` class provides a method, `createName`, to create the name. Specifically, the method creates a `Name` object that is initialized with the local name. The SAAJ API provides two signatures for the `createName` method. In one signature, the method is specified with a single parameter: a string that specifies a local name. In the other signature, the method is specified with three parameters: a string that specifies a local name, a string that specifies a prefix for the namespace to be used for the name, and a URI for the namespace (this is the signature that is shown in the previous examples).

Add Attachments (If Any)

Suppose a JAXM client wants to send an image as part of the message? The SOAP part of a message (as represented by the `SOAPPart` object) can only contain XML content. To include non-XML content such as an image, the client needs to provide it as an attachment (however, an attachment can also contain XML content). To add an attachment to a message, the client:

- Uses the `createAttachmentPart` method of the `SOAPMessage` object to create an `AttachmentPart` object that represents an attachment (a SOAP message can have multiple attachments, each requires its own `AttachmentPart` object)
- Uses the `setContent` method of the `AttachmentPart` object to set the content of the attachment (this also sets the `Content-Type` header for the attachment)
- Optionally, adds additional headers for the attachment using appropriate methods in the `AttachmentPart` object
- Uses the `addAttachmentPart` method of the `SOAPMessage` object to add the `SOAPAttachment` object to the `SOAPMessage` object

For example, the following code adds an attachment whose content is a JPEG image:

```
AttachmentPart ap = msg.createAttachmentPart();
byte[] jpegData = "...";
ap.setContent(new ByteArrayInputStream(jpegData), "image/jpeg");
msg.addAttachmentPart(ap);
```

Notice that the `addAttachmentPart` method is specified with two parameters. The first parameter is an object that contains the content for the attachment. The second parameter is a string that specifies the `Content-Type` header for the attachment.

There's another way to add an attachment. In this alternate approach, a JAXM client uses a `URL` object that is part of the `java.net` package, and a `DataHandler` object that is part of the [JavaBeans™ Activation Framework](#). Specifically, the client:

- Creates a `URL` object that represents a URL that provides the content for the attachment
- Creates a `DataHandler` object that references the `URL` object
- Uses the `createAttachmentPart` method of the `SOAPMessage` object to create an `AttachmentPart` object -- this method specifies the `DataHandler` object as a parameter
- Optionally, adds additional headers for the attachment using appropriate methods in the `AttachmentPart` object
- Uses the `addAttachmentPart` method of the `SOAPMessage` object to add the `SOAPAttachment` object to the `SOAPMessage` object

For example, the following code uses a `URL` object and a `DataHandler` object to add an attachment that is a JPEG image:

```
import java.net.*;
import javax.activation.*;

URL url = new URL("http://mysite.com/mybooks.jpg");
AttachmentPart ap =
    msg.createAttachmentPart(new DataHandler(url));
ap.setContentType("image/jpeg");
msg.addAttachmentPart(ap);
```

Send the Message (and Wait)

A standalone JAXM client uses the `call` method of the `SOAPConnection` object to send a message. The `call` method is specified with two parameters: a `SOAPMessage` object that represents the SOAP message, and an object that represents the destination of the message (this is typically a URL).

For example, the following code sends the SOAP message represented by the `message` object to a URL destination ("`http://www.bookstogo.com/bookordering`"):

```
java.net.URL urlEndpoint = new URL(
    "http://www.bookstogo.com/bookordering");
SOAPMessage reply = conn.call(message, urlEndpoint)
```

Notice that the `call` method returns a `reply` that is represented by a `SOAPMessage` object. Because this is a synchronous exchange, the client waits until it receives the reply. In other words, the reply unblocks the client.

Process the Message

After a standalone JAXM client sends a message to a service, the service processes the message. To process the message in a synchronous exchange, a service must implement the `ReqRespListener` interface in the

`javax.xml.messaging` package. For example, suppose that the service is implemented as a servlet. The following statement declares that the servlet implements the `ReqRespListener` interface:

```
import javax.xml.messaging.*;

public class FunBooks extends JAXMServlet implements ReqRespListener
```

Notice the `JAXMServlet` specification. This is a superclass for servlets that receive JAXM messages.

The `ReqRespListener` interface declares one method, `onMessage`, that is used to describe how to process a request. The method takes a `SOAPMessage` object as a parameter. When the method is called, it passes the `SOAPMessage` object to the `ReqRespListener` implementation, and returns a response. The implementation of the `onMessage` method describes how the service processes a request. The service first gets the pertinent elements of the message, that is, the envelope, body, and child elements. To do that, the service takes actions that are similar to what the standalone client does to build the message. Specifically, the service uses the `getSOAPPart` method of the `SOAPMessage` class to get the `SOAPPart` object. The service then uses appropriate `get` methods to get the envelope, header, body, and child elements (`getEnvelope`, `getHeader`, `getBody`, `getChildElements`):

```
public SOAPMessage onMessage(SOAPMessage message) {

    SOAPEnvelope menv = message.getSOAPPart().getEnvelope();
    SOAPBody sb = menv.getBody();
    Iterator sbes = sb.getChildElements(menv.createName
        ("GetBookDetails", "bp",
        "http://www.bookprovider.com"));
```

The `getChildElements` method returns an `Iterator` object that represents all the child elements associated with the `Name` object that is specified as a parameter. In this example, the `Name` object is `GetBookDetails`. The service then iterates over the `Iterator` object to access each child element, and uses the `getValue` method to get the content of each child element. For example, the following code accesses the child elements in the body of a SOAP message, and gets the content in each child element:

```
while ( sbes.hasNext() ) {
    SOAPBodyElement sbe = (SOAPBodyElement)sbes.next();
```

```

Iterator scIterator =sbe.getChildElements(
    menv.createName("searchCriteria", "bp",
        "http://www.bookprovider.com" ));
if ( scIterator.hasNext() ) {

    SOAPElement child =
        (SOAPElement)scIterator.next();
    searchCriteria=child.getValue();
} else {
    System.err.println("ERROR"+
        " Returning null" );
    return null;
}

```

The service continues processing based on the content of the message. For example, suppose the service is an online book ordering service, and the content in the child elements of the SOAP message sent to the service represent a search criterion (such as author), and a search value (such as Hemmingway). After getting the content of the child elements, the service might search a list to find books that meet the search criteria. If it finds a match, the service might then extract additional information from the list about those books. (The logic for doing the search is not shown here.)

The service then builds the reply message. To do that, it creates a `SOAPMessage` object to represent the reply message, and accesses the appropriate elements of the reply message (the envelope, body, and optionally the header). It then adds content to the reply message by adding one or more child elements to the body, and then adding content to each child element. The service can also add content to the header by adding a header element, and adding content to the header element. For example, the following code builds a reply message that contains the title of a book. The code also adds text to the header that identifies the organization providing the book:

```

SOAPMessage msg = fac.createMessage();
SOAPEnvelope env = msg.getSOAPPart().getEnvelope();
SOAPHeader header = env.getHeader();
header.addHeaderElement( env.createName(
    "organization", "bp",
    "http://www.bookprovider.com" ) ).
    addTextNode("Fun Books" );

env.getBody() .addChildElement(
    env.createName("Response" ))
    addTextNode("Reply: "+bookTitle);

```

Process the Reply

To process the reply from a service, a JAXM client needs to first get the pertinent elements of the reply message, that is, the envelope, body, and child elements. The client gets the elements in the same way as the service gets the pertinent elements in a SOAP message. The client uses the `getSOAPPart` method of the `SOAPMessage` class to get the `SOAPPart` object, and uses appropriate `get` methods to get the envelope, header, body, and child elements (`getEnvelope`, `getHeader`, `getBody`, `getChildElements`):

```

SOAPEnvelope envelope =
    reply.getSOAPPart().getEnvelope();
SOAPHeader header = envelope.getHeader();
SOAPBody sb = env.getBody();

```

The client then uses the `Iterator` object returned by the `getChildElements` method to access each child element, and uses the `getValue` method to get the content of each child element. For example, the following code accesses the child elements in the body of a SOAP message, and gets the content in each child element:

```
Iterator bpIterator = sb.getChildElements(
    envelope.createName(localname, "bp",
        "http://www.bookprovider.com"));
if ( bpIterator.hasNext() ) {
    SOAPBodyElement sbe = (SOAPBodyElement)bpIterator.next();
    String bookTitle = bodyElement.getValue();
    System.out.print("Book Title: "+bookTitle);
} else {
    System.err.println("ERROR"+
        " Returning null" );
    return null;
}
```

Exchanging Messages With a JAXM Provider

A JAXM client that uses a JAXM provider can exchange messages asynchronously as well as synchronously with a recipient such as a service, or with a service/provider pair (that is, a receiver). Here are the steps in this kind of exchange.

Just like a standalone client, a JAXM client that uses a JAXM provider:

- [Gets a connection](#)
- [Creates a message](#)
- [Adds content to the message](#)
- [Adds Attachments \(if any\)](#)
- [Sends the message](#)

However instead of getting a synchronous, point-to-point, connection with the service or receiver, the JAXM client gets a connection to a JAXM provider. The client uses that connection to send a message to the provider, which then transmits and routes the message to the service, or to the receiver (which then forwards the message to its paired service). The receiving service then [processes the message](#).

If the client exchanges the message asynchronously, it can continue processing after it sends the message (it does not have to wait for a reply). If the client exchanges the message synchronously, the client waits for the reply. The service then returns the reply to the JAXM provider (in the asynchronous case, this can be long after the request was sent by the client). If the service is part of a provider/service pair, it returns the reply to its paired JAXM provider, which then transmits it to the JAXM provider paired with the client. The provider forwards the reply to the client. The client then [processes the reply](#).

Get a Connection With a JAXM Provider

A JAXM client gets a connection with a JAXM provider by creating a `ProviderConnection` object. There are two ways to do this. One way is similar to the way a standalone client gets a connection: that is, it uses the `newInstance` method of a static connection factory class to create an instance of the class. The connection factory class for connection to a JAXM provider is `ProviderConnectionFactory`. The instance of the `ProviderConnectionFactory` class connects to a default provider implementation. The client then uses the `createConnection` method of that instance to create a `ProviderConnection` object.

```
import javax.xml.messaging.*;
```

```

ProviderConnection pc;
ProviderConnectionFactory pcf =
    ProviderConnectionFactory.newInstance();
pc = pcf.createConnection();

```

The second way to get a connection depends on the JAXM provider being registered with a naming service based on the [Java Naming and Directory Interface™ \(JNDI\)](#). Registering the provider associates it with a logical name. To get the connection to the registered provider, the JAXM client uses JNDI's lookup method, passing it the logical name of the provider. The method returns a `ProviderConnectionFactory` object. The client then uses the `createConnection` method of the returned `ProviderConnectionFactory` instance to create a `ProviderConnection` object. Here, for example, a JAXM client gets a connection to a JAXM provider whose logical name is `bookprovider`:

```

import javax.xml.messaging.*;
import javax.naming.*;

Context ctx = new InitialContext();
ProviderConnectionFactory pcf =
    (ProviderConnectionFactory)ctx.lookup("bookprovider");
ProviderConnection pc = pcf.createConnection();

```

For this approach to work, the logical name of the provider must be specified to the container when it's deployed. Remember, if you use a provider, you must deploy the client in a J2EE Web container or in a J2EE Enterprise JavaBeans container.

Create a Message

As is the case for a standalone client, a JAXM client that uses a provider creates a message by producing a `MessageFactory` instance and then using it to generate a `SOAPMessage` object. However the client in the provider case uses the `createMessageFactory` method of the `ProviderConnection` object to produce the `MessageFactory` instance.

Before a JAXM client creates a message, it needs to identify a [profile](#). It does this when it creates a `MessageFactory` instance. Information about a provider, such as its name and the profiles the provider supports, is contained in a `ProviderMetaData` object. A JAXM client uses the `getMetaData` method of the `ProviderConnection` object to create a `ProviderMetaData` object for a specific connection to a provider. The client then uses the `getSupportedProfiles` method of the `ProviderMetaData` object to identify the profiles supported by the provider. For example, the following code identifies the profiles supported by a JAXM provider associated with the `ProviderConnection` object, `pc`:

```

ProviderMetaData metaData = pc.getMetaData();
metaData.getSupportedProfiles();

```

The client then creates a `MessageFactory` instance, and specifies a `String` argument for a specific profile. For example, here a client determines whether the connected-to provider supports the ebXML Message Service Profile, and then identifies that profile in creating a `MessageFactory` object:

```

String profile = null;

for(int i=0; i < supportedProfiles.length; i++) {
    if(supportedProfiles[i].equals("ebxml")) {
        profile = supportedProfiles[i];
    }
}

```

```
break;
```

```
    }
}
MessageFactory mf = pc.createMessageFactory(profile);
```

After creating a `MessageFactory` instance, the client uses the `createMessage` method of that instance to create the `SOAPMessage` object. Because the `MessageFactory` is initialized with a profile, the `SOAPMessage` object it generates needs to be cast to an implementation of that profile. The Java WSDP includes a basic ebXML profile implementation, `EbXMLMessageImpl`. The following code creates a `MessageFactory` instance, using the basic ebXML profile implementation:

```
EbXMLMessageImpl ebxmlMsg =
    (EbXMLMessageImpl)mf.createMessage();
```

Add Content to the Message

A client that uses a JAXM provider adds content to a `SOAPMessage` object in the same way as a standalone client. However, because the JAXM provider must support a profile, the client can use methods in the profile implementation to add content to a header. The client can use these methods instead of, or in addition to, methods provided in the `javax.xml.soap` package for adding header content. For example, the basic ebXML profile implementation, `EbXMLMessageImpl`, provides the method `setSender` to identify the location of the client, and `setReceiver` to identify the endpoint address of the recipient in a message exchange. The methods create `Party` objects that hold the address information -- the information is also added to the message header. A client that uses a JAXM provider that supports the `EbXMLMessageImpl` profile implementation can specify the `setSender` and `setReceiver` methods as follows:

```
ebxmlMsg.setSender(new Party(
    "http://www.employeeportal.com/bookordering"));
ebxmlMsg.setReceiver(new Party("http://www.funbooks.com"));
```

Note that because the endpoint address is provided in the header, it does not need to be specified when the message is sent (see [Send the Message](#)).

The client then gets the `SOAPPart` object and the objects within, and adds its content:

```
SOAPPart sp = ebxmlMsg.getSOAPPart();
SOAPEnvelope envelope = sp.getEnvelope();
SOAPHeader hdr = envelope.getHeader();
SOAPBody bdy = envelope.getBody();
SOAPBodyElement sbe = bdy.addBodyElement
    (envelope.createName("SendBookDetails", "bp",
        "http://bookprovider.com"));
sbe.addTextNode("BuyISBN"+"**"+ isbn);
```

Add Attachments (If Any)

Attachments are added in the same way as for a standalone client.

Send the Message

A client that uses a JAXM provider sends a message using the `send` method of the `ProviderConnection` object. Unlike the `call` method that is used to send messages from a standalone client, the `send` method is specified with only one argument: the `SOAPMessage` object. The `call` method requires a second argument that specifies the endpoint

address of the recipient. However, for a client that uses a JAXM provider, the endpoint address is in the header of the `SOAPMessage` object, so the address does not need to be specified in the send request:

```
pc.send(ebxmlMsg)
```

Process the Message

A client that uses a JAXM provider can exchange messages synchronously or asynchronously. If the client sends the message synchronously, the receiving service processes it in the same way as though the client was standalone. In particular, the service must implement the `ReqRespListener` interface, and define within the `onMessage` method actions that get the `SOAPPart` object and the envelope, header, body, and child elements within the object. The service can then process the content and build a reply.

If the client uses a JAXM provider to exchange messages asynchronously, the service needs to implement the `OnewayListener` interface in the `javax.xml.messaging` package. For example, suppose that the recipient is a service that is implemented as a servlet. The following statement declares that the servlet implements the `OnewayListener` interface:

```
import javax.xml.messaging.*;
```

```
public class PurchaseConfirmer extends JAXMServlet implements OnewayListener
```

Notice the `JAXMServlet` specification. This is a superclass for servlets that receive JAXM messages.

The `OnewayListener` interface declares one method, `onMessage`, that is used to describe how to process an asynchronous request. The method takes a `SOAPMessage` object as a parameter. When the method is called, it passes the `SOAPMessage` object to the `OnewayListener` implementation. Unlike the case for a synchronous message request, the recipient of an asynchronous request does not have to send an immediate response. The time interval between the request and the response can, in fact, be very long. The implementation of the `onMessage` method describes how the recipient processes the asynchronous request. Here, for example, the implementation of the `onMessage` method displays a number of messages. It also uses various methods in the basic ebXML profile implementation, `EbXMLMessageImpl`, to process the `SOAPMessage` object:

```
public void onMessage(SOAPMessage message) {
    System.out.println(
        "Asynchronous On message call in receiving servlet");
```

```
    System.out.println("Here's the message: ");
    message.saveChanges();
    message.writeTo(System.out);
```

```
    System.out.println(
        "Sending an asynchronous message to origin ");
```

```
    EbXMLMessageImpl ebxmlMsg = (EbXMLMessageImpl)message;
    Party from = ebxmlMsg.getSender();
    Party to = ebxmlMsg.getReceiver();
```

```
    ebxmlMsg.setReceiver(from);
    ebxmlMsg.setSender(to);
    ebxmlMsg.saveChanges();
```

Notice the `saveChanges` method. This method is used to save changes made to the `SOAPMessage` object.

After the service processes the message, it sends the reply using the `send` method of the `ProviderConnection` object:

```
pc.send(ebxmlMsg);
ebxmlMsg.writeTo(System.out);
System.out.println(
    "***Sent message from ReceivingServlet");
```

Process the Reply

The way that a client that uses a JAXM provider processes a reply from the service is the same as the way that the service processes a message from the client. If the client uses a JAXM provider to exchange messages synchronously, the client must implement the `ReqRespListener` interface, and define within the `onMessage` method how to handle the reply.

If the client uses a JAXM provider to exchange messages asynchronously, the client needs to implement the `OnewayListener` interface, and define within the `onMessage` method how to handle the reply. For example, suppose a client is implemented as a servlet. The following statement declares that the servlet implements the `OnewayListener` interface:

```
public class BookPurchaser extends
    JAXMServlet implements OnewayListener {
```

Here is the definition of the `onMessage` method in the client. The definition describes the actions taken by the client to process a reply from the service. In this example, the client displays changes made by the service to the message initially sent by the client. The client uses the `saveChanges` method in the basic ebXML profile implementation to get the changes.

```
public void onMessage(SOAPMessage message) {
    System.out.println(
        "Sender received asynchronous reply from receiver");
    try {
        System.out.println("Here's the message: ");
        message.saveChanges();
        message.writeTo(System.out);
    } catch(Exception e) {
        e.printStackTrace();
    }
}
```

A JAXM Example

This section presents an example of JAXM in use. The example is based on a sample application that uses JAXM (as well as JAX-RPC) to access Web services. For instructions on building and running the sample application, see [Build and Run the Sample Application](#).

[Part 1 of this series](#) presented an example that illustrated how a fictitious company named BooksToGo uses JAXR to register a Web service, and how another fictitious company, BoomingBusiness.com, uses JAXR to discover the service.

Recall that BoomingBusiness.com provides a Web site to its employees. The Web site is a portal to a variety of employee services. One of the services that BoomingBusiness.com plans to make available through its Web site is an online service for ordering books. BooksToGo is a provider of that service. BooksToGo has recently changed its name to FunBooks.

BoomingBusiness.com's IT staff decides to take a messaging approach for the book order service. When an employee requests the online book ordering service from the employee portal, an underlying program will dynamically search a business registry for online book service providers. JAXM will then be used to exchange messages between employees submitting book order requests, and the registered book service providers.

In addition to FunBooks, Aerns and Cable is a company that provides online book services. Both FunBooks and Aerns and Cable want to make their services available to clients. Both companies choose to make their online book services available through JAXM. Let's examine what FunBooks and Aerns and Cable do to make their online book services available to clients through JAXM. Then let's examine what BoomingBusiness.com does to access the online book services through JAXM.

FunBooks and Aerns and Cable Register Their Online Book Services

FunBooks and Aerns and Cable use the JAXR API in the Java WSDP package to register their online book services. They choose to register the services in a UDDI registry. (In the sample application, the UDDI registry used is the Java WSDP Registry Server v1.0_01 provided in the Java WSDP v1.0.) For a description of the steps involved in registering a Web service, see the [JAXR example](#) in Part 1 of this series.

Each provider publishes information about its online book service. For example, Aerns and Cable publishes the following information:

Business Name	Aerns and Cable
Contact Information	Primary Contact: Bhakti Mehta Phone number: (877)1111111 Email Address: bhakti.mehta@ae.com
Classification Scheme (classification, code)	NAICS (Book Stores, 451211)
Service	Online Book Ordering
Service Binding	Description: JAXM-FCS (SOAP/HTTP) based binding Access Point: http://localhost:8080/ancbooks/bookordering

Notice the classification code, 451211. This is the [North American Industry Classification System \(NAICS\) code](#) for book stores. To find the online book service providers, BoomingBusiness.com's application program will query a registry for entries that have the book stores code in their classification.

Notice too the binding information: JAXM-FCS (SOAP/HTTP) based binding. This identifies the service as accessible through JAXM. In looking for online book service providers, BoomingBusiness.com's application program will look for entries that meet this criteria. The access point identifies the URL for the service endpoint. JAXM clients will specify this URL to contact the service. In this example, the URL is a localhost URL, so that the service endpoint is actually on

your machine. In a real Web service deployment, the service endpoint would probably be on a different machine.

The sample application includes a class that registers the online book order services. To see the source code for the class, look [here](#).

FunBooks and Aerns and Cable Deploy Their Online Book Services

Both online book service providers want to deploy their services in a J2EE-compliant container. (In the sample application, the container is Tomcat.) They also decide to implement the service endpoints for their services as servlets. To deploy Web server-based components such as servlets in a J2EE-compliant container, each service provider needs to create a WAR file. The service providers create their respective WAR files using J2EE tools.

To illustrate what the services look like, [here](#) is the source code for a servlet in FunBook's online book service that exchanges messages synchronously with a JAXM client. For example, when a BoomingBusiness.com employee initiates a book search by submitting search criteria in the online book order form, the "[book client](#)" in BoomingBusiness.com's employee portal sends a SOAP message synchronously to various service endpoints. When the endpoint for FunBook's online book service receives the message, this servlet processes it. Notice how the servlet:

- Gets the request
- Extracts the query criteria and query value from the request
- Checks the inventory for the books matching the criteria
- Sends the response as a SOAP attachment
- Sends any promotion information as another attachment

These actions are further described in the [Process the Message](#) section under [Exchanging Messages Without a JAXM Provider](#).

[Here](#) is the source code for a servlet in FunBook's online book service that exchanges messages asynchronously with a JAXM client. For example, when a BoomingBusiness.com employee selects a book from the results page, the [book purchaser](#) in BoomingBusiness.com's employee portal sends a SOAP message asynchronously to a JAXM provider, which then routes the message to the target endpoint. When the endpoint for FunBook's online book service receives the message, this servlet processes it and sends a confirmation message back to the sender. Notice how the servlet:

- Implements the `OnewayListener` interface
- Defines the `onMessage` method
- Sends the asynchronous reply

These actions are further described in the [Process the Message](#) section under [Exchanging Messages With a JAXM Provider](#).

BoomingBusiness.com Creates the Client

BoomingBusiness.com envisions the following flow of events for its online book ordering service. After an employee clicks a "Online Book Ordering Service" link in the employee portal:

Click to enlarge each image

1. The employee portal displays an online book ordering form.
2. The employee specifies search criteria such as the name of a book or an author.

Online Book Ordering Form

Select Criteria and enter information:

Search Criteria Search Value

ISBN

Name of book

Author of book

Online Book Ordering Form

Select Criteria and enter information:

Search Criteria Search Value

ISBN

Name of book

Author of book

3. Providers that offer books meeting the search criteria respond with information about those books.

4. The employee selects one of the books.

Results of your query

Book Supplier	ISBN	Name	Price	Description	Author	Price
• Amazon.com	123456	Web Services today and beyond	49	This book deals with the different web services technologies and how they interact with each other.	John Doe	View details
• Packtbooks	123456	Web Services today and beyond	49	This book deals with the different web services technologies and how they interact with each other.	John Doe	View details

Results of your query

Book Supplier	ISBN	Name	Price	Description	Author	Price
# Amazon.com	123456	Web Services today and beyond	49	This book deals with the different web services technologies and how they interact with each other.	John Doe	View details
• Packtbooks	123456	Web Services today and beyond	49	This book deals with the different web services technologies and how they interact with each other.	John Doe	View details

The provider of the selected book then sends a note to confirm the transaction.

Here's what BoomingBusiness.com's IT staff plans to implement in support of this interaction:

- A link for "Online Book Ordering Service" in the home page for their employee portal.
- A ["book query" JSP](#) that displays the online book ordering form.
- A ["process book query" JSP](#) that processes the search.
- A ["process result" JSP](#) that displays the search results.
- A ["book info handler" JavaBean](#) that validates search criteria entered in the online book order form, and initiates the search.
- A ["book lister" class](#) that searches for books that meet the submitted search criteria.
- A ["client query" class](#) that finds registered online book providers.
- A ["book client" class](#) that send a message synchronously to registered online book providers that expose a JAXM binding.
- A ["book purchaser" servlet](#) that sends a message asynchronously to the provider of a selected book.

Let's look at some of these components in more detail.

Book Query JSP: [Here](#) is the source code for the book query JSP. The JSP displays the online book ordering form. After the employee enters search criteria in the form, the subsequent action is to invoke the [process book query JSP](#):

```
<FORM method=post action=processBookQuery.jsp>
...
<font size=+2>Select Criteria and enter information: </font>
...
<TR valign=center><TH align=left>Search Criteria</TH>
...
<TH align=left>Search Value</TH> </TR>
```

```
<tr valign=center>
<td align=left> ISBN ...
</td>
...
```

Process Book Query JSP: [Here](#) is the source code for the process book query JSP. Notice the use of the [book info handler JavaBean](#). The JSP invokes the `validate` method in the book info handler JavaBean to validate that a search criteria was selected in the online book order form. The JSP then invokes the `executeTest` method in the book info handler JavaBean to identify which search criteria was selected, and to initiate the search.

```
<jsp:useBean id="bookInfoHandler"
  class="com.sun.eportal.bookordering.BookQueryBean"
  scope="request" />
...
<%
  if(bookInfoHandler.validate()) {
    ...
    bookInfoHandler.executeTest();
```

The process book query JSP then invokes the [process result JSP](#) to display the search results:

```
<jsp:forward page="result.jsp?searchCriteria='
  %= searchCriteria %>'&searchValue='
  %= searchValue %>'" />
```

Process Result JSP: [Here](#) is the source code for the process result JSP. The JSP displays the results page. After the employee selects a book in the results page, the subsequent action is to invoke the [book purchaser servlet](#):

```
...
<form method=get action=bookpurchaser>
...
<h1 align=center>...Results of your query ...
<jsp:useBean id="bookInfoHandler"
  class="com.sun.eportal.bookordering.BookQueryBean"
  scope="request" />
...
<table border=0>
...
...
  <TH ...>
...
    <P> Book Supplier</P>
  </TH>
  <TH ...>
    <P>ISBN</P>
  </TH>
...

```

Notice that the JSP page gets the data returned in the search by the [book info handler](#) JavaBean:

```
<%
```



```

bookVector = new Vector();
try {
String cScheme="ntis-gov:naics";
String keyName="Book Stores";
String keyValue="451211";
String serviceName="Online Book Ordering";

...

ClientQuery clientQuery = new ClientQuery( );
Vector serviceProviderInfoVector = clientQuery.query(
    cScheme, keyName, keyValue);

...

```

After the the query returns information from the registry, the book lister examines the binding information exposed by the candidate providers. If the binding is JAXM, the book lister adds the provider's service endpoint to a vector:

```

if ( spi.getCommunicationType().equals("JAXM") ) {
    jaxmEndpointVector.addElement( spi.getEndpoint() );
}

```

The book lister then uses the `constructMessage` method in the [book client class](#) to construct a SOAP message for the online book request, and the `getAvailableBooks` method to send the SOAP message synchronously to each endpoint:

```

BookClient bookClient = new BookClient();
bookClient.constructMessage ( queryType, queryValue );

...

bookVector = bookClient.getAllAvailableBooks (
    jaxmEndpointVector );

```

Client Query Class: [Here](#) is the source code for the client query class. The class uses JAXR to search a Web services registry. Notice how the query method in the class searches the registry for organizations that meet the supplied classification scheme, and retrieves information about service providers that meet the classification criteria:

```

ClassificationScheme classificationScheme =
    queryManager.findClassificationSchemeByName(
        findQualifiers, cName );
Classification classification =
    lifecycleManager.createClassification(
        classificationScheme, keyName, keyValue );

...

BulkResponse response =
    queryManager.findOrganizations(findQualifiers,
        null, classifications, null, null, null);
Collection orgs = response.getCollection();

```

For more information about using JAXR to search a Web services registry, see [Part 1 of the series](#).

Book Client Class: [Here](#) is the source code for the book client class. The class uses JAXM to send a SOAP message synchronously to a URI endpoint. The [book lister class](#) uses the `constructMessage` method in the book client class to construct a SOAP message for the online book request. Here is the source code for the `constructMessage` method:

```
public void constructMessage(String queryType, String queryValue){
    try {
        MessageFactory mf = MessageFactory.newInstance();
        SOAPMessage msg = mf.createMessage();
        SOAPPart sp = msg.getSOAPPart();
        SOAPEnvelope envelope = sp.getEnvelope();

        SOAPHeader hdr = envelope.getHeader();
        SOAPBody bdy = envelope.getBody();

        SOAPBodyElement sbe = bdy.addBodyElement
            (envelope.createName("GetBookDetails", "bp",
                "http://www.bookprovider.com"));

        sbe.addChildElement(envelope.createName("searchCriteria", "bp",
            "http://www.bookprovider.com")).addTextNode(queryType);

        sbe.addChildElement(envelope.createName("searchValue", "bp",
            "http://www.bookprovider.com")).addTextNode(queryValue);
        message=msg;
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Notice how the `constructMessage` method uses the `MessageFactory` class to create a `SOAPMessage` object for the message, and how it adds content (in this case, the search criteria and search value) to the body of the message. These actions are further explained in [Create a Message](#) and [Add Content to the Message](#).

After constructing the SOAP message, the [book lister class](#) calls the `getAvailableBooks` method in the book client class to send the SOAP message synchronously. The method is called for each endpoint associated with a registered online book provider (and that exposes a JAXM binding). Each call to the method for a specific endpoint sends the SOAP message synchronously to that endpoint. Because the exchange is synchronous, the `getAvailableBooks` method waits after sending the message. Here is part of the source code for the `getAvailableBooks` method:

```
public Vector getAvailableBooks (String endPoint){
    try {
        Vector retVector = new Vector();
        URLEndpoint urlEndpoint =
            new URLEndpoint(endPoint);
        ...

        SOAPConnection conn;
        SOAPConnectionFactory scf =
            SOAPConnectionFactory.newInstance();
```

```

conn = scf.createConnection();

SOAPMessage reply =
    conn.call(message, urlEndpoint);

```

Notice how the `getAvailableBooks` method gets a connection by creating a `SOAPConnection` object. It then uses the `call` method of the connection to send the message. The two parameters in the call are objects that represent the message and the target endpoint. These actions are further explained in [Get a Connection](#) and [Send the Message \(and Wait\)](#).

Each service at a target endpoint processes the SOAP message it receives and returns a reply. For example, after receiving the message, the FunBooks Online Book Service processes the message and returns a reply. [Here](#) is the source code for FunBook's online book service.

Because it sent a message synchronously, the `getAvailableBooks` method is blocked until it receives a reply. After the method receives a reply it continues processing. Specifically, it gets the pertinent elements of the reply message, that is, the envelope, body, and child elements. It then uses an `Iterator` object to access each child element. This is further explained in [Process the Reply](#).

```

...
SOAPEnvelope envelope =
    reply.getSOAPPart().getEnvelope();
SOAPHeader header = envelope.getHeader();

orgName = extract( envelope, header, "organization");

...
int attachmentCount = reply.countAttachments();
...

Iterator attachmentIterator = reply.getAttachments();
while ( attachmentIterator.hasNext() ) {
    ...
    ap = (AttachmentPart)attachmentIterator.next();
    System.out.println(
        "ContentType is" + ap.getContentType());
    ...
}

```

The book client expects the reply to contain two attachments: one, in content type `text/xml`, that contains book information, and the other, in content type `text/html`, that contains promotional information. It saves the promotional information attachment in a file, and transforms the book information to XML. It does the transform using the Java API for XML Parsing (JAXP) packages, `javax.xml.transform` and `javax.xml.transform.stream`. JAXP v1.2_01 is part of Java WSDP 1.0_01.

```

if ( ap.getContentType().equals("text/html") ) {
    String is = (String)ap.getContent();
    FileOutputStream fos = new FileOutputStream (
        "../webapps/ROOT/" +getFileName(endPoint)+
        ".html");

    fos.write(is.getBytes());
    System.out.println("Attachment is saved "+
        getFileName(endPoint)+".html");
}

```

```

        fos.flush();
        fos.close();
    } else {

        javax.xml.transform.stream.StreamSource
            bookSource = (StreamSource)ap.getContent ();
        ...
        TransformerFactory tfactory = TransformerFactory
            .newInstance();
        Transformer transformer =
            tfactory.newTransformer();
        transformer.setOutputProperty(OutputKeys.INDENT,
            "yes");
        transformer.setOutputProperty(OutputKeys.METHOD,
            "xml");
        ...

        file = getFileName(endPoint);
        String fileName = "../webapps/ROOT/"+file+".xml";
        FileOutputStream fos = new FileOutputStream
            (fileName) ;
        transformer.transform(bookSource,
            new StreamResult(fos));
        fos.close();
        ...
    }
}

```

Book Purchaser Servlet: [Here](#) is the source code for the book purchaser servlet. The servlet is invoked when a user selects one of the books in the results page. The servlet uses JAXM to send a SOAP message asynchronously to a JAXM provider, which then routes the message to the target endpoint. Notice how the servlet:

- Gets a connection with a JAXM provider by creating a `ProviderConnection` object:

```

pcf = ProviderConnectionFactory.newInstance();
pc = pcf.createConnection();

```

This is further described in [Get a Connection With a JAXM Provider](#).

- Identifies a profile for the exchange:

```

ProviderMetaData metaData = pc.getMetaData()
String[] supportedProfiles =
    metaData.getSupportedProfiles();
String profile = null;

for(int i=0; i < supportedProfiles.length; i++) {
    if(supportedProfiles[i].equals("ebxml") {
        profile = supportedProfiles[i];
    }
}

```

This is further described in [Create a Message](#).

- Creates a message from a `MessageFactory` instance that is initialized with the profile -- the `MessageFactory` instance uses the basic ebXML profile implementation provided in the Java WSDP:

```

mf = pc.createMessageFactory(profile);

```

```

...
    EbXMLMessageImpl ebxmlMsg =
        (EbXMLMessageImpl)mf.createMessage();

```

This is further described in [Create a Message](#).

- Adds content to the message. In this case the content is the ISBN number of the selected book. Notice the use of the ebXML profile implementation method `setSender` and `setReceiver` to identify the location of the client, and to identify the endpoint address of the recipient, respectively:

```

ebxmlMsg.setSender(new Party(from));
ebxmlMsg.setReceiver(new Party(endPoint));

SOAPPart sp = ebxmlMsg.getSOAPPart();
SOAPEnvelope envelope = sp.getEnvelope();

SOAPHeader hdr = envelope.getHeader();
SOAPBody bdy = envelope.getBody();

SOAPBodyElement sbe = bdy.addBodyElement
    (envelope.createName("SendBookDetails", "bp",
        "http://bookprovider.com"));

sbe.addTextNode("BuyISBN"+"**"+ isbn);

```

This is further described in [Add Content to the Message](#).

- Sends the message using the `send` method of the `ProviderConnection` object:

```

pc.send(ebxmlMsg);

```

Java™ Web Services Developer Pack

Part 2: RPC Calls, Messaging, and the JAX-RPC and JAXM API

Build and Run the Sample Application

You can build and run the sample application that is highlighted in the [JAX-RPC example](#) and the JAXM example.

To build and run the application:

1. Download and install the [Java WebServices Developer Pack v1.0 01](#) if it isn't already installed.
2. Download the [employeeportal-fcs.zip file](#) into a directory. This file contains the files for the sample application.
3. Unzip the `employeeportal-fcs.zip` file. This will create an `employeeportal-fcs` directory.
4. Change the current directory (using the `cd` command) to the `employeeportal-fcs` directory.
5. Set the `JWSDP_HOME` environment variable to the directory where the Java Web Services Developer Pack is installed. The command you use to set the environment variable depends on your command line or shell. For example, if you installed the Java Web Services Developer Pack in directory

/home/user/jwsdp-1_0-fcs (UNIX) or /home/user/jwsdp-1_0-fcs (Windows), enter the command:

UNIX Korn shell (ksh):

```
export JWSDP_HOME=/home/user/jwsdp-1_0-fcs
```

UNIX C shell (csh):

```
setenv JWSDP_HOME /home/user/jwsdp-1_0-fcs
```

Windows:

```
set JWSDP_HOME=c:\jwsdp-1_0-fcs
```

6. Set the ANT_HOME environment variable to JWSDP_HOME.

UNIX Korn shell (ksh):

```
export ANT_HOME=$JWSDP_HOME
```

UNIX C shell (csh):

```
setenv ANT_HOME $JWSDP_HOME
```

Windows:

```
set ANT_HOME%=JWSDP_HOME%
```

7. Ensure that ANT_HOME/bin (UNIX) or ANT_HOME\bin (Windows) is in your path.

UNIX Korn shell (ksh):

```
export PATH=$ANT_HOME/bin:$PATH
```

UNIX C shell (csh):

```
setenv PATH $ANT_HOME/bin:$PATH
```

Windows:

```
set PATH=%ANT_HOME%\bin;%PATH%
```

8. Enter the command:

```
ant clean
```

9. Enter the command:

```
ant build-all
```

10. Enter the command:

```
ant deploy
```

11. Change the directory (cd) to \$JWSDP_HOME, and enter the command:

UNIX

```
xindice-start.sh
```

Windows

```
xindice-start.bat
```

The command starts Java WSDP Registry Server v1.0_02. After you enter the xindice command, examine the xindice.log file in the \$JWSDP_HOME/logs directory. If the Registry Server is started properly, you should see the following lines in the file:

```
Database: 'db' initializing
Script: 'GET' added to script storage
Service: 'db' started
Service: 'HTTPServer' started @ http://:4080/
Service: 'APIService' started
Server Running
```

12. Open a new terminal window, and change the directory to \$JWSDP_HOME/bin (UNIX) or \$JWSDP_HOME/bin (Windows). Enter the command:

UNIX

```
./catalina.sh run
```

Windows

```
.\catalina.sh run
```

The command starts the Apache Tomcat 4.1.2 container that is packaged with the Java WSDP.

13. Open a browser
14. In the browser, enter the URL:

```
http://localhost:8080
```

The browser should then display the Java WSDP start page.

15. Go to the first terminal window. Change the current directory to the employeeportal-fcs directory, and enter the following command:

```
ant publish
```

This publishes all the service provider information into the Java WSDP Registry Server. (You only need to perform this step once.)

16. In the browser, enter the URL:

```
http://localhost:8080/eponfcs/index.html
```

The browser should then display the Employee Portal Index Page.

17. Click on either the Retirement Service link or Online Book Ordering Service link on the Employee Portal Index Page.

If you click on the Retirement Service link, the browser should display a table of retirement funds.

- Select one or more funds, for example, Happy old days funds and Old orchards funds.
- Specify an investment percentage for each selected fund, for example, Happy old days funds: 30 and Old orchards funds: 70
- Enter a monthly investment in the Investing Money field, for example, \$100
- Click the Go button. The browser should display a page with two tables: one that displays the funds you selected, the other that displays quotes received from providers of the selected funds.
- Select one of the quotes, by selecting the radio button next to it.
- Click the Confirm button. The browser should then display a confirmation message.

If you click on the Online Book Ordering Service link, the browser should display a book ordering page.

- Enter one of the following search criteria:

Enter 1 in the ISBN field,
or enter Web Services in the Name of Book field,
or enter Bill Brown in the Author of Book field

The browser should then display a list of books.

- Select one book from the list.
- Click the Submit button.