CHAPTER 2

Standards and Technologies

This chapter describes current, universally accepted Web Service standards and the J2EE platform's support for these standards. The Web services computing paradigm enables applications and services running on different platforms to easily communicate and interoperate with each other. To be universally accepted, the paradigm must give service implementors flexibility in their implementation approach. Just as important, each such implementation must be assured that it can work with another implementation. Proven technologies facilitate Web service development and universally accepted standards enable interoperability.

2.1 Overview of Web Service Standards

Standards differ from technologies. Standards are a collection of specifications, rules, and guidelines formulated and accepted by the leading market participants. While these rules and guidelines prescribe a common way to achieve the standard's stated goal, they do not prescribe implementation details. Individual participants devise their own implementations of an accepted standard according to the standard's guidelines and rules. These various implementations of a standard by different vendors gives raise to a variety of technologies. However, despite the implementation detail differences, the technologies can work together if they have been developed according to the standard's specifications.

For Web services to be successful, the Web service standards must be widely accepted. To enable such wide acceptance, the standards used for Web services

and the technologies that implement those standards should meet the following criteria:

- A Web service should be able to service requests from any client regardless of the platform on which the client is implemented.
- A client should be able to find and use any Web service regardless of the service's implementation details or the platform on which it runs.

Standards establish a base of commonality and enable Web services to achieve wide acceptance and interoperability. Standards cover areas such as:

- **Common markup language for communication**—To begin with, service providers, who make services available, and service requestors, who use services, must be able to communicate with each other. Communication mandates the use of a common terminology, or language, through which providers and requestors talk to one another. A common markup language facilitates communication between providers and requestors, as each party is able to read and understand the exchanged information based on the embedded markup tags. Although providers and requestors can communicate using interpreters or translators, using interpreters or translators is impractical because such intermediary agents are inefficient and not cost effective.
- **Common message format for exchanging information**—Although establishing a common markup language is important, by itself it is not sufficient for two parties (specifically, the service providers and service requestors) to properly communicate. For effective communication, the parties must be able to exchange messages according to an agreed-upon format. By having such a format, parties who are unknown to each other can communicate effectively.
- **Common service specification formats**—In addition to common message formats and markup language, there must be a common format that all service providers can use to specify service details such as the service type, how to access the service, and so forth. A standard mechanism for specifying service details enables providers to specify their services so that any requestor can understand and use them.
- **Common means for service lookup**—In the same way that providers need a common way to specify service details, service requestors must have a common way to look up and obtain details of a service. This is accomplished by

having common, well-known locations where providers can register their service specifications and where requestors know to go to find services. By having these common, well-known locations and a standard way to access them, services can be universally accessed by all providers and requestors.

Although they do not exhaustively discuss these basic standards, the next sections provide enough information about the standards to enable further discussion about the J2EE technologies that implement them. For complete details, see the reference section at the end of this chapter. In addition to these basic standards, more complex Web services that implement enterprise-level processes need standards for security, transactions, process flow control, and so forth.

2.1.1 Extensible Markup Language

The eXtensible Markup Language (XML), a standard accepted throughout the industry, enables service providers and requestors to communicate with each other in a common language. XML is not dependent on a proprietary platform or technology, and messages in XML can be communicated over the Internet using standard Internet protocols such as HTTP. Because XML is a product of the W3C body, changes to it will be supported by all leading players. This ensures that as XML evolves, Web services can also evolve without backward compatibility concerns.

XML is a simple, flexible, text-based markup language. XML data is marked using tags enclosed in angled brackets. The tags contain the meaning of the data they mark. Such markup allows different systems to easily exchange data with each other. This differs from tag usage in HTML, which is oriented to displaying data. Unlike HTML, display is not inherent in XML.

A Document Type Definition (DTD) or XML Schema Definition (XSD) describes the structure of an XML document. It has information on the tags the corresponding XML document can have, the order of those tags, and so forth. An XML document can be validated against its DTD or its XSD. Validating an XML document ensures that the document follows the structure defined in its DTD or XSD and that it has no invalid XML tags. Thus, systems exchanging XML documents for some purpose can agree on a single DTD or XSD and validate all XML documents received for that purpose against the agreed-upon DTD/XSD before processing the document. Code Example 2.1 shows the code from an XML document representing an individual’s contact information.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<ContactInformation>
    <Name>John doe</Name>
    <Address>
        <Street>1234, Heavenly Drive</Street>
        <City>Hell</City>
        <State>Timboktu</State>
        <Country>NoWhere</Country>
    </Address>
    <HomePhone>123-456-7890</HomePhone>
    <EMail>johndoe@nowhere.com</EMail>
</ContactInformation>
```

Code Example 2.1 XML Document Example

Code Example 2.2 is the DTD for the XML document in Code Example 2.1.

```
<!ELEMENT ContactInformation (Name, Address, HomePhone, EMail)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Address (Street, City, State, Country)>
<!ELEMENT Street (#PCDATA)>
<!ELEMENT City (#PCDATA)>
<!ELEMENT State (#PCDATA)>
<!ELEMENT Country (#PCDATA)>
<!ELEMENT HomePhone (#PCDATA)>
<!ELEMENT EMail (#PCDATA)>
```

Code Example 2.2 Document Type Definition

Unfortunately, DTDs are an inflexible way to define XML document formats For example, DTDs provide no real facility to express data types or complex structural relationships. XML schema definitions standardize the format definitions of XML documents. Code Example 2.4 shows the XSD schema for the sample XML document in Code Example 2.3.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<ContactInformation
       xmlns="http://simple.example.com/CInfoXmlDoc"
```

```
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation=
            "http://simple.example.com/CInfoXmlDoc
            file:./CInfoXmlDoc.xsd">
    <Name>John doe</Name>
    <Address>
        <Street>1234, Heavenly Drive</Street>
        <City>Hell</City>
        <State>Timboktu</State>
        <Country>NoWhere</Country>
    </Address>
    <HomePhone>123-456-7890</HomePhone>
     <EMail>johndoe@nowhere.com</EMail>
</ContactInformation>
```

Code Example 2.3 XML Document

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://simple.example.com/CInfoXmlDoc"
    xmlns=" http://simple.example.com/CInfoXmlDoc"
    elementFormDefault="qualified">
    <xsd:element name="ContactInformation">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="Name" type="xsd:string" />
                <xsd:element name="Address">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="Street"
                               type="xsd:string" />
                        <xsd:element name="City"
                               type="xsd:string" />
                        <xsd:element name="State" type="xsd:string" />
                        <xsd:element name="Country"
                               type="xsd:string" />
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
```

```
            <xsd:element name="HomePhone" type="xsd:string" />
            <xsd:element name="EMail" type="xsd:string" />
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

Code Example 2.4 XSD Schema

When considering XML schemas, it is important to understand the concept of XML namespaces. To avoid the use of the same name with different meanings in different contexts, XML schemas may define a namespace. A namespace is a set of unique names that are defined for a particular context and which conform to rules specific for the namespace. Since a namespace is specific to a particular context, each namespace is unrelated to any other namespace. Thus, the same name can be used in different namespaces without causing a duplicate name conflict. XML documents, which conform to an XML schema and have multiple elements and attributes, often rely on namespaces to avoid a collision in tag or attribute names or to be able to use the same tag or attribute name in different contexts.

Technically speaking, an XML namespace defines a collection of names and is identified by a URI reference. (Notice in Code Example 2.4 the code `xmlns="http://simple.example.com/CInfoXmlDoc"`. Code such as this indicates that the XML schema defines a name space for the various tags and attributes in the document.) Names in the namespace can be used as element types or attributes in an XML document. The combination of URI and element type or attribute name comprise a unique universal name that avoids collisions. For example, in Code Example 2.4, there is a namespace that defines the ContactInformation document's element types, such as Name and Address. These element types are unique within the contact information context. If the document included another namespace context, such as BankInformation that defined its own Name and Address element types, these two namespaces would be separate and distinct. That is, a bank name and address would not conflict with a contact name and address.

Noted that XML namespaces have a standard internal structure, and are not merely a mathematical set, and thus differ from the more common namespace notion.

2.1.2 Simple Object Access Protocol

XML solves the need for a common language, and the Simple Object Access Protocol (SOAP) fills the need for a common messaging format. SOAP enables objects not known to one another to communicate; that is, to exchange messages. SOAP, a wire protocol similar to IIOP and JRMP, is a text-based protocol that uses an XML-based data encoding format and HTTP/SMTP to transport messages. SOAP is independent of both the programming language and the platform on which the application is running, and it does not require any specific technology at its endpoints, making it completely agnostic to vendors, platforms, and technologies. Its text format also makes SOAP a firewall-friendly protocol. Moreover, SOAP is backed by leading industrial players, and can be expected to have universal support.

To enable message exchanges, SOAP defines an envelope—a SOAP body, within which the message is included, and an optional SOAP-specific header. The whole envelope (body plus header) is one complete XML document. The header entries are for intermediate processors and they enable advanced features. The body, which contains the message contents, is consumed by the recipient. SOAP neither understands nor cares about the message contents; the only restriction is that the message be in XML format.

Code Example 2.5 shows a simple but complete example of a SOAP request for obtaining a stock quote.

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="SoapEnvelopeURI"
       SOAP-ENV:encodingStyle="SoapEncodingURI">
    <SOAP-ENV:Header>
    </SOAP-ENV:Header>
    <SOAP-ENV:Body>
        <m:GetLastTradePrice xmlns:m="ServiceURI">
            <tickerSymbol>SUNW</tickerSymbol>
        </m:GetLastTradePrice>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Code Example 2.5 Example SOAP Request

This example shows how a SOAP message is encoded using XML and illustrates some SOAP elements and attributes. All SOAP messages must have an `Envelope` element and must define two name spaces: one name space connotes the

SOAP envelope (`xmlns:SOAP-ENV`) and the other indicates the SOAP encoding (`SOAP-ENV:encodingStyle`). SOAP messages without proper name space specification are considered invalid messages. The `encodingStyle` attribute is important, as it is used to specify serialization rules for the SOAP message. Moreover, there can be no DTD referrals from within SOAP messages.

While optional, the `Header` element when used should be the first immediate child after the `Envelope`. The `Header` element provides a way to extend the SOAP message by specifying requirements such as authentication and transactions. Specifying requirements as part of the `Header` tells the message recipient how to handle the message.

There are many attributes that can be used with SOAP messages. Here are two examples that illustrate the use of attributes in the `Header` element. The `actor` attribute of the `Header` element enables a SOAP message to be passed through intermediate processes enroute to its ultimate destination. When the `actor` attribute is absent, the recipient is the final destination of the SOAP message. The `mustUnderstand` attribute makes processing of the `Header` element contents more flexible.

The `Body` element, which must be present in all SOAP messages, must follow immediately after the `Header` element, if it is present. Otherwise, the `Body` element must follow immediately after the start of the `Envelope` element. The `Body` contains the specification of the actual request (such as RPC calls). The `Fault` element in the SOAP `Body` enables error handling for message requests.

Note that additional features, such as SOAP with attachments and binding HTTP, although part of the SOAP standard, are not discussed in this chapter.

2.1.3 Universal Description, Discovery, and Integration Specification

The Universal Description, Discovery, and Integration (UDDI) specification defines a standard way for registering, deregistering, and looking up Web services. UDDI is a standards-based specification for Web service registration, description, and discovery. Similar to a telephone system's yellow pages, a UDDI registry's sole purpose is to enable providers to register their services and requestors to find services. Once a requestor finds a service, the registry has no more role to play between the requestor and the provider. Figure 2.1 shows how UDDI enables dynamic description, discovery, and integration of Web services. A Web service provider registers its services with the UDDI registry. A Web service requestor looks up required services in the UDDI registry and, when it finds a service, the requestor binds directly with the provider to use the service.

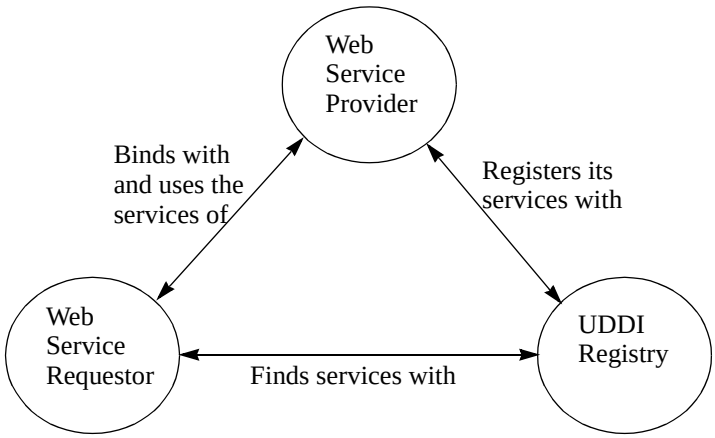

Figure 2.1 Role of a Registry in a Web Service

The UDDI specification defines an XML schema for SOAP messages and APIS for applications wanting to use the registry. A provider registering a Web service with UDDI must furnish business, service, binding, and technical information about the service. This information is stored in a common format that consists of three parts:

1. White pages—describe general business information such as name, description, phone numbers, and so forth
2. Yellow pages—describe the business in terms of standard taxonomies. This information should follow standard industrial categorizations so that services can be located by industry, category, or geographical location.
3. Green pages—list the service, binding, and service-specific technical information

The UDDI specification includes two categories of APIs for accessing UDDI services from applications:

1. Inquiry APIs—enable lookup and browsing of registry information
2. Publishers APIs—allow applications to register services with the registry

UDDI APIs behave in a synchronous manner. In addition, to ensure that a Web service provider or requestor can use the registry, UDDI uses SOAP as the base protocol. Note that UDDI is a specification for a registry, not a repository. As registries they function like a catalog, allowing requestors to find available services. They are not repositories because they do not contain the services themselves.

2.1.4 Web Services Definition Language

The Web Services Definition Language (WSDL) defines a standard way for specifying the details of a Web service. It is a general purpose XML schema that can be used to specify details of Web service interfaces, bindings, and other deployment details. By having such a standard way to specify details of a service, clients who have no prior knowledge of the service can still use that Web service.

WSDL specifies a grammar that describes Web services as a collection of communication end points, called ports. The data being exchanged are specified as part of messages. Every type of action allowed at an end point is considered an operation. Collections of operations possible on an end point are grouped together into port types. The messages, operations, and port types are all abstract definitions, which means the definitions do not carry deployment-specific details to enable their reuse.

The protocol and data format specifications for a particular port type are specified as a binding. A port is defined by associating a network address with a reusable binding, and a collection of ports define a service. In addition, WSDL specifies a common binding mechanism to bring together all protocol and data formats with an abstract message, operation, or endpoint.

Code Example 2.6 is a WSDL document for the `HelloWebService` Web service that merely says hello. The Web service, which uses SOAP as the communication protocol, expects to receive requests as strings and sends `String` type data as responses.

```
<?xml version="1.0" encoding="UTF-8"?>

<definitions name="HelloWebService"
             targetNamespace="urn:HelloWebService"
             xmlns:tns="urn:HelloWebService"
             xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

```
<types/>
<message name="HelloIntf_sayHello">
  <part name="String_1" type="xsd:string"/>
</message>
<message name="HelloIntf_sayHelloResponse">
  <part name="result" type="xsd:string"/>
</message>
<portType name="HelloIntf">
  <operation name="sayHello" parameterOrder="String_1">
    <input message="tns:HelloIntf_sayHello"/>
    <output message="tns:HelloIntf_sayHelloResponse"/>
  </operation>
</portType>
<binding name="HelloIntfBinding" type="tns:HelloIntf">
  <operation name="sayHello">
    <input>
      <soap:body
         encodingStyle=
                "http://schemas.xmlsoap.org/soap/encoding/"
         use="encoded" namespace="urn:HelloWebService"/>
    </input>
    <output>
      <soap:body
         encodingStyle=
             "http://schemas.xmlsoap.org/soap/encoding/"
         use="encoded" namespace="urn:HelloWebService"/>
    </output>
    <soap:operation soapAction=""/>
  </operation>
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
                          style="rpc"/>
</binding>
<service name="HelloWebService">
   <port name="HelloIntfPort"
     binding="tns:HelloIntfBinding">
     <soap:address
         location=
         ”http://example.COM:8000/webservice/HelloService"/>
```

```
        </port>
      </service>
    </definitions>
```

Code Example 2.6 WSDL Document for `HelloWebService` Web Service

A complete WSDL document consists of a set of definitions starting with a root `definitions` element followed by six individual element definitions—`type`, `message`, `port type`, `binding`, `port`, and `service`—that describe the services.

- The `type` element defines the data types contained in messages exchanged as part of the service. Data types can be simple, complex, derived, or array types. Types, either schema definitions or references, that are referred to in a WSDL document's message element are defined in the WSDL document's type element.
- The `message` element defines the messages that the Web service exchanges. A WSDL document has a `message` element for each message that is exchanged, and the `message` element contains the data types associated with the message.
- The `port type` element specifies, in an abstract manner, operations and messages that are part of the Web service. A WSDL document has one or more `port type` definitions for each Web service it defines.
- The `binding` element binds the abstract port type, and its messages and operations, to a formal protocol and to message formats.
- The `service` and `port` elements together define the name of the Web service and, by providing a single address for binding, assign an individual endpoint for the service. A port can have only one address. The `service` element groups related ports together and, through its `name` attribute, provides a logical name for the service.

This description is for a simple WSDL document. Each element definition has various attributes and WSDL has additional features, such as fault handling. WSDL also specifies how to bind directly with HTTP, MIME, and so forth, but these are beyond the scope of the current discussion. For more details, see the WSDL specification available at `www.w3c.org`.

2.1.5 Emerging Standards

So far we have examined existing standards, which meet the needs of simple Web services. Organizations that cross various industries have been formed to create and promote cross-platform standards. The Web Services Interoperability Organization (WS-I) is one such group. WS-I has published a WS-I Basic Profile that defines a set of cross-platform standards, such as those just examined, to promote and ensure interoperability. But other standards are required to address issues for Web services that handle complex business processes. These issues include strict security requirements, business processes interacting with other business processes and having long-lived transactions or transactions that span multiple business processes, or business processes nested within other processes. These business processes must also execute properly even when run on different platforms. Various standards bodies and organizations such as WS-I are currently working on these standards. Since these standards are still being defined and it is not yet clear which standards will be accepted as universal, we do not go into the details of emerging standards.

Now that we have examined the Web service standards, let's go on to see how J2EE supports these accepted standards.

2.2 J2EE 1.4: The Integrated Platform for Web Services

The J2EE 1.4 platform, with its main focus on Web services, integrates the existing Java-XML technologies into a consolidated platform in a standard way, thereby allowing applications to be exposed as Web services through a SOAP/HTTP interface. The J2EE 1.4 platform, using an accepted standard, integrates Java-XML technologies that enable development of Web services to the platform's existing technologies. The next sections briefly describe the Web service-specific additions made in the J2EE 1.4 platform. (Chapter 1 includes an overview of the J2EE 1.4 platform. See the J2EE 1.4 specification listed in "References" on page 59 for complete information on the platform.)

2.2.1 Java™ APIs for XML Processing

Java™ APIs for XML Processing (JAXP) is a vendor-neutral set of lightweight APIs for parsing or processing XML documents. Because XML is the common language enabling Web services, an XML parser is a necessity to understand the messages—the XML documents—exchanged among Web services. Figure 2.2 depicts how the JAXP API abstracts the parser implementations from the user application.

Keep in mind that the JAXP API is not new to the J2EE 1.4 platform. It has been part of the earlier versions of both the J2EE and Java™ 2 Standard Edition (J2SE™) platforms. In the J2EE 1.4 platform implementation, JAXP has added support for XML schemas.

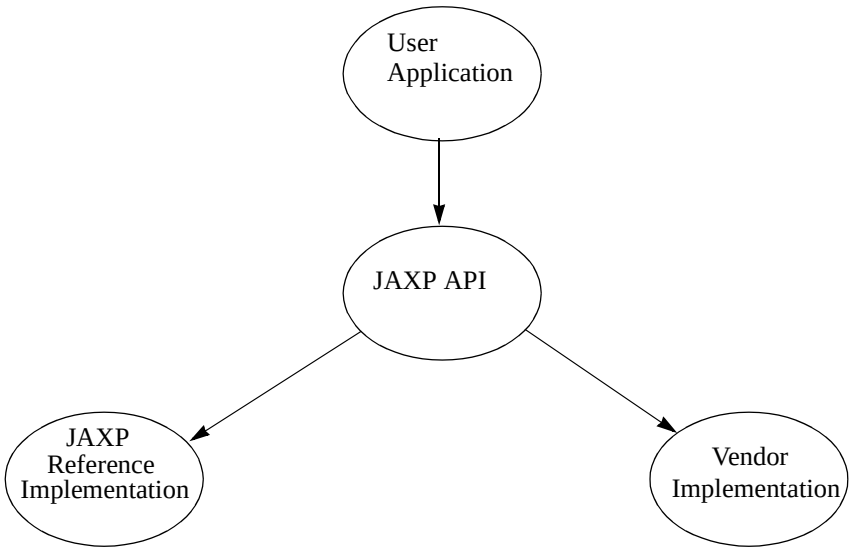

Figure 2.2 Using JAXP to Abstract Parser Implementations from User Application

Although it has its own reference implementation, JAXP allows JAXP specification-conforming parsers from other vendors to be plugged in. JAXP falls back to parsing an XML document using its own implementation if no other implementation is plugged in. JAXP processes XML documents using the SAX or DOM models, and it permits turning off XSLT engines during the document processing.

The main JAXP APIs are available through the `java.xml.parsers` package, which provides two vendor-agnostic factory interfaces—one interface for SAX processing and another for DOM processing. These factory interfaces allow the use of other JAXP implementations.

Figure 2.3 shows how the SAX and DOM parsers function.

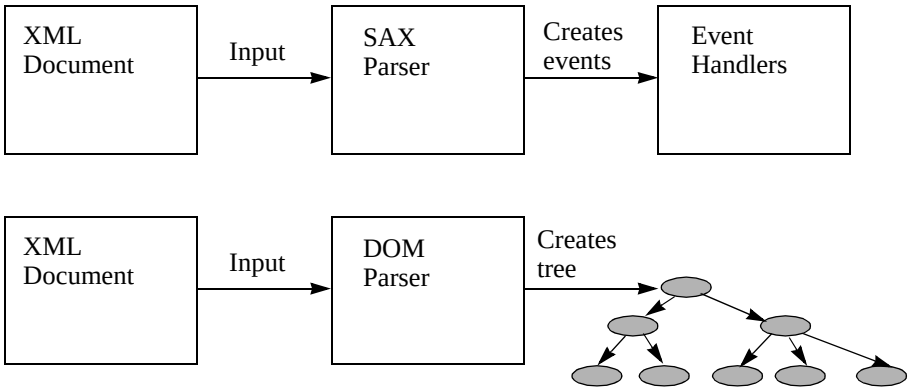

Figure 2.3 SAX- and DOM-Based XML Parser APIs

SAX processes documents serially, converting the elements of an XML document into a series of events. Each particular element generates one event, with unique events representing various parts of the document. User-supplied event handlers handle the events and take appropriate actions. SAX processing is fast because of its serial access and small memory storage requirements. Code Example 2.7 shows how to use the JAXP APIs and SAX to process an XML document. In this example, note that the `SAXEventHandler` class, which is used in the `setContentHandler` method, defines the handlers for the events generated by the SAX parser.

```
public class AnAppThatUsesSAXForXMLProcessing
       extends DefaultHandler {

   public void SomeMethodWhichReadsXMLDocument() {

       // Get a SAX PArser Factory and set validation to true
       SAXParserFactory spf = SAXParserFactory.newInstance();
       spf.setValidating(true);

       // Create a JAXP SAXParser
       SAXParser saxParser = spf.newSAXParser();

       // Get the encapsulated SAX XMLReader
```

```
            xmlReader = saxParser.getXMLReader();

            // Set the ContentHandler of the XMLReader
            xmlReader.setContentHandler(new SAXEventHandler());

            // Tell the XMLReader to parse the XML document
            xmlReader.parse(XMLDocumentName);
        }
    }
```

Code Example 2.7 Using SAX to Process an XML Document

DOM processing creates a tree from the elements in the XML document. Although this requires more memory (to store the tree), this feature allows random access to document content and enables splitting of documents into fragments, which makes it easier to code DOM processing. DOM facilitates creations, changes or additions to incoming XML documents. Code Example 2.8 shows how to use the JAXP APIs and DOM to process an XML document.

```
public class AnAppThatUsesDOMForXMLProcessing {

    public void SomeMethodWhichReadsXMLDocument() {

        // Step 1: create a DocumentBuilderFactory
        DocumentBuilderFactory dbf =
                    DocumentBuilderFactory.newInstance();

        // Step 2: create a DocumentBuilder that satisfies
        // the constraints specified by the DocumentBuilderFactory
        db = dbf.newDocumentBuilder();

        // Step 3: parse the input file
        doc = db.parse(XMLDocumentFile);

        // Parse the tree created - node by node

        for (Node child = doc.getFirstChild(); child != null;
               child = child.getNextSibling()) {
```

```
            // process the node
        }
    }
}
```

Code Example 2.8 Using DOM to Process an XML Document

2.2.2 Java™ API for XML-Based RPC

Java™ API for XML-based RPC (JAX-RPC) supports XML-based RPC for Java and J2EE platforms. It enables a traditional client-server remote procedure call (RPC) mechanism using an XML-based protocol. JAX-RPC enables Java technology developers to develop SOAP-based interoperable and portable Web services. Developers use the JAX-RPC programming model to develop SOAP-based Web service endpoints, along with their corresponding WSDL descriptions, and clients. A JAX-RPC-based Web service implementation can interact with clients that are not based on Java. Similarly, a JAX-RPC-based client can interact with a non-Java-based Web service implementation.

For typical Web service scenarios, using JAX-RPC reduces complexity for developers by:

- Standardizing the creation of SOAP requests and responses
- Standardizing marshalling and unmarshalling of parameters and other runtime and deployment-specific details
- Removing these SOAP creation and marshalling/unmarshalling tasks from a developer's responsibilities by providing these functions in a library
- Providing standardized support for different mapping scenarios, including XML to Java, Java to XML, WSDL to Java, and Java to WSDL mappings

JAX-RPC also defines standard mappings between WSDL/XML and Java, which enables it to support a rich type set. However, developers may use types that do not have standard type mappings. JAX-RPC defines a set of APIs for an extensible type mapping framework that developers can use for types with no standard type mappings. With these APIs, it is possible to develop and implement pluggable serializers and de-serializers for an extensible mapping. Figure 2.4 shows the "big picture" architecture of the JAX-RPC implementation.

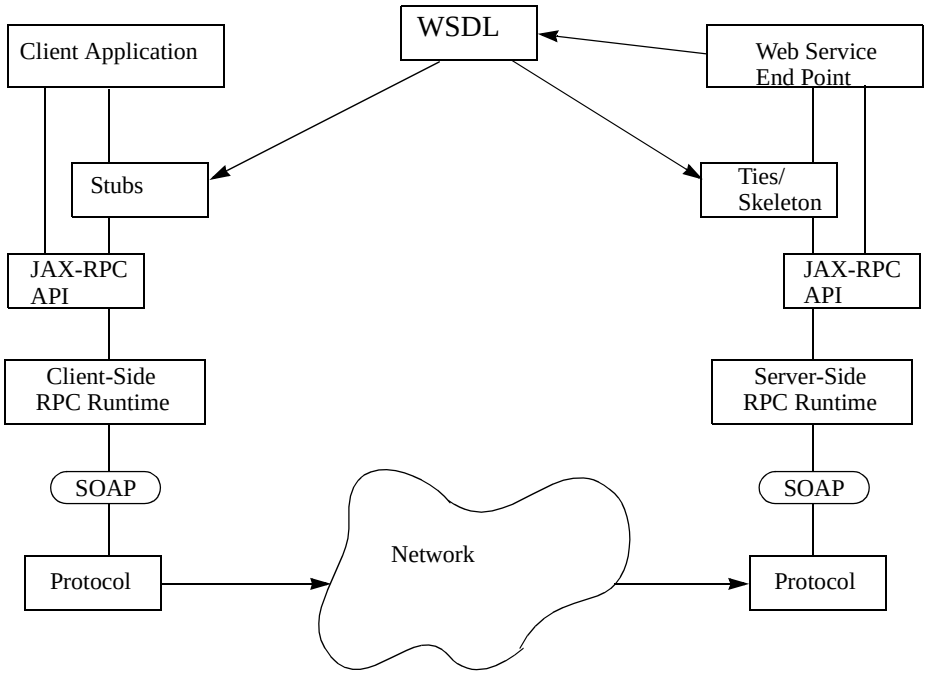

Figure 2.4 JAX-RPC Architecture

A client application can make a request to a Web service using in one of three ways:

1. Invoking methods on generated stub classes—Based on the contents of a WSDL description of a service, tools can be used to generate stub classes. These generated stubs are configured with all necessary information about the Web service and its endpoint. The client application uses the methods on the stubs to invoke remote methods available in the Web service endpoint.
2. Using a dynamic proxy—A dynamic proxy class supports a Web service endpoint. When this approach is used, there is no need to create endpoint-specific stubs for the client.
3. Using a dynamic invocation interface (DII)—In this approach, operations on target service endpoints are accessed dynamically based on an in-memory model of the WSDL description of the service.

No matter which approach is used, the client application's request passes through the client-side JAX-RPC runtime. The runtime maps the request's Java types to XML and forms a corresponding SOAP message for the request. It then sends the SOAP message across the network to the server.

On the server side, the JAX-RPC runtime receives the SOAP message carrying the request. The server-side runtime applies the XML to Java mappings, then maps the request to the corresponding Java method call, along with its parameters.

Note that a client of a JAX-RPC service may be a non-Java client. Also, JAX-RPC can interoperate with any Web service, whether that service is based on JAX-RPC or not. *Also note that developers need only deal with JAX-RPC APIs; all the details for handling SOAP happen under the hood.*

JAX-RPC supports three modes of operations:

1. Synchronous request–response mode—After a remote method is invoked, the service client's thread blocks until a return value or exception is returned.
2. One-way RPC mode—After a remote method is invoked, the client's thread is not blocked and continues processing. No return value or exception is expected on this call.
3. Non-blocking RPC invocation mode—A client invokes a remote procedure and continues in its thread without blocking. Later, the client processes the remote method return by performing a blocked receive call or by polling for the return value.

In addition, JAX-RPC, by specifying a standard way to plug in SOAP message handlers, allows both pre- and post-processing of SOAP requests and responses. These message handlers can intercept incoming SOAP requests and outgoing SOAP responses, allowing the service to do additional processing. See the JAX-RPC specification (listed in "References" on page 59) for more information on JAX-RPC.

Code Example 2.9 is an example of a JAX-RPC service interface for a simple Hello World service.

```
package hello;

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

public interface HelloIF extends Remote {
```

```
    public String sayHello() throws RemoteException;
}
```

Code Example 2.9 JAX-RPC Service Interface Example

Code Example 2.10 shows the implementation of the `HelloIF` JAX-RPC service interface.

```
package hello;

public class HelloImpl implements HelloIF {

    public String sayHello() {
        return("Hello World");
    }
}
```

Code Example 2.10 JAX-RPC Service Interface Implementation

A client, using the JAX-RPC stub model, accesses this HelloWorld service as shown in Code Example 2.11:

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

public class HelloClient {
    public static void main(String[] args) {
        try {
            Stub stub = createProxy();
            HelloIF hello = (HelloIF)stub;
            System.out.println(hello.sayHello());
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    private static Stub createProxy() {
        // Note: MyHello_Impl is implementation-specific.
        return (Stub)(new MyHello_Impl().getHelloIFPort());
```

```
        }
    }
```

Code Example 2.11 Using JAX-RPC to Access a Service

These examples illustrate that a developer has to code very little configuration and deployment information. The JAX-RPC implementation handles the details of creating a SOAP request, handling the SOAP response, and so forth, thereby relieving the developer of these complexities.

2.2.3 Java™ API for XML Registries

Java™ API for XML Registries (JAXR), a Java API for accessing business registries, has a flexible architecture that supports UDDI, and other registry specifications (such as ebXML). Figure 2.5 illustrates the JAXR architecture.

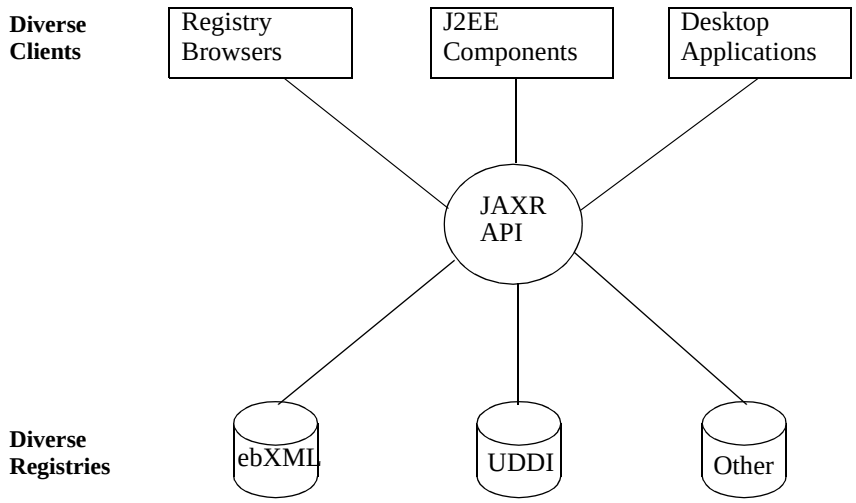

Figure 2.5 JAXR Architecture

A JAXR client, which can be a standalone Java application or a J2EE component, uses an implementation of the JAXR API provided by a JAXR provider to access business registries. A JAXR provider consists of two parts: a registry-specific JAXR provider, which provides a registry-specific implementation of the API, and a JAXR pluggable provider, which implements those features of the API

that are independent of the type of registry. The pluggable provider hides the details of registry-specific providers from clients.

The registry-specific provider plugs into the pluggable provider, and acts on requests and responses between the client and the target registry. The registry-specific provider converts client requests into a form understood by the target registry and sends the requests to the registry provider using registry-specific protocols. It converts responses from the registry provider from a registry-specific format to a JAXR response, then passes the response to the client.

Refer to the JAXR specification for more information.

2.2.4 SOAP with Attachments API for Java™

SOAP with Attachments API for Java™ (SAAJ), which enables developers to produce and consume messages conforming to the SOAP 1.1 specification and SOAP with Attachments note, provides an abstraction for handling SOAP messages with attachments. Advanced developers can use SAAJ to have their applications operate directly with SOAP messages. Attachments may be complete XML documents, XML fragments, or MIME-type attachments. In addition, SAAJ allows developers to enable support for other MIME types. JAX technologies, such as JAX-RPC, internally use SAAJ to hide SOAP complexities from developers.

SAAJ allows the following modes of message exchanges:

- Synchronous request-response messaging—the client sends a message and then waits for the response
- One-way asynchronous messaging (also called fire and forget)—the client sends a message and continues with its processing without waiting for a response

Refer to the SAAJ specification for more information.

2.2.5 Integrating Web Service Technologies into the J2EE Platform

Up to now, we have examined how the Java XML technologies support various Web service standards. Now let's see how the J2EE 1.4 platform combines these technologies into a standard platform that is portable, and integrated. Not only are the Java XML technologies integrated into the platform, the platform also defines Web service-related responsibilities for existing Web and EJB containers, artifacts, and port components. The J2EE 1.4 platform ensures portability by integrating the Java

XML technologies as extensions to existing J2EE containers, packaging formats, deployment models, and runtime services.

A Web service on the J2EE 1.4 platform may be implemented as follows:

- Using a JAX-RPC service endpoint—The service implementation is a Java class in the Web container. The service adheres to the Web container's servlet lifecycle and concurrency requirements.
- Using an EJB service endpoint—The service implementation is a stateless session bean in an EJB container. The service adheres to the EJB container's lifecycle and concurrency requirements.

In either case, the service is made portable with the definition of a port component, which provides the service's outside view for Web service implementation. A port component consists of

- A WSDL document describing the Web service that its clients can use
- A service endpoint interface defining the Web service's methods that are available to clients
- A service implementation bean implementing the business logic of the methods defined in the service endpoint interface. The implementation may be either a Java class in the Web container or a stateless session bean in the EJB container.
- Security role references enabling the service implementation to perform security checks

Container-specific service interfaces, created by the J2EE container, provide static stub and dynamic proxies for all ports. A client of a J2EE platform Web service can be a Web service peer, a J2EE component, or a standalone application. It is not required that the client be a Web service or application implemented in Java.

How do clients use a J2EE platform Web service?Here is an example of a J2EE component that is a client of some Web service. Such a client uses JNDI to look up the service, then it accesses the Web service's port using methods defined in the service interface. The client accesses the service's functionality using its service endpoint interface. A client that is J2EE component needs only consider that the Web service implementation is stateless. Thus, the client cannot depend

on the service holding state between successive service invocations. A J2EE component client does not have to know any other details of the Web service, such as how the service interface accesses the service, the service implementation, how its stubs are generated, and so forth.

To illustrate, Code Example 2.12 is a J2EE Web service interface for a Hello World service.

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

public interface HelloIntf extends Remote {

    public String sayHello() throws RemoteException;
}
```

Code Example 2.12 J2EE Web Service Interface

Code Example 2.13 shows a JAX-RPC servlet endpoint implementation of the `HelloIntf` interface.

```
import java.rmi.*;
import javax.xml.rpc.*;
import javax.xml.rpc.server.*;

public class HelloService implements HelloIntf, ServiceLifecycle {

    public void init(Object context) throws JAXRPCException {}

    public String sayHello() throws RemoteException {
        return ("Hello World");
  }

  public void destroy() {}
}
```

Code Example 2.13 JAX-RPC Servlet Endpoint Implementation

Code Example 2.14 shows the EJB service endpoint implementation for `HelloIntf`.

```
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import java.rmi.RemoteException;

public class HelloService implements SessionBean {
    private SessionContext sc;

    public HelloService(){}
    public void ejbCreate() {}
    public String sayHello() {
        return ("Hello World");
    }

    public void setSessionContext(SessionContext sc) {
        this.sc = sc;
    }
    public void ejbRemove() {}
    public void ejbActivate() {}
    public void ejbPassivate() {}
}
```

Code Example 2.14 EJB Service Endpoint Implementation

Any client can access the Hello World service using these example approaches along with well-defined configuration and deployment artifacts. Code Example 2.15 shows how a client (in this case, a servlet client) accesses the Hello World service.

```
import javax.naming.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.xml.rpc.*;

public class WebServiceClient extends HttpServlet {
```

```
    ...
    // Other servlet methods
    ...

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

       try {
           Context ic = new InitialContext();
           HelloService helloSvc =
              (HelloService) ic.lookup(
                  "java:comp/env/service/HelloService");
           HelloIntf port = helloSvc.getHelloIntfPort();
           ((Stub)port)._setProperty(
              Stub.ENDPOINT_ADDRESS_PROPERTY,
              "http://localhost:8000/webservice/HelloService");
           String ret = port.sayHello();
           System.out.println("RETURN FROM SERVICE : " + ret);
       } catch (Exception e) {
           // handle exceptions
       }
     }

    ...
    // Other servlet methods
    ...
}
```

Code Example 2.15 Accessing an EJB Service Endpoint

2.2.6 Support for WS-I Basic Profile

So far we have seen how the various Java technologies support Web service standards. We have also examined how these Java technologies have been integrated into the J2EE platform in a standard way to ensure portability of Web service implementations across J2EE platforms. Since ensuring interoperability among heterogeneous platforms is a primary force for Web services, the J2EE platform supports the WS-I Basic Profile.

As already seen in Section 2.1.5 on page 43, WS-I is an organization that spans industries and whose charter is to create and promote interoperability of Web services. WS-I has published the WS-I Basic Profile, which dictates how a set of Web service standards should be used together to ensure Interoperability. The Basic Profile covers:

- Messaging standards (such as SOAP)
- Description and discovery standards (such as UDDI)
- Security

By supporting the WS-I Basic Profile, the J2EE platform is assured of providing an interoperable and portable platform for the development of Web services.

2.3 Other Java-XML Technologies

Up to now, we have discussed the Web service-specific technologies that are a mandatory part of the J2EE platform. As such, these technologies must be present in any J2EE implementation from any vendor. Apart from these, there are other Java-XML technologies that, while not a mandatory requirement of the J2EE platform, still prove very useful for implementing Web services. While there are a number of such technologies, we discuss here only those referenced throughout this book. One such non-mandatory but useful Java-XML technology is the Java Architecture for XML Binding (JAXB), which standardizes the representation of an XML document as an in-memory object.

As we have already seen, when two parties communicate by passing XML documents between them, the XML documents should follow some structure so that the communicating parties can understand the contents of the documents. XML document structure is defined using the standard schema facility for XML documents. Of course, while developers can use a DOM or SAX parser to parse such documents, it is much easier if the various parts of the XML documents are mapped or bound to in-memory objects that truly represent the document's intended meaning, as per the schema definition. In addition to using these objects, developers have access to the schema definitions as part of their logic. Such a facility is commonly called an XML data-binding facility. JAXB provides a good quality XML data-binding facility for the J2EE platform. Figure 2.6 shows the overall architecture of the JAXB data-binding facility.

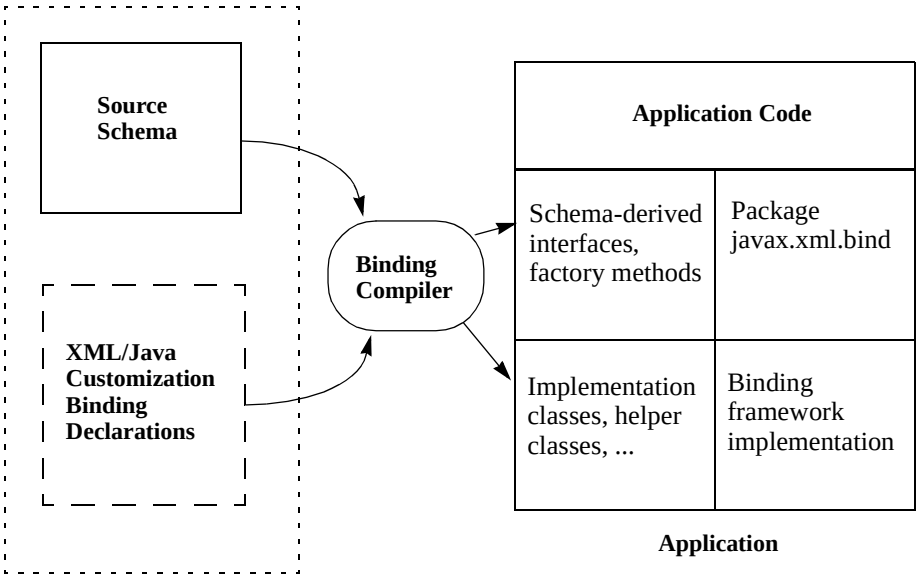

Figure 2.6 JAXB Architecture

JAXB consists of three main components:

- A **binding compiler** that creates Java classes (also called content classes) from a given schema. Complex type definitions within the schema are mapped to separate content classes, while simple types (such as attribute/element declarations) are mapped to fields within a content class. Developers use get and set methods (similar to JavaBeans get and set methods) to access and modify the object contents.
- A **binding framework** that provides runtime services—such as marshalling, unmarshalling, and validation—that can be performed on the contents classes.
- A **binding language** that describes binding of the schema to Java classes. This language enables a developer to override the default binding rules, thereby helping the developer to customize the content classes that are created by the binding compiler.

For more details on JAXB, refer to the JAXB specification available through the link provided in the next section.

Apart from JAXB, there are other emerging Java technologies that support Web service standards in terms of long-lived transactions, business process workflow, and so forth. At the time of this writing, they have not been finalized and hence will be dealt with in a future version of this book.

2.4 References

This section lists references to other sources of information about J2EE Web service technologies.

- J2EE1.4 specification @ http://java.sun.com/j2ee/download.html
- JAXP specifications @ http://java.sun.com/xml/downloads/jaxp.html
- JAXRPC specification @ http://java.sun.com/xml/downloads/jaxrpc.html
- JAXR specification @ http://java.sun.com/xml/downloads/jaxr.html
- JAXM specification @ http://java.sun.com/xml/downloads/jaxm.html
- SAAJ specification @ http://java.sun.com/xml/downloads/saaj.html
- JAXB specification @ http://java.sun.com/xml/downloads/jaxb.html
- WS-I Specifications @ http://www.ws-i.org
- SOAP specification @ http://www.w3.org
- SOAP with Attachments @ http://www.w3.org
- UDDI specification @ http://www.uddi.org/
- WSDL specification @ http://www.w3.org/TR/wsdl12

2.5 Conclusion

This chapter described the various Web service standards and the J2EE 1.4 platform technologies that support those standards. It explained why such standards—including XML, SOAP, WSDL, and UDDI—are beneficial to developers, and briefly described each standard. It showed how the platform integrates existing Java and Web service/XML technologies to allow existing and new applications to be exposed as Web services.

In addition, this chapter described the different J2EE platform XML-related APIs, including JAXP, JAXR, JAX-RPC, and SAAJ. It described these technologies from a high-level architectural point of view, and, where appropriate, illustrated their use with sample code. It also showed how to implement a Web service on a J2EE platform using either a JAX-RPC or EJB service endpoint.

Table 2.1 summarizes the standards supported by the different J2EE platform technologies.

Table 2.1 J2EE 1.4 Platform Web Service Support

Technology Name	Supporting Standard	Purpose
JAXP	XML schema	Enables processing of XML documents in a vendor neutral way; supports SAX and DOM models
JAX-RPC	SOAP	Enables exchange of SOAP requests and responses through an API that hides the complex SOAP details from the developers
JAXR	UDDI, ebXML	Enables accessing business registries with an API that supports any type of registry specification
SAAJ	SOAP with attachments	Enables exchange of document-oriented XML messages using Java APIs
J2EE for Web Services	Integrates Java XML technologies into the J2EE platform; supports WS-I Basic Profile	Enables development and deployment of portable and interoperable Web services on the J2EE platform
JAXB	Standard in-memory representation of an XML document	Provides an XML data-binding facility for the J2EE platform

Now that you have a good grasp of the Web services technologies, you are ready to proceed with specific design and implementation issues. Chapter 3 describes how to design and implement an endpoint so that your application can make its functionality available as a Web service.