Article

Service-Oriented Architecture and Web Services: Concepts, Technologies, and Tools

By Ed Ort, April 2005

Articles Index

Service-oriented architecture (SOA) is a hot topic in enterprise computing because many IT professionals see the potential of an SOA -- especially a web services-based SOA -- in dramatically speeding up the application development process. They also see it as a way to build applications and systems that are more adaptable, and in doing so, they see IT becoming more agile in responding to changing business needs. Not only is SOA a hot topic, but it's clearly the wave of the future. Gartner reports that "By 2008, SOA will be a prevailing software engineering practice, ending the 40-year domination of monolithic software architecture" and that "Through 2008, SOA and web services will be implemented together in more than 75 percent of new SOA or web services projects." But despite this strong trend, some in the IT community don't feel that the web services underpinning for an SOA is mature enough for their enterprise to consider migration to a service-oriented architecture. For others, the terms service-oriented architecture and web services draw a blank stare.

An earlier article, presented a brief overview of SOA and the role of web services in realizing it. This article supplements that earlier article. If you're not familiar with SOA and web services, this article aims to familiarize you with them. It defines some of the key terms and concepts related to SOA and web services. A critical mass of widely-adopted technologies is available now to implement and use a web services-based SOA, and more technologies, as well as tools, are on the way. Figure 1 identifies some of these technologies and tools. Each layer of the figure shows technologies or tools that leverage technologies in the surrounding layers. This article describes what these technologies and tools are.

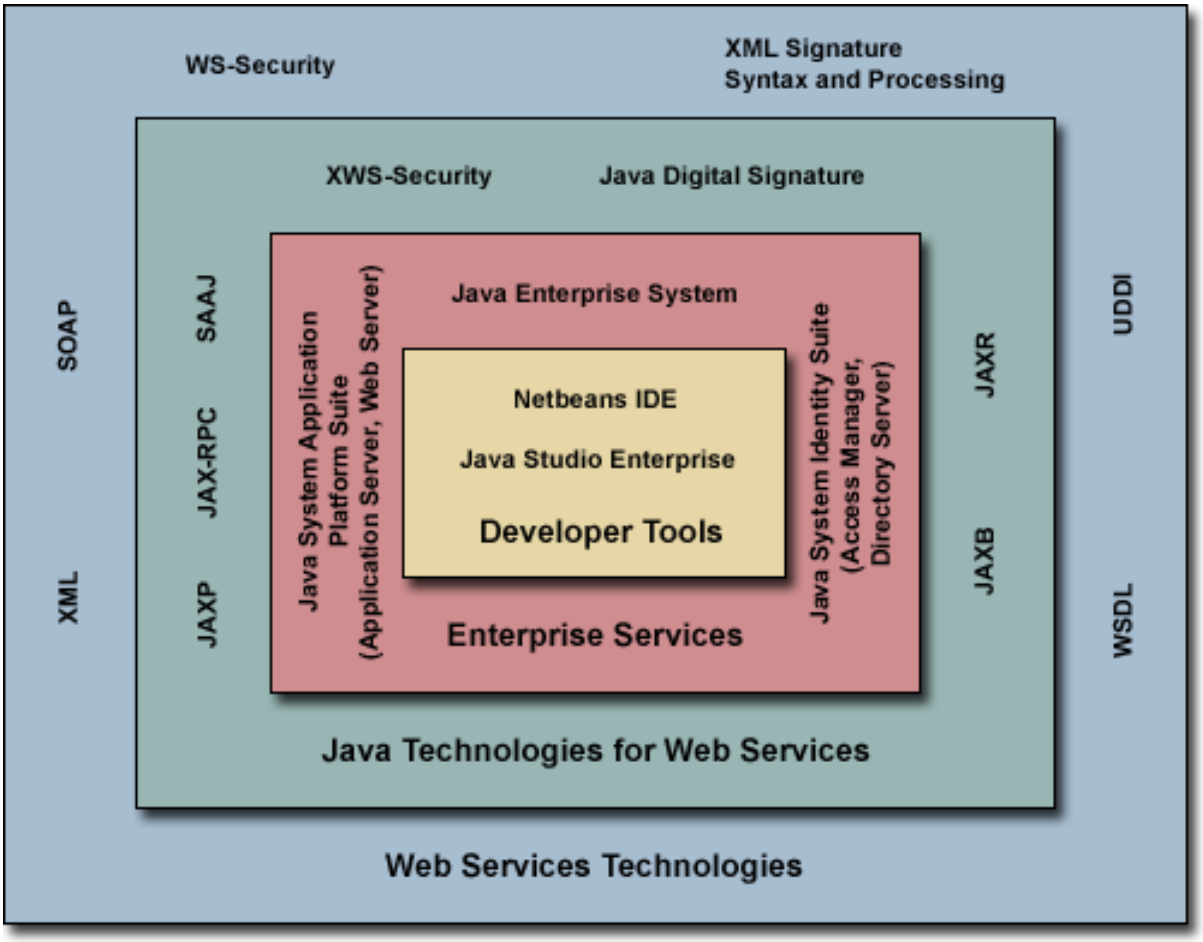


Figure 1: Web Services-Based Technologies and Tools

A critical mass of widely-adopted technologies is available now to implement and use a web services-based SOA, and more technologies, as well as tools, are on the way.

Table of Contents

Terms and Concepts

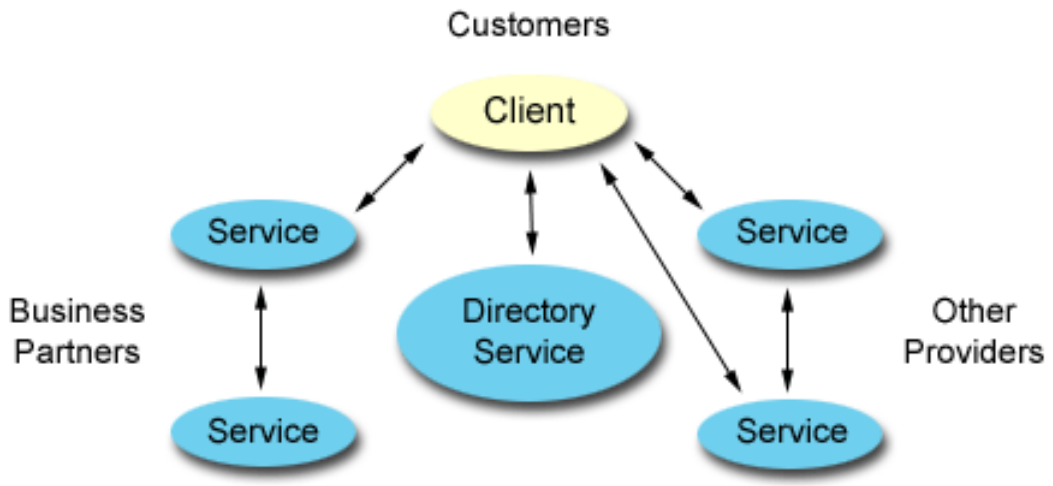


Figure 2: A Service-Oriented Architecture

A *service-oriented architecture* is an information technology approach or strategy in which applications make use of (perhaps more accurately, rely on) services available in a network such as the World Wide Web. Implementing a service-oriented architecture can involve developing applications that use services, making applications available as services so that other applications can use those services, or both.

A *service* provides a specific function, typically a business function, such as analyzing an individual's credit history or processing a purchase order. A service can provide a single discrete function, such as converting one type of currency into another, or it can perform a set of related business functions, such as handling the various operations in an airline reservations system. Services that perform a related set of business functions, as opposed to a single function, are said to be "coarse grained." Multiple services can be used together in a coordinated way. The aggregated, or composite, service can be used to satisfy a more complex business requirement. In fact, one way of looking at an SOA is as an approach to connecting applications (exposed as services) so that they can communicate with (and take advantage of) each other. In other words, a service-oriented architecture is a way of sharing functions (typically business functions) in a widespread and flexible way.

A service-oriented architecture is a way of sharing functions (typically business functions) in a widespread and flexible way.

The concept of an SOA is not new. Service-oriented architectures have been used for years. What distinguishes an SOA from other architectures is loose coupling. Loose coupling means that the client of a service is essentially independent of the service. The way a client (which can be another service) communicates with the service doesn't depend on the implementation of the service. Significantly, this means that the client doesn't have to know very much about the service to use it. For instance, the client doesn't need to know what language the service is coded in or what platform the service runs on. The client communicates with the service according to a specified, well-defined interface, and then leaves it up to the service implementation to perform the necessary processing. If the implementation of the service changes, for instance, the airline reservations application is revised, the client communicates with it in the same way as before, provided that the interface remains the same. Loose coupling enables services to be document-oriented (or document-centric). A document-oriented service accepts a document as input, as opposed to something more granular like a numeric value or Java object. The client does not know or care what business function in the service will process the document. It's up to the service to determine what business function (or functions) to apply based on the content of the document.

However what is relatively new is the emergence of web services-based SOAs. A *web service* is a service that communicates with clients through a set of standard protocols and technologies. These web services standards are implemented in platforms and products from all the major software vendors, making it possible for clients and services to communicate in a consistent way across a wide spectrum of platforms and operating environments. This universality has made web services the most prevalent approach to implementing an SOA.

Web services standards are implemented in platforms and products from all the major software vendors. This universality has made web services the most prevalent approach to implementing an SOA.

Optionally, an SOA can also include a service that provides a directory or *registry* of services. The registry contains information about the service such as it's interface. A client can discover services by examining the registry. A registry can also be coupled with a *repository* component that stores additional information about each service. This additional "metadata" can include business process information such as policy statements.

Why SOA

There are many reasons for an enterprise to take an SOA approach, and more specifically, a web services-based SOA approach. Some of the primary reasons are:

Reusability. What drives the move to SOA is reuse of business services. Developers within an enterprise and across enterprises (particularly, in business partnerships) can take the code developed for existing business applications, expose it as web services, and then reuse it to meet new business requirements. Reusing functionality that already exists outside or inside an enterprise instead of developing code that reproduces those functions can result in a huge savings in application development cost and time. The benefit of reuse grows dramatically as more and more business services get built, and incorporated into different applications. A major obstacle in taking advantage of existing code is the uniqueness of specific applications and systems. Typically, solutions developed in different enterprises, even different departments within the same enterprise, have unique characteristics. They run in different operating environments, they're coded in different languages, they use different programming interfaces and protocols. You need to understand how and where these applications and systems run to communicate with them. The work involved in doing this analysis and the development effort in tying these pieces together can be very time consuming. Witness the pain IT

organizations generally encounter when they try to integrate their applications with systems from business partners (or even with legacy systems from other parts of their own company). In an SOA, the only characteristic of a service that a requesting application needs to know about is the public interface. The functions of an application or system (including legacy systems) can be dramatically easier to access as a service in an SOA than in some other architecture. So integrating applications and systems can be much simpler.

The functions of an application or system (including legacy systems) can be dramatically easier to access as a service in an SOA than in some other architecture. So integrating applications and systems can be much simpler.

Interoperability. The SOA vision of interaction between clients and loosely-coupled services means widespread interoperability. In other words, the objective is for clients and services to communicate and understand each other no matter what platform they run on. This objective can be met only if clients and services have a standard way of communicating with each other -- a way that's consistent across platforms, systems, and languages. In fact, web services provide exactly that. Web services comprise a maturing set of protocols and technologies that are widely accepted and used, and that are platform, system, and language independent. In addition, these protocols and technologies work across firewalls, making it easier for business partners to share vital services. Promising to make things even more consistent is the WS-I basic profile, introduced by the Web Services Interoperability Organization (an organization chartered to promote web services interoperability). The WS-I basic profile identifies a core set of web services technologies that when implemented in different platforms and systems, helps ensure that services on these different platforms and systems, and written in different languages, can communicate with each other. The WS-I basic profile has widespread backing in the computer industry, virtually guaranteeing interoperability of services that conform to the profile.

Scalability. Because services in an SOA are loosely coupled, applications that use these services tend to scale easily -- certainly more easily than applications in a more tightly-coupled environment. That's because there are few dependencies between the requesting application and the services it uses. The dependencies between client and service in a tightly-coupled environment are compounded (and the development effort made significantly more complex) as an application that uses these services scales up to handle more users. Services in a web services-based SOA tend to be coarse-grained, document-oriented, and asynchronous. As mentioned earlier, coarse-grained services offer a set of related business functions rather than a single function. For example, a coarse-grained service might handle the processing of a complete purchase order. By comparison, a fine-grained service might handle only one operation in the purchase order process. Again, as mentioned earlier, a document-oriented service accepts a document as input, as opposed to something more granular like a numeric value or Java object. An example of a document-oriented service might be a travel agency service that accepts as input a document that contains travel information for a specific trip request. An asynchronous service performs its processing without forcing the client to wait for the processing to finish. A synchronous service forces the client to wait. The relatively limited interaction required for a client to communicate with a coarse-grained, asynchronous service, especially a service that handles a document such as a purchase order, allows applications that use these services to scale without putting a heavy communication load on the network.

Flexibility. Loosely-coupled services are typically more flexible than more tightly-coupled applications. In a tightly-coupled architecture, the different components of an application are tightly bound to each other, sharing semantics, libraries, and often sharing state. This makes it difficult to evolve the application to keep up with changing business requirements. The loosely-coupled, document-based, asynchronous nature of services in an SOA allows applications to be flexible, and easy to evolve with changing requirements.

Cost Efficiency. Other approaches that integrate disparate business resources such as legacy systems, business partner applications, and department-specific solutions are expensive because they tend to tie these components together in a customized way. Customized solutions are costly to build because they require extensive analysis, development time, and effort. They're also costly to maintain and extend because they're typically tightly-coupled, so that changes in one component of the integrated solution require changes in other components. A standards-based approach such as a web services-based SOA should result in less costly solutions because the integration of clients and services doesn't require the in-depth analysis and unique code of customized solutions. Also, because services in an SOA are loosely-coupled, applications that use these services should be less costly to maintain and easier to extend than customized solutions. In addition, a lot of the Web-based infrastructure for a web services-based SOA is already in place in many enterprises, further limiting the cost. Last, but not least, SOA is about reuse of business functions exposed as coarse-grained services. This is potentially the biggest cost saving of all.

Web Services Protocols and Technologies

The web services approach is based on a maturing set of standards that are widely accepted and used. This widespread acceptance makes it possible for clients and services to communicate and understand each other across a wide variety of platforms and across language boundaries. This section briefly describes the protocols and technologies that constitute these web services standards, some of which are mainstays of working on the World Wide Web. In addition to these standards, other web services standards are emerging to meet additional requirements -- especially in the areas of security and management. This section also covers some of these emerging web services standards.

XML

eXtensible Markup Language (XML) has become the de facto standard for describing data to be exchanged on the Web. As it's name indicates, XML is a markup language. It involves the use of tags that "mark up" the contents of a document, and in doing so, describe the contents of a document. An XML tag identifies information in a document, and also identifies the structure of the information. For example, the following XML markup identifies some information as `bookshelf`. The XML markup also describes the structure of `bookshelf`. The `bookshelf` structure includes two subordinate items identified as `book`. Each `book` has three subordinate items identified as `title`, `author`, and `price`.

```
<bookshelf>
   <book>
      <title>My Life and Times</title>
      <author>Felix Harrison</author>
      <price>39.95</price>
   </book>
</bookshelf>
```

Although tags such as `<book>` and `<title>` might appear to inherently give meaning to the information they identify, they really don't. Information tagged with XML tags has meaning only if people associate a particular meaning with a particular tag. If people (1) agree on the meaning of a tag, say that the `<book>` tag is used to identify a book, and that `<title>`, `<author>`, and `<price>` tags identify the title, author, and price of the book, respectively, and (2) use those tags consistently, it gives people a way to exchange data. Their applications can send XML documents to each other, and process the information in those documents, relying on the commonly understood meanings associated with the XML tags. Applications capable of interpreting these tags can then process the information according to the meaning of the information and its organization. XML

documents have a well-formed structure, for example, each XML tag must have an ending tag (such as `<bookshelf>...</bookshelf>`) and any tag that begins within another tag must end before the end of the other tag, and must be completely nested within that other tag. So that `</title>`, `</author>`, and `</price>` all come before `</book>`, and in that relative order. An XML document is typically associated with a schema that specifies its "grammar rules." In other words, the schema specifies what tags are allowed in the document, the structure of those tags, and other rules about the tags, such as what type of data is expected in a tag (or no data if it's an empty tag). Because valid XML documents must be well-formed and conform to the associated schema, it makes it relatively easy to process XML documents. As a result, XML has been generally adopted as the data language for web services.

SOAP

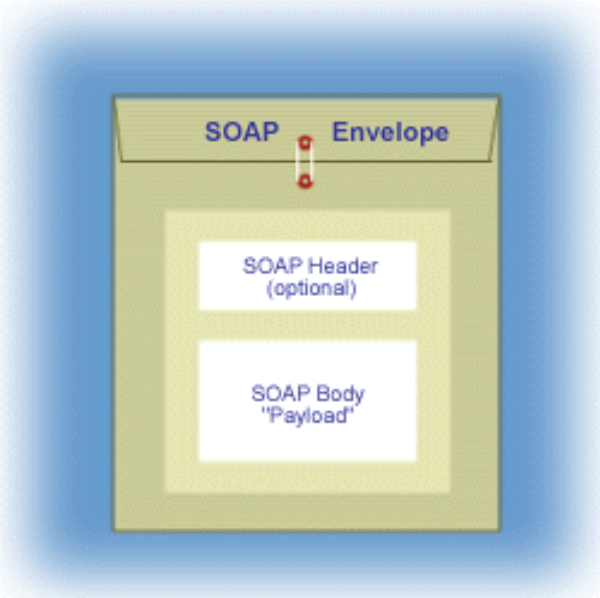


Figure 3: A SOAP Message (Conceptually)

Though agreeing on the meaning and structure of XML tags makes the use of XML an effective way to exchange data, it's not sufficient for data interchange over the Web. For instance, you still need some agreed-upon protocol for formatting an XML document so that the receiver understands what the main, "payload," part of the message is, and what part contains additional instructions or supplemental content. That's where Simple Object Access Protocol (SOAP) comes in. SOAP is an XML-based protocol for exchanging information in a distributed environment. SOAP provides a common message format for exchanging data between clients and services. The basic item of transmission in SOAP is a SOAP message, which consists of a mandatory SOAP envelope, an optional SOAP header, and a mandatory SOAP body. The envelope specifies two things: an XML namespace and an encoding style. The XML namespace specifies the names that can be used in the SOAP message. Namespaces are designed to avoid name clashes -- the same name can be used for different items, provided that the names
are in different namespaces. The encoding style identifies the data types recognized by the SOAP message. If a header is provided, it extends the SOAP message in a modular way. It's important to understand that as a SOAP message travels from a client to a service, it can pass through a set of intermediate nodes along the message path. Each node is an application that can receive and forward SOAP messages. An intermediate node can provide additional services such as transform the data in the message or perform security-related operations. The SOAP header can be used to indicate some additional processing at an intermediate node, that is, processing independent of the processing done at the final destination. Typically, the SOAP header is used to convey security-related information to be processed by runtime components. The body contains the main part of the SOAP message, that is, the part intended for the final recipient of the SOAP message.

Here's an example of a SOAP message designed to retrieve the price of a book. Notice the `<SOAP-ENV: Envelope>`, `<SOAP-ENV:Header>` and `<SOAP-ENV: Body>` elements that mark the envelope, header, and body parts of the SOAP message, respectively. The attribute value `mustUnderstand=1` in the header means that the receiver of the SOAP message must process it.

```
<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">
           </t:Transaction>
        </SOAP-ENV:Header>
        <SOAP-ENV:Body>
           <m:GetBookPrice xmlns:m="some-URI">
              <title>My Life and Times</title>
              <author>Felix Harrison</author>
           </m: GetBookPrice>
        </SOAP-ENV:Body>
</SOAP-Envelope>
```

A related standard, SOAP Messages With Attachments, specifies the format of a SOAP message that includes attachments (such as, images). SOAP messages (with or without attachments) are independent of any operating system or platform and can be transported using a variety of communication protocols, such as HTTP or SMTP. WS-I Basic Profile 1.0 references SOAP 1.1.

WSDL

How does a client know what format to use in making a request to a service? For that matter, how do the client and the service know what the request means? The answers to these questions are provided by information in an XML document, called a WSDL document, that contains a description of the web service's interface. 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 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 registration, which can be in a UDDI registry or an ebXML registry/repository. A typical scenario is that a business registers its service. 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 information in the WSDL document to construct the appropriate calls to the service.

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 client 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.

Here, for example, is a WSDL document for an online book search service. Notice in the example that the WSDL document specifies one operation for the service: `getBooks`:

```
<operation name="getBooks" ...
```

The input message for the operation, `BookSearchInput` contains a string value named `isbn`:

```
<complexType>
    <all>
       <element name="isbn"
       type="string"/>
    </all>
```

The output message for the operation, `BookSearchOutput` returns a string value named `title`:

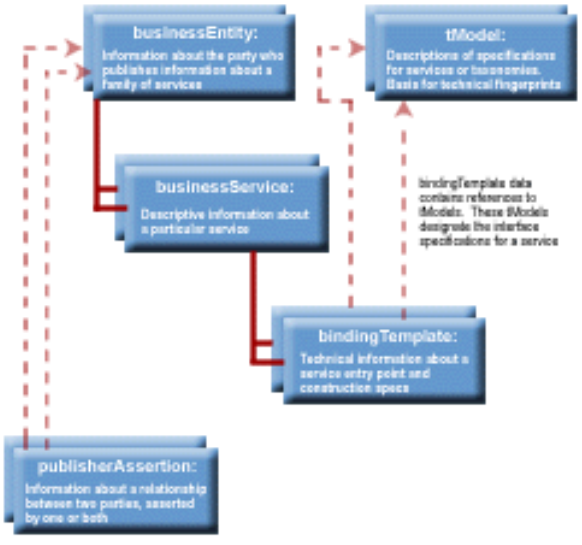
```
<complexType>
    <all>
       <element name="title"
             type="string"/>
    </all>
```

Notice too that the WSDL document specifies a SOAP protocol binding. The style attribute in the `binding` element specifies that the receiver should interpret the payload in the SOAP message as an RPC method call:

```
<soap:binding
     transport="transport=http://schemas.xmlsoap.org/soap/http"
           style="rpc"/>
```

Alternatively, the style attribute in a `binding` element could specify `"document"`. In that case, a complete document (typically an XML document) would be exchanged in the call. In fact, the document style of binding is more typical of services in an SOA. One of the advantages of passing an XML document to a service instead of an RPC-style message is that it tends to be easier to validate and to apply business rules.

The WS-I Basic Profile 1.0 specifies the constructs in WSDL 1.1 that be can be used to ensure interoperability. It also clarifies some of the construct descriptions in the WSDL 1.1 specification.



UDDI and ebXML

As mentioned earlier, an SOA can also include a service that provides a directory or registry of services. But how can a

service be described in a registry so that a program looking for that service can easily find it and understand what it does? The Universal Description, Discovery, and Integration (UDDI) specifications define how to publish and discover information about services in a UDDI-conforming registry. More specifically, the specifications define a UDDI schema and a UDDI API. The UDDI schema identifies the types of XML data structures that comprise an entry in the registry for a service. Think of a UDDI registry as a "Yellow Pages" for web services. Like the Yellow Pages directory for phone numbers, a UDDI registry provides information about a service such as the name of the service, a brief description of what it does, an address where the service can be accessed, and a description of the interface for accessing the service. The accompanying figure illustrates the schema. It shows the five types of XML data structures that comprise a registration.

Figure 4: UDDI Schema *(Click image to enlarge)*

Here is an example that shows part of a complete `BusinessEntity` structure for a hypothetical company named BooksToGo. The company provides various web services, including an online book ordering service.

```
<businessEntity businessKey="35AF7F00-1419-11D6-A0DC-000C0E00ACDD"
authorizedName="0100002CAL"
  operator="www-3.ibm.com/services/uddi">
<name>BooksToGo</name>
<description xml:lang="en">
   The source for all professional books
</description>
<contacts>
<contact>
<personName>Benjamin Boss</personName>
<phone>
  (877)1111111
</phone>
</contact>
</contacts>
```

The API describes the SOAP messages that are used to publish an entry in a registry, or discover an entry in a registry. Here, for example, is a message that searches for all business entities in a registry whose name begins with the characters "Books":

```
<find_business generic="2.0" xmlns=um:uddi-org:api-v2">
<name>Books%</name>
</find_business>
```

UDDI 2.0 is part of the WS-I Basic Profile 1.1

Note that other registry approaches, with their own specifications, exist. For example, ebXML is a comprehensive business-to-business framework that includes a registry as well as other information needed for business collaboration in a global electronic marketplace. The framework was developed as a joint effort between the Organization for the Advancement of Structured Information Standards (OASIS) and United Nations Centre for Trade Facilitation and Electronic Business (UN/CEFACT), and includes representation from an extremely wide set of businesses and other entities around the world (including standards bodies). The

registry contains a Collaboration-Protocol Profile (CPP) and a Collaboration-Protocol Agreement (CPA). A CPP is an XML document that contains information about a business (in ebXML terms, a business is referred to as a "Party") and the way it exchanges information with another Party. A CPA is an XML document that describes the specific capabilities that two Parties have agreed to use in a business collaboration.

Emerging Standards

Standards such as XML, SOAP, UDDI, and WSDL address the basics of interoperable services. They ensure that a client can find a needed service and make a request that both the client and service understand, irrespective of where the client and service reside or what language the client or service is coded in. But for a web services-based SOA to become a mainstream IT practice, other standards (what many people consider "higher-level" standards) need to be added and adopted. This is especially true in the areas of web service security and web service management. Various standards organizations, such as the World Wide Web Consortium (W3C) and (OASIS), have drafted standards in these areas that promise to gain universal acceptance. Two emerging standards of special interest are WS-Security and WS-BPEL.

WS-Security

WS-Security is a standard released by OASIS in March 2004. It describes security-related enhancements to SOAP messaging that provide for message integrity and confidentiality. Integrity means that a SOAP message is not tampered with as it travels from a client to its final destination. Confidentiality means that a SOAP message is only seen by intended recipients. WS-Security uses security tokens to enable SOAP message security and integrity. A security token is a collection of claims made by the sender of a SOAP message (such as the sender's identity). A sender is authenticated by combining a security token with a digital signature -- the signature is used as proof that the sender is indeed associated with the security token. The standard also provides a general-purpose mechanism for associating security tokens with messages, and describes how to encode binary security tokens.

WS-Security is quite flexible and can be used with a wide variety of security models and encryption technologies, such as Public-key infrastructure (PKI) and Kerberos, as well as the Secure Socket Layer (SSL)/ Transport Layer Security (TLS) protocol that provides basic end-to-end security communication on the Internet.

Here is an example taken from the WS-Security standard that illustrates a SOAP message with a security token. The example also lists and describes the lines that demonstrate WS-Security enhancements.

Other important technologies are emerging in the security area. One of them is Security Markup Assertion Language (SAML). As described by the OASIS Technical Committee responsible for it, SAML "is an XML-based framework for exchanging security information. This security information is expressed in the form of assertions about subjects, where a subject is an entity (either human or computer) that has an identity in some security domain." Through SAML, multiple services can exchange security information. As a result, services could do things like enable a single sign-on approach, where a single security entry (of say a user ID and password) could be used to sign in to multiple, related services.

Building on web services security standards such as WS-Security and SAML, the Liberty Alliance Project, a global consortium for open federated identity standards and identity-based web services, has delivered a number of specifications for identity-based web services. Conformance to these standards will enable web services to use a single identity framework. This will support, among other things, single sign-on, that is, a user will only have to sign in once, even if the user's request invokes a number of services, each requiring authentication.

WS-BPEL

Many business processes, such as the handling of a purchase order, involve multiple steps performed in a specific sequence, potentially requiring the invocation and interaction of multiple services. For this to work properly, the service invocations and interactions need to be coordinated (service coordination is also known as "orchestration") WS-BPEL (Web Services Business Process Execution Language), also identified as BPELWS, BPEL4WS, or simply BPEL, is an XML-based language that is used to coordinate web services across a single business process. WS-BPEL uses WSDL to describe the web services that participate in a process and how the services interact. For example, the following WSDL entry describes one of the services in the purchase order process:

```
<portType name="purchaseOrderPT">
   <operation name="sendPurchaseOrder">
      <input message="pos:POMessage"/>
      <output message="pos:InvMessage"/>
      <fault name="cannotCompleteOrder"
             message="pos:orderFaultType"/>
   </operation>
</portType>
```

The following entry describes part of the flow of the purchase order process:

```
<sequence>
      <receive partnerLink="purchasing"
               portType="lns:purchaseOrderPT"
               operation="sendPurchaseOrder"
               variable="PO">
      </receive>
      <flow>
         <links>
            <link name="ship-to-invoice"/>
            <link name="ship-to-scheduling"/>
         </links>
```

An OASIS Web Services Business Process Execution Language Technical Committee has been established to continue work on the BPEL specification.

Java Technologies for Web Services

Although web services are designed to be language and platform neutral, the Java programming language is ideal for developing web services and applications that use web services. The portability and interoperability of applications written in the Java programming language mesh well with the objective of web service interoperability. A core set of Java technologies for web services is integrated into the Java 2 Platform, Enterprise Edition (J2EE) 1.4 platform. These technologies are designed for use with XML, and conform to web services standards such as SOAP, WSDL, and UDDI. You can take advantage of the technologies by developing and deploying web services and applications to the J2EE 1.4 platform. (In addition, the J2EE 1.4 platform also offers a wide variety of enterprise application features such as resource pooling and transaction management.) Implementations of the Java technologies for web services are available in the J2EE 1.4 SDK

and Sun Java Application Server 8.1. The J2EE 1.4 SDK and Sun Java Application Server 8.1 are available as a single, "all-in-one" bundle, or available separately.

The Java programming language is ideal for developing web services and applications that use web services.

Supplementing J2EE 1.4 is the Java Web Services Developer Pack (Java WSDP) 1.5, which provides implementations of additional Java technologies for web services as well as updates to the web service technology implementations in the J2EE 1.4 SDK and Sun Java Application Server 8.1.

This section briefly describes the Java technologies for web services and their implementations in J2EE 1.4 and Java WSDP 1.5.

Web Services Technologies in J2EE 1.4

The Java technologies for web services in the J2EE 1.4 platform are:

- Java API for XML Processing (JAXP) 1.2
- Java API for XML-based RPC (JAX-RPC) 1.1
- SOAP with Attachments API for Java (SAAJ) 1.2
- Java API for XML Registries (JAXR) 1.0

Java API for XML Processing (JAXP) 1.2

Java API for XML Processing (JAXP) is a Java API for processing XML documents. Using JAXP, you can invoke a SAX or DOM parser in an application to parse an XML document. A parser is a program that scans the XML document and logically breaks it up into discrete pieces. It also checks that the content is well-formed. Some parsers also validate an XML document against an associated XML Document Type Definition (DTD) or XML schema. The parsed content is then made available to the application. Recall that XML has been generally adopted as the data language for web services. It's the language that's used in documents that are exchanged between clients and web services. So an XML parser, a program that essentially feeds the contents of an XML document to an application, is an important part of a web services-based SOA. The primary functional addition in JAXP 1.2 over previous JAXP releases is support for W3C XML Schema.

Simple API for XML Parsing (SAX) and Document Object Model (DOM) are parsing standards, and are the most frequently used approaches to parsing XML documents. In the SAX approach, the parser starts at the beginning of the document and passes each piece of the document to the application in the sequence it finds it. Nothing is saved in memory. The application can take action on the data as it gets it from the parser, but it can't do any in-memory manipulation of the data. For example, it can't update the data in memory and return the updated data to the XML file. In the DOM approach, the parser creates a tree of objects that represents the content and organization of data in the document. In this case, the tree exists in memory. The application can then navigate through the tree to access the data it needs, and if appropriate, manipulate it.

Here is a simple example that illustrates how JAXP is used to invoke a SAX parser:

```
import javax.xml.parsers.SAXParserFactory;
import javax.xml.parsers.SAXParser;

public class SAXRead
    extends DefaultHandler {

    public void ParseSAX() {

        // Create a SAX Parser Factory instance
        SAXParserFactory spf = SAXParserFactory.newInstance();

        // Turn on validation and namespace awareness
        spf.setValidating(true);
        spf.setNamespaceAware(true);

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

        // Get an XMLReader
        xmlReader = saxParser.getXMLReader();

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

        // Parse the XML document
        xmlReader.parse(XMLDocumentName);
        }
}
```

First, an instance of the `SAXParserFactory` class is used to generate an instance of a `SAXParser` class. This is the SAX parser. Next, the `SAXParser` class is used to get an XML reader that implements the `XMLReader` interface. The parser must implement this interface. The parser (through the `XMLReader` interface) reads the XML document. Notice that the `XMLReader` implementation is also used to set a content handler that does any parsing-related processing. The content handler part of the application would define methods to be notified by the parser when the parser encounters something significant (in SAX terms, an "event") such as the start of an XML tag, or the text inside of a tag. These methods, known as callback methods, take any subsequent actions based on the event.

Also notice that the `SAXParserFactory` can be configured to set various parser characteristics. In this example, validation and namespace awareness are turned on. This means that the parser will verify that the contents of the XML document conforms to the associated schema, and that the parser will be aware of namespaces.

Here is a simple example that illustrates how JAXP is used to invoke a DOM parser:

```
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import org.w3c.dom.Document;

public class DOMRead {

    public void parseDOM() {

        // Create a DocumentBuilderFactory instance
        DocumentBuilderFactory dbf =
           DocumentBuilderFactory.newInstance();

        // Turn on validation and namespace awareness
        dbf.setValidating(true);
        dbf.setNamespaceAware(true);

        // Create a DocumentBuilder instance
        db = dbf.newDocumentBuilder();

        // Parse the input file
        Document doc = db.parse(XMLDocumentFile);

        // Parse the tree
        }
}
```

As in SAX parsing, a factory is used to create an instance of a DOM parser. However, unlike SAX parsing, DOM parsing does not use a content handler or callback methods. The `parse` method of the `DocumentBuilder` instance returns a `Document` object that represents the entire XML document as a tree. The application must then explicitly traverse the tree and process what it finds.

JAXP comes with its own parser. (The J2EE 1.4 SDK implementation of JAXP 1.2 uses Xerces 2 as the default parser.) However the API is designed to allow any parser to be plugged in and used (provided that the parser conforms to the API).

In addition to its features for invoking parsers, JAXP can be used to transform XML documents (for example, to HTML) in conformance with the XSL Transformations (XSLT) specification. The J2EE 1.4 SDK implementation of JAXP 1.2 includes an XSLT stylesheet compiler called XSLTC.

Java API for XML-based RPC (JAX-RPC) 1.1

Java API for XML-Based RPC (JAX-RPC) is a Java API for accessing services through XML (SOAP-based) RPC calls. The API incorporates XML-based RPC functionality according to the SOAP 1.1 specification. JAX-RPC 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 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. 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.

JAX-RPC is 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. JAX-RPC on the server converts the SOAP message and then calls the web service. Then the sequence is reversed. The web service returns the response. JAX-RPC on the server converts the response to a SOAP message, which is then transported back to the client. JAX-RPC on the client converts the SOAP message and then returns the response to the application.

To make a web service available to clients through JAX-RPC, you need 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 (which identifies the remote methods that can be called by the client), and the other that implements the interface. The JAX-RPC specification defines the mapping between the definition of a JAX-RPC service endpoint and a WSDL service description. In fact, all JAX-RPC implementations must be able to produce a WSDL document from a service endpoint definition. However the mapping also makes it possible for an implementation to do the reverse -- produce a JAX-RPC service endpoint definition from a WSDL document. The J2EE 1.4 SDK provides a mapping tool, called `wscompile`. You can use a mapping tool such as `wscompile` to generate a WSDL file from a JAX-RPC service endpoint definition, or a JAX-RPC service endpoint definition from a WSDL file. The latter ability is important for web services that are not on a Java platform.

Here's an example of a JAX-RPC service endpoint definition for a web service that returns stock quotes. First, here's the JAX-RPC service endpoint interface:

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

public interface StockQuoteProvider extends Remote {

public float getLastTradePrice(String tickerSymbol)
                               throws RemoteException;
}
```

Next, the implementation class:

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

public class StockQuoteService implements
               StockQuoteProvider, ServiceLifecycle {

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

}
```

After a service endpoint is defined, the classes are compiled and deployed in a container that implements the JAX-RPC runtime system on the server. A tool such as `wscompile` can then be used to generate a WSDL file

for the service as well as other classes, interfaces, and files that are needed to handle communication between the client and the service endpoint. The classes, interfaces, and files are collectively called "artifacts."

JAX-RPC provides a lot of flexibility in the way a client can invoke a method on a web service. The client can invoke the remote method through a local object called a stub (this is one of the artifacts generated by `wscompile`). Alternatively, the client can use a dynamic proxy to invoke the method. Or the client can dynamically invoke the method using the JAX-RPC Dynamic Invocation Interface (DII). Stubs are used when a JAX-RPC client knows what method to call and how to call it (for example, what parameters to pass). A dynamic proxy is a class that dynamically supports service endpoints at runtime, without the need to generate stubs. DII gives a client a way to invoke a remote method dynamically, for example, when a client doesn't know the remote method name or its signature until run time.

Invoking a remote method through a stub is like invoking a remote method using Java Remote Method Invocation (RMI). 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.

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 the stock quote web service:

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

        }
}
```

SOAP with Attachments API for Java (SAAJ) 1.2

SOAP with Attachments (SAAJ) is another API (that is, in addition to JAX-RPC) for creating and sending SOAP messages. In fact, SAAJ is used, "under the covers," by other Java for XML APIs, such as JAX-RPC and JAXR, to create and send SOAP messages. The SAAJ API 1.2 conforms to the SOAP 1.1 specification and the SOAP with Attachments specification. This means that you can use SAAJ to create and send SOAP message with or without attachments.

To create a SOAP message using SAAJ for sending to a web service, a client gets a connection to the service, creates a message, adds content to the message, and then adds attachments (if any).

Here's an example that illustrates the steps:

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

// Get a connection
SOAPConnection conn;
SOAPConnectionFactory scf =
        SOAPConnectionFactory.newInstance();
conn = scf.createConnection();

// Create a message
MessageFactory mf = MessageFactory.newInstance();
SOAPMessage msg = mf.createMessage();

// Add content to the message
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("author");
sbe.addChildElement( envelope.createName(
                 "searchValue", "bp",
  "http://www.bookprovider.com" )).addTextNode("Hemingway");

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

// Close the connection
connection.close();
```

Notice that the client constructs the content part-by-part, following the SOAP message structure shown in the earlier discussion of SOAP.

After creating the message, the client can use SAAJ to send the message synchronously (and then wait for a reply), or asynchronously (and continue processing without waiting for a reply). Here's an example of sending the message synchronously:

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

The web service then processes the message and returns a reply:

```
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"));
while ( sbes.hasNext() ) {

...
```

Java API for XML Registries (JAXR) 1.0

Java API for XML Registries (JAXR) is a Java API that you can use to access standard registries such as those that conform to UDDI or ebXML. Using the API, you can register a service in a registry or discover services in a registry. JAXR 1.0 is compatible with UDDI 2 and the ebXML registry specifications. However the JAXR 1.0 implementation in the J2EE 1.4 SDK currently supports access to UDDI 2 registries only. Note though that support for ebXML registries is coming soon -- see The Future: Sun's SOA Initiative.

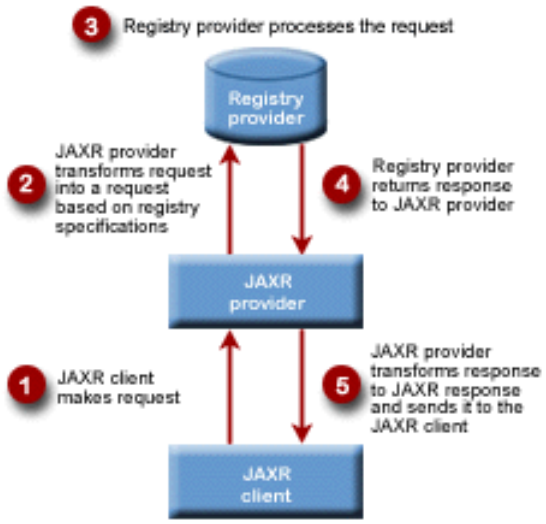


Figure 5: Processing a JAXR Request *(Click image to enlarge)*

There are three roles that take part in issuing and handling requests made through the JAXR API: JAXR clients, JAXR providers, and registry providers. The JAXR API provides interfaces and classes for use by JAXR clients and providers. A JAXR provider implements the JAXR API. For example, the implementation of the JAXR API in the J2EE 1.4 SDK is a JAXR provider. A registry provider is an implementation of a registry specification, for instance, an actual UDDI registry.

Here's how the roles interact. A JAXR client uses JAXR interfaces and classes to request access to a registry. The client sends the request to a JAXR provider. When a JAXR provider receives a request from a JAXR client, it transforms the request into an equivalent request that is based on the specifications of the target registry. The JAXR provider then passes this transformed request to a registry provider. The registry provider receives a request from a JAXR provider and processes it.

The process is then reversed. The registry provider returns a response to the JAXR provider, which transforms it to an equivalent JAXR response. The JAXR provider sends the JAXR response to the JAXR client.

Here is part of an example that shows a client using the JAXR API to discover book store services (these are classified in the NAICS classification scheme as code number 451211) in a UDDI registry:

```
import javax.xml.registry.*;
import javax.xml.registry.infomodel.*;
import java.net.*;
import java.util.*;

public class bookstoreQuery {
    Connection connection = null;
    RegistryService rs = null;
    BusinessQueryManager queryManager = null;
    BusinessLifeCycleManager lifeCycleManager = null;
    String queryURL = "some UDDI registry URI";

// Get a connection to the registry.
     bookstoreQuery.query("ntis-gov:naics", "Book Stores", "451211" );

// Define connection configuration properties
     Properties props = new Properties();
     props.setProperty("javax.xml.registry.queryManagerURL", queryURL);
     props.setProperty("javax.xml.registry.factoryClass",
        "com.sun.xml.registry.uddi.ConnectionFactoryImpl");
     factory.setProperties(props);
     connection = factory.createConnection();

// Get registry service and managers
   rs = connection.getRegistryService();
   queryManager = rs.getBusinessQueryManager();
   lifeCycleManager = rs.getBusinessLifeCycleManager();

// Get classification information
   ClassificationScheme classificationScheme =
   queryManager.findClassificationSchemeByName( cName );
   Classification classification =
     lifeCycleManager.createClassification( classificationScheme, keyName, keyValue );
   Collection classifications = new ArrayList();
   classifications.add ( classification );

// Find using the classification information
   BulkResponse response =
   queryManager.findOrganizations(findQualifiers,
     null, classifications, null, null, null);
   Collection orgs = response.getCollection();
```

Web Services Technologies and Tools in Java WSDP 1.5

Java WSDP is a package that provides early access to the latest technology implementations and tools for web services development. The most current version of the package, Java WSDP 1.5, includes implementations of a number of web services technologies that are not in J2EE 1.4, as well as newer releases of the web services technology implementations that are in the J2EE 1.4 SDK. The implementations of web services technologies that are not in J2EE 1.4 are:

- Java Architecture for XML Binding (JAXB) v1.0.4
- XML and Web Services Security v1.0

- XML Digital Signatures v1.0 EA2

The newer releases of web services technology implementations that are in the J2EE 1.4 SDK are:

- Java API for XML Processing (JAXP) v1.2.6_01
- Java API for XML-based RPC (JAX-RPC) v1.1.2_01
- SOAP with Attachments API for Java (SAAJ) v1.2.1_01
- Java API for XML Registries (JAXR) v1.0.7

In addition to these web services technology implementations, Java WSDP 1.5 also provides an Early Access implementation of the Sun Java Streaming XML Parser (SJSXP), which implements the Streaming API for XML (StAX).

Java Architecture for XML Binding (JAXB) v1.0.4

Java Architecture for XML Binding (JAXB) gives you a way of mapping an XML document into a set of Java classes and interfaces that are based on the document's XML schema. The value of doing that is that your application can work directly with Java content rather than working with XML content (as it would have to using an API such as JAXP).

The mapping is done in two steps. First you use a binding compiler provided with the JAXB implementation to bind the document's XML schema into a set of Java classes and interfaces. These classes and interfaces form the core of a binding framework. After compiling the classes and interfaces, you use methods in the binding framework to marshal the XML document into a tree of content objects. You can then access the data in the tree through other methods in the binding framework.

For example, suppose that you had an XML document, `books.xml` that was associated with a schema, `books.xsd`. Click here to see the schema. This schema defines a `<Collection>` as an element that has a complex type. This means that it has child elements, in this case, `<book>` elements. Each `<book>` element also has a complex type named `bookType`. The `<book>` element has child elements such as `<name>`, `<ISBN>`, and `<author>`. Some of these have their own child elements.

Running the `books.xsd` schema through the binding compiler generates a set of interfaces and a set of classes that implement the interfaces. Here is the interface the binding compiler generates for the `<Collection>` element complex type. Here is the generated class that implements the complex type.

To unmarshal an XML document, you:

- Create a `JAXBContext` object that provides the entry point to the JAXB API.
- Create an `Unmarshaller` object. This object contains methods that perform the actual unmarshalling.
- Call the `unmarshal` method to do the unmarshalling.

After unmarshalling, your application can use `get` methods in the schema-derived classes to access the XML data. Here, for example, is a program that unmarshals the data in the `books.xml` file and then displays the data. Notice that the program includes the following statement:

```
unmarshaller.setValidating(true);
```

This validates the source data against the associated schema.

Some other things you can do using JAXB are:

- Unmarshal XML data from other input sources such as an `InputStream` object, a URL, or a DOM node.
- Marshall an XML document from a content tree to an XML file, or to other output formats such as an `OutputStream` object or a DOM node.
- Validate the content tree against a schema.

JAXB 1.0.4 supports a subset of the W3C XML Schema (some schema constructs, such as the identity construct, are not supported). In addition, JAXB also unofficially supports the OASIS RelaxNG schema language.

XML and Web Services Security v1.0

XML and Web Services Security (XWS Security) provides message-level security for applications that use JAX-RPC to access web services. Think of message-level security as a way of supplementing the transport-layer security provided by secure Internet transport protocols such as HTTPS. In message-level security, security information is contained in the SOAP message header. One of the primary uses of message-level security is to secure a SOAP message from unauthorized access at intermediate nodes along the message path. Recall that a SOAP message can pass through a set of intermediate nodes as it travels from a client to a service, and that each node can independently process part or all of the message before forwarding it. Using message-level security, you can do things like encrypt a SOAP message and permit decryption only by the target web service (that is, not by any of the intermediate nodes). For example, this could be used to protect credit card information from exposure until it's received by the target service -- perhaps a credit verification service from a credit card company. In addition, XWS security gives you the flexibility of securing different parts of a SOAP message and in different ways. Specifically, you can secure an entire service, one or more service ports, or one or more service operations. For instance, you can encrypt some parts of the message and not other parts, you can sign with a digital signature some parts of the message and not sign others. In addition to including encryption information, the SOAP message header might include other security-related items such as an X.509 certificate or a security token. The SOAP header can also point to a repository of security information for the message.

XWS Security 1.0 implements the following WS-Security standards: SOAP Message Security V1.0, Username Token Profile V1.0, and X.509 Token Profile V1.0 (each of these describes a particular aspect of SOAP message security). XWS Security also implements the following W3C security standards: XML Signature and XML Encryption.

To use XWS Security, you need to create security configuration files for the client and service. A security configuration file is an XML file that describes the security operations to be performed on the SOAP message. For example, here is part of a simple security configuration file for a client that applies an XML Digital Signature to a SOAP message:

```
<xwss:JAXRPCSecurity
  xmlns:xwss="http://java.sun.com/xml/ns/xwss/config">
    <xwss:Service>
        <xwss:SecurityConfiguration dumpMessages="true">
          <xwss:Sign>
            <xwss:X509Token
              certificateAlias="xws-security-client"/>
            </xwss:Sign>
            <xwss:RequireSignature/>
        </xwss:SecurityConfiguration>
    </xwss:Service>
    <xwss:SecurityEnvironmentHandler>
        com.sun.xml.wss.sample.SecurityEnvironmentHandler
    </xwss:SecurityEnvironmentHandler>
</xwss:JAXRPCSecurity>
```

- The `<xwss:JAXRPCSecurity>` element identifies this as a security configuration file.
- The `<xwss:SecurityConfiguration>` element specifies the security operations to be performed.
- The `<xwss:Sign>` element specifies that a digital signature will be applied.
  The `<xwss:X509Token>`element specifies an X.509 certificate token that indicates the key used for the digital signature. The digital signature refers to this token.
- The `<xwss:RequireSignature>` element specifies that the client expects the response it receives from the service to be signed.
- The `<xwss:SecurityEnvironmentHandler>` element specifies a `CallbackHandler`, a class that gets the security information (such as private keys and certificates) needed for the signing operation.

After the needed security configuration files are created, you invoke the `wscompile` tool -- the same tool that you use to create a WSDL file and artifacts for a JAX-RPC-based web service. When you run the `wscompile` tool, you specify the `-security` option and identify a security configuration file. The tool then generates the artifacts needed for the security operations specified in the configuration file.

XML Digital Signatures v1.0 EA2

XML Digital Signatures v1.0 EA2 is an early access implementation of the Java Digital Signature API. You can use the API to sign the content of an XML document (actually, you can use the API to sign any binary data) in conformance with the W3 standard, XML-Signature Syntax and Processing, and also to validate the signature.

Here is an example that uses the Java Digital Signature API for signing an XML document. The example is adapted from a sample program provided in Java WSDP 1.5 for the Java Digital Signature API. The program uses an `XMLSignatureFactory` to create the digital signature:

```
XMLSignatureFactory fac =
       XMLSignatureFactory.getInstance("DOM",
       (Provider) Class.forName(providerName).newInstance());
```

It then creates objects that map to corresponding elements in the XML signature. For example the `SignedInfo` object corresponds to a `<SignedInfo>` element in the XML signature. The `<SignedInfo>` element contains signature information and a reference to the data to be signed.

```
SignedInfo si = fac.newSignedInfo
          (fac.newCanonicalizationMethod
            (CanonicalizationMethod.INCLUSIVE_WITH_COMMENTS,
                null),
           fac.newSignatureMethod(SignatureMethod.DSA_SHA1,
               null),
           Collections.singletonList(ref));
```

Next, the program creates a 512-bit DSA key and a `KeyInfo` object that contains the public key to be used in decrypting the signature. Here are the statements that create the DSA key:

```
KeyPairGenerator kpg =
          KeyPairGenerator.getInstance("DSA");
      kpg.initialize(512);
      KeyPair kp = kpg.generateKeyPair();
```

The program then creates an instance of the document, generates a digital signature for the document, and puts the digital signature in a file. Here are the statements that generate the digital signature:

```
DOMSignContext dsc = new DOMSignContext
        (kp.getPrivate(), doc.getDocumentElement());

       XMLSignature signature = fac.newXMLSignature(si, ki);
         signature.sign(dsc);
```

Java API for XML Processing (JAXP) v1.2.6_01

JAXP 1.2.6_01 is an updated reference implementation of JAXP 1.2. A number of enhancements have been made to the JAXP reference implementation since the release of the JAXP 1.2 implementation in the J2EE 1.4 SDK. These include new releases of the Xerces parser, and new security-related properties -- for example, you can specify that the parser not allow a specific DTD. The JAXP v1.2.6_01 implementation in Java WSDP 1.5 includes a new version of the Xerces parser (v2.6.2) and a new version of the Xalan transformer (v2.6.0).

Java API for XML-based RPC (JAX-RPC) v1.1.2_01

JAX-RPC 1.1.2_01 is an updated reference implementation of JAX-RPC 1.1. The major enhancement in JAX-RPC 1.1.2_01 is support for WS-I Basic Profile 1.1. Recall that the WS-I Basic Profile identifies a core set of web services technologies, that when implemented in different platforms, helps ensure interoperability of web services across those platforms. WS-Basic Profile 1.1 adds a number of technologies to the core set, including SOAP Messages With Attachments and the part of the WSDL 1.1 specification that covers MIME bindings. MIME stands for Multipurpose Internet Mail Extensions. It's a standard that extends the format of Internet mail to allow for things like multipart message bodies. The SOAP Messages With Attachments specification takes a MIME approach in describing how to build a SOAP message.

Through this added support, a client can use JAX-RPC to access services through XML (SOAP-based) RPC calls (just as a client could previously), but include attachments in the call (JAX-RPC maps the attachments as

method parameters). The primary difference is that the `<binding>` element of the WSDL description needs to specify one or more MIME parts that correspond to the MIME parts of the SOAP message with attachments. For example, here is an example of a `<binding>` element for an operation that adds a photo (provided as an attachment) to a photo catalog:

```
<wsdl:binding name="PhotoCatalogBinding" type="tns:PhotoCatalog">
 <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
       <wsdl:operation name="addPhoto">
         <wsdl:input>
           <mime>MultipartRelated>
             <mime:part>
               <soap:body use="literal"/>
             </mime:part>
             <mime:part>
               <mime:content part="photo" type="image/jpeg"/>
             </mime:part>
          </mime:multipartRelated>
        </wsdl:operation>
  </wsdl:binding>
```

SOAP with Attachments API for Java (SAAJ) v1.2.1_01

SAAJ 1.2.1_01 implements SAAJ 1.2. It does not provide any additional features beyond those in SAAJ 1.2 implementation in the J2EE 1.4 SDK, but does include a number of bug fixes.

Java API for XML Registries (JAXR) v1.0.7

JAXR 1.0.7 is an updated reference implementation of JAXR 1.0. The implementation allows you to set a number of properties on a JAXR connection. For example, you can specify a property that gives a hint to the JAXR provider about the authentication method to use. Or you can specify a property that sets the maximum number of rows to be returned by a UDDI provider. Most of these properties are specified by the client. For example, here is how a client would specify the authentication method hint to the JAXR provider:

```
Properties props = new Properties();
props.setProperty("javax.xml.registry.security.authenticationMethod",
    "UDDI_GET_AUTHTOKEN");
ConnectionFactory factory = ConnectionFactory.newInstance();
factory.setProperties(props);
connection = factory.createConnection();
```

Sun Java Streaming XML Parser (SJSXP) v1.0

Sun Java Streaming XML Parser Version (SJSXP) 1.0, is a high performance implementation of the Streaming API for XML (StAX). Java WSDP 1.5 includes an Early Access release of SJSXP. StAX is a "pull" API for XML. By comparison, SAX and DOM are "push" APIs. In other words, the SAX and DOM APIs read-in the XML data they encounter and provide it to the application. In the case of SAX, the API provides the data, piece-by-piece, to callback methods. In the case of DOM, the API reads in the entire document and makes it available as an in-memory tree for subsequent processing by the application. A "pull" API such as StAX, simply points to a specific item of data in an XML document. The application then processes the item, as

appropriate. In implementing StAX, SJSXP provides an object, called a cursor, that moves sequentially through an XML document. Using methods provided by the cursor, an application can move the cursor, item-by-item, through the document, and determine what type of XML construct the item represents. The application can then process the item appropriately.

For example, here is a snippet of code adapted from one of the sample programs provided with SJSXP 1.0. The sample program uses SJSXP to parse an XML file:

```
xmlif = XMLInputFactory.newInstance();
  xmlif.setProperty
    (XMLInputFactory.IS_REPLACING_ENTITY_REFERENCES,Boolean.TRUE);

FileInputStream fis = new FileInputStream(filename);
  XMLStreamReader xmlr =
    xmlif.createXMLStreamReader(filename, fis);

  while(xmlr.hasNext()){
    eventType = xmlr.next();
    switch (eventType){
          case XMLEvent.START_ELEMENT:
        System.out.println("START_ELEMENT");
      ...
  }
}
```

First an `XMLInputFactory` class is created for the input stream. The `setProperty` method of the `XMLInputFactory` class specifies properties for the parser. In the example, the property tells the parser to replace entity references it encounters. Various other parser properties can be set. The cursor's `hasNext()` method determines if there's an item in the file (that is, beyond where the cursor is currently pointing). If there is another item, the `next()` method moves the cursor to it. The `next()` method returns an integer constant that indicates the type of XML construct that the cursor is pointing to. For example, the START_ELEMENT constant indicates the start of an XML element. The action taken in the example is to print the returned constant.

You have the option of filtering an XML document so that the parser doesn't have to parse the entire document. You can also use SJSXP to write an XML document.

Java Developer Tools and Servers for Web Services

In addition to toolkits such as the J2EE 1.4 SDK and Java WSDP 1.5, a number of Java technology-based development tools and servers incorporate implementations of Java technologies for web services. These tools can simplify the process of developing and deploying web services and applications that use web services. This section highlights a number of these tools and servers, and the web services technology features they offer.

NetBeans IDE 4.1

NetBeans IDE 4.1, which is currently available as a Beta release, is a comprehensive tool for developing enterprise Java applications and web services. Using NetBeans IDE 4.1, you can develop J2EE 1.4 applications, application components (such as Enterprise JavaBeans components), and web services, and

deploy them to Sun Java System Application Server 8.1 Platform Edition. From a web services perspective, you can use the tool to generate client code so that an application can use existing web services. You can also use the tool to create and test web services. In creating a web service with the NetBeans 4.1 IDE, you can implement the web service endpoint as a web application or a session bean.

NetBeans IDE 4.1 offers a graphical environment for application and web service development. It also provides an extensible framework that integrates tools for development. NetBeans IDE 4.1 simplifies application and web services development by freeing you from doing a lot of the detailed coding needed to use J2EE application technologies and web services technologies. It allows you to focus on the logic of your applications and web services. It also frees you from having to manually invoke tools in the various steps of the development process. Instead you simply make selections in the NetBeans IDE user interface, and respond to prompts in the interface or in wizards. In response, the IDE generates the necessary code and invokes the needed tools. NetBeans IDE 4.1 fully supports the J2EE 1.4 Platform (in fact, it's the first open source IDE to do so), including all the web services technologies in J2EE 1.4. This means that when you use the NetBeans IDE 4.1 to build web services and web service clients, the IDE generates the code for the underlying web services infrastructure, such as WSDL files, SOAP messages, and JAX-RPC artifacts.

For example, the NetBeans IDE 4.1 Quick Start Guide for Web Services shows the steps in creating, deploying, and registering a web service. The web service (named `HiWS`) is very simple -- it has one method, (`sayHi`) that accepts a name of type `String` as its one parameter. In response, the service returns the string `Hi` followed by the name and an exclamation point, for instance `Hi John!` The Quick Start Guide also shows how to create a client for the web service and package it as an application. If you follow the instructions for developing the `HiWS` web service (implementing it as a session bean), here's what some of the windows in the IDE look like (the Project window and Source Editor window for the session bean class are shown):

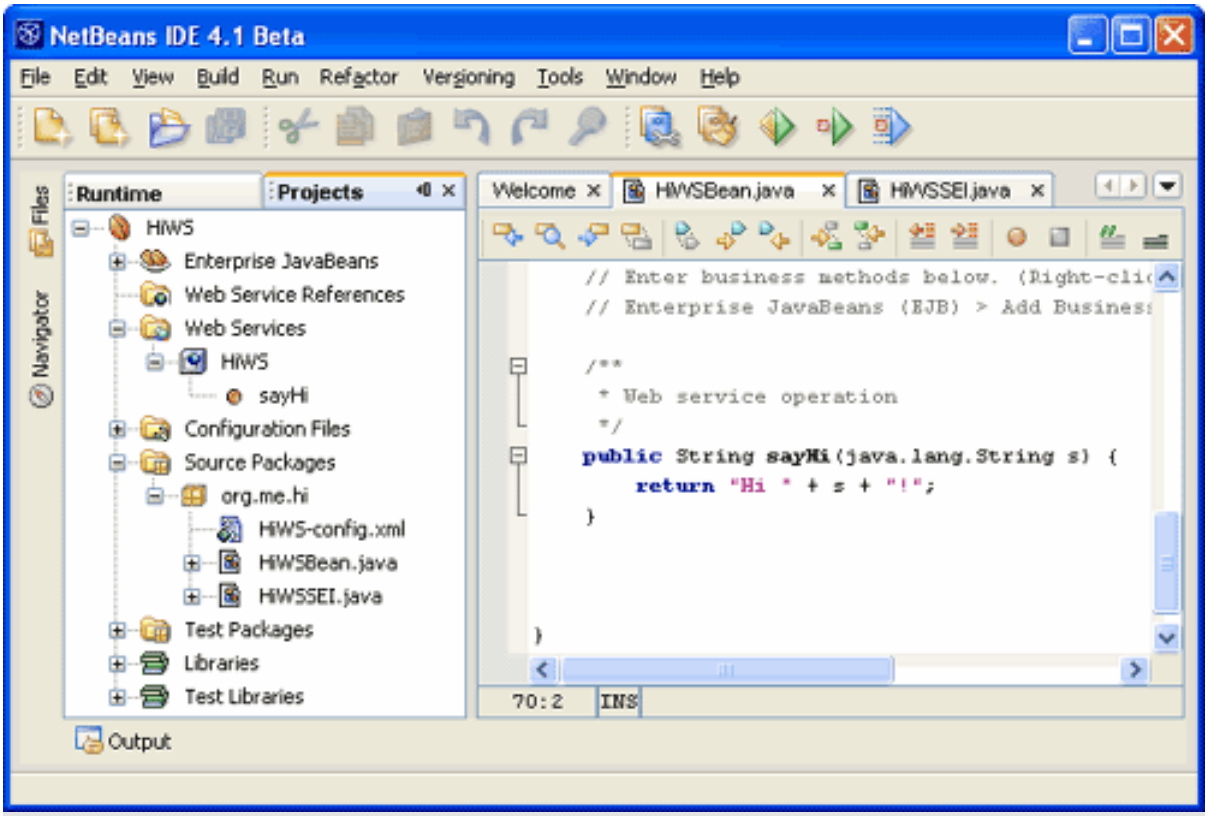


Figure 6: NetBeans 4.1 Project and Editor Windows *(Click image to enlarge)*

Notice that the Source Editor window shown in the illustration displays the contents of the session bean class. NetBeans 4.1 automatically generates most of the code for the bean class as well as the service endpoint

interface for the web service. In this example, all a developer needs to provide is the code for the method (`sayHi`) in the bean class.

The next step in the instructions deploys the web service. In this step, NetBeans runs a script that, among others things, runs the `wscompile` tool, which generates a WSDL file for the web service and JAX-RPC artifacts such as a tie. After the web service is deployed, you can direct NetBeans 4.1 to use the WSDL file to generate client code (including client artifacts, such as a stub) for the web service.

Sun Java System Application Server 8.1

Sun Java System Application Server 8.1 is a J2EE 1.4 platform-compatible server for developing and deploying J2EE applications and web services. Application Server 8.1 is available in three editions: Platform Edition, Standard Edition, and Enterprise Edition. The focus of the Platform Edition is developer productivity. This full-featured, high-performance, small-footprint container is free for development, deployment and redistribution. The Platform Edition is ideal for embedding and bundling and is already included in NetBeans 4.1, Sun Java Studio Creator, Sun Java Studio Enterprise, and Solaris. The current version of the Platform Edition, Sun Java System Application Server 8.1 Platform Edition 2005Q1, is available as a separate download, but is also available as a bundle with the J2EE SDK 2005Q1.

The Standard Edition and Enterprise Editions, build on the Platform Edition to support multi-tier, multi-machine, high-volume, mission-critical deployments. The current version of the Standard Edition, Sun Java System Application Server 8.1 Standard Edition 2005Q1, and the Enterprise Edition, Sun Java System Application Server 8.1 Enterprise Edition 2005Q1, are distributed as part of the Sun Java Enterprise System.

In addition to J2EE 1.4 platform compatibility, all editions of Application Server 8.1 integrate JavaServer Faces technology, the Java Server Pages Standard Tag Library, and the JAXP, JAX-RPC, JAXB, JAXR, SAAJ, and XWS Security components of Java WSDP 1.5.

Sun Java Enterprise System and Sun Java System Suites

The Sun Java Enterprise System integrates into a single package a comprehensive set of standards-based enterprise services. These services include web and application services, network and identity services, communication and collaboration services, portal services, and security services -- in short, the kind of services that provide the infrastructure for enterprise applications, including applications that use web services. Each of these services is provided by one or more components within the Java Enterprise System. Each component is also available as an individual product. The components for each service are listed here.

Solution-specific packages that bring together subsets of the Java Enterprise System are also available as Sun Java Suites. Each suite addresses a specific enterprise need, such as high availability, identity management, and application development and deployment. The suites are:

- Sun Java Availability Suite
- Sun Java Identity Management Suite
- Sun Java Web Infrastructure Suite
- Sun Java Application Platform Suite
- Sun Java Communications Platform Suite

The contents of each suite are listed here.

The Java Enterprise System supports JAXP, JAX-RPC, JAXB, SAAJ, and JAXR. The Java Enterprise System components that implement web services technologies are:

- Sun Java System Application Server. As mentioned earlier, Application Server 8.1 is a J2EE 1.4 platform-compatible server for developing and deploying J2EE applications and web services. Application Server 8.1 implements JAXP, JAX-RPC, JAXB, JAXR, SAAJ, and XWS Security. The Platform Edition is available as part of the Web and Application Services provided by the Java Enterprise System. The Standard Edition is available with the Sun Java Web Infrastructure Suite. The Enterprise Edition is available in the Sun Java Application Platform Suite.
- Sun Java System Access Manager. This component provides security-based services such as access control, authentication, and authorization for web applications. It implements JAXP, JAX-RPC, JAXB, SAAJ, and JAXR. Access Manager is available as part of the Network Identity Services provided by the Java Enterprise System. It is also available in the Sun Java Web Infrastructure Suite and the Sun Java Identity Management Suite.
- Sun Java System Portal Server. This component provides portal-based services such as personalization of portal content and centralized identity management. It implements JAXP, JAX-RPC, JAXB, SAAJ, and JAXR. Portal Server is available as part of the Web and Application Services provided by the Java Enterprise System. It is also available in the Sun Java Application Platform Suite.
- Sun Java System Message Queue. This component provides enterprise message services through the Java Message Service API. It implements SAAJ. Message Queue is available as part of the Web and Application Services provided by the Java Enterprise System. It is also available in the Sun Java Application Platform Suite.

The following table summarizes the web services technology support in the Java Enterprise System and its component servers:

Server	Web Services Technologies
Sun Java Enterprise System	JAXP, JAX-RPC, JAXB, JAXR, and SAAJ
Sun Java System Access Manager	JAXP, JAX-RPC, JAXB, JAXR, and SAAJ
Sun Java System Portal Server	JAXP, JAX-RPC, JAXB, JAXR, and SAAJ
Sun Java System Message Queue	SAAJ

In addition to the enterprise infrastructure services it offers, the Java Enterprise System also includes the Java Studio Enterprise and Java Studio Creator development tools.

Sun Java Studio Enterprise

Java Studio Enterprise brings together an integrated set of tools and enterprise services for developing, testing, and deploying enterprise Java applications and web services. The current release, Java Studio Enterprise 7, provides a graphical environment (including optimized wizards and property sheets) for building applications and web services that conform to J2EE 1.3 specifications. Upcoming releases of Java Studio Enterprise will phase in support for J2EE 1.4 web services technologies as well as some SOA-specific features (see The Future: Sun's SOA Initiative).

The Future: Sun's SOA Initiative

So far, the discussion of tools and servers has focused on building and deploying web services and the applications that use web services. But what about tools that assist in managing a service-oriented architecture, for example, tools that monitor service registration, control service versioning, or that secure services? In fact, tools that provide these features are on their way as an outgrowth of Sun's SOA Initiative. As part of this initiative, SOA-specific features will be phased into new releases of the Java Enterprise System, Java Studio Enterprise, and other Sun products. One of the features is a registry service that will be provided by the Java Enterprise System. This registry service is based on the freebXML Registry open source project. The registry service will implement the ebXML Registry Information Model (RIM) 3.0 and the ebXML Registry Services (RS) 3.0 specifications. The registry service allows businesses to register both services and metadata associated with those services (such as policy statements) in a registry/repository. Java Studio Enterprise will provide a JAXR provider, so that users will be able to access the registry/repository through the JAXR API.

Service and metadata registration is an important element in managing services in an SOA, but it's not the only important element. There are other elements that need to be addressed before an SOA platform can be considered fully functioning. These elements include:

- Service Orchestration. The objective here is to establish an order or pattern in which services interact with one another. Usually the order or pattern reflects the "real-life" business processes in the enterprise. Expect to see features in new releases of the Java Enterprise System that address service orchestration by supporting the BPEL specification. Also, expect to see Service-Oriented Development of Application (SODA) tools added to Java Studio Enterprise, one of them focused on visual assembly of applications using BPEL.

- Business Integration. An important objective of many businesses is to integrate corporate resources in a flexible and efficient way. An expert group led by Sun and with representation by key vendors has been working on a specification for business integration, JSR 208: Java Business Integration (JBI). This specification promises to set the standard for integrating corporate resources in an SOA. JBI defines a framework for plug-in components that comprise an integrated solution. The plug-in components can be as varied as EJB containers, BPEL process engines, or adapters that expose application or workflows as web services. Expect to see support in new releases of the Java Enterprise System for the BPI framework. Also expect to see vendors and system integrators offering plug-in components that fit in the framework.

- Service Management and Monitoring. Clearly being able to monitor and manage services is important in an SOA. As new releases of the Java Enterprise System roll out, expect to see new features for web service level monitoring, tracking, and control based on an enterprise's web service policy and service-level agreements. The monitoring features will be designed to not only track services, but also underlying infrastructure, components, and processes. Web Services Management (WS-Management) is an important, developing specification in this area. WS-Management identifies a core set of web service specifications and protocols for managing services (as well as PCs, servers, devices, applications, and other manageable entities).

- Service Provisioning. Service provisioning means deploying a service and making available the resources it requires. A key requirement of an SOA is to simplify and automate (as much as possible) the provisioning of services. Look for new features in upcoming releases of the Java Enterprise System that will enable service-level provisioning much like network services are provisioned through the NI Grid Provisioning System.

- Federated Identity and Policy. Enabling secure interactions between services and applications is a big

challenge, but an extremely important goal of an SOA. Meeting the challenge is made difficult by the wide variety of ways businesses ensure security. One approach to meeting this goals is federated identity and policy. This approach enables secure, role-based access to services, both inside an enterprise or between enterprises. In a federated identity system, each business manages its own identity, and uses standard web-based technologies to establish trust between businesses. These technologies also enable appropriate degrees of access between services. Expect to see support in new releases of the Java Enterprise System for federated identity for web services based on the Liberty Alliance specifications.

Another important facet of Sun's SOA Initiative is a services offering, called the SOA Opportunity Assessment, that helps enterprises assess their readiness for the move to a web services-based SOA. The SOA Opportunity Assessment is described in the whitepaper Assessing Your SOA Readiness.

Summary

SOA and web services are not just abstract concepts, they're real approaches to solving today's IT problems. A critical mass of widely adopted web services technologies are available today, such as XML, UDDI, SOAP and WSDL. In addition, a core set of Java technologies for web services, such as JAX-RPC and XWS-Security are now available in toolkits such as the J2EE 1.4 SDK and Java Developer Pack 1.5. Tools, such as NetBeans 4.1, are also emerging to simplify the web services development process, and soon SOA management features will be available as part of the Java Enterprise System and Java Studio Enterprise. If you haven't migrated to a web services-based SOA, the time might be now. You can get started toward web services and SOA by assessing your readiness.

For More Information

- Designing Web Services With the J2EE 1.4 Platform
- Sun Launches Comprehensive Services-Oriented Architecture Initiative
- Java BluePrints Solutions Catalog
- Java Adventure Builder Reference Application 1.0.1
- WS-I Basic Profile 1.0
- WS-I Basic Profile 1.1
- Web Services Interoperability Organization
- Organization for the Advancement of Structured Information Standards (OASIS)
- United Nations Centre for Trade Facilitation and Electronic Business (UN/CEFACT)
- Java Technology and Web Services
- Java 2 Platform, Enterprise Edition (J2EE) 1.4 platform
- J2EE 1.4 SDK
- Java Web Services Developer Pack (Java WSDP) 1.5
- Webinar: The Java Web Services Developer Pack 1.5
- NetBeans IDE 4.1 Beta
- Sun Java Studio Creator
- Sun Java Studio Enterprise
- Sun Java Application Server 8.1
- Sun Java Enterprise System & Sun Java System Suites
- Assessing Your SOA Readiness
- JavaOne Online Conference Sessions

- On the Road to SOA: Building Real-World Enterprise Web Services
- Using SOA and Java 2 Platform, Enterprise Edition (J2EE) to Ease Integration and Customization While Reducing Overall Implementation and Customization Costs

About the Author

Ed Ort is a staff member of java.sun.com and developers.sun.com. He has written extensively about relational database technology, programming languages, and web services.