Chapter 2

The Basic Web Services Stack

Many of the challenges presented in the previous chapter revolve around interoperability challenges on multiple operating systems and/or middleware packages. These include high integration costs, lack of industry standards, and high deployment costs. Web Services have the potential of addressing many of these issues, and this chapter discusses the cornerstone technologies that are essential for Web Services. These include Extensible Markup Language (XML), Service-Oriented Access Protocol (SOAP), Web Services Definition Language (WSDL), and Universal Description, Discovery and Integration (UDDI). Chapter 3 then follows with emerging standards around security, scalability, and so on. This chapter concludes with a discussion on how Web Services can augment the technologies discussed in Chapter 1.

Before delving into the underlying technologies, let's take a broad view of the various roles of a Web Services architecture, which is also sometimes referred to as a service-oriented architecture.

Service-Oriented Architecture (SOA)

An **SOA** is called service oriented because the central idea is that a client (which can be a person or a computer) needs a particular set of services to be fulfilled. Of course, before the client can request the service, it needs to find the provider (which previously published the service); this location service is provided by a **service broker,** who typically operates a repository. Upon request, the service broker returns a document that allows the client to first locate and then **bind** to the provider. Thus, the three key roles in an SOA are

- Client
- Service broker
- Service provider

Figure 2–1 illustrates the roles and the sequence of events in an SOA.

The role of the broker may not be immediately obvious, especially for a small set of services. However, keep in mind that a client may ask for multiple services, each of which may have a different provider. Registering the services in a central registry that can be searched by clients provides them with the flexibility needed to perform queries based on a dynamically changing set of criteria—they do not have to statically bind themselves to the provider. Without a registry, the client would have to hard-code the location of the service provider, which can obviously lead to maintenance difficulties.

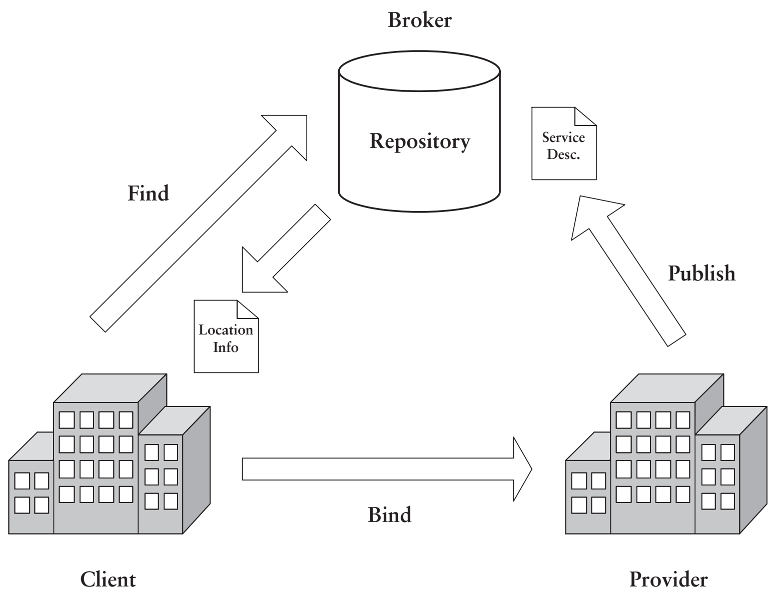


Figure 2–1 The roles in an SOA.

SOAs have been present for a number of years, but, again, they have been used with proprietary protocols and technologies. For example, both CORBA and DCOM provide a naming service and a location service. With the advent of Web Services, the idea has been more widely adopted because of the use of standards-based technologies. The most fundamental standard of all is a common language for describing data—this is the role of XML.

Extensible Markup Language (XML)

To say that XML has been popular with the industry press is an understatement. At the moment, every major vendor has announced support for XML in one form or another, and innovative uses for XML are emerging almost daily.

But what is exactly is XML? It is not a programming language like Java, C++, or C#; that is, it cannot be used to write applications per se. Rather, it is a **meta-language** that can be used to create self-describing, modular documents (data), programs, and even other languages, commonly referred to as **XML grammars**.[1] These documents are often used for data exchange between otherwise incompatible systems.

XML is incredibly diverse and includes a host of other technologies including **XPointer**, **XLink**, and Resource Description Framework (**RDF**). Our intent here is not to give an in-depth discussion of XML, but to provide enough background information to help you understand the implications of how XML is being used in the context of other technologies such as SOAP, WSDL, and UDDI—the foundation of Web Services. Strictly speaking, Web Services can be implemented by using only XML, but for the purposes of our discussion, we are defining Web Services to be built on XML, SOAP, WSDL, and UDDI over a transport protocol such as HTTP. This definition will become clearer as our discussions progress.

The **Worldwide Web Consortium** (**W3C**), an international standards body, began working on XML in mid-1996 and released XML 1.0 in 1998. XML was heavily inspired by the **Standard Generalized Markup Language** (**SGML**), but, in many ways, XML is more readable and

1. An example of an XML grammar is wireless markup language (WML), a popular language for creating (appropriately enough) wireless applications. WSDL and UDDI are also XML grammars.

simpler. The real value of XML is not in its innovativeness as much as its industry acceptance as a common way of describing and exchanging data (and, as we will see later with WSDL and SOAP, XML can also be used to describe applications and invoke them as well).

XML Syntax

As a markup language, XML uses tags to describe information (the tags are highlighted in bold in the following example).

```
<?xml version="1.0" encoding="UTF-8"?>
<Order>
    <Customer>
          <name>John Doe</name>
          <street>1111 AnyStreet</street>
          <city>AnyTown</city>
          <state>GA</state>
          <zip>10000</zip>
    </Customer>
</Order>
```

A tag, enclosed in brackets (<>), is a label or a description (e.g., **street** in our example) of the data that follows, which is called an **element** (the element for **street** in our example is **1111 AnyStreet**). The element is delimited by a similar tag preceded by a slash (**/**), to indicate the end of the element. In our example, the element **1111 AnyStreet** is terminated by the closing tag <**/street**>.

The first line in our example is a convention used to signal the XML parser (the program that has to parse the XML document) that the incoming document is an XML document. Also, the **Customer** element has several child elements:

John Doe
1111 AnyStreet
AnyTown
GA
10000

You may have already noticed one advantage of XML—since it is a text-based language, XML is fairly verbose and therefore human readable. However, this advantage can also be a disadvantage: because they are verbose, XML documents can quickly become very large for complex data sets. There are other points worth noting about XML:

- **XML is extensible.** Unlike HTML, which has a fixed number of tags, XML allows the developer to define any number of tags—whatever is necessary to solve the problem. In our example, the document represents an abstraction of a customer and includes fields to describe the customer.
- **XML is hierarchical.** Elements can have subordinate elements under them. In the example, the **Customer** element contains several child elements.
- **XML is modular.** By allowing documents to reference other documents, XML provides for modular designs and promotes reuse.
- **XML does not include built-in typing.** This data enforcement and validation is provided through document type definitions (DTDs) and XML schemas, two concepts that will be discussed in further detail later.
- **XML does not make any assumptions about the presentation mechanism.** This is unlike HTML, which does make these assumptions. In fact, XML has to be coupled with another technology (such as **XSLT** or **Cascading Style Sheets**) to be displayed. This separation stems from one of XML's primary goals of being a way of exchanging data; oftentimes data is exchanged between systems and hence may not need to be displayed at all.
- **XML is programming language independent.** Since XML is not a programming language per se, it can be used as a common mechanism for data exchange between programming languages and, as we will see later, a common way of connecting applications as well (via SOAP).
- **XML provides validation mechanisms.** Through the use of DTDs and XML schema, XML documents can be validated to determine whether the elements are correct (i.e., whether the values are within a specified range).

Some of the main XML concepts that are especially relevant to Web Services include parsers, namespaces, DTDs, and XML schemas.

XML Parsers

Processing an XML document requires the use of an XML parser, a program that can decompose the XML document into its individual elements. There are two major categories of XML parsers: **Document Object Model** (**DOM**) and **Simple API for XML** (**SAX**).

DOM is a language-neutral API for accessing and modifying tree-based representations of documents such as HTML or XML documents. Developers can use language-specific DOM parsers to programmatically build and process XML documents.

DOM parsers have two major shortcomings:

- The entire XML document is represented in memory; this can lead to performance issues if the XML document is exceedingly large.
- Since the API is language independent, it is quite generic; therefore more steps are often required to process an XML document than would be the case if it were optimized for a particular implementation language. This has led to language-specific variants such as the **JDOM** parser, which is tuned for the Java language.

The SAX parser is an event-based parser and can be used only for reading an XML document. A SAX parser works from event registration. The developer registers event handlers, which are then invoked as the XML document is processed. Each event handler is a small block of code that performs a specific task. The main advantage of a SAX parser over a DOM parser is that the former does not require the entire document to be in memory—the XML document is processed as a stream of data, and the event handlers are invoked. While SAX is easier to work with than DOM, there are some disadvantages:

- Once the XML document has been read, there is no internal representation of the document in memory. Thus, any additional processing requires the document to be parsed again.
- A SAX parser cannot modify the XML document.

Thus it is important to understand the needs of the application before selecting an XML parser.

Well-Formed and Valid XMLs

XML documents must conform to a certain set of guidelines before they can be processed. This leads to two terms that are used to describe the state of a document: well formed and valid.

A **well-formed** XML document is one that follows the syntax of XML and that can be *completely* processed by an XML parser. If there are syntax errors in the document, then the parser rejects the entire document. As far as an XML parser is concerned, there is no such thing as a partially well-formed XML document.

A **valid** XML document is a well-formed document that can also be verified against a DTD, which defines constraints for the individual elements—the order of the elements, the range of the values, and so on. A **validating** XML parser is one that can validate an XML document against a DTD or XML schema, which are described next.

DTDs and Schemas

XML offers two mechanisms for verifying whether or not a document is valid. A DTD is an external document that acts as a template against which an XML document is compared. The XML document references this DTD in its declaration, and the XML parser (assuming it is a validating parser) then validates the elements of the XML document with the DTD. A DTD can specify the order of the elements, the frequency at which elements can occur (for example, an order can contain 0–*n* line items), etc.

While a powerful concept, DTDs have many shortcomings.

- The concept of a DTD predates that of XML (it originated from SGML) and does not conform to XML syntax. This increases the learning curve and can lead to some confusion.
- A DTD does not support data types; this means it is impossible to specify that a given element must be bound to a type. Using the order example under "XML Syntax" earlier in this chapter, there is no way to specify that the line item count needs to be a positive integer.
- An XML document can reference only one DTD; this limits how much validation can occur.

- A DTD cannot enforce data formats; i.e., there is no way to specify that a date must be of the mm/dd/yyyy format.
- DTDs were invented before the standardization of namespaces and consequently do not support namespaces, which can lead to many element name collisions. For more on namespaces, see the next section.

Because of these limitations, applications that have to process XML documents include a lot of error checking functionality. Additionally, SOAP, one of the cornerstone technologies of Web Services, prohibits the use of DTDs in the document declarations.

To address the shortcoming of DTDs, the W3C produced the **XML schema specifications**. XML schemas provide the following advantages:

- The XML schema grammar supports namespaces.
- XML schemas include a predefine set of types including string, base64 binary, integer, positive integer, negative integer, date, and time, along with acceptable ranges and data formats.
- XML schemas also allow for the creation of new types (simple and complex) by following a well-established set of rules.

XML Namespaces

An enterprise system consists of dozens if not hundreds of XML documents. As these XML documents are merged from other sources, inevitably there will be duplicate element names. This can cause problems because each element must have a unique name. XML resolves this name collision issue through the use of **namespaces** (Java provides a similar feature through **packages**). Each element is prefixed with a namespace and therefore has to be unique only for that given namespace rather than globally. In practice, the prefix is usually the name of the company, although any Uniform Resource Locator (URL) will do.[2] Thus, an element name is composed of two parts: the namespace and the name of the element. By qualifying the name of each element with a qualifier, the likelihood of a name collision is greatly reduced. Consider the file system, for example. For a given

2. Technically, the identifier is usually a Uniform Resource Identifier (URI). For our purposes, we will ignore the distinction between a URI and a URL.

directory, a filename must be unique. However, there can be multiple identical filenames as long as each exists in a different directory. In a sense, the directory provides the namespace and qualifies the filename to resolve filename conflicts.

Service-Oriented Access Protocol (SOAP)

One of the challenges of performing integration using traditional middleware is the lack of a universal protocol. By being XML based and not tied to any particular language, SOAP has evolved to become the primary de facto standard protocol for performing integration between multiple platforms and languages.

SOAP originally meant **Simple Object Access Protocol,** but the term has been unofficially redefined to mean **Service-Oriented Access Protocol** because SOAP is not simple and certainly not object oriented; the latter point is important because not all languages are object oriented.

This flexibility in the protocol allows a program that is written in one language and running on one operating system to communicate with a program written in another language running on a different operating system (i.e., a program written in perl running on Solaris can communicate with another program written in Java running on Windows 2000). There is at least one SOAP implementation for each of the popular programming languages including perl, Java, C++, C#, and Visual Basic.

Advantages of SOAP

Before discussing the characteristics of SOAP, let's examine why it has become so popular.

- **SOAP is a fairly lightweight protocol.** Some of the earlier distributed computing protocols (CORBA, RMI, DCOM, etc.) contain fairly advanced features such as registering and locating objects. At its core, SOAP defines only how to connect systems and relies on additional technologies to provide registration features (UDDI) and location features (WSDL).
- **SOAP is language and operating system independent.** In this respect, SOAP is unlike many other middleware technologies

(such as RMI, which works only with Java, and DCOM, which works only on Microsoft Windows and NT).

- **SOAP is XML based.** Instead of relying on proprietary binary protocols (as is the case with CORBA and DCOM), SOAP is based on XML, a ubiquitous standard. As previously noted, XML is fairly readable.
- **SOAP can be used with multiple transport protocols.** These include HTTP, Simple Mail Transfer Protocol (SMTP), file transfer protocol (FTP), and Java Message Service (JMS). Most of the examples in this book will focus on HTTP since it is the most commonly used protocol with SOAP-based systems.
- **SOAP can traverse firewalls.** SOAP needs no additional modifications to do this. Contrast this with CORBA- or DCOM-based systems, which require that a port be opened on the firewall. This is a key requirement for building distributed systems that have to interact with external systems beyond the firewall. (This is also a disadvantage, as we will see later.)
- **SOAP is supported by many vendors.** All major vendors including IBM, Microsoft, BEA, and Apache provide support for SOAP in the form of SOAP toolkits (the IBM and Apache SOAP toolkits are two of the most popular).
- **SOAP is extensible.** The header values (specified in the **Header** element) in the XML document can be used to provide additional features such as authentication, versioning, and optimization. These features are discussed further in the next chapter.

Disadvantages of SOAP

On the down side, SOAP does have some disadvantages.

- **There are interoperability issues between the SOAP toolkits.** It seems ironic that there would be interoperability issues with a technology that promotes interoperability, but this is mostly attributable to the infancy of the SOAP specifications. These have been identified and documented, and the various vendors have been quite cooperative in resolving these differences.
- **SOAP lacks many advanced features.** Much has been written about the advantages of SOAP as a lightweight protocol, but

there are a host of missing features such as guaranteed messaging and security policies.

Many of these issues can be addressed through third-party technologies such as Web Services networks, which are discussed in further detail in later chapters.

SOAP Basics

SOAP is built on a messaging concept of passing XML documents from a sender to a receiver (also called the **endpoint**). The XML document becomes known as a SOAP document and is composed of three sections: **Envelope**, **Header**, and **Body**. Figure 2–2 illustrates the structure of a SOAP document.

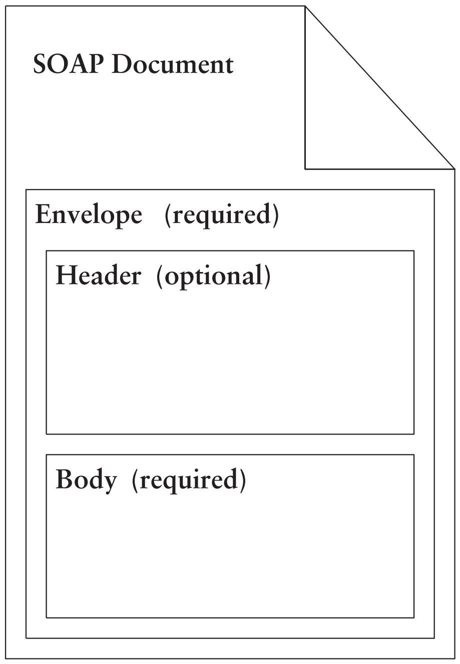


Figure 2–2 The structure of a SOAP document.

The SOAP standards define three major parameters:

- **Envelope body structure.** The envelope contains information such as which methods to invoke, optional parameters, return values, and, where something did not execute successfully, optional exceptions (known as SOAP **faults**).
- **Data encoding rules.** Since SOAP has to support multiple languages and operating systems, it has to define a universally accepted representation for different data types such as float, integer, and arrays. More complex data types (such as Customer) require custom coding, although some toolkits, such as GLUE, inherently provide this mapping.
- **Usage conventions.** SOAP can be used in a multitude of ways, but they are all variations of the same actions: a sender sends an XML document, and the receiver, optionally, returns a response in the form of an XML document (this is the case of a two-way message exchange). As mentioned previously, the XML document may contains faults if errors occurred during processing.

By allowing receivers to be chained together, SOAP-based architectures can be quite sophisticated. Figure 2–3 shows five common architectures that are used with SOAP-based systems—Fire and Forget, Request Response, Notification, Broadcast, and Workflow/Orchestration.

Any link in the processing chain that is not the endpoint is referred to as an **intermediary**. The SOAP specifications allow an intermediary to process a SOAP message partially before passing it to the next link in the processing chain (which can be another intermediary or the endpoint). You will see an example of this in the discussion of the SOAP header later in the chapter.

Migrating from XML to SOAP

Migrating from XML to SOAP is a fairly straightforward procedure. The migration includes these steps:

- Adding optional **Header** elements
- Wrapping the body of the XML document in the SOAP body, which in turn is included in the SOAP envelope
- Declaring the appropriate SOAP namespaces

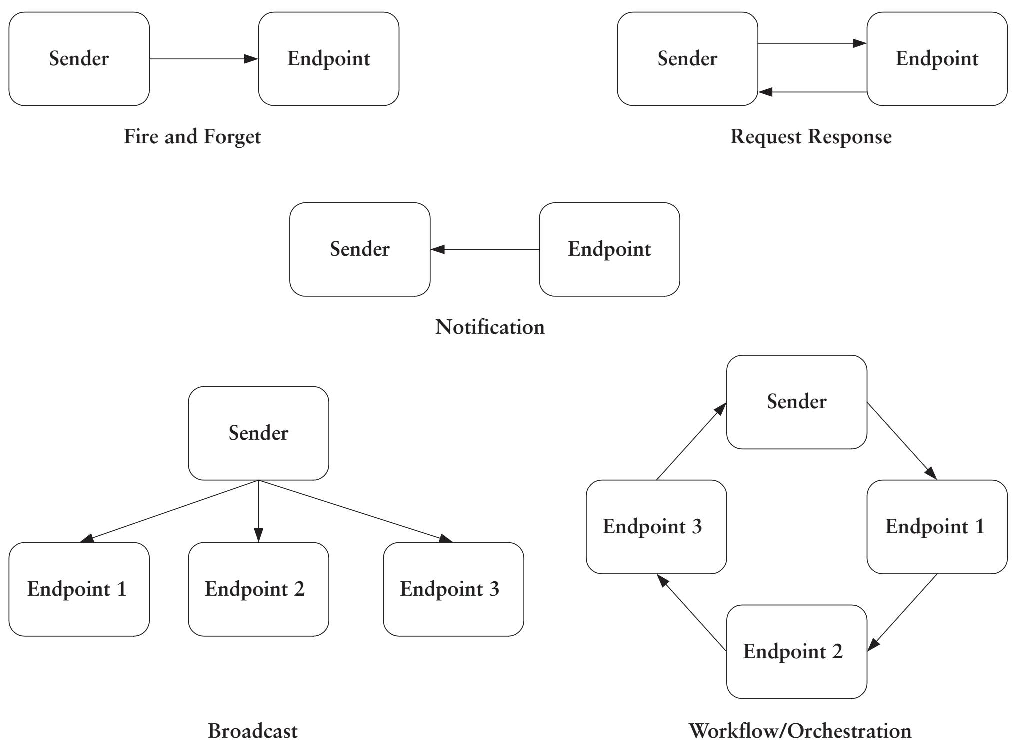


Figure 2–3 Common SOAP architectures.

- Adding optional exception handling
- Specifying the protocol that should be used

For example, converting our earlier XML document to SOAP involves adding the following parts (highlighted in bold; for the sake of simplicity, the HTTP fragment has been stripped away):

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
    xmnls:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope"
    xmnls:xsi="http://www.w3.org/1999/XMLSchema-instance"
    xmnls:xsi="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Header>
    . . . [optional header information]
</SOAP-ENV:Header>
<SOAP-ENV:Body>
<Order>
```

```
<Customer>
     <name>John Doe</name>
     <street>1111 AnyStreet<street>
     <city>AnyTown</city>
     <state>GA<state>
     <zip>10000</zip>
</Customer>
</Order>
<SOAP-ENV:Body>
<SOAP-ENV:Envelope>
```

The next sections describe these additional parts.

SOAP Envelope

The SOAP envelope is the container for the other elements in the SOAP message. A server-side process called a **SOAP handler** can use the availability of the SOAP envelope (along with the *http://schemas.xmlsoap.org/soap/envelope/* namespace declaration) to determine whether the incoming XML document is a SOAP message or not. The handler can be part of the application server, or it can be an external product such as Cape Clear CapeConnect. SOAP handlers are explained in more detail later in the section on adding SOAP support.

SOAP Header

As part of their extensibility design goal, the architects of SOAP provided the **Header** element to allow SOAP messages to be extended generically while still conforming to the SOAP specifications. If a SOAP **Header** element is present (and there can be more than one **Header** element present), it has to be the first child of the **Envelope** element. Each **Header** element can in turn have child elements.

Two examples of using header information to provide extensibility include

- embedding authentication information
- specifying an account number for use with a pay-per-use SOAP service

A SOAP intermediary can use this header information to determine whether the incoming message is properly authorized before forwarding it (to either another intermediary or the endpoint).

Exception Handling

In cases where a SOAP handler cannot decipher a message, a SOAP fault is generated, identified by the **Fault** element. Its child element, **faultcode**, identifies the category of errors that can happen. SOAP 1.1 defines four values for the **faultcode** element:

- **VersionMismatch.** The recipient of the message found an invalid namespace for the SOAP envelope element.
- **MustUnderstand.** The recipient encountered a mandatory Header element it could not understand. Remember, header elements are optional.
- **Client.** The fault was in the message being received. Possible causes: missing elements, malformed elements, and the like.
- **Server.** The fault occurred on the recipient side, i.e., a server error.

Note that an application is free to extend these values using a (.) notation. For example, a value of **Client.Login** can be used to specify that there was a problem with a client login.

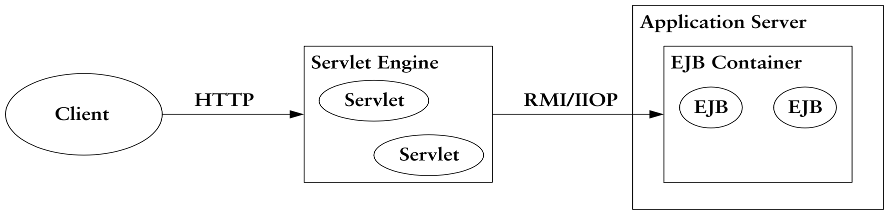
In addition to the **faultcode** element, there are two other elements that can be used to provide further clarification on the fault:

- **faultstring.** This element provides a readable explanation on why the fault occurred.
- **detail.** The value of the **detail** element indicates that the problem occurred while processing the **body** element. If the **detail** element is not present, then the fault occurred outside of the body of the message.

Adding SOAP Support

One of the advantages of adopting SOAP is that the support can be built on top of existing technologies. Figure 2–4 shows a typical J2EE Web-based architecture without support for SOAP.[3] Adding SOAP support to such a system typically requires the addition of a SOAP

3. This deployment shows the servlet engine separated from the application server, a format that provides more scalability. A simpler, albeit less scalable, deployment would be to use the servlet engine that is built into the application server.



EJB = Enterprise Java Beans
IIOP = Internet Inter-Orb Protocol

Figure 2–4 Typical J2EE deployment.

handler (if the application server cannot support SOAP requests), which parses incoming SOAP requests and then calls the appropriate native method in the implementation language. Recall that SOAP is a protocol, not a programming language; hence, the request must be mapped to an entry point in an executing application. The entry point can be a method in a class (for object-oriented systems such as Java, C++, or C#) or a function name (for systems such as perl, which are not object oriented).

Common SOAP handlers include CapeConnect from Cape Clear, Iona's XMLBus (see Appendix D for a more detailed discussion), and Apache Axis. In summary, a system is said to be SOAP compliant if it can take an incoming SOAP request, forward it to the appropriate endpoint, and package the result back in a SOAP response.

Figure 2–5 illustrates the addition of a SOAP handler to the J2EE environment shown in Figure 2–4.[4]

While SOAP provides many useful features, it is still incomplete because it does not address this issue: how does an endpoint unambiguously describe its services? Likewise, another outstanding issue: how does a requester locate the endpoint? These two features are provided by two other key technologies—WSDL and UDDI.

4. Many of the application servers have announced SOAP support, which means the SOAP handler may be part of the application server as well.

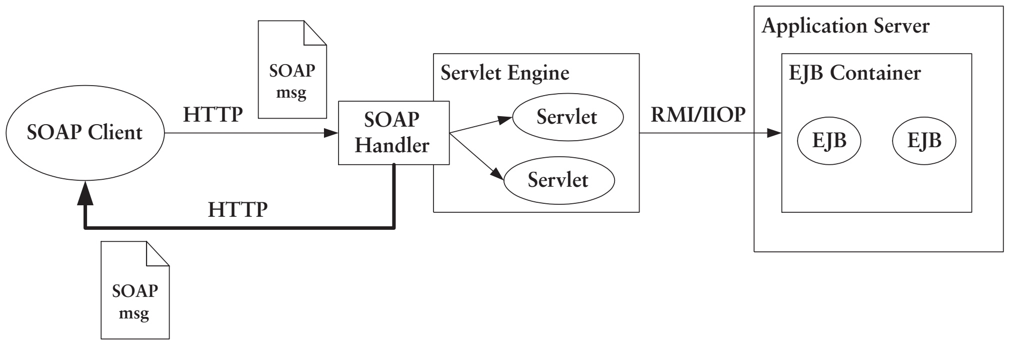


Figure 2–5 Adding SOAP support to J2EE environment.

Web Services Definition Language (WSDL)

To enable a client to *use* a Web Service effectively, there first has to be a mechanism for *describing* that Web Service. At first glance, this may seem difficult, but the challenges are many. We can provide a description in prose format (such as a **README** file), but it would not be practical to describe all the different ways a Web Service can be used in prose. We can also list some examples of how the Web Service can be used effectively, but, again, that may not adequately describe all the combinations for which a Web Service can be invoked.

The problem of succinctly and unambiguously describing a Web Service is similar to the challenge faced by compiler writers—the conventional solution is to use a grammar tree to describe the syntax of a language. While quite effective, a grammar tree is rarely decipherable for those without a strong background in compiler theory.

WSDL was created in response to the need for unambiguously describing the various characteristics of a Web Service. As an XML grammar, WSDL is not easy to learn, but it *is* considerably less intimidating than a programming language (such as C or C++).

A WSDL document is a well-formed XML document that lists the following characteristics for one or more Web Services.

Publicly accessible functions.[5] A WSDL lists all the operations a client can expect a Web Service to support.

- **Input and output parameters along with associated types.** In order to invoke each operation, the client needs to know the expected parameters for each input operation and the expected output of each operation. Again, this is identical to a normal function declaration. In order to support portability between languages and operating systems, all data types are defined in XML schema format.
- **Binding and address information for each Web Service.** To allow loose coupling between a requester and a Web Service, WSDL specifies where the service can be found (usually a URL) and the transport protocol that should be used to invoke the service (remember that a Web Service can be used with multiple protocols).

In essence, WSDL defines a contract that a provider is committed to supporting, and, in the spirit of separating implementation from interface, WSDL does not specify how each Web Service is implemented. As a point of comparison, WSDL can best be likened to CORBA's IDL.

Most of the existing toolkits (GLUE, IBM Web Services Toolkit [WSTK], BEA Web Services Workshop, Cape Clear CapeStudio, etc.) have built-in functionality to automatically parse and generate WSDL files (although, due to the immaturity of the tools, the generated files still require some manual tweaking). Even so, it is still worthwhile to understand the structure of a WSDL document.

WSDL Syntax

The WSDL specifications list six major elements:

- The **definitions** element is the root element containing the five remaining elements; it defines the name of the service and declares the namespaces used throughout the document.

5. The astute reader will notice the use of the term "functions" rather than "methods." SOAP is not object-oriented and does not support object-oriented terminology; therefore, WSDL is the same.

- The **message** element represents a single piece of data moving between the requester and the provider (or vice versa). It declares the name of the message along with zero or more **part** elements, each of which represents either a single parameter (if this is a request) or a single return value (if it is a response). If there is no part, then the request requires no parameter or there is no return value, depending on whether the message represents a request or a response. Note that each message element declares only the name (which is used by the **operation** element below), value(s), and the type of each value; it does not specify whether the message is for input or output—that is the role of the next element.
- The **portType** element represents a collection of one or more operations, each of which has an **operation** element. Each operation element has a **name** value and specifies which message (from the **message** element) is the input and which is the output. If an operation represents a request/response interaction (a method invocation with a return value), then the operation would include two messages. If an operation represents only a request with no response or a response with no request (e.g., an automatic notification from the provider with no request from the requester), it would include only a single message. In Java terms, a portType can best be thought of as an interface; an operation can best be thought of as a single method declaration; a message can best be thought of as a individual piece of an operation, with each message representing (if the operation is an input) a parameter name and the associated type or (if the operation is an output) return value name and the associated type.
- The **types** element is used to declare all the types that are used between the requester and the provider for all the services declared in the WSDL document.
- The **binding** element represents a particular portType implemented using a specific protocol such as SOAP. If a service supports more than one protocol (SOAP, CORBA, etc.), the WSDL document includes a listing for each.
- The **service** element represents a collection of **port** elements, each of which represents the availability of a particular binding at a specified endpoint, usually specified as a URL where the service can be invoked.

Invoking Existing Web Services: A Sample

To invoke a Web Service, we can either write a SOAP client or use an existing generic one. The *www.soapclient.com* site provides a Web interface that allows us to enter the WSDL file and invoke the service. Before we can launch the service, we need to find the WSDL file. In this case, we can find some sample WSDL files at *www.xmethods.net*, a public repository of Web Services. For our example, we will invoke a Web Service that can print the traffic conditions of a specified California highway. Use the following instructions:

- Visit the *www.soapclient.com/soaptest.html* site.
- Type *www.xmethods.net/sd/2001/CATrafficService.wsdl* in the WSDL address field.
- Select **HTML** instead of **XML** for the output.
- Click **Retrieve**, which loads the WSDL file from across the Internet.
- In the textfield, type **101** (for Highway 101) and click **Invoke** to invoke the service.
- The resulting screen should print text that explains the current conditions for Highway 101.

This example illustrates how straightforward it is to invoke a Web Service from a browser. The user, in most cases, will not even be aware that Web Services are being used to return the values. Of course, the user can just as easily be a program, in which case the program would programmatically pass the appropriate parameters.

Universal Description, Discovery, and Integration (UDDI)

The vision behind **UDDI** is to provide a distributed repository that clients can search (during design time and runtime) to find Web Services. Originally launched as a collaboration among Microsoft, IBM, and Ariba in September 2000, the UDDI consortium has since grown to include hundreds of members.

UDDI can be thought of as two major concepts:

- **The specifications.** These standards describe how such repositories should work and include three major concepts: white pages, yellow pages, and green pages. These will be described further below.
- **The implementations of the specifications.** Microsoft and IBM are UDDI operators of two public repositories, called b**usiness registries,** which are the first public implementations of the UDDI specifications. A company can register at one repository and be confident that the entry will be replicated to the other repository (currently, the entries are replicated every 24 hours). However, for security reasons, any updates must be performed at the repository where the service was first registered. Of course, like everything else that is related to Web Services, the entries in the UDDI repositories are XML data. As mentioned, the business registries are examples of public registries; we will discuss private registries in detail below.

UDDI Categories

A UDDI repository contains entries about businesses, the services these businesses provide, and information on how those services can be accessed. Modeled after a phone book, a UDDI directory has three categories:

- **White pages** contain basic information about a service provider, including the provider's name, a text description of the business (potentially in multiple languages), contact information (phone number, address, etc.), and other unique identifiers such as the Dun & Bradstreet (D&B) rating and the D&B D-U-N-S Number.
- **Yellow pages** include the classification of either the provided service or the registered company using standard taxonomies. Examples include
  - Standard Industrial Code (SIC)
  - North American Industrial Classification System (NAICS): a classification scheme specific to the United States, Canada, and Mexico

    - Universal Standards Products and Services Classifications (UNSPSC): an open global standard used extensively by catalog and procurement systems
    - Geographic taxonomies: location-based classifications; for example, US-CA indicates a business in California
- **Green pages** contain the technical entries for the Web Services—the address of the Web Service, the parameters, etc.

The entries in a UDDI directory are not limited to Web Services; UDDI entries can be for services based on email, FTP, CORBA, RMI, or even the telephone.

UDDI Data Model

The UDDI data model includes an XML schema that provides four major elements:

- The **businessEntity** element represents the owner of the services and includes the business name, description, address, contact information categories, and identifiers. Upon registration, each business receives a unique **businessKey** value that is used to correlate with the business's published service. The categories and identifiers can be used to specify details about a business, such as its NAICS, UNSPSC, and D-U-N-S codes—values that can be useful when performing searches.
- The **businessService** element has information about a single Web Service or a group of related ones, including the name, description, owner (cross-referenced with a unique **businessKey** value of the associated **businessEntity** element), and a list of optional **bindingTemplate** elements. Each service is uniquely identified by a **serviceKey** value.
- The **bindingTemplate** element represents a single service and contains all the required information about how and where to access the service (e.g., the URL if it is a Web Service). Each binding template is uniquely identified by a **bindingKey** value.

The service does not have to be a Web Service; it can be based on email (SMTP), FTP, or even the fax.

- The **tModel** element (shortened from "technical model" and also known as the **service type**) is primarily used to point to the external specification of the service being provided. For a Web Service, this element (more specifically, the **overviewURL** child element) should ideally point to the WSDL document that provides all the information needed to unambiguously describe the service and how to invoke it. If two services have the same tModel key value, then the services can be considered equivalent (thus allowing the requester potentially to switch from one service provider to another). Here is a useful metaphor: a tModel that can be thought of as an interface for which there can be multiple implementations, presumably from different companies since it does not make sense for a firm to implement more than one service for a given tModel (just as it would not make sense for a single class to implement the same interface in different ways).

UDDI Usage Scenarios

As mentioned earlier, a service-oriented architecture involves three roles: client, provider, and service broker. To illustrate the flexibility of the UDDI model, we'll use the example of the MegaBucks Consortium.

MegaBucks Consortium (a financial consortium) wants to create a standard way of valuing small retail businesses and then publish this information so that its member firms can implement Web Services to conform to that standard. In this example, the consortium operates a private registry (see the next section on types of UDDI registries for more information on private versus public registries). To allow its members to access this standard, the consortium needs to

- Produce a WSDL file to define the specifications to which a valuation service should adhere. The provider of the valuation service would most likely be a member of the financial consortium.
- Publish the WSDL file at a public location (for example, *www.megabucks.org/valuation.wsdl* on its server, or any publicly accessible location).

- Create a tModel to represent the valuation service specifications (these specifications are described in the WSDL file mentioned in our first bullet point).
- Publish the tModel in its registry. As part of the publishing process, the registry issues a unique key for the tModel (5000X, for example[6]) and stores the URL of the WSDL file in the overviewURL child element of the tModel element.

Figure 2–6 illustrates this sequence of events.

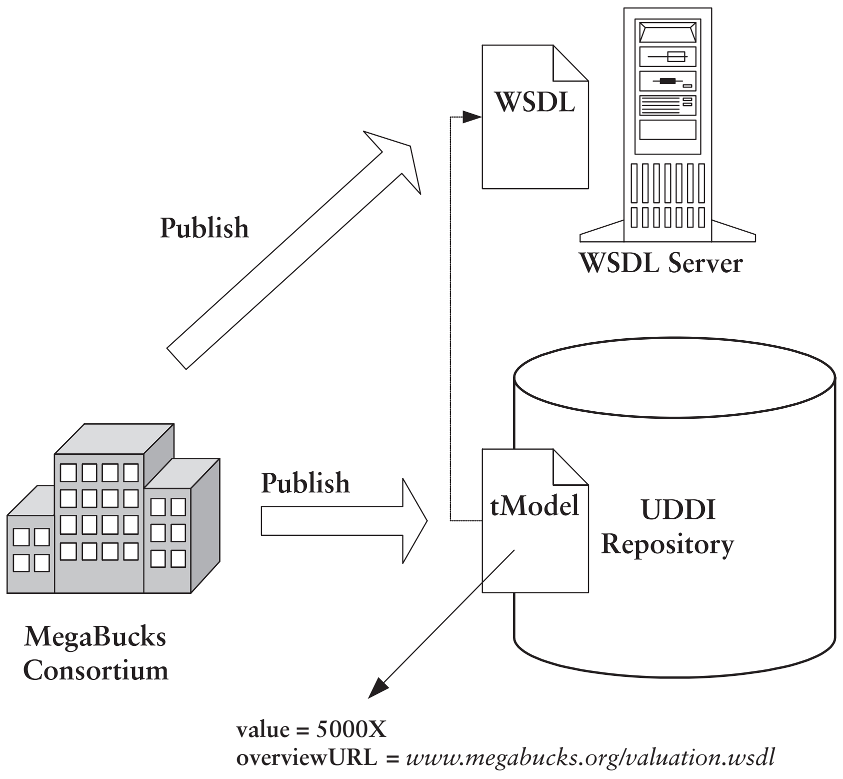


Figure 2–6 Creating a tModel.

6. The value of a tModel key is considerably more complex than this, but this simplification is adequate for our purposes.

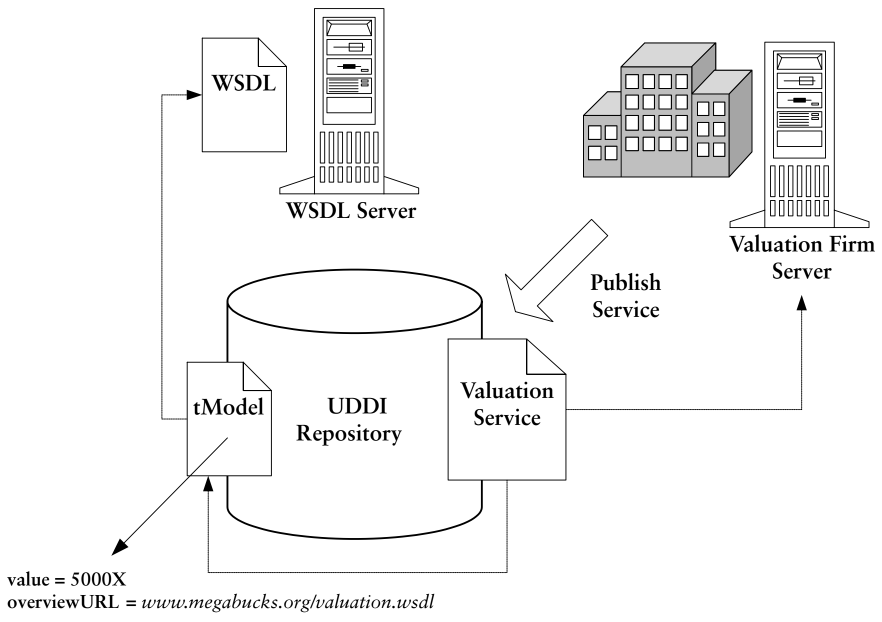


Figure 2–7 Using the tModel.

A firm that wants to publish a Web Service to comply with this standard (presumably a member of the MegaBucks Consortium) needs to do the following (see Figure 2–7 for the sequence of events outlined in the bullets):

- Publish its business to the private registry operated by MegaBucks Consortium.
- Publish the Web Service with a bindingTemplate, access point of which is the URL of the firm's Web Services implementation and whose tModel's value (5000X) is that of the tModel published by MegaBucks Consortium. In essence, this member firm is advertising that its valuation service complies with a set of specifications (captured in the tModel) established by a consortium (in this case, Mega-Bucks Consortium).

As mentioned earlier, multiple firms can publish Web Services that implement the same tModel.

A firm that actually wants to invoke the valuation service (e.g., a holding company that buys other businesses) first has to know the value that represents the valuation service (in this case, 5000X) and must use this value to locate the Web Service either statically (by browsing the operator nodes via the Web) or programmatically. If the search results in more than one service, then the client can use other criteria (price, location, etc.) before selecting a provider.

In our example:

- MegaBucks Consortium is a publisher (because it is publishing a service—the tModel) and a service broker (because it is hosting a repository that requester firms are searching).
- The member firm—the one that is providing the actual valuation service—is the provider.
- The holding company is the client.

Figure 2–8 illustrates the requester locating the valuation service and invoking it.

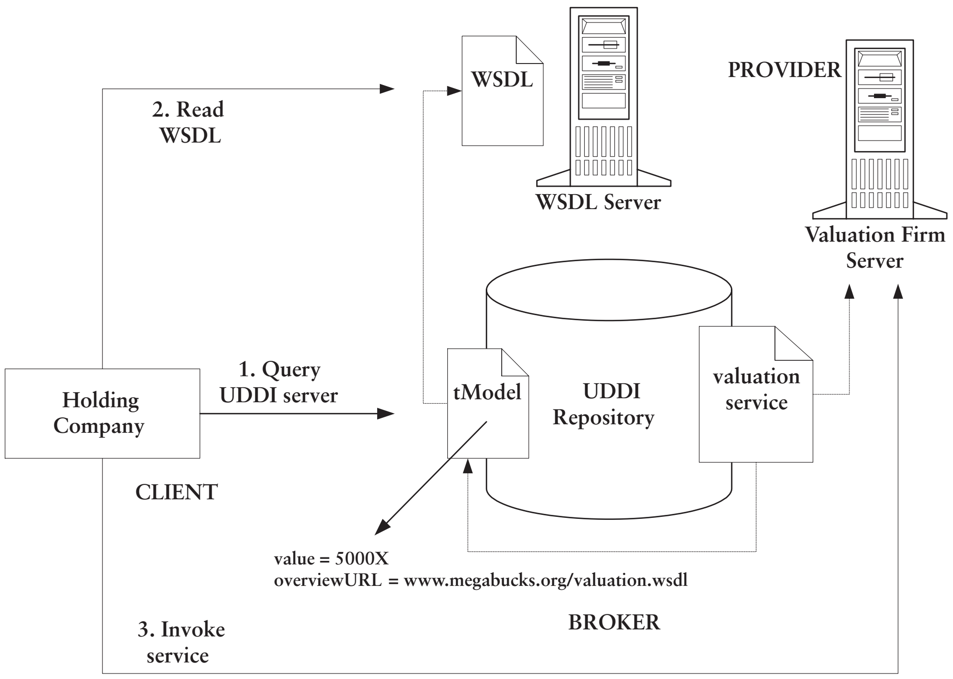


Figure 2–8 Locating and invoking a Web Service.

Types of UDDI Registries

UDDI registries can be categorized into two major groups:

- **Public.** Anyone can publish an entry in a public registry, which has no process to ensure the validity of its entries. (The business registries operated by IBM and Microsoft are examples of public registries.) Because an entry is not validated, there may be questions as to whether the business actually exists, whether the services are even provided, and whether the services are delivered at an acceptable level. For these and other reasons, many believe that public registries will not be feasible for a long time.
- **Private.** A private registry is a more likely scenario for the majority of the firms because each firm can enforce certain criteria on an entry before it is published to the repository. There are different variations of private registries including
  - **EAI registry.** This is useful for large organizations that want to publish commonly used services by various departments or divisions. Without a central repository, these services are often duplicated. An example would be a service for accessing the human resource legacy system.
  - **Portal UDDI.** The registry is located behind a firewall. Therefore, the external users can search for entries, but only the operators of the portal can publish or update the entries in the portal. In a sense, this is the model of the portals today. Users can browse and invoke services (such as stock quotes), but they cannot add new services (although they can personalize the views).
  - **Marketplace UDDI.** Only members of the marketplace (typically a closed environment) can publish and search for services. This type of registry is appropriate for vertical industries. The marketplace operator can establish qualifying criteria before an entry is added to the repository and can then provide additional fee-based services such as certification, billing, and nonrepudiation.

Web Services and Other Technologies

Now that we have explained the basics of Web Services, it is worth reviewing whether Web Services will coexist or whether they will replace many of the technologies we've discussed. Remember, a WSDL document advertises the methods that can be invoked, and SOAP provides the mechanism for invoking the methods. However, there is still a need to have back-end applications take the SOAP request and perform the processing. This functionality is still provided by some applications, which can be written in a client-server or *n*-tier architecture. The following scenarios elaborate how existing technologies can be affected by the adoption of Web Services.

- **Application servers, middleware, and object-oriented technologies.** Recall that application servers are written predominantly in Java; it follows that applications using application servers must be written in Java as well. Without SOAP, Java applications must use either RMI (which allows communications only with other Java programs) or CORBA (fairly expensive and difficult to learn) for integration to legacy systems. Most popular application servers now provide SOAP support. Furthermore, through SOAP, applications built with an application server (i.e., Java applications) can now communicate with programs in other languages, regardless of the language in which they are written (provided that language has SOAP support). However, keep in mind that, in many cases, CORBA is still the only viable solution for connecting systems speaking different languages and/or on different operating systems because it defines many features (real-time extensions, etc.) that are not available with Web Services. The key is to determine what needs to be done and address the missing functionality. In many cases, a viable option is to use third party products, such as Web Services networks, to address some of the gaps in the existing standards. We'll talk more about Web Services networks in Chapter 6.
- **ERP, CRM, and EAI systems.** ERP and CRM systems provide the core functionality for many firms and will continue to do so even with the emergence of Web Services. In some cases, Web Services may replace the simpler integration scenarios between CRM and ERP currently addressed by the lower-end EAI solutions. However, as of now, the base Web Services do not address

many of the advanced features found in the higher-end EAI solutions—transaction control, message integrity, queuing, to name a few. For an in-depth look at the types of cases in which it can be beneficial to use Web Services in place of traditional integration, see the case studies in Appendix B. For a more thorough discussion how EAI and Web Services will coexist, see the JCA section of Chapter 4.

- **EDI.** There are conflicting points of view about whether Web Services will make EDI obsolete; quite a few believe that EDI will be around for a long time. First, a lot of money has been invested in EDI by major corporations such as Wal-Mart and General Motors. Furthermore, EDI provides a data exchange mechanism and a set of predefined business processes. As of this writing, without the adoption of ebXML or something similar, Web Services do not address the issue of business processes. For more information on ebXML, see Appendix A.

Phases of Adoption

In most surveys conducted by leading analyst firms (Gartner Group, Forrester, IDC, and others) with IT decision makers, many respondents have consistently ranked Web Services as a technology that will be adopted in their enterprises. However, the adoption will not happen in a single large wave. According to these same firms, the adoption will happen in the three distinct phases described in the next three sections (for other perspectives on the phases of adoption, see Appendix C, which includes in-depth interviews with executives at Web Services firms).

Phase I (2002–2003+)

In this phase, organizations will adopt Web Services as a more affordable way of performing application integration behind the firewall; they will launch pilot projects to gain some hands-on experience. A natural point of entry will be when a firm chooses to use Web Services instead of conventional middleware to integrate **enterprise information portals** (**EIPs**) from multiple disparate data sources. Because most firms already have a portal strategy and/or deployment, this would be a low-risk incremental strategy to save on integration costs. The lack of Web Services transactional standards will not be a huge deterrent here since many information portals are not transactional in nature.

Figure 2–9 illustrates how Web Services are used with an enterprise portal.

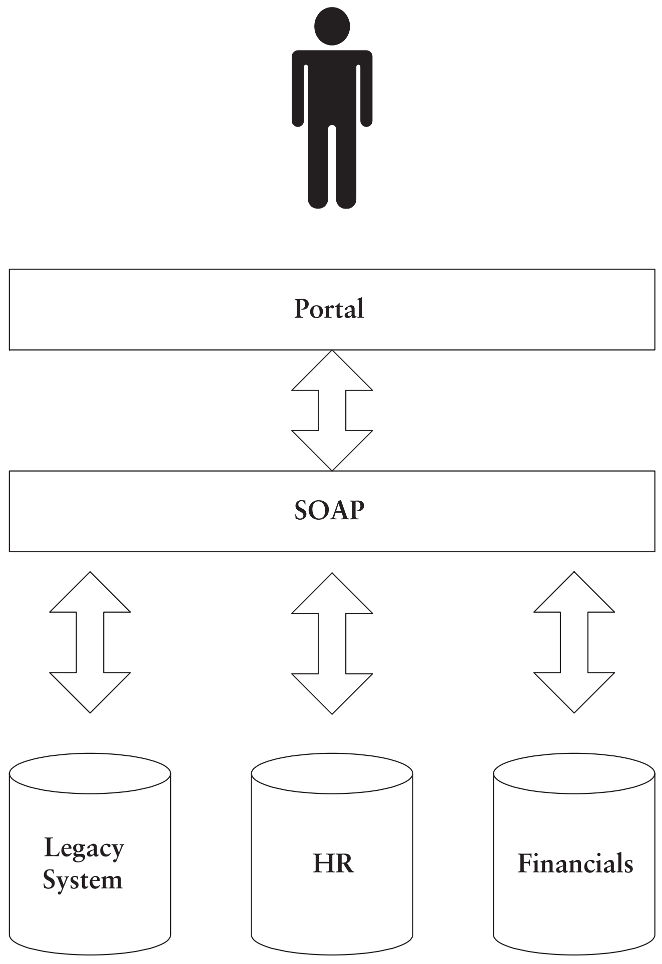


Figure 2–9 Portal integration via SOAP.

Phase II (2003–2005)

As the standards mature (especially regarding security, messaging, and transaction control), organizations will start integrating business processes and applications beyond the firewall. Workflow standards will also mature to the point where organizations can build sophisticated, collaborative systems with trading partners. For a more thorough discussion of workflow standards, see the Silver Stream interview in Appendix C.

Phase III (2006+)

By this time, the repositories should contain a critical mass of publicly available Web Services. This will allow business analysts to start building complex applications by statically assembling these available Web Services, which were once the exclusive province of developers. This may even include using software agents—programs that can act on behalf of a user—to dynamically change the behavior of the system by dynamically reconfiguring the workflow to react to changing business conditions.

Summary

The term Web Services defines a set of lightweight protocols and standards (SOAP, WSDL, and UDDI) that facilitate integration. Adding a thin layer on top of standard XML, SOAP provides a lightweight protocol for exchanging data, invoking applications remotely, and handling exceptions. SOAP does not provide the functionality itself, but instead provides a platform- and language-neutral way of forwarding an incoming request (frequently an HTTP request, although FTP and SMTP are also supported) to the appropriate method or function in an existing application and then returning the value to the requester.

WSDL provides a way for service providers to advertise the list of operations they are willing to support. WSDL is quite verbose and is often generated by tools rather than being written by developers.

UDDI provides a set of specifications for companies to use when registering their services. Major companies, including IBM and Microsoft, are also hosting public repositories, which are currently not being audited. For most organizations, it is more practical to build private registries that can be updated and searched only by trusted parties.

In many cases, Web Services will not replace the existing technologies (application servers, EDI, EAI, etc.); they will instead coexist with them. The next chapter discusses outstanding issues that need to be addressed in rolling out Web Services at an enterprise level.