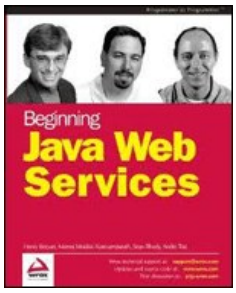


Beginning Java Web Services

by Henry Bequet

Chapter 1: Introducing Java Web Services

Overview

Ideally, it would be easy to define what web services are and what technologies and components make up a web service. Unfortunately, most of the industry has some trouble with a common definition of web services.

According to the Gartner Group:

> "Web services are software components that interact with one another dynamically via standard Internet technologies, making it possible to build bridges between IT systems that otherwise would require extensive development efforts."

Forrester Research says that web services are:

> "Software designed to be used by other software via Internet protocols and formats."

There are probably as many definitions as there are companies promoting web services. But as we begin our discussion on web services, and how to interact with them using Java, we should probably try to define them ourselves. We will do that first by describing them, listing some of their characteristics, and then by showing the standards that make up web services. We will look at how web services relate to Java as an implementation language, a hosting platform, and as a consumer of web services. In short, this book is an introduction to understanding, building, and using web services in the context of Java programming.

In this chapter we will look at:

- What web services are
- A brief discussion on XML
- The business reasons for web services
- The technical reasons for web services
- Java and web services

Web Services Basics

Let's begin by describing what web services are and what they do. Web services:

- Are independent of language, development tool, and platform
- Are based on industry-accepted open standards
- Are loosely coupled, service-based applications that present an API for communication using established protocols to facilitate connectivity and reduce complexity
- Allow automated and direct machine-to-machine communication regardless of the systems involved
- Allow the discovery of published web services utilizing a common mechanism for searching for and publishing services

As we mentioned, another way to define web services is to list the standards by which people would judge whether a particular application is a web service. This is more difficult than it appears, because there is no one standard specification for a web service. Nevertheless, the standard definition includes the following:

- **Extensible Markup Language** or **XML** for data representation in an independent fashion. XML is used not only for the data definition, but also in all of the descriptions within other protocols (for example, the SOAP envelope, which will be addressed in Chapter 2).
- **HyperText Transfer Protocol** or **HTTP** for transportation across the Internet or within Intranets. This provides the mechanism for both data transport and for request/response types of communication.
- **Simple Object Access Protocol** or **SOAP**, which provides a mechanism for request/response definition and allows conversations in a Remote Procedure Call (RPC) format.
- **Universal Description, Discovery and Integration** or **UDDI**, which provides a means of locating services via a common registry of providers and services.
- **Web Services Description Language** or **WSDL**, which provides the detailed information necessary to invoke a particular service. WSDL and UDDI are somewhat complementary, but also overlap in certain areas.

Some people will argue that other protocols, such as messaging, business process management, security or overall management of the application that provides the service also belong to the definition, but to date there has not been complete agreement over inclusion of these items in the industry accepted definition of web services.

Service-Oriented Architecture (SOA)

A concept that we will encounter frequently during our discussions is that of software as a service, or a **Service-Oriented Architecture (SOA)**. The concept of designing software as a service, independent of a user interface has been a part of the software engineering landscape for well over a decade. Early attempts at distributed computing such as the Distributed Computing Environment (DCE) and the Common Object Request Broker Architecture (CORBA) led to the realization that the business logic of an application is not necessarily coupled with the presentation logic. It could be completely decoupled and offered as a service with a defined Application Programming Interface (API).

The Internet, with HTML as a presentation layer, lacked all but the most basic programming constructs, helping to drive the adoption of SOA as a more common design philosophy. Mobile or wireless devices such as cell phones and PDAs with micro-browsers or other presentation schemes highlighted the need to separate the presentation from the business logic.

The separation of the logic into a service provides a number of benefits. The user interface is no longer required to run on the same physical tier as the business logic, and both can be scaled independently. Several platforms exist that provide strong support for services, such as Java and .NET. Both provide transaction management, persistence and some state management, removing many of the complexities of service design (for example, writing thread-safe, multi-threaded code).

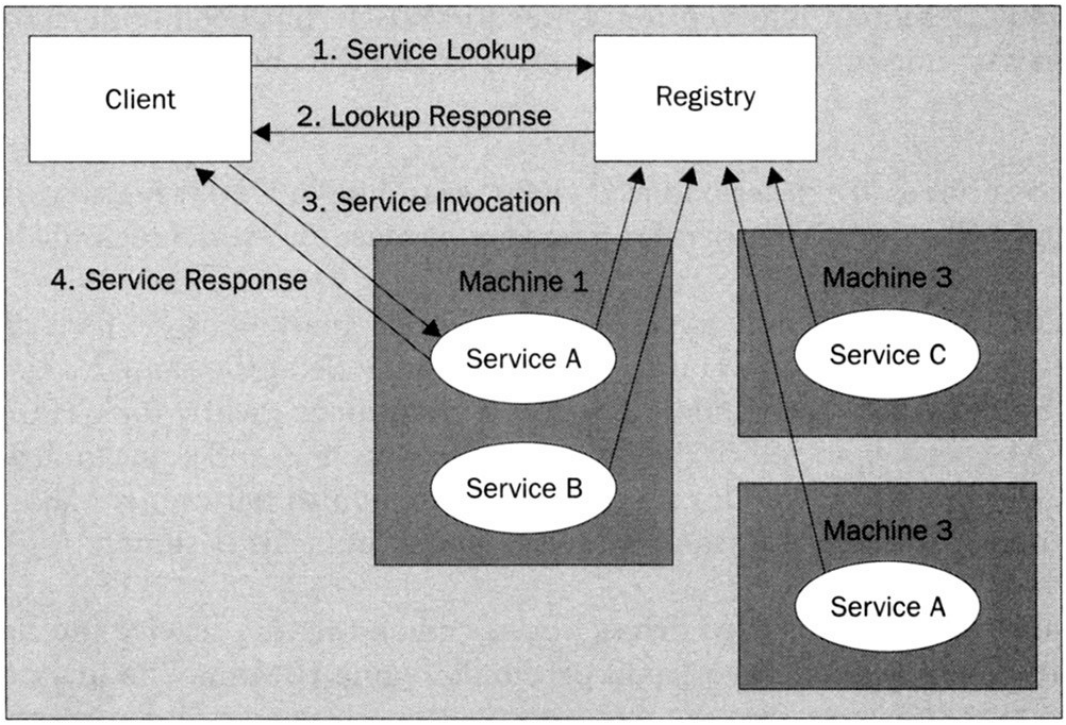
The introduction of Java, with the concept of "write once, read anywhere", accelerated the adoption of service-oriented architectures, first with Enterprise JavaBeans (EJB) and then with the release of the entire Enterprise platform (J2EE), which provided particular paradigms for implementing user interfaces (specifically JavaServer Pages and servlets) as well as increased support for the development of services.

It was quickly realized that for services to be truly neutral and platform independent, a neutral representation of the data exchanged between the client and the service was needed. The Extensible Markup Language (XML) proved to be the needed component.

In an all-Java environment, these would likely be enough, but most corporate environments are not homogeneous. Not every program can utilize RMI to communicate with the server, and the interface exposed by the J2EE server for services leaves out some of the features that we need (like discovery).

A web service is the next logical abstraction of the services architecture. It makes the discovery, description, and invocation of services (for example, written in Java) simple and transparent to clients running on any machine, language, and platform. So, an Active Server Page or an application written in Delphi and running on a Windows PC can easily access services written in Java and running on Unix or a mainframe.

The diagram below represents the typical service contact of Service-Oriented Architecture:



A client contacts a public registry to locate a particular service. It then contacts that service to obtain the service signature to understand the data that is required for the operation, and the data that is returned. Note that data is a very loose term - the return will be in XML in web services, but what is represented in that XML may be a document, an array, or a list of data.

Once the client understands the service signature, it can invoke the service and receive a response. This may be done in a single invocation, or the client and the service may communicate multiple times in a conversation to complete a business transaction.

There are a number of observations that need to be made regarding this diagram:

- First of all, it should be noted that a machine (usually identified by an IP address) could host more than one service.
- It should also be noted that a service could be provided by more than one source (for example, Machine 1 and Machine 3 both host Service A).
- A client would interrogate the registry to obtain the correct service location (a URL in most cases), and then invoke the service.

For web services this is an idealized representation. A UDDI registry would handle the service lookup and response, but the identification of a specific service would typically be by human intervention rather than by automatic recognition.

> **Note** One of the biggest debates in the web services community is to what degree UDDI will actually be used, and whether it will be used on a global basis, or within industries. Most insiders predict that UDDI will not be a global registry on the same scale as the Domain Naming Service (DNS), which helps locate particular machines (for example, http://www.wrox.com is a DNS lookup that ties to a particular server).

Once a service has been identified, the client would need to examine the description - in the form of an XML-based WSDL document - of the service to understand the particulars of the invocation. In most cases, this would require human intervention. Once the WSDL has been understood, an invocation can take place. This may be one or more conversations between the client and the service to accomplish a business task.

In many cases, web services are described as a client, searching a UDDI registry, finding a service, and using it automatically. This won't happen for a variety of reasons - some technical and some social.

First of all, UDDI doesn't provide any provisions for service guarantees. So for example, if you wish to purchase a hard drive, IBM and Joe's Hard Drive Shack may be at the same level. There's no way to know that IBM provides a 99% satisfaction level (if in fact it does), while Joe's Hard Drive Shack won't even provide an RMA service to return defective drives. You may still want to deal with Joe (maybe his prices are insane), but there's no way for a machine to distinguish between a major multi-national corporation and the fellow down the street who sells parts out of his basement.

Additionally, locating providers of hard drives doesn't mean that the service interface (described via WSDL) is the same. Joe may want your credit card only, while IBM may be in a position to accept a purchase order. So to invoke a service, human intervention is once again a necessity.

> **Note** This is a hypothetical situation and in no way implies that IBM provides a web service that sells hard drives via purchase orders. Or that Joe's Hard Drive Shack, if it exists, only takes credit cards.

There is an effort underway to create a web services MetaData specification, which might answer some of these questions. Industry standardization of service interfaces could also alleviate some of these issues as well, but for the moment they exist.

The second observation is that a complete business transaction may involve more than one invocation of a service function. In cases such as these, the atomicity of the entire conversation is critical. This means that if a business transaction, such as ordering a hard drive requires multiple invocations, then there needs to be some mechanism for ensuring that the whole transaction completes or is rolled back.

> **Note** Work is underway on what is known as the "Business Transaction Protocol" to address this need, but for now be warned - it's definitely possible to create only a portion of a business transaction. This should be no surprise to experienced coders - the same situation has existed in databases and other transactional systems for years. Web services currently lack a standard protocol for addressing this need.

What are Web Services?

Let's start with an example that might help us to understand the characteristics of web services.

> **Important** Having agreed upon the technologies involved in web services (HTTP, XML, SOAP, WSDL, and UDDI), we can infer that a web service is a software component that provides a consistent, transparent API for invocation of services by using message-oriented transport mechanisms, which can be dynamically located and bound, and by utilizing XML for data representation and transformation.

As such, any client that can bind itself to the particular service, regardless of client platform, language or hardware, can invoke a web service.

Consider a simple application designed to let us monitor the checkbook. This is a standalone, single-user application with a simple purpose to keep track of debits and credits for our checking account. Suppose we have several accounts in different banks. Each month we receive a statement generated by each bank's computer listing the transactions. Then we enter them into the program and make sure we haven't missed anything or that the bank hasn't made a mistake.

Now we connect our computer to the Internet. The banks can send us statements via e-mail, but we still have to enter them into the program manually because the program was not designed to be aware of other applications. Therefore, the developer adds a data import facility in the next version of the program. Let's assume it's fairly basic and requires a fixed format, one that not all of the banks follow. Some banks may put the transaction type first, some may put the date, and some may put the amount first. The next release of the software won't be for another year. So, even though this is useful to an extent, we will have to do other things if we don't want to type the data in.

We may use a spreadsheet to manipulate the data from the bank and translate it into the format required by the checkbook program. This

reduces the time to enter the data, but why can't the checkbook program talk directly to the bank whenever we want, or automatically get the information without us having to do anything at all (other than review it so we don't bounce a check)?

The answer is that our program and all the software involved in the communication, such as the bank's mainframe accounting system, need to communicate with one another. Since it's not just the checkbook program but other vendors' programs too that are involved, we need to move towards standards and approaches that facilitate application interoperability. Web services provide us with a platform in which these issues can be handled.

Of course there are alternatives such as Electronic Data Interchange (EDI), private leased lines for communication, which essentially render the system a giant, client-server application, and object technologies, such as Common Object Request Broker Architecture (CORBA), and Enterprise JavaBeans (EJB). These technologies have salient points but exhibit some drawbacks when compared to web services.

EDI requires too much overhead and requires intense effort to implement. Business reasons usually preclude allowing outside companies access to internal applications in the manner of a client-server program, and the more the number of companies interacting, the more likely it is that they will use different systems, requiring multiple terminals and not allowing transparency of data. CORBA and EJB provide fairly robust communication mechanisms, but the APIs are either complex (CORBA), or proprietary (EJB). Web services allow us to make the service available to anyone we want, regardless of computer, as long as Internet access (or intranet if we are only within a company) is available.

To solve our dilemma, we need to develop several things. First of all, to address the common problem of data opaqueness, we need a way to represent our data in a format that our applications can understand. We need to describe elements of arbitrary complexity in data in such a way that an application can clearly distinguish the elements. It must be self-describing, so that any application that receives the data will be able to use what it needs without being written expressly for that data format. There are a number of choices available for solving this data problem.

We could try to establish an industry standard for the data interchange. Industries have attempted this with EDI before, but unfortunately EDI is fairly limited in its capacity. We could develop a programming interface, but that reflects the underlying language or platform of the system, which is not optimal. Or we could use XML to encode the data, which is the approach that web services takes. This has the advantage of establishing a neutral format for the data, regardless of the systems involved. With the banking example, we can see how XML can help us.

Suppose Bank A sends its data like this:

```
    12/2/2002, DEBIT, 123.45, SHOPRITE AUTOMATIC TRANSACTION
    12/4/2002, CHECK, 1200.00, 1270
    12/27/2002, CREDIT, 4999.00, PAYROLL DIRECT DEPOSIT
```

We can see that the format is date, transaction type, amount, and description. The data is comma separated, the amount is in dollars, and the dates use the American format - month, day, and year. In this case, we can see that the record length is variable, as are the field lengths.

Bank B may use a different format. Suppose it sends its data like this:

```
CR2002-11-230000000123.45              Payroll Deposit
DT2002-12-010000000050.00              Shoprite Auto Trans
CK2002-12-020000012345.00              Check# 2001
```

We can see that Bank B formats the data as transaction type, date, amount, and description. The record length and field lengths are fixed, with the amount being padded with zeros, the description right justified, the transaction type using abbreviations (a code of some sort), and the date is in year, month, and day.

We as humans can see it. Computer applications may not be able to do the same. We need something that will let us describe the data, so the application can be sure of what it means. This is where XML comes in.

XML - Extensible Markup Language

This book is not intended to provide extensive coverage of XML. For a detailed look at XML, see *Professional XML 2nd Edition* from Wrox Press (ISBN 1-86100-505-9). Nevertheless, a brief description of the basics of XML will be helpful. Chapter 2 will cover more advanced topics such as XML Namespaces.

As we have seen, XML stands for Extensible Markup Language. Like HTML, and its predecessor, SGML, XML uses tags to define elements of a document.

XML was envisioned to be a self-describing language that would allow both humans and computers to read or use the data described by an XML document with equal facility. XML provides a facility for creating tags, embedding additional information within the tags, and providing a validation mechanism, which allows a computer program to validate that the document is well formed and complete (unlike HTML, which can be malformed). Extensive information on XML is available at http://www.w3.org/XML.

To begin with, XML is case sensitive (unlike HTML), and requires either an opening and closing tag, or a tag that both opens and closes an element. An element name must begin with a letter, an underscore, or a colon, though in practice the colon should not be used. The name must not begin with the string "XML" in any combination of upper or lowercase.

An XML document consists of an optional prolog, which typically describes the version of XML, a body, which contains the information, and an optional epilog, which contains comments or processing instructions.

Within the body, there are **elements.** These elements may contain other elements - they are the basic building blocks of XML. Elements are

delimited by **tags,** which consist of an element type name enclosed between angle brackets (<>). Each element must have a start tag and an end tag. Optionally an empty element (one with no data, just instructions) can be included using the format: `<element name />`. So for example, a tag might be:

```
<account_number>1571</account_number>
```

or:

```
<level name="1"/>
```

Elements can be nested within other elements at an arbitrary depth. The second tag example shown above also illustrates the concept of **attributes** within a tag - the tag is `level`, but the attribute is `name`.

Another important aspect of XML is the comment, which is XML is indicated like this:

```
<!-- ...Some Comment Text ... -->
```

Occasionally data elements may contain characters that would otherwise be recognized as part of the markup language (`–, <, >`). To make sure that these characters are not misinterpreted, XML includes a tag known as `CDATA`. The tag looks like:

```
<! [CDATA [...]]>
```

This means that anything within the inner brackets `[…]` is not interpreted by the XML system as XML, but rather as a data element. This allows us to include a variety of information in a tag that would otherwise be interpreted as XML and displayed or processed incorrectly.

Document Type Definitions

Earlier we discussed the ability to validate an XML document as a significant advantage of XML. In order to validate an XML document, a validation program must be able to reference the document's **Document Type Defintion (DTD).** It defines the document's vocabulary. It also allows a **parser** (a software program that can interpret XML to a certain extent) to validate the document's conformity to the definition. For more information on DTD visit http://www-106.ibm.com/developerworks/library/buildappl/.

DTDs are fairly esoteric. They contain a number of elements that help describe a document, but these elements do not necessarily lend themselves to easy inspection, as they define a specific grammar and taxonomy for elements of a document. A document needs a DTD to be validated, and multiple documents can have a common DTD.

XML Example

Here's an example XML document, taken from the book *Professional XML 2nd Edition* from Wrox Press (ISBN 1-86100-505-9):

```
<holiday>
    <journey>
        <from>London Gatwick</from>
        <to>Orlando, Florida</to>
        <date>2000-02-15 11:40</date>
        <flight>BA1234</flight>
    </journey>
    <journey>
        <from>Orlando, Florida</from>
        <to>London Gatwick</to>
        <date>2000-03-01 18:20</date>
        <flight>BA1235</flight>
    </journey>
    <visit>
        <hotel>Orlando Hyatt Regency</hotel>
        <arrival>2000-02-15</arrival>
        <departure>2000-03-01</departure>
    </visit>
</holiday>
```

We can see that a holiday consists of multiple journeys, plus multiple visits (in this case, only one). Each `journey` element has nested elements that describe the end points of the journey, plus the date and the flight.

Each `visit` describes the lodging, plus the date of arrival and departure. There could, of course, be multiple visits.

This document describes a high-level holiday, the journeys required to accomplish it, and the visits needed to complete the holiday. It omits the prolog and epilog. Still it is a good example of the kind of encoding that an XML document provides.

In our data, let's see how we can define a transaction. We will need to know the start and end of the transaction data, the type of transaction, the amount of the transaction, the date of the transaction, and a description. Bank A data could look like this in XML:

```
<statement>
  <transaction type="debit">
      <date>
          <month>12</month>
```

```
            <day>2</day>
            <year>2002</year>
        </date>
        <amount>123.45</amount>
        <description>SHOPRITE AUTOMATIC TRANSACTION</description>
    </transaction>
    <transaction type="check" number="1270">
        <date>
            <month>12</month>
            <day>4</day>
            <year>2002</year>
        </date>
        <amount>1200.00</amount>
        <description>1270</description>
    </transaction>
    <transaction type="credit">
        <date>
            <month>12</month>
            <day>27</day>
            <year>2002</year>
        </date>
        <amount>4999.00</amount>
        <description>PAYROLL DIRECT DEPOSIT</description>
    </transaction>
  </statement>
```

The `statement` element is important as the top level can have only one set of data. This element encapsulates as many transactions as are needed.

Bank B data would look like this:

```
  <statement>
    <transaction type="credit">
        <date>
           <year>2002</year>
           <month>11</month>
           <day>23</day>
        </date>
        <amount>123.45</amount>
        <description>Payroll Deposit</description>
    </transaction>
    <transaction type="debit">
        <date>
           <year>2002</year>
           <month>12</month>
           <day>1</day>
        </date>
        <amount>50.00</amount>
        <description>Shoprite Auto Trans</description>
    </transaction>
    <transaction type="check" number="2001">
        <date>
           <year>2002</year>
           <month>12</month>
           <day>2</day>
        </date>
        <amount>12345.00</amount>
        <description>Check 2001</description>
    </transaction>
  </statement>
```

This is a complete XML document, though not fully informative. We are missing the prolog and the epilog sections. Additionally, there is no DTD that tells us what the data format is.

Let us suppose that Bank A orders the elements of a date as month, day, year while Bank B orders them as year, month, day. Since the tag completely specifies the type of information, an application can read the data correctly regardless of the order. Since XML is text based, it can be transmitted over HTTP and can also traverse firewalls and proxy servers easily.

Now we have solved one problem - that of representing the data - but we still have several other problems left to tackle. We need a way to discover how the bank offers this information, and ways to describe the service and to ask for the information once we have located the service and understood how to invoke it. UDDI, WSDL, and SOAP are candidates for this type of operation. It's important to note that subsets

of these standards are also appropriate candidates.

An all-SOAP solution is certainly a possibility, although it presupposes the ability to locate and define the API for a service. But that presupposition is often the case, and thus a number of web services protocols may be unnecessary at the lowest level. They exist, however, because the lowest level is not the only level on which interaction takes place. Other topics such as business transactions must take precedence over pure interaction based protocols.

Universal Description, Discovery and Integration (UDDI) is the means by which services can be published. It is a standard developed and promulgated by a consortium of software vendors and other interested business parties. Information on UDDI can be found at http://www.uddi.org.

Why do we need to locate a service? It is for the same reason we need a phone book. While we may remember the phone numbers of a few friends, when we need access to someone we don't know, we go to the phone book and look them up. We use the white pages or a telephone directory, when we want to look someone up by name and the yellow pages when we want to look them up by category of business.

This is a good area to compare to J2EE's method of finding services. In order to locate a J2EE service, we need to know its complete JNDI name. While JNDI is useful, it does not provide the category hierarchy of UDDI. It is also completely bound to Java, and does not allow clients of other languages to access its functionality.

UDDI allows service providers to establish a provider identity, known as the business entity and register the services it is willing to provide. Business services provide binding information that allows for identification of the type of information to be sent to access the services and a `tModel`, which gives directions to the calling application as to where to access the service. UDDI uses XML to describe services. UDDI and `tModel` will be explained in greater detail in Chapter 7.

There are multiple ways of identifying services. As an example, Bank A and Bank B would both register with UDDI as business entities and would describe the services they provide, in this case statement provision.

Now that we know how to locate a service, we need a description of the service so we know how to use it. For this we will use **Web Services Description Language (WSDL),** which is an XML grammar for describing network services as collections of communication endpoints capable of exchanging messages. WSDL represents a contract between the service requestor and the service. The abstract definition of endpoints and messages is separated from their concrete network deployment or data format bindings. This is also an XML document - in this case one - for describing web services.

We need WSDL to understand the semantics of a particular service. It allows us to define the data types used by the service, the message to be used, and even the binding to a particular invocation mechanism (usually SOAP). Although Bank A and Bank B share a common statement format, they may have different WSDL descriptions. Bank A may require the account number and password, while Bank B may require the account number and social security number of the owner.

Next, we need **Simple Object Access Protocol (SOAP).** This allows us to actually invoke a service - pass a document containing information needed by the service from our application to the service and receive a reply. Bank A and Bank B have different computer systems. SOAP allows us to contact the appropriate computer, transmit our message, and get our reply. It's important to note that SOAP is also based on XML and that each SOAP message is an XML document.

Now that we know a little about what web services are, let us take a brief look at why we need them.

Business Reasons for Web Services

There are both business and technical reasons for web services. Our discussion will begin with some of the business reasons for web services and try to briefly outline what business needs are driving the creation of standards and why. While not every business will have all of these needs, most will have at least some of them.

Probably the most common problem in software development is the need to extend the life of a system or program that was written some time ago. The Year 2000 (Y2K) crisis illustrated this problem effectively, driving home the fact that software often outlives even the most optimistic lifespan estimates of its authors. Y2K was caused by a set of shortcuts used by developers to gain speed, efficiency, or space in code written as early as forty years ago. When the time came to correct these shortcuts, people discovered all sorts of additional surprises.

One of the biggest challenges was that the code wasn't documented and the authors had long since retired. Another was that there were incompatibilities between versions of the languages used like the difference between modern English and the archaic style used by Shakespeare. There were differences in the size of variables from version to version, causing difficulties for anyone who made assumptions about variable lengths.

The lessons Y2K taught us are still valid and in many cases developers are creating smaller, more localized versions of the Y2K crisis in companies across the globe. If we look closely, we'll see a number of common problems. These include:

- **Hardware Dependence**
  Assumptions about word size, byte ordering, and variable length can cause difficulties. For example, if we assumed that an integer was two bytes and had an array of integers that held ten integers, we could assume that the array was twenty bytes long. In some languages, we could access the individual bytes and perform operations on them. Now if we try interfacing this with a system on a different platform, one that defines an integer as four bytes, all of our operations would be wrong.

- **Language Dependence**
  Systems are usually written in single programming language. However, that is beginning to change as languages have varying concepts of

design, such as procedural or object-oriented.

- **Data Opaqueness**
  The lack of a common, transparent format for data and its representation leads to all kinds of problems. Differences in the format of data representation drive up the cost of any integration project.

- **Paradigm Inadequacies**
  The most common programming paradigm is to build a system based on the requirement to perform a business task. Usually this system has user interface, business logic, and some data. Until recently, the concept of building a system did not include the idea that other systems may want to access the business logic or data of the system.

  As an analogy, consider building a house and after it has been built deciding that you want a patio deck with French doors. Unfortunately, the wall you want to put the door in has wiring, plumbing, and heating on it, because you never considered the possibility that you might want a door there. In the world of computer software, programs such as screen scrapers were the first attempt to access mainframe, terminal-based software by using personal computers. Although this approach lessened the problems faced by companies as they tried to make their systems work together, it didn't totally remove them.

The basic problem is to make applications on one computer talk to applications on another computer, or a set of other computers. Thanks to advances in computer technology, computers of any make can communicate by sending electronic signals to each other. What to do with these signals is the concern of operating systems and applications.

Enterprise-to-Enterprise Connectivity

Even after the demise of the dotcom era, the Internet does have a significant influence on the way most corporations do business.

Fundamentally, the adoption of the Internet was about breaking down communication barriers. Some of these barriers were technical while others were simply posturing for business purposes. The worldwide reach of the Internet has allowed customers, partners, and employees to challenge the traditional means of doing business.

Communication needed between enterprises is supplied very elegantly by web services. Large corporations, companies with many divisions and lines of services have a wide network of relationships with other business concerns, as well as with their customers. In any enterprise, electronic communication is a prerequisite.

Corporations utilize a variety of means to communicate with one another. File transfer is common, as is the use of Electronic Data Interchange (EDI) messages.

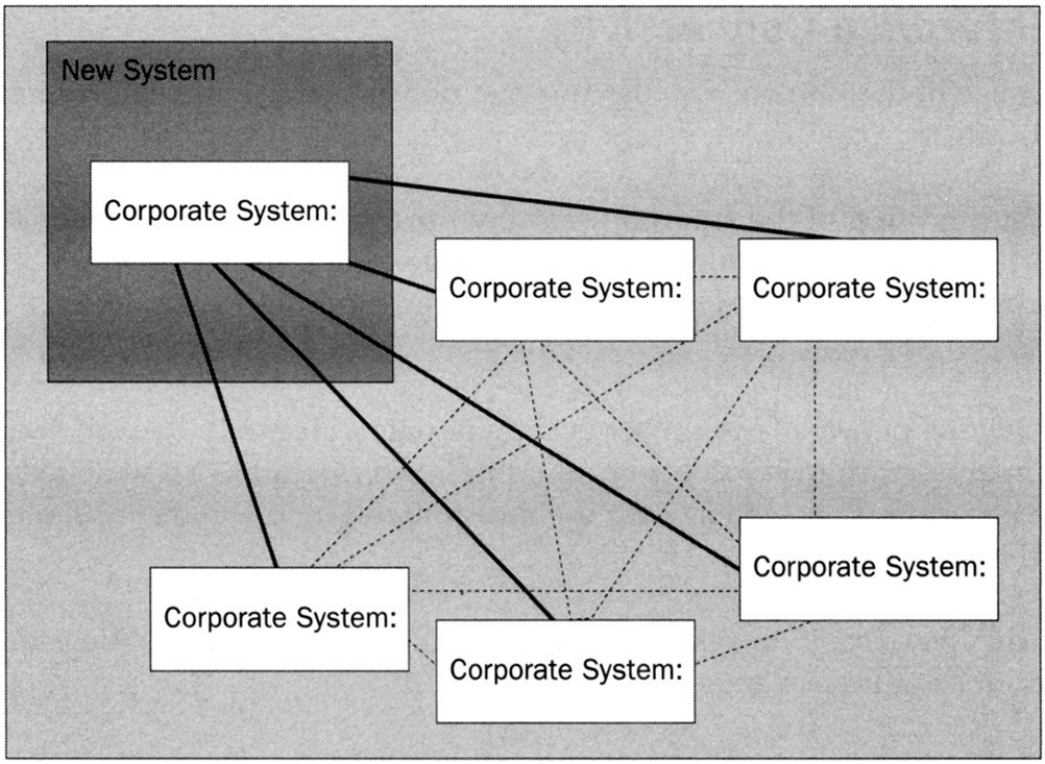
EDI transactions generally work from implementation guidelines that specify fixed message formats. Traditional EDI refers to the use of rigid transaction sets with business rules embedded in them. This model simply does not work in today's rapidly changing business environment. EDI works fine, as long as you keep within your supply chain and have stable content. But business today does not always have that kind of predictability, and EDI transactions has been found difficult to map from one industry to another, and even between different segments of the same industry.

What is the state of EDI today? Industry statistics indicate that the use of EDI in B2B exchanges is far out in front of other web protocols. An IDC study calculated the total value of goods and services exchanged in business-to-business electronic commerce in 2001 at $2.3 trillion, with EDI transactions valued at $1.8 trillion or about 78 percent of that total, and Internet-based e-commerce providing the remaining $0.5 trillion.

E-mail is also used to exchange information. Unfortunately, none of these mechanisms are appropriate for the needs of the business. File transfer is not standardized, EDI is standardized but not easily adaptable, and e-mail is not sufficient in itself to accomplish the task, which is to exchange information.

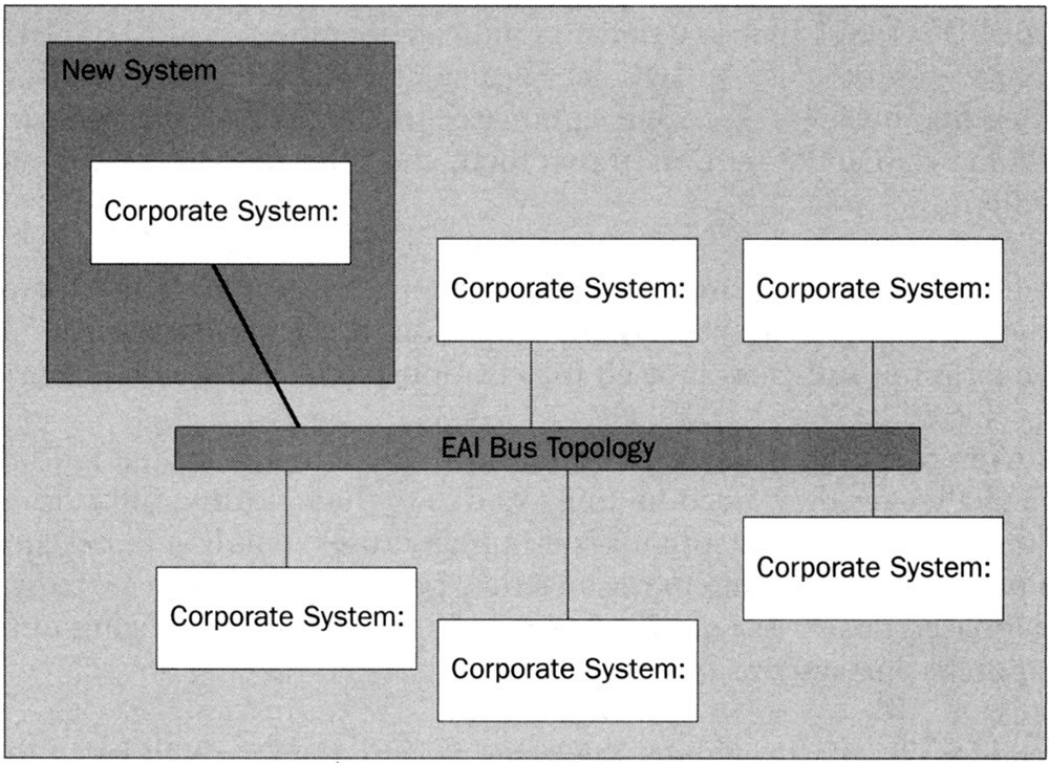
The Internet era made corporations aware of their need to communicate. At the same time, it also made them aware of the challenges they faced in doing so due to their existing infrastructure. It was no coincidence that the startups were first to market in many cases, building a presence in the face of larger corporations and moving at a dizzying pace since they had no legacy infrastructure. They were able to build their Internet-based business quickly because they didn't have to figure out how to integrate their six separate purchasing systems to deliver product.

The challenge faced by IT staff throughout the industry is to do more with less. Old systems must be perpetuated, not replaced. Yet, every new system added to the mix requires interfaces to every other system. As seen in the diagram below, adding one new system requires multiple new interfaces to be created: (n-1) where n is the total number of systems after the addition:



Enterprise Application Integration

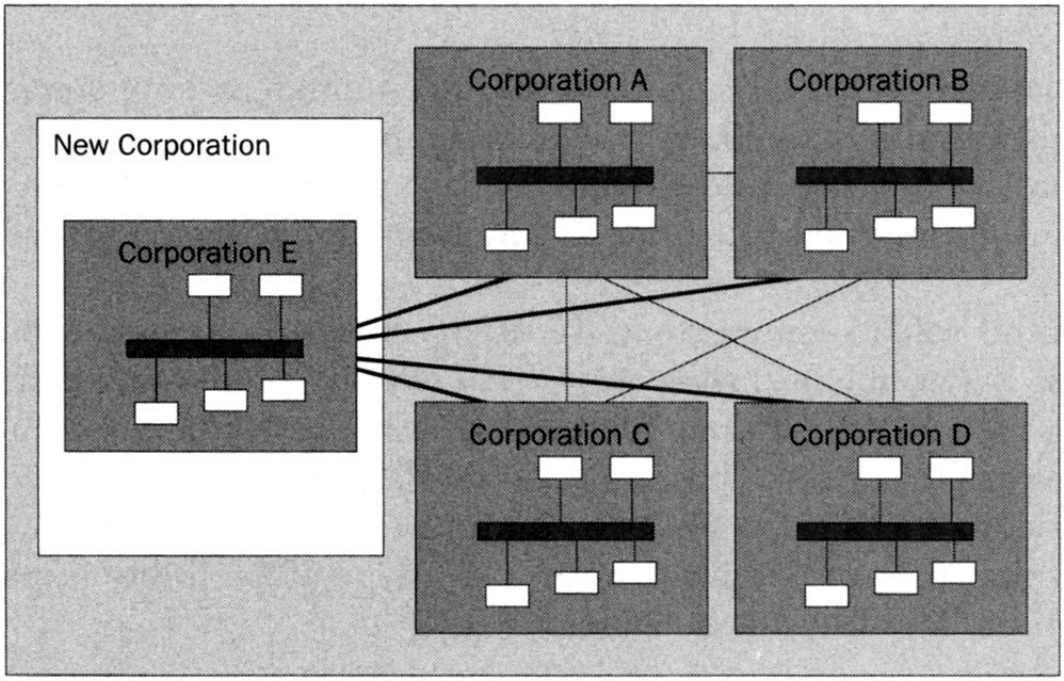
Enterprise Application Integration (EAI) is an attempt to solve part of that problem. EAI introduces a hub or bus concept. Interfaces are written to the hub, with each system requiring only a single interface rather than multiple interfaces:



From a business perspective, EAI is about doing more with less. For example, EAI enables multiple billing systems to function as a single billing conglomerate. From an external perspective, this amalgamation allows users to think of billing as a single system, even though it may be a conglomerate of multiple applications. It allows a business to create a standard interface for its systems so as to reduce the complexity of making them interoperate. From an internal perspective, EAI is a definite aid.

However, it breaks down at the borders of the corporation, as it is proprietary and not truly Internet based. The proprietary nature of EAI

solutions means that at the enterprise level, a single, simple interface does not exist between corporations. Companies using different EAI products once again had to write multiple interfaces and since not all corporations use EAI, other interfaces needed to be developed as well:



Web services solve that problem. The UDDI registry allows a company to advertise its services to any and all. WSDL allows it to describe the interfaces by which it delivers these advertised services. SOAP allows for the transmission of messages adhering to the interfaces between cooperating partners. Since the technology is based on open standards, implementations on a wide variety of platforms exist, removing the need for proprietary interfaces. It is worth noting that all the major EAI vendors either already support web services, or will support them in the near future.

A number of simplifications are at work here. We have not touched the topic of security at all. Neither have we discussed what business services should be exposed through UDDI. Also we have not looked at situations where more than two corporations need to interact in the same business transaction. Nevertheless, we have begun to solve the basic business problem - allowing corporate systems to interact with one another. We will look at the answers to some of those questions (such as security) in later chapters. However, it is worth looking at application integration from a business perspective as well.

Application Integration

We also have the problem of application integration. Major corporations have a number of systems, often because of mergers and acquisitions or multiple lines of business having differing needs. One of the consequences of this type of environment is difficulty in obtaining timely and accurate information about the state of a corporation. Another consequence is that duplicate systems and data exist.

Often, the question "System of Record" becomes an issue. "System of Record" is the concept that the repository should be the only receptacle for the data, the system of record against which copies are compared. An example is the concept of a customer record. If a company employs a billing system, an accounting system, an inventory system, a tracking system, and a manufacturing system, all of these may have some concept of a customer. Depending upon which transaction, or what state, the question of which system is the master, the true data, becomes a key question. At times it can be impossible to designate the key system. Money is lost when records don't match and shipments are sent to the wrong address, or not billed, or shipped twice.

Another issue is the use of multiple systems for fulfillment of a business process. A corporation might take orders via its web site. To do so, it needs to have a web application. It needs to access inventory and track work in progress. It may use a third party provider to do credit card transactions, while having an accounting application to track the funds. It might do its own shipping, or might use a different shipping firm each time. It might have to provide interim status of an order. All these systems need to be coordinated to provide the customer with their order in a timely, seamless fashion.

When we consider that these systems are likely to reside on different computers with different operating systems, and that the applications are likely to be written in different languages, we can start to gain an appreciation for the task of the corporate IT staff. Package solutions are of some aid, as they typically provide some mechanisms for interaction, but it's important to have some means of uniting these systems.

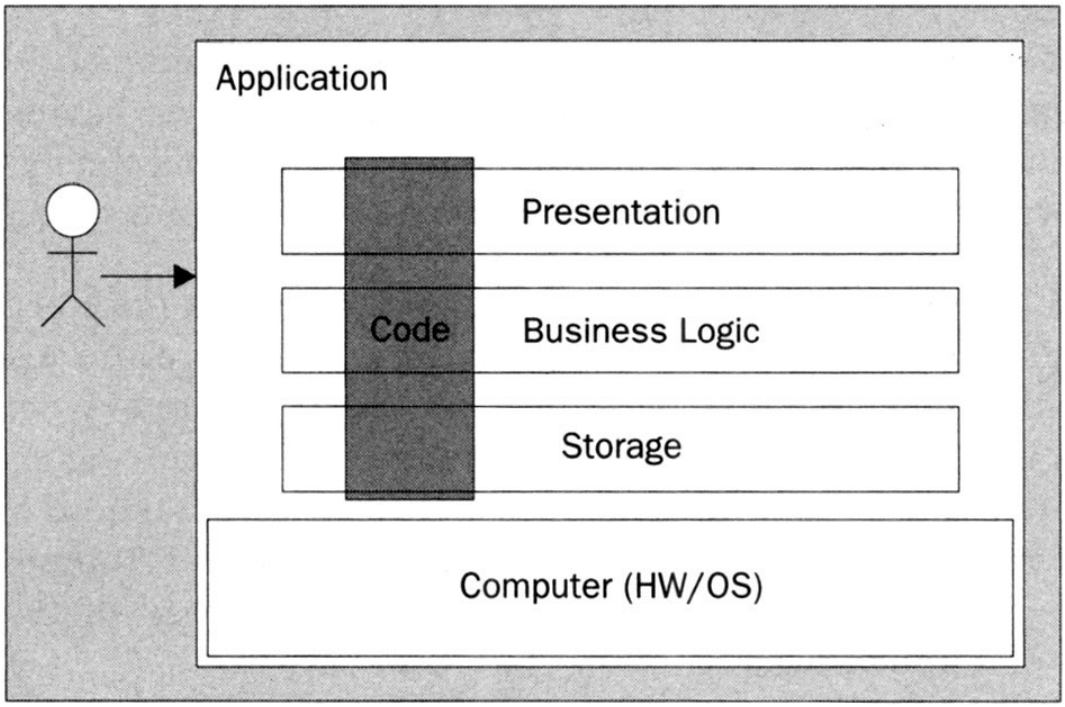
While EAI has filled this role to some extent, the ability to use the same technology within the enterprise that we use between enterprises once again offers a unique way to achieve simplification and reduce costs through the use of standards. Rather than using a proprietary solution within the enterprise and writing interfaces or intermediate programs to translate to web services, building the application bridge using web services ensures that services within the enterprise can be at any time migrated to services provided by the enterprise to it's partners. This type of flexibility is crucial to success in an environment where the rules of business change swiftly.

Technical Reasons for Web Services

In reality, the technical reasons are often at the heart of the business reason as the inability to do things technically often escalates into a business problem. There are many reasons for selecting web services; mentioned here are a few of the more prevalent ones.

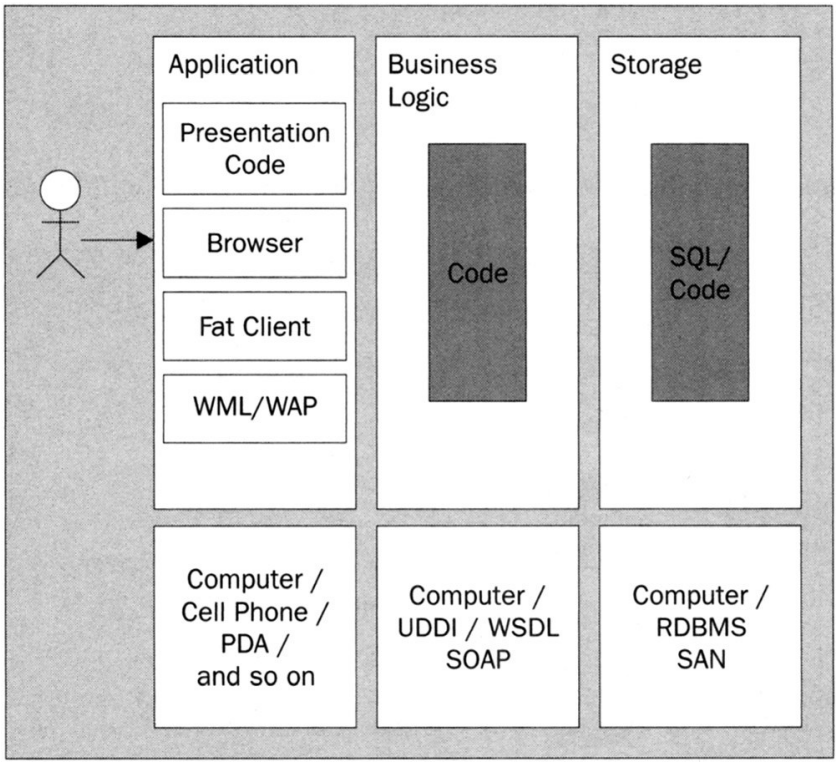
Service-Oriented Architecture

Code within an application contains a mixture of presentation logic, business logic, and storage code, running on a single machine. Client-sever technology moved the storage code to a centralized database server, but left the remaining architecture unchanged. In this model, adding a web interface, or adapting for a cell phone, or integrating into a messaging system requires massive changes to the application structure, even though the business logic could remain unchanged. The diagram below illustrates typical application architecture before the concept of **Service-Oriented Architecture (SOA):**



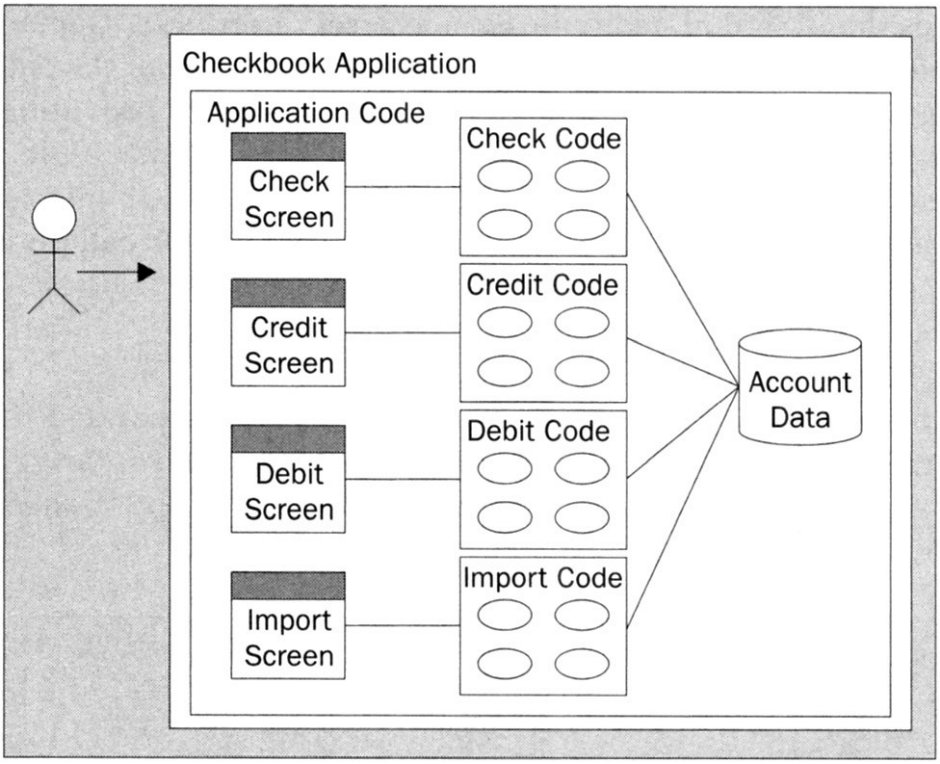
Basically, SOA implies the design of an application as a set of services. Demand for this design approach and technologies to support it have increased with the rise of the Internet. HTML made a poor application programming language, but was fairly adequate for providing a presentation layer. SOA divides the design of an application into services. A client can make use of a service to accomplish a task. This became a useful concept when people needed to add a web interface, an internal interface, interact with the financial system, and let users look at things on their cell phones. The service concept let developers design code that could be accessed in a number of ways (these ways or layers could, in fact, also be considered services in their own right).

In the figure below, the presentation of the interface to the user is separated from the business logic, which in turn is separated from the storage logic. Each partition can run on separate hardware platforms and perform discrete tasks. Adding a new system that makes use of the business logic requires no change in the code or the application structure:

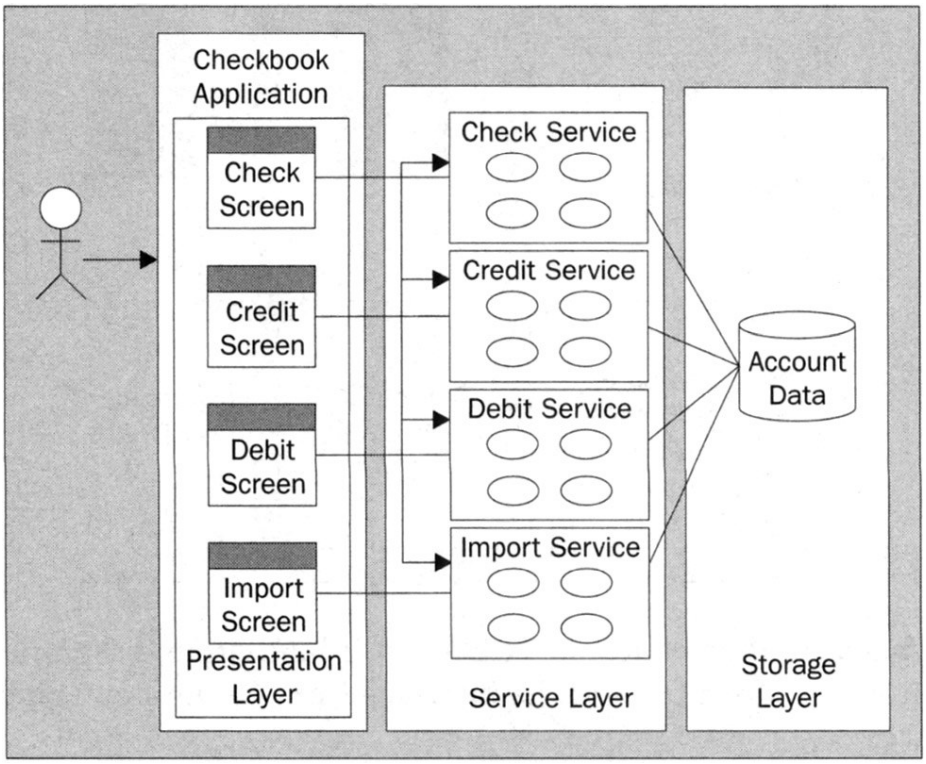


Web services fits perfectly into the concept of SOA - it is built around the principles of service-oriented architecture. It also provides abstraction of the interface from the implementation language and platform independence.

Let us look at our checkbook application again, but from a different view. We started by assuming that it was a single-user desktop application. Now let's suppose that we want to share it among other users (maybe we've started a small business and want to try to use it in multi-user mode to track orders). As it was written for a single user, with no concept of services, we would have a lot of difficulty modifying it to work for multiple users:



Now, if the checkbook application were written as an SOA design, it would look something like this:



Now we can treat the application as the presentation code, or any other code that makes use of the service. We might have other applications such as direct banking, that do not need a user interface at all, but make use of the services. Also, notice that we can define the Import Service simply by utilizing the other services - there is no need to duplicate the code of those services. This illustrates that the service layer may call into itself and that there may be a chain of services that fulfill a particular business need.

With this type of design we can create other clients and distribute them as needed - as HTML pages, applications, wireless apps, and so on. The SOA design provides tremendous flexibility.

Platform Independence

A related technical issue is that of **platform independence**. In an enterprise, it is rare to have one single platform, hardware or software. More commonly, there is a mix of hardware and software from multiple vendors. One goal of platform independence is that programs may run anywhere, but a related and more important objective is to acquire ability to ignore the platform a particular service or application runs on. In reality, true portability of applications isn't nearly as important as this ability. Most applications, especially custom applications, will run on a particular platform. It is important that the platform does not become a limitation to the accessibility of the application by other platforms.

Web services provide platform independence. The main protocols of web services are ubiquitous–XML is platform neutral and HTTP, which provides the most common transport mechanism, is available on almost every platform. By designing applications to be web-services enabled, we can achieve the power of platform independence and allow clients to access our business services in whatever manner necessary, from any platform needed.

Consolidation and Abstraction

One of the issues in IT support is the cost of maintenance. As the number of systems and applications grows, these costs can skyrocket. A common approach to solving this problem is **consolidation** of systems. The idea is that multiple systems perform some of the same tasks. Thus by removing some of the systems we can have the same functionality with fewer moving parts and reduce costs by reducing hardware, licensing, or personnel. Mergers and acquisitions often result in this type of environment with different systems performing the same tasks. For example, one company may use Peoplesoft, the other SAP. When the companies are merged, both systems may need to be retained for some time.

One of the approaches to consolidation is **abstraction**. Abstraction is a classic pattern in computer software, providing an API layer between the software being abstracted and the software that uses the services. By abstraction, changes in the implementation behind the API are not necessarily exposed to the client, making it easier to do design modifications.

In the case of duplicate systems, creating an abstraction allows for migration of functionality from one system to another at an ordered pace. Once the abstraction is in place, clients call an interface that provides the services. Where the service is executed is not meaningful–as long as the service is provided the client is satisfied. This way, IT staff can begin the task of reducing duplication. Once the API is in use, portions of one system can be migrated to the other and eventually one of the duplicate systems can be retired. No user needs to know when this happens, because the services do not change.

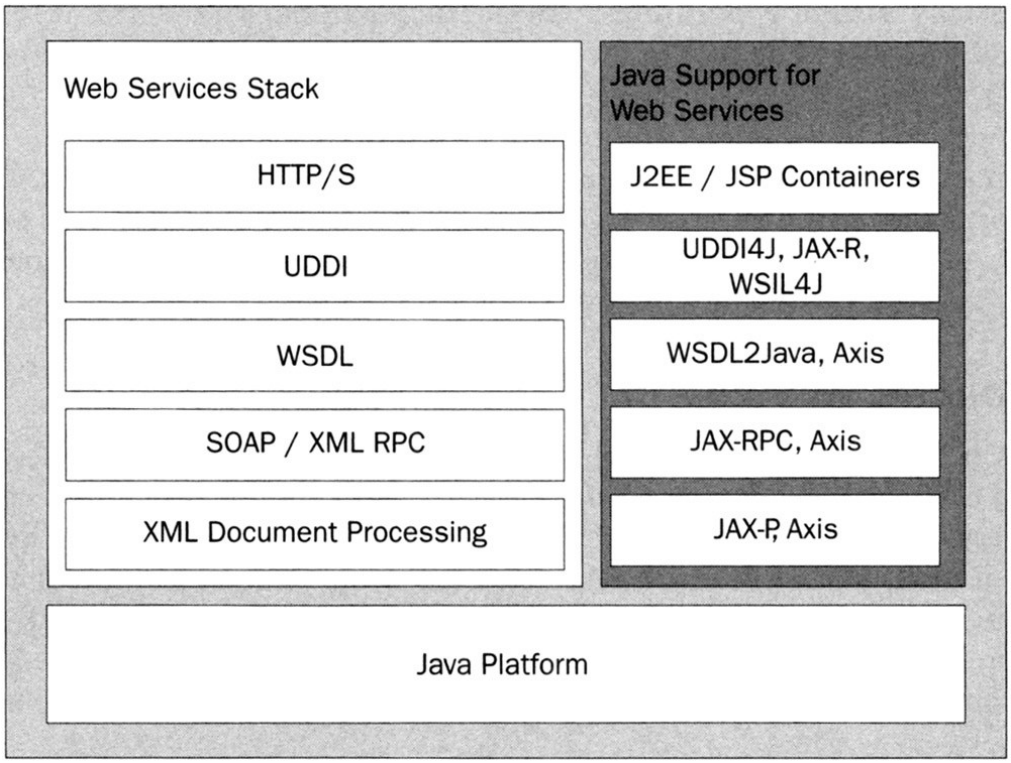
Obviously, web services fits quite nicely into this arena as well. UDDI and WSDL provide the building blocks for creating an API to abstract the systems in question and the routing information in WDSL allows services to be redirected as needed to allow parts of a system to be turned off

without disruption of service.

Java and the Web Services Stack

Until this point, we have discussed Java only in passing. The first part of the chapter was intended to provide a feel for the concepts of web services, which are not confined to any single language or platform.

At this point, a couple of valid questions might be: Why Java at all? What need is there for Java as a platform, or as a programming language? The answer is that, as with all services and applications the actual implementations still need to be written in a programming language and run on a platform. In this book we will be using the Java platform, with Java as the development language, along with HTML (via JSP) and XML. For the sake of this discussion we will be referencing the Java SDK version 1.4. Let's look at the diagram below:



The diagram above illustrates the web services stack and some of the key Java features that support it. At the bottom of the stack is the Java Development Kit (JDK) that contains the Java Runtime Environment (JRE) and represents the basic Java platform, the virtual machine. There are many ways that Java APIs can support web services; this diagram shows the web services stack and its relationship to Java APIs that affect each layer, rather than how web services are implemented using Java technologies. What this means is that this book is about using Java technology with web services, rather than building things like a UDDI registry in Java. The following sections explain the previous diagram in detail.

Standard Transport Mechanism

The standard protocol for web services is the **Hypertext Transfer Protocol (HTTP)** and its encrypted variant, **HTTP/S**. The HTTP protocol is a simple request/response paradigm and is synchronous. This means that any call to a service waits until the service responds. Sometimes this is referred to as blocking on a call. If the service crashes or takes an inordinate amount of time to respond, the call may time out and the caller would then consider the invocation as a failure. As with an ordinary web page, a web service is invoked using a **Universal Resource Locator (URL)** that allows the service to be contacted. HTTP allows for invocation of URLs and ensures that the request and response are routed correctly.

The HTTP/S protocol provides an encrypted channel between the HTTP server (the web server or whatever software is providing HTTP support) and the client (your browser, application, and so on). Typically, this is done when using a username and password to authenticate the user of the service. Chapter 9 will examine security issues in greater detail.

Third-party providers offer elements that enable Java to support HTTPS. J2EE (Java 2 Platform, Enterprise Edition) servers provide an environment that includes HTTP/S as part of their implementation of the Java servlet container. Standalone JSP containers also provide the necessary functionality.

Asynchronicity and Other Protocols

HTTP and HTTP/S are not the only mechanisms possible for communication and the request/response model is not sufficient for everyone's needs. Some computing tasks may take a significant amount of time – they may be calculation intensive, or may need human input via some

workflow mechanism. For example, first time shoppers may have to receive human approval if their initial order is over a specified dollar figure. So not all web service requests can necessarily be completed in a synchronous fashion; we also need asynchronicity.

Asynchronicity is the ability for a client to communicate with a service in a fashion outside the publish/subscribe, request/response mechanism. It is important because many business transactions require multiple conversations to record a business transaction. Not all of these conversations can be accounted for in synchronous invocations.

Java provides several different mechanisms for asynchronous communication. In the J2EE specification, the Java Messaging Service (JMS) provides facilities for a number of paradigms, including point-to-point and multicast messaging. The Java API for XML Messaging (JAX-M) provides an asynchronous service for web services and implements a pure Java version of the SOAP protocol. These technologies will be covered in detail in Chapter 8.

Discovery via UDDI

UDDI allows for the registration of an entity as a service provider as well as the ability to register the services the company is willing to provide. Although a global UDDI registry is often discussed, many developers predict the rise of industry-specific, private or semi-private registries rather than one global UDDI registry similar to the DNS service for networking.

Java provides the Java API for XML Registries (JAXR) to allow developers to access a UDDI registry, query its contents, and make entries. Querying the UDDI registry can be done via URL access as well; it is not limited to API access.

The use of JAXR to access the UDDI registry will be seen in detail in Chapter 7, where we will also discuss several utilities (UDDI4J and WSIL4J) that allow for quick conversions between UDDI XML and Java objects/classes. We will also discuss the Axis tool as an implementation of XML RPC and related APIs. It is an implementation of the SOAP submission to W3C by the Apache organization (http://xml.apache.org/axis).

Description via WSDL

WSDL is the language used for specifying detailed information about a service, which allows an application to understand how to invoke the service and what parameters are required. An actual WSDL document is complex and includes elements such as imports, types, schemas, messages, ports, bindings, and services. The details of WSDL will be covered in Chapter 5.

Java supports WSDL via several tools. The Java2WSDL tool, which is part of the IBM web services Toolkit allows for the generation of WSDL from a Java class, while the WSDL2Java tool, also a part of the toolkit, allows developers to generate Java class stubs from a WSDL document.

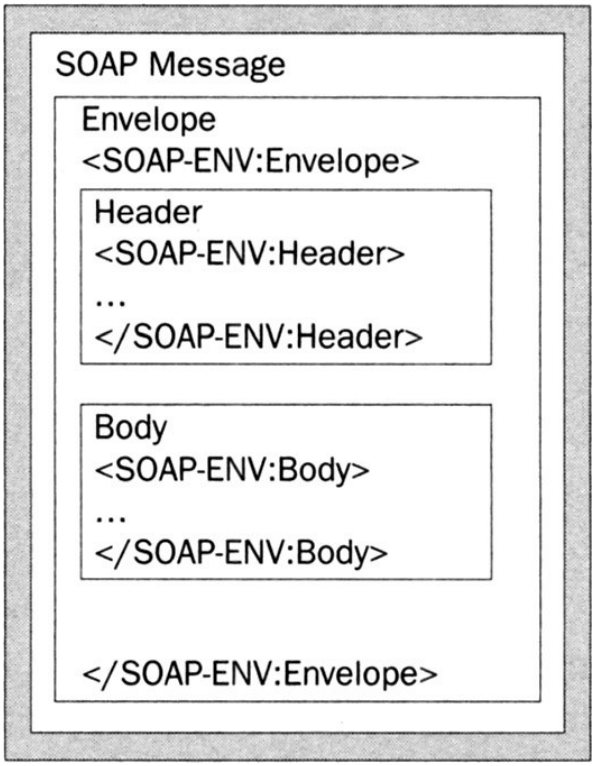
An example of a WSDL document taken from the `StockQuote` example in Chapter 5 is shown below. We can see that a number of messages are defined, particularly one for a `getQuoteRequest` and one for a `getQuoteResponse`, which allows us to request a stock quote and receive a reply. Note that the messages start with `<wsdl :message …>`. You can also see the definition of the operation, named `GetQuote`, and its relationship to the two messages. It's envisioned that UML modeling tools will be able to generate WSDL documents such as that shown below. The WSDL document is as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<wsdl:definitions targetNamespace="http://stockquote.iws.wrox.com"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:impl="http://stockquote.iws.wrox.com-impl"
  xmlns:intf="http://stockquote.iws.wrox.com"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <wsdl:message name="Exception" />
  <wsdl:message name="getQuoteRequest">
    <wsdl:part name="in0" type="SOAP-ENC:string" />
  </wsdl:message>

  <wsdl:message name="getQuoteResponse">
    <wsdl:part name="return" type="SOAP-ENC:string" />
  </wsdl:message>

  <wsdl:porType name="StockQuote">
    <wsdl:operation name="getQuote" parameterOrder="in0">
      <wsdl:input message="intf:getQuotequest" />
      <wsdl:output message="intf:getQuoteResponse" />
      <wsdl:fault message="intf:Exception" name="Exception" />
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="StockQuoteSoapBinding" type="intf:StockQuote">
    <wsdlsoap:binding style="rpc"
```

```
        transport="http://schemas.xmlsoap.org/soap/http">
      <wsdl:operation name="getQuote">
        <wsdlsoap:operation soapAction=" " />
        <wsdl:input>
          <wsdlsoap:body
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            namespace="http://stockquote.iws.wrox.com" use="encoded" />
        </wsdl:input>
        <wsdl:output>
          <wsdlsoap:body
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            namespace="http://stockquote.iws.wrox.com" use="encoded" />
        </wsdl:output>
        </wsdl:operation>
      </wsdl:binding>

    <wsdl:service name="StockQuoteService">
      <wsdl:port binding="intf:StockQuoteSoapBinding" name="StockQuote">
        <wsdlsoap:address
          location="http://localhost/axis/services/StockQuote"/>
      </wsdl:port>
    </wsdl:service>
  </wsdl:definitions>
```

Invocation via SOAP

Simple Object Access Protocol or SOAP is one of the many binding protocols available to WSDL, but it is the standard and most commonly used one for web services. SOAP allows a web service to be invoked by a client application irrespective of the language, in which the client and providing application are implemented. Java provides the Java API for XML RPC (JAX-RPC) to allow developers to execute Remote Procedure Calls (RPC). JAX-RPC implements a pure Java version of the SOAP protocol. The use of JAX-RPC and SOAP via the use of the Axis toolkit is illustrated in Chapter 3. The following diagram shows the representation of a SOAP message:



This diagram shows the anatomy of a SOAP message. It includes the envelope, header, and body. SOAP may use HTTP to accomplish communication between a client and a service, which must provide a SOAP processor to decode the message. Common web servers such as IIS and Apache provide such processors.

The envelope marks the beginning and end of a SOAP message. The header is optional, and can be used to describe attributes of the message. The body is required and contains the actual message, which may be one or more blocks of information.

As a preview of things to come, here is a snippet of a SOAP message that we will look at in detail in Chapter 4. You can see the start of the

envelope, with the tag `<SOAP-ENV:Envelope …>`. It defines the schema and the namespaces to be used. Then within the envelope, we see the body, which begins with the `<SOAP-ENV:Body>` tag. Within the body is a response from the `HelloWorld` service to a call on the `SayHello` method:

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Body>
    <ns1:sayHelloResponse
      SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:ns1="urn:helloworld">
      <sayHelloReturn xsi:type="xsd:string">Hello Reader!</sayHelloReturn>
    </ns1:sayHelloResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Data/Document Processing Using XML

Java provides the Java API for XML Processing (JAXP) to facilitate the processing of an XML document. It includes mechanisms for parsing and manipulating XML documents such as the DOM and SAX representations of an XML document, as well as an implementation of XSLT. Chapter 4 will expand on the examples from Chapter 3 and will demonstrate document processing using JAXP and the Axis toolkit.

While we are discussing Java and XML we need to discuss two topics – DOM and SAX. DOM stands for Document Object Model and is one way for Java developers to access and influence XML documents. SAX stands for the Simple API for XML, and also provides an API for XML access via Java. To a certain extent, the choice of DOM or SAX is left to the programmer, but it is also part of the design specification for a particular service.

Summary

In this chapter we discussed some of the reasons that led to the development of web services. We looked at the shortcomings in a number of programming models related to distributed applications and examined how web services provided an excellent way to support both internal and external abstraction of systems. We discussed XML, SOAP, UDDI, and WSDL as they relate to the creation and publishing of web services.

We also discussed the infrastructure of web services, and covered how XML, SOAP, and RPC reflect its reality.

From a Java perspective we also included a variety of documents. We examined how web services relate to Java and what Java APIs are available. These included JAX-M, JAX-R, JAX-P, and JAX-RPC. We briefly outlined these relationships and mentioned the chapters in which they will be covered in detail. Let us now proceed with the rest of this book for a detailed description of web services.