Architecting Web Services

WILLIAM L. OELLERMANN, JR.

Apress™

Architecting Web Services
Copyright ©2001 by William L. Oellermann, Jr.

ISBN (pbk): 1-893115-58-5

Printed and bound in the United States of America 12345678910

Editorial Directors: Dan Appleman, Gary Cornell, Karen Watterson, Jason Gilmore
Technical Reviewers: Thomas Lewis, David McKinstry, Roger Guess III
Managing Editor: Grace Wong
Marketing Manager: Stephanie Rodriguez
Developmental Editor and Project Manager: Kari Brooks
Copy Editors: Tom Gillen, Beverly McGuire
Page Composition: Impressions Book and Journal Services, Inc.
Artist: Tony Jonick
Indexer: Rebecca Plunkett
Cover Design: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010
and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany

In the United States, phone 1-800-SPRINGER; `orders@springer-ny.com`; `http://www.springer-ny.com`
Outside the United States, contact `orders@springer.de`; `http://www.springer.de`; fax +49 6221 345229

For information on translations, please contact Apress directly at 901 Grayson Street, Suite 204, Berkeley, CA, 94710
Phone: 510-549-5938; Fax: 510-549-5939; `info@apress.com`; `http://www.apress.com`

CHAPTER 2

Web Services Architecture

SINCE WEB SERVICES ARE such a shift in how we approach distributed development, understanding how to model such systems is an important step in understanding how to properly design, develop, and apply Web services. Because these services can be shared across the Internet, the numbers of ways in which they can be modeled make identifying all the possibilities a challenge. For every Web services model one could conceive, extending it one level further through another partnership could lead to yet another model.

The challenge in laying out our architecture is in being able to accommodate all of the scenarios in which they may be used. Our architecture should provide a consistent method for designing Web services independent of the number of participants, the participants' level of involvement, and the technologies and platforms chosen. Let's start by identifying some of the main application scenarios for utilizing Web services.

Web Services Partnership Scenarios

Think of an application utilizing Web services as a chain of links, with each link representing a function in the application (see Figure 2-1). Any links between different partners would represent processes shared via Web services. The connection between those links would then represent the communication between a service and its consumer. That means that a Web service can also be a consumer of another Web service. The first link in any such chain is the only one not eligible to be a Web service. It represents the user interface for the entire application. The partner owning the first link is the "originator" of the application and is responsible for the entire user experience and thus owns the application.

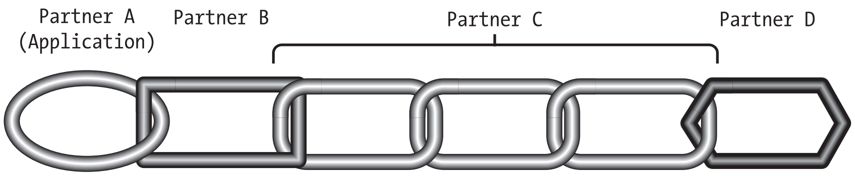

Figure 2-1. The partners in a Web services-based application

Just like a chain depends on each link, the entire application depends on each of its contributing links. This dependency is something that can be shielded from the end user to some extent, but inherently, whatever functionality the link is providing is absent if the link is inoperable. This is no different from any other application that is unable to access functionality from one of its logic or data components.

Figure 2-1 shows four participants in the partnership "chain." Let's say the chain represents the mobile application accessed in our hypothetical traffic scenario from Chapter 1. Partner A is the mobile application owner (perhaps the cellular service provider), partner B is the airline, partner C the hotel, and partner D a third-party vendor providing a calendaring service. In this particular partnership scenario, the application partner knows it is exposing some set of functionality, but does not necessarily know whether it is coming all from one vendor or from multiple vendors.

If each link represents a step (a term that here loosely indicates value), some partners contribute more to the overall application than others. Partners A, B, and D all contribute one link, but partner C contributes three links. Even though they provide more functionality, that doesn't necessarily mean that partner C has the dominant position in the business relationship, since partners A and B are necessary for partner C to be accessed by a user. Partner C may have chosen to focus only on the logic of the functionality and forgo the investment in designing a presentation for end users. However, that means in another application partner C might be directly utilized by the application owner (see Figure 2-2).

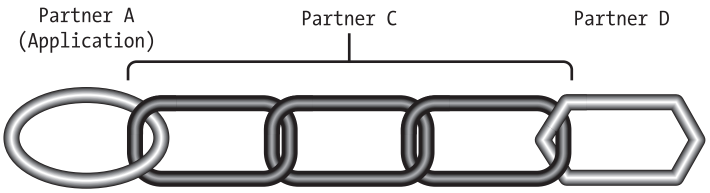

Figure 2-2. The shifting of partners in a Web services-based application

Here you can see that partner C is connected directly to partner A. Perhaps the mobile application wanted to provide just a hotel reservation system, and so the airline's functionality wasn't needed for this application. In another application scenario, partner A might have realized that partner B wasn't providing any value and was only acting as a reseller of partner C's service. Bypassing partner B might not only save costs for the application, but also provide better performance, since a hop is eliminated.

A *hop* represents a system on the network as the data is transferred between the caller and the caller's destination. A smaller number of hops implies faster throughput given a consistent transfer rate between hops.

When partner C originates the application (see Figure 2-3), partner C must provide functionality over and above what it provided for the other applications. This is because partner C is now responsible for presenting the application to the end user. Previously, partner C was concerned only with providing the functionality for its piece of the process, not delivering the entire application. This scenario is like the hotel company providing the mobile application for the user stuck in traffic. The merits of doing this are obviously up for debate, but the idea is that the option exists.

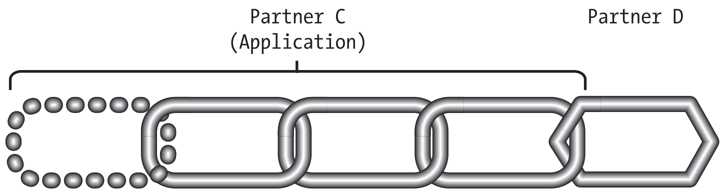

Figure 2-3. Partner C as the application owner

One more high-level scenario that needs mentioning is the owner/broker model. In this kind of partnership, the originator connects to each partner in the process directly and essentially brokers the services into a single application (see Figure 2-4).

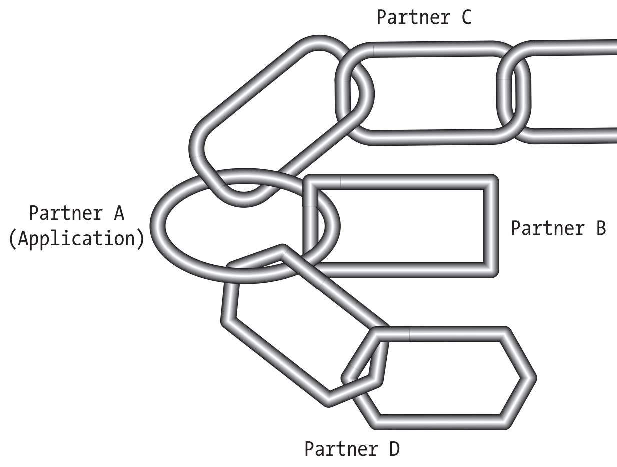

Figure 2-4. Partners in a brokered Web services application

In this scenario, the mobile application provider may be expanding its reservation application to utilize several major hotel chains and their reservation systems. There is less dependency between the partners, since there is no stacking of functionality. This arrangement is appropriate, since each partner is a competitor with the others and would have no incentive to support the others. However, the owner is still just as dependent on each partner for its functionality.

NOTE *This may seem to be the ideal scenario for a Web service, but it may not be possible. You may have some partners that only provide partial service, but are partnering with others to provide a more robust service. In other instances, there may be core functionality that a partner needs, and the partner may be using a Web service partner for that service. A good example of this would be a financial authorization process. A broker may be using a shipping partner, and that shipping partner may use another service to authorize a credit card or account.*

Of course, some hybridization of these models is also possible (see Figure 2-5). Some partners further down the application chain may act as brokers, and some partners may have entire application chains behind their service offering. It doesn't really matter to the partners, since the end result is the same regardless of the model.

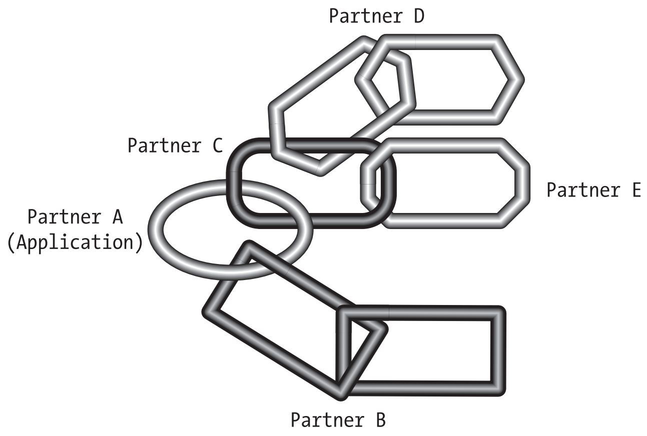

Figure 2-5. Partners in a hybrid Web services application

This idea is very similar to other complex partnerships that exist between businesses today. For example, take a hardware vendor that manufactures computers. While the hardware vendor may assemble the system, the vendor doesn't make every item in it. The vendor has a partner that provides the video card, for instance. If you take this example one level further, the video card manufacturer probably does not make every item on its cards. It may have a partner that provides the memory for its components. You could probably take this example even one level further by identifying the suppliers the memory manufacturer uses to build the memory chips. If you extend this out to every component in a computer, you can see just how many partnerships are involved in the end product.

As you can see, Web service applications can get very complex, depending on how many partners integrate their services. Hopefully, though, you are starting to realize that you could expose Web services to a large pool of users, as well as mix and match services from other providers to build a variety of applications. To aid in this reusability, there are certain rules that have to be followed so that developers on any platform can easily integrate your Web service into their application. Once you have this established, you can then scale this out by integrating additional services just like you would any other type of native object. We will look at specific application scenarios later that will further illustrate this concept.

Scale refers to the ability for an application or process to grow its capacity to meet demand. To scale an application out simply means to increase its load capacity.

Because a Web service is a little different from the applications we are accustomed to, our architecture analysis begins by looking at the communication architecture: that is, the architecture that defines the communication, or transport mechanism, between Web services and their consumers. It is important to understand how consumers communicate with Web services before looking closely at the internal workings, because this communication is what really differentiates Web services from other exposed processes. Once you understand how the externals work, you will have a much better appreciation for the issues that have to be addressed internally.

Then we will look directly at the individual layers of a Web service application, independent of the communication mechanism. This helps to provide a big picture view of the process of using Web services and a breakdown of the *n*-tier view.

Communication Architecture

In the communication architecture, we define the service and consumer interfaces as well as the transport mechanism between these interfaces. In a macro view, you can think of the communication architecture as the protocol that the logical architecture can utilize. It defines the mechanism for two partners to communicate via a Web service—the intersection of two links in our chain analogy, as shown in Figure 2-6. This level is independent of where components reside, how many layers of participants are involved, and all the details of any specific implementation.

The term *logical* refers to the conceptual view of a system versus the physical view. A logical view of an *n*-tier application would have a distinct separation between the presentation, business, and data layers even if they are physically contained on a single server.

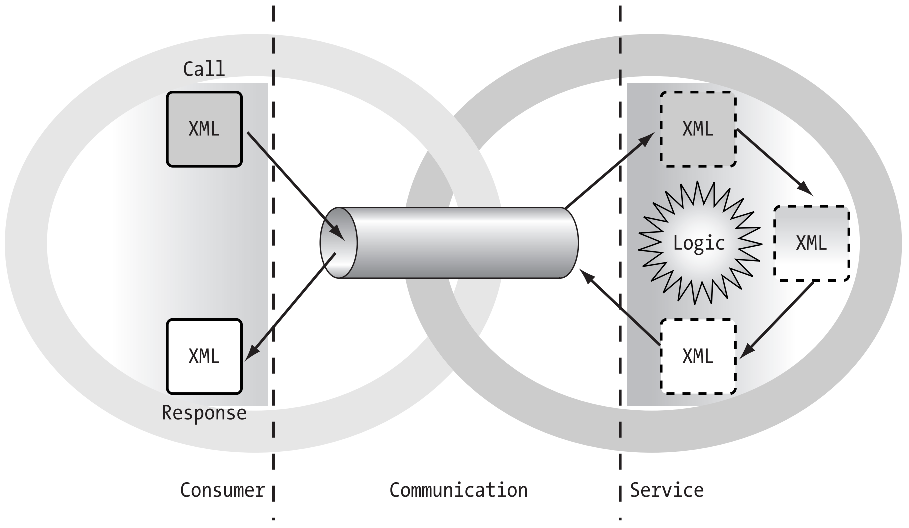

Figure 2-6. The communication architecture for a Web services call

The process is broken into three main components: the consumer, the communication, and the service. The consumer defines the entity utilizing the Web service, the communication defines how the consumer is interacting with the service, and the service defines the provider of the Web service. Each of these components is crucial for Web service execution. Furthermore, each of these three components must be implemented properly to qualify it as a Web service.

Remember that our definition of a Web service is any process that can be integrated into external systems through valid XML documents over Internet Protocols.

While this definition does not mention many specifics beyond the transport, it does demand enough structure to limit our options and focus industry efforts for building Web services. To confirm this, let's look again at our communication architecture, broken up into subcomponents with some of the technologies at our disposal for implementation (see Table 2-1).

Table 2-1. Web Services Communication Technology Options

CONSUMER		TRANSPORT		SERVICE	
REQUESTOR	***PARSER***	***PROTOCOL***	***PAYLOAD***	***LISTENER***	***RESPONDER***
Any logical process or entity that can initiate TCP/IP or UDP requests	DOM	TCP/IP	XML	ASP/JSP/ Java Servlets	Any logical process or entity
	SAX	UDP		Any component that can receive TCP/IP or UDP requests	

While you could certainly justify a different arbitrary breakdown, this one works out nicely with each component containing two subcomponents representing the key functionality of each. Each subcomponent's column lists some options for implementing that functionality. Note that all of the technologies listed are readily available today through multiple tools on multiple platforms. Quite a few areas have unrestricted technology options available for implementation. While a number of tools are available to build this functionality, there will be more "out-of-the-box" choices in the future to help deploy Web services more quickly with more robust services. When we discuss the details of actual implementations, we'll confirm that Web services can be built and consumed with current technologies alone.

Now that we've gotten a high-level view, let's look at each of the components of this architecture in more detail.

The Transport Layer

We'll start by examining the transport mechanism because it is at the heart of making Web services possible. This mechanism facilitates the service interaction by getting the request to the Web service and returning the response to the consumer. Without this step, we would have a service of little value. The importance of standardizing the communication aspects of Web services can't be overstated. In fact, the industry advances in this area are what have produced the environment in which Web services can be conceived and utilized. Without consistent communication methods, every implementation with a new partner might require an entire infrastructure and set of services. This is counterproductive to the goal of interoperability, and specifically, interoperability over the Internet. Fortunately, that requirement of working via the Internet helps establish our baseline requirements, and all of our communication structure falls into place from there.

The transport can be broken up into two components: protocol and payload. The protocol defines how we actually communicate between the Web service and its clients. The payload is the data being transferred, and later we will look at how it is formatted and how we can work with it.

Transport Protocol

This discussion of the protocol for Web services refers to the Open Systems Interconnection (OSI) reference model, an industry-standard model developed by the International Organization for Standardization (ISO) for identifying the layers of a network application. The OSI reference model has seven layers:

- Application
- Presentation
- Session
- Transport
- Network
- Data link
- Physical

Just as they are listed, you can think of these layers as being stacked from the physical layer up to the application layer. Not all layers are required for a functional network application, but if they exist, they can interact with each other. The communication for a Web service application simply specifies two layers in the OSI reference model: network and transport. However, the option selected in the transport layer inherently limits your options in the application layer as well.

The network layer for all Web services is the Internet Protocol (IP), the established standard for all communication over the Internet. By specifying IP, we are limiting our options in the transport layer to TCP (Transmission Control Protocol) and UDP. The only real differences between these two protocols are their reliability and performance. TCP is connection oriented, and UDP is connectionless. This means that TCP tries to ensure delivery at a cost to overall performance, while UDP focuses on the highest possible performance. While it might seem that TCP is the obvious choice for applications, UDP is much more effective for high-bandwidth applications in which the quality doesn't have to be perfect, such as voice. Still, TCP is much more widely used and has many more commercial applications established for working with it and developing on it (hence the common industry reference to TCP/IP when discussing the network of the Internet).

CAUTION *While UDP does not guarantee delivery, many people expect it to have the same reliability as TCP/IP. Unfortunately this expectation can be deceiving, since applications built on UDP have a noticeable performance improvement. Do not be tempted into developing on UDP without understanding the repercussions and taking the time to build in extra error handling at the application level in case something goes wrong and your application doesn't work!*

Both TCP and UDP are stateless, high-efficiency, routable transport mechanisms that can be very effective when used correctly. Like many things, if you don't depend on them for more than what they are, they will work just fine. However, you can get more enhanced services through the application layer.

Your options in the application layer for Web services are predefined in the sense that they must be fully compliant with TCP/IP or UDP. These are often called the Internet application protocols and include, but are not limited to, HyperText Transfer Protocol (HTTP), secure HTTP (HTTPS), File Transfer Protocol (FTP), and Simple Mail Transfer Protocol (SMTP). All these applications represent port numbers over the IP layer. Port numbers then translate to a socket on the system that allows a client and server to establish a connection. Table 2-2 shows some of these port numbers.

Table 2-2. Some TCP/IP Application Ports

APPLICATION	PORT
FTP	21
SMTP	25
HTTP	80
HTTPS (Secure Sockets Layer, SSL)	443

The applications listed are for TCP/IP, because UDP applications are typically not named and instead are just represented by their port number. For example, we would call an FTP application "a Port 21 application." This would probably change if UDP became as widely used commercially as TCP/IP. Your choice of the transport and application layers comes down to your application requirements. There are options that allow you to perform a multitude of services: simulate remote procedure calls; have a high-performance media service; or send secure, encrypted data to a server. We will take a hands-on look at some of these choices later.

NOTE *Another alternative protocol for the Internet is Internet Control Message Protocol (ICMP), which is not discussed here because its use is reserved for TCP/IP to communicate connection errors to the system.*

Transport Payload

The data transferred during the communication of two systems is referred to as the *payload*. This is how we can send data to a Web service and receive responses. Again, the establishment of standards in this area was critical to enabling a consistent method for defining the data both syntactically and semantically. This means that we not only have a standard format to intelligently parse the data, but we know how to identify specific pieces to interpret or transform it. This allows us to work meaningfully with the data. Let's take a closer look at the enablers of our data component: format and definition.

Format—Speaking the Same Language

The importance of speaking the same language (that is, agreeing to a syntax) is critical to any kind of communication, including Web services. The easier it is to use the language, the quicker and more extensive adoption will be. The lack of a consistent language means that partners would spend more time translating.

If an English-speaking tourist travels to China, how long does it take that tourist to use a translation book to order a glass of water from a non-English-speaking waiter? The tourist has to look up the correct words and then try and pronounce the words correctly and use them in the correct context. While this may seem extreme for a computer application, it still shows how potentially costly and ineffective translating can be. It doesn't matter how fast you are at translating; it is far less productive than speaking the same language.

XML has provided the means for two or more systems to speak the same language. This flexible language describes data syntactically in an industry-accepted standard. While simple in concept, its importance and significance to Web services, and application integration in general, cannot be overstated. We will use XML in our Web services to format our payloads between the consumer and the service. By using a standard like XML, Web services can and will be able to take advantage of tools built for working with XML. Nearly every language, development environment, and service takes, or will take, advantage of XML in some way, and that translates very well to working with Web services.

Definition—Communicating the Same Meaning

Although we have a means for describing our payload, we also need a way of defining it. The semantics of our data will enable everyone "speaking our language" to understand our vocabulary. This means that others will not only be able to read our data, but also have the ability to know what the data is and work with it.

This issue is even evident in the same language. People in both the United States and England speak the same language, but does that mean they share the same understandings? In England, there is a mechanism for taking people vertically up a building's floors called a "lift." In the United States, *lift* is usually used as a verb and means to pick something up. Now, when someone from the United States hears *lift* for the very first time, does that person have any hope of understanding what the speaker means if the speaker is from England? Because they share enough commonality due to the fact they speak the same language, the person from England can either explain the concept in more common words, as I did earlier, or say the word *elevator*. The case is similar in data transfers when everyone talks the same language. It is always possible to "map" a new term to a previously understood term or terms. While our mind remembers to treat the words or phrases as synonyms in the future, an application has to make a record of the relationship in some data store for future reference.

NOTE *It may seem like this idea of mapping words or phrases is the same as translating languages, but they are different processes. You cannot map two words or phrases if they are not synonymous in the language. Before and after the mapping, the context is the same, the usage is consistent, and the meaning doesn't change.*

A standard in this area also enables us to have a more efficient means for communicating the details of our Web services to partners and consumers. Fortunately, we again have some mechanisms in place to make this easy: the Document Type Definitions (DTDs) and XML Schema Definitions (XSDs). A service owner would provide one or the other to communicate the definition of any payload involved in calls and responses. We will talk about these two standards in much greater detail in Chapter 3.

The Web Service

The service contains all the components that make up the actual Web service interface. These components reside on the service owner's infrastructure and can be implemented in a variety of ways. The decisions of platform, technology, tools, and services supporting the service are at the owner's discretion. Furthermore, these decisions can be changed without impacting any existing consumers because the change takes advantage of the Internet's stateless nature. What does it mean to be stateless?

State is the act of remembering what is going on between the interactions of systems, or "calls." That may sound oversimplified, but that is pretty much it. Whenever you have an application or process that manages state, it monitors its users to know if they are logged on, if they are in the middle of doing something, what they might have done yesterday, and so on. The Internet is inherently a stateless environment. That means that every time you make a request to a Web site, the application inherently has no idea if you have ever been there before. Of course there are Web applications on which you can create accounts and that remember what you are doing, because information is being maintained by those sites on the application layer.

CAUTION *Many HTTP servers come with standard and custom services for maintaining state, but there is usually a performance and/or scalability tradeoff. No method is foolproof, so be aware of the implications and risks for each and only use them when necessary.*

Any Web application maintaining any state is using some method to identify users (cookies, HTTP data, and so on) and then storing information about their activities (usually in a database) as their state. This state can then be referenced by the application during later requests by that user. This is how Web-based applications circumvent the stateless environment through which they are communicating. The browser certainly helps in the effort by managing cookies and keeping information in its cache. Any application that does this, though, has to gracefully handle any truncated processes, because the user at any moment can get disconnected or abandon a session, and the application will never be notified.

> A *session* is an instance of a single user working with an application in a defined time period.

NOTE *This is why all sessions on Web servers have a timeout property. Without it, the application would always be assuming the user is still connected and would continue maintaining old information.*

Two vital components are required for an application to function as a Web service: a listener and a responder. While these two components can be physically one entity, or distributed among numerous entities (see Figure 2-7), the functionality they provide is what is important for this discussion.

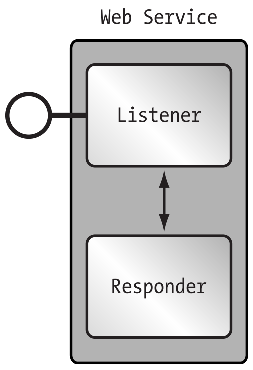

Figure 2-7. A functional view of a Web service

Listener: Waiting for the Call

The listener of a Web service is responsible for handling a Web services call from a consumer. The availability of this listener determines the availability of a Web service because it is essentially the interface. The process of handling a Web services call involves two steps.

The first step is capturing the incoming call. This capturing can manifest itself in many ways with a wide range of complexity. Possibilities for implementation range from having a script document hosted on a Web site to exposing an FTP directory.

The second step is the delivery of the payload package to the actual service logic. Although the listener could contain the actual logic of the Web service, this is not necessarily the best design. Separating the logic and the listener allows all listeners to be aggregated into one location or function, thus reducing the deployment time for each additional Web service. It also provides an extra layer on top of your logic, making it a more secure design.

You can think of this relationship between the listener and the service logic as being much the same as how a post office relates to the mailboxes on the street. The listener serves as the distribution center for the neighborhood (in our case, the service provider), and a process delivers the call to the actual address (in our case, the service).

Again, this functionality can be implemented through a wide range of technologies, but the key is that it is a persistent implementation. If you provide a Web service for your partners and customers, the expectation is that your service is as reliable as your Web site, if not more so. Thus, it is important that your actual implementation is persistent and efficient, and that all starts with your listener. We will look at some approaches for making your implementation persistent and efficient in actual code in Chapters 6 and 7.

Responder: Answering the Call

The responder of a Web service is responsible for providing a response to be returned to the calling consumer. It only provides a response because the listener owns the connection to the client. The response cannot be returned through another process because of the nature of the communication process. The consumer makes the request of the service and either waits for a response or expects no response from that service. Although only a UDP-based Web service could be truly asynchronous, you could simulate an asynchronous call via TCP/IP by providing a simple acknowledgement of the client call and releasing the connection. (This is essentially what happens inherently with SMTP.) You could then later send a response to the request at your convenience, not tying up your listener or the client (see Figure 2-8 for an example). This essentially turns a consumer into a service provider, since it requires a persistent listener on the consumer's side. This is a viable design, but these two steps should be considered two distinct Web services, since they are both acting as a client and a server.

Synchronous refers to the inline execution of a process or set of processes. No other processing can occur in an application during a synchronous call. *Asynchronous* refers to the execution of a process that does not restrict the application from other activity.

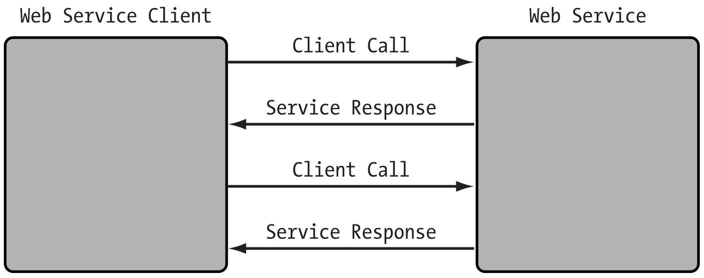

Figure 2-8. Asynchronous or delayed response Web service

This dual Web services solution places much higher demands on your consumer. The challenge of this approach is that you are turning a client-server relationship into a server-client relationship, which means that you are establishing a client-server connection to the client that originally called you. Typically, the client that called you would actually be a server, so it should be capable of supporting the relationship. But does it have an active listener waiting for this response, which is essentially a request? Does the consumer have the logic to handle and process this delayed response? You need to consider these challenges to determine if this is a practical implementation for your Web service.

NOTE *There is a misnomer about synchronous versus asynchronous calls that I will clarify for our discussion. There are very few asynchronous calls over the network. Client applications usually receive at least some notification that a packet has been delivered somewhere. What are thought to be asynchronous calls are actually asynchronous processes. This is accomplished by tying two synchronous calls together. The first one acts as a channel for the send process, the second as a channel for the significant response process. Again, we can relate this to mail delivery. If you were to send off for a rebate, is that synchronous or asynchronous? The sending of the rebate is synchronous because once you fill it out and place it in a mailbox, you know that it will be delivered. However, the entire process is asynchronous because you have no guarantee of getting the rebate.*

The response may or may not depend on the data payload submitted by the client. For instance, a Web service could simply respond with a confirmation that the call was received, or it may process the data and provide a custom response. When the response is simply a confirmation of receipt, the responder may be included in the listener component. But the more your response increases in complexity, the more beneficial it will be to separate these components. In fact, for enterprise-level Web services, you will likely have a responder that is logically separated from all of your Web service application logic to get the most reuse of that functionality.

Since the data in a response has to be encapsulated in a valid XML document, this component might have to build the response or at least transform it from another format. Then again, it could simply pass XML data straight through it. This starts to lead us to implementation designs, which we will discuss later.

Just like the listener, the responder platform and technology choices are independent of the consumer's. However, you may be tied to decisions made in this area for your listener. Assuming that performance is important for the service, you should stick with one set of technologies internally, since you have that luxury when there is single ownership.

The Web Service Consumer

Just as the name implies, the consumer is responsible for using the Web service and for initiating the interaction with the service, not vice versa. Every application or process that uses a Web service is considered a consumer. That means that for a chain of linked partners (as we saw in Figure 2-1), every partner except the last one (partner D) was a consumer. We will see in "Logical Architecture," later in this chapter, how this stacking of consumers can make our Web services extensible.

NOTE *In this book the word consumer refers to the direct caller of the Web service. This might also be called the client since this could be classified as a client-server architecture. However, neither of these names is used in this book to describe the end user. The end user is the person or persons possibly using the service through an application and is likely to be a customer, or at least a client of the actual consumer.*

With our definition of a consumer, we have recognized another fundamental aspect of Web services: namely, that Web services are built exclusively for programmatic access by other applications. Table 2-1, earlier in this chapter, reinforces this by leaving out any mention of a presentation tier. There is no need for a presentation tier internal to our Web services architecture because the consumer calling it has complete control of the end user experience. Even though the presentation layer itself has no place in the Web service architecture, a Web service can provide presentation information, as we will see in Chapter 5.

The consumer of a Web service probably has the widest range of possibilities of all the components when it comes to functionality. The consumer may be responsible for the presentation to the end user brokering multiple Web services, or simply passing through a Web services call. This becomes more obvious when you realize that even different consumers of the same Web service may want to use a Web service in entirely different ways. One consumer may pass information straight through to a client, and another may want use the Web service behind the scenes, completely masking its existence. We will look at some of these possibilities in more detail when we start looking at code in the sample applications. Right now we will only concern ourselves with the functionality that is necessary to make the call to the Web service and "handle" its response.

With today's toolsets and services, that means developing a custom application, or at least expanding the functionality of an existing application. Keep in mind that the great thing about consumers in this architecture is that they are truly independent of the Web service. That means that the options of how to design it, what technology is chosen, and even which platform to use are entirely open and unaffected by those same choices made by the Web service owners. The consumer is completely independent and could be completely unknown to the service, so the details of its implementation and execution are of no consequence to it.

The consumer will consist, by nature, of two different logical, if not physical, parts: the caller and the parser (see Figure 2-9). I stress the logical separation of these two functions, because, just like the service's functional components, they are independent of each other.

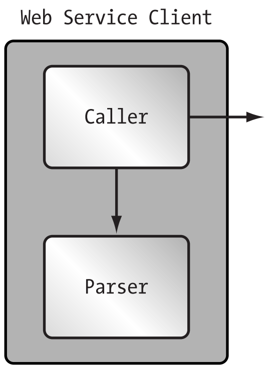

Figure 2-9. A functional view of a Web service client

Caller: Making the Request

The caller component is responsible for initiating the entire process of instantiating a Web service. It is responsible for building the payload for, and making the actual call to, the Web service. The nice thing about the caller is that it can be very generic in nature, so you can get a lot of reuse from a caller component. It can be so generic because of the communication structure we have defined. Regardless of what the consumer wants to do with a Web service, the communication with it is fairly consistent.

As we discussed with the service component, the call to the Web service is actually synchronous. Regardless, all the complexity of the caller lies in the intricacies and nuances of network communication. The steps involved are executed in the following order:

1. Build the payload.

2. Send the payload to the service.

3. Wait for the response from the service.

4. Pass the response to the parser.

More steps could be added to this process, but these are the minimal steps necessary to facilitate the Web service interaction by the consumer. We will look at this in more detail in Chapter 8 when we discuss consuming Web services.

Parser: Handling the Response

Since Web services respond to the consumer in an XML document format, most consumers have the ability to parse XML. This will almost always be necessary, but there are scenarios in which XML could be passed straight through, as we will see in the presentation models in Chapter 5. This functionality is so basic to Web service consumption that it is best isolated and accessed as a reusable object or component. This will keep you from having to incorporate that logic directly in each of your consumer implementations. While this could be done through custom logic, some standards have been defined to keep developers from having to design their own algorithms for parsing XML.

NOTE *Although there is technically a difference between the two, I will treat the words object and component the same for the purpose of this discussion and will use them interchangeably.*

One of those standards is the Document Object Model (DOM). This model allows the loading of an XML document into a tree structure, referencing each of the elements as nodes. This is a good method for parsing XML when you want to work with an entire document. An alternative to the DOM is the Simple API for XML (SAX) standard, which allows for easy access to a single node in an XML document. We will look at both of these access technologies in much more detail in Chapter 4.

Just like the Web service itself, the technology and platform choices made for the consumer are independent of any other entity. You will not be diminishing your capabilities to access true Web services by any choices you make.

CAUTION *I have mentioned several times how the technology choices made at each level have no impact on the other components in a Web service application. While these choices will not prevent you from using any Web service, there may be advances or more robust services in some vendor implementations of XML or Web services that may make the deployment and consumption of Web services easier. The issue to be aware of is a vendor's customization of Web services that actually makes the communication, service, or consumer proprietary. This may be a tradeoff you are willing to make, but you need to be aware of the decision you are making.*

The parser component of a Web service consumer can have the most variation on a per implementation basis because the utilization of a Web service has unlimited possibilities. This becomes more obvious when you realize that even different consumers of the same Web service may want to use it in entirely different ways. One consumer may pass information straight through to a client, and another may want to use the Web service behind the scenes, completely masking its existence. We will look at some of these possibilities in more detail when we start looking at code in the sample applications.

Now that we have a good understanding of how clients communicate with Web services, let's take a look at the actual makeup of a Web service.

Logical Architecture

The logical architecture identifies the functions to be performed by a system without consideration for physical systems or objects. In this case, we will look at the functional components of a Web service from the call on up. We will group these functions into layers to help us relate to current architectures in Web-based applications. This will also help us to later identify ownership and responsibilities in the Web service call. In general, a logical architecture is not intended to be strictly adhered to during implementation. However, it should be useful as a platform for discussion and analysis. Through this exercise, we will look at every component at its lowest possible level. Many implementations may leave out or group components for good reason, but we break them down to recognize the distinction of functionality.

NOTE *Layers and tiers are terms that I will use interchangeably. They both represent a logical grouping of services that may or may not also be physically grouped.*

In the functions themselves we will look at a couple of different aspects. First, we will look at the possible tasks that need to be accomplished at that level. This will help us to identify solutions and opportunities later in our design discussion. Second, we will identify the data that must be passed between the layers based on the functionality. This will help us develop a better understanding of the distribution of responsibility and again help us to identify opportunities during the design of our Web services. We will not look at the process for making a Web service call, because this subject was covered by our earlier look at the communication architecture. In this analysis we will just identify when the interaction between layers could or should be handled through a Web services call.

You can think of this exercise as similar to storyboarding an application. Just like that process, this architecture serves as a focus for discussion and ensures that we look at the same issues during design and implementation. Specifically, we look at where functionality goes and where it might interact with other functionality, but we do not concern ourselves with the implementation details or how it actually works.

> *Storyboarding* is the process of walking through the process or flow of an application and designing a graphical view of the interface that exposes all of the required functionality.

Web Applications

To establish a frame of reference, it is probably worthwhile to review the logical architecture for existing Web applications that are not Web services. This architecture has proven to be very robust. It has managed to grow with the evolving technology because it is a logical architecture. Various changes in tools, implementation, user base, and even physical distribution have not been able to break this model. In fact, when we look at the logical architecture for Web services, we notice that it is merely an extension to this very architecture. So understanding this model will, in essence, be the starting point for understanding the Web services model. It will also help us to distinguish the differences between Web applications and Web services in our designs.

CAUTION *It is common for less-experienced developers to confuse the logical architecture with the physical architecture in these discussions. The physical architecture is different because it designates objects with the appropriate functionality. The two n-tier models do not always match up, and, in fact, the two are more likely to not match in implementations. For example, data access is a responsibility of the data layer in the logical architecture, but it may make more sense for certain business layer objects to access the data model directly.*

This architecture contains three layers: data, business, and presentation. Each of these layers performs important functionality to achieve the end product of a distributed, Web-based application. As you see in Figure 2-10, the business layer serves as the go-between between the presentation and data layers, and clients can only connect to the presentation layer.

NOTE *The business layer is often called the logic or logical layer. The presentation layer is also sometimes called the interface layer. However, this usage will cause more confusion as integration with other applications and organizations becomes more prevalent.*

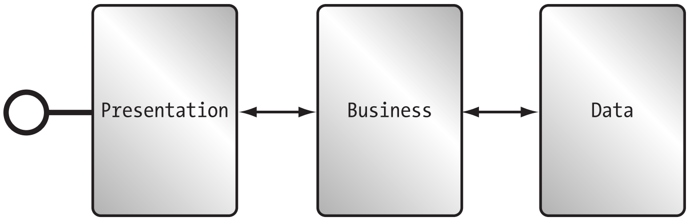

Figure 2-10. Logical architecture of an n*-tier Web application*

Although it may not be obvious, the presentation layer is responsible for initiating all processes in the application. More precisely, the client calling the presentation layer initiates the process. This means that nothing actually happens independently of client interaction in a true *n*-tier application. Any processes running automatically on the server fall outside of this model.

Also, all interaction in the tiers is synchronous because the initial call is synchronous. If the presentation layer does not get a response from the business layer, the client tier eventually times out, and all hope for fulfilling that request is gone.

Now that we have the high-level view, let's take a quick look at each layer.

NOTE *Notice that there is no inclusion of the client in this architecture, only the acknowledgement of an available interface. This is because no integration or shared ownership is possible in a traditional Web application. The application is designed as an independent entity, and that is one of the existing models that Web services is attempting to break.*

Data Layer

The data layer in any application contains the data source(s) and its components. The data source can take many forms, but the most common form is usually a relational database with tables, stored procedures, and triggers. The tier also contains the data access logic. This is the functionality for managing the connection(s) to the data source and making the actual queries.

This seems fairly simple, but whenever you start putting extended logic in stored procedures, the clarity of this layer starts to blur. By *extended logic* I mean logic that goes beyond the expected functions of data filtering or extraction of normalized data. For instance, if you have a stored procedure that extracts different data based on a parameter, runs some calculations, and includes some formatting, is it now part of the business layer? Is it in the object that knows which stored procedure to call or the stored procedure that has the intelligence built-in to look up a customer and all of its properties and activities? Although the benefit of doing this is debatable, we consider stored procedures fundamentally as part of the data tier, since ultimately stored procedures are really just tools for extracting the data.

The purpose of our data access objects is to isolate the data source from the business logic. This allows the application to not be "married" to the database implementation. While this certainly doesn't make changes in your database solution easy, it could make it possible to change the database without rewriting the entire application.

Business Layer

The business layer contains the application logic, a series of programmatic algorithms that make the necessary decisions to produce the appropriate output. Tasks in this layer can range from complex math calculations to simple data filtering. Because these tasks are so varied, always make sure you choose an appropriate solution for the functionality needed. These solutions can vary from a script in a packaged application server to custom code using just a programming language. Depending on the functionality, performance, and scalability required, there are generally several solutions available.

Since this layer often contains many business rules, you will likely want to be able to reuse the manifestation of this logic (that is, components or objects). The first step to accomplishing this is implementing a solution that is reusable in your current infrastructure. That is the idea of where compatible technologies and objects start to play a significant role.

To make the most of this strategy, you then want to break your logic down into the appropriate chunks. This means breaking your processes down as small as reasonably possible. These can then be aggregated into the necessary transactions and functions to execute your services and applications. The idea is to isolate low-level functions that are likely to be reused in completely different processes. I call this the lowest-common-denominator approach. If you consider applications as numbers and you need to produce the numbers 9, 12, 27, and 48, you can build functions that give you exactly 9, 12, 27, and 48, respectively. However, you will get more reuse in the future if you break these applications into smaller parts. In this case, you can build functions to produce just a 2 and a 3 and assemble them appropriately, since $3 \times 3 = 9$, $3 \times 2 \times 2 = 12$, $3 \times 3 \times 3 = 27$, and $3 \times 2 \times 2 \times 2 \times 2 = 48$. This also allows you to build other applications in the future, such as 6, 8, and so on. But if you never need a 2, or a 6 or 8 for that matter, you may not want to break it down that far. If you could create a function that produced a 4 more efficiently, wouldn't it make sense to do so? Probably. Since the lowest common denominators are 4 and 3, that is likely the best solution.

> A *transaction* is a unit of work that should either succeed or fail as a whole.

The client does not access the business layer directly. That is the responsibility of the presentation layer. To accommodate this, the business layer needs some type of hook(s) into it that can be accessed externally. This is often done through the interface of the objects in the business layer.

Presentation Layer

The presentation layer is responsible for the presentation of the application to the end user. This actually involves two functions, the display of information and the collection of information. Any logic necessary to complete these functions is considered part of the presentation layer, not the business layer. This logic includes validation, transformations, layout, format, and style.

Validation is the process of ensuring that the end user enters appropriate and acceptable data. The presentation layer has to ensure that the data collected is acceptable because the data typing for the data is much stricter in the business and data tiers. For example, if a user enters an alpha character for a number, the database raises an error because it obviously isn't allowed. Although you could gracefully handle the error and return it to the user, why waste the cycles traveling back through the tiers? If you catch the error up front, you can impact fewer layers and have the user resolve the error much more quickly. In fact, as a rule, you want to handle as many of these issues as close to the user as possible to avoid taxing the entire system.

Transformation is very similar to validation in that the data may be corrected intelligently without involving the user at all. A common example of this is stripping out certain characters from input data. If a user enters "888-555-1212", the presentation layer may strip out the "-" characters to keep the information as a pure numeric data type for more efficient storage.

Dictating the layout involves the placement of information and controls in the user interface. It takes a good UI to make a successful, intuitive client, and it all starts with the layout. It means knowing when to collect the information through an open text field or a predefined dropdown list. It also means knowing when enough is enough when it comes to collecting information in the individual steps of a process.

Formatting is the process of ensuring that data is presented in the appropriate manner. When you work with real numbers and want to dictate the specific digits displayed behind the decimal, you can often do this through formatting. Another example of this is the treatment of numbers as currency. By using a format routine, the presentation layer can take a number like "3214" and turn it into "$3,214".

Finally, styling is the process of adding window dressing to the information displayed in the presentation layer. Common examples of this would be font treatments and colors. There are a number of ways this can be accomplished, and we will touch on some of these later in our sample Web services.

To complete its objective, the presentation layer has to communicate with the business layer to pass and receive data. That means the designs and technologies chosen at each layer must be compatible with those of the other layers. The advances in technology to enable this integration between the presentation and business layers have enabled Web-based applications to become effective enterprise application solutions.

Web Services

Before we discuss the logical architecture for Web services, let us get a better understanding of why Web services don't have the same architecture as Web applications. While we discussed the difference between shared information and shared processes as well as the difference between Web applications and Web services, we need to understand how this distinction impacts the architecture.

Web services do not have a presentation layer. While Web services can contribute to the information defining the presentation, it does not, and in fact could not, dictate the actual presentation delivery to the end user. This goes back to the whole notion of shared ownership for the entire application. That means that proprietary technology and inconsistent implementations could be huge barriers to integration for either the service owner or, even more likely, the consumer. We have already established the communication architecture, which helps with masking the proprietary technologies involved. Now our logical architecture will provide a consistent breakdown that will identify the responsibilities in the tiers and who the owners of those tiers could and should be.

Although we are looking at the architecture for a Web service, it is an incomplete view of the entire application. Because a consumer is responsible for activating the application, it is important to look at both to (1) get a full understanding of Web services usage and (2) relate it to the architecture of existing Web applications, with which we are familiar. Trying to design a Web service without considering its consumers would be like designing a motor without considering the vehicle in which it is going. What will be the demands on the motor? Is it going to be used to carry heavy loads or just move as quickly as possible? Is it a small or large vehicle? Won't we build far better motors knowing the answers to these questions? Although it isn't quite that extreme, because we have established a baseline for all of our "motors" to work with vehicles via our communication architecture, we really need to analyze the Web service and its consumers so we better understand the big picture. So, let's take a look at the overall logical architecture for both (see Figure 2-11).

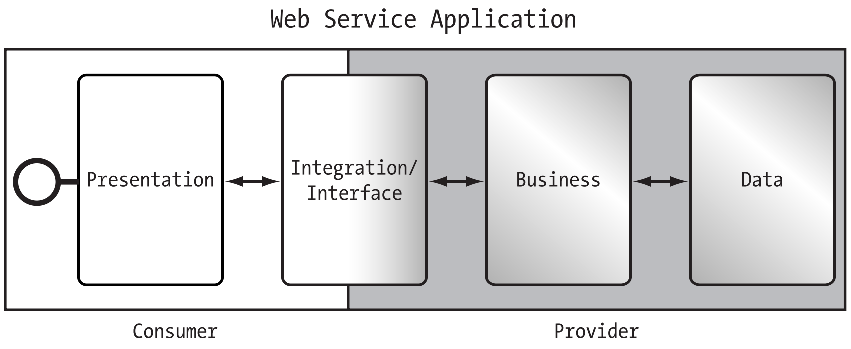

Figure 2-11. Web service and consumer logical architecture

In the logical architecture for Web services, we will continue to have the same layers of data, business, and presentation. However, we added a fourth layer, the integration/interface layer, between the presentation and business layers. In a Web service, the presentation layer resides on the consumer's infrastructure, and the business layer resides on the service owner's infrastructure. The integration/interface layer crosses the organizational boundaries and allows for the shared ownership of Web services. This layer bridges the provider and consumer by executing the communication architecture outlined earlier for Web services.

The Service

First, we will look in detail at the service itself. Even though it depends on a consumer to be accessed by an end user, it is a freestanding entity. If a Web service is built and available but not being accessed, is it still a Web service? Is a car still a car when nobody is driving it? Of course!

In the Web service logical architecture (see Figure 2-12), you see that we still have three tiers, but instead of the presentation tier exposing the application, we have the interface tier. Since the presentation layer is shared, the data and business logic exposed play a more significant role. If you take into account that the interface tier is simply a connection to the service, it becomes obvious that the real value differentiating different Web services is contained in those two layers. Let's take a closer look.

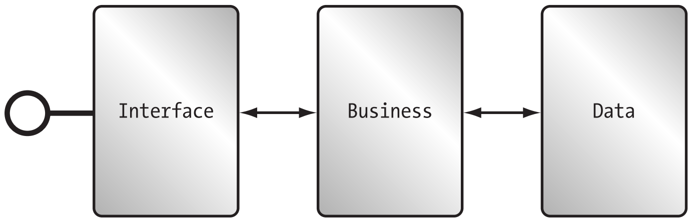

Figure 2-12. Web service logical architecture

Data Layer

Unlike with other enterprise applications, it might be inappropriate to label the data layer as the foundation. Depending on the functionality, the service may be passed every piece of data it needs from the consumer. For instance, if the service is a calculator, the functionality may be contained entirely in logic. However, it could be argued that the more valuable services still have some local set of data they are either referencing or depending on to provide their functionality. After all, any logic might be reverse engineered, and the value of the service could be compromised. If the service provider owns or has access to proprietary data, it is harder to replace.

The real impact on the data layer in a Web service application is the fact that, at some point, the data exposed is defined in XML. This doesn't require data to originate as XML, but that is an option. Multiple transformations of data from one type to another ultimately affect performance of the service. For instance, if you are building your Web service in COM objects and use ActiveX Data Objects (ADO) to extract data into a recordset, encapsulate the necessary data into a property bag to marshal across business objects, and then expose the results in XML, you've gone through three very significant data transformations. There are decisions you can make to avoid this, and we will look at these later.

> *Marshalling* is the process of transferring data between processes. Marshalling objects is more intensive than marshalling strings because the objects have to do more encoding and rebuilding.

Business Layer

The business layer defines the processes behind the service and contains all application-specific logic. This layer is likely to be the least affected by the fact the application is exposed as a Web service. In fact, you should closely scrutinize any changes you might make internally to an existing application's business layer to make it a Web service. This could easily add overhead to an application that is used through channels other than Web services.

When creating a Web service from scratch (meaning not only the Web service, but also the functionality it is exposing), there are decisions that can be made to support a Web service architecture going forward to optimize both performance and extensibility. These decisions revolve around standardizing the approaches to handling data and maintaining security. Again, since all information is collected and exposed as XML, it would be beneficial to keep consistency in your interfaces and transformations. Also, determining a security scheme to account for public programmatic access to your logic may keep you from having to reconfigure, or even worse redesign, your business layer later when limitations are encountered.

The business layer is also the tier of the Web service that could be a consumer of other Web services. This makes sense when you consider that the business layer is responsible for the logic and accessing various local data sources and/or objects. Web services could be utilized just like other objects, and this is the layer at which that integration occurs (see Figure 2-13).

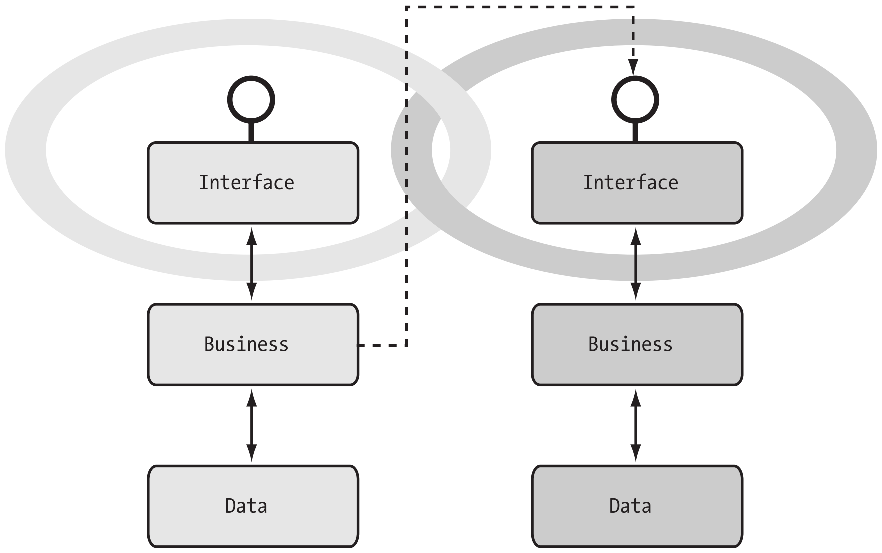

Figure 2-13. Web service integration with another Web service

NOTE *Don't interpret the preceding figure too literally without considering the components that are required in the business layer consuming the Web service. If the business layer is accessing a Web service, it must have the Web service consumer logical architecture, which we will discuss shortly, contained in it.*

This same model holds true for traditional applications**.** Regardless of whether the application is a Web service, a Web site, or even a fat-client application, an *n*-tier application references Web services from its business layer.

A *fat client* is commonly referred to as a client application that contains custom logic or processes to facilitate user-interaction with an application.

Interface Layer

This is where we get into the very interesting aspects of Web services. The interface layer is where partnerships are established. For this reason, this layer exposes the service's functionality to its consumers. Don't confuse this with the end user interface, since this interface is designed for programmatic access only. This layer works with the consumer's integration logic to provide the complete tier for our communication architecture.

The interface layer contains the functionality required to expose our Web service. We have to receive the call from the consumer, pass it to our application logic, receive the response from our logic, and return the appropriate information back to the consumer. In the communication architecture, we encapsulated this functionality in a listener and a responder. The listener handles the inbound processes, and the responder the outbound. Both of these components reside in the interface layer.

Although the Web service is not responsible for the presentation layer, it may need to provide information to the owner of the presentation to expose processes directly to the end user. This may involve the filtering and transformation of information or may involve entirely separate processes for extracting data specific to this function. Regardless, any functionality specific to this information is encapsulated in the interface layer.

This would be considered a value-added service to the Web service itself, just as providing a user manual on how to change the oil to the car manufacturer buying our motor would be considered value added. Can you build a motor without providing this information? Probably. The car manufacturer is certainly familiar enough with the process itself they could write the instructions for changing the oil. However, providing the manual makes using the motor easier, and more car manufacturers would be inclined to use your motor because of this value-added service. It also ensures that the car manufacturer is telling the user exactly how you want the oil to be changed. It helps ensure that users aren't abusing your motor (thus making users happier with your product) because they are getting the information straight from the source. Will the car manufacturer use the information? Probably, because of the savings in time and money of reusing our work. Will the manufacturer pass it on directly to the car owner or compile all information about the vehicle into one all-encompassing manual? That depends, but the point is that it is at the car manufacturer's discretion. This same concept can be applied to Web services and the services you provide around it.

Other services may benefit the Web service provider more than its consumers. We will talk about these and other value-added services you can provide later. The main thing to remember for now is that these services are part of the interface layer of your Web service. No matter how complex and robust the functionality, it is all embodied in the interface layer of your logical architecture.

Consumers

In our analysis of the communication architecture, we started to look at the consumer responsibilities and functionality. In this view, we will look a little closer and relate its functionality back again to traditional Web applications. We will start with a high-level view of the entire consumer entity and its layers (see Figure 2-14). In this diagram, we are reinforcing the fact that the consumer owns the presentation layer and its portion of the integration layer.

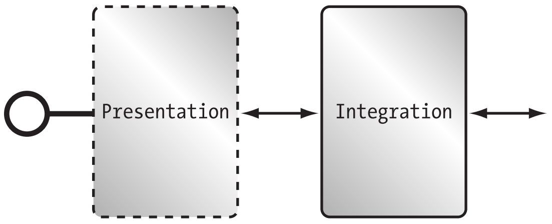

Figure 2-14. Web Service consumer logical architecture

Notice that the presentation layer is actually displayed differently. This is because this layer is optional in the sense that there may be no presentation layer to the application or the consumer of the service may not be the partner ultimately responsible for it.

Although it may be disturbing to think that the consumer may not have a presentation layer, there are many examples where this may be the case. For instance, you could have an automated back end process utilizing Web services to transfer data. Also, if the consumer is a Web service itself, it would pass its information on to another consumer, and it may or may not own the presentation layer. This gets back to the idea of stacking multiple Web services to provide a complete application (see Figure 2-15).

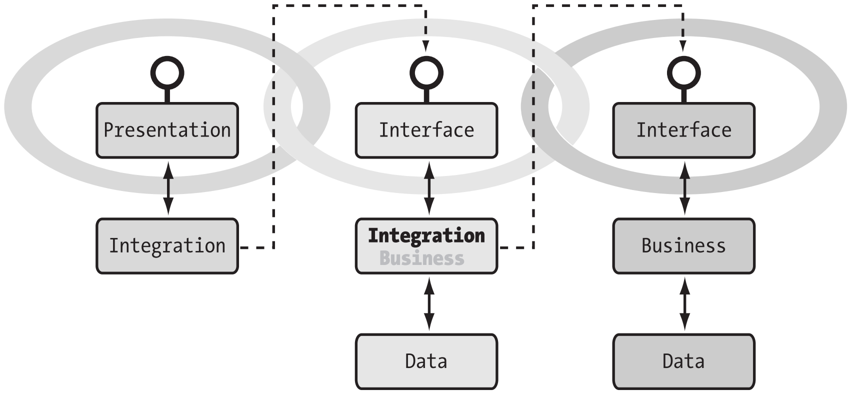

Figure 2-15. Logical architecture of an application with stacked Web services

Remember that the business layer of the first Web service also doubles as the integration/application layer for the second Web service, since it is acting as a consumer of it. Through the rest of the discussion in this chapter, we will work with a simple partnership of one consumer and one Web service. Thus, the consumer will own the presentation layer. Later, we will build logical diagrams of more complex applications based on these same rules.

Integration Layer

In our Web service, the integration/inferface layer was responsible for its interface. In the consumer, the integration portion of that layer is responsible for calling the Web service. Together, these two components allow for the integration of the external process into the local processes. Since the consumer calls the service from its business logic and integrates the two, the distinction of a consumer's integration layer is purely logical, not physical.

As we discussed earlier in "Communication Architecture," the consumer is responsible for packaging up the input data, making the call to the Web service, receiving the response, and parsing out its returned data. These functions were split into the caller and the parser, with the caller responsible for the outbound processes and the parser for the inbound. The additional function not explicitly stated is the communication with the presentation tier. Similar to the relationship in a traditional *n*-tier application between the business and presentation tiers, the consumer's presentation tier would call the integration tier for its functionality, or it could be retrieved through the business layer. Either way, the presentation layer is returned the information necessary to display to the end user. The complexity in this relationship is largely determined by how much of this information came from the Web service versus the application itself. In other words, to what extent must the presentation data be intergrated? The answer to this question can be as easy as "none" and as complex as "a lot!" With more integration comes more logic, but also more functionality and more reuse. As you can suspect, with less integration come fewer benefits. However, if you are interested in Web services, you probably want to maximize your integration potential with your partners.

In our earlier oil change manual scenario, how hard would it have been for the car manufacturer to utilize our oil change documentation? It depends on how integrated the company wanted to make it. The company could have passed it directly on to the end user, as a separate document, but it wouldn't have any of the car manufacturer's branding or information on it. They could have made several copies and manually inserted them into the manual, but it would have looked out of place. If the company had it rekeyed into a word processor, it could have merged it into the existing manual and made it fit much better, but any changes would cause a lot of rework. Certainly there are different levels of effort involved, along with different benefits. By exposing it through a Web service, we can provide a current electronic copy of the manual every time the company requests it. If all of the car manufacturer's partners cooperate on using the same process, how easy would it be to create manuals for the car with every component up to date? Very!

NOTE *In case you didn't notice, we just referenced a scenario encountering the same issues that we saw in Chapter 1 when integrating Web applications through redirection, frames, and duplicate content versus Web services!*

Once a consumer has the logic for working with a Web service, it should have the logic for working with a whole host of Web services. In fact, in the integration layer, calls may be made to multiple Web services. This is the concept of brokering Web services that we saw in Figure 2-4. This may complicate the integration layer a little, but it is primarily the same functionality so will likely benefit from reusable objects. The additional complexity comes from the integration of the resulting information into the business logic.

This is where the definition of data plays an important role. It will be up to the integration layer to parse the data returned to the consumer so that it can work with it or pass it on. This may mean simply putting the information into an XML-compatible data model like the DOM or transforming it into a more system-specific structure like an ADO recordset (if you are working in a COM environment).

TIP *Be aware that when you start to transform data between various formats, your performance is going to suffer. This is not to say transformations should never be used, because they will be necessary. You are just much better off establishing a consistent model format for your data in each tier instead of catering to the model or format preferred by each language or methodology utilized in the system.*

When the application has to work with the data, the consumer has to have a very clear understanding of what the data is and what it means. Referencing one of the industry standards for defining XML datasets will help the consumer accomplish this. However, it is up to the Web service owner to provide a definition and comply with it. If all partners are working with the same definitions, life will be much better for the consumers. Otherwise, the consumers will have to work the data into an acceptable format.

In this case, a complex design will likely be necessary for consistent integration. For instance, what would be easier for the car manufacturer to automate: assembling 100 .txt files or assembling 35 .doc, 30 .txt, 15 .rtf, 5 .sdw, and 15 .ps files from all of its partners into a manual? Sure, either one can be done, but isn't the former going to happen more often, a lot sooner, more easily, and at a lower cost?

Fortunately, if information is going to be passed directly to the end user, there is usually a lower level of understanding about the information required by the business layer. In fact, the integration layer might not even have to share the same physical tier as the business layer. That is because the presentation layer may call it separately from any other logic calls. This means that integrating Web services may not modify your existing business objects, but simply add to them. And minimizing the impact on existing code is always a good thing!

Even in these instances, the integration layer will still have some logic. Some filtering, transforming, or modifying may occur. The integration layer will at least have to parse the data so that the presentation layer can work with it. What do I mean by work with it? Let's find out.

Presentation Layer

The presentation layer can be the most complex layer in an application that consumes Web services. Additionally, this is the layer that can have the most variation from application to application. This is because all of the partnerships and collaboration of services have to come together at this point. They can be as simple as a single consumer calling a single service or as complex as a consumer brokering multiple application chains with multiple partners.

Our logical architecture will be able to support any of these models, so we have a fairly robust architecture on our hands. This will help us to break down the complexity into sublayers so that we can be certain we have accommodated nearly every conceivable scenario.

It could also be argued that this layer should be the most thought out, because it carries the most significance on a business level. It is the presenter of the application to the end user and is also probably one of the reasons the end user is there in the first place. In this competitive industry, there aren't too many poorly designed applications or Web sites with massive success these days.

> **NOTE** *Of course we remember from our earlier analogy that a chain is only as strong as its weakest link. However, many organizations emphasize the presentation more than the back-end processes, since they are more visible, and often perception is reality.*

While I am presenting the logical architecture for a Web service consumer, this same breakdown of the presentation layer could also apply to traditional *n*-tier applications. It is a little more relevant here, since this architecture forces more structure on the design due to its complexity. Web applications are known for being very forgiving to less-structured designs because their demands have been relatively light. This is similar to the forgiving nature of HTML versus XML. You can get away with more inaccuracies in HTML because it isn't considered mission critical, and often "close enough" is fine. Web services require a bit more discipline to be implemented correctly, and this architecture should help achieve that.

One of the biggest reasons for this stricter adherence is the shared nature of the presentation for applications utilizing Web services. As I mentioned before, Web services can contribute to the presentation layer. While this is not a requirement of Web services in general, it is a feature that consumers need to support architecturally. Supporting this means that the integration of the presentation information needs to happen systematically in a way that is effective and repeatable. Keep in mind that we are not talking about actual implementations, but rather the architecture defining the structure for our designs (see Figure 2-16).

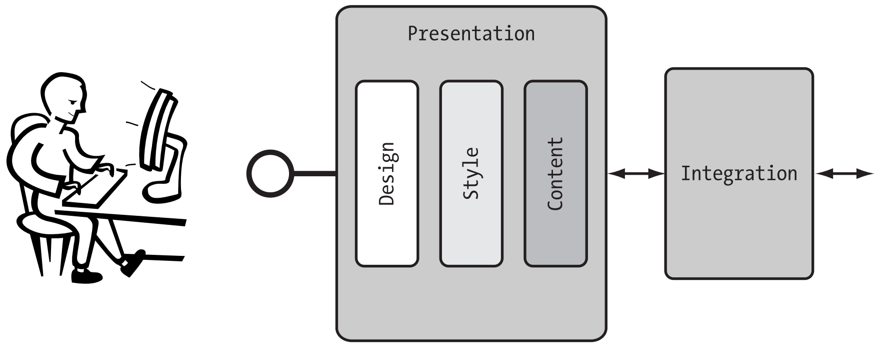

Figure 2-16. Web service consumer presentation logical sublayers

NOTE *Something worth mentioning at this point is that the focus of the presentation layer's abilities center on Web-based clients. Inherent functionality is exposed through HTML on a browser that allows for incredible flexibility in the presentation layer. (For example, it is a simple process to reference .GIF images on external systems over the Internet.) To compensate for these limitations in applications with custom clients that are not Web based, additional functionality would have to be added to your client. Of course you could make your life easier and build your interface for a Web browser!*

Content Layer

The name of this layer tends to give away its jurisdiction. While it is easy enough to say that the content layer is responsible for all the content of the application's presentation, it is really not quite that simple when you take a closer look. That content may be local, or it may be external to the application's environment. Rather than dismissing it as the directory containing all the HTML and graphic files for the content, you need to think of it as a reference to all of the content for the presentation. This puts the content layer in more of a logical state than a physical one.

Just as with current Web sites, it is simple for the local application to reference external as well as internal content. That content may take the form of HTML code, client-side scripts, images, and other supported file formats. However, the content layer may be very dynamic, since the application layer can also pass information that may reference external content. This content could be either embedded or defined in the data or simply referenced by the data. For example, the hotel service in our traffic scenario may have passed our reservation system the necessary data to define a drop-down box that allows users to select a bed size. This would allow the hotel to directly define a piece of information critical to its room availability Web service. The hotel may also reference an image on its own Web site for the purpose of cobranding the reservation application.

> *Cobranding* is the practice of partnering to share the exposure from a public marketing vehicle, in this case a public Web application, by giving exposure to the two or more organizations involved. A good example of this in everyday life is the packaging of toys from a movie inside children's meals at fast food restaurants. This effort gains publicity for both the restaurant and the movie.

CAUTION *It is worth re-emphasizing that the reservation system will have the ultimate choice of whether to utilize this information. The hotel cannot enforce, through technology, any requirements it may make of its consumers to use or not use the data it returns when accessed. We will look later at how the hotel can better protect its processes.*

Another example of service-driven content would be content derived logically by the integration layer based on what was received from the Web service call. An excellent example of this is error handling. If the Web service cannot be reached, the consumer's integration layer should handle the failure and provide some kind of data to the presentation layer to keep the end user from getting stuck in a dead end.

Regardless of the method, all of these processes result in data consisting of the content layer. We could consider our content layer a toolbox of information for building the application's presentation. By being able to work with a simple content layer, the rest of the presentation layer will have no concept of service-fed versus local content. It can treat all content as a group and thus reduce the complexity of the remaining functionality.

The content required of the application beyond the Web service functionality would also be contained in this layer. This is often referred to as base-level content for the application, since it is used independently of the Web service. This content may still be dynamically referenced and utilized by the business layer, but the integration layer will generally have no impact on this content.

Style Layer

The style layer is responsible for the formatting and treatment of all content on the site. This layer is entirely owned by the local application, since its objective is to maintain a consistent look throughout the application, regardless of whether the functionality is provided by a Web service or the local business layer. While it is possible for a Web service to define some styling, it is generally counterproductive, since the consumer will probably want to control it.

This styling can take many different forms, including cascading style sheets (CSSs), eXtensible Style Sheet Language (XSL), and a number of programming languages. Through these methods, the style layer can define how to present everything from currency to HTML form controls to plain text. This is done by either asking the client to map content to a style, as with CSS, or by transforming the content to another format prior to pushing it to the client.

> **CAUTION** *Although XSL can be used to have the browser map content to a style, this is limited to newer versions of browsers and, more importantly, is more limiting to XSL's capabilities than manipulating the content server side.*

Although the style layer defines these functions, even in the case of server-side manipulations, it is not actually responsible for making the changes or transforming the content. This would require the style layer to contain the logic of which pieces of content it should be associated with. That is actually a more involved process, since it is often affected by other content outside of its control. For that, you need functionality that is aware of everything being presented to the user, not just one piece of content. This is the responsibility of the design layer.

Design Layer

The design layer is responsible for building an application's presentation from all the tools provided through the content and style layers. This is the layer with all the control over what content goes where and how much space it consumes, what content is styled, and what content is discarded. The design layer is ultimately responsible for the look and feel of the site.

While much of this functionality can be provided statically through programmatic logic, modifying dynamic content will probably be required, depending on the application's processes. This may be as simple as referencing random images for a look of "freshness" or as sophisticated as transforming raw data into its final format for presentation to the user. While we will look at how to implement this functionality in our sample applications later, suffice it to say here that we demand much more of our presentation layer than we have in traditional *n*-tier applications.

Although this layer can contain quite a bit of sophistication, the ideal design layer implementations will offload as much logic as possible to the integration layer so that it can focus on just the presentation aspects. For instance, if the design layer is getting back 100 data points and filtering for just one to be shown, try to push that responsibility back to the integration or business layer.

The reason for streamlining this layer is twofold. First, any kind of purist will want to limit the amount of logic creeping into the presentation layer, just as you would in an *n*-tier application. Second, if you are following the practice of splitting your presentation and business layers onto separate systems, your presentation system could quickly become a bottleneck from the additional responsibilities it has taken on in dynamically building the presentation. While you can always add more systems to one tier of the system, you want to avoid getting into great disparities of system counts between the tiers. Adding logic to this layer would only compound that issue.

Now that we have discussed all the levels in the presentation layer, let us look at how they might fit together visually. In the sample layout model presented in Figure 2-17, you see a single view with many content elements from different sources. In that layout may be modules of straight content or styled content, depending on the design and the content in question. No distinction is made to where the content originated because it doesn't matter to the presentation layer. In our design scenarios, we will get much more in depth on content sourcing and how that is handled. The focus of this illustration is to present each of these sublayers and where their responsibilities lie.

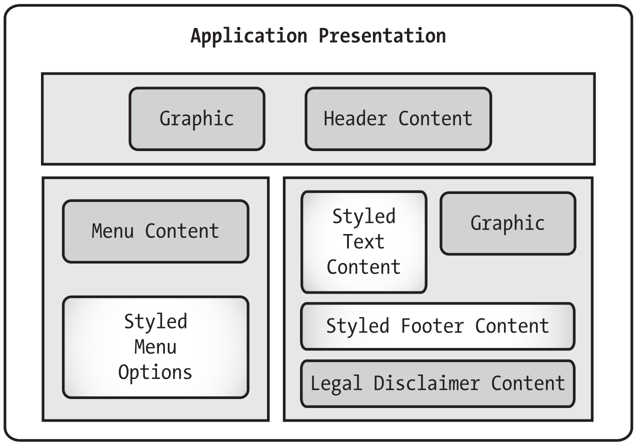

Figure 2-17. Visual segmentation of presentation sublayers

Summary

It is important to have a solid understanding of how to architect Web services before you start building them. This means having a clear picture of how the logical and communication architectures work together to provide the functionality necessary for Web services (see Figure 2-18). Unlike some architecture models, this is not simply an architecture of convenience, but rather an architecture of necessity. Any process exposed as a Web service must have an interface layer that allows consumers to interact with it. Likewise, any application that consumes Web services must have an integration layer. Together, these layers fulfill the communication architecture.

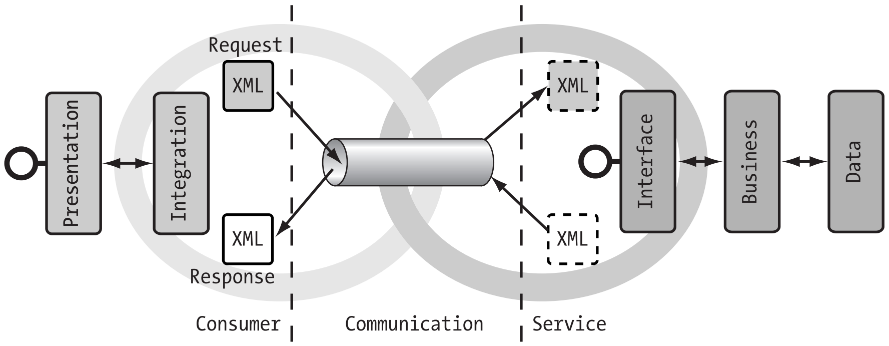

Figure 2-18. Combined logical and communication architecture for a Web services call

In our illustrations and examples, we have focused on applications in which different owners provide the service and consumer. Does this architecture change if its owner is the only consumer of the service? Since this is a logical architecture, the answer is no. The same functions and processes have to be provided regardless of the identities of the consumer and end user. However, in implementation, some things might be done differently if the service and consumer had the same owner. We will look at this scenario in our sample applications in Chapter 8.

The next question might be whether it makes sense for an organization to expose their services as a Web service. This must be answered on a case-by-case basis. For example, in our computer manufacturing process, it might make sense for the video card manufacturer to sell its product directly to consumers. It would require some work on packaging and probably enhancing the company's customer service, but it could certainly be feasible. However, under very few scenarios does it make a lot of sense for a smaller organization to expose a process as a Web service if it will be the only organization using it. That would be like the video card manufacturer using resources on the packaging for its video cards and then purchasing every card made. In fact, if the company cannot get enough external consumers to purchase its video cards, a benefit will likely not be realized because the market reuse won't be there to justify the design and development effort. Likewise, whatever application architecture is being utilized in an organization will continue to serve as the best solution in many instances.

In the next chapter we will take a brief look into some of the newer technologies needed to implement Web services. An understanding of these will be key to knowing when and how to use them in our implementations.