Chapter 1

From Objects to Web Services

Web services have been put into practical use for many years. In this time, developers and architects have encountered a number of recurring design challenges related to their usage. We have also learned that certain design approaches work better than others to solve certain problems. This book is for software developers and architects who are currently using web services or are thinking about using them. The goal is to acquaint you with some of the most common and fundamental web service design solutions and to help you determine when to use them. All of the concepts discussed here are derived from real-life lessons. Proven design solutions will also be demonstrated through code examples.

Service developers are confronted with a long list of questions.

- How do you create a service API, what are the common API styles, and when should a particular style be used?
- How can clients and services communicate, and what are the foundations for creating complex conversations in which multiple parties exchange data over extended periods of time?
- What are the options for implementing service logic, and when should a particular approach be used?
- How can clients become less coupled to the underlying systems used by a service?
- How can information about a service be discovered?
- How can generic functions like authentication, validation, caching, and logging be supported on the client or service?

- What changes to a service cause clients to break?
- What are the common ways to version a service?
- How can services be designed to support the continuing evolution of business logic without forcing clients to constantly upgrade?

These are just a few of the questions that must be answered. This book will help you find solutions that are appropriate for your situation.

In this chapter, you'll learn what services are and how web services address the shortcomings of their predecessors.

What Are Web Services?

From a technical perspective, the term **service** has been used to refer to any software function that carries out a business task, provides access to files (e.g., text, documents, images, video, audio, etc.), or performs generic functions like authentication or logging. To these ends, a service may use automated workflow engines, objects belonging to a *Domain Model* [POEAA], commercial software packages, APIs of legacy applications, Message-Oriented Middleware (MOM), and, of course, databases. There are many ways to implement services. In fact, technologies as diverse as CORBA and DCOM, to the newer software frameworks developed for REST and SOAP/WSDL, can all be used to create services.

This book primarily focuses on how services can be used to share logical functions across different applications and to enable software that runs on disparate computing platforms to collaborate. A platform may be any combination of hardware, operating system (e.g., Linux, Windows, z/OS, Android, iOS), software framework (e.g., Java, .NET, Rails), and programming language. All of the services discussed in this book are assumed to execute outside of the client's process. The service's process may be located on the same machine as the client, but is usually found on another machine. While technologies like CORBA and DCOM can be used to create services, the focus of this book is on web services. **Web services** provide the means to integrate disparate systems and expose reusable business functions over HTTP. They either leverage HTTP as a simple transport over which data is carried (e.g., SOAP/WSDL services) or use it as a complete application protocol that defines the semantics for service behavior (e.g., RESTful services).

Terminology

Web service developers often use different terms to refer to equivalent roles. Unfortunately, this has caused a lot of confusion. The following table is therefore provided for clarification and as a reference. The first column lists names used to denote software processes that send requests or trigger events. The second column contains terms for software functions that respond or react to these requests and events. The terms appearing under each column are therefore synonymous.

Client	Service
Requestor	Provider
Service consumer	Service provider

This book uses the terms "*client*" and "*service*" because they are common to both SOAP/WSDL services and RESTful services.

Web services were conceived in large part to address the shortcomings of distributed-object technologies. It is therefore helpful to review some history in order to appreciate the motivation for using web services.

From Local Objects to Distributed Objects

Objects are a paradigm that is used in most modern programming languages to encapsulate behavior (e.g., business logic) and data. Objects are usually "fine-grained," meaning that they have many small properties (e.g., `FirstName`, `LastName`) or methods (e.g., `getAddress`, `setAddress`). Since developers who use objects often have access to the internals of the object's implementation, the form of reuse they offer is frequently referred to as white-box reuse. Clients use objects by first instantiating them and then calling their properties and methods in order to accomplish some task. Once objects have been instantiated, they usually maintain state between client calls. Unfortunately, it wasn't always easy to use these classes across different programming languages and platforms. Component technologies were developed, in part, to address this problem.

Components were devised as a means to facilitate software reuse across disparate programming languages (see Figure 1.1). The goal was to provide a means whereby software units could be assembled into complex applications much like electronic components are assembled to create circuit boards. Since developers who use components cannot see or modify the internals of a component, the form of reuse they offer is called black-box reuse. Components group related objects into deployable binary software units that can be plugged into applications. An entire industry for the Windows platform arose from this concept in the 1990s as software vendors created ActiveX controls that could be easily integrated into desktop and web-based applications. The stipulation was that applications could not access the objects within components directly. Instead, the applications were given binary interfaces that described the objects' methods, properties, and events. These binary interfaces were often created with platform-specific interface definition languages (IDLs) like the Microsoft Interface Definition Language (MIDL), and clients that wished to use components frequently had to run on the same computing platform.

Objects were eventually deployed to remote servers in an effort to share and reuse the logic they encapsulated (see Figure 1.2). This meant that the memory that was allocated for clients and distributed objects not only existed in separate address spaces but also occurred on different machines. Like components, distributed objects supported black-box reuse. Clients that wished to use distributed objects could leverage a number of remoting technologies like CORBA, DCOM, Java Remote Method Invocation (RMI), and .NET Remot-

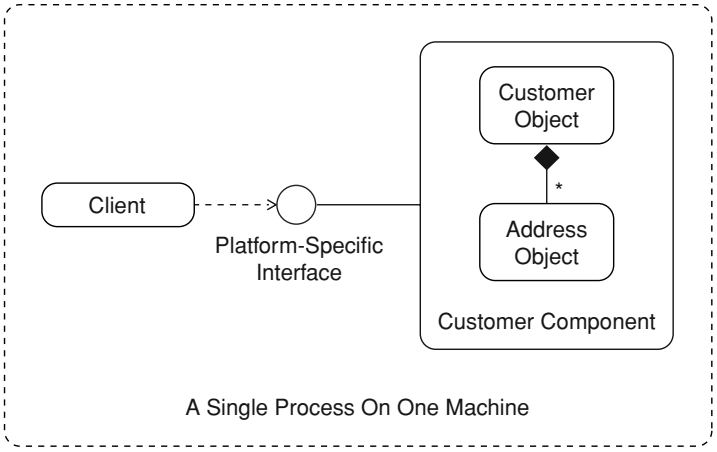

Figure 1.1 *Components were devised as a means to facilitate reuse across disparate programming languages. Unfortunately, they were often created for specific computing platforms.*

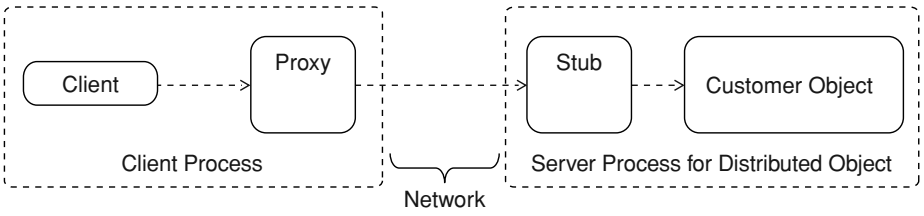

Figure 1.2 *Objects were frequently used in distributed scenarios. When a client invoked a method on the proxy's interface, the proxy would dispatch the call over the network to a remote stub, and the corresponding method on the distributed object would be invoked. As long as the client and distributed object used the same technologies, everything worked pretty well.*

ing. The compilation process for these technologies produced a binary library that included a *Remote Proxy* [GoF]. This contained the logic required to communicate with the remote object. As long as the client and distributed object used the same technologies, everything worked pretty well. However, these technologies had some drawbacks. They were rather complex for developers to implement, and the process used to serialize and deserialize objects was not standardized across vendor implementations. This meant that clients and objects created with different vendor toolkits often had problems talking to each other. Additionally, distributed objects often communicated over TCP ports that were not standardized across vendor implementations. More often than not, the selected ports were blocked by firewalls. To remedy the situation, IT administrators would configure the firewalls to permit traffic over the required ports. In some cases, a large number of ports had to be opened. Since hackers would have more network paths to exploit, network security was often compromised. If traffic was already permitted through the port, then it was often already provisioned for another purpose.

Distributed objects typically maintained state between client calls. This led to a number of problems that hindered scalability.

- Server memory utilization degraded with increased client load.
- Effective load-balancing techniques were more difficult to implement and manage because session state was often reserved for the client. The result was that subsequent requests were, by default, directed back to the server where the client's session had been established. This meant that the load for client requests would not be evenly distributed unless a sophisticated

infrastructure (e.g., shared memory cache) was used to access the client's session from any server.

- The server had to implement a strategy to release the memory allocated for a specific client instance. In most cases, the server relied on the client to notify it when it was done. Unfortunately, if the client crashed, then the server memory allocated for the client might never be released.

In addition to these issues, if the process that maintained the client's session crashed, then the client's "work-in-progress" would be lost.

Why Use Web Services?

Web services make it relatively easy to reuse and share common logic with such diverse clients as mobile, desktop, and web applications. The broad reach of web services is possible because they rely on open standards that are ubiquitous, interoperable across different computing platforms, and independent of the underlying execution technologies. All web services, at the very least, use HTTP and leverage data-interchange standards like XML and JSON, and common media types. Beyond that, web services use HTTP in two distinct ways. Some use it as an application protocol to define standard service behaviors. Others simply use HTTP as a transport mechanism to convey data. Regardless, web services facilitate rapid application integration because, when compared to their predecessors, they tend to be much easier to learn and implement. Due to their inherent interoperability and simplicity, web services facilitate the creation of complex business processes through service composition. This is a practice in which compound services can be created by assembling simpler services into workflows.

Web services establish a layer of indirection that naturally insulates clients from the means used to fulfill their requests (see Figure 1.3). This makes it possible for clients and services to evolve somewhat independently as long as *breaking changes* do not occur on the service's public interface (for more on breaking changes, refer to the section What Causes Breaking Changes? in Chapter 7). A service owner may, for example, redesign a service to use an open source library rather than a custom library, all without having to alter the client.

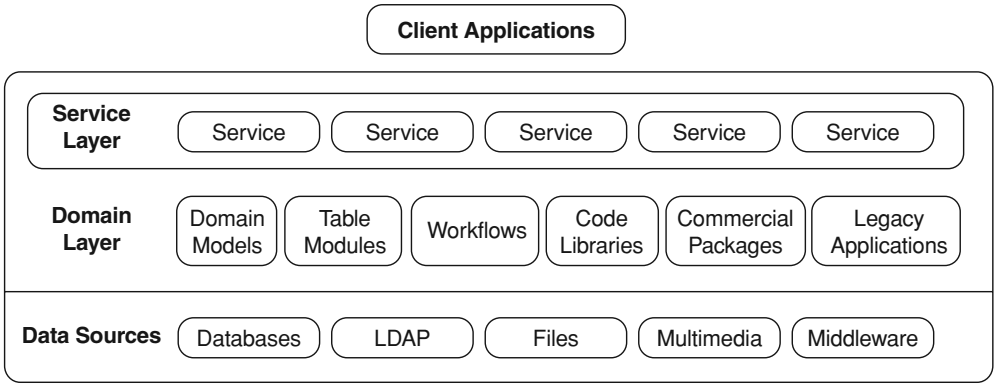

Figure 1.3 *Web services help to insulate clients from the logic used to fulfill their requests. They establish a natural layer of indirection that makes it possible for clients and domain entities (i.e., workflow logic,* Table Modules, Domain Models *[POEAA], etc.) to evolve independently.*

Web Service Considerations and Alternatives

While web services are appropriate in many scenarios, they shouldn't be used in every situation. Web services are "expensive" to call. Clients must **serialize** all input data to each web service (i.e., the request) as a stream of bytes and transmit this stream across computer processes (i.e., address spaces). The web service must **deserialize** this stream into a data format and structure it understands before executing. If the service provides a "complex type" as a response (i.e., something more than a simple HTTP status code), then the web service must serialize and transmit its response, and the client must deserialize the stream into a format and structure it understands. All of these activities take time. If the web service is located on a different machine from the client, then the time it takes to complete this work may be several orders of magnitude greater than the time required to complete a similar in-process call.

Possibly more important than the problem of latency is the fact that web service calls typically entail distributed communications. This means that client and service developers alike must be prepared to handle partial failures [Waldo, Wyant, Wollrath, Kendall]. A partial failure occurs when the client, service, or network itself fails while the others continue to function properly. Networks are inherently unreliable, and problems may arise for innumerable reasons. Connections will occasionally time out or be dropped. Servers will be overloaded from

time to time, and as a result, they may not be able to receive or process all requests. Services may even crash while processing a request. Clients may crash too, in which case the service may have no way to return a response. Multiple strategies must therefore be used to detect and handle partial failures.

In light of these inherent risks, developers and architects should first explore the alternatives. In many cases, it may be better to create "service libraries" (e.g., JARs, .NET assemblies) that can be imported, called, and executed from within the client's process. If the client and service have been created for different platforms (e.g., Java, .NET), you may still use a variety of techniques that enable disparate clients and services to collaborate from within the same process. The client may, for example, be able to host the server's runtime engine, load the services into that environment, and invoke the target directly. To illustrate, a .NET client could host a Java Virtual Machine (JVM), load a Java library into the JVM, and communicate with the target classes through the Java Native Interface (JNI). You may also use third-party "bridging technologies." These options, however, can become quite complex and generally prolong the client's coupling to the service's technologies.

Web services should therefore be reserved for situations in which out-of-process and cross-machine calls "make sense." Here are a few examples of when this might occur.

- The client and service belong to different application domains and the "service functions" cannot be easily imported into the client.
- The client is a complex business process that incorporates functions from multiple application domains. The logical services are owned and managed by different organizations and change at different rates.
- The divide between the client and server is natural. The client may, for example, be a mobile or desktop application that uses common business functions.

Developers would be wise to consider alternatives to web services even when cross-machine calls seem justified.

- MOM (e.g., MSMQ, WebSphere MQ, Apache ActiveMQ, etc.) can be used to integrate applications. These technologies, however, are best reserved for use within a secured environment, far behind the corporate firewall. Furthermore, they require the adoption of an asynchronous communications style that forces all parties to tackle several new design challenges. MOM solutions often use proprietary technologies that are platform-specific. For complete coverage of this topic, see *Enterprise Integration Patterns:*

Designing, Building, and Deploying Messaging Solutions [EIP]. Web services often forward requests to MOM.

- A certain amount of overhead should be expected with HTTP due to the time it takes for clients and servers to establish connections. This added time may not be acceptable in certain high-performance/high-load scenarios. A connectionless protocol like User Datagram Protocol (UDP) can be a viable alternative for situations like these. The trade-off, however, is that data may be lost, duplicated, or received out of order.
- Most web service frameworks can be configured to stream data. This helps to minimize memory utilization on both the sender's and receiver's end because data doesn't have to be buffered. Response times are also minimized because the receiver can consume the data as it arrives rather than having to wait for the entire dataset to be transferred. However, this option is best used for the transfer of large documents or messages rather than for real-time delivery of large multimedia files like video and audio. For situations like these, protocols such as Real Time Streaming Protocol (RTSP, www.ietf.org/rfc/rfc2326.txt), Real Time Transport Protocol (RTP, http://tools.ietf.org/html/rfc3550), and Real Time Control Protocol (RTCP, http://tools.ietf.org/html/rfc3605) are usually more appropriate than HTTP.

Services and the Promise of Loose Coupling

Services are often described as being loosely coupled. However, the definitions for this term are varied and cover a broad array of concerns. Coupling is the degree to which some entity (e.g., client) depends on another entity. When the dependencies are many, the coupling is said to be high or tight (e.g., high coupling, tightly coupled). Conversely, when the dependencies are few, coupling is considered to be low or loose (e.g., low coupling, loosely coupled).

It is certainly true that web services can eliminate the client's dependencies on the underlying technologies used by a service. However, clients and services can never be completely decoupled. Some degree of coupling will always exist and is often necessary. The following list describes a few forms of coupling that service designers must consider.

- **Function coupling:** Clients expect services to consistently produce certain results given certain types of input under particular scenarios. Clients are therefore indirectly dependent on the logic implemented by web services. The client will most certainly be affected if this logic is implemented

incorrectly or is changed to produce results that are not in accordance with the client's expectations.

- **Data structure coupling:** Clients must understand the data structures that a service receives and returns, the data types used in these structures, and the character encodings (e.g., Unicode) used in messages. If a data structure provides links to related services, the client must know how to parse the structure for that information. The client may also need to know what HTTP status codes the service returns. Service developers must be careful to refrain from including platform-specific data types (e.g., dates) in data structures.
- **Temporal coupling:** A high degree of temporal coupling exists when a request must be processed as soon as it's received. The implication is that the systems (e.g., databases, legacy or packaged applications, etc.) behind the service must always be operational. Temporal coupling can be reduced if the time at which a request is processed can be deferred. Web services can achieve this outcome with the *Request/Acknowledge* pattern (59). Temporal coupling is also high if the client must block and wait for a response. Clients may use the *Asynchronous Response Handler* pattern (184) to reduce this form of coupling.
- **URI coupling:** Clients are often tightly coupled to service URIs. That is, they often either have a static URI for a service, or follow a simple set of rules to construct a service URI. Unfortunately, this can make it difficult for service owners to move or rename service URIs, or to adopt new patterns for URI construction since actions like these would likely cause clients to break. The following patterns can help to reduce the client's coupling to the service's URI and location: *Linked Service* (77), *Service Connector* (168), *Registry* (220), and *Virtual Service* (222).

What about SOA?

Many definitions for Service-Oriented Architecture (SOA) have been offered. Some see it as a technical style of architecture that provides the means to integrate disparate systems and expose reusable business functions. Others, however, take a much broader view:

> *A service-oriented architecture is a style of design that guides all aspects of creating and using business services throughout their lifecycle (from conception to retirement).* [Newcomer, Lomow, p. 13]

Service Oriented Architecture (SOA) is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. [OASIS Ref Model]

These viewpoints suggest that SOA is a design paradigm or methodology wherein "business functions" are enumerated as services, organized into logical domains, and somehow managed over their lifetimes. While SOA can help business personnel articulate their needs in a way that comes more naturally than, say, object-oriented analysis, there are still many ways to implement services. This book focuses on several technical solutions that may be used to create a SOA.

Summary

By eliminating coupling to specific computing platforms, web services have helped us overcome one of the main impediments to software reuse. However, there are many ways to go about designing services, and developers are confronted with a long list of questions that must be resolved. This book will help you find the solutions that are most appropriate for your situation.

This page intentionally left blank