

Team LiB

NEXT ►



Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More

By Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, Donald F. Ferguson

Publisher: **Prentice Hall PTR**

Pub Date: **March 22, 2005**

Print ISBN: **0-13-148874-0**

Pages: **456**

[Table of Contents](#) | [Index](#)

Overview

The Insider's Guide to Building Breakthrough Services with Today's New Web Services Platform Using today's new Web services platform, you can build services that are secure, reliable, efficient at handling transactions, and well suited to your evolving service-oriented architecture. What's more, you can do all that without compromising the simplicity or interoperability that made Web services so attractive. Now, for the first time, the experts who helped define and architect this platform show you exactly how to make the most of it. Unlike other books, Web Services Platform Architecture covers the entire platform. The authors illuminate every specification that's ready for practical use, covering messaging, metadata, security, discovery, quality of service, business-process modeling, and more. Drawing on realistic examples and case studies, they present a powerfully coherent view of how all these specifications fit together and how to combine them to solve real-world problems.

Service orientation: Clarifying the business and technical value propositions
 Web services messaging framework: Using SOAP and WS-Addressing to deliver Web services messages
 WSDL: Documenting messages and supporting diverse message interactions
 WS-Policy: Building services that specify their requirements and capabilities, and how to interface with them
 UDDI: Aggregating metadata and making it easily available
 WS-MetadataExchange: Bootstrapping efficient, customized communication between Web services
 WS-Reliable Messaging: Ensuring message delivery across unreliable networks
 Transactions: Defining reliable interactions with WS-Coordination, WS-AtomicTransaction, and WS-BusinessActivity
 Security: Understanding the roles of WS-Security, WS-Trust, WS-SecureConversation, and WS-Federation
 BPEL: Modeling and executing business processes as service compositions

Web Services Platform Architecture gives you an insider's view of the platform that will change the way you deliver applications. Whether you're an architect, developer, technical manager, or consultant, you'll find it indispensable. Sanjiva Weerawarana, research staff member for the component systems group at IBM Research, helps define and coordinate IBM's Web services technical strategy and activities. A member of the Apache Software Foundation, he contributed to many specifications including the SOAP 1.1 and WSDL 1.1 specifications and built their first implementations. Francisco Curbera, IBM research staff member and component systems group manager, coauthored BPEL4WS, WS-Addressing, and other specifications. He represents IBM on the BPEL and Web Services Addressing working groups. Frank Leymann directs the Institute of Architecture of Application Systems at the University of Stuttgart. As an IBM distinguished engineer, he helped architect IBM's middleware stack and define IBM's On Demand Computing strategy. IBM Fellow Tony Storey has helped lead the development of many of IBM's middleware, Web services, and grid computing products. IBM Fellow Donald F. Ferguson is chief architect and technical lead for IBM Software Group, and chairs IBM's SWG Architecture Board.

Team LiB

NEXT ►

Team LiB

◀ PREVIOUS

NEXT ▶

Chapter 1. Service-Oriented Architectures

This chapter introduces the concept of Services and the associated architectural paradigm, Service-Oriented Architecture (SOA). SOA has emerged as a direct consequence of specific business and technology drivers that have materialized over the past decade. From the business side, major trends such as the outsourcing of noncore operations and the importance of business process reengineering have been key influences driving the surfacing of SOA as an important architectural approach to business information technology (IT) today. From the technology side, the past 15 years have resulted in a realization of the importance of middleware standards and architectures, learning in particular from the successes and failures of distributed object systems and message-oriented middleware.

As an architectural concept, SOA permits multiple approaches for the realization and deployment of an IT system that has been designed and built around its principles. This book focuses on a specific technology that, arguably, has the most significant commercial visibility and traction, and that is Web services. Web services technology is a standards-based, XML-centric realization of an SOA. This chapter examines the motivation behind Web services and some of the main architectural concepts of SOA. [Chapter 3](#), "Web Services," presents an overview of a Web services platform that provides the foundation to accomplish an SOA stack. [Chapter 2](#), "Background," offers some required background so that you can understand the rest of this book.

Team LiB

Team LiB

◀ PREVIOUS

NEXT ▶

◀ PREVIOUS

NEXT ▶

1.1. Virtual Enterprises

Most companies produce and sell products to their customers to achieve defined business goals, such as specific financial targets. Their business processes prescribe how products are designed, manufactured, marketed, distributed, and sold. From a certain level of abstraction, a company's business processes are a direct reflection of the products it offers. Therefore, the speed of changing the existing infrastructure or creating new business processes corresponds to the speed of changing or creating products.

To survive in highly dynamic markets and constantly changing environments, companies must be flexible. Flexibility here means the ability to react quickly (preferably faster than the competition) to new customer requirements, new product offerings by existing and new competitors, technology advances, and so on. The flexibility of an enterprise is a reflection of its ability to adapt its business processes quickly.

The constituents of the definition of a business process (see [\[Ley 2000\]](#)) translate directly into actions that are necessary to alter business processes: changing the flow of activities of a business process, changing the actors who are performing these activities, changing the tools that support these activities, and so on. These might result in moving the execution of (a subset of) the activities of the business processes to partners. The result could be multiple business partners having to cooperate to perform the business processes of a given company. The company in fact becomes a virtual company (or virtual enterprise). This term indicates that externally, the company looks like a real company performing all of its business processes itself but in practice, that is not what is actually happening. Others are partially running the company's business processes, making the company virtual in this sense.

1.1.1. Business Process Optimization

The model of a business process prescribes the order in which a business must execute the activities that constitute it and under which conditions it must perform each of the activities. Collectively, this kind of prescription is called control flow. The control flow massively influences business-relevant properties of a business process such as its total cost and its overall duration and thus the competitiveness of a company:

- The execution of each step or activity within a business process is associated with certain costs, such as people costs that are associated with the activity if human intervention is required, or the cost of computer resources that are required to execute activities within the IT environment. Based on this information, a company can derive the overall cost for performing a business process. For example, if policy associated with a credit allocation process determines that someone must check credits with an amount greater than \$1,000, and more customers are asking for credits above this limit, a business could change the policy to set manual intervention at a higher value to reduce the overall costs for running the process.
- Activities have temporal characteristics, such as the average duration for executing an activity or the average time elapsed until an activity is picked up and performed. Based on this information, you can derive the average interval for executing a business process. For example, when you are booking a business trip, you can reserve hotel and flight reservations in parallel, resulting in a shorter execution time for the overall business process relative to its sequential execution.

There are other business-relevant properties of business processes and the activities they encompass. Changing the control flow of a business process might alter the corresponding properties of the business process in an unexpected manner. For example, introducing parallelism between activities might result in a longer duration of the overall process in those cases in which the parallel activities compete for the same resources (such as staff members who have rare skills).

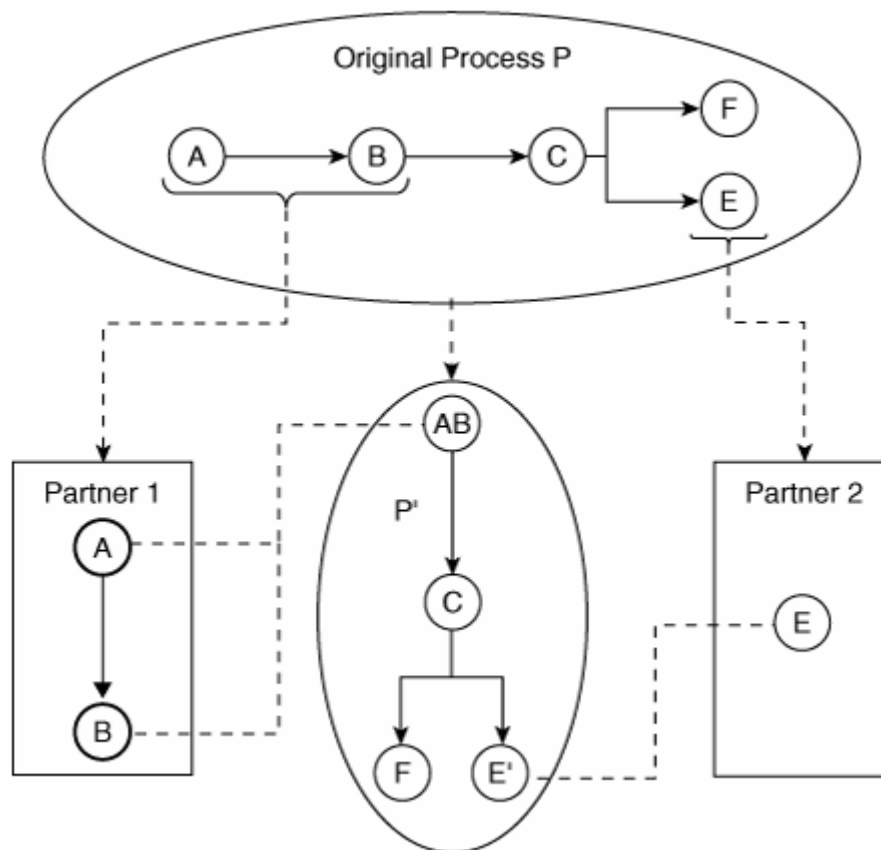
Therefore, modeling a business process is a non-trivial endeavor, supported by specialized design and analysis tools. After having modeled and optimized a business process, it must be executed in exactly the way it was specified to ensure the business properties that the process is optimized for. Special middleware is available that enforces the correct and model-compliant execution so-called workflow systems [[Ley 2000](#)]. Workflow systems allow monitoring of the business properties of individual business processes or aggregations of business processes at runtime. They also support the analysis of corresponding execution histories. Based on monitoring and analysis of results, you can change the model of a business process, if required, to further optimize it, especially when benchmarking shows that the execution of a business process is not competitive in terms of its key business parameters, such as cost or duration.

Sometimes, modifying the control flow of a business process is insufficient to hit the target values for certain business properties. For example, the cost structure within a company or wages for certain required staff might be too high to meet business expense targets. In such situations, a company's business process cannot gain competitiveness without significant re-engineering. The company (or a certain branch or location within a company) should probably determine its core competencies, focus on executing only the activities that correspond to these core competencies, and outsource the noncore competencies. Of course, the constraint is that outsourcing must result in hitting the target objectives of the subject business properties. SOA and Web service technology facilitates this kind of outsourcing in a secure, reliable manner.

1.1.2. Collaborations, Mergers, and Acquisitions

[Figure 1-1](#) demonstrates an original process (denoted by "P") that a company runs. The company determines that it is no longer competitive in performing activities A, B, and E. It finds two partners that can run these activities and meet the business goals. Partner 1 runs activities A and B, and Partner 2 runs activity E. The company re-engineered the original process into a new process (P'). In the new process, activity AB kicks off the execution at the first partner (Partner 1), and E' starts the execution at the second partner (Partner 2). The company now has a new process P', which results in increased competitiveness.

Figure 1-1. Outsourcing activities to partners.



In general, a company can work in several ways with partners that run the outsourced activities. One way is for a business to acquire a partner that can effectively run some of the activities that it could not run in a competitive manner before. A company might take this approach if the activities that it has trouble performing competitively are core to its business. A company might also choose this acquisition if it is cheaper to acquire the partner than pay the partner to perform the activities. A company might also choose to merge with a partner, essentially combining its own business with that of the partner in a peer manner. This could exploit synergies between the two companies, further reducing costs and enhancing the competitiveness of the aggregate. Usually, mergers and acquisitions result in business processes that are run in a distributed manner, across partners or virtual enterprise as described here.

Mergers or acquisitions can have a deep impact on companies, and they, thus, generally take a less radical approach. They determine appropriate partners and negotiate long-term contracts for performing specific activities on behalf of the outsourcing company. These contracts especially include service level-agreements, which are specifications of service objectives such as availability of the outsourced

activities, penalties applied when objectives are not met, bonuses paid if the objectives are overachieved, and so on [Dan 2004]. The result is a network of collaborating partners.

Situations might be highly dynamic. Multiple partners might provide the same services, and a company should choose on a case-by-case basis one of these to perform specific activities. For example, the cost of performing activities on behalf of a company might change depending on the actual load at each partner side. The company can then determine the partner on a best-price basis.

Collaboration might not only result from process optimization endeavors, but from supply chains that are typical within an industry. In this situation, a company and its partners are not distinguished. All partners are peers, collaborating to realize an overall business process that is spread over the partners. Some industries even have standards to define the various kinds of partners, the activities each of them runs, and how they relate. For example, RosettaNet [RN] defines such a standard and is currently moving toward Web service technology [RNWS].

1.1.3. Resource Sharing

In addition to business process optimization, businesses outsource activities or complete business processes for total cost of ownership considerations. Perhaps a business process is competitive overall, but the overall cost for running parts of it on IT equipment that the company owns is too high. In this situation, a company looks for a partner to run the corresponding activities or the whole business process on behalf of the company. This partner, also called application service provider (ASP), not only runs parts of the business process on its IT equipment, but it also runs the corresponding software, covering the whole spectrum from managing prerequisite software to monitoring the software and performing periodic backups.

Web service technology, technology from the area of Grid computing that is based on Web services, and autonomic computing facilitate the ASP model and extend it toward a model called utility computing (see [Ley 2004], [Rappa 2004]). Within this utility computing model, computer resources and other IT related resources are offered in a similar manner to traditional utilities such as electricity or telephone. The usage of the utility IT resources to run parts of a business process is metered and charged on this basis. You will read more about this topic later.



1.2. The Need for Loose Coupling

Enabling communication with services, possibly dynamically discovered, requires loose coupling between a requester and the service to be used. The notion of loose coupling precludes any knowledge or assumptions about the specific platforms that the requester or the service provider run on, specifics about the implementation that either of the partners uses, or the formats and protocols used to interoperate between them. Making such assumptions significantly limits the usefulness of a service or the choice of services that might be available. The notion of loose coupling is a fundamental underpinning of SOA. The following section sheds more light on it.

1.2.1. Issues with Current Distributed System Technologies

Distributed system technology that has been developed in the past allows building applications, the

elements of which can run on different machines in a network. However, dealing with the business scenarios outlined earlier has some other issues.

Fragility of Object Systems

Typically, current distributed system technology is centered on object technology. In this case, a service is offered as a method of a class implemented by an object. A requester who wants to use a service must tie the whole object into his application. In other words, a person who needs a single function has to use the whole class or even the whole class hierarchy within his program. When the class used or the class above it in the class hierarchy changes, he must also change the application that uses that class. The requester and the service are tightly coupled, which means that the requester's application is fragile because of its association with and dependence on this detail.

Lack of Interoperability

Today, different distributed system technologies such as Common Object Request Broker Architecture (CORBA), Java 2 Platform, Enterprise Edition (J2EE), and Component Object Model (COM) are based on quite different and incompatible object models [Emm 2000]. As a result, interoperability between these platforms is difficult. Communication between a requester on one platform and a service on another is at best cumbersome, if it's possible at all. Not only are the object models different, but higher-level frameworks (middleware) that support them (such as transactions, security, management, and messaging) are also incompatible. This significantly compounds the interoperability problem because you have to deal with quality of service differences. These differences can include running a transaction with participants on different platforms that operate incompatible transaction services, which is not an easy task!

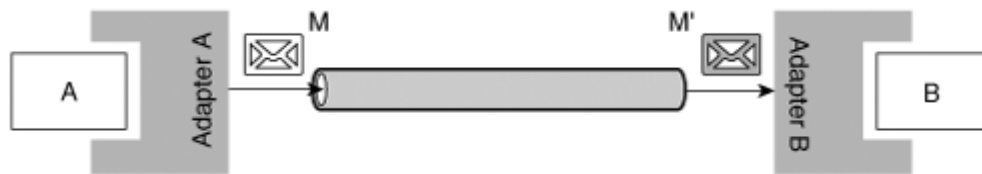
1.2.2. Advantages of Message-Oriented Middleware

A significant consequence of these interoperability problems is that islands of middleware and corresponding applications that are based on this middleware have been created. In parallel, the ability to easily integrate applications, especially from different islands, has become a strong requirement (Enterprise Application Integration EAI). In tandem, products that provide special integration middleware have been created. A key aspect of such integration middleware is message orientation.

Adapters and Channels

Message orientation refers to messages being exchanged between the applications that are to be integrated. For this purpose, adapters wrap existing applications that need to be integrated. One kind of adapter signals the occurrence of an event that needs to be passed to other applications (source adapter), and another kind of adapter receives such events and passes them to the application that they are wrapping (target adapter). Events are represented by messages that are transported via a so-called "channel" from the source adapter to a target adapter. The channel ensures a certain quality of services, such as "exactly once delivery". Also, the channel can have other side effects, such as transforming the message into a format that the recipient understands. [Figure 1-2](#) shows a source adapter A that wraps application A. Adapter A passes a message of format M to the channel, which transforms the message into format M' and delivers it reliably to target adapter B. Adapter B in turn understands how to pass the data from M' to B. See [Hohpe 2004] for additional perspectives in this space.

Figure 1-2. Message-based integration.



Based on this message-based architecture, applications A and B are loosely coupled. For example, the owner of application A can change the format of message M, generally without affecting his ability to integrate application B. The message M that A produces can be appropriately transformed by the channel into the format M' that B expects. A message-based approach like this fosters loose coupling.

Interaction Patterns

Often, interaction and communication styles, other than the synchronous request/response approach that is predominant in distributed object systems, are necessary for loosely coupled interactions. For example, the asynchronous send and receive pattern that was mentioned in the previous section can increase a requester's perspective of the overall availability of the requested service. Operating in a "send-and-forget" mode, the requester doesn't have to synchronously wait for a response from the service. The requester doesn't have to deal with potential connection problems between the requester's side and the service's side because the channel can operate in a store-and-forward manner, finally delivering the message to the target destination.

Other patterns are also desirable. For example, a message might be delivered to multiple recipients, only a subset of which responds to the originator of the message. This pattern is useful in auction-type scenarios in which a request for bids is sent and bids are returned. This also contributes to loose coupling because the requester is unaware of who the recipients are. The underlying integration middleware deals with these aspects. To overcome rigidity in distributed object systems, you must support the ability to define message exchange patterns (MEP), which provide advanced interaction patterns. MEP is discussed in [Chapter 6](#) in more details.

Web service technology is built on the concept of messaging. As a result, requesters and services can run on different platforms with channels connecting them. Wrappers hide the implementation specifics of the wrapped application function. In other words, requesters and services that are built according to different programming models can interact with each other. Protocols that are underlying a channel are hidden from the communication partners, and different formats can be transformed within a channel. The messaging architecture underlying Web services SOAP also foresees exchange of information required by higher-level functionality, such as transaction context or security context. Therefore, a messaging-based approach supports many of the requisite features at the beginning of this section.

1.2.3. Future Proofing

The concept of a wrapper allows switching implementations of application functions without impact to the other communication partner. This in turn facilitates loose coupling between a requester and a service (their corresponding wrappers). That is fine, but a universally agreed or standard approach to specifying the wrappers is required to describe, in a machine-readable way, the interface that a service provides to its potential users. The Web Services Description Language (WSDL) provides precisely this capability.

Technology Abstraction

WSDL provides a standard way of describing the interfaces of the wrappers that hide the specific implementation details of a service. Such an interface describes, in abstract terms, the functions that services provide. A requester can use any service that implements a particular interface to satisfy his requirement. This contributes to loose coupling between a requester and the service and provides some element of future-proofing. It gives the requester the freedom to select a different service implementation of the same interface at any time. In particular, the requester can benefit from any future advancements in implementation technology for services by being based on WSDL interfaces.

Provider Abstraction

Services are described not only by their WSDL interfaces, but also in terms of the quality of service they provide. For example, a service might assert that it can participate in a transaction, thereby ensuring overall integrity of a series of service invocations. It might also assert that it can cope with encrypted messages, ensuring that in the course of an interaction between a requester and the service, no confidential data will be revealed to unauthorized parties. The ability to annotate quality of service descriptions is important, and incorporated into Web service technology by means of the Web service policy specifications discussed later in this book ([Chapter 7](#), "Web Services Policy").

Also, you can associate services with business-relevant data. The directory features of the Universal Description, Discovery, and Integration (UDDI) technology (discussed later) offer and support such a capability and allow information about the company that is providing the service, including contact information, additional documentation, and geographic details. This allows discovery of suitable services based on detailed business criteria. For example, as a requester, a restaurant might want an online service that supplies ordering, settlement, and delivery of vegetables, meat, and so on within a distance of less than 50 miles.

As a consequence, the boundary for describing and discovering services is moved from specialized IT personnel toward business professionals. Again, this contributes to loose coupling by supporting discovery of providers that offer identical services and allowing switching between them dynamically with little or no change to the application.



1.3. What Is a Service?

In everyday life, you can point to many metaphors with which you can associate the concept of a service. These might include utilities such as water, gas, telephone, or electric, in addition to credit card services, transportation, travel agents, instant messaging, Internet service providers, search engines on the Web, and so on. These services represent some sort of publicized package of functionality. They are composable. For example, a travel agent makes use of transportation (airline and rental car) and credit card services. Services are discoverable based on their descriptions, terms and conditions for use, and so on, based on metadata that fully describes the service. Actual use of a service is often based on an agreed-upon contract with the provider, including what in detail is provided and what the associated quality of services are, such as availability, cost, and other specified conditions that govern its use. Generally speaking, the user of the service requires little or no knowledge as to the specifics of how the service is implemented or how it is provided. The following sections relate these characteristics and apply them to software services.

1.3.1. Evolution of Major Software Granules

The idea of packaging software into artifacts that have a wider context, use, and applicability than a

single application has been given a lot of attention since the early days of computing. This objective also relates to more flexible units of deployable software. Additional benefits are derived through separation of concerns, which leads to a significantly better understanding of the overall anatomy of complex software systems. This in turn enables improved software development methodologies necessary for tackling today's complex problems, in addition to improvements in software maintainability. Over the past decades, the ideas that have been applied to this problem have evolved quite significantly.

Functions and Packages

One of the first attempts to decompose software was centered on functional decomposition that resulted in functions or subroutines as individual software units. Such a unit offers coherent functionality that you can easily understand and build. This decomposition into functions enables modularization of software systems. However, this approach frequently brings about partial simplification that can only address simple problems. Added value comes with aggregations of related functions that help solve more complex tasks that are associated with a particular problem domain. Such approaches have, for example, been pioneered by Ada in the form of packages.

Objects and Classes

A package is a mere syntactic unit. To improve on and extend this concept further, the notion of a class evolved. A class describes the behavior of objects from a problem domain; therefore, it carries certain semantics in addition to being a syntactic unit combining functions and data. Classes, together with the concept of polymorphism and inheritance, have turned out to be a powerful concept for building software. Polymorphism allows the resulting software to become more flexible, whereas through inheritance, class lattices can be built that stimulate reuse and ease communication among developers within a given community.

Although classes represent objects and carry some semantics, a class lattice mostly captures syntaxnamely, the signature of the functions aggregated. The semantics that a class lattice captures are typically understood only within a limited community, such as a development team. Few class lattices are known, the semantics of which are shared across larger communities.

In contrast to this, services are described not only via their interfaces, but also via business terms. Thus, the semantics that a service description provides go beyond that of a class and a class lattice. As such, services support comprehension by much broader communities.

Components

Besides being coarser than objects and classes, components go further by dealing with deployment. Addressing the issue of deployment allows a component provider to focus specifically on application logic, while qualities of services are folded in by the runtime environment that is hosting the component. For that purpose, quality of service requirements are factored and parameterized by deployment descriptors [J2EE]. By setting appropriate values in the deployment descriptors, you specify the behavior of a component with respect to such things as transactions, security checking, state management, and so on. In addition, you can resolve dependencies on other components and resources (such as databases and queues) via deployment descriptors. Finally, you can set up values governing application-specific context (such as local tax rates) via deployment descriptors.

Dealing with these aspects of deployment eases customization. You can tailor a piece of application logic to a company's specific use by setting values in deployment descriptors accordingly. The container that is hosting the application logic interprets the deployment descriptors and ensures the specified

behavior. For a customized context that the application logic requires, the deployment descriptor provides the standardized contract. Deployment descriptors enable components to become tradeable artifacts.

1.3.2. The Software Version of a Service

Services represent another step forward in the evolution of software packages. Services can provide an abstraction of specific component models, allowing users of these components to think only in terms of Web services and ignore specific details of the component model and how it is implemented. For example, services can hide the details about whether the component is a J2EE-based component or a .NET based component. This provides significantly enhanced interoperability between computing platforms that have programming models based on these differing component models, such as WebSphere and .NET.

Characteristics of a Service

A service is available at a particular endpoint in the network, and it receives and sends messages and exhibits behavior according to its specification. The service has specific functionality and is deployed with appropriate quality of service at the endpoint. (For example, the service is set up with certain transactional or security or reliable messaging behavior.) The functional aspects of a service are specified using WSDL, and the constraints and conditions that are associated with the use of the service are specified via policies that you can attach to various parts of the WSDL (see [chapter 7](#)). Interfaces and policies describe the terms and conditions that govern the use of the service. These are published so that potential users of the service can discover and be given all the information they need to bind (perhaps dynamically) to that service.

The information that is published about a service provides details of what the service is and does (its semantics). It further provides all the information that allows the environment hosting a potential user to access the service and successfully interact with it. This can include information about the transport protocols that can be supported and used to send a message to the service, the wire format of this message expected by the service, whether and how the message has to be encrypted or signed.

Although the environment that is hosting needs this information, the requester doesn't need this kind of access information, and it certainly doesn't need to understand details about the implementation of the service. You can implement this service as an EJB, a database stored procedure, or a CICS or IMS transaction made available via a wrapper. The environment that is hosting the service and receiving the request message, often referred to as a container, must deal with that detail necessary to interact with a particular implementation. The requester can think in terms of using a Web service. In that sense, Web service technology is virtualization technology for making use of services, but an implementer of a service can use the technology he is acquainted with to build the service implementation.

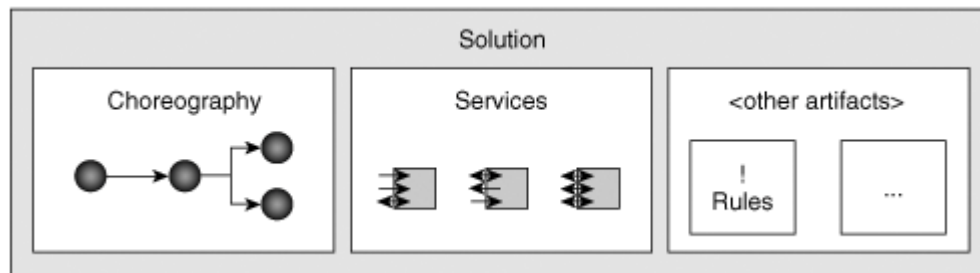
SolutionsComposition of Services

One valuable aspect of the services model is that you can create new services from existing ones without leaving the service paradigm. A technology called choreography facilitates the composition and orchestration of the execution of existing services to create more powerful ones (see [Chapter 14](#), "Modeling Business Processes: BPEL"). Choreographies support the idea of programming in the large (see [[Ley 2004](#)], [[Ley 1997](#)], [[Wied 1992](#)]), which fosters the creation of new functionality, in particular by non-IT personnel.

Using the terminology introduced earlier, a business process corresponds to the choreography of

services designed to address a high-level business problem. The choreography specifies which services to use, in which order, and under which conditions. Services within the choreography correspond to an activity, some kind of well-identified step in the process. You can define the choreography as a service and incorporate it as an activity within another choreography, again, providing functions from the business domain of a company. Because the collection of services and the choreography that uses them solve a business problem, this collection is sometimes called a solution [BaS04]. Without examining the detail, note that a solution typically contains additional artifacts, such as user interfaces or more complex rules that the choreography uses (see [Figure 1-3](#)).

Figure 1-3. The concept of a solution.



Choreographies use services and render the results as services again. Therefore, one choreography can serve as a base for other choreographies, together with services that have been implemented based on other technologies. Thus, choreography provides a powerful technology for recursive composition of services.

Team LiB
Team LiB

◀ PREVIOUS NEXT ▶
◀ PREVIOUS NEXT ▶

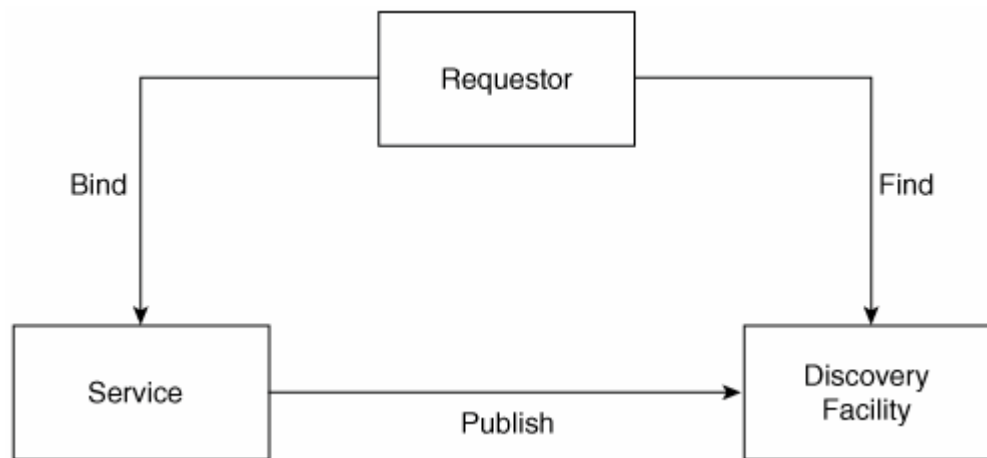
1.4. Service-Oriented Architecture

SOA is a specific architectural style that is concerned with loose coupling and dynamic binding between services. Some critically important factors at the heart of SOA are necessary to make it work effectively.

1.4.1. Bind/Publish/Find

The basic principles that underpin SOA are illustrated in [Figure 1-4](#). First, it is necessary to provide an abstract definition of the service, including the details that allow anyone who wants to use the service to bind to it in the appropriate way. Second, those people who are the providers of services need to publish details of their services so that those who want to use them can understand more precisely what they do and can obtain the information necessary to connect to use them. Third, those who require services need to have some way to find what services are available that meet their needs. Also, to make this bind/publish/find approach work well, standards must be defined to govern what and how to publish, how to find information, and the specific details about how to bind to the service. Web services technology is at the heart of addressing these questions and is the subject of this book. It also forms a basis for the notion of a service bus (infrastructure) that supports SOA.

Figure 1-4. The SOA Triangle.

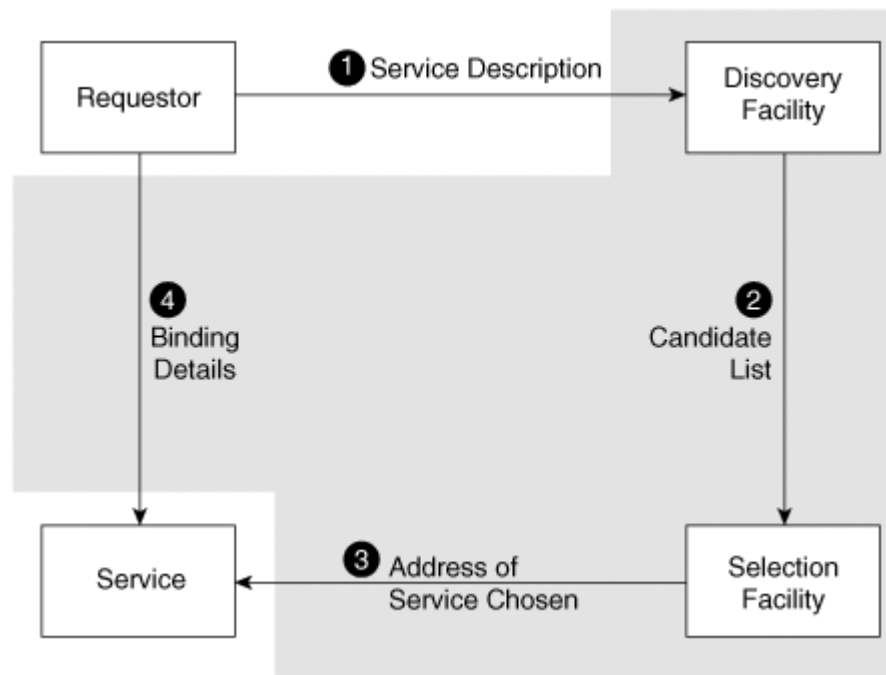


The provider of a service describes its semantics, that is, the functions it supports in addition to its operational characteristics and required details about how to access the service. This descriptive information (or metadata) about the service is published to a directory or registry. A requester uses a discovery facility that is associated with the registry to find services that he is interested in and selects the one that is required. Based on the access information published about the service, the requester can bind to the selected service.

Descriptive information about a service, as indicated earlier and described in detail later, is based on WSDL and policy. At one end of the spectrum of service publishing and discovery is a separate registry or directory, such as UDDI, which holds WSDL and policy information about registered service and additional information such as business data about the service provider. This information is queried for services that match the requester's criteria. At the other end of the spectrum, more rudimentary mechanisms can provide access to and retrieval of metadata about a service, the existence of which is known to the requester perhaps by private means. However, after a service has been chosen, the information regarding how to access the service is used next to actually bind to it and send requests to it.

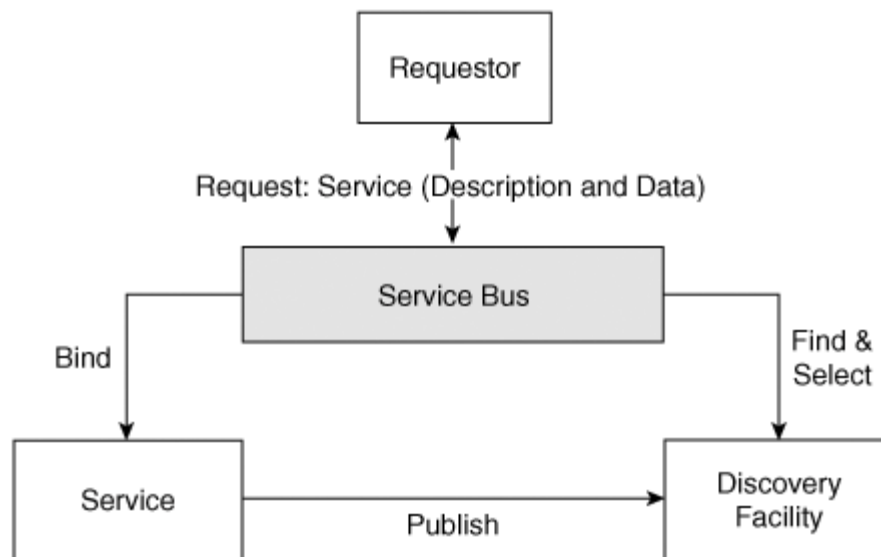
[Figure 1-5](#) illustrates the various steps and artifacts that dynamically discover and bind a service. First, the requester describes the service needed via functions required and nonfunctional properties. Based on this input, the discovery facility produces a list of candidate services that satisfy those needs. From this candidate list, the requester chooses a service. The selection facility shown for that purpose can be as simple as performing a random selection or as sophisticated as considering additional criteria such as cost of use. After the requester has chosen the service, he determines its address and binding details, such as which protocol to use and which message format is required. This information is either already returned within the candidate list details or is retrieved in a separate step.

Figure 1-5. Selecting and binding services.



This processing looks cumbersome. However, middleware called a service bus (see [Voll 2004], for example) can simplify the process and make it more transparent. As illustrated in [Figure 1-6](#), the requester simply passes to the service bus the description of the service and the data that the request message expects to be sent. Next, the requester gets the response to his declarative request. Based on this request, all services that qualify under the service description are undistinguishable for the requester, and the bus can unilaterally select from all candidates. The bus virtualizes these candidate services from the requester's perspective. For this purpose, the bus implements the shaded area in [Figure 1-6](#). In other words, the bus takes the service description and uses a discovery facility to find the list of qualifying services, selects one of them, retrieves the necessary binding information, binds the service accordingly, transforms the input data from the requester properly, sends the corresponding request message to the service, receives the response, and passes this response in the proper format to the requester. A service bus provides a significantly improved ease-of-use experience for a Web servicebased implementation of SOA.

Figure 1-6. The role of the service bus in SOA.



The functionality that a service bus provides goes beyond virtualizing application functionality. Not only can application functions be described via WSDL, but also the usage of resources such as processing power (CPUs) or storage (disks) can be considered as services thus being described via WSDL. Thus, a requester might describe his need for processing power and storage to the bus, and the bus will make this available. Of course, this requires much more sophistication, including scheduling features and extensions of the Web service model. Nevertheless, the service bus is a major enabling technology for a distributed computing environment called Grid (see [\[Fost 2004\]](#), [\[Czaj 2004\]](#)). You can think of it as service-oriented middleware.

Note that CORBA already had some aspects of the SOA in it based on its Trader services. However, from an overall SOA perspective, these trader services were incomplete. Often, they were not implemented, and they certainly weren't universal in the sense of being platform neutral.

1.4.2. Framework for SOA

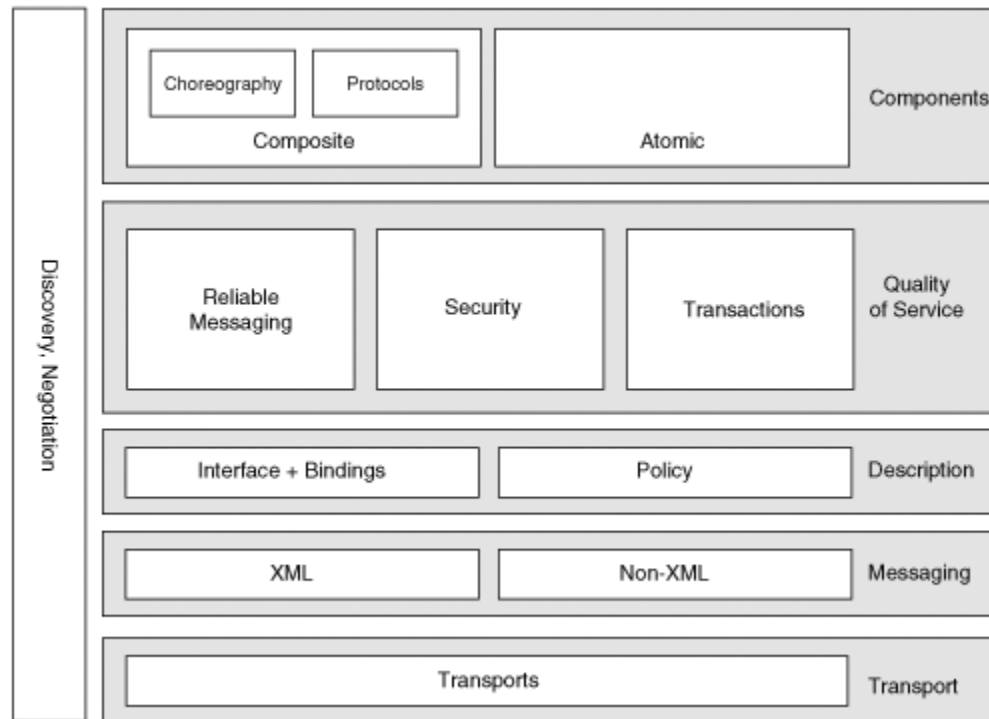
Web service technology is a base for implementing SOA, and the service bus is the centerpiece of this implementation. Thus, we want to sketch the high-level architecture of a service bus.

[Figure 1-7](#) depicts the high-level architecture of the service bus as a stack of SOA capabilities. The bottom layer presents its capabilities to cope with various transport protocols to communicate between a service and a requester. The messaging layer on top of the transport layer enables the bus to deal with messages, both XML and non-XML. The latter is important, for example, if a requester and the service needed reside in the same J2EE application server and the bus can simply transport the data in its Java rendering, avoiding unnecessary transformations. The next layer of the bus facilitates and deals with the description of services in terms of functions supported, quality of services of these functions, and supported binding mechanisms. The actual quality of services that the bus enforces based on appropriate parameterization via policies resides in the layer that follows. This layer copes with security aspects such as message integrity, confidentiality, and nonrepudiation, but also with reliable transport of messages and various kinds of transactions. The top layer represents the various kinds of virtual components that Web services represent. This layer has atomic services that are not composed as far as a requester's experience with the service is concerned. Composed services that the service bus inherently supports are choreographies and societies of services that cooperate following an agreement protocol to decide on the success of the cooperation at the end of the cooperation. Finally, another layer provides features for

discovery of services and their descriptions and to agree on a mode of interaction between a requester and a service.

Figure 1-7. The SOA stack: High-level architecture of the bus.

[\[View full size image\]](#)



[Chapter 3](#) revisits this topic and describes how various specifications fit into this SOA stack, with special focus on Web service specifications.

Team LiB
Team LiB

◀ PREVIOUS NEXT ▶
◀ PREVIOUS NEXT ▶

1.5. Summary

This chapter covered two major drivers that foster SOAs: an ever-increasing number of collaborations between enterprises that result from focusing on core competencies and enhancing agility, and the pressing need to solve the integration problem by loosening the coupling between functional components. This chapter explored the concept of a service as an evolution of software component technology and discussed the impact of services on application structures. It also presented the principles of SOA and introduced a framework of an environment that supports SOA applications.

Team LiB

◀ PREVIOUS NEXT ▶

Team LiB

◀ PREVIOUS

NEXT ▶

Chapter 3. Web Services: A Realization of SOA

People often think of Web services and Service-Oriented Architecture (SOA) in combination, but they are distinct in an important way. As discussed in [Chapter 1](#), "Service-Oriented Architectures," SOA represents an abstract architectural concept. It's an approach to building software systems that is based on loosely coupled components (services) that have been described in a uniform way and that can be discovered and composed. Web services represents one important approach to realizing an SOA.

The World Wide Web Consortium (W3C), which has managed the evolution of the SOAP and WSDL specifications, defines Web services as follows:

A software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with XML serialization in conjunction with other Web-related standards.

Although Web services technology is not the only approach to realizing an SOA, it is one that the IT industry as a whole has enthusiastically embraced. With Web services, the industry is addressing yet again the fundamental challenge that distributed computing has provided for some considerable time: to provide a uniform way of describing components or services within a network, locating them, and accessing them. The difference between the Web services approach and traditional approaches (for example, distributed object technologies such as the Object Management Group Common Object Request Broker Architecture (OMG CORBA), or Microsoft Distributed Component Object Model (DCOM)) lies in the loose coupling aspects of the architecture. Instead of building applications that result in tightly integrated collections of objects or components, which are well known and understood at development time, the whole approach is much more dynamic and adaptable to change. Another key difference is that through Web services, the IT industry is tackling the problems using technology and specifications that are being developed in an open way, utilizing industry partnerships and broad consortia such as W3C and the Organization for the Advancement of Structured Information Standards (OASIS), and based on standards and technology that are the foundation of the Internet.

This open, standards-based approach in which every Web services specification is eventually standardized by an industry-wide organization (such as W3C or OASIS) introduces the possibility that the specifications described in this book might undergo significant changes before becoming formal standards. This is a natural consequence of the standardization process in which both technology vendors and consumers provide input and push their requirements into the final standard. However, the basic concepts and the design supporting each of the specifications are unlikely to change in fundamental ways, even if the syntax is modified or the supported set of use cases is significantly expanded. At the time of publication, several of the specifications covered in this book have already been submitted to standards, and significant changes may ensue in some of them (for example, in the case of WS-Addressing, now being discussed at W3C). Readers interested in the details of the specifications should be aware of this fact and carefully follow the results of the standardization process. Please refer to the Web site, www.phptr.com, "Updates and Corrections," where you will find the latest updates to the specifications covered in this book.

Team LiB

Team LiB

◀ PREVIOUS

NEXT ▶

◀ PREVIOUS

NEXT ▶

3.1. Scope of the Architecture

The high-level schematic introduced in [Chapter 1](#) ([Figure 1-7](#)) illustrates a layered view of the important foundational capabilities that are required of SOA. This chapter introduces a specific rendering of this conceptual framework with a particular collection of Web services specifications that are based on and extend basic Internet standards that were described in [Chapter 2](#), "Background." Note that specifications used in this rendering are those that IBM has developed in a collaborative effort with other industry partners, most notably Microsoft. The description in this chapter is intended to give a high-level "fly by" only, with the express purpose of providing an overall summary perspective. The following chapters of this book discuss these Web services specifications in much greater detail.

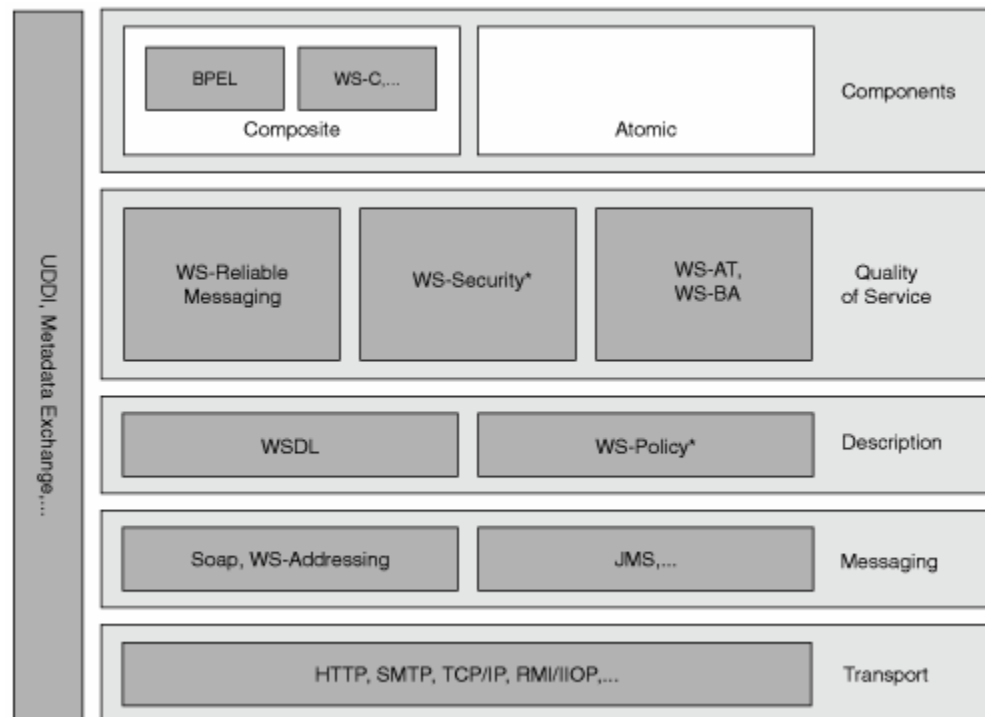
Web services had its beginnings in mid to late 2000 with the introduction of the first version of XML messaging SOAP, WSDL 1.1, and an initial version of UDDI as a service registry. This basic set of standards has begun to provide an accepted industry-wide basis for interoperability among software components (Web services) that is independent of network location, in addition to specific implementation details of both the services and their supporting deployment infrastructure. Several key software vendors have provided these implementations, which have already been widely used to address some important business problems.

Although the value of Web services technology has been demonstrated in practice, there is a desire to use the approach to address more difficult problems. Developers are looking for enhancements that raise the level and scope of interoperability beyond the basic message exchange, requiring support for interoperation of higher-level infrastructure services. Most commercial applications today are built assuming a specific programming model. They are deployed on platforms (operating systems and middleware) that provide infrastructure services in support of that programming model, hiding complexity, and simplifying the problems that the solution developer has to deal with. For example, middleware typically provides support for transactions, security, or reliable exchange of messages (such as guaranteed, once-only delivery). On the other hand, there is no universally agreed standard middleware, which makes it difficult to construct applications from components that are built using different programming models (such as Microsoft COM, OMG CORBA, or Java 2 Platform, Enterprise Edition (J2EE) Enterprise Java Beans). They bring with them different assumptions about infrastructure services that are required, such as transactions and security. As a consequence, interoperability across distributed heterogeneous platforms (such as .NET and J2EE) presents a difficult problem.

The Web services community has done significant work to address this interoperability issue, and since the introduction of the first Web services, various organizations have introduced other Web services-related specifications. [Figure 3-1](#) illustrates a population of the overall SOA stack shown in [Figure 1-7](#) with current standards and emerging Web services specifications that IBM, Microsoft, and other significant IT companies have developed. The remainder of this chapter provides a high-level introduction to these Web services specifications that realize more concretely the capabilities that are described in the SOA framework in [Chapter 1](#) and that extend the earlier Web services technology of XML, SOAP, and WSDL to provide secure, reliable, and transacted interoperability. The specifications define formats and protocols that allow services to interoperate across those vendor platforms that provide conformant implementations, either natively or by mapping them onto existing proprietary middleware offerings.

Figure 3-1. Web services architecture.

[\[View full size image\]](#)



```

<definitions targetNamespace="...">
  <!-- WSDL definitions in this document -->
  <!-- referenced using "tns" prefix -->

  <types>
    <!-- XSD definitions for this service -->
    <!-- referenced using "xsd1" prefix -->
    <xsd:schema>
      <xsd:import
namespace="http://www.purchase.com/xsd/svp-svc">
    </xsd:schema>
  </types>

  <message name="purchaseResponse">
    <part name="purchaseResponse"
element="xsd1:PurchaseStatus"/>
  </message>
  <message name="purchaseRequest">
    <part name="purchaseRequest"
element="xsd1:PurchaseRequest"/>
  </message>
  <message name="ServicePacValidationInput">
    <part name="spvDataInput"
      element="xsd1:ServicePacValidationData"/>
  </message>
  <message name="ServicePacValidationOutput">
    <part name="spvDataOutput"
      element="xsd1:ServicePacValidationData"/>
  </message>

  <portType name="spvPortType">
    <operation name="purchaseServicePacs">
      <input name="purchaseInput"

```

```

message="tns:purchaseRequest"/>
    <output name="purchaseOutput"
        message="tns:purchaseResponse"/>
    </operation>
    <operation name="validateServicePac">
        <input name="Input"
            message="tns:ServicePacValidationInput"/>
        <output name="Output"
            message="tns:ServicePacValidationOutput"/>
    </operation>
</portType>

<binding name="spvBinding" type="tns:spvPortType">
    <wsp:PolicyReference
        URI="http://www.purchase.com/policies/DSig">
    <soap:binding style="document"

transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="purchaseServicePacs">
        <wsp:PolicyReference URI=
            "http://www.purchase.com/policies/Encrypt">
        <soap:operation soapAction=

"http://www.purchase.com/spvPortType/purchaseServicePacsRequest"/>
    </operation>
    <operation name="validateServicePac">
        <soap:operation soapAction=

"http://www.purchase.com/spvPortType/validateServicePacRequest"/>
    </operation>
</binding>

<service name="spv-svc">
    <port name="spv-svc-port" binding="tns:spvBinding">
        <soap:address
location="http://www.purchase.com/spv"/>
    </port>
</service>
</definitions>

```

Team LiB

Team LiB

◀ PREVIOUS

NEXT ▶

◀ PREVIOUS

NEXT ▶

3.2. Transport Services

Web services is basically an interoperable messaging architecture, and message transport technologies form the foundation of this architecture. Web services is inherently transport neutral. Although you can transport Web services messages by using the ubiquitous Web protocols such as HyperText Transport Protocol (HTTP) or Secure HTTP (HTTPS) to give the widest possible coverage in terms of support for the protocols (see [Chapter 2](#)), you can also transport them over any communications protocol, using proprietary ones if appropriate. Although transport protocols are fundamental to Web services and clearly are a defining factor in the scope of interoperability, the details are generally hidden from the design of Web services. A detailed discussion of these is not included in the scope of this book.

Team LiB

Team LiB

◀ PREVIOUS

NEXT ▶

◀ PREVIOUS

NEXT ▶

3.3. Messaging Services

The messaging services component of the framework contains the most fundamental Web services specifications and technologies, including eXtensible Markup Language (XML), SOAP, and WS-Addressing. Collectively, these specifications form the basis of interoperable messaging between Web services. XML (discussed in [Chapter 2](#)) provides the interoperable format to describe message content between Web services and is the basic language in which the Web services specifications are defined.

3.3.1. SOAP

SOAP, one of the significant underpinnings of Web services, provides a simple and relatively lightweight mechanism for exchanging structured and typed information between services. SOAP is designed to reduce the cost and complexity of integrating applications that are built on different platforms. SOAP has undergone revisions since its introduction, and the W3C has standardized the most recent version, SOAP 1.2.

SOAP defines an extensible enveloping mechanism that scopes and structures the message exchange between Web services. A SOAP message is an XML document that contains three distinct elements: an envelope, a header, and a body. The envelope is the root element of the SOAP message. It contains an optional header element and a mandatory body element. The header element is a generic mechanism for adding extensible features to SOAP. Each child element of header is called a header block. SOAP defines several well-known attributes that you can use to indicate who should deal with a header block and whether processing of it is optional or mandatory. The body element is always the last child element of the envelope, and it is the container for the payload—the actual message content that is intended for the ultimate recipient who will process it. SOAP defines no built-in header blocks and only one payload, which is the Fault element used for reporting errors.

SOAP is defined independently of the underlying messaging transport mechanism in use. It allows the use of many alternative transports for message exchange. You can defer selection of the appropriate mechanism until runtime, which gives Web service applications or support infrastructure the flexibility to determine the appropriate transport as the message is sent. In addition, the underlying transport might change as the message is routed between nodes. Again, the mechanism that is selected for each hop can vary as required. Despite this general transport independence, most first-generation Web services communicate using HTTP, because it is one of the primary bindings included within the SOAP specification.

SOAP messages are transmitted one way from sender to receiver. However, multiple one-way messages can be combined into more sophisticated message exchange patterns. For instance, a popular pattern is a synchronous request/response pair of messages. The messaging flexibility that SOAP provides allows services to communicate using a variety of message exchange patterns, to satisfy the wide range of distributed applications. Several patterns have proven particularly helpful in distributed systems. The use of remote procedure calls, for example, popularized the synchronous request/response message exchange pattern. When message delivery latencies are uncontrolled, asynchronous messaging is needed. When the asynchronous request/response pattern is used, explicit message correlation becomes mandatory.

Messages can be routed based on the content of the headers and the data inside the message body. You can use tools developed for the XML data model to inspect and construct complete messages. Note that such benefits were not available in architectures such as DCOM, CORBA, and Java Remote Method Invocation (RMI), where protocol headers were infrastructural details that were opaque to the

application. Any software agent that sends or receives messages is called a SOAP node. The node that performs the initial transmission of a message is called the original sender. The final node that consumes and processes the message is called the ultimate receiver. Any node that processes the message between the original sender and the ultimate receiver is called an intermediary. Intermediaries model the distributed processing of an individual message. The collection of intermediary nodes traversed by the message and the ultimate receiver are collectively referred to as the message path.

To allow parts of the message path to be identified, each node participates in one or more roles. The base SOAP specification defines two built-in roles: Next and UltimateReceiver. Next is a universal role in that every SOAP node, other than the sender, belongs to the Next role. UltimateReceiver is the role that the terminal node in a message path plays, which is typically the application, or in some cases, infrastructure that is performing work on behalf of the application. The body of a SOAP envelope is always targeted at the UltimateReceiver. In contrast, SOAP headers might be targeted at intermediaries or the UltimateReceiver

SOAP is discussed in detail in [Chapter 4](#), "SOAP."

3.3.2. WS-Addressing

WS-Addressing provides an interoperable, transport-independent way of identifying message senders and receivers that are associated with message exchange. WS-Addressing decouples address information from the specific transport used by providing a mechanism to place the target, source, and other important address information directly within the Web service message. This specification defines XML elements to identify Web services endpoints and to secure end-to-end endpoint identification in messages. This specification enables messaging systems to support message transmission through networks that include processing nodes such as endpoint managers, firewalls, and gateways in a transport-neutral manner.

WS-Addressing defines two interoperable constructs that convey information that transport protocols and messaging systems typically provide. These constructs normalize this underlying information into a uniform format that can be processed independently of transport or application. These two constructs are endpoint references and message information headers.

A Web services endpoint is a referenceable entity, processor, or resource in which Web services messages can be targeted. Endpoint references convey the information needed to identify/reference a Web services endpoint, and you can use them in several different ways. Endpoint references are suitable for conveying the information needed to access a Web services endpoint, but they also provide addresses for individual messages that are sent to and from Web services. To deal with this previous usage case, the WS-Addressing specification defines a family of message information headers that allows uniform addressing of messages independent of underlying transport. These message information headers convey end-to-end message characteristics, including addressing for source and destination endpoints and message identity.

Both of these constructs are designed to be extensible and reusable so that other specifications can build on and leverage endpoint references and message information headers. WS-Addressing is covered in detail in [Chapter 5](#), "WS-Addressing."

3.4. Service Description

Service description defines metadata that fully describes the characteristics of services that are deployed on a network. This metadata is important, and it is fundamental to achieving the loose coupling that is associated with an SOA. It provides an abstract definition of the information that is necessary to deploy and interact with a service.

3.4.1. WSDL

Web Services Description Language (WSDL) is perhaps the most mature of metadata describing Web services. It allows developers to describe the "functional" characteristics of a Web service: what actions or functions the service performs in terms of the messages it receives and sends. WSDL offers a standard, language-agnostic view of services it offers to clients. It also provides noninvasive future-proofing for existing applications and services and allows interoperability across the various programming paradigms, including CORBA, J2EE, and .NET.

WSDL is an XML format for describing (network) services as a set of endpoints that operate on messages containing either document-oriented or procedure-oriented information. The operations and messages are described abstractly and then bound to a concrete network protocol and message format to define an endpoint. Related concrete endpoints are combined into abstract endpoints (services). WSDL is extensible to allow description of endpoints and their messages regardless of what message formats or network protocols are used to communicate.

A WSDL document has two parts: abstract definitions and concrete descriptions. The abstract section defines SOAP messages in a language- and platform-independent manner. In contrast, the concrete descriptions define site-specific matters such as serialization.

WSDL provides support for a range of message interaction patterns. It supports one-way input messages that have no response, request/response, and one-way sends with or without a response. The last two patterns enable a service to specify other services that it needs. WSDL is discussed in detail in [Chapter 6](#), "Web Services Description Language."

3.4.2. Policy

Although WSDL and XML Schema describe what a service can do by providing a definition of the business interface (including business operations such as open/close account, debit/credit/transfer, and so on), they do not provide information about how the service delivers its interface or what the service expects of the caller when it uses the service. How the service implements the business interface, the sort of permissions or constraints it expects of or provides to requesters, and what is expected or provided in a hosting environment is incredibly important in ensuring the correct interaction with and execution of the service. For example, does the service require security, and if so, what specific scope and type? Does it support transactions? What outcome protocols are supported? To achieve the promise of an SOA, it is important to extend the current Web service interface and message definitions to include the expression of the constraints and conditions that are associated with the use of a Web service.

Although you can use inherent extensibility of XML and WSDL to achieve some of these requirements, a much better approach is to define a common framework for Web services constraints and conditions that allows a clear and uniform articulation of the available options. Such a framework must enable those constraints and conditions associated with various domains (such as security, transactions, and reliable messaging) to be composable, so that Web service providers and consumers are not burdened

with multiple domain-specific mechanisms. Also, such a framework can provide support for determining valid intersections of constraints and conditions, where multiple choices are possible. Although the programming that is implementing the business logic of the Web service can deal explicitly with the conditions and constraints, providing a declarative model for this factors such issues out of business logic, thereby providing an important separation of concerns. This allows for more automated implementation by middleware and operating systems, resulting in significantly better reuse of application code by the organizations that provide, deploy, and support Web services. The WS-Policy family of Web services specifications provides an extensible framework that is intended to specifically deal with the definition of these constraints and conditions.

The WS-PolicyAttachments specification offers a flexible way to associate policy expressions with Web services. The WS-Policy specification defines a common framework for services to annotate their interface definitions to describe their service assurance qualities and requirements in the form of a machine-readable expression containing combinations of individual assertions. The WS-Policy framework also allows the use of algorithms to determine which concrete policies to apply when the requester, provider, and container support multiple options. WS-Policy is critical to achieving interoperability at the higher-level functional operation of the service. Security, transactions, reliable messaging, and other specifications require concrete WS-Policy schema. This allows services to describe the functional assurance that they expect from and provide to callers.

The WS-Policy specifications are discussed in detail in [Chapter 7](#), "Web Services Policy."



3.5. Discovery Services

The transport, description, and messaging layer are fundamental to allowing Web services to communicate in an interoperable way using messages. However, to facilitate this, it is necessary to collect and store the important metadata that describes these services. The metadata must be in a form that is discoverable and searchable by users who are looking for appropriate services they require to solve some particular business problem. Also, such metadata aggregation and discovery services are a useful repository/registry in which many different organizations might want to publish the services that they host, describe the interfaces to their services, and enable domain-specific taxonomies of services.

3.5.1. UDDI

The Universal Description and Discovery Interface (UDDI) is a widely acknowledged specification of a Web service registry. It defines a metadata aggregation service and specifies protocols for querying and updating a common repository of Web services information. Solutions developers can query UDDI repositories at well-known locations at design time to ascertain those services that might be compatible with their requirements. After they locate a directory, they can send a series of query requests against the registry to acquire detailed information about Web services (such as who provides them and where they are hosted) and bindings to the implementation. They can then feed this information into an assortment of development time tools to generate the appropriate runtime software and messages required to invoke the required service. Solutions can also query UDDI repositories dynamically at runtime. In this scenario, the software that needs to use a service is told at execution time the type of service or interface it requires. Then it searches a UDDI repository for a service that meets its functional requirements, or a well-known partner provides it. The software then uses this information to dynamically access the service.

UDDI repositories can be provided in one of three ways:

- **Public UDDI** These are UDDI repositories that can serve as a resource for Internet-based Web services. An example of this is the UDDI Business Registry [UBR] hosted by a group of vendors led by IBM, Microsoft, and SAP that is replicated across multiple hosting organizations.
- **Intra Enterprise UDDI** An enterprise has a private internal UDDI repository that provides much more control over which service descriptions are allowed to be placed there and used by application developers within that specific enterprise.
- **Inter Enterprise UDDI** This basically scopes the content of the UDDI to services that are shareable between specific business partners.

As discussed in [Chapter 1](#), service discovery (publish/find) plays an important role in an SOA. You can achieve this in other ways, but within a Web services world, UDDI provides a highly functional and flexible standard approach to Web services discovery.

UDDI is covered in detail in [Chapter 8](#), "UDDI."

3.5.2. MetaData Exchange

WS-Policy proposes a framework that extends the service description features that WSDL provides. Having more refined service descriptions, qualified by specific WS-policies, supports much more accurate discovery of services that are compatible with the business application that is to be deployed. In a service registry (such as a UDDI registry), queries of WS-Policy-decorated services enable the retrieval of services that support appropriate policies in addition to the required business interface. For example, a query might request all services that support the creditAuthorization WSDL interface (port type), use Kerberos for authentication, and have an explicitly stated privacy policy. This allows a service requester to select a service provider based on the quality of the interaction that delivers its business contracts.

Although service registries are important components of some Web services environments, it is often necessary to address the request of service information directly to the service itself. The WS-MetaDataExchange specification defines protocols that support the dynamic exchange of WS-Policy and other metadata that is relevant to the service interaction (such as XML Schema and WSDL descriptions) between interacting Web services endpoints. WS-MetaDataExchange allows a service requester to ask a service provider directly for all or part of its metadata, without the involvement of a third-party registry. Using the WS-MetaDataExchange protocol service, endpoints can exchange policies at runtime and use them to bootstrap their interaction from information about the settings and protocols to be applied. This is especially useful when not all policy information is in a repository or when a requester receives a reference to a service through some mechanism other than a direct query on a registry. This direct dynamic exchange of policies also supports the customization of each interaction based, for example, on the identity of the other endpoint or any other aspect of the context under which the interaction takes place. This flexibility allows Web services to be designed to offer different qualities of service for different targeted audiences.

WS-MetaDataExchange is discussed in detail in [Chapter 9](#), "Web Services Metadata Exchange."

3.6. Quality of Service

Specifications in this domain are related to the quality of the experience associated with interaction with a Web service. The services in this layer specify the requirements that are associated with the overall reliability of Web services. The specific issues involving this layer include security, reliability of message delivery, and support for transactions (guaranteeing and agreeing on the outcome of a business application).

3.6.1. WS-Security

Security is of fundamental concern in enterprise computing. WS-Security is the basic building block for secure Web services. Today, most distributed Web services rely on transport-level support for security functions (for example, HTTPS and BASIC-Auth authentication). These approaches to security provide a basic minimum for secure communication, and the level of function they provide is significantly less than that provided by existing middleware and distributed environments. WS-Security uses existing security models (such as Kerberos and X509). The specifications concretely define how to use the existing models in an interoperable way. Multihop, multiparty Web service computations cannot be secure without WS-Security.

Security relies on predefined trust relationships. Kerberos works because participants trust the Kerberos Key Distribution Center. Public Key Infrastructure (PKI) works because participants trust the root certificate authorities. WS-Trust defines an extensible model for setting up and verifying trust relationships. The key concept in WS-Trust is a Security Token Service (STS). An STS is a distinguished Web service that issues, exchanges, and validates security tokens. WS-Trust allows Web services to set up and agree on which security servers they trust, and to rely on these servers.

Some Web service scenarios involve a short sporadic exchange of a few messages. WS-Security readily supports this model. Other scenarios involve long, multimessage conversations between the Web services. WS-Security also supports this model, but the solution is not optimal.

Protocols such as HTTP/S use public keys to perform a simple negotiation that defines conversation-specific keys. This key exchange allows more efficient security implementations and decreases the amount of information encrypted with a specific set of keys. WS-SecureConversation provides similar support for WS-Security. Participants often use WS-Security with public keys to start a conversation or session, and they use WS-SecureConversation to agree on session specific keys for signing and encrypting information.

WS-Federation allows a set of organizations to establish a single, virtual security domain. For example, a travel agent, an airline, and a hotel chain might set up such a federation. An end user who logs into any member of the federation has effectively logged into all of the members. WS-Federation defines several models for providing federated security through protocols between WS-Trust and WS-SecureConversation topologies. In addition, customers often have "properties" when they deal with an enterprise, and WS-Federation allows the setting up of a federated property space. This allows each participant to have secure controlled access to each member's property information about the end users.

The WS-Security family of specifications is discussed in detail in [Chapters 12](#), "Security," and [13](#), "Advanced Security."

3.6.2. Reliable Messaging

In the Internet world, communication channels are typically unreliable. Connections break, messages fail to be delivered or are delivered more than once, and perhaps in a different sequence to that in which they were sent. Communication can become even more of an issue when the exchange of messages spans multiple transport layer connections. Although techniques for ensuring reliable delivery of messages are reasonably well understood and available in some messaging middleware products today (such as IBM WebsphereMQ), messaging reliability is still a problem. If messaging reliability is addressed by Web service developers who are incorporating techniques to deal with this directly into the services they develop, there is no guarantee that developers of different Web services will make the consistent choices about the approach to adopt. The outcome might not guarantee end-to-end reliable interoperable messaging. Even in cases in which the application developers defer dealing with the reliable messaging to messaging middleware, different middleware products from different suppliers do not necessarily offer a consistent approach to dealing with the problem. Again, this might preclude reliable message exchange between applications that are using different message-oriented middleware.

WS-ReliableMessaging addresses these issues and defines protocols that enable Web services to ensure reliable, interoperable exchange of messages with specified delivery assurances. The specification defines three basic assurances:

- In-order delivery The messages are delivered in the same order in which they were sent.
- At least once delivery Each message that is sent is delivered at least one time.
- At most once delivery No duplicate messages are delivered.

You can combine these assurances to give additional ones. For example, combining at least once and at most once gives exactly one delivery of a message. The protocol enables messaging middleware vendors to ease application development and deployment for Web services by providing services that implement these protocols, possibly layered over their existing proprietary message exchange protocols. WS-Reliable Messaging protocols allow different operating and middleware systems to reliably exchange messages, thereby bridging different infrastructures into a single, logically complete, end-to-end model for Web services reliable messaging.

WS-ReliableMessaging is discussed in detail in [Chapter 10](#), "Reliable Messaging."

3.6.3. Transactions

Dealing with many of today's business scenarios necessitates the development of applications that consist of multiple Web services exchanging many messages. An example might be a group of financial institutions setting up a financial offering that involves insurance policies, annuities, checking accounts, and brokerage accounts. Such applications can be complex, executing across heterogeneous, loosely coupled distributed systems that are prone to failure, and introducing significant reliability problems. For such applications, you must deal with the failure of any component Web service of the application within the context of the whole application. A coordinated orchestration of the outcome of the participating services that make up the business application is essential so that a coherent outcome of the whole business application can be agreed upon and guaranteed. Therefore, it is important that the Web services involved are able to do the following:

- Start new tasks, the execution and disposition of which are coordinated with other tasks.
- Agree on the outcome of the computation. For example, does everyone agree that the financial

packages were set up?

WS-Coordination, WS-AtomicTransaction, and WS-BusinessActivity define protocols that are designed specifically to address these requirements.

WS-Coordination is a general mechanism for initiating and agreeing on the outcome of multiparty, multimessage Web service tasks. WS-Coordination has three key elements:

- A coordination context This is a message element that is associated with exchanges during the interaction of Web services. This coordination context contains the WS-Addressing endpoint reference of a coordination service, in addition to the information that identifies the specific task being coordinated.
- A coordinator service This provides a service to start a coordinated task, terminate a coordinated task, allow participants to register in a task, and produce a coordination context that is part of all messages exchanged within a group of participants.
- An interface Participating services can use the interface to inform them of an outcome that all of the participants have agreed upon.

Although WS-Coordination is a general framework and capability, WS-AtomicTransaction and WS-BusinessActivity are two particular protocols that compose with and extend the WS-Coordination protocol to define specific ways to reach overall outcome agreement. They extend this framework to allow the participants in the distributed computation to determine outcome robustly .

WS-AtomicTransaction defines a specific set of protocols that plug into WS-Coordination to implement the traditional two-phase atomic ACID transaction protocols. However, traditional atomic transactions and the WS-AtomicTransaction protocol are not always suitable. For example, this protocol is generally not appropriate for use with many types of business transactions. Transaction protocols for business transactions have to deal with long-lived activities. These differ from atomic transactions in that such activities can take much longer to complete. Therefore, to minimize latency of access by other potential users of the resources being used by Web services participating in the activity, you need to make the results of interim operations visible to others before the overall activity has completed. In light of this, you can introduce mechanisms for fault and compensation handling to reverse the effects of tasks that were completed previously within a business activity (such as compensation or reconciliation). WS-BusinessActivity defines a specific set of protocols that plug into the WS-Coordination model to provide such long-running, compensation-based transaction protocols. For example, although WS-BPEL defines a transaction model for business processes, it is WS-BusinessActivity that specifies the corresponding protocol rendering. This, again, is an example for the composeability of the Web services specifications.

WS-Coordination and Transaction specifications are covered in detail in [Chapter 11](#), "Transactions."



3.7. Service Components

The existing Web services standards do not provide for the definition of the business semantics of Web services. Today's Web services are isolated and opaque. Overcoming this isolation means connecting Web services and specifying how to jointly use collections (compositions) of Web services to realize much more comprehensive and complex functionality typically referred to as a business process. A

business process specifies the potential execution order of operations from a collection of Web services, the data that is shared between these composed Web services, which partners are involved, and how they are involved in the business process, joint exception handling for collections of Web services, and so on. This composition especially allows the specification of long-running transactions between composed Web services. Consistency and reliability are increased for Web services applications. Breaking this opaqueness of Web services means specifying usage constraints of operations of a collection of Web services and their joint behavior. This, too, is similar to specifying business processes.

3.7.1. Composition of Web Services

Business Process Execution Language for Web services (WS-BPEL, often shortened to BPEL) provides a language to specify business processes and process states and how they relate to Web services. This includes specifying how a business process uses Web services to achieve its goal, and it includes specifying Web services that a business process provides. Business processes specified in BPEL are fully executable and are portable between BPEL-conformant tools and environments. A BPEL business process interoperates with the Web services of its partners, whether these Web services are realized based on BPEL or not. Finally, BPEL supports the specification of business protocols between partners and views on complex internal business processes.

BPEL supports the specification of a broad spectrum of business processes, from fully executable, complex business processes over more simple business protocols to usage constraints of Web services. It provides a long-running transaction model that allows increasing consistency and reliability of Web services applications. Correlation mechanisms are supported that allow identifying statefull instances of business processes based on business properties. Partners and Web services can be dynamically bound based on service references.

WS-BPEL is discussed in detail in [Chapter 14](#), "Modeling Business Processes: BPEL."



3.8. Composeability

One of the key guiding principles that has governed the specification of the Web services discussed in this book is that of composeability. Each of the Web service specifications addresses one specific concern and has a value in its own right, independently of any other specification. For example, developers of applications can adopt reliable messaging to simplify development of their business application, use transactions as a method of guaranteeing a reliable outcome of their business application, or use BPEL to define their complex business applications. However, although each of these specifications stands on its own, all the specifications are designed to work seamlessly in conjunction with each other. The term composeable describes independent Web service specifications that you can readily combine to develop much more powerful capabilities. Composeability facilitates the incremental discovery and use of new services; consequently, developers must implement only that which is necessary at any given point in time. The complexity of a solution is a direct consequence of the specific problem that is being addressed.

The basic Web service specifications, WSDL and SOAP in particular, have been designed to support composition inherently. An important characteristic of a Web service is the multipart message structure. Such a structure facilitates the easy composition of new functionality. You can add extra message

elements in support of new services in such a way that does not directly alter the processing of and pre-existing functionality. For example, you can add transaction protocol information to a message that already includes reliable messaging protocol information and vice versa without the protocols conflicting with each other, and in a way that is compatible with the pre-existing message structure.

[Figure 3.2](#) illustrates this by way of a simple Web service message that contains elements associated with three different Web services specifications. Lines 37 are associated with WS-Addressing, lines 814 with WS-Security, and lines 1518 with WS-ReliableMessaging. Each of these elements is independent and can be incorporated and used independently without affecting the processing of other elements present. This enables transactions, security, and reliability of Web services to be defined in terms of composable message elements.

Figure 3-2. Web service message composable.

[\[View full size image\]](#)

```

1 <S:Envelope...>
2 <S:Header>
3 <wsa:ReplyTo>
4 <wsa:Address>http://business456.com/User12</wsa:Address>
5 </wsa:ReplyTo>
6 <wsa:TO>HTTP://Fabrikam123.com/Traffic</wsa:To>
7 <wsa:Action>http://Fabrikam123.com/Traffic/Status</wsa:Action>
8 <wssec:security>
9 <wssec:BinarySecurityToken
10 Value="wssec:x509v3"
11 EncodingType="wssec:Base64Binary"
12 dXJcY3TnYHB...Ujmi8eMTaW
13 </wssec:BinarySecurityToken
14 </wssec:Security
15 <wsrm:Sequence>
16 <wsu:Identifier>http://Fabrikam123.com/seq1234</wsu:Identifier>
17 <wsrm:MessageNumber>10</wsrm:MessageNumber>
18 </wsrm:Sequence>
19 </S:Header>
20 <S:Body
21 <app:TrafficStatus
22 xmlns:env="http://highwaymon.org/payloads">
23 <road>520W</road>
24 <speed>3mph</speed>
25 </app:TrafficStatus>
26 </S:Body>
27 </S:Envelope>

```

A good example of the power of composable Web services in practice is the requirement for service consumers to determine the assurances that are provided by a particular service that they might want to use. This enables the consumer to ascertain whether a particular service meets desired expectations, and if so, in what way. The services must document their requirements in terms of their specific support for transactions, security and reliable messaging, and so on. WS-Policy enables Web services to incrementally augment their WSDL in an independent way to provide a more complete and flexible description of elements that have to be added to the SOAP message to interact with the service successfully.

Team LiB

Team LiB

◀ PREVIOUS

NEXT ▶

◀ PREVIOUS

NEXT ▶

3.9. Interoperability

SOA and Web services promise significant benefits: reduced cost and complexity of connecting systems

and businesses, increased choice of technology suppliers leading to reduced cost of technology ownership, and increased opportunities for businesses to interact both with customers and suppliers in new and profitable ways. The fundamental premise of Web services is that standardization, predicated on the promise of easy interoperability, resolves many of the long-standing issues facing businesses today. However, Web services and the SOAs that are based upon them are an emergent market. As such, the technologies and specifications that various organizations are defining for Web services are in flux. Issues relate to ambiguity of interpretation of specifications or standards, in addition to differences and insufficient understanding of interactions between them. For Web services to be successful, these specifications must be able to truly provide interoperability in a manner that is conducive to running a business or producing products that can effectively leverage Web services technology. The IT leaders behind the Web services specifications realize that interoperability is in the best interests of all industry participants. In 2002, they created the Web Services Interoperability Organization (WS-I).

3.9.1. WS-I

WS-I [WSI] is an open industry organization that is chartered to promote Web services interoperability across differing platforms, operating systems/middleware, programming languages, and tools. It works across the varied existing industry and standards organizations to respond to customer needs by providing guidance and best practices to help develop Web service solutions. Membership of WS-I is open to software vendors of all sizes, to their customers, and to any others who have interests in Web services. The work of WS-I is carried out by the members in WS-I working groups, generally consisting of individuals who have a diverse set of skills (developers, testers, business analysts, and so on). Members can actively participate in one or more WS-I working groups, based on their specific interest or expertise.

WS-I was formed specifically for the creation, promotion, and support of generic protocols for interoperable exchange of messages between Web services. Generic protocols are protocols that are independent of any actions necessary for secure, reliable, and efficient delivery. Interoperable in this context means suitable for and capable of being implemented and deployed onto multiple platforms. Among the deliverables that WS-I is creating are profiles, testing tools, use case scenarios, and sample applications.

A profile consists of a list of Web services specifications at specified version levels, along with recommended guidelines for use, or the exclusion of inadequately specified features. WS-I is developing a set of profiles that support interoperability. Profiles facilitate the discussion of Web service interoperability at a level of granularity for those people who have to make investment decisions about Web services and in particular Web services products. WS-I focuses on compatibility at the profile level. To avoid confusion, it is likely that only a few profiles will be defined. There is already a consensus on those standards that form the most basic Web services profile, and it is likely although not mandatory that as additional profiles emerge, they will indeed be based on this basic profile. In addition to references to industry standards and emerging specifications, a profile might contain interoperability guidelines that can resolve ambiguities. Such guidelines constrain some of the specifications or standard MAYs and SHOULDs, often the source of interoperability problems, such that they become MUSTs or MUST NOTs to satisfy the requirements of the use cases and usage scenarios. The first, or Basic, WS-I profile pertains to the most basic Web services, such as XML Schema 1.0, SOAP 1.1, WSDL 1.1, and UDDI 2.0.

The testing tools monitor and analyze interactions with and between Web services to ensure that exchanged messages conform to the WS-I profile guidelines. Sample applications are being developed to demonstrate the implementation of applications that are built from Web Services Usage Scenarios, which conform to a given set of profiles. Implementations of the same sample application on multiple

platforms, using different languages and development tools, allow WS-I to show interoperability and provide readily usable resources for Web services developers and users.

WS-I is committed to building strong relationships and adopting specifications developed by a wide array of organizations, such as the Internet Engineering Task Force (IETF), Open Applications Group (OAGi), OASIS, OMG, UDDI, W3C, and many others. These organizations serve the needs of a vast range of communities and customer bases. It is the plan of WS-I to engage these groups and work together to meet the needs of Web services developers and customers.



3.10. REST

Despite the name, Web service technology offers several advantages in non-Web environments. For example, Web service technology facilitates the integration of J2EE components with .NET components within an enterprise or department in a straightforward manner. But as shown, Web services can be implemented in Web environments, too, on top of basic Web technologies such as HTTP, Simple Mail Transfer Protocol (SMTP), and so on. Representational State Transfer (REST) is a specific architectural style introduced in [F00]. Simply put, it is the architecture of the Web.

Consequently, the question arises about how Web services compare to the Web, or how the corresponding underlying architectural styles SOA and REST compare.

3.10.1. "Representational" in REST

The basic concept of the REST architecture is that of a resource. A resource is any piece of information that you can identify uniquely. In REST, requesters and services exchange messages that typically contain both data and metadata. The data part of a message corresponds to the resource in a particular representation as described by the accompanying metadata (format), which might also contain additional processing directives. You can exchange a resource in multiple representations. Both communication partners might agree to a particular representation in advance.

For example, the data of a message might be information about the current weather in New York City (the resource). The data might be rendered as an HTML document, and the language in which this document is encoded might be German. This makes up the representation of the resource "current weather in New York City." The processing directives might indicate that the data should not be cached because it changes frequently.

3.10.2. "State Transfer" in REST

Services in REST do not maintain the state of an interaction with a requester; that is, if an interaction requires a state, all states must be part of the messages exchanged. By being stateless, services increase their reliability by simplifying recovery processing. (A failed service can recover quickly.) Furthermore, scalability of such services is improved because the services do not need to persist state and do not consume memory, representing active interactions. Both reliability and scalability are required properties of the Web. By following the REST architectural style, you can meet these requirements.

3.10.3. REST Interface Structure

REST assumes a simple interface to manipulate resources in a generic manner. This interface basically supports the create, retrieve, update, and delete (CRUD) method. The metadata of the corresponding messages contains the method name and the identifier of the resource that the method targets. Except for the retrieval method, the message includes a representation of the resource. Therefore, messages are self-describing.

Identifiers being included in the messages is fundamental in REST. It implies further benefits of this architectural style. For example, by making the identifier of the resource explicit, REST furnishes caching strategies at various levels and at proper intermediaries along the message path. An intermediary might determine that it has a valid copy of the target resource available at its side and can satisfy a retrieval request without passing the request further on. This contributes to the scalability of the overall environment.

If you are familiar with HTTP and URIs, you will certainly recognize how REST maps onto these technologies.

3.10.4. REST and Web Services

At its heart, the discussion of REST versus Web services revolves around the advantage and disadvantages of generic CRUD interfaces and custom-defined interfaces.

Proponents of REST argue against Web service technology because custom-defined Web service interfaces do not automatically result in reliability and scalability of the implementing Web services or cacheability of results, as discussed earlier. For example, caching is prohibited mainly because neither identifiers of resources nor the semantics of operations are made explicit in messages that represent Web service operations. Consequently, an intermediary cannot determine the target resource of a request message and whether a request represents a retrieval or an update of a resource. Thus, an intermediary cannot maintain its cache accordingly.

Proponents of Web service technology argue against REST because quality of service is only rudimentally addressed in REST. Scenarios in which SOA is applied require qualities of services such as reliable transport of messages, transactional interactions, and selective encryption of parts of the data exchanged. Furthermore, a particular message exchange between a requester and a service might be carried out in SOA over many different transport protocols along its message path with transport protocols not even supported by the Web. Thus, the tight coupling of the Web architecture to HTTP (and a few other transport protocols) prohibits meeting this kind of end-to-end qualities of service requirement. Metadata that corresponds to qualities of services cannot in contrast to what REST assumes be expected as metadata of the transport protocols along the whole message path. Therefore, this metadata must be part of the payload of the messages. This is exactly what Web service technology addresses from the outset, especially via the header architecture of SOAP.

From an architectural perspective, it is not "either REST or Web services." Both technologies have areas of applicability. As a rule of thumb, REST is preferable in problem domains that are query intense or that require exchange of large grain chunks of data. SOA in general and Web service technology as described in this book in particular is preferable in areas that require asynchrony and various qualities of services. Finally, SOA-based custom interfaces enable straightforward creation of new services based on choreography.

You can even mix both architectural styles in a pure Web environment. For example, you can use a regular HTTP GET request to solicit a SOAP representation of a resource that the URL identifies and specifies in the HTTP message. In that manner, benefits from both approaches are combined. The

combination allows use of the SOAP header architecture in the response message to built-in quality of service that HTTP does not support (such as partial encryption of the response). It also supports the benefits of REST, such as caching the SOAP response.



3.11. Scope of Applicability of SOA and Web Service

As indicated throughout the first three chapters of this book, Web service technology provides a uniform usage model for components/services, especially within the context of heterogeneous distributed environments. Web service technology also virtualizes resources (that is, components that are software artifacts or hardware artifacts). Both are achieved by shielding idiosyncrasies of the different environments that host those components. This shielding can occur by dynamically selecting and binding those components and by hiding the communication details to properly access those components.

Furthermore, interactions between a requester and a service might show configurable qualities of service, such as reliable message transport, transaction protection, message-level security, and so on. These qualities of services are not just ensured between two participants but between any number of participants in heterogeneous environments.

Given this focus, the question about the scope of applicability of SOA in general and Web service technology in particular is justified. As with most architectural questions, there is no crisp answer, no hard or fast rule to apply. Given this, common sense should prevail. For example:

- SOA is not cost effective for organizations that have small application portfolios or those whose new interface requirements are not enough to benefit from SOA.
- SOA does not benefit organizations that have relatively static application portfolios that are already fully interfaced.
- If integration of components within heterogeneous environments or dynamically changing component configurations is at the core of the problem being addressed, consider SOA and Web service technology. SOA offers potentially significant benefits to organizations with large application portfolios that undergo frequent change (lots of mergers and acquisitions or frequent switching of service providers).
- If reusability of a function (in the sense of making it available to all kinds of requesters) is important, providing the function as a Web service is a good approach.
- Currently, the XML footprint and parsing cost at both ends of a message exchange does take up time and resources. If high performance is the most important criterion for primary implementation, consider the use of Web service technology with care. Use of binary XML for interchange might help this, but currently there are no agreed-upon standards for this.
- Similarly, if the problem in hand is within a homogeneous environment, and interoperation with other external environments is not an issue, Web service technology might not have significant benefit.

[Team LiB](#)[Team LiB](#)[◀ PREVIOUS](#)[NEXT ▶](#)[◀ PREVIOUS](#)[NEXT ▶](#)

3.12. Summary

This chapter provided a high-level overview and understanding of the structure and composition of a Web services platform. This overview was presented as a rendering of the fundamental concepts of service orientation that were introduced in [Chapter 1](#), with a specific set of Web services specifications that IBM, Microsoft, and other significant industry partners developed. This platform represents the basic core of a new Web-based distributed computing platform that overcomes some of the problems of earlier distributed computing technologies. Although the platform is not complete, it certainly forms a viable foundation on which to build additional higher-level, value-added infrastructure and business services. It also seeks to address other aspects of a more complete distributed computing platform. Some additional potential future topics therefore may need to be addressed to provide this more complete platform, and these are covered in [Chapters 17](#), "Conclusion," and [18](#), "Futures."

[Team LiB](#)[◀ PREVIOUS](#)[NEXT ▶](#)