SOA Approach to Integration
XML, Web services, ESB, and BPEL in real-world SOA projects

Matjaz B. Juric
Ramesh Loganathan
Poornachandra Sarang
Frank Jennings

Chapter No. 4
"SOA and Web Services Approach for Integration"
In this package, you will find:
A Biography of the authors of the book
A preview chapter from the book, Chapter NO.4 "SOA and Web Services Approach for Integration"
A synopsis of the book’s content
Information on where to buy this book

About the Authors

Matjaz B. Juric holds a Ph.D. in computer and information science. He is Associate Professor at the University of Maribor and the director of Science Park project. In addition to this book, he has authored/coauthored Business Process Execution Language for Web Services (English and French editions), BPEL Cookbook: Best Practices for SOA-based integration and composite applications development, Professional J2EE EAI, Professional EJB, J2EE Design Patterns Applied, and .NET Serialization Handbook. He has published chapters in More Java Gems (Cambridge University Press) and in Technology Supporting Business Solutions (Nova Science Publishers). He has also published in journals and magazines, such as SOA World Journal, Web Services Journal, Java Developer’s Journal, Java Report, Java World, eai Journal, theserverside.com, OTN, ACM journals, and presented at conferences such as OOPSLA, Java Development, XML Europe, OOW, SCI, and others. He is a reviewer, program committee member, and conference organizer.

Matjaz has been involved in several large-scale projects. He has been consultant for several large companies on the SOA projects. In cooperation with IBM Java Technology Centre, he worked on performance analysis and optimization of RMI-IIOP, integral part of the Java platform. Matjaz is also a member of the BPEL Advisory Board.

Matjaz is author of courses and consultant for the BPEL and SOA consulting company BPELmentor.com. For more information, please visit http://www.bpelmentor.com/.

My efforts in this book are dedicated to my family. Special thanks to Ana and to my friends at the Packt Publishing and University of Maribor.
Ramesh Loganathan has 16 years of Systems Engineering and R&D Management experience in technology-intensive product development organizations, including Sonic Software (Technical Director—India Dev Center), Pramati Technologies (VP, Engineering), and Informix (Principal Engineer). Ramesh has full lifecycle experience setting up and managing product development organizations and motivating high-caliber engineering teams. He has strong insight into Systems software, Middleware Technology, Database internals, Internet Architectures, and frameworks. Ramesh has led engineering efforts building software infrastructure products at Pramati and Sonic Software. After a brief engagement with Sonic/Progress, Ramesh is now VP-Middleware Technologies at Pramati, driving the product direction and setting up a new Technology Consulting business around Middleware Systems.

Ramesh has worked with several organizations in India and in the US including IBM, Lever, Compaq, TCS, Informix, and Integra.

Ramesh is an accomplished Technologist and evangelist, regularly speaking at workshops and seminars. He is active in Tech fora, JCP, and SPEC organizations. He is a member of several Standards Expert groups including J2EE 1.4, and a founding member of ebXMLIndia.org and hyd-eclipse.org. Ramesh is actively engaged with academia and researchers and is an Adjunct Faculty member at IIIT-H teaching two courses on Middleware systems.

Poornachandra Sarang, Ph.D. runs a Software Consulting and Training firm in the name of ABCOM Information Systems (http://www.abcom.com) and is currently an adjunct faculty in the Univ. Dept. of Computer Science at University of Mumbai. Dr. Sarang has previously worked as a Visiting Professor of Computer Engineering at University of Notre Dame, USA and has been a Consultant to Sun Microsystems for several years. Dr. Sarang has spoken in number of prestigious international conferences on Java/CORBA/XML/.NET organized by O'Reilly, SYS-CON, WROX, SUN, Microsoft and others. He has authored several articles, research papers, courseware and books.

Frank Jennings, works in the Information Products Group of Sun Microsystems Inc. He has more than nine years of experience in Java, SOA, and System Design. He is an Electronics Engineer from Madras University and has worked for several open-source projects.

SOA Approach to Integration
XML, Web services, ESB, and BPEL in real-world SOA projects
Integration of applications within a business and between different businesses is becoming more and more important. The needs for up-to-date information that is accessible from almost everywhere, and developing e-business solutions—particularly business to business—requires that developers find solutions for integrating diverse, heterogeneous applications, developed on different architectures and programming languages, and on different platforms. They have to do this quickly and cost effectively, but still preserve the architecture and deliver robust solutions that are maintainable over time.

Integration is a difficult task. This book focuses on the SOA approach to integration of existing (legacy) applications and newly developed solutions, using modern technologies, particularly web services, XML, ESB, and BPEL. The book shows how to define SOA for integration, what integration patterns to use, which technologies to use, and how to best integrate existing applications with modern e-business solutions. The book will also show you how to develop web services and BPEL processes, and how to process and manage XML documents from J2EE and .NET platforms. Finally, the book also explains how to integrate both platforms using web services and ESBs.

What This Book Covers

Chapter 1 is an overview of the challenges in integration and why integration is one of the most difficult problems in application development. We will identify the best strategies for SOA-based integration and discuss top-down, bottom-up, and inside-out approaches. You will learn about different types of integration, such as data-level integration, application integration, business process integration, presentation integration and also, B2B integrations.

Chapter 2 will help you understand what SOA is. You will see that SOA is a very comprehensive enterprise integration paradigm that builds on many existing concepts. Web services standards provide a strong foundation for SOA infrastructure. You will also learn about the Enterprise Services Bus, which is presently one of the leading integration infrastructure options.

Chapter 3 discusses various design anomalies that may arise while designing XML schemas. Some of the broad categories covered in this chapter are design recommendations for architecting domain-specific XML schemas, tips for designing XML schemas with examples, using XSL effectively for translating Infosets from one form to another, securing XML documents with encryption and digital signature, and XML serialization and the differences between SAX, DOM, and StAX.

Chapter 4 discusses the architecture of web services and their benefits. This chapter provides an in-depth coverage of the various patterns that can be applied while creating SOA using web services. You will learn the essential differences between EAI and B2B and how to apply SOA integration techniques in this space. The chapter also discusses several guidelines for creating interoperable web services. Finally, a complete, albeit trivial, example of creating web services on the .NET and Java EE platforms is discussed.

Chapter 5 will familiarize you with the BPEL language, and a process-oriented approach to integration. The characteristics of process-oriented integration architectures are discussed in this chapter. You will learn how to identify business services and service lifecycles. Then the role of executable business processes, which reduce the semantic gap between business and IT, is explained. The chapter introduces the most important technology—BPEL. You will learn about characteristics of BPEL and identify the differences between executable and abstract processes. The basic BPEL concepts and the role of WSDL are discussed.

Chapter 6 takes a look at how ESB provides a concrete infrastructure for SOA, extending the simple services model to include a robust services bus with extensive mediation functionality.
SOA and Web Services Approach for Integration

In Chapter 2, you were introduced to how Service-Oriented Architecture (SOA) can be used for application integration. We saw that the integration itself may be restricted within an Enterprise or may involve third parties (B2B or B2C). Service-Oriented Architectures are complex. Most SOA implementations do not take off because most of the time it is not clear when, where, and how to begin. In this chapter, you will learn many useful tricks and tips to successfully apply SOA techniques for application integration within and outside your enterprise.

You will learn the following in this chapter:

- **Designing Service-Oriented Architectures**: Here, you will first learn the concepts behind Service-Oriented Architectures, why and how SOA helps in building more flexible solutions, followed by the design patterns for SOA and the guidelines for creating them.

- **Designing Web Services**: In this section, you will learn how to create web services for implementing SOA. The various patterns discussed in the SOA section will now be covered in depth in the context of SOA implementation using web services.

- **Differences between B2B and EAI Web Services**: SOA can be used for B2B or EAI kind of applications. In this section, we will study the implications of using web services in these two scenarios.

- **Interoperable WSDL**: A WSDL (Web Services Description Language) document describes the interface to a web service and the binding information. Typically, SOA may be used in a scenario where applications running on disparate platforms need integration. In such cases, the WSDL documents that describe the web services interfaces to different applications must provide compatibility for integration. This section describes how to create interoperable WSDL documents.

Interoperability Challenges in Web Services: An SOA might use several web services deployed on disparate platforms and technologies. These web services must interoperate with each other. Several specifications have come up to achieve this interoperability. In this section, you will study these specifications and their use in creating interoperable web services.

Developing Interoperable Web Services: In this section, we will look at a complete application that shows how to create web services for the .NET and J2EE platforms that interoperate with each other.

Designing Service-Oriented Architectures
Over the last several decades, corporations have developed applications based on a wide range of systems and technologies. Though these systems are componentized, the integration of these systems poses a challenge due to their heterogeneity. The need for integration arises due to globalization and a wide acceptance of e-business. Globalization has increased the competition. The customer pressure keeps continuously increasing to provide a better quality of service. Customer needs change more often due to the offerings made by competitors over the Internet. Today's IT infrastructure must adapt to these demanding changes.

SOA Evolution
In the 1980s, applications were mostly vertical, built to meet the customer requirements in a vertical market segment. The software solutions were sufficient to meet the needs of a vertical industry. For example, an automobile industry never felt the need for interacting with its suppliers by electronic means. The same was true in the case of most other industries. Very rarely there was a need to communicate with other businesses. This is shown in the following figure:

![Vertical 1980s and Earlier](image)
In the late ‘80s and early ‘90s, we saw the need for business applications to grow horizontally to cooperate with business partners. The industry saw the evolution of B2B (Business-to-Business) collaborations through components now spreading across several industry verticals. These components were now distributed giving rise to an extended supply chain, providing customers and business partners access to services. This is illustrated in the following figure.

In today’s world, the way that businesses operate has changed tremendously. Businesses not only want interaction with their partners, but they allow their customers and employees to access their business services electronically. Today, we talk about B2C (Business-to-Customer), whereby customers have a direct access to the services offered by businesses. Exposing the business logic to an untrusted user base poses its own challenges in terms of security, integrity, and so on. Besides, such services must be user friendly and must hide the complexities of the internal business processes from the end customer. This is where the true need for Service-Oriented Architecture is felt. Businesses should offer services rather than an interface to their business logic. The business logic is implemented in several components — exposing the interface to these components results in tight coupling with the business logic. A client application consumes the service through a well-defined interface to the service and does not care about how it is implemented.
Such interactions are depicted in the following figure of today’s complex IT requirements.

The above figure illustrates a typical Travel Agency scenario. A traveler interacts with the travel agency. The travel agency interfaces with several airlines, hotels, and car rental companies. It also interfaces with several banks for online payments, accounting, etc. Each of these organizations in turn interfaces with several other businesses. The total network soon becomes complex. However, this is the requirement of today's businesses and as IT professionals, we are supposed to provide solutions to these demanding requirements.
Chapter 4

IT Evolution

Looking at current business needs, the IT environments in today's world need to be more flexible, and must quickly adapt to the constantly changing business requirements. The applications running on heterogeneous environments must communicate and integrate seamlessly. IT environments have been evolving along the lines of business requirement evolution as illustrated in the following figure.

![IT Evolution Diagram](image)

In the early years of computing, we had only monolithic applications running on stand-alone machines. From the monolithic systems of early '60s, the industry saw the development of structured, client/server, 3-tier, N-tier, distributed systems, and finally the service-oriented architectures of the modern age. The service-oriented architectures attempt to meet today's business requirements. They are loosely coupled, location transparent, and protocol independent. SOA hides the underlying technology architectures from the service consumer. The service implementation may be on a Java EE (earlier J2EE) or .NET platform, or it may even be a legacy application running on an IBM mainframe. The service consumer need not know the platform on which the service is running; the service implementation is totally transparent to the consumer.

While implementing such complex systems based on SOA, the use of patterns plays an important role in success. Patterns provide the solutions to well-known problems solved by others over many years. Patterns at the code and architecture levels have been well documented, well accepted, and almost standardized. The patterns for creating Service-Oriented Architectures (SOA) are still evolving. There are many who have identified and published their findings, but a standard catalog of these patterns is yet to come. In this chapter, we will look at the patterns documented by IBM for creating SOA applications.
Patterns
Patterns are based on the proven successful experiences of the past. The various patterns for e-business as suggested by IBM are shown in the following figure of the hierarchy of patterns.

These patterns are briefly discussed in the following paragraphs. We will discuss a few of the important patterns in the context of SOA in the next section.

Business Patterns
At the top, we start with Business patterns. These define the interface to consumers that include customers, employees, and business partners. The business patterns arrange the various business assets for the interaction with consumers. The following are the four business patterns:

- Self-Service
- Collaboration
- Information Aggregation
- Extended Enterprise

The business patterns provide the most abstract view of the business services to the consumer. They define the interaction between the various business assets to provide a very high-level view of the business service. These patterns are explained in later sections.

Integration Patterns

Sometimes, a single business pattern may not be sufficient to meet the customer requirements. In such cases, we apply integration patterns that tie together multiple business patterns to achieve the desired output. The integration patterns differ from the business patterns in that they do not solve a specific business problem on their own. Rather, they facilitate a more advanced business function by gluing one or more business patterns together. They also help in the feasibility of the composite patterns. The following are the two integration patterns:

- Access Integration
- Application Integration

The Access Integration patterns define how the business services are accessed and the Application Integration patterns define how the applications interoperate with each other. These patterns are discussed in more detail in the next section.

Composite Patterns

Composite patterns combine business and integration patterns. Like other patterns, they provide solutions to recurring problems. There can be numerous combinations of business and integration patterns used to solve a specific business problem. However, problems that recurrently occur across industries can be solved with the use of a specific combination of business and integration patterns and are documented as composite patterns. These are listed below:

- e-Marketplace
- Electronic Commerce
- Portals
- Account Access

For example, creating an e-Marketplace such as ebay.com requires interactions with a wide range of customers. Here, several business services from different industries need to collaborate to provide a business service to an end customer. Such business services involve auction, banking, shipping and delivery, and so on. The composite pattern combines business and integration patterns to achieve this.
Application Patterns
The application patterns define the interaction between the various application components. These are more abstract than the architectural patterns you may have studied elsewhere. In the case of architectural patterns, the interaction between the components is defined. In the case of application patterns, we define the interaction between the applications. Such applications may be internal applications within an enterprise or may involve third-party applications.

Runtime Patterns
Runtime patterns describe the IT infrastructure. They defines the logical middleware components and their interactions with each other. We will discuss the following runtime patterns later in the chapter.

- Direct Connection
- Runtime patterns for Broker

The runtime patterns define the arrangement of nodes and how they connect to each other.

Product Mappings
Finally, the product mappings define the known software products for implementing the runtime patterns. The IBM catalog documents the mappings of IBM's various products such as IBM Websphere, DB2, and so on to implement the runtime patterns. As these are very vendor specific, we will not be discussing these in this book. An interested reader may refer to the IBM site (http://www.ibm.com/redbooks) for further details.

Guidelines
The application of these patterns requires a careful study of a business problem. For example, you may be developing a Portal. You will have to decide who are the users of this portal? What business services need to be offered to these clients, who are the business partners, and what kinds of interactions with the system they require, based on the answers to these questions, you will create a list of business assets needed to provide the services to clients. These assets are then arranged in well-known patterns to achieve the desired business results. This results in creating enterprise architecture. We will not discuss the guidelines for the use of specific patterns here as we discuss them in more depth later in the chapter.
Designing Sound Web Services for Integration

The Web Services technology plays an important role when applying the concepts of Service-Oriented Architectures. The web services technology is based on open standards such as:

- XML — eXtensible Markup Language
- SOAP — Simple Object Access Protocol
- WSDL — Web Services Description Language
- UDDI — Universal Description, Discovery, and Integration

The use of open standards enables the interoperability between different vendor solutions. The existing solutions can be wrapped as web services and new services can be developed without the need to know who the consumer is. The consumer can consume any web service irrespective of the platform on which it is running using the standard web protocols. This enables the just-in-time integration of the applications and allows the business to establish new partners on the fly. Thus, the web services technology is the right candidate for creating SOA.

Web Services Architecture

The web services architecture is shown in the following figure.
A service provider creates the service and publishes it on a UDDI registry for consumers to discover it. A consumer queries the registry and obtains a reference to the service interface from the registry. After the interface is obtained, the consumer creates a programming interface to the service. The consumer then consumes the service using standard SOAP protocols. The request is directed through a web server protected by firewalls to the service provider. This is one way of invoking the service. Another way of invoking the service would be to use a messaging server in place of a web server.

**Web Services Benefits**

The approach of building your SOA with web services as the means of implementation offers several benefits as listed here:

**Self-Contained**

Web services are self-contained in the sense that they do not require any components to be installed on the client side. On the server merely a Servlet engine, an EJB container, or a .NET runtime is required for deploying the service. When the service is deployed and ready to run, a client can consume the service without the need for any software installations on its machine. You can contrast this with other technologies such as DCOM, or RMI where the client stub must be installed on the consumer machine before the client can access the service.

**Self-Describing**

Web services are self-describing. An interface to a (web) service is published through a WSDL document. Such a WSDL document defines the format for the message exchange and the data types used in messages. To consume a service, the client needs to know only about the format and contents of a request and response message.

**Modular**

Web services provide a further abstraction on the existing component technologies based on J2EE, CORBA, DCOM, and so on. Using these various technologies, we create components. The web services compose these components to offer a service to the client. The interface to the components is not exposed to the client. This results in a modular software development resulting in creating a more abstract view of a business service.
Accessible Over the Web
Web services are published, located, and invoked over the Web. Web services use standard web protocols. The service description is published using WSDL; the service is located with the help of a UDDI registry and it is invoked using SOAP. All these protocols are web-based.

Language, Platform, Protocol Neutral
As web services are based on open XML standards, they are language neutral; a client written in any language can access a web service written in any other language. Web services are platform neutral; the consumer and service may be running on two independent platforms. Web services are transport neutral; the service can be invoked using any standard network protocol.

Open and Standards-Based
The web services technology is based on open standards making web services easily interoperable with other web services. These standards are XML-based and are SOAP, WSDL, and UDDI.

Dynamic
Web services can be discovered and consumed at run time, without the need to have any compile-time knowledge of them. In most other technologies, the client needs compile-time knowledge of the component interface. An exception to this is CORBA (Common Object Request Broker Architecture), which provides a DII (Dynamic Invocation Interface) for run-time discovery and invocation of the service. A similar dynamic invocation interface is also available in the Java and .NET platforms.

Composable
Web services can be aggregated into a larger service. As seen in the earlier chapters, we can use orchestration engines for composing web services into a larger service. Such orchestration can be coded using well-accepted BPEL (Business Process Execution Language).

Having considered the benefits of web services, we will now look at the patterns that may be applied while creating web services to implement SOA.
Patterns

In the previous section, we looked at the various patterns for e-business. We will now study these patterns in more depth in the context of an SOA implementation. In an SOA approach, the focus is on creating and reusing loosely coupled services rather than creating coarse-grain applications. The building blocks are services, which can be composed to meet the business needs. Such services are self-contained, modular, and composable into larger services. Applications on the other hand are usually large and inflexible. We will now describe the various patterns in the context of SOA.

Self-Service Business Pattern

The self-service business pattern captures direct interaction between the business user and the service provider. The user may be a customer, an employee, a business partner, or a stake-holder in the company. The service provider is the business that is providing the desired service to the consumer. Thus, the self-service business pattern nicely fits into the SOA paradigm where the main building block is the service.

The architecture of a self-service business pattern is shown in the following figure.

In this pattern, the presentation tier consumes the service provided by the web tier, which in turn consumes the service provided by the back end. This provides the direct interaction between the service consumer and the service provider with the help of services invoked in a chain-like fashion.
Guidelines
To apply the Self-Service business pattern, analyze the business requirement to assimilate what can be offered as a service. An example could be that employees need to look up salary benefits. Usually, these benefits do not get modified over a very long period of time. A list of benefits may be made directly accessible to the employees so as to eliminate repetitive requests made to the payroll department. You may provide a secured web page to employees for viewing the list of benefits. The web page may also provide querying facilities on the benefits database.

Another example is reserving a seat on an airplane. A few years ago, tele-check-in facilities were not available. A traveller had to check-in well in advance at the airline's counter to get a good seat. With the introduction of tele-check-in you can now reserve the seat couple of days in advance. However, this still requires human intervention. Recently, many airlines have started offering this as an online service to the customers. The seating map of the airplane is made directly accessible to the customer over Internet. The customer can select the seat of her or his choice and gets an online confirmation of the reservation. The application of the Self-Service pattern fits perfectly in such a situation.

Extended Enterprise Business Pattern
The previous pattern (Self-Service) provided an interaction between the consumer and the service. In the extended enterprise business pattern, the interaction between the two partner businesses is explored. The collaborative businesses expose their functionality as services. A business can access the service exposed by its partner through a programmatic interface. A business may act as a service consumer, a service provider, or both. The architecture for this pattern is illustrated in the following figure.
This pattern essentially is a manifestation of application integration pattern. However, due to SOA implementation based on open standards, such integration is loosely coupled. The loose coupling also facilitates the integration of disparate technologies. In the case of application integration, the coupling may not be always possible and is usually tightly coupled. The application of SOA also mandates meeting additional QoS (Quality of Service) requirements such as security, performance, and availability.

**Guidelines**

Consider the case of the automobile industry. An automobile manufacturing company depends on several other manufacturers for its spare parts. The company maintains its inventory of spare parts and when it runs low on stocks, a new order is placed with the supplier. A few years ago, the access to the entire inventory was closely guarded and the suppliers had to wait for a physical intimation from the company. A part of the inventory may now be exposed directly to the supplier. The supplier can monitor the inventory levels and supply the goods to the company when a threshold low is reached. Rather than exposing the application interface to the inventory management system, this may be implemented as SOA. The desired service is defined and coded to expose the relevant inventory status to individual suppliers. The Direct Connection pattern for the Extended Enterprise fits perfectly in this scenario. It involves the interaction between the two cooperating (partner) businesses. To expose the inventory to the associated business partners may require a tight implementation of security. Creating secured web services is discussed later in this book.

**Application Integration Pattern**

The application integration patterns capture the best practices in integrating the back end applications and data. They are observed in the EAI (Enterprise Application Integration) space and are helpful in defining process automation and workflows. Process integration help companies connecting applications and its users together within and across enterprise boundaries. Such interactions may be serial or parallel. A serial interaction is classified as a series of 1-to-1 interactions between a source and multiple targets in a time-sequenced manner. In the case of parallel interaction, such interactions between the source and multiple targets are concurrent. Using both serial and parallel together, you can classify the interactions in four categories as follows:
• No Serial, No Parallel — Here the messages are transported on a single path to a single target and this is the simplest connection.
• Serial, No Parallel — A single series of operations is done on multiple targets sequentially.
• No Serial, Parallel — Messages are switched, split, and joined on multiple paths to multiple targets.
• Serial, Parallel — Multiple series of operations are done on multiple targets by splitting and joining.

**Application Integration Patterns**

The patterns for Application Integration are classified further as Process- or Data-focused depending on what they perform. A Process-focused integration pattern defines the functional process flow between the applications and services. A Data-focused integration pattern defines the logical integration of the information or the data used by the applications.

The Process-focused integration patterns are further classified into the following four categories:

• Direct Connection application pattern
• Broker application pattern
• Serial Process application pattern
• Parallel Process application pattern

When applied to the Extended Enterprise domain, these are classified as:

• Exposed Direct Connection application pattern
• Exposed Broker application pattern: Router variation
• Exposed Serial Process application pattern

We will now discuss each of these patterns.
Direct Connection Application Pattern

This is the simplest pattern and defines a 1-to-1 interaction between a pair of applications. These interactions may be complex, which might be broken down into multiple elementary interactions. The pattern addresses these connections. The following figure illustrates this pattern.

The connections may require the application of certain business rules such as data mapping rules, security rules, and so on. The connection may be message or call oriented. These are further classified as synchronous or asynchronous. Generally, the call-oriented connection is synchronous while the message-oriented connection is asynchronous—whether it is synchronous or asynchronous is decided by the integration needs.

In the case of an Extended Enterprise domain, the Exposed Direct Connection application pattern allows the applications to communicate directly across the enterprise boundaries. Thus, this can be applied only in the case of trusted patterns and requires a highly secure channel for communication across the enterprise boundary.

Guidelines

The Direct Connection application pattern maps perfectly into the SOA paradigm. There is a 1-to-1 connection between a service consumer and a provider. The services may be classified based on the functionality and QoS (Quality of Service). The connection rules may be modeled on these factors. A consumer may discover a desired service from the registry. Thus, the application of this pattern fits perfectly in the SOA domain.

The connection may be defined logically rather than physically. This result in creating a Service Bus, which is a subset of the Enterprise Service Bus discussed in an earlier chapter and covered in more depth in Chapter 6.
Broker Application Pattern

The Broker application pattern is based on 1-to-N connections. A single interaction initiated by a source application is distributed across multiple target applications. This is illustrated in the following figure.

This pattern helps in reducing the clutter of point-to-point connections between the applications. Among the several target applications, the applications that require a common interaction with the source application are isolated and grouped together. The interaction rules are defined in the Broker Rules tier. The decomposition and composition of interaction are managed by the broker rules.

The pattern may have another variation based on how the interaction is routed. This is illustrated in the following figure.

In this case, the rules for routing the interaction request are defined in the Router Rules tier. The interaction is routed to a single target application from a logical group of applications. The route is decided by the routing rules.

In the case of an Extended Enterprise domain, this pattern defines the interaction between a source application and the multiple target partner applications.

**Guidelines**

The Broker application pattern facilities SOA. Legacy applications can be wrapped as services. A Service Broker provides a desired service to a service consumer. The type of service and its composition and decomposition are decided at run time providing a very flexible solution in application integration.

**Serial Process Application Pattern**

This is an extension of the broker application pattern discussed in the previous section. In this case, a source application initiates an interaction with multiple target applications as in the case of a broker application pattern. However, the interaction, which is essentially a consumption of a service, now consists of invoking a series of business processes serially in a desired sequence. Basically, it facilitates the orchestration of several business processes for the desired interaction by the source application. This is illustrated in the following figure.

![Serial Process Application Pattern Diagram](image)

The Serial Process application pattern facilitates the separation of process flow logic from the logic of the individual applications. The flow is controlled by the Serial Process Rules tier. These rules not only define the control and data flow rules, but also define the execution rules for each target application. The intermediate results are stored in a WIP (Work-in-progress) database. The execution rules are stored in a registry.
In the case of an Extended Enterprise domain, the pattern defines the interaction of a source application with the series of target partner applications in a pre-determined sequence.

**Guidelines**
The Serial Process application pattern fits perfectly in the SOA paradigm. The businesses provide autonomous services. A consumer requires a series of services to be executed in a desired sequence. The orchestration flow may be defined at run time. The consumer requests a certain kind of business service that is decomposed into smaller services and executed sequentially on a set of target applications.

An example of this can be a travel reservation. A traveller desires a hotel and a car reservation besides her air travel reservation. These services are offered, obviously, by different business. Hotel, car, and airlines reservations operate independently of each other. The traveller initiates a reservation request for desired travel dates. The request is split into multiple service requests that are executed sequentially by partner businesses. When all the partner requests are processed, the reservation confirmation or non-availability is communicated to the consumer.


**Parallel Process Application Pattern**
This is an extension of the Serial Process application pattern where the sub-processes are executed concurrently on multiple targets. This is illustrated in the following figure.
The concurrent execution of sub-processes requires a more sophisticated execution engine. The processes must be split, executed, and joined properly. The final result depends on the success/failure of each process. The consumer request may require an intermediate splitting and joining of a business operation. In such cases, the intermediate results dictate the further execution of the consumer request. Thus, the execution unit shown in the Parallel Process Rules tier can be very complex.

**Guidelines**

This pattern is observed in the implementation of a BPEL engine. Considering our example of a traveller in the previous section, the reservation request for a hotel, car and airlines may be executed in parallel as these are independent of each other. Depending on the outcome of each reservation, the ultimate confirmation or denial is communicated to the requester. The implementation of this pattern is best done with the use of a commercial off-the-shelf orchestration engine. Such engines typically use BPEL for creating business processes. The use of BPEL results in ease of defining and modifying the processes. Applying this pattern on custom-built application components is too complex and should be avoided.

**Runtime Patterns**

The application patterns discussed so far overlay the runtime patterns. The runtime pattern uses a set of nodes to group functional and operational components. The nodes are interconnected to form a pattern. The business logic of the application is deployed on nodes.

**Nodes**

We consider the following definitions for nodes in the context of runtime patterns:

**Application Server/Services**

An application server may consist of a servlet or EJB container. It typically hosts web applications and does not generally support HTTP connections. The HTTP requests from the presentation tier are redirected through a web server redirector.

**Rules Repository**

The rules repository, as the name suggests, stores the rules for controlling the mode of operation of an interaction. The rules, as seen earlier, may consist of data mapping rules, security access rules, availability rules, and so on. The use of a repository node is optional.
Chapter 4

Router
The Router node is similar to the Broker node discussed earlier. It is responsible for routing a request to one of the appropriate target nodes. The router rules provide separation of the application logic from the distribution logic.

Protocol Firewall Node
A firewall controls the flow of information between an internal secured application and an external unsecured consumer. A firewall controls the traffic flow with the help of filters. Though generally this is considered as a first line of defence, it may be combined with comprehensive security systems that provide message encryption, content filtering, and intrusion detection. A firewall may be of two types: a protocol firewall or a domain firewall. A protocol firewall is a typical IP router.

Domain Firewall Node
A domain firewall is implemented generally as a dedicated server node. With the help of a domain firewall, we create a demilitarized zone for added security.

Connectors
In addition to the nodes, some connector definitions are important to us in the context of runtime patterns. A connector facilitates the interaction between two components. Depending on the required level of detail, a connector may be classified as follows:

- **Primitive connector**—represents a simple connection between the two components. This is considered an un-modeled connector as it does not provide any functionality other than a simple connection between the components.
- **Component connector**—provides an additional functionality in the connection between two components. This is also called a modeled connector.

A connector may be an Adapter connector, Path connector, or both.

Adapter Connector
An Adapter connector contains some business logic that transforms the messages and the data between the consumer and the source blocks to match the data and protocol requirements of each side. Thus, it enables the logical connectivity between the source and target components.

Path Connector
A Path connector provides a physical connectivity between the components. It may be as simple as a wired connection between the components or may be as complex as an Internet.

We will now study runtime patterns for integration.

Direct Connection Runtime Pattern
The Direct Connection pattern is depicted in the following figure.

In this pattern, a source application connects directly to a target application using a connector. The connector itself may be explicitly or implicitly modeled. Depending on the connector and interaction variation, the connector may be classified as follows:

- Adapter Connector
- Path Connector
- Message Connector
- Call Connector
- Call Adapter Connector
The Adapter and Path connectors are explained previously. The connector is called a message connector whenever we use messaging services for connection. The Call Connector indicates a direct call to the service, while a Call Adapter Connector indicates a call through an adapter.

The source and target applications are modeled using the Application Server/Services node. The Rules Repository and Domain QoS Providers are optional and need not be shown in the pattern artefact. As discussed earlier, the Rules Repository contains the rules for connection. The QoS Provider defines the various Quality-of-Service attributes for the connection.

In SOA, a rules repository may be implemented as a service registry.

This pattern provides a direct connection between a service consumer and a service provider with the help of connectors. The pattern is classified into the following categories depending on the connector used:

- Single Adapter connector
- Coupling Adapter connector
- Service Bus

Each of these is discussed further.

**Direct Connection Pattern using Single Adapter**

This pattern is depicted in the following figure.
The Basic Direct Connection pattern uses a single adapter to connect a service consumer to a service provider. The adapter provides message and data transformation to match the different protocol requirements of the consumer and the provider. This is a very important pattern in the implementation of service-oriented architecture. This is typically used for providing a service-oriented interface to a legacy application. The Rules Repository node shown in the pattern models the service directory. The consumer looks up the directory to discover services and select an appropriate service to use.

**Direct Connection using Coupling Adapter**

This pattern is depicted in the following figure.

In this case, multiple adapters are coupled to achieve the desired transformation. This improves the adapter reusability in multiple point-to-point connections. The coupled adapters together support the transformation of request and response between the consumer and the provider.
Direct Connection using Service Bus

This pattern is depicted in the following figure.

In this case, we assume that the various Application Server nodes are connected using a common Service Bus. A Source application connects to a desired Target application using this service bus. A source application may connect to more than one target application as depicted in the previous figure. Each connection may use a different connection pattern.

In the pattern diagram, the model adapter connectors and connection rules node are not shown. This is to emphasize the use of the service bus. The service bus minimizes the number of adapters required for each point-to-point connection and is an extension of the Direction Connection with coupling adapter runtime pattern.
Along with the service bus, the adapter connectors may be explicitly modeled as shown in the following figure.

Here, the service bus is said to be of **X-type** as each of the application services connects to this X-type connector. An example of such an X-type service bus could be an HTTP service bus or a JMS (Java Messaging Service) service bus. The X-type adapter connectors bridge the service consumers and providers of different types to the underlying service bus. The service bus itself may span across multiple tiers, and may even cross enterprise boundaries.

A rules repository node may be added to the above pattern to enable the consumers to search for the services with desired characteristics. Such services may be offered within the enterprise or outside the enterprise. The service bus shown is a subset of the Enterprise Service Bus discussed in Chapter 6.

**Runtime Patterns for Broker**

As in the case of application patterns discussed earlier, a broker may be introduced in the runtime nodes, which will act as a message distributor. A source application connects to many target applications through a broker. A typical configuration for this pattern that uses a router to connect to multiple target applications is shown in the following figure.

As in the case of application patterns, the router defines the rules for routing the consumer requests to one of the target applications. During the call, the router converts the transport protocols between the consumer and the provider. It also transforms the message formats between the two parties.

Another variation of this pattern is shown in the following figure.
In this case, the router connects to the service bus and is responsible for transforming a service request from one protocol to another. For example, an HTTP call may be converted to a JMS call or an RMI-IIOP call.

Again, a service registry may be implemented by creating a Rules Repository node for clients to discover the services.

Having studied the various business and application patterns for SOA implementation, we will now look at the implementation of SOA in B2B and EAI domains.

Differences between B2B and EAI Web Services
Let us first look at the differences between B2B and EAI. The major differences may be listed as follows:

1. EAI as the name suggests acts within an enterprise to solve a local problem, while B2B as the name suggests acts across the enterprises.
2. EAI aims at integration of application and data sources within an enterprise, while businesses integrate for purposes such as a supply chain or collaborating on a common product design.
3. B2B mandates implementations of community management, user profile management, and sophisticated security management, while such services are not required for EAI.
4. B2B may require a deep support for standards such as OBI (Open Buying on Internet), XML, cXML (Commerce XML), and EDI (Electronic Data Interchange), while EAI has no requirement for such standards.
5. The connectivity to a single application in EAI is relatively small, while in the case of B2B the number of partner connections can be large. Also, the connectivity is unpredictable in B2B.

Having considered the differences between EAI and B2B, let us look at the differences in SOA implementation in the two cases.
Interface Design

As EAI is within an enterprise, the restrictions on the interface design can be relaxed as compared to B2B. You may use the SOA approach for exposing the services. However, you may decide not to use standard web protocols while implementing SOA. The protocols may be totally proprietary if it eases the integration. Secondly, the protocols don’t need to be web-based as there may not be a need to access the service over the Web. An example of one such protocol is OpenEAI Message Protocol. This is a messaging protocol that expresses all actions on enterprise data objects in terms of request/reply and publish/subscribe messaging models. It also includes administrative information required for implementing security, routing, logging, and auditing. Other protocols are Omri and Indigo; these are recognized by Microsoft for B2B applications.

Compare this with the B2B situation in which the interface has to be exposed using standard web protocols as the service will be invariably accessed over the Web potentially even by unknown users. Even in the case of known users, there could be differences in the platform and technologies used by the consumer and the server. Thus, the use of standard web protocols becomes a mandatory requirement in order to make your service universally accessible.

The other factor that needs to be considered during the interface design is the security implementation. The messages must be protected and the data integrity must be guaranteed. This is a mandatory requirement in case of B2B, while in case of EAI this may be relaxed. The security can be easily compromised in the case of EAI as the interaction is restricted only within the organization. As a matter of fact, in most of the EAI situations, the security is totally discarded. This is mainly due to the complexities involved in implementing and managing security. Also, the secured channel reduces the system performance.

Now, we will look at the need for and use of a service registry in these two cases.

Use of a Service Registry

The service registry stores the information about the various services. In the case of EAI, as the services are offered and consumed locally, the use of a service registry is not recommended. Only in the case of large enterprises, where the number of services could be large, is a service registry suggested. If the number of services is small, it will be easier to publish them by other means such as paper or electronic documents rather than storing them in service registries.
In the case of B2B, the use of service registries becomes a mandatory requirement. In a B2B situation, it is important that the business must make its services publicly known. An unknown consumer can look up the service registry for a desired service. Once a desired service is identified, the consumer can obtain its interface and bind to the service provider to consume the service.

Using a service registry also requires an additional effort in coding the client applications and managing the registry. It also results in additional processing time. In the case of EAI as the services are available locally, these overheads are unjustifiable. In the case of B2B, there is no option other than to bear these overheads.

Writing Interoperable WSDL Definitions

As seen from the discussions in the earlier sections, web services technology can easily be used to implement SOA and to integrate applications running on different platforms. Every platform has its own data representation format and data type system. A web service must provide a universally accepted data type system to take care of the disparities in data types of various platforms. WSDL too, which is a grammar to describe the web service interface, must support interoperability. Although WSDL is not a mandatory requirement in the implementation of web services, it is widely supported. Thus, it is very important for us to understand how to create interoperable WSDL. To create an interoperable WSDL, the developer needs to create a WSDL that is compliant with the Basic Profile defined by WS-I (Web Service Interoperability Organization). The Basic Profile is discussed in the next section. The problem is that in many cases WSDL is created easily with the vendor tools and this WSDL may not truly comply with the Basic Profile.

To create a WSDL compliant to the WS-I Basic Profile, you will need to code it by hand and then verify it with the WS-I provided tools. Writing WSDL by hand is not only time consuming but it is also error prone. Thus, most times it is easier to use vendor-specific auto-generated WSDL. This WSDL may then be modified to remove any platform-specific idiosyncrasies.

The following Listing gives a template for WSI Basic Profile-compliant WSDL. You may use this template to easily create a compliant WSDL for your web services.

```xml
<?xml version="1.0" encoding="utf8"?>
<wsdl:definitions targetNamespace="http://www.mycompany.com"
 xmlns:tns="http://www.mycompany.com"
 xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
<wsdl:types xmlns:xsd="http://www.w3.org/2001/XMLSchema" >
<xsd:schema elementFormDefault="qualified"
 http:="http://www.mycompany.com"
```
targetNamespace=" ">
  <xsd:element name="MyElement1" type="tns:Element1Type"/>
  <xsd:complexType name="Element1Type">
    <xsd:sequence>
      <xsd:element name="First" type="xsd:int"/>
      <xsd:element name="Second" type="xsd:int"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:element name="MyElement2" type="tns:Element2Type"/>
  <xsd:complexType name="Element2Type">
    <xsd:sequence>
      <xsd:element name="First" type="xsd:int"/>
      <xsd:element name="Second" type="xsd:int"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
</wsdl:types>
<wsdl:message name="InputMessage">
  <wsdl:part name="InputDocument" element="tns:MyElement1"/>
</wsdl:message>
<wsdl:message name="OutputMessage">
  <wsdl:part name="OutputDocument" element="tns:MyElement2"/>
</wsdl:message>
<wsdl:portType name="MyWebServicePortType">
  <wsdl:operation name="requestResponseMyServiceOperation">
    <wsdl:input message="tns:InputMessage"/>
    <wsdl:output message="tns:OutputMessage"/>
  </wsdl:operation>
  <wsdl:operation name="oneWayOperation">
    <wsdl:input message="tns:InputMessage"/>
  </wsdl:operation>
</wsdl:portType>
<wsdl:binding name="MyWebServiceSoap" type="tns:MyWebServicePortType"
  xmlns:http="http://schemas.xmlsoap.org/wsd1/http/*"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:mime="http://schemas.xmlsoap.org/wsd1/mime/">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
    style="document"/>
  <wsdl:operation name="requestResponseMyServiceOperation"/>
The template shown in the above listing contains a single request/response operation and a single one-way operation. The template also defines two types and the two corresponding elements MyElement1 and MyElement2. You may modify the template and add more operations and types as required by your service. Using this template, you can now easily create an interoperable WSDL that is compliant to the Basic Profile by following the simple steps listed next:

1. Replace all occurrences of MyWebService with the name of your web service.
2. Replace all occurrences of http://www.mycompany.com with the namespace of your service.
3. Define the complex data types required by your service by modifying the code shown below. Assign an appropriate name for your data type and create the desired sequence of data types for your desired new complex data type.

```xml
<xsd:element name="MyElement1" type="tns:Element1Type"/>
<xsd:complexType name="Element1Type">
  <xsd:sequence>
    <xsd:element name="First" type="xsd:int"/>
    <xsd:element name="Second" type="xsd:int"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:element name="MyElement2" type="tns:Element2Type"/>
<xsd:complexType name="Element2Type">
  <xsd:sequence>
    <xsd:element name="First" type="xsd:int"/>
    <xsd:element name="Second" type="xsd:int"/>
  </xsd:sequence>
</xsd:complexType>
```

4. Define input and output messages for your service. We assume a document-centric service here. Assign the desired name for the messages and select the appropriate data types. You can do so by replacing the attribute values for the name, part, and element tags in the code below:

```xml
<wsdl:message name="InputMessage">
  <wsdl:part name="InputDocument" element="tns:MyElement1"/>
</wsdl:message>
<wsdl:message name="OutputMessage">
  <wsdl:part name="OutputDocument" element="tns:MyElement2"/>
</wsdl:message>
```

5. In the portType tag you will need to set the operations required by your web service. This may be one-way or request/response type. In the code below two operations are shown. The first one is of type request/response and the second one is of type one way. Modify this code to assign the desired names for the operations and assign the appropriate input and output messages defined earlier in your WSDL schema.

```xml
<wsdl:portType name="MyWebServicePortType">
  <wsdl:operation name="requestResponseMyServiceOperation">
    <wsdl:input message="tns:InputMessage"/>
    <wsdl:output message="tns:OutputMessage"/>
  </wsdl:operation>
  <wsdl:operation name="oneWayOperation">
    <wsdl:input message="tns:InputMessage"/>
  </wsdl:operation>
</wsdl:portType>
```
6. Assign the desired name and type for the binding in the \texttt{wsdl:binding} tag.
\begin{verbatim}
<wsdl:binding name="MyWebServiceSoap"
    type="tns:MyWebServicePortType"
\end{verbatim}

7. In the operation tag set the desired operation name, specify the desired soapAction and the input and output by modifying the following lines of code. This code represents the request/response type of operation.
\begin{verbatim}
<wsdl:operation name="requestResponseMyServiceOperation">
    <soap:operation
        soapAction="http://www.mycompany.com/OutputMessage"
        style="document"/>
    <wsdl:input>
        <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
        <soap:body use="literal"/>
    </wsdl:output>
</wsdl:operation>
\end{verbatim}

8. For one-way operation, make modifications similar to the previous bulleted item in the code lines below:
\begin{verbatim}
<wsdl:operation name="oneWayOperation">
    <soap:operation
        soapAction="http://www.mycompany.com/InputMessage"
        style="document"/>
    <wsdl:input>
        <soap:body use="literal"/>
    </wsdl:input>
</wsdl:operation>
\end{verbatim}

9. Finally, modify the following line in the service tag to specify the URL for your service.
\begin{verbatim}
<soap:address location="http://www.yourCompany.com/WebApplication1/MyWebService.asmx"/>
\end{verbatim}

**Validating Interoperable WSDL**

As we have hand-coded the interoperable WSDL in the previous section, there is no guarantee that it can be implemented as the platforms may not support all the features specified in the WSDL. To ensure that the WSDL is valid and can be implemented use the following steps:
1. Create a test web service for your platform. While creating a test web service, implement all the web service’s methods by creating a test code. Ensure that all the input and output documents are covered in the test code.

2. Create client code for testing your web service. Call all the service methods of the test service. Ensure that all the defined messages are exchanged between the client and the web service. Validate the results of each service call.

3. Create a test client on another platform and do a similar testing to that suggested in step 2. If the platform choice for your web service is Java EE, then create a client on the .NET platform to verify the interoperability. If your test web service is on the .NET platform, use Java for creating a test client. Minimally, test the interoperability between these two popular platforms, .NET and Java EE.

4. Now, create a test web service for the other platform and do a cross testing across different platforms as in steps 2 and 3 above.

5. If any of these tests require you to modify the WSDL, do so and iterate through all the steps above until your WSDL is finalized.

6. Once you thoroughly test the implementation of your WSDL, publish it in a UDDI directory.

**Interoperability Challenges in Web Services**

As seen clearly from the ongoing discussions so far, the introduction of web services provided an elegant solution to integrate the diverse applications existing in this world. Before the introduction of web services, several technologies were available for remote invocation of services. These included, but were not limited to, DCE (Distributed Computing Environment), RPC (Remote Procedure Calls), CORBA IIOP (Common Object Request Broker Architecture Internet InterOperable Protocol), Java RMI (Remote Method Invocation), and Microsoft DCOM (Distributed Common Object Model). Web services differ from these technologies on the following grounds:

- Programming Language Independence
- Platform Independence
- Broad Industry Support

These benefits provided a great business value as the applications written in one programming language could now be easily accessed over the web by a client written in an altogether different language. Also, the existing applications need not be reengineered; they simply need to be wrapped as a web service.
Thus, with the introduction of web services, everybody jumped on the bandwagon quickly and as a result the implementations did not quite adhere to the specifications. Thus, we had several web services that were aimed at interoperability, but did not meet the requirements as the underlying protocol implementations did not match the specifications one hundred percent. The specifications are a number of XML standards, which are widely supported. These are:

- SOAP (Simple Object Access Protocol), used for invoking web services.
- WSDL (Web Services Description Language), used for describing the interface to a web service.
- UDDI (Universal Description, Discovery, and Integration), a service registry for publishing information about web services.

The above specifications came into existence at the beginning of this century. The first one was SOAP. SOAP 1.1 was published as a W3C note on Nov 8, 2000. It provided an envelope to encapsulate the following:

- Application messages
- Encoding rules for data types
- Conventions for representing remote procedure calls

The WSDL 1.1 specification was published on March 15, 2001. WSDL provided a convention for defining application messages, operations, and bindings to SOAP, HTTP, and MIME. The UDDI version 2 was published in July 2002.

Due to a sudden rush to implement these specifications, especially the most important one, which is SOAP, the market saw the SOAP specification published even before the finalization of the XML schema. The then published SOAP specification had its own type system called SOAP Encoding. Eventually, when the XML schema specifications were published, SOAP specifications needed modifications.

WSDL had its own problems. WSDL is more suitable for machines than human-beings. WSDL is difficult to understand and implement due to the many layers of abstraction defined in it.

UDDI comparatively suffered less than these other two specifications. The use of a service registry is not mandatory for implementing and using web services. Thus, this specification was the last one to get published and thus faced lesser issues on interoperability.

The market rushed to implement these various specifications from the notes published by W3C even before the standardization work was completed.
WS-I Specifications

To resolve these issues, finally the Web Services Interoperability Organization (WS-I) was formed in 2002 with the help of SAP, IBM, Microsoft, and others. The main purpose of WS-I was to bring the vendors and customers together to resolve the issues of interoperability. A working group called the Basic Profile WG was formed for this purpose. The working group delivered the Basic Profile (BP) 1.0 in August 2003.

The SOAP Encoding was disallowed in a BP-compliant web service. The committee also published a BP-conformant real-world Supply Chain Management application. It also identified and documented common usage patterns of web services. It also published a test tool for verifying service artefacts for BP conformance. The success of BP 1.0 was demonstrated by a sample application created with the joint efforts of 10 different vendors including BEA, IBM, Microsoft, Oracle, SAP, and SUN.

We will now discuss different versions of the Basic Profile and the modifications made in each of these.

WS-I Basic Profile 1.0

The key features of BP 1.0 are as follows:

- Use of SOAP, a lightweight XML-based messaging protocol. SOAP is used for transferring information through web service request and response messages.
- SOAP encoding is prohibited. The use of XSD data types is mandated.
- The use of HTTP binding with SOAP is necessary.
- Requires the use of HTTP 500 status response for faulty SOAP messages.
- The HTTP POST method must be used instead of any other HTTP methods.
- WSDL 1.1 specification is used to describe the web service interface.
- Necessitates the use of rpc/literal or document/literal forms of WSDL SOAP binding.
- Prohibits the request-response and notification style operations.
- WSDL SOAP binding extension with HTTP is used as a transport rule.
- Requires WSDL descriptions for UDDI.

The above list essentially summarizes the key features of BP 1.0. Thus, to create an interoperable web service, you must adhere to the listed rules to make the service BP 1.0 compliant. Failing this, the service may not interoperate with other services.
SOA and Web Services Approach for Integration

WS-I Basic Profile 1.1

WS-I Basic Profile 1.1 brought about further changes in the specifications. These are summarized below:

SOAP Changes

In SOAP, the following changes were made:

- Disallowed constructs:
  - An ENVELOPE SHOULD NOT contain the namespace declaration xmlns:xml="http://www.w3.org/XML/1998/namespace".
  - A DESCRIPTION SHOULD NOT contain the namespace declaration xmlns:xml="http://www.w3.org/XML/1998/namespace".

- Attributes on SOAP1.1 elements:
  - The soap:Envelope, soap:Header, and soap:Body elements in an ENVELOPE MUST NOT have attributes in the namespace "http://schemas.xmlsoap.org/soap/envelope/".

- SOAP action HTTP header:
  - A RECEIVER MUST NOT rely on the value of the SOAPAction HTTP header to correctly process the message.

WSDL Changes

In WSDL, following changes were made:

- XML Namespace declarations:

- WSDL documentation element:
  - In a DESCRIPTION the wsdl:documentation element MAY be present as the first child element of wsdl:import, wsdl:part and wsdl:definitions in addition to the elements cited in the WSDL1.1 specification.
• **Bindings and Parts:**
  ○ An **ENVELOPE MUST** contain exactly one part accessor element for each of the \texttt{wsdl:part} elements bound to the envelope's corresponding \texttt{soapbind:body} element.
  ○ In a doc-literal description where the value of the parts attribute of \texttt{soapbind:body} is an empty string, the corresponding **ENVELOPE MUST** have no element content in the \texttt{soap:body} element.
  ○ In an rpc-literal description where the value of the parts attribute of \texttt{soapbind:body} is an empty string, the corresponding **ENVELOPE MUST** have no part accessor elements.

• **Part Accessors:**
  ○ The part accessor elements in a MESSAGE described with an rpc-literal binding **MUST** have a local name of the same value as the \texttt{name} attribute of the corresponding \texttt{wsdl:part} element.

### WS-I Basic Profile 1.2

The following changes were made in BP 1.2:

### SOAP Changes

• **XML Envelope Serialization:**
  ○ An **ENVELOPE MUST** be serialized as XML 1.0.

• **Unicode BOMs:**
  ○ A **RECEIVER MUST** accept envelopes that include the Unicode Byte Order Mark (BOM).

• **XML declarations:**
  ○ A **RECEIVER MUST** accept messages with envelopes that contain an XML Declaration.

• **Character Encodings:**
  ○ A **RECEIVER MUST** ignore the encoding pseudo-attribute of the envelope's XML declaration.
- **SOAP Envelope Structure:**
  - An ENVELOPE MUST conform to the structure specified in SOAP 1.1 Section 4, “SOAP Envelope”.
  - An ENVELOPE MUST have exactly zero or one child elements of the soap:Body element.

- **SOAP Defined Faults Action URI:**
  - An ENVELOPE MUST use the http://www.w3.org/2005/08/addressing/soap/fault URI as the value for the wsa:Action element when present, for either of the SOAP1.1 defined VersionMismatch and MustUnderstand faults.

- **SOAP MustUnderstand or VersionMismatch fault Transmission:**
  - A RECEIVER that receives a SOAP envelope that generates either a SOAP MustUnderstand or VersionMismatch fault SHOULD transmit such a fault on the HTTP response message, regardless of the value of the wsa:ReplyTo or wsa:FaultTo SOAP headers present in the message.

- **Use of wsa:Action and WS-Addressing WSDL Binding:**
  - An ENVELOPE that includes a wsa:Action SOAP header block and which is described using WSDL 1.1 description MUST conform to WS-Addressing WSDL Binding, Section 5.1.

- **Understanding WS-Addressing SOAP Header Blocks:**
  - When a message contains multiple WS-Addressing SOAP header blocks with at least one of those header blocks containing a soap:mustUnderstand='1' attribute, then a RECEIVER MUST understand all the WS-Addressing SOAP header blocks or none of them.

- **Valid Range of Values for SOAPAction when WS-Addressing is used:**
  - When wsa:Action MAP is present in an envelope, the containing MESSAGE MUST specify a SOAPAction HTTP header with either a value that is an absolute URI that has the same value as the value of the wsa:Action MAP, or a value of "" (empty string).
Use of Non-Anonymous Response EPR in a Request-Response Operation:

- If an INSTANCE sends a MustUnderstand or VersionMismatch fault generated as a result of an invocation of a Request-Response WSDL operation, it MUST send that fault in the entity body of HTTP response using the same HTTP connection as the request message of that operation.
- If an INSTANCE sends a response, which is neither a MustUnderstand nor VersionMismatch fault, as a result of an invocation of a Request-Response WSDL operation and the response EPR has a non-anonymous wsa:Address value, then the response MUST be sent in the entity body of an HTTP request in a separate HTTP connection specified by the response EPR using the SOAP 1.1 Request Optional Response HTTP binding.

WSDL Changes

- WSDL and Schema Import:
  - In a DESCRIPTION, the namespace attribute of the wsdl:import MUST NOT be a relative URI.
- WSDL documentation Element:
  - In a DESCRIPTION the wsdl:documentation element MAY be present as the first child element of wsdl:import, wsdl:part and wsdl:definitions in addition to the elements cited in the WSDL1.1 specification.
- Multiple GED Definitions with the same QName:
  - A DESCRIPTION SHOULD NOT contain multiple global element declarations that share the same qualified name.
- Multiple Type Definitions with the same QName:
  - A DESCRIPTION SHOULD NOT contain multiple type definitions that share the same qualified name.

WS-I Basic Security Profile 1.0

The Basic Security Profile was created to address the interoperability issues of secured web services. The profile addresses several key areas listed next:

- Transport Layer Security
- SOAP Message Security
• Username Token Profile
• X.509 Certificate Token Profile
• XML-Signature
• XML Encryption, Algorithms
• Relationship of Basic Security Extension Profile to Basic Profile
• Attachment security

The security profile does not completely guarantee interoperability. However, it addresses the most common problems experienced in practical implementations to increase the probability of interoperability.

The focus is laid on the interoperability characteristics of two main technologies:

• HTTP over TLS—technology that protects the confidentiality of all information that flows over an HTTP connection
• SOAP Message Security

It does not prohibit the use of any encryption algorithms; however, it recommends some TSL & SSL cipher suits.

It is a requirement that the partners exchanging the messages must agree on the following:

• Which elements must be signed and/or encrypted
• Which elements may be signed and/or encrypted
• Which security tokens must be present
• Which security tokens may be present

The profile puts the following conditions on the applications:

• The Envelope, Header, or Body elements must not be encrypted. Encrypting these elements breaks the SOAP processing model and is therefore prohibited.
• A SOAP intermediary INSTANCE MUST NOT remove or modify any HEADER_ELEMENT unless that SOAP intermediary is acting in the role specified by the S11:actor attribute of that HEADER_ELEMENT.
• Messages may be signed and encrypted, potentially by multiple entities signing and encrypting overlapping elements. A signature applied before encryption has different security properties than encryption applied before a signature.
• SOAP Message Security defines a Timestamp element for use in SOAP messages. (Time stamp must contain only one CREATED & EXPIRES element)
Thus, to create interoperable secured web services, the conditions just listed must be satisfied. Note that the list is by no means complete, and the reader is referred to the WS-I site (http://www.ws-i.org) for full coverage of the security profile. The previous discussions merely give an overview of what is required to create secured interoperable web services.

Guidelines for Creating Interoperable Web Services

Fundamentally, web services are interoperable. Thus, regardless of the client's hardware and software, it should be able to run a web service. The functionality of the web service should remain independent of the following:

- Application platform such as Weblogic server, SunOne App Server, .NET Server, and so on
- Programming language such as Java, C++, C#, Visual Basic
- Hardware such as PC, PDA, Mainframes
- Operating systems such as Unix, Linux, Windows, and so on
- Application data models

However, we have seen previously that due to differing implementations of the specifications by different vendors, some web services may not correctly interoperate with others. To create interoperable web services the following tips may be useful.

Avoid using Vendor-Specific Extensions

Some vendors may extend certain specifications such as SOAP and WSDL. Avoid using such extensions in your applications.

Use the Latest Interoperability Tests

WS-I publishes the tools for interoperability tests. Use the latest version of these tools while testing for interoperability. This will ensure the BP conformance of your web services.
Understand Application Data Models
When you integrate two applications, it is most likely these two applications will be using different data models though they may be providing similar functionality such as accounting. Understand carefully the data models of the two interacting applications and reconcile them in a common model.

Understand Interoperability of Data Types
All the data types of the two interacting applications may not be compatible to each other. Thus, when you pass parameters and receive the resulting values from a method call, if these data types are not compatible, the two applications will not interoperate correctly.

Having considered the various aspects of implementing SOA, the requirements for creating interoperable web services and various standards for interoperability, we will now take a practical approach to learning by demonstrating the creation of interoperable .NET and J2EE web services in the next section.

Java EE and .NET Integration using Web Services
So far, we have looked into the various aspects of interoperability between web services deployed on disparate platforms. If the web services follow the compliance requirements of the Basic Profile discussed earlier, they can interoperate easily. Fortunately for us, most of the vendors have updated their development platforms to meet the WS-I compliance requirements for creating web services. In this section, we will look at the integration of web services deployed on two popular platforms, Java EE and .NET.

Sample Integration Scenario
We will develop a .NET web service that will be deployed on a .NET server. We will also develop a Java web service that is deployed on a Sun Application Server. We will then write a C# console application that calls both the services and prints the results of the two web service calls on the user console. The application architecture is illustrated in the following figure.
I have kept this application very simple so that we can focus more on the compliance requirements. The service methods on both the web services simply return a greeting message to the caller.

**Developing the Java Web Service**

First, we will develop a Java web service. Creating a Java web service on the latest Java EE platform is an easy task. Write a simple Java class as shown in the following listing.

```java
package endpoint;

import javax.jws.WebService;

@WebService
public class Hello
{
    public String SayHello()
    {
        return "\nHello from Java Service!";
    }
}
```
You need to import the `javax.jws.WebService` package as seen in the listing. The `Hello.java` file contains a public class `Hello`. To convert this into a component that can be deployed as a web service, simply annotate the class with the `@WebService` attribute. The deployment tool then provides the required plumbing to expose this class as a web service. In the class, we write the service methods that can be invoked by a service consumer. We write a single method in this class called `SayHello` that returns a greeting message to the caller.

## Deploying the Service

The easiest way to build and deploy the web service is to use a vendor-supplied IDE. You may use Java Studio Enterprise or NetBeans IDE to build and deploy the service. I used NetBeans 5.5 IDE to deploy the service. The NetBeans 5.5 version provides a template for creating web services. This template creates EJB components for service objects. As I said, to keep the things simple, I avoided using this template. Rather, I used the `ant Build` to compile and deploy the project. You will find the sample `build.xml` file in your NetBeans installation. This file is also available in the code download for this book.

NetBeans IDE can be configured for deployment to any server. I used SunOne Application Server PE 9 for testing. NetBeans also comes with a bundled Tomcat server. You may use this for deployment. If you decide to use Sun App Server for deployment, you will have to add the server in the NetBeans configuration. Incidentally, the Sun Application Server is installed on your machine as a part of Java EE installation.

Once you deploy the service, you can verify it by examining the WSDL generated during the deployment process. To look up the WSDL, type the following URL in your browser:

```
http://localhost:8080/Hello/HelloService?WSDL
```

This will show the generated WSDL in the browser window. We will now examine this WSDL to verify that this is BP compliant.

## WSDL for Java Web Service

The WSDL is shown in the following listing.

```xml
<definitions targetNamespace="http://endpoint/" name="HelloService">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://endpoint/">
        schemaLocation="http://DRSARANG:8080/Hello/HelloService/__container$publishing$subctx/WEB-INF/wsdl/HelloService_schema1.xsd"/>
    </xsd:schema>
  </types>
</definitions>
```
Chapter 4

If you examine this document carefully against the BP conformance requirements, you will find that this file is 100% BP compliant. Thus, to import this as a reference in the .NET client, when we develop it later, would be very easy.
Developing the .NET Web Service

Creating a .NET web service using Visual Studio IDE is as simple as creating a Java web service using NetBeans IDE. I used Visual Studio 2005 for creating the web service and the test client application. The IDE provides a template for creating an ASP.NET Web Service. Follow the default project options while creating the web service. I used the **NetService** as the name for my project and selected C# as the development language. The wizard generates a default class for the web service with a default service method. I modified this service method to send a greeting message to the caller. The modified source file is shown in the following listing.

```csharp
using System;
using System.Web;
using System.Web.Services;

[WebService(Namespace = "http://tempuri.org")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
{
    public Service ()
    {
        // Uncomment the following line if using designed components
        // InitializeComponent();
    } 

    [WebMethod]
    public string SayHello() {
        return "Hello from .NET service";
    }
}
```

As can be seen from the listing, the **Service** class inherits from the **System.Web.Services.WebService** class. The class is attributed with two attributes **WebService** and **WebServiceBinding**. The **WebService** attribute specifies the namespace for the defined web service. The **WebServiceBinding** attribute defines the conformance target. In our example, the conformance target is Basic Profile 1.1 as specified by the constant from the **WsiProfiles** class. Within the class definition, each desired method that is to be invoked as a web service method should be annotated using the **WebMethod** keyword. In our example, the **SayHello** method is declared as a web method that can be invoked using SOAP.
Deploying the .NET Web Service

Once you write the code for the web service, it can be deployed using the wizard provided in the VS.NET IDE. You may now look up the generated WSDL by opening the following URL in your browser:


Note that you will need to set up the appropriate port number in the above URL. The generated WSDL is shown in following listing.

```xml
<wsdl:definitions targetNamespace="http://tempuri.org/">
  <wsdl:types>
    <s:schema elementFormDefault="qualified" targetNamespace="http://tempuri.org/">
      <s:element name="SayHello">
        <s:complexType/>
      </s:element>
      <s:element name="SayHelloResponse">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="SayHelloResult" type="s:string"/>
          </s:sequence>
        </s:complexType>
      </s:element>
    </s:schema>
  </wsdl:types>
  <wsdl:message name="SayHelloSoapIn">
    <wsdl:part name="parameters" element="tns:SayHello"/>
  </wsdl:message>
  <wsdl:message name="SayHelloSoapOut">
    <wsdl:part name="parameters" element="tns:SayHelloResponse"/>
  </wsdl:message>
  <wsdl:portType name="ServiceSoap">
    <wsdl:operation name="SayHello">
      <soap:operation soapAction="http://tempuri.org/SayHello" style="document"/>
      <wsdl:input>
      </wsdl:input>
      <wsdl:output>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:portType>
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="SayHello" style="document"/>
  <soap:operation soapAction="http://tempuri.org/SayHello"/>
</wsdl:definitions>
```
If you compare this WSDL against the BP conformance requirements, you will find that this is indeed BP 1.1 conformant.
Developing the Test Client

We will now develop a C# console application for consuming the previously created web services. Use the project wizard in VS.NET to create a console application. Follow the default while creating the application. I used **TestClient** as the name for my project. The wizard generates the skeleton code for a console application. You will need to add code to this skeleton to invoke the two web services. Before you do so, you will need to add web references to the two services. The IDE provides a menu for adding these references. Use the WSDL URLs specified earlier for locating the web services. Once you complete adding the web reference, you can modify the client code. The intelli-sense feature in the IDE will now resolve the references correctly.

The modified code is shown in the following listing.

```csharp
using System;
using System.Collections.Generic;
using System.Text;

namespace TestClient
{
    class Program
    {
        static void Main(string[] args)
        {
            String str = "Testing .NET, Java Web Services Integration";
            NetService.Service service1 = new NetService.Service();
            str += service1.SayHello();
            JavaService.HelloService service2 =
                new JavaService.HelloService();
            str += service2.SayHello();
            Console.WriteLine(str);
            Console.ReadKey();
        }
    }
}
```

The application obtains the reference to the NetService by instantiating the Service class. The code then calls the service method on the obtained object reference. Similarly, a reference to the JavaService is obtained by instantiating the HelloService class. The SayHello method is invoked on the obtained object reference. Finally, the application prints the two greeting messages on the user console. Note that when you invoke the service method, the binary method call gets converted into a SOAP call. Similarly, the response from the web service is returned to the application as a SOAP response. The underlying runtime transforms the binary call to SOAP and the SOAP response to the C# return type as defined by the method.

**Summary**

SOA has become a buzz world in today's IT industry. From the component-oriented approach, we have now moved into the service-oriented approach. Businesses publish the offered services rather than the interfaces to their components. While designing SOA for an enterprise application, the study of patterns plays a vital role in the success of SOA implementation. In this chapter, the various SOA patterns and guidelines for applying those in real-life situations were covered.

The web services technology perfectly complements the creation of SOA. The chapter discussed the architecture of web services and its benefits. The chapter covered in depth the various patterns that can be applied while creating SOA using web services.

Web services may be used in both EAI and B2B problem spaces. The chapter covered the essential differences between EAI and B2B and how to apply SOA integration techniques in these spaces.

Merely exposing your application as a web service is not sufficient. Any client should be able to use your service with ease. Web services are inherently interoperable. However, due the varying implementations of the web service specifications, usually these are not interoperable. To make web services interoperable, a consortium called the Web Services Interoperability (WS-I) organization was formed. The working group of WS-I created several documents for defining the requirements of creating interoperable secured web services. This chapter discussed these specifications. If you create a web service that is BP (Basic Profile) compliant, it will be interoperable with other services.

The chapter also discussed several guidelines for creating interoperable web services. Finally, a complete trivial example of creating web services on two popular platforms, .NET and Java EE, was discussed. We demonstrated by writing a .NET client that these services are interoperable.
Where to buy this book


Free shipping to the US, UK, Europe, Australia, New Zealand and India.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.