From Technologies to Solutions

SOA Approach to Integration

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

Matjaz B. Juric
Ramesh Loganathan
Poornachandra Sarang
Frank Jennings

SOA Approach to Integration

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

Matjaz B. Juric

Ramesh Loganathan

Poornachandra Sarang

Frank Jennings

BIRMINGHAM - MUMBAI

SOA Approach to Integration

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

Copyright © 2007 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, Packt Publishing, nor its dealers or distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: November 2007

Production Reference: 1211107

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 978-1-904811-17-6

www.packtpub.com

Cover Image by Vinayak Chittar (vinayak.chittar@gmail.com)

Credits

Authors

Matjaz B. Juric

Ramesh Loganathan

Poornachandra Sarang

Frank Jennings

Reviewers

Manish Verma

Clemens Utschig

Senior Acquisition Editor

Louay Fatoohi

Development Editor

Mithil Kulkarni

Technical Editor

Ajay.S

Editorial Manager

Dipali Chittar

Project Managers

Patricia Weir

Abhijeet Deobhakta

Indexer

Bhushan Pangaonkar

Proofreader

Chris Smith

Production Coordinator

Shantanu Zagade

Cover Designer

Shantanu Zagade

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.

About the Reviewers

Manish Verma is VP, Delivery, at Fidelity National Information Service's software development center in Chandigarh, India. Manish has 14 years of experience in all the aspects of the software development lifecycle, and has designed integration strategies for client organizations running disparate systems. Manish's integration expertise is founded on his understanding of a host of technologies, including various legacy systems, .NET, Java technology, and the latest middleware. Prior to Fidelity National, Manish worked as a software architect and technical lead at Quark Inc., Hewlett Packard, Endura Software, and The Williams Company.

Manish writes on technical topics on a regular basis. His current focus areas are integration of disparate systems, and web services security, transactions, access control, identity management, and provisioning.

You can contact Manish at mverma@fnisindia.com.

Clemens Utschig works within the Oracle SOA Product Management Team at Oracle Headquarters, Redwood Shores, where his responsibilities include cross-product integration as well as the growth of the developer community on OTN.

Apart from technology, his focus is on project management and consulting aspects, as they relate to SOA implementations.

As a native Austrian, his career started in Europe at the local consulting services branch, working with customers on J2EE and SOA projects, where he also founded the local Java community within Oracle Austria.

He is a frequent speaker at international conferences where he evangelizes technology and the human factor as they relate to shifts in IT strategy.

Table of Contents

Preface 1

Chapter 1: Integration Architecture, Principles, and Patterns 5

Integration Challenges 6
Current Situation 8
Effective Information Systems 9
Replacing Existing Applications 9
Requirements and Strategies 11
Single Data Input 11
Information Access with Low Latency 12
Importance of a Centrally Managed Integration Project 13
Responsibility to Define Integration Architecture 14
Responsibility to Select Integration Infrastructure and Technologies 15
Development and Maintenance of Integration Documentation 15
Integration Architecture Steps and Approaches 16
Bottom-Up Approach 17
Top-Down Approach 21
Sound Integration Architecture Benefits 23
Types of Integration 24
Data-Level Integration 25
Application Integration 26
Business Process Integration 28
Presentation Integration 29
Business-to-Business Integration 29
Integration Infrastructure 30
Communication 31
Brokering and Routing 32
Transformation 33
Business Intelligence 33

Transactions 34
Security 34
Lifecycle 34
Naming 35
Scalability 35
Management 35
Rules 36
Integration Technologies 36
Database Access Technologies 37
Message-Oriented Middleware 37
Remote Procedure Calls 39
Transaction Processing Monitors 40
Object Request Brokers 41
Application Servers 42
Web Services 43
Enterprise Service Buses 45
The Integration Process 46
Choosing the Steps and Defining the Milestones 46
Sound Practices 48
Iterative Development 48
Incremental Development 49
Prototyping 50
Reuse 50
Integration Process Activities and Phases 50
Integration Patterns 52
Summary 53
Chapter 2: Service- and Process-Oriented Architectures for Integration 55
Defining Service-Oriented Architectures 57
Why SOA in the Integration Space? 60
Islands in the Enterprise IT Landscape 60
The Integration Problem 62
Custom Integration Application and Its Issues 63
Inverted View: Reusable Services, Simple Integration Processes 65
Enter SOA: A Services-Based Integration Architecture 65
Concepts and Principles of SOA 66
Paradigm Shift—from Self-Contained Applications towards "Services" 66
Service Orientation 67
Component-Based Services 68
The Internet Simplifies Remote Services 69
Consuming Services 71
Introducing SOA Architecture 71

Service Abstractions 72
Service Invocation and Service Implementation 73
Process Engines 73
Messaging Abstractions 73
Synchronous and Asynchronous Messages 74
Service Registries 74
Quality of Service 75
Communication Infrastructure 75
What is a "Bus"? 75
XML and Web Services: SOA Foundation 76
Using XML in Middleware 76
Middleware Mechanics for Services 76
XML-Based Mechanism to "Invoke" Services 77
Services over the Web via SOAP 79
Web Services—Protocols for SOA 79
Technology Agnostic System-to-System Interaction 81
Service Description—Using WSDL 83
Discovering the Services—UDDI 83
Containers to Host Web Services 84
Standards Foundation 84
Application Platforms (JAVA EE) Hosting Web Services 88
Using Services to Compose Business Processes 89
Simple Integration Applications 89
Simple Business Processes—Orchestrating the Services 90
Choreography—Multi-Party Business Process 91
SOA Security and Transactions 93
Security Challenges in a Services Environment 93
Simple Middleware Systems Security 94
Security in Java Infrastructure 95
Microsoft.NET Security 96
Web Services Security for Loosely Coupled Services 96
Emerging Web Services Security Standards 97
Transactions in SOA 98
Web Services Transaction—A Standard 99
Infrastructure Needed for SOA 99
Service Execution and Communications 100
Types of Component Services 101
Service Containers (Execution Engines) 101
Communication Infrastructure—Under the Covers 103
Communication "Bus"—At the Core 104
MOM 105
XML Backbone (XML, Transformations, and Persistence) 105
Reliability and Scalability 106
Managing a Distributed SOA Environment 106
Options for SOA Infrastructure 107
Web Services 108

Application Platforms (JAVA EE / .NET) 108
Simple Messaging-Based Custom Infrastructure 109
Integration Platforms (EAI) 109
ESB—Enterprise Service Bus 110
Designing Services and Processes for Portability 110
Adoption Considerations 111
Think Services 112
Model the Business Data as XML 113
Processes in BPEL 114
New Applications—Prepare for SOA/POA 114
Design for Infrastructure (Vendor) Independence 114
Transition to Process-Oriented Architectures 115
Services and Processes Coexist—But Services First 117
Process—Orchestration of Services 117
POA—Shifting the Focus to "Processes First" 118
Concepts and Principles of Process-Oriented Architectures 119
POA—Processes First. Services... Maybe! 119
POA Enables Top-down Design—Using just Processes 120
Analysts Become Programmers 120
POA Changing Software Development Roles 121
Process Standards 122
Infrastructure for Process-Oriented Architectures 123
Summary 124
Chapter 3: Best Practices for Using XML for Integration 125
Introduction 125
Domain-Specific XML Schemas 125
Validating XML Documents 127
Mapping Schemas 129
Choosing Processing Models 129
Fragmenting Incoming XML Documents 131
Design Recommendations 131
Default Namespace—targetNamespace or XMLSchema? 133
Localize Namespace vs. Expose Namespaces 137
Advantages of Localizing Component Namespaces within the Schema 138
Advantages of Exposing Namespaces in Instance Documents 138
Global vs. Local Declaration 139
Russian Doll and Salami Slice Designs 139
Element vs. Type 140
Zero, One, or Many Namespaces 141
Use the Heterogeneous Namespace Design 142
Use the Homogeneous Namespace Design 142
Use the Chameleon Design 143
Using XSL for Transformation 143
xsl:import and xsl:include 143

Securing XML Documents 146
XML Encryption 147
Encrypting an XML File 148
SSL versus XML Encryption 150
XML Signatures 151
Guidelines for Securing Your Services 152
XML Streaming and DOM 153
Pull Parsing versus Push Parsing 153
What is StAX? 154
StAX and Other JAXP APIs 154
Performance Considerations 155
Limit Parsing of Incoming Documents 156
Use the Appropriate API 156
Choosing Parser 157
Reduce Validation Cost 157
Referencing External Entities 158
Dynamically Generated Documents 158
Using XML Judiciously 159
Summary 159
Chapter 4: SOA and Web Services Approach for Integration 161
Designing Service-Oriented Architectures 162
SOA Evolution 162
IT Evolution 165
Patterns 166
Business Patterns 166
Integration Patterns 167
Composite Patterns 167
Application Patterns 168
Runtime Patterns 168
Product Mappings 168
Guidelines 168
Designing Sound Web Services for Integration 169
Web Services Architecture 169
Web Services Benefits 170
Self-Contained 170
Self-Describing 170
Modular 170
Accessible Over the Web 171
Language, Platform, Protocol Neutral 171
Open and Standards-Based 171
Dynamic 171
Composable 171
Patterns 172
Self-Service Business Pattern 172

Extended Enterprise Business Pattern 173
Application Integration Pattern 174
Application Integration Patterns 175
Direct Connection Application Pattern 176
Broker Application Pattern 177
Serial Process Application Pattern 178
Parallel Process Application Pattern 179
Runtime Patterns 180
Nodes 180
Connectors 181
Direct Connection Runtime Pattern 182
Runtime Patterns for Broker 186
Differences between B2B and EAI Web Services 188
Interface Design 189
Use of a Service Registry 189
Writing Interoperable WSDL Definitions 190
Validating Interoperable WSDL 194
Interoperability Challenges in Web Services 195
WS-I Specifications 197
WS-I Basic Profile 1.0 197
WS-I Basic Profile 1.1 198
WS-I Basic Profile 1.2 199
WS-I Basic Security Profile 1.0 201
Guidelines for Creating Interoperable Web Services 203
Avoid using Vendor-Specific Extensions 203
Use the Latest Interoperability Tests 203
Understand Application Data Models 204
Understand Interoperability of Data Types 204
Java EE and .NET Integration using Web Services 204
Sample Integration Scenario 204
Developing the Java Web Service 205
Deploying the Service 206
WSDL for Java Web Service 206
Developing the .NET Web Service 208
Deploying the .NET Web Service 209
Developing the Test Client 211
Summary 212
Chapter 5: BPEL and the Process-Oriented Approach to Integration 213
Process-Oriented Integration Architectures 214
Service Composition 217
Orchestration and Choreography 218
Complexity of Business Services 219

Identifying Business Services 219
Development Lifecycle 220
SOA and Executable Business Processes 222
Example Business Process 224
BPEL for Service Composition 225
What We Can Do with BPEL 226
Executable and Abstract Processes 227
BPEL and Other Process Languages 228
Languages for Choreography 228
Modeling Notations 229
Writing BPEL Processes 229
Process Interface 230
Partner Links 231
Partner Link Types 231
Variables 232
Handlers 232
Fault Handlers 233
Event Handlers 233
Compensation Handler 234
Scopes 235
Overview of BPEL Activities 235
Developing an Example BPEL Process 238
Services Used in the Process 239
Resource Data Service 239
Rating Service 243
Billing Service 246
Adding Partner Link Types to the Service's WSDL 248
Define a WSDL Interface for the BPEL Process 250
Writing the BPEL Process Logic 252
Process Declaration 253
Defining Partner Links 254
Declaring Variables 255
Writing the Process Definition 256
Adding a Fault Handler 263
Adding an Event Handler 264
Deploy and Run the Process 265
Summary 268
Chapter 6: Service- and Process-Oriented Approach to Integration Using Web Services 269
Enterprise Service Bus 269
From Just Services to an Enterprise Bus 270
ESB Architecture 275

Defining ESB 276
Middleware for Middleware Technologies 278
Modeling the Enterprise Document Flows 280
ESB Services: Built on Documents/Messages 286
ESB Infrastructure Components 289
Built on Web Services Standards 293
Service Containers—The Primary Tier of the Bus 296
Inside the Container 298
External View of Services: Documents Sent to Abstract "Endpoints" 301
JBI—A Standard Container to "host" Services 304
Communication Infrastructure 306
Bus Services—Mediation, Transformations, and Process Flows 307
Why Mediation? 308
Infrastructure Mediation 310
Intelligent Content-Based Routing 312
Transformation Services 313
ESB Processes: Extending the WS Process Model 315
Security and Transactions 319
Security Considerations in Integration Architecture 319
ESB Security—Built on WS-Security 321
Transaction Semantics for Enterprise Integration 324
Distributed Transactions and Web Services 327
Realizing Transactions in ESB 329
Reliability, Scalability, and Management 330
Reliability Concepts 330
Achieving Reliable Communication through ESB 334
High Availability in ESB—Leveraging the Messaging Platform 336
Scalability and Performance of ESB 338
Control and Management of ESB 341
Application Development Considerations 346
Integration Application Constituents 346
ESB—Application Design Approach 348
Comparing ESB with Other Technologies 350
ESB—Helps Avoid Vendor Lock-Ins 354
Extending ESB to Partners 356
Summary 358
Index 359

Preface

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.

Who is This Book for

The target audience for this book are architects and senior developers, who are responsible for setting up SOA for integration for applications within the enterprise (intra-enterprise integration) and applications across enterprises (inter-enterprise integration or B2B).

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

There are two styles for code. Code words in text are shown as follows: "We can include other contexts through the use of the `include` directive."

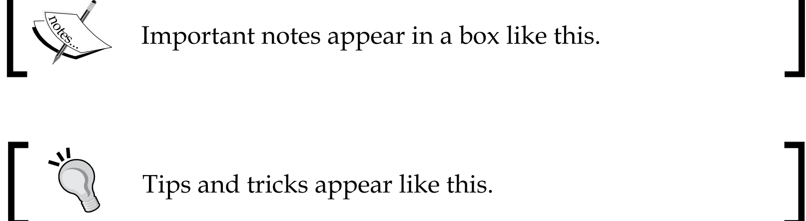
A block of code will be set as follows:

```
// set the response content type
Response.ContentType = "text/xml";
// output the XML header
Response.Write("<?xml version=\"1.0\" encoding=
                            \"UTF-8\" standalone=\"yes\"?>");
Response.Write("<response>");
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items will be made bold:

```
<html>
  <head>
    <script type="text/javascript" src="file.js"></script>
  </head>
</html>
```

New terms and **important words** are introduced in a bold-type font. Words that you see on the screen, in menus or dialog boxes for example, appear in our text like this: "clicking the **Next** button moves you to the next screen".



Reader Feedback

Feedback from our readers is always welcome. Let us know what you think about this book, what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply drop an email to `feedback@packtpub.com`, making sure to mention the book title in the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or email `suggest@packtpub.com`.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer Support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in text or code—we would be grateful if you would report this to us. By doing this you can save other readers from frustration, and help to improve subsequent versions of this book. If you find any errata, report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **Submit Errata** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata are added to the list of existing errata. The existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with some aspect of the book, and we will do our best to address it.

1 Integration Architecture, Principles, and Patterns

The growing needs for information availability and accessibility present new challenges for application development. Stand-alone applications cannot fulfill the growing needs anymore. There are two forces working in parallel with regard to the need for integration. First, it is necessary to allow for application integration within an enterprise, and second, there are growing needs to ensure inter-enterprise or "business-to-business" integration.

The majority of companies, however, still have existing legacy applications, developed using different architectures and technologies, which have usually not been designed for integration. Companies cannot afford to write off or replace them over night, because they are mission critical; also they cannot afford to develop their entire information systems from scratch in today's business environment.

In addition, companies will undoubtedly need to introduce new applications and systems from time to time. New solutions are usually based on modern architectures, which differ significantly from architectures used by existing legacy applications. These new applications also have to be integrated with existing applications; and existing applications have to be integrated with each other to fulfill the information availability and accessibility goals. To make things even more difficult, there is often a significant investment already in place for a variety of application integration technologies.

We can see that integrating applications is a difficult task, maybe even one of the most difficult problems facing enterprise application development. To fulfill these integration objectives, several methods, techniques, patterns, and technologies have been developed over the years, ranging from point-to-point integration over enterprise application integration (EAI) and business process management to service oriented architectures (SOA).

In this book, we will focus on the latest architectures, principles, patterns, and technologies, which promise to simplify integration, make it more efficient, and enable end-to-end support for business processes as a consequence of well integrated information systems. These architectures and principles are known as **Service-Oriented Architecture**—SOA. We will explain what is behind the service- and process-oriented approaches to integration. We will also talk about technologies, particularly **Web services**, **Business Process Execution Language** (BPEL), and **XML**.

In the first chapter, we will look at the integration challenges, become familiar with different architectures and principles, and will take a quick overview of integration patterns. We will look at:

- Integration challenges
- Requirements and common strategies
- Integration architectures
- Types of integration
- Integration infrastructure
- Integration technologies
- The Integration process
- Integration patterns

Integration Challenges

The ability to instantly access vital information that may be stored in a variety of different applications may influence the success of a company. For each company, the presence of effective information infrastructure that avoids the need for employees to perform numerous manual tasks like filling in paper forms, and other bureaucracy, is very important. Employees should not have to contend with such inefficiencies and 'irritations' such as switching between different applications to get their work done, reentering the same data more than once in different applications, or waiting for data to be processed. Ideally, a well-integrated system should offer end-to-end support for business processes with instant access to information, no matter which part of the system is used. Similar consideration hold true for companies that want to be successful in e-business or those that want to improve their position in the virtual world.

Companies are realizing the importance of integration at different speeds. Some of them are already fully involved in integration projects with many solutions already working, as they have seen the advantages of integration and understand how to achieve successful integration. Other companies are aware that integration is important but, although they may have started integration projects, they do not have the results yet, mainly because the integration projects have not been successful. Further still, some companies are only now realizing the importance of integration, and this could in fact be too late for them. Such companies may be looking for ways to achieve integration fast, without spending too much money, and without assigning too many staff members to the integration project. But cutting corners and attempting to implement only the most needed parts of integration in the shortest possible time will most likely result in only partially working solutions at best.

The problem that makes things worse is the fact that managers are often not familiar with all the complexity hidden behind the integration process. Sometimes, even the "IT people", the architects and developers, do not fully understand the traps behind integration. Most importantly, managers might not understand that integration is a topic that is related to the company as a whole, and not with the IT department only.

Another scenario that leads to the same disorganized approach is when the management of a company does not see the need for integration yet, but the IT department is aware that integration is needed and should be initiated as soon as possible. Therefore, the integration team starts to build a partial solution to solve the most urgent problems. As the integration is not approved from the top, the IT sector does not get enough resources, it does not have enough time and money, and, most significantly, it does not have authorization to start to solve the integration problem globally. Most developers will agree that these are all-too-familiar situations.

Integration seems to be one of most important strategic priorities, mainly because new innovative business solutions demand integration of different business units, enterprise data, applications, and business systems. Integrated information systems improve the competitive advantage with unified and efficient access to the information. Integrated applications make it much easier to access relevant, coordinated information from a variety of sources. In effect, the total becomes more than the sum of its parts. It's easy to see that integration can be an important and attractive strategic priority.

Current Situation

Typical companies that have existed more than just a few years rarely have integrated information systems. Rather they are faced with a disparate mix of heterogeneous existing systems. Companies will typically have different applications, developed over time. These include:

- Applications developed inside the company
- Custom-built but outsourced solutions
- Commercial and ERP applications

These applications have typically been developed on different platforms, using different technologies and programming languages. Because they have been developed over time, different applications use different programming models. This is manifested through:

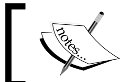
- Combinations of monolithic, client/server, and multi-tier applications
- Mix of procedural, object-oriented, and component-based solutions
- Mix of programming languages
- Different types of database management systems (relational, hierarchical, object) and products
- Different middleware solutions for communication (message-oriented middleware, object request brokers, remote procedure calls, etc.)
- Multiple information transmission models, including publish/subscribe, request/reply, and conversational
- Different transaction and security management middleware
- Different ways of sharing data
- Possible usage of EDI, XML, and other proprietary formats for data exchange

Another important aspect is the integration that has already been implemented between the existing applications. This includes everything from simple data exchange to use of middleware products. In addition to all this diversity of architectures, technologies, platforms, and programming languages the companies have kept introducing new applications—and with new applications they have introduced modern application architectures too.

Effective Information Systems

Ultimately, the driving force behind all these situation is an effective information system. The important point here is that an information system is, by definition, only as effective as the integration between different applications. Applications developed in the past have usually not been designed for integration. This is why in the majority of older applications no or only very limited ways exist to achieve interoperability with other applications. There is often little, if any, documentation relating to their development and processes. Even today, some applications are developed with little or no concern given to how to connect them with other systems. For these reasons, integration is as relevant to existing legacy systems as it is to modern solutions.

The reason for integration lies in business expectations. From the business viewpoint, the goal is to maximize the benefits of each application and the information system as the whole. Particularly, end-to-end support for business processes and an efficient way of user interaction are important. Separate applications cannot fulfill the requirements anymore. The data is partitioned among different applications; each has its own view on the data, which makes joining it a difficult problem. Therefore users often cannot obtain the global picture, and valuable information is locked in these applications that could be shared and distributed to all new groups of information consumers: remote employees, business partners, and customers, to name just a few of them. In addition, separate applications are also highly unsuitable for supporting modern front-end applications, such as web access, business-to-business and business-to-customer collaborations—for example integrating back-office applications with a front-end customer web-order system.



Integrated information systems seamlessly support business processes.

In order to stay competitive, companies must modernize and improve the functionality of their information systems. Managers see an information system as a tool through which they can maximize the company benefits. They are prepared to invest in them because they know what return of investment they can expect; they can foresee the efficiency increase and the productivity boost.

Replacing Existing Applications

Improving the functionality of an information system is possible in several ways. The most obvious way is to replace the old applications with a freshly developed solution. Even if, at first sight, this seems like an attractive solution, we can easily see why it is inapplicable in most cases. Developing and deploying new software overnight is impossible; also, migration of live (in-use) systems can incur significant costs.

If we were able to develop an integrated information system that would cover all the requirements, this would perhaps offer the ultimate solution. Unfortunately, there are several factors that prevent us from doing this.

First of all, to develop new information system from scratch requires time and resources that in most cases, we simply cannot afford. There have been many years of development invested into the existing applications, and re-implementing all of this functionality would require a lot of knowledge, time, and resources. It is very unlikely that this would be possible in a reasonable or acceptable time. The existing systems also contain a lot of knowledge that we would have to recapture.

Likewise, it is infeasible to introduce commercial solutions, like ERP systems, in any great hurry. In a sense, each company evolves like a living organism, with a lot of specialties and a distinctive way of doing business. All these properties have to be reflected in a business's information systems. Adapting commercial solutions requires time, and in the eyes of managers, time is money.

Companies today are also so complex that ERP systems could not cover all possible needs. Research from leading consulting companies suggests that ERP can only account for 30% to 40% of information needs. The remainder has to be covered by custom solutions. This is also the reason why the majority of ERP suppliers, such as SAP, or Oracle, have introduced interoperability interfaces into their products.

Even if we could develop the new information system in a reasonable amount of time, we would still need to deploy it, to transfer all data, and to teach the users how to use the new system. These are some of the reasons why there will be only a few companies that will attempt to replace their entire information systems. Such companies will soon realize that the replacement is more complicated, more complex, more time consuming, and more costly than even their worst-case scenarios could have predicted.

Replacing existing systems with new solutions therefore almost always requires more time and more money than previously planned, even in the most pessimistic scenarios. Incorporating all the peculiarities and special cases in the software requires time and knowledge. It depends on the company if it has business processes well documented, but there are few companies with detailed documentation for all the business processes that would need to be developed. For those that do not have all the necessary documentation, it is even more difficult. Often, there is no single person that understands the entire process.

It is clear that replacing existing systems with new solutions will often not be a viable proposition. Too much time and money has been invested in them and there is too much knowledge incorporated in these systems. Therefore a standard ways to reuse existing systems and integrate them into the global, enterprise-wide information system must be defined. In this book, we will see that SOA is today seen as the architecture that provides answers to these integration challenges.

Requirements and Strategies

To be successful with integration, companies must have an integration strategy. Management must support the integration process and provide the necessary priorities to enable a centralized control and management of the integration. There must be clear requirements specifying what is expected from an integrated system and goals have to be defined. Integration is today best seen as an ongoing process. Two important goals for integrated information systems that should be fulfilled are single data input and information access with low latency.

Single Data Input

Single data input ensures that data is entered into the information system only once. Although this sounds like an obvious goal, it is very difficult to achieve it 100% in the real world. Entering the data only once means that we have to provide a level of integration where the user does not have reenter the same data again for no matter what purpose. This might not be so difficult to achieve for main, or primary applications. But don't forget that the users often use many subsidiary applications as well, which provide solutions to some every day problems. This includes everything from serial mail printing to analysis of business data. For these operations, they typically use productivity tools and transfer the data from the information system to these tools. Those users that are not very familiar with the technology end up retyping the same data.

It is obvious that a single data input guarantees data consistency and minimizes the mistakes that are provoked by data retyping and managing local stores of enterprise data. Let us look at the example of serial letters.

In a non-integrated system, the user would probably reenter the data to some specialized, self-developed solution, in the 'Office' style product of a word processor. Very rarely will the user be so familiar with the technology as to be able to implement an automatic data transfer. If the user on the other hand, is somewhat familiar with the technology, he or she will do the data transfer manually, which will still require a lot of effort. He or she will extract data for the information system application that takes care of customer management and then change the format manually, probably using an editor and a filter tool to import the data into the word

processor. When he or she has to do the serial letter again, it is likely that he or she will use the old data and hope that not too much has changed. Doing the whole data transfer all over again will probably be too time consuming and the user will think twice before doing it. Rather he or she will risk not including the recent changes in the addresses. As a result, some new customer might not get the correspondence, and for some existing customers it may be sent to old addresses. Remember also, that the user will typically guess how many modifications have been done to the original data, because an analysis would take too long and would be too difficult to do (based on the fact that the user would have to implement it himself or herself).

Information Access with Low Latency

The other important goal is data access with low latency, sometimes also called data access without latency. Information access with low latency ensures that the changes made in one part of the information system become visible in all related parts immediately—or in the shortest possible time. Achieving this can be a very complicated task even in a smaller information system. If we imagine a distributed information system, this can get considerably more complicated.

On the other hand, is achieving low latency very important for today's online systems? Imagine an information system that enables users to book coach tickets over the phone. The clerk on the phone will enter the destination and the dates that the customer wants to travel on. Using a low latency information system, the clerk will be able to answer the question if the seats are available for a given destination at a given time. Otherwise, this information will have to be mailed to the customer, complicating the process and making it more costly. Introducing this latency will also mean that the company will not be able to sell all the seats but will always need a certain number of seats reserved just in case. We could find many similar examples; therefore, a general answer is that low latency is very important for today's information systems.

It is only a small step from the mentioned scenario to the e-business example where the customers access the information system themselves through a web-based interface. The customers expect that they will get information immediately; however, the information given to customers also has to be reliable.

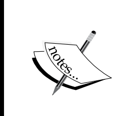
Achieving low latency in front-end systems requires there to be a low latency strategy implemented in the back-end systems; this means in the integrated enterprise information system.

The main reason why low latency is difficult to achieve is because the functionality required to perform business tasks is dispersed across different applications. Additional to this, is the physical distribution of applications and communication links that do not provide enough throughput and reliability.

Solving the problem of the dispersion among applications can be achieved in a number of different ways. Historically, this problem has been addressed at the data level only. The typical solutions include building consolidated databases and data warehousing.

These techniques have to take snapshots of the data from time to time and store them in the central data storage in a format that is suitable to do analysis and other ways of validating and analyzing it. This approach was, however, not very good at achieving low latency, because in the real world the data transfers can be run only a few times a day. To be fair, we have to mention that data warehouses were never intended to solve the low latency problem.

Direct access to data in real time is much harder to implement, particularly if we have to deal with many distributed data sources. Real-time access has several performance and scalability limitations. The most efficient way to access data is through application logic.



Both requirements, single data input and information access with low latency, are related to the ultimate objective of integration—end-to-end support for business processes provided by the information system, which can be relatively easily adapted to the changes in business processes.

Importance of a Centrally Managed Integration Project

It should be clear that integration is not only a technical problem. Rather, it also requires a good level of organization and management in order to be successful. Ultimately, it requires support from the senior management. Senior managers have to recognize the need for integration first, and that this integration must take place on an enterprise-wide scale. For successful integration, it is crucial that the company initiates the integration project with enough resources, priorities, and authorizations to be able to achieve the coordination of all involved parties.To achieve this, there are several possibilities.

If a company has a centralized IT department, then it can take over the responsibility for integration too. However, we have to be careful to ensure that there are actually enough staff members available for the task at hand. One, two, or three persons with part-time responsibilities are not enough to make the integration project take off. Obviously, this will depend on the size of a company and the number of application domains, but often this is another area where management and reality do not agree.

It will also depend on the planned schedule. Although integration today is one of the main priorities of companies, it is still worth mentioning that planning a very long schedule for integration will probably result in a lack of success. The business and its requirements are prone to rapid changes, and an integration architecture that is defined over a long period of time will be unable to incorporate such changes.

Of course, a centralized IT department is not always the case. Often, companies will have IT departments (or similar structures) distributed all over the company. This can be a consequence of merging or even a consequence of the past needs because of the size of the company. Still, we have to identify somebody who will be responsible for the integration and who will be able to coordinate all the involved parties. Assigning the responsibility to one of the IT departments would be unwise because this would place this IT department above the others. This can lead to the other IT departments being unwilling to fully cooperate—and this can be enough to make integration unsuccessful.

It is a much better idea to introduce and organize a new central body that we will call an integration service, responsible for integration. The integration service should have three major responsibilities:

- Definition of integration architecture
- Selection of integration infrastructure and technologies
- Development and maintenance of integration documentation

Responsibility to Define Integration Architecture

The first function of the integration service is the operational responsibility of defining the integration architecture. This includes the identification of architectural and semantic questions regarding the integration. The major responsibility of the integration service is to define logical, high-level integration architecture.

As we will see later in this section, the top-down approach has to provide a sort of 'city plan', which will show us the available 'roads' and allow us to place individual 'houses'. The city plan represents a high-level integration architecture. It would be best if the integration architecture models the integrated information system as it should be, that is, as we would like it to be. Such architecture would, however, also need to enable reuse of at least some of the existing applications and to minimize the dependencies between applications. This last requirement will enable us to replace existing applications with new systems.

The integration service should also resolve the important semantic issues that have to be centrally implemented. This includes unification of the data model, issues regarding the interfaces between applications, the format of messages sent between applications, and so on.

Responsibility to Select Integration Infrastructure and Technologies

Another important responsibility is to identify, select, deploy, and maintain a consistent integration infrastructure, based on a set of interoperable technologies. Particularly, the identification and selection of integration infrastructure should be a coordinated effort and the integration service must take into account all aspects for existing applications and try to find an optimal solution. Consistent technology for integration is important.

It is not necessary that the integration service selects only one technology for integration. It may well be that there is no single technology that would be able to incorporate all of the applications. If we select more than one technology, we should provide instructions on how to achieve interoperability between them and make these instructions available to all parties involved.

Development and Maintenance of Integration Documentation

The third responsibility of the integration service is the development and maintenance of the integration documentation. Typically, the documentation consists of a set of interface models that describe the communication between the application domains and applications. These models will be realized from the integration design. The technology selection will also have influence on the how these interfaces are implemented.

The integration service will have to take care that the project teams follow the integration guidelines and that they modify their applications to accommodate the interfaces and technology necessary for integration. They should also include the support for new software that they develop.

Further, the integration service should contact the application vendors that have developed and supplied applications used in the company. They should find out how to incorporate the necessary solutions and technology into their applications and should also define the schedules.

We have to be aware that for successful integration, the answers to the technology questions alone are not sufficient. Today, there are practically no technological barriers for integration. It would be very difficult to find platforms, operating systems, or applications that would be impossible to integrate.

However, there can be significant difficulties when integrating two applications on a single operating system, written in the same programming language, and using the same database management system, because we have to solve many semantic differences between them. Integration is dependent on the application architecture and to the interoperability interfaces, particularly their availability. A technology cannot avoid organizational problems. From this perspective, a solid organizational structure for existing systems integration is essential and is a key success factor.

Integration Architecture Steps and Approaches

Integration is not an easy task, and to succeed, we will have to plan and design it carefully. Therefore, we first have to select the applications that we will include in integration. Applications in each company can be put into the following two classes:

- Primary applications, which are important for the whole company and used by a large number of employees. These applications have been either developed either in-house, outsourced, or bought and are under the control of IT department.
- Subsidiary applications often used by a single employee. Most likely they have been developed without the knowledge of the IT department by the user himself or herself in a productivity tool, like office applications and similar. These subsidiary applications ease the every day work of this employee and implement the tasks that this employee has to perform, but are not supported by the primary applications. Examples include printing circular letters, generating specific reports, etc.

When designing the integration architecture, it is very important that we include both the primary and the subsidiary applications. The users have developed subsidiary applications because they need them, and they will not want to live without them. If we do not implement them as a part of integrated information system, the users will continue to use these self-developed solutions. This will result in several problems, including manual transfers of data between the primary and subsidiary applications and important data being stored on local computers in a variety of formats, like word processors, spreadsheets and similar. Accessing this data will be very difficult and the objectives of integration will not be met.

Therefore, before we begin with integration, we should do an analysis of all the applications that have been used in a company from large company-wide applications to the small personal solutions of individual employees. After we have selected the applications, we have to choose the approach to integration. We will look at two different approaches to integration and discuss their advantages and disadvantages:

- Bottom-Up Approach
- Top-Down Approach

Let's look at the bottom-up approach first.

Bottom-Up Approach

The bottom-up approach to integration focuses on individual problems that arise because of the lack of integration between applications. It solves the individual problems through uncoordinated integration projects, without building the global integration architecture.

In order to define the bottom-up approach to integration, let's consider an example of a mobile phone operator that has several separate applications that do not communicate. These applications are:

- An application for customer management
- Two different applications for accounting the domestic and international calls
- An application that manages the accounts of customers
- An application that prints invoices
- An application that handles the tariffs and costs of calls

To keep the example simple, let's suppose that there are no connections between the applications. However, it is obvious that there should be many. Due to the lack of integration, users have to reenter the same information two or three times, which not only makes the work take longer than necessary, but also introduces an element of human error through typing mistakes. These can be difficult to track down, thus losing additional time. The IT department and the users of these applications are fully aware that integration would bring important benefits. At the very least, connections are required between the invoice printing applications, both call accounting applications, the customer management application, and the account management application.

Let's also presume that in this particularly company, the IT department does not have the authority to make independent decisions and cannot convince the upper management of the necessity of integration. The users of these applications also don't have important positions in the company because they are mostly clerks.

Due to regulations, competition between the mobile phone operators is not a serious issue, so management does not prioritize the need for complex analysis and other up-to-date information. They are satisfied with information that is several days old, which can therefore be prepared manually.

In this particular scenario, it is easy to see that convincing management of the need for integration would be a difficult, if not an impossible, task. Now, we may wonder why management does not see the need for integration. Unfortunately, too often a business must feel pain before management gives attention to a problem—this is reactive rather than proactive management, and, unfortunately, is very typical. In our scenario, things have not developed far enough for this to happen.

In our case, the IT department can wait until management experiences the negative effects of their inefficient systems, and then starts to look towards integration. Or, the IT department can take things into their own hands and try to implement the most needed integration parts between the applications. Unfortunately, both approaches will ultimately lead into the same problems.

If the IT department waits and does nothing regarding the need for integration, the time will come when management start to feel that there is a problem that can be solved with integration. In our scenario, this will most likely be the situation when competition in the market starts to increase. At this point, management will look for ways in which to minimize costs; they will initially identify the lost hours of the workforce, because of the unnecessarily repetitive nature of some of the tasks (for example, reentering the data, manually changing the structure of the data, and so on). They will also feel the need for accurate and on-time information, which can only be effectively produced by an integrated information system.

Given the example, it will likely be a little too late by the time management senses the problem. Therefore, management will ask for the IT department to implement the fastest possible scenario to get results as soon as possible. Almost definitely, management will not take the time to understand everything that they need from the information system. They will see a subset of problems, most likely the most obvious one. In our example, this would be, say, the potential integration between the invoice printing, call accounting, customer, and account management applications mentioned previously. Furthermore, management will probably not consider anything beyond what will satisfy their own narrow view of the problem.

Now most IT departments, because they don't have a choice anyway, will agree and try to implement the connections between applications that they need most and that can be built in the quickest time. They will start to look for solutions on how to integrate the relevant applications in the easiest way. Let's say that in our present example not all applications have been developed within the company. Some of them are commercial applications, and some of them have been outsourced.

The IT department therefore is not familiar with the finer details of all the applications. To apply changes to the applications, the IT department in our scenario would again have to contract external companies. This would result in additional costs and would almost certainly extend the schedule. This option will, therefore, most likely be dismissed.

So, the IT department will search for ways in which to integrate their applications without modifying them (or at least that minimize the amount of modification). The most obvious potential solution is to use data-level integration. The IT department could choose to simply move the data between the databases of different applications. It is, however, important to recognize the basis of this decision, and the fact that we do not blame the IT department in such cases. Indeed, if an IT department were to consult an external expert, this consultant would almost certainly make the same decision, given the facts and the requirements defined by the management. The challenge thus now becomes how to bridge different databases.

Let us look at a possible outcome of this potential solution. The IT department would probably first implement the data transfer from customer management application to the invoice printing application. Then, they might implement the data transfer from the domestic and international call accounting applications to the tariff and cost management application. The data transfer that had previously been done manually would simple port this data to the tariff and cost management application, which would then return the actual consumption in currency. The same approach would then be applied to other connections too.

After implementing all necessary data transfer connections between the applications, it could seem like the integration problem had been solved. However, the success would only be true on the surface, and new problems would soon emerge. If the IT department follows this pattern of implementing partial solutions, it will soon lead to a system of applications that will have several dependencies that they will not even be aware of.

In the proposed scenario, almost every application would have to be connected to all the others. This might not seem a lot when connecting two or three applications, but if we consider (more typically) twenty to fifty or more applications, it clearly leads to an exponential increase in complexity. Small changes to one application could require modifications to all the data transfer bridges to other applications. In fact, the maintenance of an integrated system will be more time consuming and costly than the maintenance of the applications themselves, which will make the benefits of integration less obvious.

This approach to integration also soon limits how far we can go, and has several other disadvantages. Here we will consider just two:

- We access the databases directly, bypassing the business logic. This makes it necessary to duplicate any business logic in all data transfer bridges, making the maintenance even more difficult. Accessing the database directly is risky because it is easy to break the business rules (and therefore the overall integrity).
- The integration based on data exchange between applications requires that a process that transfers data runs from time to time. We can do this every day, every hour, or even every few minutes, but we can hardly implement it continuously, because this would require too many resources.

Numerous examples for similar integration attempts can be found. Let us consider the demand for web-based customer front ends that have been and still are very popular. Basically, this was the attempt to provide Internet access to existing systems, often to more than one. Expressed in other words, this is nothing else but an integration project to integrate the newly developed front end with existing back-end applications. More explicitly, integration between existing systems is the foundation for implementing this kind of application. However, companies didn't have time to wait for the integration to be done first. Rather, they needed a web-based front end "over night". Companies that were late in identifying this need were in a particularly bad position.

Given that most companies did not have the integration solved in time, they again searched for fast solutions that were not based on solid foundations. It is most likely, such web-based front-end systems were implemented with point-to-point links to each existing application without a global understanding of the problem. Again, this solution looked successful for a short period of time, but a second look tells us otherwise—here we also see a point-to-point solution, which requires a lot of maintenance. Additionally, this time we also had to modify existing systems a little. We will talk more about this later when we consider extensions to existing systems.

What was not done in this project was to take the time to get a global understanding of the integration, the challenges, and the requirements that an integrated system has to fulfill. Rather, the integration was attempted with an ad hoc solution, without planning and architecting, and the data structures have remained separate and inconsistently defined. Enhancements to the systems will dispose of the management problems, and security questions will become important, not to mention other questions—transactional behavior, lifecycle, naming, and so on.

Actually, this is an interesting situation. Remember that the need for integration is primarily a consequence of the lack of global architecture and planning in the past. Thus, implementing the integration without planning is a choice only the most optimistic people will select—or people who are given little choice, because the management has made a decision. Management sometimes sees the purpose of the IT department to provide the solution in the minimum of time that is cost effective in the short term. Such management will often not want to hear that a longer term total integration solution would provide much better long-term results.

It is not reasonable to expect that attempting integration in the same way that has brought us un-integrated systems will this time result in usable integration architecture and solutions.

The complicated thing is that the company will probably not realize this early enough and will continue to support other projects in the same way. Soon the point of no return will be reached when additional layers are patched on top. The system will reach an unmanageable level of complexity where it will be impossible to modify them because a small modification will provoke the need for changes in so many tightly coupled parts.

This is why the bottom-up approach to integration will most likely bring the company a step forward at first, but soon the company will regret it, needing to go two or three steps backward. An information strategy that focuses on individual, uncoordinated integration projects will, over time, introduce even more problems into the company's information infrastructure. Thus, attempting to perform integration from the bottom-up approach is highly discouraged. In the following section, we'll see why the top-down approach is much better.

Top-Down Approach

Lack of planning and architecting was one of the principal reasons why many of today's companies have different applications running on different systems that are not interoperable in any ideal way. As such, it is unreasonable to expect that we can achieve an efficient level of integration without precise planning and designing.

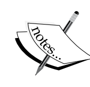
Information system building is essentially an engineering discipline. Therefore, let's consider taking a hint from civil engineering, defining the architecture before we begin. To use an analogy, we wouldn't start to build a house without plans, except maybe for a dog kennel! Indeed, there is no real damage done if a dog kennel falls down; it is not even a major problem if the dog does not like the house (we can always replace the dog!).

Extending our analogy further, let's consider the construction of a whole community of houses. When we start building a house in civilized surroundings, like in a city or a town, we cannot simply build the house anywhere we would like to. We have to position it according to environmental plans and designs. We also cannot build connecting roads to other houses in any random way we like. This would result in one-to-one communications between houses that would be unmanageable; it would be almost impossible to oversee their construction and navigate between them. Rather, we again have to stick to a global design where it has been chosen how the roads will be positioned and interconnected.

It is a similar situation in application integration. We cannot simply start to connect the existing applications without a good plan and architecture. This would almost certainly result in a large number of point-to-point connections that are very difficult to manage and maintain. Rather, we have to define the architecture of the integrated system first, and only then start to look at the existing applications and decide how they can fit into the integration architecture. This approach has several advantages and will enable us to build an architecture into which the existing applications will fit as well as new-generation systems. It will also enable us to replace and re-engineer existing systems with new developments. This way of thinking is known as the top-down approach to integration.

The top-down approach is nothing but a defined approach for defining global integration architecture. An integration architecture should, by definition, be comprehensive and define all foreseeable aspects of the business problems that can occur on the macro or micro scale. This effectively strengthens the integrity and consistency of integration throughout the whole company. The integration architecture should also recognize the dependencies between applications and the application development. It should provide guidelines and priorities by which the management will plan and schedule the tasks and requirements addressed to the IT department and the information system. It is infeasible, for example, to start to build a business-to-business integration before we have realized an adequate integration within our existing applications.

The top-down approach focuses on integration as a global activity. It defines a comprehensive integration architecture that defines all foreseeable aspects of the business problems related to integration, analyzes existing dependencies, sets guidelines, and defines priorities. It manages integration as a large coordinated project and gathers the efforts of all involved parties.



In real-world cases, often bottom-up and top-down approaches are used together, where the best practices of both are combined. Such an approach, often called an inside-out approach, reflects the requirements of the integration project and takes into account various criteria, including the existing applications, architectures, and requirements related to the new, integrated system.

Sound Integration Architecture Benefits

Sound integration architecture usually provides the following benefits:

- Reusability: Services perform a set of operations that are described and accessed through an interface. The interfaces enable the interoperability between services and promote the reuse of functionality.
- Encapsulation: Services on each tier are strongly encapsulated. The only way to access the service is through the interface. The client of the service does not know and does not have to know the internal details of a service. The encapsulation enables replacing the implementation of a service without influencing the rest of the system as long as the interface and the functionality remain unchanged.
- Distribution: The access to services is not limited to a single computer or even a single process. They can be distributed (and/or replicated) among computers without modifications in the source code of clients. Communication details are handled by the middleware layer, thus achieving location transparency.
- Partitioning: Putting the functionality in the appropriate tiers enables us to build thin clients, put the business logic into the middle tier, and solve the persistence in the back-end. By using the abstraction layers, we can achieve flexible deployment options and configurations.
- Scalability: Matching the capacity of the middle tier to the client demands by implementing performance optimization techniques in the middle tier and the ability to physically distribute and replicate the critical services enables good control over the scalability over a long time period.
- Enhanced performance: Applications can take advantages of server-side optimization techniques, like multiprocessing, multithreading, pooling, resource and instance management, and thread management, without changing code and allowing dynamic configuration changes.
- Improved reliability: Single points of failure, as well as bottlenecks, can be eliminated using replication and distribution.

- Manageability: With the separation in multiple tiers, it is much easier to locate the services that need to be modified. Most frequent are changes in the business logic that is located in a separate tier. The changes there do not require costly and time-consuming reinstallations. Rather, they can be managed centrally.
- Increased consistency and flexibility: As long as the interfaces between the tiers and the interfaces inside the tier stay unchanged, the code can be modified without impact to other parts of the system. In multi-tier architectures, it is much easier to adapt the information system to the changing business needs. A change in the business tier services will consistently effect all applications.
- Support for multiple clients: Different kinds of clients can access the business logic through the same interface.
- Independent development: Services can be developed independently of other services. Interfaces between services define the contract between them that enables independent development (as long as the contracts are respected).
- Rapid development: Application developers can focus on rapidly developing and deploying the business functionality, while remaining transparent to the underlying infrastructure. Services can be used in unpredictable combinations to form applications.
- Composition: Services can be composed in a variety of ways, which provides great flexibility and support for business processes.
- Configurable: Different implementations of the same service can be readily interchanged at run-time, enabling you to provide the capabilities that you need without redesigning applications.
- Security: Caution has to be taken that measures are applied to address different aspects of security, such as authorization, authentication, confidentiality, non-repudiation, etc.

Types of Integration

Integration architecture is usually built systematically in several layers. The idea behind this is to break the problem into several smaller problems and solve each sub-problem step by step (similar to, for example, the way in which network architecture is broken up into layers, as defined by ISO OSI). Therefore, today integration can be seen in several layers. We usually start building the integration architecture at the lowest layer and climb gradually. Omitting a layer is a short-term solution to speed up the process, but we will almost certainly have to pay back this time later. The most important types of integration are:

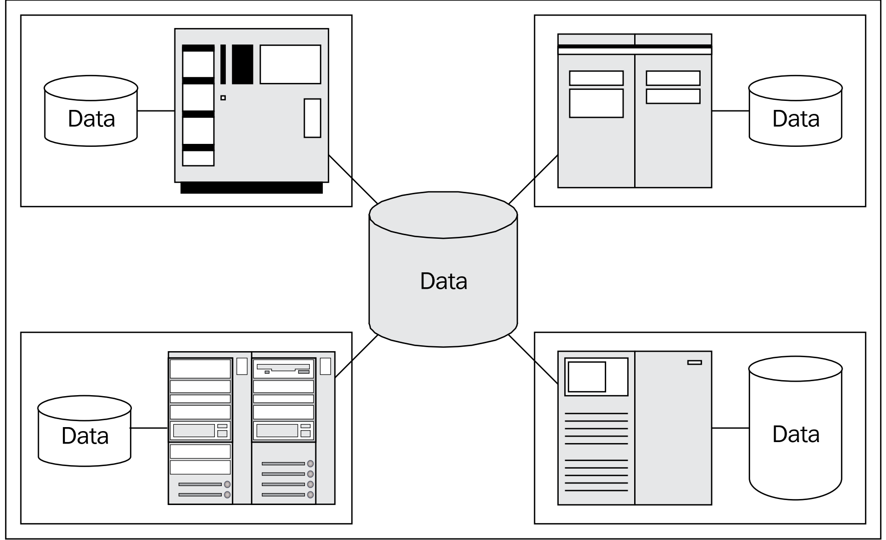
- Data-level integration
- Application integration
- Business process integration
- Presentation integration

These types of integration refer to integration within the enterprise as well as to the integration between enterprises (B2B). Let's now examine these types of integration individually.

Data-Level Integration

Data-level integration focuses on moving data between applications with the objective of sharing the same data among these different applications. It is often the starting point where a company starts to work on integration.

From a technical perspective, data-level integration is a relatively simple approach that is well understood by most developers. Accessing databases is relatively easy and there are several tools available that make the data sharing easier and faster. Also, data-level integration does not require changes to the applications. The following figure shows data-level integration.



Although the technology for accessing databases is well understood and not too difficult, the whole task of implementing data-level integration is not easy at all.

The problem lies in the complexity of the databases and in their number. To move the data between databases, it is not enough to be familiar with the technology. We also have to understand what data is stored in which database and in what form. We have to know when and how we can extract the data. Even more important is to be familiar with the type and structure of the destination database. Only then will we be able to store data in the database in a format that is understood by all the applications that use this database and that doesn't break the consistency of the database.

It is the semantics of the databases that is the most difficult part of data-level integration. It is not unlikely that we will have to deal with several hundred different databases. Some of them will belong to legacy applications, some of them to newer applications. Probably we will not be familiar with all of the applications. This also means that we will not be familiar with the structure of all of the data involved and the way they are stored.

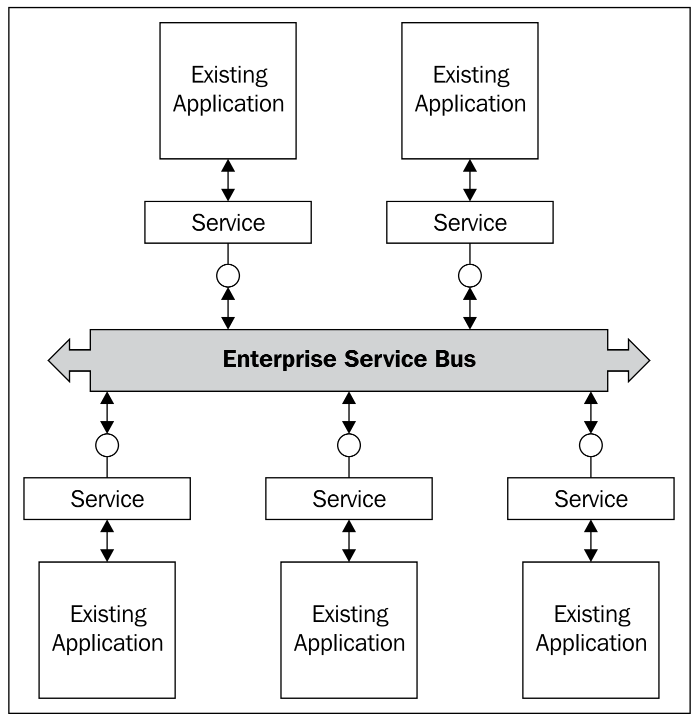
Sometimes we will have limited access to the databases because of contract restrictions. Then we will have to find other possibilities to access the data, for example using application programming interfaces (APIs), which we will discuss in the next section. If this is not applicable, we might be able to access flat files containing exported data and to add information using flat files that are imported by the target application. This, however, adds another layer of complexity to data-level integration.

Application Integration

Application integration focuses on sharing functionality—business logic; and not just pure data as in data-level integration. Application integration is usually achieved through the use of application programming interfaces (APIs). Applications that expose their functionality through APIs enable access to the functionality in a programmatic way without using the user interface.

Previously, developers did not realize the usefulness of APIs. Therefore most older applications do not have them. Since then, most newer applications have accepted the concept of services that an application can provide to other applications. In newer existing applications, it will be more likely to find APIs. Through the use of APIs, we can access the functionality of existing systems. However, the APIs exposed by different existing applications will differ in the way we have to access them and which technology we have to use to access them.

The objective of application integration is therefore twofold: to understand and use APIs for accessing the required functionalities; and to mask the technology differences between different technologies used for APIs and their access. The later is achieved using services, which expose the interfaces (APIs), as shown in the following figure.



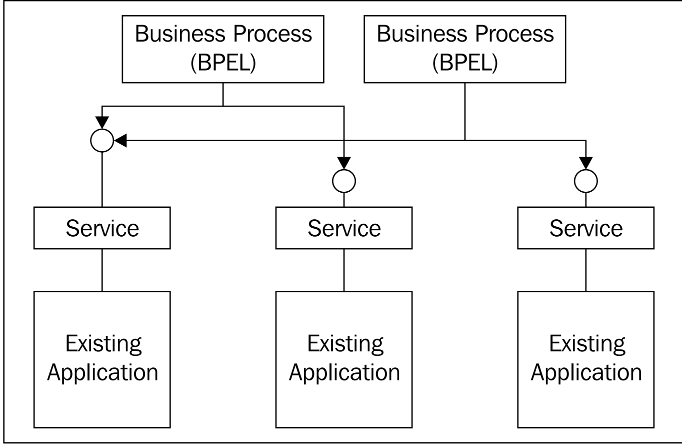
Please notice that the interfaces provide the contract between the applications. As long as the interfaces stay unchanged, this means that the contracts have not been changed. But it also means that the interfaces are the entities that tie different parts of the information system together. Nobody cares how the interfaces are implemented, or what applications actually execute in the background. We can change the applications that implement a certain interface without influencing the whole information system or partial applications—as long the interface stays unchanged.

Therefore, great care has to be taken in the definition of interfaces. Today, we understand good interfaces as those that are loosely coupled. This can be achieved primarily by sharing data only, without behavior; structuring the data; and using open standard technologies. We will come back to this aspect later in this book.

Business Process Integration

Business process integration enables non-compromise support for business processes in the enterprise where existing solutions take part in distinctive steps of the business process. It exposes the functionality as abstractions of business methods through interfaces.

Business process integration presents the enterprise-wide information system as we would like to have it—or as we would build it if we could build it anew, with clear requirements for what we would like to get from the integrated system and with the knowledge and support of modern technologies. This means that the information system interfaces are based on a new designed architecture. However, the functionalities are not re-implemented; rather, they use existing applications. Those existing applications are remodeled in a way that they expose the functionality of the business process tier and fit into the modern application architecture. Finally, the different pieces are glued together, usually by using a business process modeling and execution language, such as BPEL (Business Process Execution Language), as shown in the following figure.



SOA, BPEL, and related technologies today provide new opportunities for making integrated information systems more flexible and adaptable to business process changes. This way our information systems can get more agile, provide better support for changing requirements, and align closer to business needs. We have to be aware that achieving business process integration is often connected with business process reengineering and is not a sole technical problem. It, however, requires the implementation of several technical layers as the foundation and integrates applications at a higher-level of abstraction.

Presentation Integration

Often, after achieving business process integration, we will continue with the presentation integration. Because existing applications are now remodeled and encapsulated on the middle tier, where they expose their services through high-level interfaces, it becomes crucial that the user gets a unified view of the information system as well. As long as the user has to switch between legacy applications, they should be fully aware that they are using old applications.

Presentation integration results in an integrated system that provides a unified presentation layer, through which the users can access the functionality of the integrated system. Because they use the newly developed presentation layer, they are not aware of the diversity of existing applications that are executing in the background. The presentation layer also accesses the functions through common interfaces, provided by business tier, developed in the business process integration phase. Therefore, the presentation layer is decoupled and not aware of the details of existing applications.

With the development of a unified presentation tier, we hide the fact that in the background different applications, some legacy and other new developed, are executing. This way, we improve the efficiency of end users and provide a way to replace parts of legacy systems in the future without influencing the other parts of the system.

We will consider presentation integration as a step in which we define and implement a common user interface—usually a portal—for the business-method-level integrated information system, such that the user interface will provide the illusion of a new system, and add the missing piece to the multi-tier integration architecture. We should not look at presentation integration as simple user interface integration, adding web or graphical user interfaces, which we may have covered by application extension. We should also not consider presentation integration only as a way to extract the data from existing applications through the user interfaces.

Business-to-Business Integration

Today, the integration of applications inside a company is not sufficient. There are growing needs to enable inter-enterprise integration, often referred to as business-to-business (B2B) integration, or e-business. E-Business places particular new challenges for an information system. The requirements today are very high and gone are the days when a company could just publish off-line catalogs on its web pages. What is expected is online, up-to-date information, efficiency, reliability, and quality. Even well known names from traditional business cannot expect that their position will be maintained in an e-business environment without effort.

Of course, the prerequisite for efficient e-business or B2B integration is an integrated enterprise information system, possibly on the business-process level, which must be at both ends. Only this level of integration enables on-demand processing of requests. Customers today are expecting immediate response and are not satisfied with batch processing and several days of delays in confirming orders, for example. However, this is often the case, when e-business is not backed by an efficiently integrated enterprise information system. Immediate responsiveness, achieved by highly coupled integration of back-end (enterprise information) and front-end (presentation) systems is a key success factor.

Although this sounds obvious, research from leading consulting companies like Gartner Group shows that today there are very few front-end systems that are efficiently integrated with the back end. Most of these non-integrated applications will fail to meet the business expectations. The primary reason is the lack of enterprise integration, which is the most important prerequisite for both a successful e-business and an efficient company.

Another important fact is that most front-end applications can use existing and legacy systems as back-end solutions. Making the integration between such systems efficient will be the key success factor. In particular, immediate response and immediate propagation of data to all related applications will be the major problem. Front-end applications not efficiently supported by back-end systems will certainly fail to meet all requirements.

Integration Infrastructure

Let us now focus on the required infrastructure services for integration. We will identify the services from a high-level perspective and separate them into the horizontal and vertical layers. The services in horizontal layers will provide basic infrastructure services useful for the majority of existing and new-generation applications. The vertical layer services will provide functionalities related to a specific task within infrastructure that can span through several horizontal layer services.

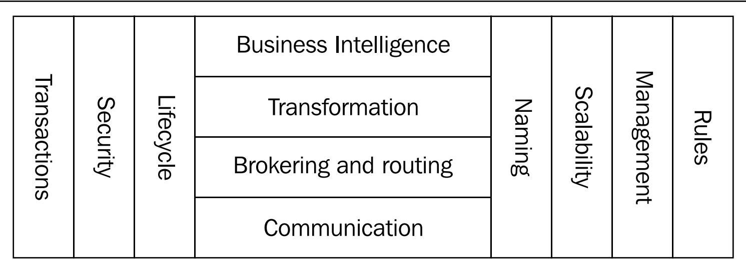
The services on the horizontal layer include:

- Communication
- Brokering and routing
- Transformation
- Business intelligence

The vertical layers are:

- Transactions
- Security
- Lifecycle
- Naming
- Scalability
- Management
- Rules

The relations between the services are shown in the following figure:



Let's now examine these services in finer detail, starting with the four horizontal layers that we've mentioned above.

Communication

The primary responsibility of the communication service is to provide the abstraction for communication details. It provides the transparency for accessing different remote systems and unifies the view on them. This ensures that the developers do not have to deal with the low-level communication details. As the communication layer does not execute business logic, it enables the separation of business logic and the communication services, but allowing communication between them.

Different types of middleware provide different communication layer services. The most commonly used for application integration are the database access technologies, like JDBC, that provide the data-layer abstraction to access different databases through a unified view. Message-oriented middleware (MOM) provides asynchronous communication through sending and receiving messages through a message queue or a message channel. The remote procedure call provides

communication services for synchronous, procedural-oriented communication. Somewhat similar are the object request brokers, which provide an object-oriented view on the distributed entities. Enterprise services busses (ESB) are the latest reincarnation of the integration broker targeted to fulfill the objectives of SOA.

Typically, all mentioned middleware uses certain standard or custom protocols to achieve communication. These protocols include SOAP (Simple Object Access Protocol), HTTP, TCP/IP, IIOP (Internet Inter-ORB Protocol), but also proprietary protocols.

The communication layer can also provide location transparency. This means that the actual location is managed separately from the application logic. This enables flexibility through deployment and configuration. The location transparency functionality is often connected with the introduction of the naming and directory services, which are then used as a repository for storing such information.

Brokering and Routing

The brokering and routing layer takes care of implementing the technical side of integration. No matter what type of integration we use, this layer should adapt the communication between applications in such way that all participating applications will be able to interoperate. This layer is essential for integration and actually has a number of responsibilities.

First, it has to provide the way to gather the required data from multiple sources, most likely existing and new-generation application and/or data stores. This responsibility is called aggregation, because data is gathered from different sources to represent a business concept, like an invoice or order.

Then, this data will have to be processed. Again, we will use a mix of existing and new-generation systems. Each of these applications will probably have its own interfaces and message syntax. Therefore, the brokering and routing layer will have to transform the data and messages into suitable parts that can be processed by individual applications.

Finally, this layer will have to gather the results of all applications and present them consistently. This part is called synthesis, meaning combining of results into a meaningful business notion.

To achieve these three steps automatically, the brokering and routing layer will need metadata information that will define the participating applications, methods, messages, and interfaces, and the sequence of operations involved. This layer also has to provide a means of handling events, making it appropriate for a declarative environment. Usually, it will associate events with certain operations that have to be performed.

Transformation

Transformation of data structures, their representations, and technologies has always been very important. In the past, small, custom-written programs that have read the source and transformed it to the destination format have usually solved the problems related to the transformation. With the advent and global use of markup languages, particularly XML, which has become the de-facto standard for data exchange, transformations have achieved a new level of maturity.

Therefore, transformation is today considered as a service that has to be provided by the integration infrastructure. Transformation engines are usually based on XSLT (Extensible Stylesheet Language for Transformations), which allows a relatively easy specification of data and schema transformations, and even allows specifying transformation rules as templates in a declarative way. Another advantage is that XSLT transformations can be executed on any XSLT engine, making transformation independent of the programming language, platform, and other restrictions.

Usually, XSLT transformation is made easier with the support of several tools, which provide graphical editors that allow us to graphically construct transformations using drag-and-drop techniques. These tools are becoming part of development environments on one hand, and part of integration technologies, such as ESBs on the other hand, which further simplifies transformations.

Business Intelligence

The business intelligence layer is responsible for presenting the high-level interface to access business information to other applications and to the users. The business-intelligence layer presents data to users in an understandable form. With the growth of e-commerce, the business-intelligence layer also takes some responsibilities for B2B integration.

Today, the business-intelligence layer is supported by a flexible unified presentation tier, most likely in the form of personalized portals. Personalized portals enable the delivery of valuable personalized business data and content directly to employees, business partners, and customers.

In addition to data and content delivery, the business-intelligence layer is often connected with data-processing technologies like Online Analytical Processing (OLAP), data mining, decision-support systems, and executive-information systems. These sources analyze enterprise data and provide information like estimation, forecasting, time-series analysis, and modeling.

Next, we'll turn our attention to the vertical service layers that define the integration infrastructure.

Transactions

The integration infrastructure has to provide the means for carrying out the business operations in a transactional manner. Therefore, it has to be able to invoke several operations on different existing and new-generation systems. It has to support the atomic ACID transaction model and long-running transactions with compensation semantics, usually referred to as business activities.

So, adherence to the transactional semantics means that any operation performed on one or more applications that causes a state change or changes to permanent data has to be performed as operation that guarantees that the consistency of the system is preserved. It also has to isolate the operation from other operations to a certain degree and guarantee that the outcomes of operations are written to the persistent storage.

Security

The integration infrastructure has to provide ways to constrain access to the system. Similarly, as the integration infrastructure horizontal services define a unified way to access the different applications, they should also define a way in which to manage security, possibly in a declarative way. The security should include all four horizontal layers. It should be able to reuse the existing application security, and base the security on roles that are defined with a single user log in. The security system should not be based on different passwords for different applications or even parts of applications. It should relate to all the important aspects, like communication channel encryption, authentication, authorization, and auditing.

Lifecycle

The integration infrastructure should provide ways to control the lifecycle of all applications involved. It should enable existing applications to be replaced one by one or even by parts without having influence on the other applications in the integrated system. The replacement should be possible step by step, when business needs dictate it and when enough resources are available. It should also provide ways to do the replacement while the system stays online. This functionality is often achieved by minimizing the dependencies between applications and specifying ways for the applications to interoperate.

Naming

A unified naming service will allow for the implementation of location transparency and will enable the replacement of one resource with another if this is required. The naming service in usually implemented with a naming and directory product that enables storing and looking for name-related information. Ideally, the naming service is unified and provides one logical picture of the enterprise, although it is physically implemented using replication and distribution to avoid a single point of failure.

Scalability

The integration infrastructure should be designed with scalability in mind. It has to access information about clients and provide concurrent access to the applications. It has to incorporate solutions that will enable enough room for extending the load demands on the system. Achieving scalability in an integrated system can be a difficult problem because we have to take into account existing applications that probably have not been designed for the kind of scalability that we would like to achieve. Particularly, the requirements for the number of concurrent clients probably have not been so strict. Therefore, we should think of scalability and probably implement some prototypes to test what levels of performance we can expect. We should also the load testing tools that will enable us to simulate high loads and assess the relevant performance criteria.

The integration infrastructure, however, cannot fully compensate for bad application architecture and design. Therefore, we will have to asses the architecture of existing applications to discover how well they scale. For newly developed applications, on the other hand, we should follow sound design and performance-related practices to achieve the best possible scalability.

Management

We also have to provide ways to manage the integration infrastructure. Many solutions, particularly custom applications, leave this out, which results in difficulties at the maintenance stage. The management layer should provide methods and tools to manage horizontal and vertical services. It should provide easy configuration and version management. A declarative management system enables access for changing and updating the parameters without needing to modify the source code and re-deploy the solutions. Remote management enables the infrastructure management to be carried out from remote locations, which minimizes the need for trained personal on site.

Rules

The horizontal services require specific rules for performing communication, brokering, routing, and business-intelligence tasks. These rules should not be hard-coded into applications, but should rather be declaratively specified inside the integration infrastructure. This includes the definitions, data formats, data transformations and flows, events, information processing, and information representation. Often these rules are stored in a repository, which provides a centralized storage to avoid duplication and inconsistencies.

Integration Technologies

Comprehensive enterprise-wide integration infrastructure usually requires more than one technology. Typically, also, because of the existing technologies, we will have to use a mixture of technologies. When selecting and mixing different technologies, we have to focus on their interoperability.

Interoperability between technologies will be crucial because we will use them to implement the integration infrastructure. Achieving interoperability between technologies can be difficult even for technologies based on open standards. Small deviations from standards in products can deny the "on-paper" interoperability. For proprietary solutions, interoperability is even more difficult. It is not only the question of if we can achieve interoperability, but also how much effort we have to put in to achieve it. Technologies used for integration are often referred to as middleware.

Middleware is system services software that executes between the operating system layer and the application layer and provides services. It connects two or more applications, thus providing connectivity and interoperability to the applications. Middleware is not a silver bullet that will solve all integration problems. Due to over-hyping in the 1980s and early 1990s, the term "middleware" has lost popularity, but is coming back in the last few years. The middleware concept, however, is today even more important for integration, and all integration projects will have to use one or many different middleware solutions. Middleware is mainly used to denote products that provide glue between applications, which is distinct from simple data import and export functions that might be built into the applications themselves.

All forms of middleware are helpful in easing the communication between different software applications. The selection of middleware influences the application architecture, because middleware centralizes the software infrastructure and its deployment. Middleware introduces an abstraction layer in the system architecture and thus reduces the complexity considerably. On the other hand, each middleware

product introduces a certain communication overhead into the system, which can influence performance, scalability, throughput, and other efficiency factors. This is important to consider when designing the integration architecture, particularly if our systems are mission critical, and are used by a large number of concurrent clients.

When speaking of middleware products, we encompass a large variety of technologies. The most common forms of middleware are:

- Database access technologies
- Message-oriented middleware
- Remote procedure calls
- Transaction processing monitors
- Object request brokers
- Application servers
- Web services
- Enterprise service buses
- Several hybrid and proprietary products

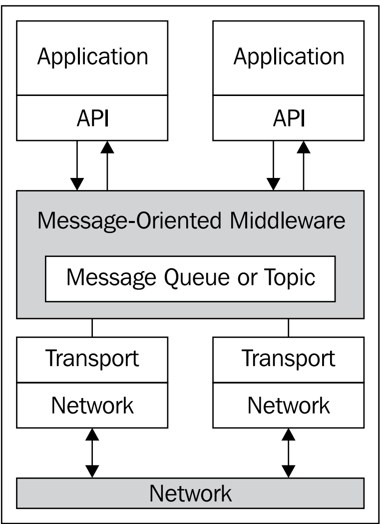
Database Access Technologies

Database access technologies provide access to the database through an abstraction layer, which enables us to change the actual DBMS without modifying the application source code. In other words, it enables us to use the same or similar code to access different database sources. Therefore, database access technologies are useful for extracting data from different DBMSs. The technologies differ in the form of interfaces to the database they provide. They can offer function-oriented or object-oriented access to databases. The best known representatives are Java Database Connectivity (JDBC) and Java Data Objects (JDO) on the Java platform, and Open Database Connectivity (ODBC) and Active Data Objects (ADO.NET) on the Microsoft platform.

Message-Oriented Middleware

Message-oriented middleware (MOM) is a client/server infrastructure that enables and increases interoperability, flexibility, and portability of applications. It enables communication between applications over distributed and heterogeneous platforms. It reduces complexity because it hides the communication details and the details of platforms and protocols involved. The functionality of MOM is accessed via APIs. It typically resides on both ends, the client and the server side. It provides asynchronous communication and uses message queues to store the messages temporarily. The applications can thus exchange messages without taking care of the

details of other applications, architectures, and platforms involved. The messages can contain almost any type of data, asynchronous nature of communication enables the communication to continue even if the receiver is temporary not available. The message waits in the queue and is delivered as soon as the receiver is able to accept it. But asynchronous communication has its disadvantages as well. Because the server side does not block the clients, they can continue to accept requests even if they cannot keep pace with them, thus risking an overload situation. The basic architecture is shown in the following figure.

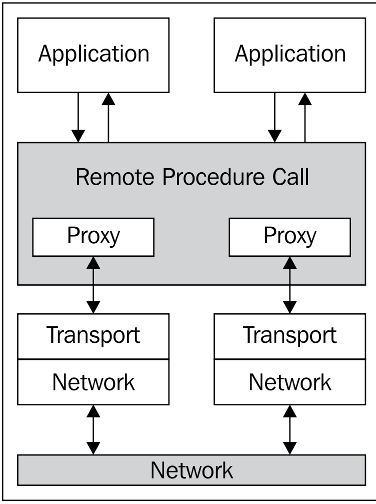


MOM products are proprietary products and have been available from the mid eighties. Therefore, they are incompatible with each other. Using a single product results in dependence on a specific vendor; this can have negative influence on flexibility, maintainability, portability, and interoperability. MOM product must specifically run an each and every platform being integrated. Not all MOM products support all platforms, operating systems, and protocols. We will, however, see that the Java platform provides ways to achieve relatively high independence from a specific vendor through a common interface, used to access all middleware products—the Java Messaging Service (JMS).

Remote Procedure Calls

Remote procedure calls are also a client/server infrastructure intended to enable and increase interoperability of applications over heterogeneous platforms. Similar to MOM, it enables communication between software on different platforms and hides almost all the details of communication. RPC is based on procedural concepts—developers use remote procedure or function calls. The first implementations date back to the early 1980s.

The main difference between MOM and RPC is the manner of communication. While MOM supports asynchronous communication, RPC promotes synchronous, request-reply communication (sometimes referred to as "call/wait"), which blocks the client until the server fulfills its requests. The next figure shows two applications communicating using RPC. To achieve remote communication, applications use procedure calls. RPC middleware hides all communication details, which makes using remote procedure calls very similar to local procedure calls.



RPC guards against overloading a network, unlike the asynchronous mechanism, MOM. There a few asynchronous implementations available, but they are more the exception than the rule. RPC increases the flexibility of architecture by allowing a client of an application to employ a function call to access a server on a remote system. RPC allows the remote access without knowledge of the network address or any other lower-level information. The semantics of a remote call is the same whether or not the client and server are collocated. RPC is appropriate for client/server applications in which the client can issue a request and wait for the server to return a response before continuing with its own processing. On the other hand, RPC requires that the recipient is online to accept the remote call. If the recipient fails, the remote calls will not succeed, because the calls will not be temporarily stored and then forwarded to the recipient when it is available again, as is the case with MOM.

RPC is often connected with the Distributed Computing Environment (DCE), developed by the Open Systems Foundation (OSF). DCE is a set of integrating services that expand the functionality of RPC. In addition to RPC, the DCE provides directory, time, security, and thread services. Over these fundamental services, it places a layer of data-sharing services, including distributed file system and diskless support.

Transaction Processing Monitors

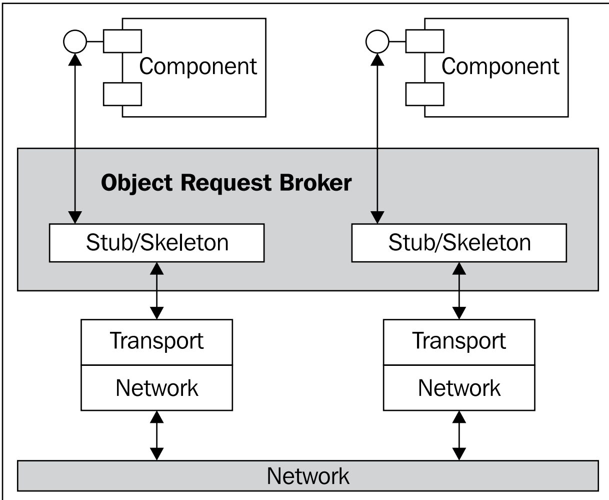
Transaction processing (TP) monitors are important middleware technology in mission-critical applications. They represent the first generation of application servers. TP monitors are based on the concept of transactions. They monitor and coordinate transactions among different resources. Although the name suggests that this is their only task, they have at least two very important additional roles: providing performance management and security services. They provide performance management with load balancing and resource pooling techniques, which enable efficient use of computing resources and thus a larger number of concurrent clients. TP monitors map client requests through application service stateless routines to improve system performance. They can also take some application transition logic from the client. They also provide security management where they enable or disable access of clients to certain data and resources. TP monitors can be viewed as middle-tier technology and this is why they are predecessors of todays application servers.

TP monitors have been traditionally used in legacy information systems. They are based on the procedural model, use remote procedure calls for communication between applications, and are difficult to program because of complex APIs through which they provide the functionality. In spite of that, they have been successfully used for more than 25 years. TP monitors are proprietary products, which makes migration from one product to another very difficult.

Object Request Brokers

Object request brokers (ORBs) are a middleware technology that manages and supports the communication between distributed objects or components. ORBs enable seamless interoperability between distributed objects and components without the need to worry about the details of communication. The implementation details of ORB are not visible to the components. ORBs provide location transparency, programming language transparency, protocol transparency, and operating system transparency.

The communication between distributed objects and components is based on interfaces. This enhances maintainability because the implementation details are hidden. The communication is usually synchronous, although is can also be deferred synchronous or asynchronous. ORBs are often connected with location services that enable locating the components in the network. ORBs are complex products but they manage to hide almost all complexity. More specifically, they provide the illusion of locality—they make all the components appear to be local, while in reality they may be deployed anywhere in the network. This simplifies the development considerably but can have negative influence on performance. A basic outline of ORB architecture is shown in the next figure:



ORB products may choose different scenarios as to how and where to implement their functionality. They can move some functions to the client and server components or they can provide them as a separate process or integrate them into to the operating system kernel. There are three major standards of ORBs:

- OMG CORBA ORB compliant
- Java RMI and RMI-IIOP
- Microsoft COM/DCOM/COM+/.NET Remoting/WCF

There are many ORB products compliant with the CORBA ORB specifications and various implementations of RMI and RMI-IIOP. Particularly, RMI-IIOP is important, because it uses the same protocol for communication between components as the CORBA ORB, namely the IIOP (Internet Inter-ORB Protocol). This makes RMI-IIOP interoperable with CORBA.

Application Servers

Application servers handle all or the majority of interactions between the client tier and the data persistence tier. They provide a collection of already mentioned middleware services, together with the concept of a management environment in which we deploy business logic components—the container. In the majority of application servers, we can find support for web services, ORBs, MOM, transaction management, security, load balancing, and resource management. Application servers provide a comprehensive solution to enterprise information needs. They are also an excellent platform for integration. Today, vendors often position their application servers as integration engines, or specialize their common purpose application severs by adding additional functionality, like connections to back-end and legacy systems and position their products as integration servers. Although such servers can considerably ease the configuration of different middleware products, it is still worth thinking of what is underneath.

Whether used for integration or new application development, the application servers are software platforms. A software platform is a combination of software technologies necessary to run applications. In this sense, the application servers, or more precisely the software platforms that they support, define the infrastructure of all applications developed and executed on them. Application servers can implement some custom platform, making them the proprietary solution of a specific vendor (these are sometimes referred to as proprietary frameworks). Such application servers are more and more rare.

On the other hand, application servers can support a standardized, open, and generally accepted platform, such as Java enterprise Edition. The following lists the most important aspects of a platform:

- Technical aspects define the software technologies that are included in the software platform, the architecture of the applications developed for that platform, interoperability, scalability, portability, availability, reliability, security, client contracts, possibilities to grow and accommodate new solutions, and so on. In terms of integration, a very important aspect is the interoperability with other systems.
- Openness enables the application server vendors and third-party companies to have some possibility of influencing the development of the platform. Different solutions exist, from fully closed platforms that bind us to a certain vendor, to the fully open platforms, for example the open-source initiative, where everything, even source code, is free and can also be freely modified. Open platforms are often defined with specifications. These are documents that strictly define the technologies included in the platform and enable different vendors to implement the platform (as application servers for example). A tight specification guarantees consistency and a platform defined in terms of specifications can also have a reference implementation and a set of compatibility tests.
- Interoperability among platform implementations is crucial for the adoption of a certain platform. Particularly, the way the platform regulates additions and modifications is crucial. The stricter a platform is with the implementation of the core specification, the better chances it has to be successful and to gain a large market share. Each platform, however, has to provide ways for application servers to differentiate their product, possibly through implementing some additional functionality.
- Cost of the platform is also an important factor and it is probably the most difficult to assess because it includes the cost of the application server and other development software, the cost of hardware, the training, and the cost of the maintenance of the applications through their lifecycle.
- Last (and perhaps least important) is maturity, from which we can predict how stable the platform is. The more mature the platform is, the more it has been tested and the more has been proved that it is suitable for large scale applications.

Web Services

Web services are the latest distributed technology. They provide the technological foundation for achieving interoperability between applications using different software platforms, operating systems, and programming languages. From the technological perspective, web services are the next evolutionary step in distributed architectures. Web services are similar to their predecessors, but also differ from them in several aspects.

Web services are the first distributed technology to be supported by all major software vendors. Therefore, it is the first technology that fulfills the universal interoperability promise between applications running on disparate platforms. The fundamental specifications that web services are based on are SOAP (Simple Object Access Protocol), WSDL (Web Services Description Language), and UDDI (Universal Description, Discovery, and Integration). SOAP, WSDL, and UDDI are XML based, making web services protocol messages and descriptions human readable.

From the architectural perspective, web services introduce several important changes compared to earlier distributed architectures. They support loose coupling through operations that exchange data only. This differs from component and distributed object models, where behavior can also be exchanged.

Operations in web services are based on the exchange of XML-formatted payloads. They are a collection of input, output, and fault messages. The combination of messages defines the type of operation (one-way, request/response, solicit response, or notification). This differs from previous distributed technologies. For more information, please refer to WSDL and XML Schema specifications.

Web services provide support for asynchronous as well as synchronous interactions. They introduce the notion of end-points and intermediaries. This allows new approaches to message processing. Web services are stateless and utilize standard Internet protocols such as HTTP (Hyper Text Transfer Protocol), SMTP (Simple Mail Transfer Protocol), FTP (File Transfer Protocol), and MIME (Multipurpose Internet Mail Extensions). So, connectivity through standard Internet connections, even those secured with firewalls, is less problematic.

In addition to several advantages, web services also have a few disadvantages. One of them is performance, which is not as good as distributed architectures that use binary protocols for communication. The other is that plain web services do not offer infrastructure and quality of service (QoS) features, such as security, transactions, and others, which have been provided by component models for several years. Web services fill this important gap by introducing additional specifications:

- WS-Security: Addresses authentication and message-level security, and enables secure communication with web services.
- WS-Coordination: Defines a coordination framework for web services and is the foundation for WS-AtomicTransaction and WS-BusinessActivity.
- Transaction specifications (WS-AtomicTransaction and WS-BusinessActivity): specify support for distributed transactions with web services. AtomicTransaction specifies short duration, ACID transactions, and BusinessActivity specifies longer running business transactions, also called compensating transactions.

- WS-Reliable Messaging: Provides support for reliable communication and message delivery between web services over various transport protocols.
- WS-Addressing: Specifies message coordination and routing.
- WS-Inspection: Provides support for dynamic introspection of web service descriptions.
- WS-Policy: Specifies how policies are declared and exchanged between collaborating web services.
- WS-Eventing: Defines an event model for asynchronous notification of interested parties for web services.

Enterprise Service Buses

An Enterprise Service Bus (ESB) is a software infrastructure acting as an intermediary layer of middleware that addresses the extended requirements that usually cannot be fulfilled by web services, such as integration between web services and other middleware technologies and products, higher level of dependency, robustness, and security, management, and control of services and their communication.

An ESB addresses these requirements and adds flexibility to communication between services, and simplifies the integration and reuse of services. An ESB makes it possible to connect services implemented in different technologies (such as EJBs, messaging systems, CORBA components, and legacy applications) in an easy way. An ESB can act as a mediator between different, often incompatible, protocols and middleware products.

The ESB provides a robust, dependable, secure, and scalable communication infrastructure between services. It also provides control over the communication and control over the use of services. It has message interception capabilities, which allow us to intercept requests to services and responses from services and apply additional processing to them. In this manner, the ESB acts as an intermediary.

An ESB usually provides routing capability to route the messages to different services based on their content, origin, or other attributes and transformation capability to transform messages before they are delivered to services. For XML-formatted messages, such transformations are usually done using XSLT (Extensible Stylesheet Language for Transformations) or XQuery engines.

An ESB also provides control over the deployment, usage, and maintenance of services. This allows logging, profiling, load balancing, performance tuning, charging for service use, distributed deployment, on-the-fly reconfiguration, etc. Other important management features include the definition of correlation between messages, definition of reliable communication paths, definition of security constraints related to messages and services, etc.

An ESB should make services broadly available. This means that it should be easy to find, connect, and use a service irrespective of the technology it is implemented in. With broad availability of services, an ESB can increase reuse and can make the composition of services easier. Finally, an ESB should provide management capabilities, such as message routing, interaction, and transformation, which we have already described.

The Integration Process

Integration problems are usually very complex. Just imagine how many different people have to take part and on how many different levels. The business units have to coordinate the integration efforts, and define the information exchange and technical baselines. The information provided by all business units should also be consistent. Redundancies in the data model have to be identified, and removed on a company-wide level. Physical and electronic supply chains should be based on the same data views and all business units should behave to a common pattern, giving the same answers to the same questions.

Choosing the Steps and Defining the Milestones

Integration can be seen as a rather special software development project, because a lot of attention will have to be put on existing applications. Unfortunately, it is not a simple or small project, but usually a large project that will need a relatively long time to finish. During its course, it will consume a lot of resources, like developer time and money, so it is important that integration is done and managed in a defined and consistent way. In other words, the integration project has to follow sound software development practices.

Therefore, the integration project should be started and managed similarly to a large software development project. The integration project, however, in some aspects differs considerably from software development projects that do not have to take existing applications into account.

The quality of an information system is only as good as the architecture that underlies it. The investment in information system development will have a much greater long-term payback when we consider the needs of a family of problems rather than just the problem of the moment.

Before we continue the discussion of the integration project, let's consider what the possible roles are we will most likely find ourselves in:

- We might be employed in the IT department of a large company that needs to integrate its existing systems into a global, integrated information system. In this case, the major responsibility will fall on the IT department, which will be responsible for realizing the integration project.
- We might be part of a software company that is hired to do the integration for a certain company. The responsibility for successfully realizing the integration will be on the software company completely.
- The third possibility is that we will be hired by a company that is performing the integration, only to modify some existing application for the purposes of integration. In this case, we will not be responsible for the whole integration, but only for the application that we have to modify.

From these three roles, the third scenario is obviously the easiest. If we are responsible for a single application only, then we will be involved in integration only partially. Setting up the integration project, defining the integration architecture, infrastructure, and most other difficult tasks will be performed by the company that has hired us. We will have to follow the specifications that we receive to help to modify the existing application in order to fit into the integration architecture.

The first scenario, where the IT department takes over the integration project for its company, is more difficult. The responsibilities of the IT department will be large and will include setting up the integration project, designing the integration architecture, selecting the infrastructure, analyzing the existing applications, developing the integrated information system, etc. The IT department will usually be familiar with existing applications that the company has. It will also have direct contact to employees, which will simplify a little finding out the requirements. In contrast, for the IT department this is very likely the first integration project, therefore the experience is minimal.

The second scenario is the most difficult one. If we are a software company that takes over the integration project, we will have similar responsibilities to the IT department. That is, setting up the integration project, designing the integration architecture, selecting the infrastructure, analyzing the existing applications, developing the integrated information system, etc. The software company will also not be familiar with the existing applications, with their functionality, and all finer details. Getting this information might be difficult if the employees are not cooperative in the company for which we perform the integration. However, on the plus side, we will probably have a lot of experience from previous integration projects, and experience with many different middleware technologies.

Sound Practices

The integration process defines the sequence of activities that have to be done in a disciplined manner in order to successfully develop an integrated information system. The goal of an integration process is to guarantee high quality of the integrated solution that will satisfy the customer and will be done on schedule and within the allocated financial resources. An integration project is based on an integration process. A well-managed integration project, based on the proposed integration process outline, will provide the basis for us to plan, implement, and finish the integration successfully.

The integration process is tightly connected to the software development process, with which it shares several activities. Each company that develops software has defined a software development process. The software development process quality and maturity (as assessed with the Capability Maturity Model (`http://www.sei.cmu.edu/cmm/cmm.html`), for example) will have impact on the ability to accomplish the integration successfully. A better defined development process will reach a higher maturity level, and the chances that the integration will be successful will become much better.

Before we look into the details of integration process activities, let's first look at the sound development practices that are particularly important for integration. There are four important practices that should be considered in each integration project:

- Iterative development
- Incremental development
- Prototyping
- Reuse

Let's take a closer look at each of these considerations, in turn.

Iterative Development

We have already identified that the integration project will be a very large project that will require a lot of time and resources to finish. On the other hand, we have also seen that no company will be willing to wait too long (for example a year or more) from the initiation of the integration project until the usable integrated solutions are deployed. Iterative development solves this problem.

When using iterative development, we do not look at the integration as one large whole, but rather partition it into several small pieces, which we implement step by step.

Instead of waiting for the solution until the end of the project, we are able to deliver intermediate results. When developing iteratively, we can evaluate the results of every iteration, and improve them in the next iterations (look also at incremental development). This also enables us to assess the progress level.

Iterative development is a good practice also when the requirements are not completely defined. As we will see, today it is practically impossible to define the requirements at the beginning of integration and not to change or modify them later. With iterative development, we can relatively easily adapt to changing requirements, because we can apply changes to the intermediate releases.

Although iterative development is more difficult to manage for the project manager, it allows us to build a trust relationship between the developers and the customers. Customers will be able to see and use intermediate results. However, it might be difficult to assess how many iterations we will need, and limiting the number of iterations is not the correct solution to reduce the schedule.

Incremental Development

With iterative development, we have partitioned the integration into several smaller tasks that will deliver some usable products. However, it is unlikely that we will solve each task in our first attempt adequately. Usually, we learn from our own mistakes and improve ourselves. In other words: "Before we make things right, we make them wrong."

Incremental development is a strategy to improve in small steps to finally reach the stated goals. This requires that we partition the problem into sub-problems (iterative development). Incremental development enables us to work faster and to return to the same problem later to improve it.

Although incremental and iterative development strategies are commonly used together, they are actually two different strategies. Iterative development partitions the problem into sub-problems, while incremental development supplements and improves partial solutions step by step. Iterative and incremental development strategies are important to integration and enable us to partition the integration into several sub-problems and allow us to manage the changing requirements. If possible, they should be used combined for integration projects.

Prototyping

Prototyping is a strategy that enables us to find the correct solutions for a given problem without spending too much time and money on the problem. With prototyping, we can build pilot solutions to integration challenges and assess whether they can be developed in the way we would like to and with the technologies we would like to use. Prototyping is commonly used to assess, verify, and validate chosen architectures and solutions for integration. Often prototyping is used to verify the requirements and the performance and scalability as well.

Reuse

Reuse is the ability to develop new applications with the use of existing solutions. This is also the exact goal of integration; to develop an integrated information system via the reuse of existing applications, without modifying them too much. The problem is that to achieve reuse we must have software components that have been developed specifically for this purpose. The majority of existing systems are not designed with this in mind, so we will have to search for solutions that will make existing systems reusable. When discussing the practice of reuse, we should also mention reuse of a higher abstraction layer—the reuse of ideas and sound solutions in the form of pattern reuse.

Following these practices will enable us to achieve integration successfully and deliver partial results quickly. This is an important fact, because companies that start integration are not able to wait a long time (it can take a few years in a large company) to get the whole integrated system at once. They need partial results that solve their most urgent needs for integration. These most urgent needs will influence the decision what to do first and how to partition the integration into smaller tasks. However, a disciplined development path is still needed to effectively solve even the partial needs. Ad hoc solutions, although faster at first sight, will not fulfill the long-term integration needs.

Integration Process Activities and Phases

Integration requires that we analyze existing applications, design the integration architecture, select the integration infrastructure, design the solution, implement the integration, etc. All these are activities of an integration process and they have to be performed step by step. Here, we will classify these activities more strictly and define the sequence in which they should be performed. In general, we can partition the activities into technical and supporting activities. Technical activities include:

- Requirements gathering
- Analysis of existing applications

- Selection of integration infrastructure
- Problem domain analysis
- Design
- Implementation
- Testing
- Deployment

In addition to technical activities, we will also need support activities. The important support activates are:

- Project management
- Configuration and change management
- Communication with the environment

Integration is usually achieved in four phases which are:

- Data-level integration
- Application integration
- Business process integration
- Presentation integration

Data-level integration focuses on moving data between applications with the objective of sharing the same data among these different applications. Application integration focuses on sharing application functionality—business logic; and not only pure data as in data-level integration. Business process integration focuses on end-to-end support for business processes, both private (within organizations), public (between two or more organizations). Presentation integration focuses on a common user interface (typically a portal) for the integrated information system.

The four integration phases are usually done step by step, although not necessarily. Sometimes, we can skip a phase. The question is how to connect the activities and the phases of an integration process. The answer is that we will perform the activities for each integration phase. The technical activities that will not differ from phase to phase are the first three activities: requirements gathering, analysis of existing applications, and selection of integration infrastructure. We will have to perform these activities before a distinction between the four integration phases will be made. We will also perform all support activities equally for all phases.

The rest of the technical activities will be tied to the integration phase. These activities will differ in data-level integration, in application integration, business process integration, and presentation integration.

Integration Patterns

At first glance, concrete integration problems and the corresponding solutions are seldom identical. However, after working on several integration projects, we can see that problems and the solutions can be classified into common categories—integration patterns. Integration patterns will help us to understand the different solutions and methods for integration and allow us to choose the best solution for our problem. They allow us to look at integration problems from a certain abstraction level.

Integration patterns describe proven methods and processes used for integration within the enterprise or beyond. They emerge from classifications of common integration solutions. Each integration pattern defines a common integration problem and a sound solution. The most important integration patterns include:

- Integration broker pattern (also known as integration messenger)
- Wrapper pattern (integration adapter, integration connector)
- Integration mediator pattern
- Single-step application integration
- Multi-step application integration
- Virtual service (integration façade)
- Data access object (DAO) (Data exchange pattern) pattern
- Data mapping pattern
- Direct data mapping
- Multi-step data mapping
- Process automator pattern

For more information on integration patterns, please look at the various pattern catalogs and books, such as *J2EE Design Patterns Applied*, from Wrox Press.

Summary

In this chapter, we have overviewed the integration challenges and figured out that integration is one of the most difficult problems in application development. Therefore, it has to be planned carefully, it has to be based on sound integration architectures, on selected infrastructure and technologies, and managed according to integration process best practices.

We have seen that integration is not a solely technical problem, although technologies play an important role. Therefore, we have overviewed integration technologies and middleware products, ranging from data access technologies over remote procedure calls, message-oriented middleware to web services and enterprise services buses. We have also identified the best strategies for integration and discussed top-down, bottom-up, and inside-out approaches.

We have overviewed different types of integration, such as data-level integration, application integration, business process integration, presentation integration, and also B2B integrations. We have seen that the ultimate goal of each company is to have an information system, which not only provides end-to-end support for business processes, but is also easy adaptable to reflect the changes in business processes quickly and efficiently.

Achieving such integrated information systems is not an easy task. Today, we believe that Service-Oriented Architectures (SOA) and the related technologies, processes, and patterns provide ways to develop such information systems. In the next chapters, we will become familiar with various aspects of SOA, BPEL, web services, and XML.

2

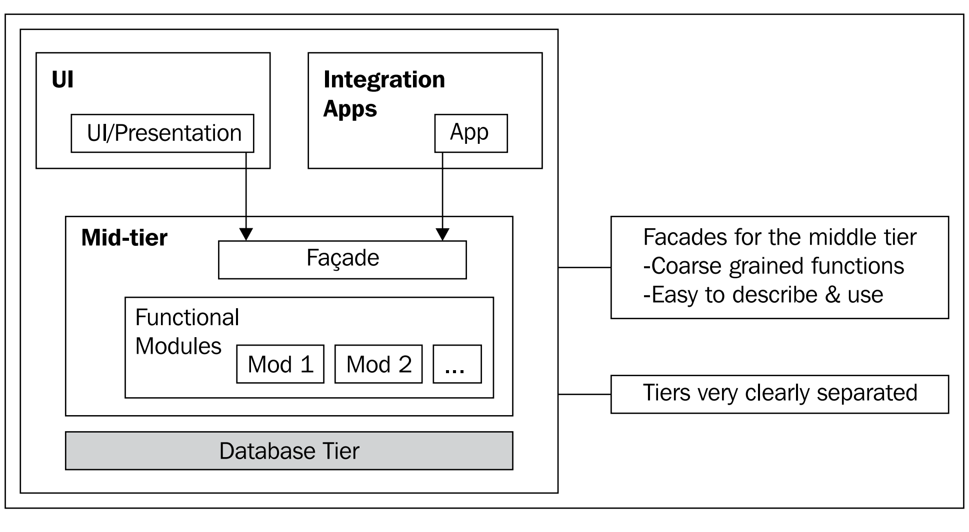
Service- and Process-Oriented Architectures for Integration

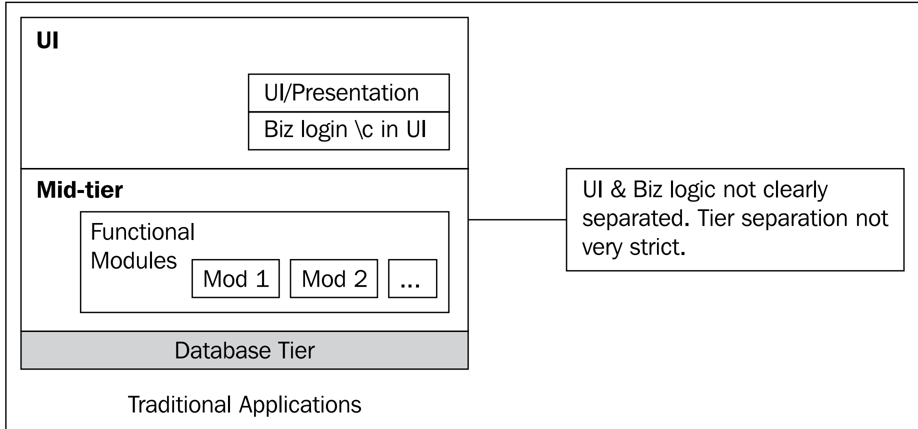
This chapter introduces the Service-Oriented Architecture, starting from how service orientation evolved from traditional software architectures. We will study the origins of SOA, the anatomy of services, orchestrating services for Enterprise Integration, the infrastructure needed for SOA, and how all these lead to Process-Oriented Architectures.

Traditionally, the design of applications has been targeted at direct access by users via the application UI. The approach to designing such systems centers around the user flows and expected user functionality. To provide this end-use functionality, the functional processing layer would use additional business modules or components and the data tier as needed. Such applications typically solve a local departmental problem. Given that departments in a company function together to serve the organizational purpose and goals, it is highly likely that there will be operations that span access across multiple applications.

As automation increases, applications will need programmatic access to functionality provided by other applications. For example, an order processing system may now need to be accessed from a home-grown CRM application to 'get_order_status-given-customer_name'. If the application was designed well, there will be a simple coarse-grained method available that will provide this functionality. In effect, the application would have been 'designed' for being accessed for services required by other applications—for Enterprise Integration. This notion of services is at the core of "Service-Oriented Architecture", or SOA as it is often referred to.

In the still emerging SOA architectures, a fundamental assumption is that applications can be accessed as much by other "applications" as directly by the end users. The following figures highlight this difference between Traditional and the Emerging services-centric architectures.





Traditional Applications

Now, is "service" really a new concept? We have had RPC, CORBA, RMI, and such for a very long time. More recently, we have had EJB and Web Services. We could always "invoke" functions or procedures running on a remote machine. The functions are remote procedures, providing some functionality running remotely. Then what is this hype about Service-Oriented Architecture (SOA)? Is it a new technology? New Methodology? Or, is it a new design paradigm?

In the following sections of this chapter, we will discuss these questions, starting from how service orientation evolves from traditional software architectures.

Defining Service-Oriented Architectures

So what is SOA? In short, it is an Enterprise Architecture where applications are designed to provide coarse-grained services, which are consumed by Business Processes or other integration applications. Service-Oriented Architecture is both a design concept and an architecture. The design concept in SOA is about designing applications/systems that have well defined self-describing access interfaces, with the services being composed into business processes. The architecture is about having simple mechanisms to use these access-interfaces for Integration of the Enterprise.

Extensive hype aside, SOA is primarily a design paradigm. The utility of SOA is amplified by the architecture embracing the ease of systems being able to talk to each other with the advent of ultra-low-impedance access mechanisms such as 'XML over HTTP through intranet/Internet'. This enables loosely coupled services. The architecture will also provide a stable, scalable, enterprise-grade infrastructure for hosting and accessing services with the required level of service quality.

This SOA design paradigm implies that along with defining UI screens and flows, the application architects must also design well-defined non-UI business services. The, functional requirements of this application for access from other applications can be easily provided for using a simple service method provided by the application. This again is not rocket science. Those that know distributed multi-tier architectures, and the more recent good practices for designing JAVA EE (the erstwhile J2EE, now called Java Enterprise Edition, or JAVA EE) applications with EJB (Enterprise Java Beans), will know that the UI/presentation and business-processing layers have to be separate with the business processing layers having well defined interfaces that provide functionality expected from the UI layers. This in itself is the first requirement for SOA; and beyond this UI vs. business separation, SOA requires well defined coarse-grained functions for access by any consumer of this function—be it the UI layers or other applications. Coarse grained here refers to well defined business functions, where to get a service one has to access just one method.

Application Tiers vs. Application Cells

Some history: Informix (my Alma Mater) was one of the pioneers in the space of embedding JVM into the DBMS. When we were considering adding J2EE Server into the stack, one of the end-solution architectures proposed was a fully self-contained processing environment that included the J2EE Server, running on an embedded JVM, with native high-performance JDBC (that seamlessly meshes in with the DBMS's C modules) running inside the DBMS process.

A very powerful model, wherein the applications can now be partitioned as "cells"—horizontally and vertically with each cell being self contained with all business logic (written as standard portable J2EE components) running inside the DBMS, which also provides the app's data storage for that cell. Once componentized, the component can be accessed from any client. The 'clients' could be other business components or thin Java/C clients—that could "orchestrate" a business logic using the various business components running in any of the "cells" of the partitioned app.

But (as often) this was way ahead of its times. While the technology s/w framework could easily be built, the utility would come only if application design considered the "vertical and horizontal" partitioning of the application. And from our experience up until that point, even using distributed databases was not so common—leave alone distributed processing components! This distributed app design, even today, in spite of all the SOA hype, is easier said than done.

Now coming to SOA, the basic concept is similar—have coarse-grained business processing units/services that abstract the processing and data access completely with a simple access abstraction (like web-services or... even as EJB session beans—Yes!!). Buts SOA does much more than just this!

In a simple two-application case, this inter-application access is probably trivial. It is less trivial where there are very independent applications/solutions that need to be accessed for realizing a business function (like, say, an SAP-ERP and a Peoplesoft-HR and a proprietary J2EE-based warehouse application needing to talk to each other or, say, generating enterprise management reports that span these systems). Enterprises have always had these integration requirements and have had some technology solutions in the past—be it EDI or HTTP-based AS2 or more recently Enterprise Application Integration (EAI) platforms. These provide a mechanism for systems and applications to interact with each other. EDI had a data-flow-based model, and EAI had an adapter-based access model. In all these cases, there was functionality being accessed (as in the case of EAI) or triggered as in the EDI data flows. However, these were not seen as explicit functionality being accessed or invoked. The end functionality was not "seen" as "services".

SOA brings the notion of services to the fore, as a key consideration in application architectures. Services are independent entities in enterprise infrastructure, which can be made available anywhere in the enterprise and are easily accessible from anywhere in the enterprise. It is much easier than EDI as a programming model and friendlier in terms of enabling integration of disparate technologies (.NET and J2EE). And it also provides a strong conceptual basis in its service abstractions and service orientation.

Service Orientation is the central design principle in SOA, where applications have to be designed up-front for services, with well defined coarse-grained interfaces. Unlike the popular perception that "Service Orientation" is a neat thing and the next band-wagon to jump right on, this core concept is not exactly new. When defining any multi-tier application, where the presentation and business logic are well separated, there would be well defined entry points into the processing "mid" tier. Be they façade EJB Session beans or DCOM components or such entry points even in proprietary apps (like even in the old stored-procedure days!). SOA today is more about reinforcing such entry points and identifying a simple mechanism to use these façades in, say, a business process.

The challenge here, though, is in identifying these well-defined middle-tier processing "components". Once this is done, SOA is a breeze! With a simple set of application attributes:

- The application has a well defined and separated functionality layer.
- The functional layer has well defined functional entry points (façades).
- The entry points are coarse-grained functions.
    - What is coarse grained? Coarse-grained functions are fairly atomic functions where a single function by itself, without requiring much done prior to it or post it, can realize well defined business functionality. Like, say, getOrderDetails (given an order number). In a well designed SOA environment, this could be achieved without requiring additional calls; say making a call to get customerID first, given customerID getting order name, from order name getting ordered, from ordered getting order lines, and from order line get line details. Instead, there should be just one call that returns the complete order.

- Well defined data interchanges:
    - XML-based descriptions of information flow are the best for SOA, even if not mandatory.
    - This would be in line with the SOA use cases in which heterogeneous applications in diverse environments (both hardware and software) will need to interact with each other. XML ensures minimal impedance or surprises when data is sent or received.
- Powerful and flexible Business Process modeling environment:
    - Most Integration requirements across applications and information systems can be modeled as simple flowcharts. A business process programming environment that simulates flowcharts is ideal. Now, SOA being an enterprise level integration environment, the class of problems it attempts to solve is anything but simple. The interactions are often modeled using complex flow systems such as UML Activity diagrams.

Why SOA in the Integration Space?

To understand why SOA is needed, we should look into how IT systems evolve in most organizations and how the need arises for these systems to interact with each other.

Islands in the Enterprise IT Landscape

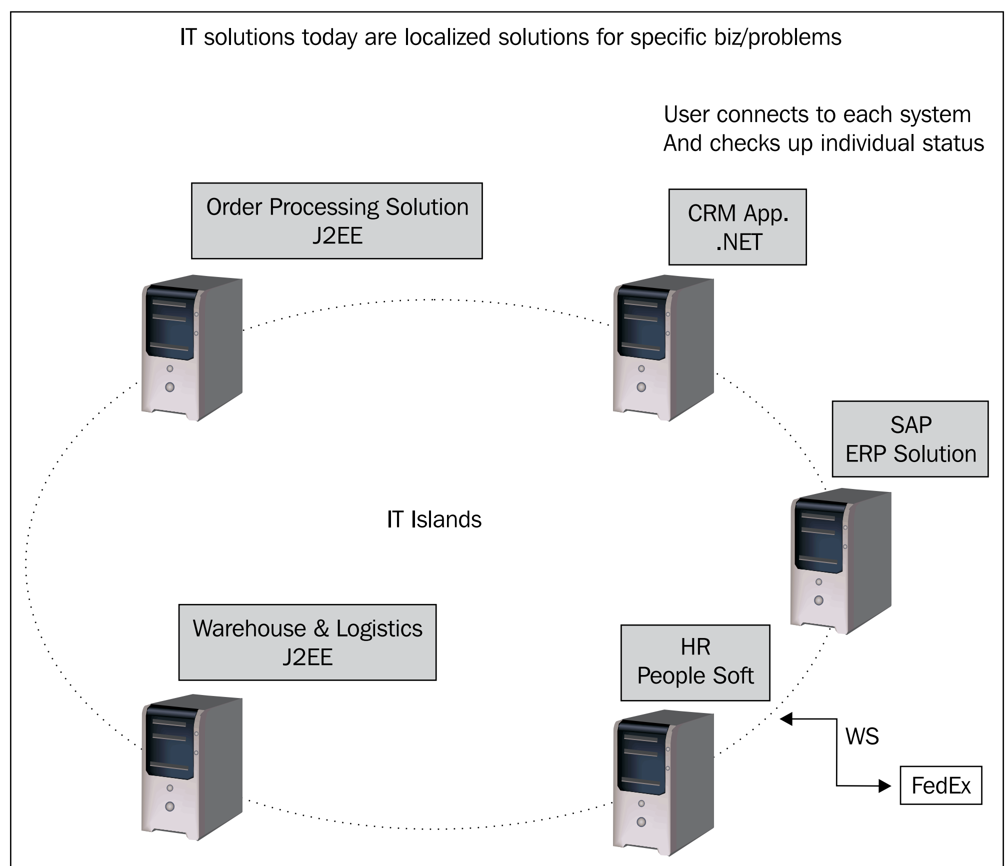
IT adoption is largely driven by immediate business needs—to help improve the efficiencies or the reliability of existing business processes. It always automates existing business functions—that are probably in place using manual processes to begin with.

Typically, when IT is first considered to replace non-IT manual processes, the scope of the work and the purpose is largely driven by individual departments. The departmental problems are the foremost and most critical problems that need to be addressed.

While it is conceivable that an organization decides to automate all functions of the enterprise in one single stroke—this is not practical for several reasons:

- Each department is responsible for its business processing.
- Each department is closest to the problems faced—say an order processing department will be aware of the errors faced in manual order registers.
- Each department has its IT budget.

So the Manufacturing group will go for manufacturing planning and scheduling solutions. The Warehousing department will go for an IT solution to automate the warehousing and shipping operations. Human Resources will go for a HR application, and so on.

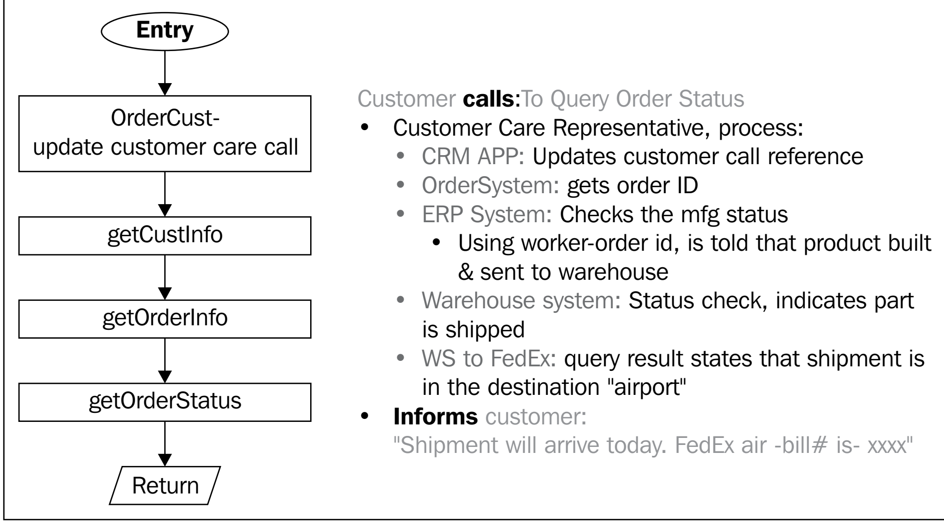


Once all departmental problems are solved with localized IT solutions, the state of any enterprise will be that there are islands of IT solutions within the organization—each solving one business function very well (see the previous figure showing the islands of IT infrastructure).

The Integration Problem

While IT solutions maybe in islands, the business, however, doesn't exist in islands. The departments within an organization serve a common organizational purpose—say to manufacture and sell the products/services that the company is in business for. Given this, there will be quite a number of interactions between the various departments in fulfilling each department's functions. In that process, there will be business operations that will span the use of multiple systems.

Let's look at a scenario: say, in an enterprise that builds to order, customers calling in with queries on the status of the orders are quite common. For a customer-care representative to service such queries, he/she may need to access Customer records, Order details, Manufacturing status, Warehouse data, and shipping details.



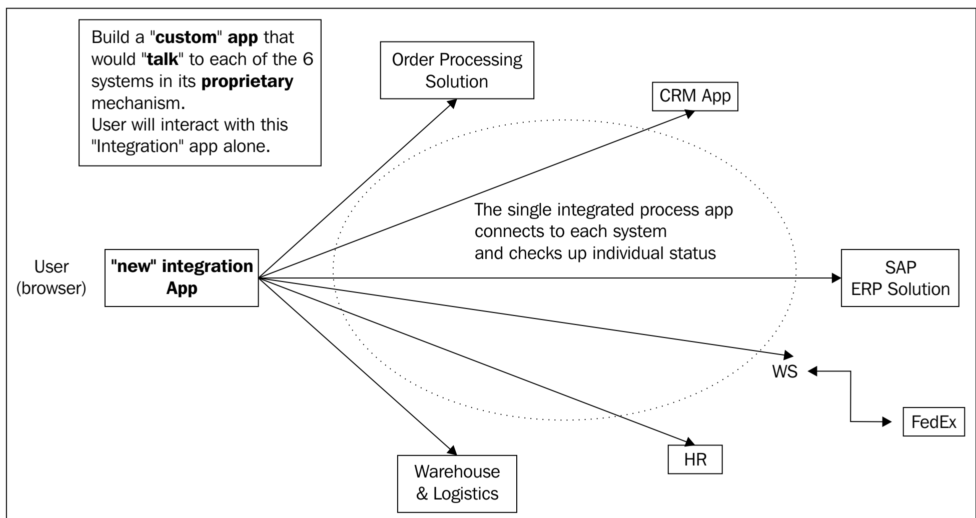
Assuming that all the departments have their IT solutions in place, there is likely to be a CRM Application, an Order Processing system, Manufacturing solution (SAP/Baan), Warehousing and Logistics solution, and a Shipping company like Fed-Ex or DHL. The customer-care representative will need to access all these systems, and may be trained in all the applications, and will have all applications opened on his/her desktop, and switch between the applications as needed per the business process above.

While there is a solution to serve the business need here, it is quite time consuming. And to top it up, it is error prone as well. If, say, when copying the customer number or order number from one application's window to the other, if there is an error an incorrect order will end up getting used. And any response made to the customer based on this incorrect info will be incorrect. Even when there are no errors, just the process of accessing multiple systems and switching between applications and performing the actions sequentially on the various applications would be time consuming. In these days of hard and deep competition, poor or inefficient service would mean a lost customer! Organizations often push the limits on efficiency and accuracy.

So, it is only a question of time before there is a strong business need, driven by either quality of service (accuracy) or by the cost of operations (time taken to serve a customer), to automate this business process—so that the customer-care representative enters minimal information needed and there is an automated solution that accesses the required systems and fetches the needed order status info.

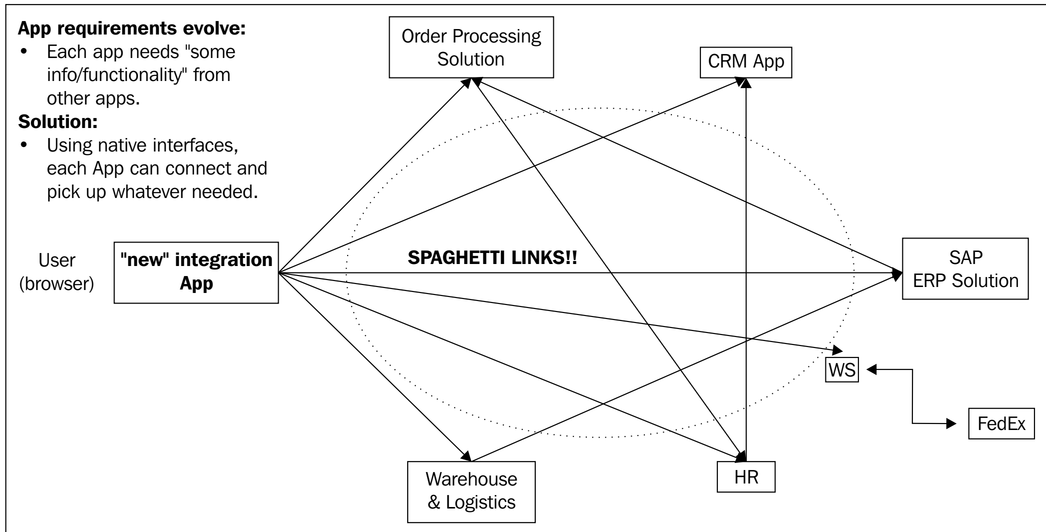
Custom Integration Application and Its Issues

The simplest solution to address the customer-care integration problem would be to build a custom Integration Application that is written to access the various systems using whatever access API/mechanism the systems provide. Several mechanisms are available to programmatically access various application platforms and environments. The integration application would need to use the APIs native to each such back-end system and provide the programmatic access.



Once the organization and the business end users see the possibility of authomating access to multiple systems, soon there will be other such business cases seen and demands made to provide solutions that require accessing multiple systems. For the first few, the custom integration application for each case is a reasonable approach. But as the number of such applications increase, this would be a fairly complex situation to manage, especially as the access APIs to each of the application are native and custom for each application. The IT group that manages the integration application will need to be aware of multiple different application systems and their access APIs and mechanisms—ranging from JAVA EE and .NET to existing enterprise application systems such as SAP/Baan/Mainframe. Now, these are not exactly similar technologies, and often times are not even available on similar platforms, and integrating across these is never an easy task. Any group managing a custom integration application will have a very formidable task on hand in building the required expertise across all needed technologies and solutions!

Now extend the problem to an extreme, where every system in the organization needs to access one or more other systems. Then it gets really nightmarish as shown in the following figure of an extended integration problem.



To alleviate the problem, there could be a generic integration API/approach like the Java Connector Architecture. This surely reduces the programming complexity—so long as all the solutions have the adapters/client layers required to access them from the common chosen technology like JCA. But this does not reduce the redundancies in the organization. Multiple teams will need to understand the JCA adapters of each system. There will be multitude of applications in the enterprise that are just integration applications—each doing similar access to various back-end applications: not the best of situations for an optimal utilization of the IT resources.

Inverted View: Reusable Services, Simple Integration Processes

This is when one needs to invert the view. Instead of every integration application being so intrusively aware of every other solution/application in the organization, we could look at a solution where there is a more common approach—where all the integration applications look at a very generic approach, where any service from any application is accessed in a common way. And each application exposes its "services" to this common framework.

Programmatically, this is not very different from EAI platforms and standard connectors such as JCA in the Java world. The difference though is in the underlying mechanics!

In the inverted view, each IT group will ensure they expose the needed services to the common framework once. And any solution/process needing these services can access them by a simple mechanism provided by the framework. So the first-level API access, be it adapters, custom APIs or JCA type mechanisms, is dealt with once, and the business services are exposed to the common framework. Now this common framework could be a low-impedance mechanism like web services, which are easily understood and have simple mechanisms available to access them.

Enter SOA: A Services-Based Integration Architecture

This inverted view is at the core of service-oriented architecture—where each application "wires" the needed business services into a common "fabric". The business services will be coarse grained with well defined interfaces. All services will be available in the same common fabric, in a common manner—independent of where the service is running or what back-end system it is connecting to.

Any application or solution needing "any" service in the organization can access it through this fabric. Even better, as all services look the same after being "wired" and can be accessed by exactly the same access mechanism, this access mechanism can be abstracted into simple service agnostic utilities. This enables defining the integration applications as simple processes. Programmatically, they may resemble simple flow-charts, with each block being a service invocation, even though the complexity in such processes is better represented in more evolved process models such as UML Activity diagrams. These processes are commonly referred to as Business Processes.

Concepts and Principles of SOA

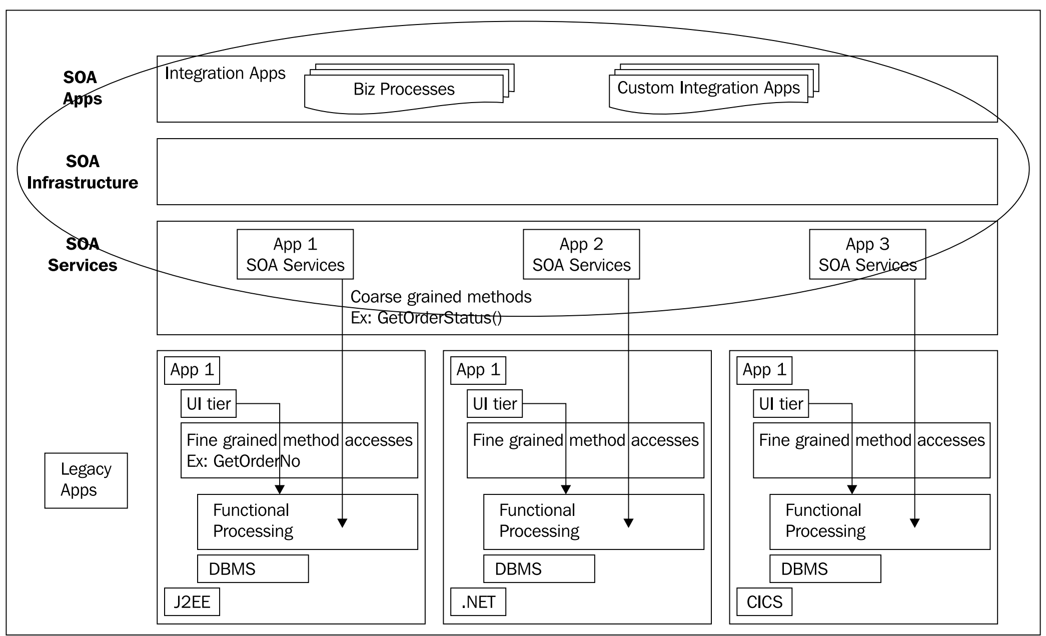
The principles of SOA essentially involve orienting applications in the enterprise towards providing services, and consuming these services in simple flowchart-like business processes that are modeled very closely on the enterprise business processing.

Paradigm Shift—from Self-Contained Applications towards "Services"

A paradigm shift in Enterprise Application architectures is evident now from self-contained applications—solution islands—towards an application services layer. The value seen from enterprise integration combined with lowered impedance in connecting various applications is accelerating this shift. The need to incorporate the access to such services in a generic manner in the organization can be effected by a layered infrastructure to enable SOA.

For any given solution, one can look at 2-tier, 3-tier, or n-tier architectures. These will still continue for the individual applications being built. For these, and definitely for the existing legacy solutions, there will be a well defined Services layer in the emerging IT architectures. By IT architectures, I mean the whole IT infrastructure that spans multiple applications on different technologies allowing them to co-exist and still effectively interoperate. Enabling such an infrastructure is becoming a major priority for several IT organizations and CIOs.

This introduces two new layers in the IT architectures. One for the Services and one above the services in terms of the business-processes that use these services.



In short, 2-tier systems separated data storage from the application. UI and business processing coexisted in a single tier. 3-tier architecture separated UI, processing and the DB. And now in SOA, as shown in the previous figure of SOA application layers, the paradigm shifts from just n-tier to granular services—accessible from a business processes/integration applications layer—a complete new "layer" of applications—beyond just the UI accessing the applications.

Service Orientation

Service Orientation is about having coarse-grained interfaces to business functionality. The notion of coarse-grained interfaces is itself not exactly new. Any well designed application has well structured business modules and each module in such applications will have well defined functional methods/operations. In more recent times, in application platforms such as JAVA EE, CORBA, and .NET, the notion of the 'Façade' design pattern has been quite popular. For example, because in JAVA EE applications, typically the business functionality written as EJB session beans is accessed from multiple front-end layers—like web access modules (JSP/Servlets), custom UI, mobile applications (say through J2ME), and such. To ensure minimal duplication of programming work and avert any redundancies of business logic, such logic is encapsulated into one 'façade' and this is used from any of the front-end tiers. In the same spirit of abstraction, the granularity of this 'façade' is

expected to be very coarse-grained. One need not invoke several methods to perform any business function. Most functions will need just one method. The better the functional analysis and design, the coarser will be the granularity.

Component-Based Services

As a defining concept, Component-based Service development is at the crux of SOA.

Object-Oriented and Distributed components are well established concepts. Object Orientation is a good design approach for designing extensible systems with the complexities of various modules and parts of the system well abstracted and "hidden". And distributed components are an extension of the simple RPC mechanism, wherein it is not just a remote method that is invoked (as in RPC), but a remote "object" that is accessed. An object with its data and processing encapsulated into a single entity—as in conventional "objects" in OOP. Good popular examples of distributed components are EJB and DCOM.

Component-based services are not too different from distributed components. In fact, distributed components very well qualify as "Component-based services"—especially when the distributed components are well described and easy to use. Going against the spirit of extremely simple and flexible communication though, EJB and DCOM have a tightly higher coupling to the wire protocols and are a little restrictive in terms of what domain the client applications could be running on and in. For example an EJB application will necessarily need an EJB client application. Most JAVA EE vendors require their specific client software (JARs) available on the client side. Though there is a more portable IIOP standard that could reduce the vendor dependency, it is still not as open as, say, XML over HTTP!

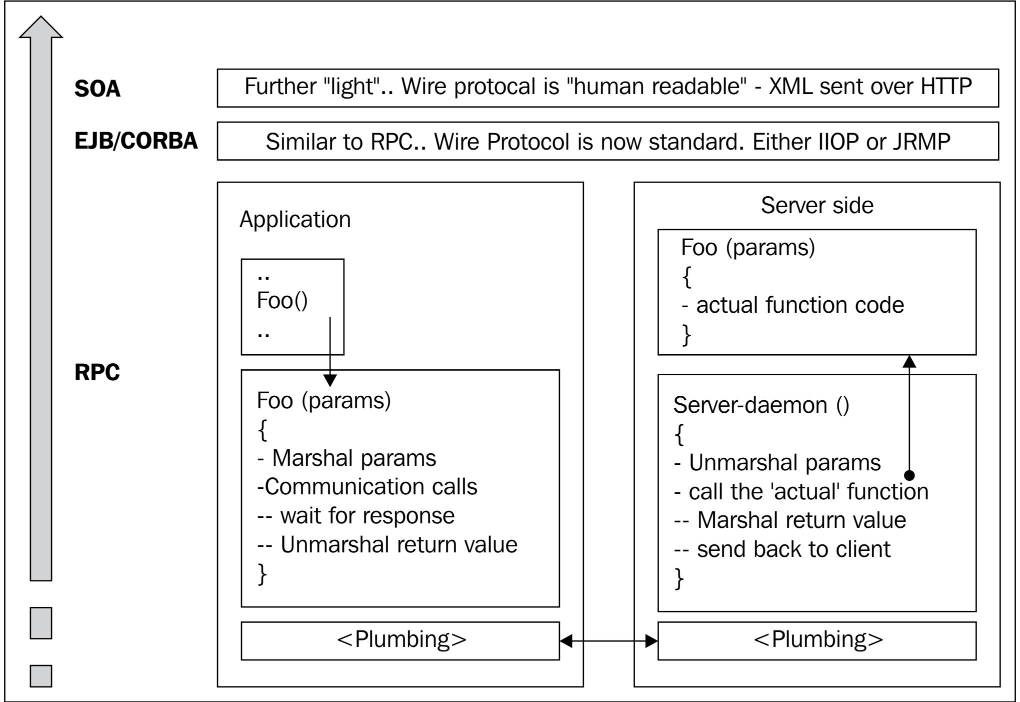
In short, though EJB and DCOM qualify very well as Component-based Services, other approaches that reduce the communication and client-domain dependencies would form an even better platform.

As a development model and system design approach, Component-based Service development would require that the 'façade' functionality is well defined for each of the systems/applications. And these are implemented at the top of the solution. Once such façades exist in the solution, exposing them as "distributed components' in any technology would be much simpler—essentially boiling down to writing a stitch layer that ties the "component" into the specific technology/fabric used as the SOA infrastructure.

The Internet Simplifies Remote Services

A framework for service did exist even in the days of simple RPC mechanisms—like DCE or Tuxedo. It was just that these had a very tight binding of the client and server components, and, were programmatically very tightly coupled. It was not easy to have one without very deep and intrusive knowledge of the other. With the advent of CORBA and later EJB, this binding was loosened a bit. I could have an EJB client application and EJB server components developed separately and even deployed on different vendor's platforms. The interoperability standards were so well defined that I could develop the client and server modules on say Weblogic and deploy them on Websphere or Pramati. Even further, I could have the server component deployed in a different server in production and the client component that was built and tested on Weblogic would always work out of the box.

This was a step ahead. But the coupling was still not loose enough. There was a binding into Java EE/CORBA technologies here (see the following figure of middleware systems). So we cannot have an arbitrary set of technologies that can coexist easily. While RPCs did support coarse-grained services, they, however, were not adequate for enterprise-wide integration in heterogeneous application environments.



The Internet brought in significant change in IT infrastructure in several areas. The Internet accelerated the push of SOA in its current incarnation by making it easier to access remote services through the omnipresent Internet and HTTP. XML, which is a generalized form of HTML, slowly moved in as a very widely accepted format for data interchange.

This power of Internet and XML was leveraged and a simple mechanism to describe, invoke, and execute RPC came out in the form of Web Services. A simple XML document describes the service/business function. The request is composed as an XML document (per the SOAP specification), and sent over HTTP to a web address. There is a servlet or SOAP processor on the other end that un-marshals the request in the XML document and executes the required service running on the other end. So the service developer needs to worry about the WS façade and the required access through whatever the APIs/mechanism available to access the system, just once. Once the service is available as a web service, then any application/solution needing access to this application can access it as a simple web service.

Consequently, the access to any solution becomes simple and independent of the technologies and infrastructure that are in use at the target application solution. Then there is a logical corollary that follows from this independence: that the services should be simple and easy to understand in the organization—without needing deep understanding of the business processes in that department. This should be possible without needing multiple steps to perform any operation. In essence, the services exposed in this low-impedance manner need to be simple and very coarse grained.

This is 'Service Orientation'. Each application exposes its services to a common mechanism for integration with other applications or use in business processes. Service orientation is about being aware that no application will exist in isolation, and will have other systems and solutions that will need to access its services, and ensuring that these can be accessed as easily as possible—both in terms of granularity of the service that simplifies the description and in terms of the technology required to access these services.

Each application solution development team works on analyzing the business needs and usages, identifies a set of simple coarse-grained services that the rest of the organization might need to access, and implements the same in some simple, preferably organization-wide, standard framework/mechanism for other systems to access. Once services are available, anyone needing these services will access all of them in the same manner, regardless of the specific technologies at play in each of the systems.

Consuming Services

Once services are available in the SOA platform, what would one do with them? Two key usage scenarios exist:

1. Consume the services from a client/integration application.

By consume, I mean use the service—invoke the service in any other application/program as part of its normal programming logic. Say in an ERP application, whenever Customer details are needed, one could make a call to a CRM Service that returns the customer details given customer-ID. This could be done by invoking the service as part of the program code that performs normal ERP processing.

2. Compose the services into Business Processes.

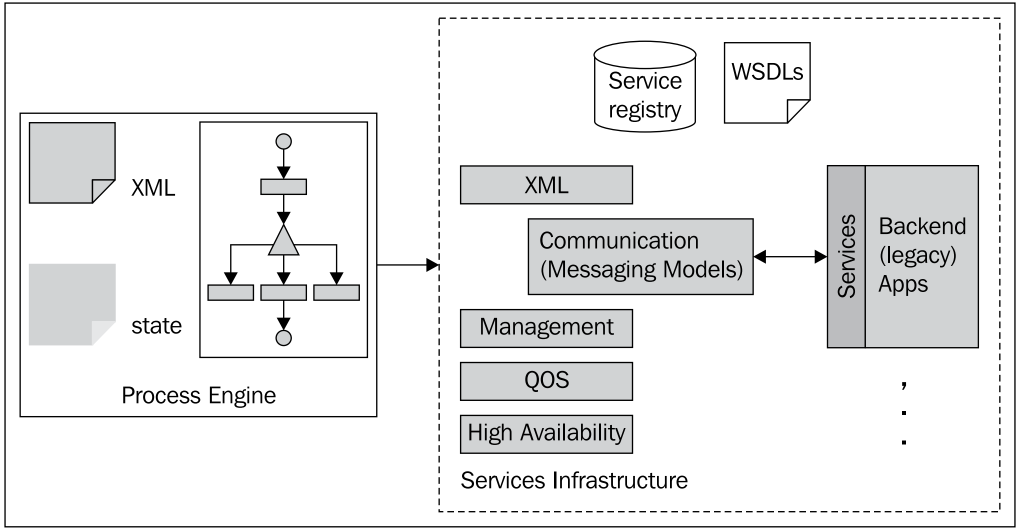
The other more popular approach to using services is in Business Processes, by "orchestrating" the services. Orchestrating essentially involves "stringing together" the various services available in the domain to perform a business function or process. This is a relatively higher-level programming approach, where the orchestration language, with its very simple programming flow constructs, is very amenable to graphically defining the process, much like drawing a flow chart or an UML Activity diagram.

BPEL is a good example of a rapidly emerging standard Orchestration language.

Introducing SOA Architecture

SOA has two key parts—the "service" and the "architecture. The service is essentially the outward view of applications within the IT organization, where each application provides the required "business services" for access from other applications. The "architecture" is the organization-wide approach to "using" the services. As there are multiple applications/solutions within the organization, there has to be a decision to provide an application/solution space that spans multiple applications.

To do this using services exposed by each application, and by standardizing on an infrastructure to provide and consume these services, and the services consumed via a process-based framework, would all be comprised by the SOA "Architecture", as shown in the following figure of SOA architecture constituents.



Service Abstractions

A service essentially consists of the actual implementation, its interface, and the invocation mechanisms. The service implementation is by and large determined by the required functionality from the service. This would have very minimal dependencies on the platform or the service's operating environment. The dependency, though, arises in its interface layers. The service's infrastructure must have the ability to "invoke" the service implementation code. This will require some handling specific to the service's infrastructure. This could be configuration-based handling, or in some cases may require actual code-level handling, like implementing some well defined interfaces.

The service then needs to be "described". This is the interface of the service. This is best done in XML, to ensure a loose coupling of the service implementation and client layers. WSDL is a popular mechanism to describe the service interface. The interface defines the logical model of the service—the service name, operations in the service, and its inputs, outputs, and faults. This by itself doesn't describe anything about the actual invocation mechanics. This is done in the service bindings, also typically included in the service description file/mechanism. WSDL supports describing the bindings.

The service bindings include the details on what protocol is to be used, and exactly how the inputs are to be sent and outputs expected.

The loose coupling of the service and its consumers is important in an SOA platform. The service invocation must be completely independent of, and agnostic of the service implementation. The only dependency is the service description (say WSDL) and agreed wire protocols (like SOAP).

Service Invocation and Service Implementation

Any SOA architecture must provide for a well defined mechanism to implement services and access services. These are complementary in the sense that the invocating and the invoked ends must both "speak" the same language. Web Services is a good example of this approach—where SOA over HTTP is the "language".

Process Engines

Process engines, or Process Servers, provide the execution environment for business processes. The business processes are the primary means of consuming the services available in an SOA platform. The business processes may be defined in an environment or language supported by the engine. Once defined and configured, the engine will manage the lifecycle of the process instances—from starting a new execution run instance through the management of state, all service accesses as defined by the process definition, managing the data flows and process state, and in more advanced cases managing long-running processes including persisting the process state if needed.

Messaging Abstractions

Communication in SOA systems involves sending the requests and responses between the service consumers and service providers. Regardless of the specific protocols used, there are data or information "packets" that are sent. A request message essentially involves specifying the details of the service to be invoked (the service and operation name) and providing the input data—typically one or more documents. Likewise, a response could arrive as another message, with the return status and data. Even a simple web service request going as SOAP (an XML document) over HTTP can be seen as a communication packet or message. The SOAP document is the message.

Messaging systems have well defined semantics. Depending on the specific communication infrastructure supported by the SOA platform, messaging platforms may or may not be used. In SOA environments, the notion of messaging is applicable regardless. The "message" may be sent via an HTTP connection, or via a Messaging Platform (Java Messaging Servers or Microsoft MQ or any other messaging platform).

Synchronous and Asynchronous Messages

Messages may be delivered either synchronously or asynchronously. In Sync mode, the sender waits till the receiving program has received the messaged, processed it, and responded to the sender. In Async mode, the sender sends the message, and continues its processing regardless of when the receiving program receives and processes the message. If the sending program needs to receive status or data from the other end, typically this is done at a later point by looking for another message containing the status and response.

As service invocation relies on messages being sent, the services can also adopt the same synchronous or asynchronous behavior. In cases where there is asynchronous service invocation—where a service is invoked, and at some later point in time the response from the service is received—there is no waiting for the response in a synchronous mode. There is no client program or application component that is just waiting on the response—which happens in typical synchronous invocation mechanisms. In Async invocation, the client could continue to do other processing, and when ready look for the response message. In Synchronous services, the client program or application just waits for the response—however long it takes.

Service Registries

SOA environments are loosely coupled with the service providers and consumers likely to be distributed across the enterprise. The services are described by their interfaces and the consuming end needs no more information than this, and probably the communication details like the IP, port, and protocol. In such a distributed services environment, there needs to be a mechanism to list all available services and provide the necessary metadata that describes the services and probably provide the standard descriptions such as WSDL where available. Such information on the available services is typically maintained in Service Registries.

These may be based on standards such as UDDI or may be specific to the SOA platform.

Quality of Service

In an enterprise-wide distributed services infrastructure, the availability, response times and reliability of services determine the quality of user experience. Services may have different levels of operational criticality, based on which the required quality expectations are set.

The QoS parameters must be specifiable by the systems administrators. The SOA infrastructure must ensure compliance to the defined service response times and the availability. To provide this quality of service, the service and process activity in the infrastructure should be tracked, logged, and reported. This business activity monitoring and management is at the core of QoS.

Communication Infrastructure

While SOAP and Web Services describe clearly what a service is and how it is to be described and how a request/response should be composed and sent over the wires, this does not, however, fully address the communication problem—it defines a simple mechanism for point-to-point access. While this is OK for the simple integration solutions that we saw earlier in this chapter, it is not OK for the organization-wide infrastructure that we are now discussing. While the communication as SOAP over HTTP is not adequate, the description capability provided by Web Services is very powerful. A good SOA infrastructure will retain these WS-based abstractions for describing a service and also for composing a request to be sent on the wires, but the actual communication protocol may be something different from HTTP. Say, it could be a messaging platform (like any of the messaging implementations such as TIBCO, MS-MQ, or Sonic). What this does is offer a very strong and reliable communication fabric that inherently is designed for a many-to-many access scenario, unlike the point-to-point nature of HTTP and simple sockets.

What is a "Bus"?

A "bus" in essence extends the notion of a service further to indicate that there is a services bus, wherein all services are available on the bus and any client application/ process is also on the bus consuming the available services. Taking this a step further this could also be a document flow scenario, where a business document flows through this bus and gets processed by various services along the way from entry to exit. The additional document- or message-centric processing that is built onto a services bus is the value addition that platforms such as Enterprise Services Bus offer.

XML and Web Services: SOA Foundation

As we discussed, SOA essentially involves services, service consumption, and the infrastructure to invoke services from the client, middleware infrastructure to "ship" the request to the server, and a service container to execute the services. The advantages of going with a standards-based infrastructure are significant for an IT organization, essentially, in terms of preserving the investments in the integration application and the services.

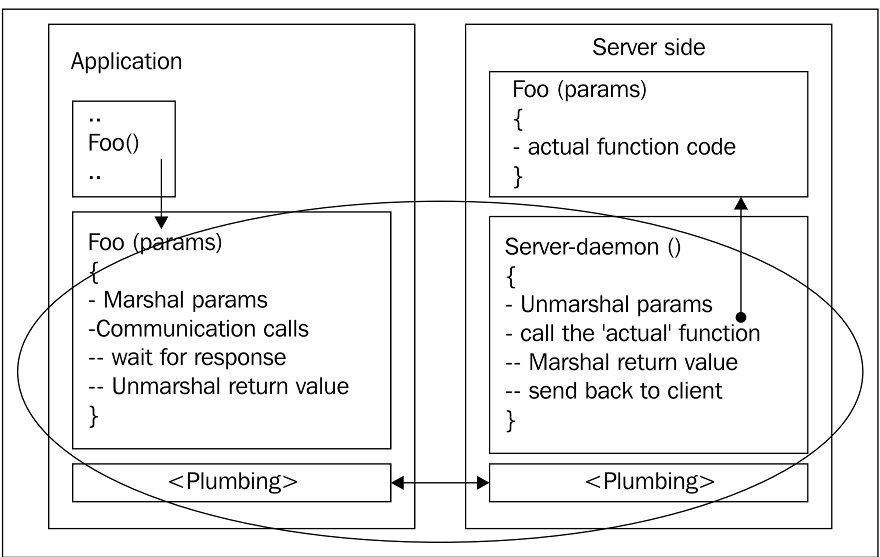
Using XML in Middleware

SOA is more a design approach and concept than a technology. To that extent, SOA could be implemented using any programming environment from a simple 3GL such as C or Java to established platforms such as JAVA EE and .NET, Or the more contemporary Web Services. However, in the spirit of the usage scenario—which is applications in heterogeneous environments needing to "talk" to each other, it is imperative that there be a simple connectivity without having much restriction on the technologies at play on the various applications themselves. Say the order processing system could be in JAVA EE, SAP could be the ERP system, and warehousing solutions could be home grown, written in C. These systems should still be able to interoperate as "services" in a chosen integration environment.

To facilitate such flexibility, the integration abstractions have to be simple and well supported on various application platforms (like JAVA EE, .NET, custom ERP, etc.). This requires standardized representation of a service (its definition and the contract) as well as simple mechanisms for communications over the wire. What better approach can there be than using a universally understoodd representation language like XML and a well accepted and widely available communication protocol like HTTP?

Middleware Mechanics for Services

A core function of middleware systems is to provide the needed mechanisms and handling for service invocation—send the request and fetch the response. This involves transporting an RPC/service request across the wires between two machines. Construct the request details, deconstruct and execute the required procedure/service, and return the results—again "constructing" the results to be deconstructed on the client end (as shown in the following figure of typical middleware).



The two ends of any request could be of very differing environments—hardware, OS, and application infrastructure. Given this, the wire protocol adopted by the middleware plumbing has to be as platform neutral as possible.

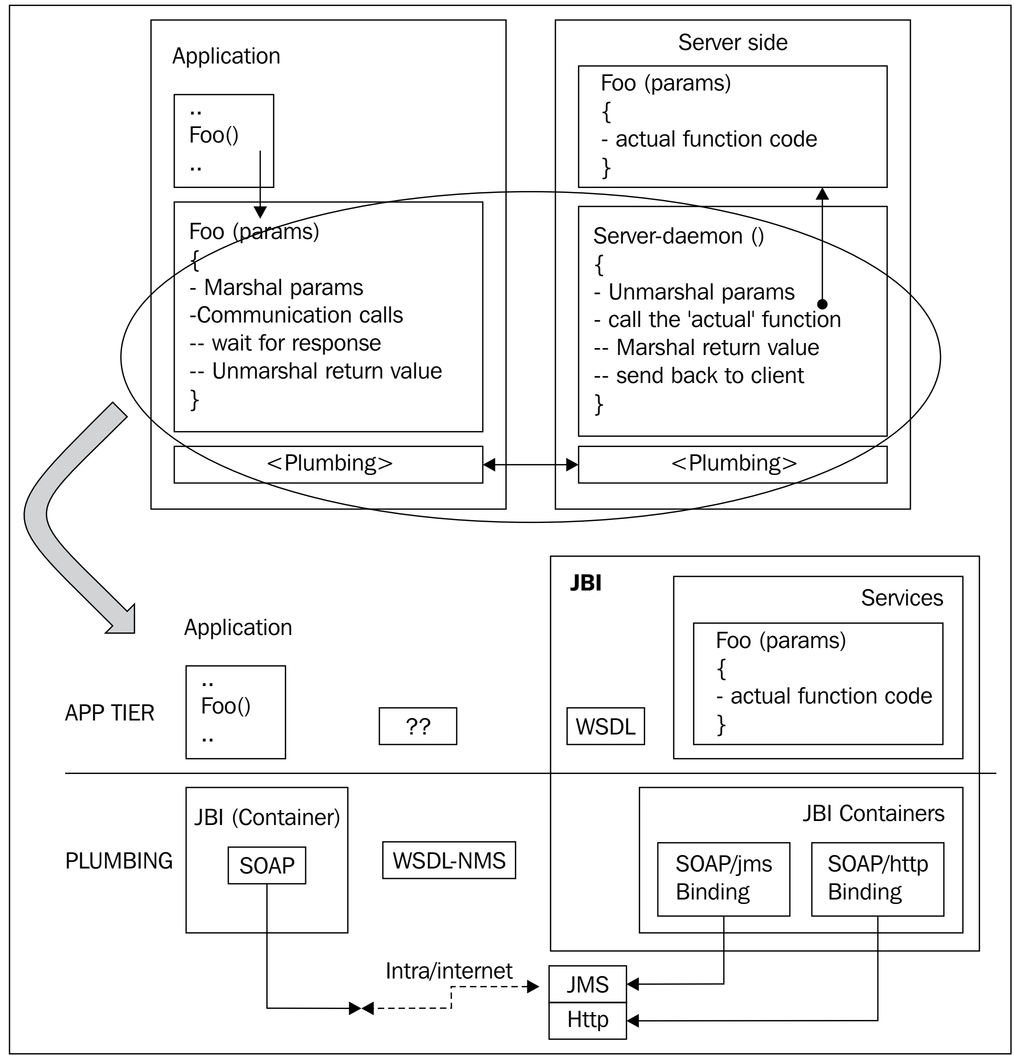
There have been several open standards-based technologies in the past including the IIOP that came out from the CORBA domain and the more recent JRMP adopted by JAVA EE for its EJB infrastructure. Incidentally, JAVA EE also moved over to IIOP to enable CORBA and EJBs to interoperate. Now all of these are well defined wire protocols, but by no means easy to work with when the specific middleware system is not in place—meaning, imagine making an EJB request by writing the "wire" utilities to compose, send and process JRMP- or IIOP-based requests. Here, XML offered a unique advantage. The "language" of XML is well understood and even human readable. It formed a perfect base on which to build the "wire" protocol for services plumbing in SOA.

XML-Based Mechanism to "Invoke" Services

In the late 1990s, XML emerged as a universal and ubiquitous data representation language for all kinds of data ranging from web content (HTML is a relaxed form of XML, though HTML predates XML), to systems and application metadata, to business data within the organization and for exchange across organizations—and much more. Given its widespread understanding and the utilities available to construct and process XML documents, it was just a question of time before it came into use in middleware space. The request could now be structured as an XML

document, sent over any transport—simple sockets, HTTP, FTP—to the end point where the service is available, there is a server application listening for these XML files, and once a file arrives it "un-marshals" the request details, executes the request/service, and returns the response, again, as an XML document. Essentially, this is a wire protocol for services built on XML.

Rapidly, as shown in the following figure of standards providing the middleware plumbing, a complete standard for making service requests/invocations using XML as the wire representation of the request/response emerged—called SOAP. And the whole domain of accessing "services" over the Web, essentially RPCs over the Web, came to be called generically Web Services.



Several standard specifications are continuing to emerge for middleware systems and SOA even as we write this book. Among them Web Services-related standards hold most promise for widespread adoption, due to their very broad based support—ranging from technology providers and systems integrators to end-user IT organizations.

Services over the Web via SOAP

Services over the Web were made popular by a few leading vendors that took the initiative to define an XML-based standard—SOAP—Simple Object Access Protocol. SOAP described the structure of an XML document that represents a service request and its response.

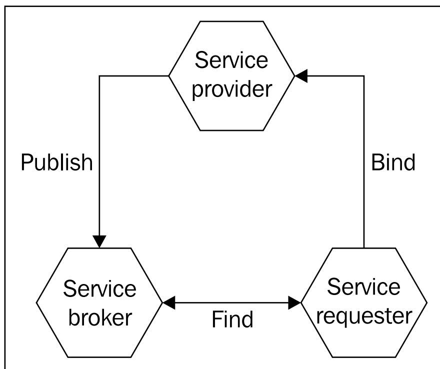
The XML document was intended to be delivered over the Web, just as web pages are delivered. The request is submitted as the POST body, and the response is returned similarly to how a normal HTML page is returned to a browser. This sheer simplicity in terms of communication infrastructure, needing just simple web (HTTP) connectivity between the requesting end and the service provider end, significantly reduced the remote procedure access complexity that hitherto had required complex environments such as CORBA.

Now, what are Web Services? Web Services are a simple mechanism to provide "services" over the "Web". SOAP is essentially an XML-based transport over HTTP, enabling services to be available for access over the Web—Web Services.

Web Services—Protocols for SOA

Starting from a simple standard that focussed on defining the communication protocols when requesting services over the Web, SOAP has now grown to cover a whole gamut of issues ranging from services description to complex data types, attachments, security, transactions, and more. Now Web Service standards define everything from describing the interface of a service (WSDL-2) to the representation of request parameters over the wire (WSDL2's Normalized Message Service) to security and reliability requirements.

Web services simplified applications talking to each other. This set the ground for a full-fledged integration infrastructure based on services. This has now further been extended to providing solutions beyond just integration.

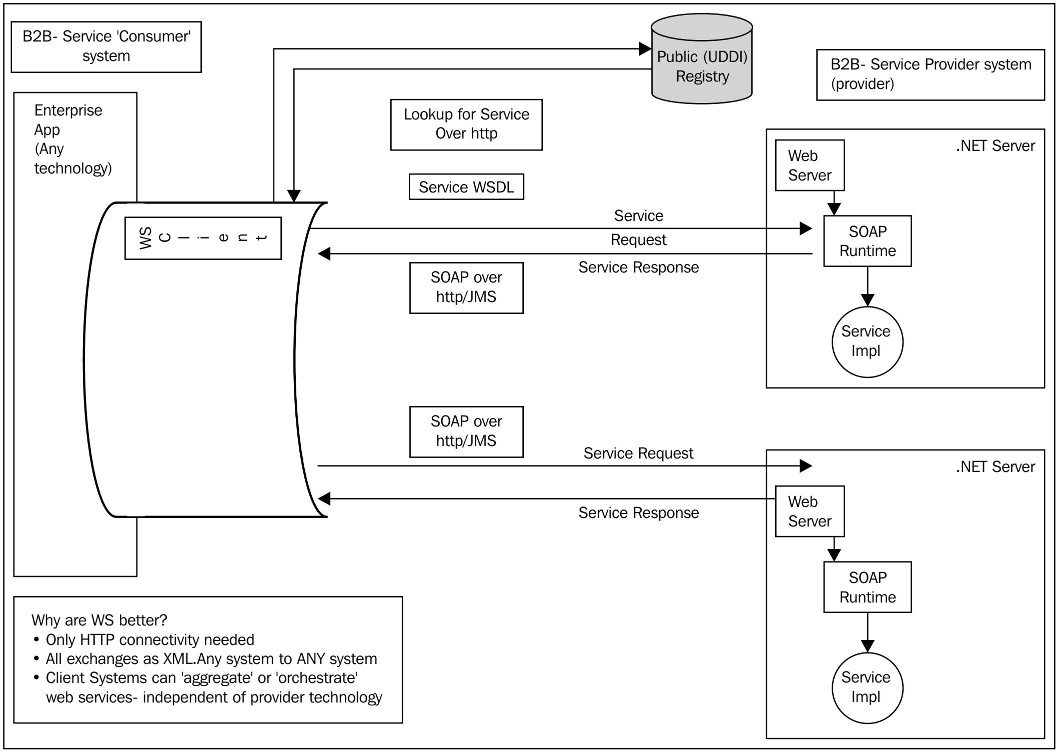


To support the required abstractions, the activities that need to happen in any service-oriented environment will be:

1. A web service needs to be defined, including its interfaces and invocation methods.
2. The web service needs to be published (see the previous figure).
3. It needs to be located to be invoked by potential users.
3. The service needs to be invoked to be of any benefit.
4. A simple mechanism to "un-publish" the service when it is no longer available is needed.

Technology Agnostic System-to-System Interaction

Web Services provide a simple mechanism for integrating disparate systems regardless of the domain or the technology the individual systems may be running on (see the following figure of web services for technology-agnostic integration).



Key attribute of Web Services:

- Promote interoperability:
    - Minimizes the requirements for shared understanding.
    - This is realized by relying on XML as the primary information/data representation format. The invocation essentially involves just "transferring" data. No programs or code such as stubs or class bytecodes are exchanged here. This is the most important aspect that results in the loose coupling of the services.

- Late binding of Services (just-in-time integration):
    - A service is identified just by its URL and a WSDL description of its contract/interface. Beyond that there is no assumption on exactly where the service is running, the actual "binding" can occur at the invocation time, transparent to the WS client, on the WS server end.
    - Further, dynamic service discovery and invocation (publish, find, bind) is also possible, to further limit the bindings, where even the location of the service (URL) is dynamic. Such environments will allow applications with looser coupling.
- Powerful Encapsulation:
    - A service is identified just by its WSDL—which is an interface that defines the name of the service, its operations, and the required input parameters and outputs/faults emitted by the service operation.
    - This completely "hides" the actual service implementation from the applications that use the service. The language in which the service is written, the application infrastructure on which it is running, the OS, the DB, and every other aspect of the actual runtime of the "service" is hidden from the application that uses it.
    - This provides for a very high level of complexity encapsulation (the service could be run in any environment—regardless of the client applications), flexibility (its operating environment can be transparently changed, at any time), and scalability (clustering and load balancing), and application extensibility (can modify the functionality of the service quite easily).
- Enables easy access to legacy applications:
    - All it takes is to "wrap" a legacy application as Web Services—Provide the definition of the services in a WSDL, and provide the implementation of the "wrappers" that access the legacy system through any of its native APIs available. Once these wrappers are written and exposed as standard Web Services, any application that needs to access the legacy system can do so without any knowledge of or dependency on the specific application environment of the legacy solution.

Service Description—Using WSDL

WSDL, Web Services Description Language, is the most common language to describe services. WSDL is written as an XML document. WSDL has two key parts:

- Interface: the list of operations in the service and the contract of each operation. The contract includes describing the number and type of inputs and outputs for/from the operation.
- Bindings: the specific details on where the service is located and the protocol (say, HTTP, or JMS) required to access the service.

Discovering the Services—UDDI

UDDI, ***U**niversal **D**escription, **D**iscovery, and **I**ntegration,* is a web-based distributed directory that enables businesses to list themselves on the Internet and discover each other, similar to a traditional phone book's yellow and white pages.

Though originally envisaged as a mechanism for discovering services over the Web, it is more useful in an enterprise SOA scenario where services provided by various departments in the organization are available for composing business processes or for use in other custom integration scenarios.

When discussing UDDI, though as a technology it is useful in both in the intranet within the enterprises and on the Internet, intra-enterprise scenarios were unduly dwarfed by the hyped up Internet-based B2B and marketplace scenarios as the key consumers of WS—and by extension, UDDI. Now there is more value seen within the enterprise than for internet marketplace usages. All said and done, Web Services is one technology that stands out just for its sheer simplicity! With other layers like ESB adding on top of WS, there is now more meat in the technology offering for Enterprises to adopt WS.

Once there are services in a distributed organization, a services registry will become imperative. Even if not as originally envisaged for UDDI—where systems will automatically discover services and actually use the services. More likely usage is that when business processes are being composed based on business needs, service registries may be used to locate services and use them in the processes. Such usages will be more for information purposes, rather than dynamic and automatic discover-and-use scenarios. For such registries, though UDDI does offer a standard mechanism, using UDDI is not necessary. Any registry provided by the SOA framework will serve the purpose.

Containers to Host Web Services

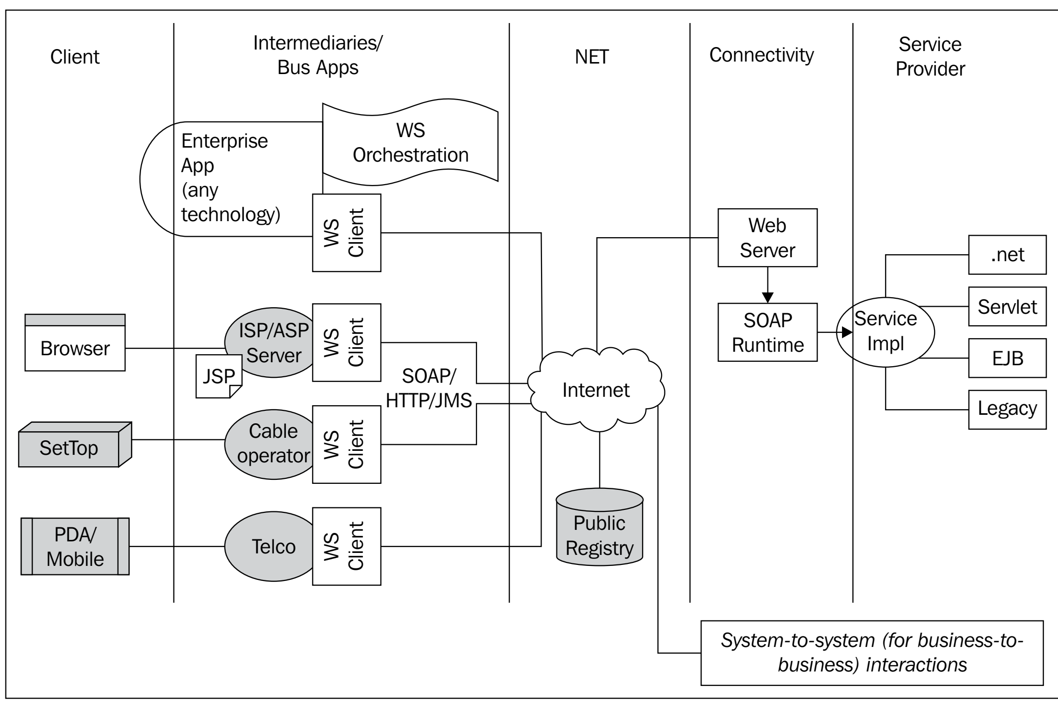
Web Services essentially provide a mechanism to make a request to a service, represent the request on the wires, and have a simple response mechanism to the request. The containers that host the services will receive, understand, and process the Web Service requests.

The containers are programs, either free standing or running on other application platforms such as .NET or JAVA EE, that host the services. The logic and processing in the services are written using a language supported by the platform and the service is "deployed" on the container. The service may perform processing in its code, or may just delegate the processing to an existing legacy system. The containers are, in short, a loose non-standard entity today—they could be anything that can receive a Web Service request, identify the service being invoked, direct the call to that service, execute it, and return the response back to the calling program.

Web Services have well defined standards to describe the services, manage the security and transactions, and represent the requests and responses on the wires. What Web Service standards don't (yet) define, though, is the actual infrastructure for providing the "middleware" for SOA. By and large, the WS standards support a simple P2P (peer-to-peer) model: a service consumption point (application) and a service provider (endpoint). In a more widespread services environment, that is more a many-to-many scenario than a simple P2P, WS by itself is not adequate. Add reliability and other requirements, and then there is a lot more additional infrastructure needed to actually provide the run-time environment. The key problem here is to provide a services container—beyond just a simple web server and servlets that provide SOAP handling, more as an afterthought from an infrastructure otherwise meant for other purposes—to serve content or execute dynamic servlets/functions.

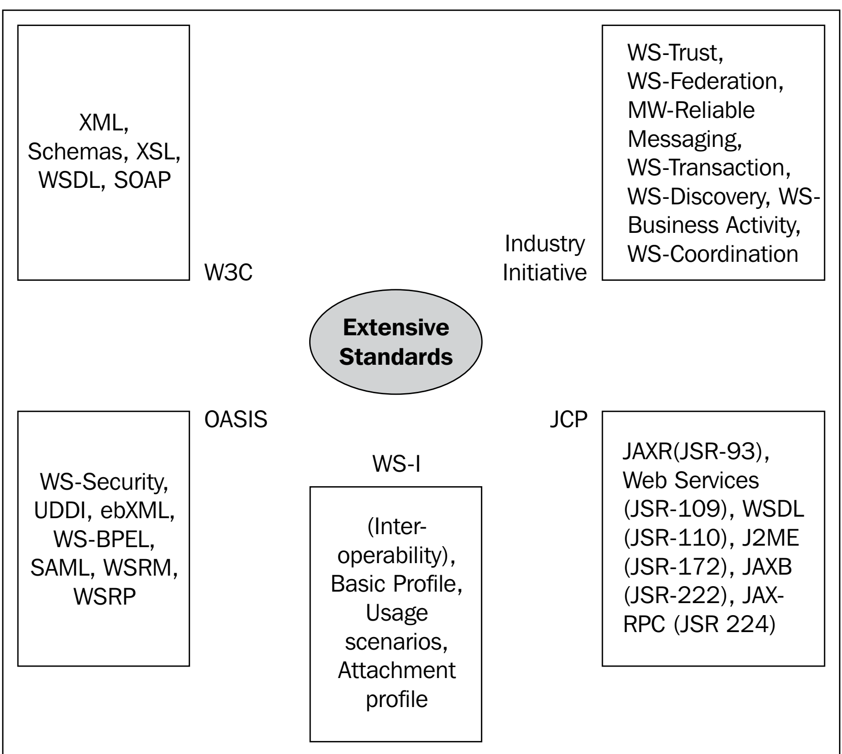
Standards Foundation

Web Services evolved from SOAP being contributed by Microsoft and taken to higher levels of maturity by IBM, before it was contributed to W3C, and got backing from most vendors. Considering that the target domain of Web Services is for systems running in heterogeneous environments to be able to "talk" to each other, for this to be a reality the system "developers" have to talk to each other. This includes OS vendors, application platform vendors, hardware (server) vendors, enterprise application vendors, and just about anyone that may be producing or consuming "services". A typical environment is shown in the following figure.



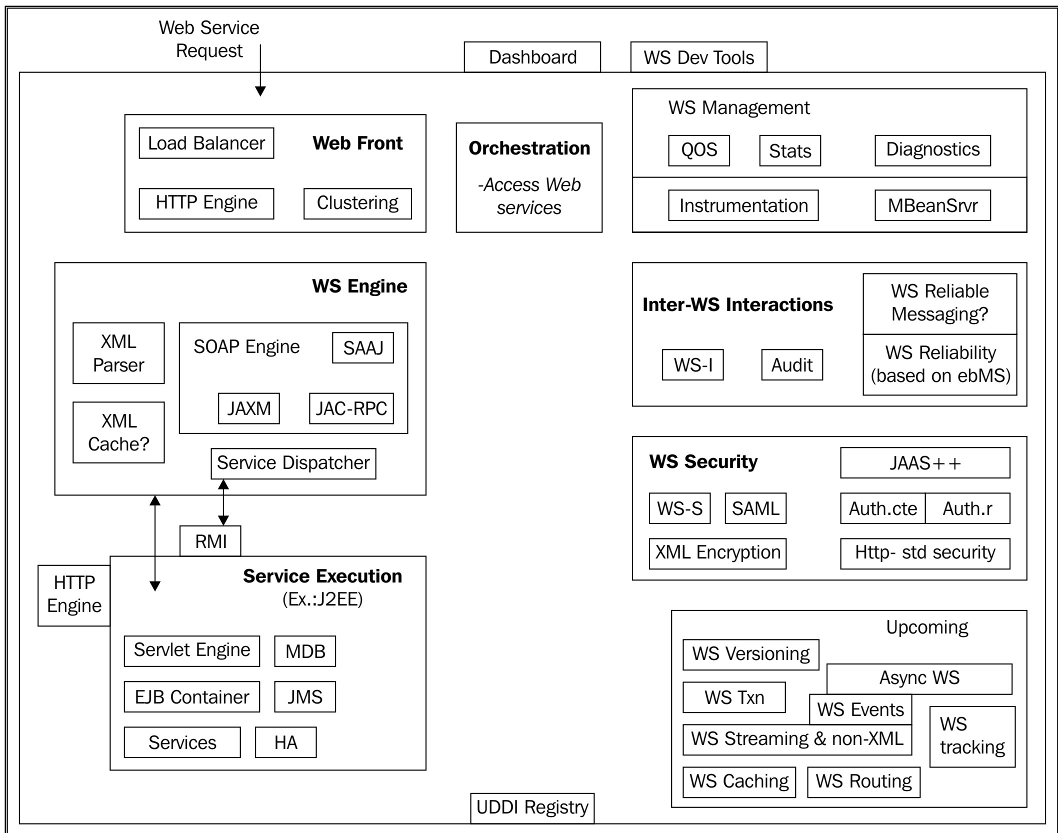
Once the market saw the value in Web Services, it was just a question of time before all vendors lent their support and muscle to Web Services. Getting W3C to host the standard also helped gain wider acceptance. Soon, a very extensive set of standards emerged—and still continues to emerge. The Java vendors, through the JCP, adopted many of the Web Services standards into the Java/JAVA EE fold.

This set a detailed framework of standards for Web Services infrastructure and applications. A summary is shown in the following figure.



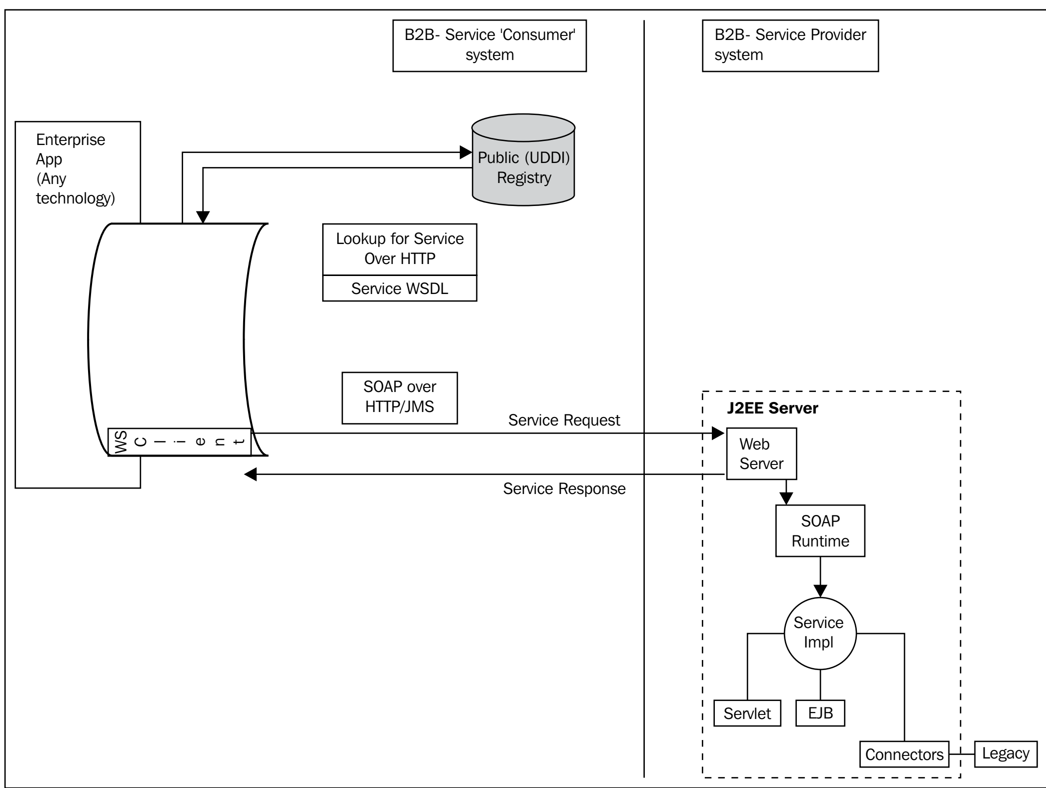
With standards emerged infrastructure and platform solutions. All existing vendors adopted Web Services into their core infrastructure. JAVA EE vendors now provide out-of-the-box support to both host Web Services and to invoke external Web Services. Enterprise application vendors like SAP provide out-of-the-box support to access the ERP functions of SAP as Web Services. XML-aware security infrastructure including firewalls came up to provide the plumbing needed for the XML traffic. XML databases showed up assuming all the data flowing through the enterprise as XML documents will need to be stored and retrieved.

Slowly, but surely, a very elaborate infrastructure emerged, based on the Web Service standards. see the following figure.



Application Platforms (JAVA EE) Hosting Web Services

Web Services can be implemented in any platform. The SOAP handling layers can either be built as part of an application or as an out-of-the box infrastructure provided by vendors. JAVA EE and .NET, the leading application platforms, both provide good support for Web Services extending their base application models, as shown in the following figure. In JAVA EE, any Session Bean or a Servlet can be set up to be a Web Service by just providing the required meta-information in the package descriptors—without requiring any programming.



Using Services to Compose Business Processes

In SOA solutions, once the services are defined, the primary use case for services is in Business Processes. Given that the services themselves provide business functionality, an application layer that uses these services is bound to be at a much higher level in terms of functional layers. And it is quite unlikely that the processing involved at this layer of the Enterprise Solutions space will be programmatically too involved. It would be closer to high-level business processes that would typically span departments within an organization and in most cases, can be stated clearly in a flowchart-like depiction. These are called Business Processes in integration terminology. And the process of "stringing" together services to "perform" a business function is called "Orchestration".

Simple Integration Applications

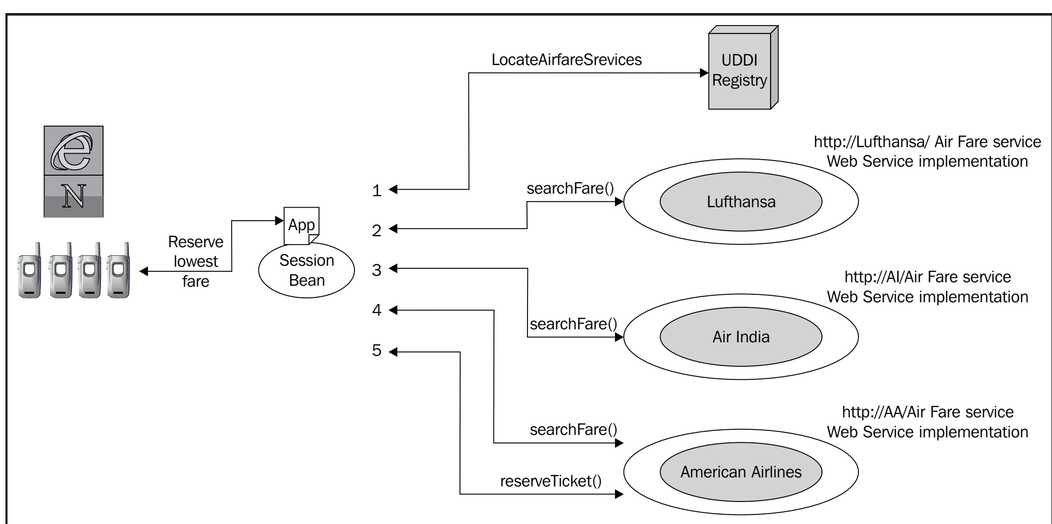
Once services are available in a services infrastructure, and a mechanism is available to "invoke" these services, then the immediate usage option would be to build these accesses into any application that needs access to these "services". This is done by programming the service access into these applications, or in some cases, writing new integration applications that would access the various services available to enable some business process, As in the example of a customer order tracking query, which we discussed earlier in the chapter.

Simple integration applications are programmatic in nature, with the service access made from other programming environments such as C# or Java. Given that such integration accesses would typically be at the first level of business functions that use heavier business functions at the departmental level, there could be alternative integration approaches that more closely model the relatively simple nature of such processes. These are now broadly called Business Processes.

Business processes model the actual business flows in an enterprise that span various departments in the enterprise. In SOA, with services being exposed by the departmental applications, stringing together these services to support an actual business function produces what is commonly known as a "Business Process".

Simple Business Processes—Orchestrating the Services

The act of putting together a set of services to perform a business process is called Orchestration. A trite example often used to highlight this is an airline ticket reservation example, using Web Services exposed by each airline (highlighted in the following figure).



Business processes essentially capture the "who-what-when" in a business function. To describe any given business process, one needs to articulate who does what and when. This will describe who will execute this service, where there is workflow involved. The process should also state when each of the steps will be expected—in terms of what are the preceding steps needed, and what should be the flow paths from each of the steps. The "what" aspect will describe the specific Web Service to be called at each step in the process—and what should be the inputs to the step based on the current state of the process as influenced by the preceding steps that executed in the process.

Orchestration could theoretically be effected in any programming environment. One could make the Web Service (or services in any other SOA environment) call from a Java or C++ program, from a Servlet, or even from a legacy programming environment like a mainframe. All that is required is the programmatic connectivity to make Web Service calls. However, in the spirit of simplifying enterprise integration through SOA, and given that the business processes can often be expressed entirely as simple flowchart-like descriptions, it just seems natural that there could be a better programming environment for defining business processes.

With XML being already well imbibed into SOA, XML became the choice to define the languages for business processes. BPEL (Business Process Execution Language) for Web Services, from OASIS, is a front runner in this space. BPEL is a notation based entirely on XML—the business process definition (program) is an XML file. BPWL-J is a variant of BPEL, where Java snippets could be embedded into the BPEL file—much like embedding Java codes into HTML, in JSPs.

Business Processes, defined in any language, will be executed in business process execution engines. Key aspects of a Business Process environment are Process Definition in languages such as BPEL, process instances—the runtime instance of every execution of the process, workflows—where user intervention is involved as part of the process, and activities—the steps in a process.

Advantages of Business Processes

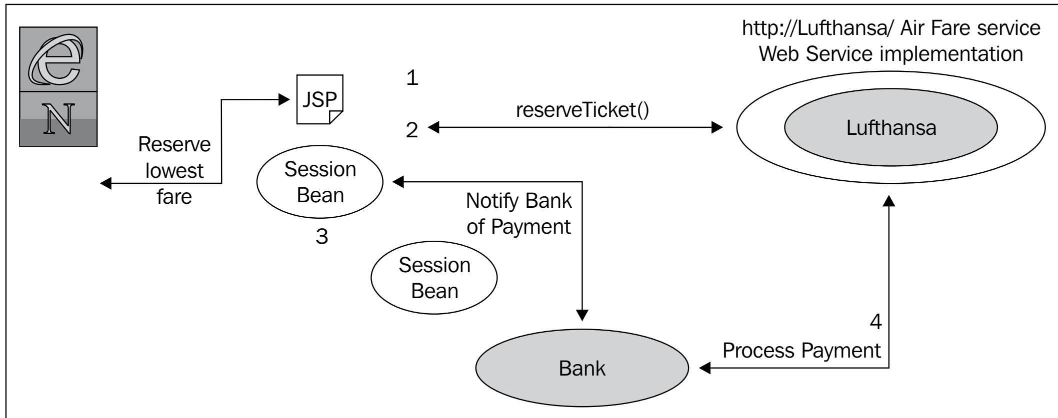
Business Processes offer a highly simplified programming model that very effectively captures the business processes in any organization. Business Processes represent a simple programming model that closely models the loosely coupled nature of business flows when they span departments/applications. The processes are components themselves, and can be made available as services for access from other business processes or even externalized and made available as Web Services. Business processes form an important layer in SOA infrastructure, which clearly captures all the business processes in the organization, with these processes using services from various other applications in the organization.

Choreography—Multi-Party Business Process

Orchestration is typically a business process involving one "role"—a single "execution point" for the process. Business doesn't always exist in such simple terms. Especially when there are independent entities involved—as in Supply Chain and other similar extended enterprise environments. Choreography models such multi-party business processes can be seen as an extension of Orchestration, with the business processes here involving multiple parties. (Refer to the previous figure.)

To state it simply, Choreography is about coordinating a set of individual processes being executed by each party, with steps in the processes accessing the other parties. Choreography describes the interactions and dependencies among the interactions across the parties.

As with anything in the Web Services and Integrations space, standards play a very crucial part in the technology evolution in an inherently heterogeneous enterprise environment. WS-CDL and WSCI (Web Services Choreography Interface) are two such standards from the W3C.



WSCI describes how Web Service operations—such as those defined by WSDL—can be choreographed in the context of a process in which the Web Service participates. Interactions between services—either in a business context or not—always follow and implement choreographed message exchanges (processes). WSCI will always co-exist with languages such as BPEL that describe the actual process at each party in a multi-party scenario. WSCI is a starting point for the WS-Choreography working group from W3C that is now looking at the broader Choreography space.

Collaboration between Enterprises (B2B)

Collaboration is a type of Choreography targeting business-to-business interactions. Apart from defining the choreography of the "exchanges" between the parties, Collaboration also describes the various mechanics and contracts involved in a B2B scenario—such as the Partner Profiles that are used to negotiate the exchanges and Partner Agreements that form the description of the actual interactions.

The ebXML standard, from OASIS, is a front runner in this space. The ebXML Business Process Specification Schema (BPSS) from OASIS provides a framework to configure business systems for executing business collaborations across partner organizations in an extended enterprise. The specification models the interactions by defining the business exchanges (the individual document flows) first, and then defining the choreography of such flows. The current version of the specification schema addresses collaborations between two parties (Binary Collaborations) as well as collaborations involving more than two business partners (Multiparty Collaborations) as a synthesis of binary collaborations.

In a Nutshell, Orchestration and choreography both relate to connecting Web Services in a collaborative fashion. The capabilities offered by the available standards will be vital for building dynamic, flexible processes. The trend is to have a set of open, standards-based protocols for designing and executing these interactions involving multiple Web Services. Of course, which standards gains sufficient traction remains to be seen though a few like BPEL are holding a lot of promise.

We will see more of this in the subsequent chapters in this book.

SOA Security and Transactions

Security and Transaction management are important and implicit infrastructure functions expected from Application Platforms. Security infrastructure includes Authentication, Authorization, Trust, Encryption, Audit logging, and Non-Repudiation.

In SOA, the security infrastructure depends on the specific platform chosen for implementing the SOA solution. If a well established application platform such as JAVA EE is the chosen platform, then the security infrastructure provided by such application platforms will be quite adequate to support most security requirements. For example, in the case of JAVA EE, other than non-repudiation, all other functionality listed above will be possible.

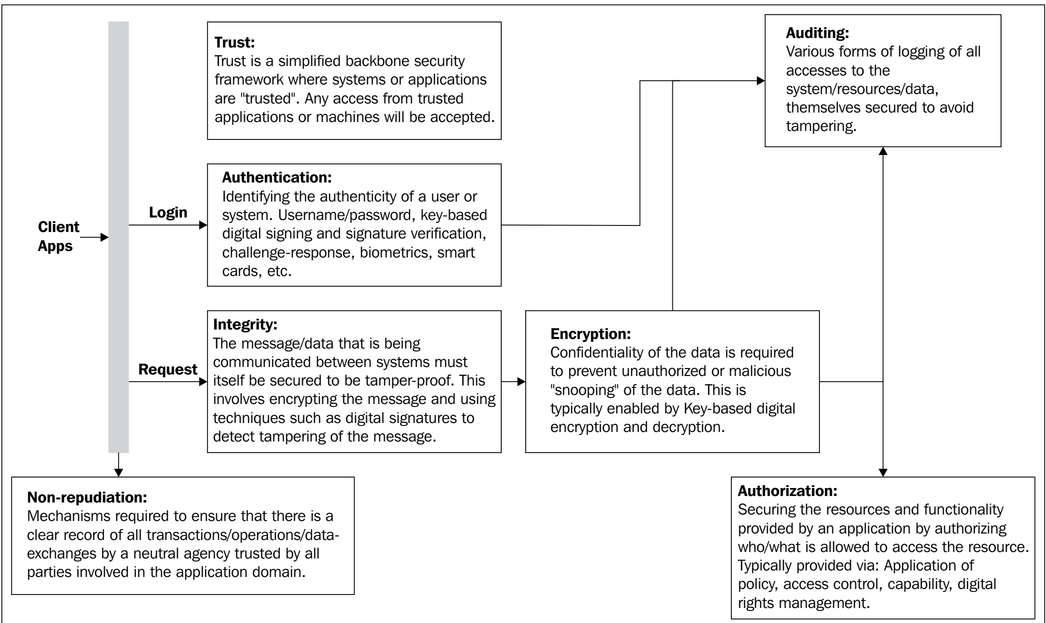
However, given that SOA is a framework typically used in enterprise-integration space implicitly supporting multiple technology domains, solutions beyond application platform vendors will be needed. This is where a further level of security infrastructure, beyond what is provided by application platforms, is needed. In this space, the Web Service security standards (WS-* suite) are the most comprehensive. As they target the Web Services space, they inherently will support heterogeneous technology domains.

Security Challenges in a Services Environment

The SOA services environment is inherently distributed to deliver integrated, interoperable Services solutions. Ensuring the integrity, confidentiality, and security of the services through a comprehensive security model is critical, both in an intranet SOA scenario and more so in an extranet/Internet SOA scenario.

Apart from providing all the security capabilities expected from enterprise application platforms, SOA security will need to address the key challenge of providing widely distributed security domains and enabling interoperability across all participants in the SOA solution space.

This includes authentication, authorization, trust-based interactions, passing user credentials and tokens across systems, and more, as shown in the following figure of security infrastructure.



Simple Middleware Systems Security

While the full-fledged security infrastructure for service-oriented application environments is under specification and implementation, current SOA infrastructure will also need to provide some security. In most cases, the security will take the form of the support provided by the base infrastructure on which the SOA platform/application is built.

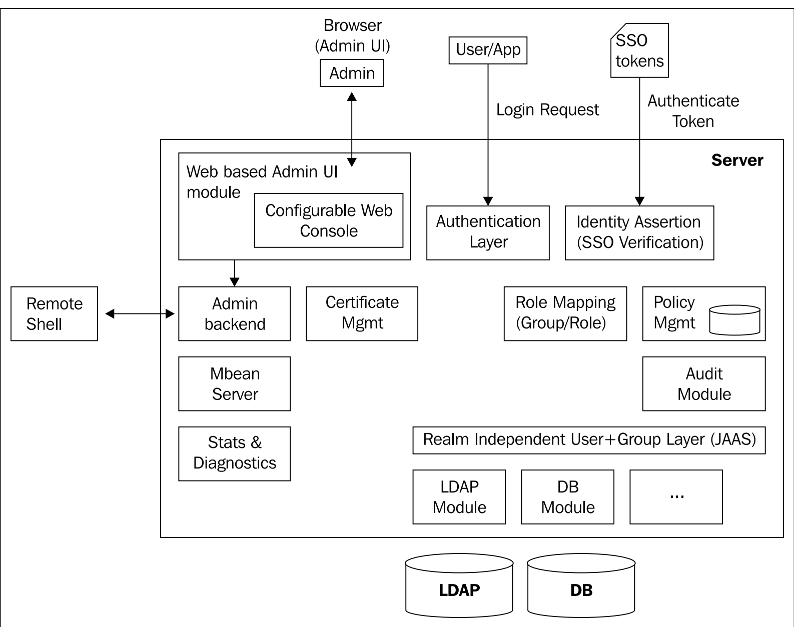
In most SOA solutions (home-grown or otherwise), one can expect one of:

- Web Server
- JAVA EE Server
- Message Server

All the three above provide well defined security infrastructure.

Security in Java Infrastructure

In the case of Web and JAVA EE, for HTTP access security support includes authentication and secure communication (via HTTPS). In the case of JAVA EE, there is well defined support for Authorizations to control the access to the various resources including Servlets and Session Beans (EJBs). In the case of Messaging platforms, security will include authentication and access control to the various destinations—both to send and to receive messages.



Using the above, a fairly secure SOA platform will be available. However, the infrastructure will be specific to the chosen platform/vendor, as shown in the previous figure, and will not be technology (such as JAVA EE or .NET) neutral. Evolving standards such as Web Services Security standards address the security needs of SOA solution scenarios better.

Microsoft.NET Security

Microsoft .NET platform provides an integrated security model that ties in with all the security features that are part of the Windows operating system. This includes authentication and authorization. The authentication includes Basic Authentication and Digest Authentication. For distributed applications spread across multiple servers, the .NET security solution includes Integrated Windows Authentication, wherein once a user logs in the information is shared between applications using either the Kerberos or challenge/response protocols to authenticate users. For Internet-based applications, .NET provides integration with the Microsoft Passport service.

Web Services Security for Loosely Coupled Services

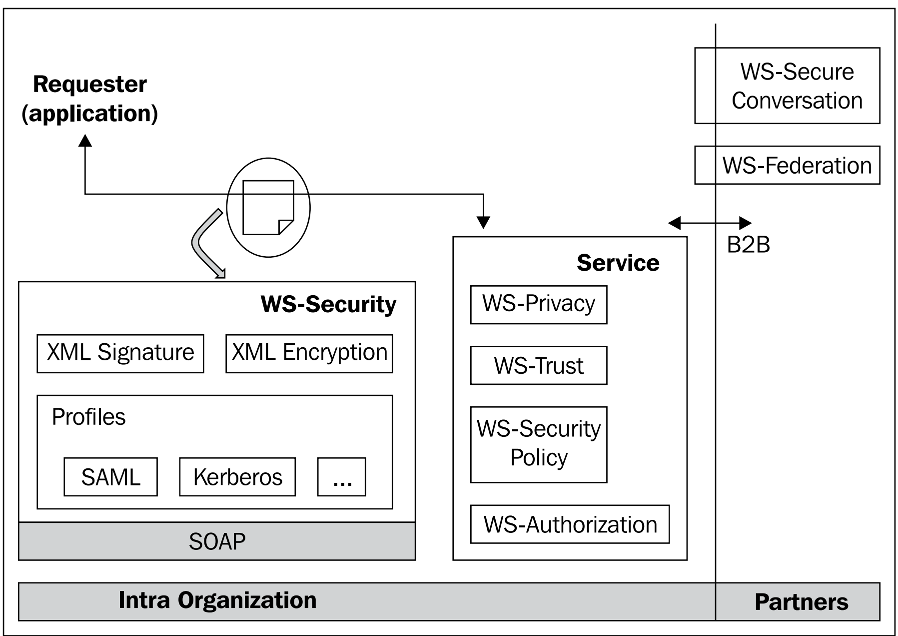
The security models in application infrastructures such as .NET or Java are primarily designed for single-application or single-server security. At best, they support accessing user databases that may reside in any enterprise-wide repository such as Directory Services. In some cases, where there is a distributed security model available, they are predominantly in tight trust-based domains, wherein a set of applications, in a fairly localized context, share trust and security interoperability.

In SOA, the level of distribution is expected to be much higher. Spanning geographies, and in many cases, even spanning enterprises. The prevalent application infrastructure security models fall short when there cannot be the same levels of tight domains or trust levels as are available with the conventional application infrastructure environments.

The emerging Web Services architecture provides the ability to deliver integrated, interoperable solutions. Ensuring the integrity, confidentiality, and security of Web Services through a well defined security model is critical. Web Services Security is an initiative from the industry for developing a set of Web Service Security specifications that address various aspects of security ranging from providing protection and privacy for messages exchanged in a Web Service environment to providing authentication and authorization capabilities for access to services. The WS-Security model brings together formerly incompatible security technologies such as public-key infrastructure, Kerberos, and others in a practical manner for use in the heterogeneous and distributed IT environment prevalent today.

Emerging Web Services Security Standards

Emerging Web Services standards include a message security model (WS-Security) that provides the basis for the other security specifications. Layered on this, we have a policy layer, which includes a Web Service endpoint policy (WS-Policy), a trust model (WS-Trust), and a privacy model (WS-Privacy). Together, these initial specifications provide the foundation upon which we can work to establish secure interoperable Web Services across trust domains (see the following figure of security-related standards).



The current specifications include:

- WS-Security: describes how to attach signature and encryption headers to SOAP messages. In addition, it describes how to attach security tokens, including binary security tokens such as X.509 certificates and Kerberos tickets, to messages.
- WS-Policy: describes the capabilities and constraints of the security (and other business) policies on intermediaries and endpoints (e.g. required security tokens, supported encryption algorithms, privacy rules).
- WS-Trust: describes a framework for trust models that enables Web Services to securely interoperate.
- WS-Privacy: describes a model for stating privacy preferences for Web Services and their consuming applications.

WS-Security is the core here that directly layers on the SOAP mechanisms.

Additionally, there are many other contemporary topics in distributed application environments that are being addressed in the Web Services context. These include:

- WS-SecureConversation: will describe how to manage and authenticate message exchanges between parties including security context exchange and establishing and deriving session keys.
- WS-Federation: will describe how to manage and broker the trust relationships in a heterogeneous federated environment including support for federated identities.
- WS-Authorization: will describe how to manage authorization data and authorization policies.

WS-SecureConversation and WS-Federation are both relevant in B2B scenarios where there are long-running business operations (as is common in Supply Chain automation). In an intra-organization SOA scenario, these specifications may not be of much use. WS-Federation is the one with least likelihood of early adoption as this requires well accepted multiple security and trust domains in place for automated interactions across these domains.

Transactions in SOA

In a distributed application environment, it becomes imperative to have support for distributed transactions. In the 1990s, X-Open standards formed the basis for distributed transactions and all major application platforms, and database vendors provided support for the TP/XA standards for distributed transactions. This standard relied on the two-phase protocol for supporting transaction semantics in multi-resource distributed application architecture, where each resource is the likes of a database server instance. In this traditional distributed transaction model, though, there is a restriction that there be one single global transaction manager for this distributed environment. This is an acceptable restriction in a homogenous environment like say JAVA EE application server as the mid-tier and a transaction coordinator with multiple back-end resources participating in a simple 2PC configuration. Even though platforms such as JAVA EE and CORBA provided good support for this, they did not address the heterogeneous environment even in this space—where there would be multiple different JAVA EE vendors that need to participate in a single transaction domain. Standards such as JTS and IIOP did support this—however, the implementations available were not sufficient or were not robust. In a heterogeneous technology environment such as multiple technologies existing in the business processing tier, there are no good models for distributed transactions.

Web Services and SOA being an even more loosely coupled environment, the challenges for transaction coordination are further amplified.

Extended Transactions or Activity Services are an emerging concept. In conventional transactions there is a simple model for defining transaction boundaries with an option either to commit or roll back the transaction. In Extended Transactions, where there are very diverse resources participating in the transaction, and in a collaboration type scenario, the transactions may span extended periods of time, making conventional transaction semantics impractical. In such environments, there is an alternative model where the rollback is implemented by providing compensating business operations. Say there is a reserve-inventory business method, the "rollback" for this operation will be effected by another business method called "unreserved-inventory".

Web Services Transaction—A Standard

WS-Transactions is the initiative from a few vendors, now driven by WS-I, to provide transaction support in services over the Web (now SOA) environment. The Web Services Transactions specifications describe an extensible coordination framework (WS-Coordination) and specific coordination types for short-duration, ACID transactions (WS-AtomicTransaction), and longer-running business transactions (WS-BusinessActivity).

The WS-Transaction interface defines what constitutes a transaction and what will determine when it has completed successfully. Each transaction is part of an overall set of activities that constitute a business process that is performed by cooperating Web Services. The overall business process is formally described using the Business Process Execution Language (BPEL). WS-Coordination is a companion specification that defines the context and exactly how information is exchanged during the business process.

Infrastructure Needed for SOA

SOA Solutions will include two distinct layers—applications and the infrastructure. Applications will include the Service implementations, XML schemas and transformations, and the Business Processes. The Infrastructure will provide the framework for hosting these services and manage the service execution "plumbing".

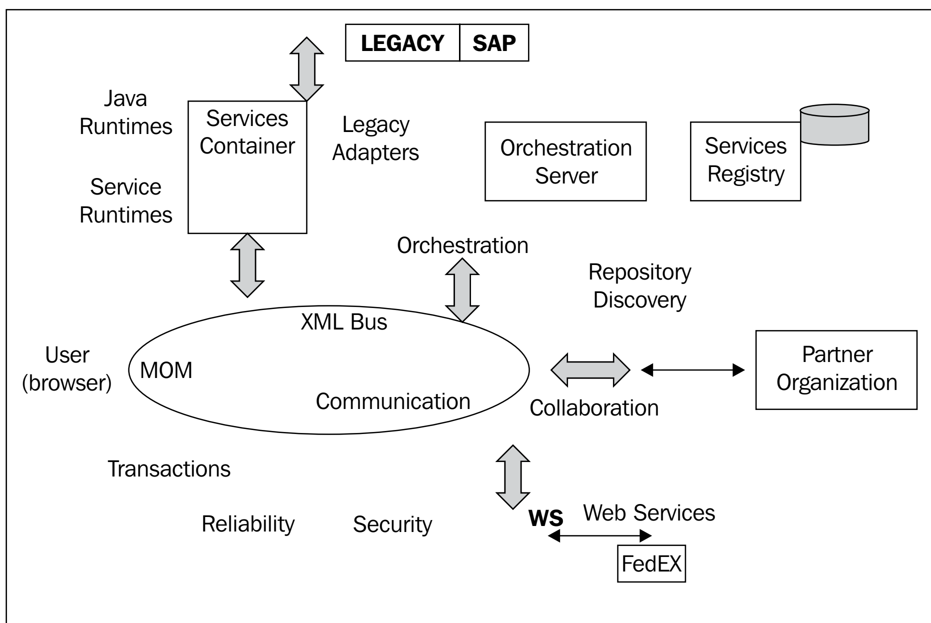
Even though SOA is more of a concept than infrastructure, the SOA solutions will need an environment to run in. Whether it is home-grown infrastructure built on existing platforms or it is a ready SOA infrastructure, there are some common infrastructure elements that will need to be available. Let's look at what these are.

Service Execution and Communications

SOA at its core involves the services runtime, invocation, and communications framework, and services consumption and orchestration engines. Most of these have a strong technology and standards basis. Any SOA infrastructure must factor in the prevailing technology approaches and standard specifications.

SOA Infrastructure (see the following figure of the building blocks of SOA infrastructure) is mostly services centric with the key constituents being:

- Service container—for executing the services
- Service implementation framework
- Adapters—to connect to legacy back-end applications
- Service discovery—registries
- Orchestration of Services (process engines)
- XML Handling
- Communication bus
- Web Service call outs—to enable accessing external services via HTTP as Web Services
- Security
- Transactions
- Clustering and High Availability



Types of Component Services

Services in an SOA can be of three types:

Simple—basic services that have a simple interface, and when "invoked" perform a business function.

Composite—these are aggregate services, which when invoked perform a sequential set of business functions, where each such function is a simple service. Such composite services would typically be business processes, say written in a process language such as BPEL.

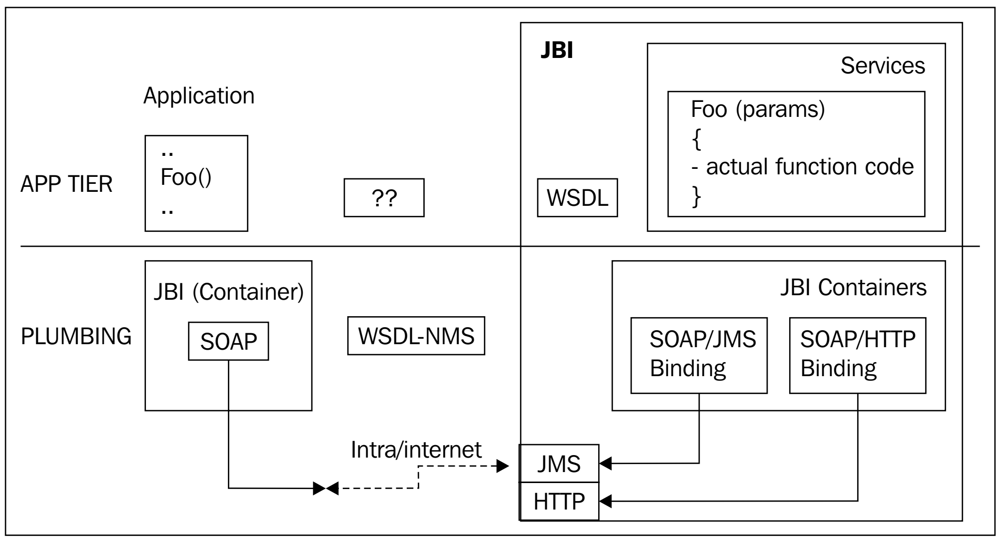
Conversational—in an extended enterprise, involving partners such as suppliers, there could be business processes that span executing business functions across multiple organizations. A good example would be Supply Chain automation. These are again like business processes, but with long latencies in between—and conversational in nature—requiring maintenance of a conversational state. For example, the specific PO against which various process steps are executed and the current state of the business process will be the conversational state for such a process. This will be available at all points of the execution of this process.

Simple Services are the fundamental building blocks of any SOA solution. These services can either be ground-up services, which have self contained service functionality, or could be services that "wrap" existing legacy functionality and tie them into the SOA framework. In a typical Enterprise Integration scenario, there would be more of the latter than the former. But in emerging new application architectures, it is quite possible that a complete first-level business application is written using SOA as the base paradigm, wherein the application modules are all implemented as Services.

Service Containers (Execution Engines)

the service invocation mechanism and the service implementation are the most important parts of the infrastructure. This essentially involves a framework to host the services and a mechanism to "invoke" to these services.

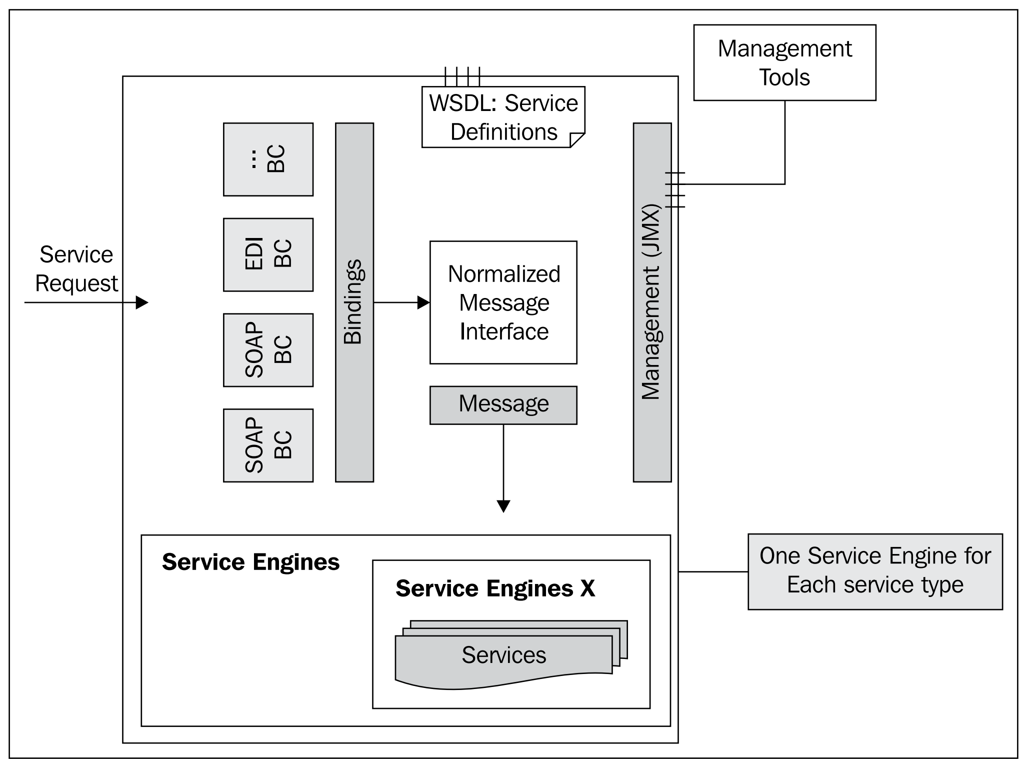
Both service implementation and service invocation are directly dependent on the chosen platform for hosting the services. Given that SOA is just a concept, just about any server-side infrastructure could be used to host the services. These include custom Java/C programs, JAVA EE, .NET, and CORBA.



Expectations from this hosting environment, as shown in the previous figure of service containers, will be:

- A programming abstraction to define the services
- An infrastructure to execute services in (EJBs, .NET, CORBA)
- A well defined mechanism to describe services (WSDL, CORBA IDL, EJB interface and name, etc.)
- Client-side invocation models (HTTP to servlets, EJB calls over RMI, CORBA over IIOP, Web Service calls using SOAP over HTTP)
- A clear framework to ship requests from the client to the server and send responses back, such as JRMP, IIOP and SOAP

JBI (Java Business Integration) is an attempt from the Java Community, through the JCP, to solve the services container problem. A generic framework is being evolved to provide a run time environment for the services. A Service can be written in a generic way, and be hosted on a JBI-compliant container from any vendor. This is not possible in a simple WS environment. WS standards only describe how a request is represented over the wire. They do not provide for any mechanism to "host" services. JBI tries to fill this gap. (See the following figure.)



Communication Infrastructure—Under the Covers

Communication infrastructure is the next most significant aspect of any SOA infrastructure. Simple Web Services do not have nor expect an elaborate communication platform. Just simple HTTP over the Internet will suffice, as it is essentially a point-to-point model. An enterprise-grade SOA infrastructure, though, is typically a many-to-many environment. Several service providers and several service consumers. This will need a powerful communication backbone.

Communication infrastructure could range from simple sockets and HTTp in P2P environments, the hub-and-spoke mechanisms employed by Application Platforms such as JAVA EE and .NET, to the complete enterprise-grade messaging-based ESB communication infrastructure.

Communications being at the core of any SOA infrastructure, it will be expected to be highly reliable and secure. High availability and performance are the other two key factors. The communication could be the weakest link in the performance path.

Communication "Bus"—At the Core

In an enterprise-wide SOA scenario, services will be available from multiple applications running on multiple systems, probably even geographically distributed. The communication infrastructure at the bottom of the SOA infrastructure stack must be capable of handling such a configuration and still service high loads and throughputs. One rapidly emerging approach is to have a communication "bus". Taking the communication beyond just a simple RPC mechanism—to a service-highway concept, wherein service requests are pumped "into" the bus and at any given point multiple requests could be in "transit" in the bus and get delivered to the required service endpoint (where the service is running). Likewise, responses also get shipped back to the service invocation point/application through this bus.

The use of a bus is very transparent to the actual SOA application and is deep in the guts of the SOA infrastructure, and will be well abstracted by the SOA infrastructure vendor.

The advantages that a communication bus offers over conventional P2P frameworks like simple HTTP or even regular RPC frameworks such as EJB, is that:

- P2P can handle one client-to-server connection well:
    - It is not meant to handle a many-to-many scenario, where there are multiple clients (in our case, multiple SOA integration applications/processes), and multiple servers (the services and their containers).
- RPC frameworks are inherently meant to be in a hub-and-spoke model—where all the client applications are at the spokes and the server providing the services is at the hub. In scenarios where there are multiple hubs, the hubs will need to talk to each other. And here again an RPC abstraction is typically adopted, providing inherent scalability limitations.

A well designed communication bus, however, would be able to handle the above distributed configurations well—as there will not be any hub. The communication bus will provide the required abstractions to hide the fact that there is geographical distribution. So an endpoint in the local New York office or in the London office will "seem" the same to a client application that is accessing both these endpoints. And under the covers, the bus will be optimized to treat the communication packets as messages and not as RPC requests. So there will be additional optimizations possible such as aggregation, compression, and such like, which, while not reducing a single service-request's response time, will significantly increase the throughputs.

MOM

Message-oriented Middleware systems provide a framework for executing services using a messaging platform as the primary service transport mechanism. Typically, in MOM a full-fledged Message broker will provide the messaging infrastructure required. Enterprise Services Bus is a good example of MOM (though it does a lot more than just MOM). We will discuss more of ESB later in the book.

XML Backbone (XML, Transformations, and Persistence)

In a multi-application enterprise scenario, it is hard to imagine interactions happening without the ubiquitous XML involved. This is in addition to any use of XML in the actual middleware plumbing (such as use of SOAP).

In the SOA infrastructure, services are essentially business functions that could either run self-contained (say, their logic is in Java) or are "wrappers" for back-end legacy processing, like say an SAP or Baan or mainframe or even JAVA EE/.NET. The primary objective of the infrastructure is to enable systems to "talk" to each other and also enable simple business processes that "orchestrate" these services to provide some aggregate business logic.

Now, the communication could be based on any communication framework—simple sockets, HTTP, or a good Messaging backbone (Sonic uses Sonic MQ—an enterprise-class distributed JMS implementation). And the actual messages that flow are service requests (not much unlike a SOAP request). And in a well designed Enterprise Integration environment, request parameters and responses also are XML documents.

Once XML becomes the key form for information flows through the systems, then additional needs immediately crop up, say, like XML transformations. For example, a Purchase Order in the ERP system may have different fields, when compared to how the OrderProcessing Application understands a PO. And when a PO is sent from one to the other, somewhere it must be transformed from one structure to the other. So XSLT/XQuery comes into play. And once XML documents are flowing, one can quickly think of cases where the "processing" of the document is "attached" to the document, rather than passing the doc as an input to a "process". In the mode where process is "attached" to the doc, what actually happens is that the infrastructure handles the doc and its processing through the various Service containers/nodes. This is a significant feature of good Enterprise Services Bus implementations. We will discuss this later in the book.

Reliability and Scalability

In Enterprise IT infrastructure, Reliability and Scalability are very important to ensure the required uptime and service levels of the IT systems. In traditional application platforms, this is a relatively bounded problem—as only one platform or one cluster needs to be made reliable and scalable. In SOA solutions though, given the massively distributed nature spanning geographies, the problem gets compounded.

Reliability mechanisms should ensure that the service requests are guaranteed to reach the service containers and the responses to reache the invocation client/integration application. Reliability should ensure that the infrastructure survives the crash of any of the containers—ensuring the uptime of the system using clustering and other solutions. The system should be scalable for larger system loads. It should be able to handle spikes well and its architecture should have a provision to handle higher load requirements by expanding the cluster.

The Reliability and Scalability support is very transparent to the actual SOA application programming. The application itself would run on any platform regardless of the reliability or scalability of the platform. Therefore the reliability and scalability are purely deployment runtime considerations, to ensure that the application is up and running per the required service level guarantees.

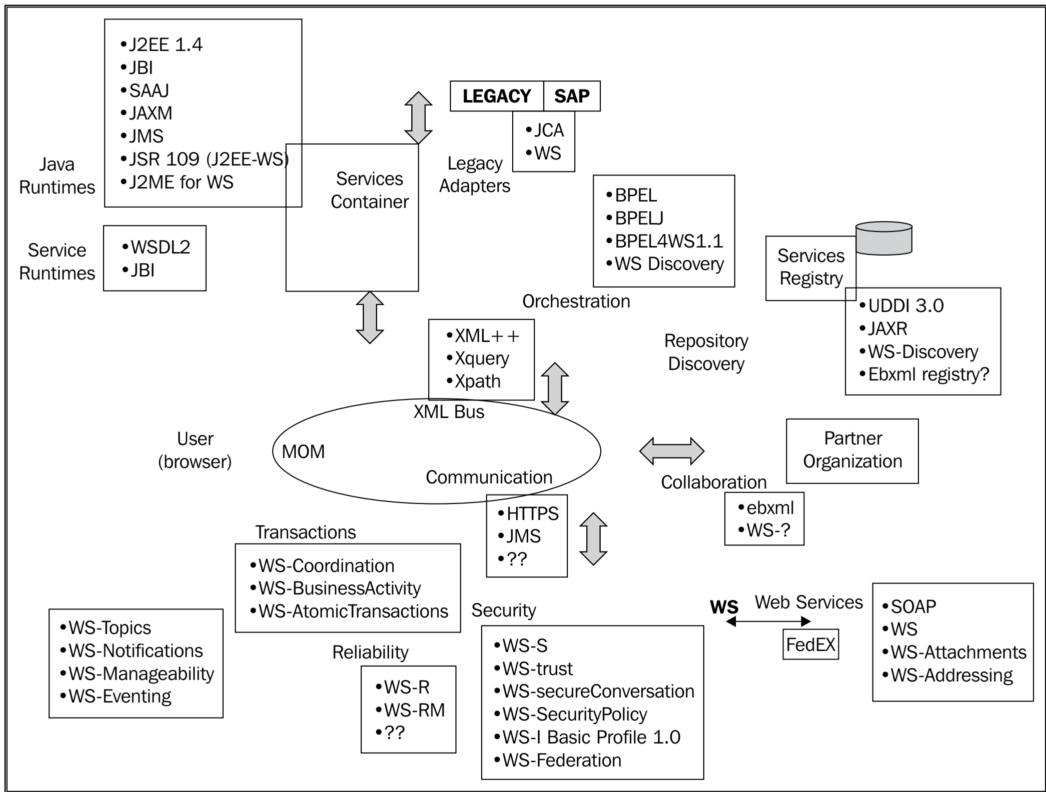
Managing a Distributed SOA Environment

The inherently distributed SOA infrastructure forms a "single" IT fabric, requiring a homogenous management environment. A contradiction of sorts—here is a solution that is integrating heterogeneous solutions—both contemporary and legacy. In such an environment, expecting a single management framework is not practical. At least not for the complete management needs, which involve both the SOA layers and the actual underlying systems (mainframe solutions, JAVA EE, .NET, ERP, et al.). SOA management is an interesting space to watch. There are players already in the space. But I find most of the solutions to be very superficial—they may at best provide fabric-level visibility. This may serve some management needs. But the ideal solution is one that offers both the fabric-level information and also the internal system-level management information, kind of like an "integrated" monitoring/management solution. Now, how this is possible is a space to watch. The likes of Unicenter, OpenView, and Tivoli may extend their existing solutions to include SOA fabric management as well.

Options for SOA Infrastructure

The SOA concepts must be well understood before embarking on any SOA-based solutions in organizations. Unlike application platforms like JAVA EE, there are neither clearly defined runtime infrastructures nor standards. While there are quite a few standards in the space, they only address parts of the infrastructure. They don't offer a complete platform like JAVA EE did for Application Servers. Even so, the standards in the space do help reduce vendor dependencies. This has to be an important consideration when choosing the SOA infrastructure.

SOA standards that are coming up—largely driven by the Web Services-related standards and to some extent driven by the Java-related Java Community Process standards. All aspects of the infrastructure are being addressed ranging from service definition and service containers to service registries, security, transactions, and reliability. A comprehensive overview is shown in the following figure of SOA technology standards.



An IT organization may, based on the SOA concepts, "assemble" the required infrastructure or use ready-made infrastructure such as ESB solutions. Some of the options are Web Services-based infrastructure, Application Platforms, MOMs, Integration Brokers, and ESB.

Web Services

Web Services infrastructure being widely prevalent due to the omnipresence of HTTP and XML, Web Services are a good option for SOA. Web Service infrastructure may be built using the open-source Apache Axis framework or the more elaborate infrastructure provided by the likes of Cape Clear or Systinet.

However, web services alone are insufficient for SOA integration projects. Web Services provides a simple mechanism to define web services and call web services. It doesn't provide any communication infrastructure. In most cases, it would be simple HTTP over TCP/IP—in a P2P mode and doesn't handle a widely distributed enterprise services environment well. Current Web Services standards lack specifications for management of the enterprise qualities of service (reliability, security) required. They also do not provide the general-purpose mediation and process management required to bridge the gaps between the needs of web service consumers and the capabilities of web service producers.

Application Platforms (JAVA EE / .NET)

Application platforms are environments meant to run self-contained applications, provide very good abstractions for designing and partitioning the applications—into UI, Business and data tiers—and further, organizing the functional modules with a clear separation in place between modules. JAVA EE and .NET are two such popular platforms—well suited for standalone applications where the relationships between application components and external resources such as databases or other applications do not change frequently. An application is typically deployed into a single server or a cluster of like servers.

Like Web Services, application platforms do not handle distributed application environments well—where there are multiple applications running across heterogeneous servers, or servers in a distributed environment. They cannot provide a unified view or manage such applications and processes. As a result, these platforms are poorly suited for managing a large number of dynamically configured services in an enterprise SOA.

Good SOA infrastructure will need service containers that allow the selective deployment of services exactly when and where you need them. Unlike application platforms where you have to install an entire application server stack everywhere, an individual piece of integration functionality is needed. This results in unnecessarily high costs for licensing, installation, and cost of ownership over time.

Simple Messaging-Based Custom Infrastructure

Messaging products (such as IBM MQ Series or MS-MQ) provide a very good foundation for building services infrastructure. However, as they do not provide any support for services as such; the services infrastructure has to be built on top of this. For SOA solution developers, this is surely an option, though, it would come at a high cost of building and maintaining the services infrastructure.

Messaging (MOM) infrastructure-based SOA frameworks effectively decouple services through the messaging server. Even so, MOM products lack a service interface. Services are represented by a message sent to a destination. Services must be written to directly call the messaging middleware, and indicate specifically where and how messages should be delivered. Any time the implicit relationship between services changes (e.g. altering message routing as part of a business process), the application code needs to be modified and the application re-deployed.

This kind of hard-coded dependency is difficult to change, manage, or track. Services linked through MOM also have implicit dependencies on their message protocol, format, data encoding, types and structures, component models, security, and error handling.

Integration Platforms (EAI)

Traditional EAI products were specifically designed for application integration. They performed application-to-application mapping and binding in a hub-and-spoke model, which concentrates too much complexity at a single point for integration of more than a handful of applications.

In development, it is too expensive and impractical to staff a project with IT professionals deeply knowledgeable about the EAI product and about the semantic details of all the applications to be integrated. In deployment, scaling an EAI product's centralized hub-and-spoke architecture to accept additional duties requires excessive computing resource in a single server or cluster of like servers run in a single LAN segment.

ESB—Enterprise Service Bus

An ESB is software infrastructure that simplifies the integration and flexible reuse of business components using a service-oriented architecture. An ESB makes it easy to dynamically connect, mediate, and control services and their interactions. The defining concepts of an "enterprise service bus" are:

- An ESB provides **enterprise-grade** qualities of service:

  Reliability, fault-tolerance, and security are inherent properties of the service binding, and are the responsibility of the ESB platform. Services delegate reliability and communication to the ESB infrastructure; they need not implement low-level communications themselves.

- **Services** on the ESB are all first-class citizens:

  Services are broadly available and configured for mediated interaction with any other service. The only programming required is the actual service implementation. No other code is needed to access or to control the services. This would be provided by the ESB infrastructure. Though Web Services will naturally be supported, the ESB will also provide connectivity to a broad range of technologies such as JAVA EE and .NET components, bespoke applications, and legacy MOM systems.

- An ESB implements a **bus** topology to make services broadly available for reuse:
  Bus topologies can scale to connect and host distributed application and infrastructure services in an arbitrarily large deployment. Hub-and-spoke or star topologies rely on a central broker, and while appropriate for managing a few resources inside a single LAN, they are unsuitable for a broad-scale SOA deployment.

Designing Services and Processes for Portability

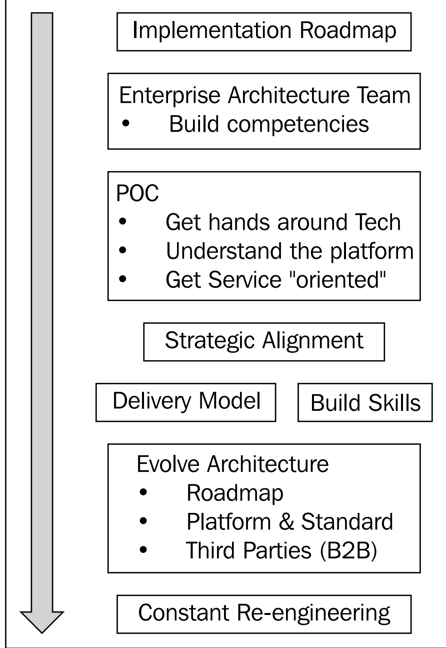
SOA addresses enterprise-wide IT infrastructure. Any SOA solution built in the organization will span several systems and technologies. This brings up a few challenges in terms of platform support of the infrastructure chosen, and also the vendor dependence of the enterprise on the chosen infrastructure. Ensuring that the vendor providing the SOA infrastructure supports all needed platforms and legacy connectivity is easily done when choosing the infrastructure platform.

The typical SOA development process involves identifying the back-end services needed, writing the service wrappers or service processing logic, defining the interfaces, deploying the services, and using/invoking the services from other applications or orchestrated processes. The challenge here is in ensuring that the SOA solutions built on this platform are not overly dependent on the specific vendor's platform. The last thing an enterprise needs to deal with is to have all the enterprise integration solutions built on a specific infrastructure and find for some reason the vendor or the products are no longer available or supported. SOA being solutions that span departments, such a risk with a vendor puts complete organizational functioning at risk.

In this section, let us discuss some of the design approaches and proactive design measures that will help mitigate this risk.

Adoption Considerations

SOA and POA are both enterprise-wide architectures in nature due to accesses to multiple applications or departments. The decision to adopt SOA or POA is normally not made by a single IT group or for a localized business solution. It generally will span multiple departments in the organization. Given this, a basic strategy could be (as shown the following figure of SOA adoption key steps):



- Start within the business layer of its application and integration projects.
- Use SOA-based web services to connect its business services to multiple interaction channels.
- The developers can use message queuing or other application-to-application protocols as the need may be.
- Then, the enterprise can add additional services to provide increased flexibility and quality of service.

Enterprises will have to make significant investments in metadata repositories.

- Align the SOA services with actual business practices, recognizable to the users.
- Start with major vertical services (such as order entry), and the horizontal services that support them (such as document faxing). Proof of concepts (POC) can be based on these.
- Web services not a must, if internal access is all that is required.

Service Implementation:

- When implementing a service, keep in mind that other services may end up using that service. When that service is upgraded, you don't want to have to retest all the other services that use it.
- Instead of tying each incoming request to a specific service, it may be best to create a middleware layer, called a service delivery bus, to handle routing.

The organization should explore the "fit" of the SOA/POA for the organization's IT requirements. The Proof-Of-Concept (POC) will help in making this assessment. In that process, evolve the overall organization-wide adoption approach, build basic SOA/POA skills, assess and decide on an infrastructure vendor, and most importantly get a buy-in from the various stake-holders.

Think Services

The basic design approach to SOA solutions involves "service orienting" the existing applications. This involves analyzing the business domain and identifying services that may be needed by other departments or cross-departmental processes from this application.

SOA is a new way of thinking about Enterprise Architectures. The change is philosophical.

- Software is a tool that supports a whole business process.
- Work with business processes rather than applications.
- Think of your applications as a collection of services tied to a business process.
- Databases become common resources rather than isolated information.

SOA adoption will initially start as an approach to integrating existing applications in the enterprise. Later, it may evolve into a serious architecture for new applications to be developed.

Model the Business Data as XML

When talking about working with existing applications in the enterprise, the connectivity and data exchange from/to those applications become the first problem to address. There are many approaches to connecting to a legacy system—including the more contemporary Java Connector Architecture and Web Services. Regardless of the specifics, one unavoidable requirement will be to model the data exchanges for the various business functions to be expected from the applications. As we have discussed earlier in this chapter, XML is the best language for this purpose. When defining services, two key activities are involved: analyze the business functions to identify the Services and identify the data interchanges between the various departments. The latter should result in defining the XML Schemas for the various data interchanges involved.

The XML Schemas should ideally be a common definition, with organization-wide scope, if possible, maintained in a common Schema Repository accessible by all applications in the organization. This exercise will also provide an opportunity to analyze the business processes and re-engineer the processes and data interchanges for cleaner definition and separation.

XML is the language that links new tools to legacy applications. It ties businesses to businesses and describes business processes—both internal processes and processes that span the extended enterprise. Also, it describes business relationships and automates contract negotiations, using ebXML.

At service level, the arguments and the returns are to be modeled as XML Messages. This means that the data model should be defined by its XML Schema. The service interface should also be described in an XML document—using the Web Services Definition Language (WSDL).

Processes in BPEL

The services defined have to be used to provide the business functionality needed in the SOA environment. As we discussed in this chapter, there are a few different approaches to using them. Of these, defining the business processes in a standard language such as BPEL should be preferred. Custom integration applications should be avoided as they are more programmatically involved, and would therefore go against the grain of SOA—where the business processes can easily be reconfigured, with changing business needs.

New Applications—Prepare for SOA/POA

Once SOA is adopted in the enterprise, and an infrastructure is available for SOA, any new application coming up for solving any localized departmental problems must from its conception be prepared for SOA. This will involve ensuring that there are well defined façade "services" and also getting these services "wired" to the SOA platform—proactively, even if there is no immediate need for these services from any other solution or business process.

Taking a step further, the services so defined could be used even for the local departmental solution being built. The Services could form the mid-tier providing the functional processing. The UI for the application could use these services in a looser tier separation.

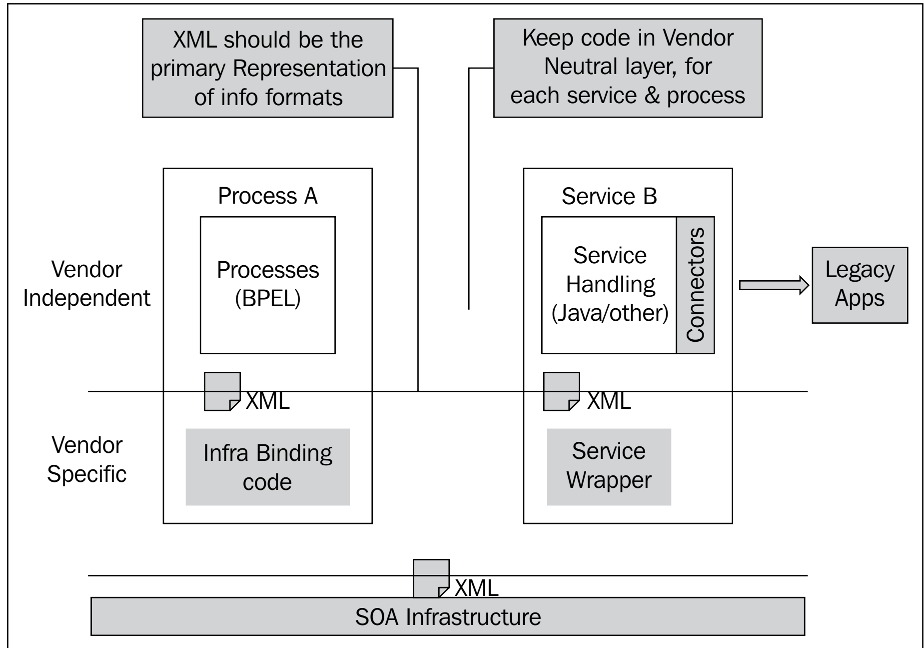
Design for Infrastructure (Vendor) Independence

The services and processes will inevitably have a binding to the specific infrastructure chosen. Even so, care could be taken to minimize the reliance on this specific infrastructure platform.

Some of the trends underway that come into play here (shown in the following figure of SOA infrastructure) are:

- Web Services standards are gaining a lot of traction.
- The standards address areas ranging from describing services to the wire protocol for service invocation, message formats, service containers, process definition languages and much more.
- Most leading vendors in the SOA infrastructure space are supporting the emerging Web Services standards.

When designing the SOA solutions in the enterprise, care must be taken to ensure that as much as possible of the service and processes code is in vendor independent layers—with simple stitch or wrapper classes that can "attach" this vendor-neutral code to the vendor-specific infrastructure. This is important to ensure portability of the SOA solutions being built in the organization.



XML should be the primary means of representing information. XML being ubiquitous, this further aids the vendor independence. All data interchanges should be designed as XML. All transformation of data should be defined using XSLT or XQuery.

Transition to Process-Oriented Architectures

We have discussed SOA so far. Processes exist in an SOA as artifacts that provide the required business functionality by orchestrating the services in the SOA. We can understand services, processes, and the relation between the two. Now, what is Process-Oriented Architecture (POA)?

Let's just chew on this a little: SOA is about the basic business applications layer. POA is about the business-processes layer that uses the application layer underneath. The layers here are:

- Business processes (supply chain PO processing)
- Applications (manufacturing or order processing solutions)
- Application platforms (JAVA EE, .NET, etc.)
- Databases
- Operating system
- Hardware
- Network
- Internet

SOA is about individual applications and solutions in the IT infrastructure looking outward, thinking about providing access to this system from other systems. It is also about facilitating this access via designing simple well defined business interfaces/methods and integrating them with an SOA framework—be it a self-contained tightly integrated infrastructure (EJB/CORBA) or broader Web Services or even any of the more current ESB implementations.

Process Orientation is yet another solution design paradigm. POA is centered on the processes that "use" the services while Service Orientation is about "providing" the services. Process Orientation is about thinking of business requirements as "processes", which are implemented by "orchestrating" the services to be provided by the individual applications serving as service providers.

Process orientation extends the prevailing standard BPM (Business Process Management) model. In the latter, the focus is on point solutions, and in the former it will be more a top-down complete solutions approach. In a typical architecture framework Business Processes are a layer above existing applications and the services are implemented in the applications. The paradigm transition to POA is the same process approach as before but with processes implemented first and then service components.

Services and Processes Coexist—But Services First

With both SOA and POA having Processes and Services, things get a bit murky. A quick search on the Internet will reveal that there is hardly any discussion/product/technology on SOA that does not discuss processes! Every SOA infrastructure product today includes capabilities to "compose" the processes! This is not surprising as just having "services" is not enough if they cannot be used. And given that we are talking about a macro-application layer above existing applications, this layer has to be simple application logic—simpler than the underlying heavyweights. Say, the MRP application or the CRM or the OrderProcessingSystems will be inherently much more complex than a business process that uses services across all the three of these apps.

If SOA already includes processes in its semantic model, how is POA different from SOA? Is this just another term? Is its emphasis needed even?

Looking into it a bit further, it is more about drawing a distinction between SOA being the single system's design paradigm (in ensuring there are services), and POA being about designing macro business applications using simple processes that access the services provided by the individual systems. In other words, SOA is about the individual business systems/solutions in the organization. POA is about a design approach to "integrating" these applications. SOA focuses is on Services. POA focuses on Processes—via well considered process-oriented design of that layer of macro solutions.

The technologies referred to in both cases are similar—Web Service specifications, BPM standards, reliable communications, security, etc. This is not surprising, as the processes do access the services. One does not exist without the other. In a SOA system, one has to describe how the services provided by the SOA system will be "accessed" by processes. In a POA system, one has to describe how the services required by the process will be "provided".

Process—Orchestration of Services

As we discussed earlier in this chapter, a Process is essentially orchestration of the services—stringing together a set of services to provide a business function. The business processes that use the services are one of the most important usage scenarios of the services in an SOA environment.

The orchestration could be done in any custom programming environment such as Java, or done in process-oriented languages such as BPEL. Services being coarse grained and the processing generally requiring supporting business processing, the logic in these processes would also be fairly simple. Consequently, most process language constructs are also fairly simple, unlike the lower-level languages such as C, C#, or Java. Even though there may be other forms of using a service, such as writing a custom integration application, the sheer simplicity of the process languages makes them the easiest form of using a service.

This simplicity of the interfaces of services and the logic in the processing makes the processes easy to design as simple functional flow diagrams involving service callouts and simple flow control.

POA—Shifting the Focus to "Processes First"

I came across the term POA for the first time in an article written by the chief analyst at BPMG.org. However, the justification in that piece was that as a process by itself can be a service in an SOA, the architecture can also be called Process-Oriented Architecture. What is the dominant concept here—Process or Service? Processes use services? Or, process itself is a service?

First off, it must be understood that both SOA and POA solve the same enterprise integration problem, and are not fundamentally very different. Both environments have services and processes that use the services. The difference, if any, is essentially in what is more important in the domain—will one start with processes or with services, when designing a new solution.

POA can be seen as a layer on top of SOA. POA is about the automated business processes that can be closely modeled around the actual business processing in organizations. Processes can be easily designed and implemented by business analysts themselves—without needing dedicated programmers for implementing processes. The process languages are all simple with powerful visual design tools available for modeling the processes graphically—say, not having to deal with the likes of the XML structure of the BPEL documents.

Process Orientation starts where Service Orientation leaves off. Service Orientation would ensure that all the IT systems and solutions in the organization are available on the SOA platform chosen for the enterprise. Once services are available, then the business analysts can start building the processes that use these low-level services provided by the various departmental applications. Often, business process advocates talk about the three layers in IT infrastructure as being the Data-oriented layer (the DBMSs), the Service-Oriented layer, and the Process-Oriented layer. Data is well abstracted in enterprises these days. All solutions and legacy applications are abstracted by the SOA. So, Processes in the POA layer will have simple common abstractions to deal with—the services provided by SOA.

One point that you should note is that there are no clearly defined or accepted definitions that articulate the difference between POA and SOA. BPMN.org pushes the term Process-Oriented Architectures extensively, making the distinction that SOA is about services alone, and POA provides the business processes layer. SOA infrastructure providers assume that processes and orchestration of services are an inherent part of the SOA platform.

Concepts and Principles of Process-Oriented Architectures

POA is, like SOA, more a software design approach than an infrastructure. Software architectures that have processes at their core would qualify as Process-Oriented architectures.

Enterprise IT organizations will adopt SOA for integrating the various IT investments already present in the organization. Here, the focus is the "services" provided by the existing systems. Once SOA is in place and services available, the services will be used to compose business processes around them. While in the end there will be services and processes in any SOA-enabled organization, the evolution will be that services come up first and then processes are built. The focus here is more to integrate the enterprise. Integrate the existing legacy systems and solutions, to form a homogenous IT fabric in the organization. Towards this goal, getting services from all the systems into one common services framework will be the focus—at least initially. Using these in business processes comes later.

POA—Processes First. Services... Maybe!

In a POA environment, processes are at the core. Processes are defined first, and then the required services implemented. As a process is a service, processes themselves could be steps in other processes. A whole application architecture that is centered on processes is possible in POA environments, so much so that services by themselves may even be rendered redundant and unnecessary for some applications.

As with any programming environment where a function/method can be called from other functions/methods, processes are also iterative. A process can call services. A process can call other processes.

In SOA, the term service is used primarily to emphasize the point that one of the main goals of an SOA is to abstract the software that provides these services. However, because the services of an SOA are indeed processes, this blurs the lines between business processes and services.

Process as a service is an inverted view. SOA is clean—where each service in the architecture provides some "service" that can be called out from anywhere else in the system, say, from a "process". The service internally "executes" some "processing". Now this begs the question, "Is OrderProcessing() a service, or a process?". In conventional terms, this is a "service" (or a method). "Process" has come to assume the meaning that it is a string of steps (services) performed to realize some business functional processing. Surely this is kind of a catch-22. Is a process a string of services? Or, is the service itself a process.? Can a service internally execute a bunch of services? Even worse, can a service internally include a series of processes?

POA Enables Top-down Design—Using just Processes

The POA design paradigm is entirely about processes. Systems are designed ground-up around processes. Start with high-level business processes, identify the required sub-processes and services, and implement the lower layers as clarity emerges.

POA enables building business applications top-down, starting from the high-level business processes and then detailing the sub-processes and all the way down to the services and database access at the lowest-level of the solution. The services may exist or in neo-application architectures, there may not be any of the services other than data services. In self-contained new business solutions, one can think of a complete business solution implemented just using processes, and possibly a database. In any enterprise integration environment though, one should expect access to legacy solutions to be from the business processes.

Analysts Become Programmers

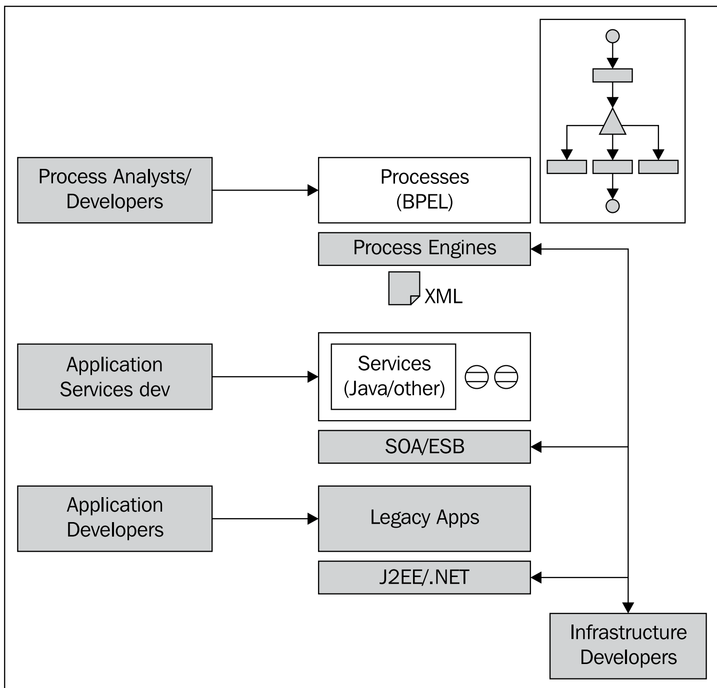
Once we accept the notion that a process itself can be a service, and used in other processes, conceivably one could think of a complete application that is written just as processes. Looking at how business solutions are designed, there are always business analysts that understand the business requirements and functioning; based on which the analyst articulates the software requirements. These requirements then form the inputs to the developers that will go and design and build the solution. Having platforms such as the application platforms does ease the software complexity by offloading a lot of the processing to the platform.

POA further enhances the platform, with the platform now being capable of running services and processes. The software for this platform will include a lot of processes. These processes will not be too different from the requirements gathered by the analyst. Extending the analyst's role a bit, the analyst can now use process design tools and construct the business process needed without requiring developers. In effect, the business processes are developed by the analyst himself or herself. This, of course, presupposes that there are intuitive graphical process development tools.

Extending the model further, one could imagine applications that are composed of just processes.

POA Changing Software Development Roles

Before the onset of frameworks and application platforms, software consisted of complex programs handling everything from UI and business logic to communication and data storage. With DBMS, the roles got separated into database programmers/designers and application developers. With the arrival of application platforms such as JAVA EE and .NET, there was clear separation of platform developers that provide the infrastructure, application developers that build the JAVA EE applications, and possibly UI developers that build the front-end user interfaces. With SOA, a variant of the application developers is now in place to build the services and service wrappers abstracting the connectivity to legacy solutions.



POA uses all the abstractions of SOA and adds a new layer called Process Designers. As shown in the previous figure of developer roles in POA, the Business Analysts themselves perform the function of the process developers. Given this, in POA-enabled enterprises, the development roles will now be Process Designers, Application Service Developers and Application Developers (for legacy solutions). These roles are well supported by the infrastructure developers—either in-house or from the infrastructure platform vendors that build the infrastructure. The Application Services Developers will also need to manage the XML artifacts in the environment, which includes the XML Schema development and the XML transformations like XSLT/XQuery.

Infrastructure developers manage the lower-level application platforms on which services may be running—such as JAVA EE or ESB.

In SOA, all the above roles exist. With existing applications being a starting point for SOA, the presence of application developers to develop and maintain these applications is needed. Service developers will be required to wire these applications into the common SOA framework. Process Analysts will be required to build the Integration Business processes. However, in POA, it is possible that there are more Processes than there are Services or even Applications. In well designed POA applications, processes themselves may play the role of services (as sub-processes), and applications may not exist at all in well defined POA environments.

Process Standards

Standards in the Business Processes space are being driven by a few organizations including BPMI.org, OASIS, and W3C. There is a braod-based vendor support for these standards. By and large, a few standards such as BPEL and design notations such as BPMN developed by BPMI are gaining broader support.

Some of the main standards are:

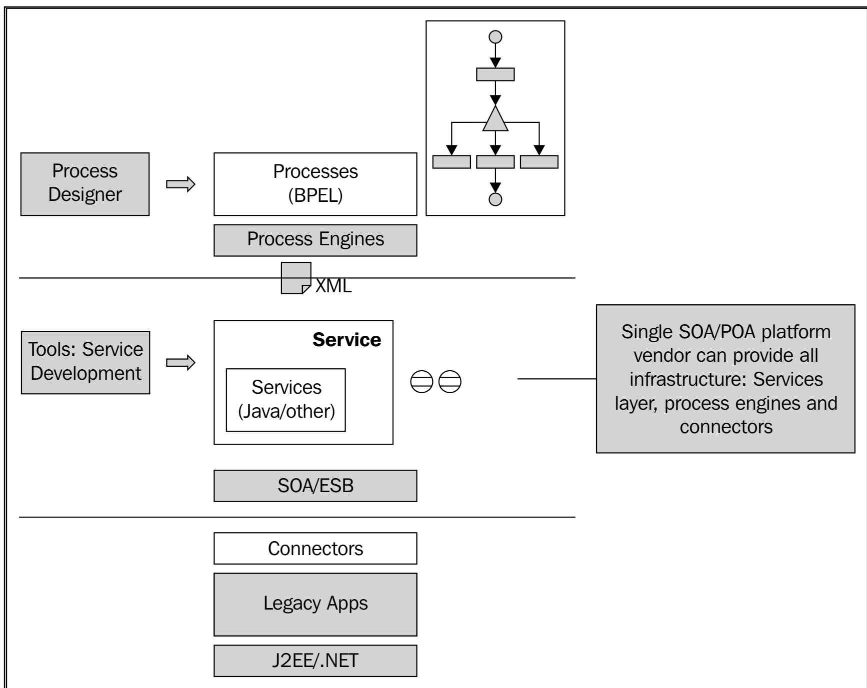
- BPEL4WS: Just one of a number of emergent standards in the general area of Business Process Management, also known as BPM
- WSCI: Web Services Choreography Interface
- BPML: Business Process Modelling Language
- XPDL: XML Processing Description Language
- BPSS: Business Process Specification Schema

The Business Process Management Initiative (BPMI.org) touts BPML, with backing from BEA Systems (who also support BPEL4WS), Sun Microsystems, and SAP, among others. The Workflow Management Coalition (WfMC) offers XPDL. OASIS with support from UN is pushing BPSS, which is part of the larger Electronic Business XML (ebXML) initiative run by OASIS. For simple processes, of the competing standards, BPML and BPEL4WS are the leading standards.

Both BPML and BPEL4WS address the same space, with the familiar SOAP, WSDL, and UDDI standards at their core—where BPEL4WS uses the (simultaneously announced) WS-Coordination and WS-Transaction standards to provide support for simple and extended transactions through Web Services, BPMI endorses the WSCI standard as its Web Services underpinning.

Infrastructure for Process-Oriented Architectures

Infrastructure for POA is not too different from the SOA infrastructure we discussed earlier in this chapter. The key elements are all the same—providing the infrastructure to host processes and run the Services, in a secure manner on and high-performing communication backbone. (See the following figure about POA infrastructure layers.)



Summary

Service-Oriented Architecture is itself very nascent, still in the early stages of serious enterprise adoption. As we discussed in this chapter, SOA is a very comprehensive enterprise-integration paradigm that builds on many existing concepts. The Web Services standards provide a strong foundation for SOA infrastructure. Enterprise Services Bus is presently one of the leading integrated infrastructure options. With traction increasing in enterprises, more concrete infrastructure options will evolve. More standard development models will emerge. And along with them newer application development models on the lines of Process-Oriented Architecture may emerge.

Later in this book, we will discuss the Web Service standards and how XML is at their core leading on to the Enterprise Services Bus.

3

Best Practices for Using XML for Integration

Introduction

When designing XML documents for integration, there are number of considerations to be aware of. Before designing the XML schema, it is necessary to understand the domain. You also need to consider how your application intends to receive and send documents, and how and when to go about validating those documents. It is also important to separate the XML document processing from the application's business logic processing.

The standard schemas for several vertical domains are easily available. Your first preference while designing a schema for your application is to look for a standard vertical schema. If a standard schema does not fully meet your requirements, you should consider extending a standard schema to meet your application requirements. Whether you design your own domain-specific schema or rely on standard vertical schemas, you still must understand the dynamics of mapping the application's data model to the schema. You also need to consider the processing model, and whether to use a document-centric model or an object-centric model.

Domain-Specific XML Schemas

Despite the availability of more and more vertical domain schemas, application developers still may have to define application-specific XML schemas that must be agreed upon, and shared between interoperating participants. With the introduction of modern schema languages such as XSD, which introduced strong data typing and type derivation, XML schema design shares many of the aspects of object-oriented design especially with respect to modularization and reuse.

The design of domain-specific XML schemas breaks down according to the definition of XML schema types, their relationship to other types, and any constraints to which they are subjected. The definitions of such XML schema types, relationships, and constraints are typically the result of the analysis of the application domain vocabulary (also called the business vocabulary). As much as possible, schema designers should leverage already-defined public vertical domain schema definitions to promote greater acceptance and interoperability among intended participants. The designers of new schemas should keep interoperability concerns in mind and try to allow for reuse and extensibility.

Generally, document schema design and the layout of document instances closely parallel object-oriented design. In addition, design strategies exist that identify and provide well-defined solutions to commonly recurring problems in document schema design.

Keep the following recommendations in mind when designing an XML schema:

- Adopt and develop design techniques, naming conventions, and other best practices similar to those used in object-oriented modeling to address the issues of reuse, modularization, and extensibility. Some of the common design techniques are discussed in the later sections.
- Leverage existing horizontal schemas, and vertical schemas defined within your industry, as well as the custom schemas already defined within your enterprise. You can check out `http://www.xml.org` for information on design guidelines specific to your industry like healthcare, defense, and insurance.
- Do not solely rely on self-describing element and attribute names. Comment and document custom schemas.
- Use modeling tools that support well-known schema languages such as XSD. Some of the tools are Netbeans 5.5 Enterprise Pack, Eclipse, and Visual Studio.NET. You may find many open-source tools listed at `http://xmlbeans.apache.org/docs/2.0.0/guide/tools.html`.

Keep in mind that reusing schemas may enable the reuse of the corresponding XML processing code.

Sending and Receiving XML Documents

XML schemas of documents to be consumed and produced are part of the overall exposed interface of an XML-based application. The exposed interface encompasses schemas of all documents passed along with incoming and outgoing messages regardless of the message-passing protocol—SOAP, plain HTTP, or message queuing.

Typically, an application may receive or return XML documents as follows:

- Received through a web service endpoint
- Returned to a web service client
- Through a message queue or topic when implementing a business process workflow or implementing an asynchronous web service architecture
- Exchanging XML documents through a plain-socket connection

Note that a generic XML-based application can additionally receive and return XML documents through a servlet over plain HTTP.

There are circumstances when a web service may internally exchange XML documents through a message queue or topic. When implementing an asynchronous architecture, the interaction layer of a web service may send XML documents asynchronously using messages to the processing layer. Similarly, when a web service implements a workflow, the components implementing the individual stages of the workflow may exchange XML documents using messages. From a developer's point of view, receiving or sending XML documents through a message queue or topic is similar in principle to the case of passing documents as SOAP message attachments. XML documents can be passed through a message queue or topic as text messages or in a serialized form.

Validating XML Documents

Once a document has been received or produced, a developer may—and most of the time must—validate the document against the schema to which it is supposed to conform. Validation, an important step in XML document handling, may be required to guarantee the reliability of an XML application. An application may legitimately rely on the parser to do the validation and thus avoid performing such validation itself.

However, a valid XML document may still be invalid in the application's domain. This might happen, for example, when a document is validated using DTD, because this schema language lacks capabilities to express strongly-typed data, complex unicity, and cross-reference constraints. Other modern schema languages, such as XSD, more rigorously—while still lacking some business constraint expressiveness—narrow the set of valid document instances to those that the business logic can effectively process. Regardless of the schema language, even when performing XML validation, the application is responsible for enforcing any uncovered domain-specific constraints that the document may nevertheless violate. That is, the application may have to perform its own business logic-specific validation in addition to the XML validation.

To decide where and when to validate documents, you may take into account certain considerations. Assuming a system—by system we mean a set of applications that compose a solution and that define a boundary within which trusted components can exchange information—one can enforce validation according to the following observations:

- Documents exchanged within the components of the system may not require validation.
- Documents coming from outside the system, especially when they do not originate from external trusted sources, must be validated on entry.
- Documents coming from outside the system, once validated, may be exchanged freely between internal components without further validation.

For example, a multitier e-business application that exchanges documents with trading partners through a front end enforces document validity at the front end. Not only does it check the validity of the document against its schema, but the application also ensures that the document type is a schema type that it can accept. It then may route documents to other applications or servers so that the proper services can handle them. Since they have already been validated, the documents do not require further validation. In a web service, validation of incoming documents is typically performed in the interaction layer. Therefore, the processing layer may not have to validate documents it receives from the interaction layer.

Some applications may have to receive documents that conform to different schemas or different versions of a schema. In these cases, the application cannot do the validation up front against a specific schema unless the application is given a directive within the request itself about which schema to use. If no directive is included in the request, then the application has to rely on a hint provided by the document itself. Note that to deal with successive versioning of the same schema—where the versions actually modify the overall application's interface—it sometimes may be more convenient for an application to expose a separate endpoint for each version of the schema.

To illustrate, an application must check that the document is validated against the expected schema, which is not necessarily the one to which the document declares it conforms. With DTD schemas, this checking can be done only after validation. When using DOM, the application may retrieve the system or public identifier (`SystemID` or `PublicID`) of the DTD to ensure it is the identifier of the schema expected, while when using SAX, it can be done on the fly by handling the proper event.

When relying on the schemas to which documents internally declare they are conforming (through a DTD declaration or an XSD hint), for security, and to avoid external malicious modification, you should keep your own copy of the schemas and validate against these copies. This can be done using an entity resolver, which is an interface from the SAX API (`org.xml.sax.EntityResolver`) that forcefully maps references to well-known external schemas to secured copies. An alternative to this would be to use Schematron—a simple and powerful Structural Schema Language that is not based on grammars but on finding tree patterns in the parsed document.

Mapping Schemas

After defining the application interface, and the schemas of the documents to be consumed and produced, the developer has to define how the document schemas relate or map to the data model on which the application applies its business logic. We refer to these document schemas as external schemas. These schemas may be specifically designed to meet the application's requirements, such as when no preexisting schemas are available, or they may be imposed on the developer. The latter situation, for example, may occur when the application intends to be part of an interacting group within an industry promoting standard vertical schemas (for example, UBL or ebXML schemas).

Choosing Processing Models

An XML-based application may either apply its business logic directly on consumed or produced documents, or it may apply its logic on domain-specific objects that completely or partially encapsulate the content of such documents. Domain-specific objects are Java objects that may not only encapsulate application domain-specific data, but may also embody application domain-specific behavior.

An application's business logic may directly handle documents it consumes or produces, which is called a document-centric processing model, if the logic:

- Relies on both document content and structure
- Is required to punctually modify incoming documents while preserving most of their original form, including comments, external entity references, and so forth

In a document-centric processing model, the document processing may be entangled with the business logic and may therefore introduce strong dependencies between the business logic and the schemas of the consumed and produced documents. Moreover, the document-centric processing model does not promote a clean separation between business and XML programming skills, especially when an application developer who is more focused on the implementation of the business logic must additionally master one or several of the XML processing APIs.

There are cases that require a document-centric processing model, such as:

- The schema of the processed documents is only partially known and therefore cannot be completely bound to domain-specific objects; the application edits only the known part of the documents before forwarding them for further processing.
- Because the schemas of the processed documents may vary or change greatly, it is not possible to hard-code or generate the binding of the documents to domain-specific objects; a more flexible solution is required, such as one using DOM with XPath.

A typical document-centric example is an application that implements a data-driven workflow: Each stage of the workflow processes only specific information from the incoming document contents, and there is no central representation of the content of the incoming documents. A stage of the workflow may receive a document from an earlier stage, extract information from the document, apply some business logic on the extracted information, and potentially modify the document before sending it to the next stage.

Generally, it is best to have the application's business logic directly handle documents only in exceptional situations, and to do so with great care. You should instead consider applying the application's business logic on domain-specific objects that completely or partially encapsulate the content of consumed or produced documents. This helps to isolate the business logic from the details of XML processing.

A pure object-centric processing model requires XML-related issues to be kept at the periphery of an application—that is, in the web service interaction layer closest to the service endpoint, or, for more classical applications, in the web tier. In this case, XML serves only as an additional presentation medium for the application's inputs and outputs.

Note that the object - and document-centric processing models may not be exclusive of one another. An application may be globally document-centric and exchange documents between its components, and some components may themselves locally process part of the documents using an object-centric processing model. Each component may use the most suitable processing model for performing its function. For example, consider an application in the automobile industry. An automobile manufacturing company may interact with its customers using a B2C model and at the same time integrate with its business partners using a B2B model. The customer interaction in this case may be document-centric, while the partner interaction that can be more tightly coupled could be object-centric.

Fragmenting Incoming XML Documents

When your service's business logic operates on the contents of an incoming XML document, it is a good idea to break XML documents into logical fragments when appropriate. The processing logic receives an XML document containing all information for processing a request. However, the XML document usually has well-defined segments for different entities, and each segment contains the details about a specific entity.

Fragmenting an incoming document can be viewed as a centralized implementation of the flexible mapping design. Fragmenting an incoming document, by suppressing redundant parsing of the incoming document and limiting the exchanges between stages to the strictly relevant data, improves performance over a straightforward implementation of flexible mapping. However, it loses some flexibility because the workflow dispatching logic is required to specifically know about (and therefore depend on) the document fragments and formats expected by the different stages.

Fragmenting a document has the following benefits:

- It avoids extra processing and exchange of superfluous information throughout the workflow.
- It maximizes privacy because it limits sending sensitive information through the workflow.
- It centralizes some of the XML processing tasks of the workflow and therefore simplifies the overall implementation of the workflow.
- It provides greater flexibility to workflow error handling as each stage handles only business logic-related errors while the workflow dispatching logic handles document parsing and validation errors.

Design Recommendations

When you design an XML-based application, specifically one that is a web service, you must make certain decisions concerning the processing of the content of incoming XML documents. Essentially, you decide the "how, where, and what" of the processing: You decide the technology to use for this process, where to perform the processing, and the form of the content of the processing.

In summary, keep in mind the following recommendations:

- When designing application-specific schemas, promote reuse, modularization, and extensibility, and leverage existing vertical and horizontal schemas.

- When implementing a pure object-centric processing model, keep XML on the boundary of your system as much as possible—that is, in the web service interaction layer closest to the service endpoint, or, for more classical applications, in the presentation layer. Map document content to domain-specific objects as soon as possible.
- When implementing a document-centric processing model, consider using the flexible mapping technique. This technique allows the different components of your application to handle XML in a way that is most appropriate for each of them.
- Strongly consider validation at system entry points of your processing model—specifically, validation of input documents where the source is not trusted.
- When consuming or producing documents, as much as possible express your documents in terms of abstract `Source` and `Result` objects that are independent from the actual XML-processing API you are using.
- Consider a "meet-in-the-middle" mapping design strategy when you want to decouple the application data model from the external schema that you want to support.
- Abstract XML processing from the business logic processing using the XML document editor design strategy. This promotes separation of skills and independence from the actual API used.

Tips for Designing XML Schemas

While designing an XML Schema, make sure that you design based on your system-specific requirements. There is more than one way of designing a schema and they all might fit your architecture. Even before you start designing your schemas, you should have the design objectives captured and documented for further reference.

Enumerating the design objectives is very important prior to designing your schema because you may or may not be able to refine your schema based on the inequity you discover later. The following sections discuss some of the common dilemmas of Schema design.

Default Namespace—targetNamespace or XMLSchema?

When designing a schema, you have three options pertaining to namespace. They are:

1. Making `XMLSchema` (`1.http://www.w3.org/2001/XMLSchema`) as the default namespace.
2. Making the `targetNamespace` (user defined) as the default namespace.
3. Not specifying the default namespace at all.

Which one of the above options is the best practice for designing your schema?

The XML Schema you design should have at least two namespaces (except for no-namespace schemas). They are:

1. `XMLSchema` namespace (`1.http://www.w3.org/2001/XMLSchema`*)*
2. `targetNamespace`

The possible case for designing your schema is restricted to the below mentioned options:

Case 1: Make `XMLSchema` the default namespace, and explicitly qualify all references to components in the `targetNamespace`.

The snippet below is from a sample purchase order schema, which explains the Case 1 usage.

```
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
             targetNamespace="http://www.porder.org"
             xmlns:po=" http://www.porder.org"
             elementFormDefault="qualified">
   <include schemaLocation="po.xsd"/>
   <element name="purchaseOrder">
       <complexType>
          <sequence>
              <element name="shipTo">
                 <complexType>
                     <sequence>
                        <element ref="po:Address"
                           maxOccurs="unbounded"/>
                     </sequence>
                 </complexType>
              </element>
          </sequence>
       </complexType>
   </element>
</schema>
```

This schema includes the purchase order (`po.xsd`) schema, which contains the declaration of the `Address` element. The `Address` element is defined as follows:

```
<xs:complexType name="Address">
    <xs:sequence>
       <xs:element name="name"   type="xs:string"/>
       <xs:element name="street" type="xs:string"/>
       <xs:element name="city"   type="xs:string"/>
       <xs:element name="state"  type="xs:string"/>
       <xs:element name="zip"    type="xs:decimal"/>
    </xs:sequence>
    <xs:attribute name="country" type="xs:NMTOKEN"
     fixed="US"/>
</xs:complexType>
```

The schema shown in the first code snippet refers to the above schema's `Address` complex type.

Note that in this case, `XMLSchema` is the default namespace. That is the reason why we don't have namespace qualifiers for our components like `element`, `complexType`, and `sequence`.

From first code snippet, it is evident that the namespace prefix, '**po**' is associated with the `targetNamespace`. And any reference to the components in the targetNamespace is explicitly qualified by '**po'**.

Case 2: Make the `targetNamespace` the default namespace, and explicitly qualify all components from the `XMLSchema` namespace.

The snippet below is from a sample purchase order schema, which explains the current usage.

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             targetNamespace="http://www.porder.org"
             xmlns=» http://www.porder.org»
             elementFormDefault=»qualified»>
   <xsd:include schemaLocation=»po.xsd»/>
   <xsd:element name=»purchaseOrder">
        <xsd:complexType>
           <xsd:sequence>
              <xsd:element name="shipTo">
                 <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element ref="Address"
                            maxOccurs="unbounded"/>
```

```
                    </xsd:sequence>
                </xsd:ComplexType>
            </xsd:element>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
</xsd:schema>
```

In the above snippet `xsd` points to `XMLSchema` namespace and the default namespace is `targetNamespace` and hence a namespace qualifier is not required for the 'Address' as Case 1.

Case 3: No Default Namespace—qualify both `XMLSchema` and `targetNamespace`.

The snippet below is from a sample purchase order schema, which explains the current usage.

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             targetNamespace="http://www.porder.org"
             xmlns:po=" http://www.porder.org"
             elementFormDefault="qualified">
   <xsd:include schemaLocation="po.xsd"/>
   <xsd:element name="purchaseOrder">
        <xsd:complexType>
           <xsd:sequence>
              <xsd:element name="shipTo">
                 <xsd:complexType>
                    <xsd:sequence>
                       <xsd:element ref="po:Address"
                          maxOccurs="unbounded"/>
                    </xsd:sequence>
                 </cxsd:omplexType>
             </xsd:element>
          </xsd:sequence>
       </xsd:complexType>
   </xsd:element>
</xsd:schema>
```

In the above snippet, `xsd` points to the `XMLSchema` namespace and `po` points to the `targetNamespace`. The `XMLSchema` elements are fully qualified along with reference to 'Address'.

The following table discusses the pros and cons of the cases discussed in the earlier sections.

Case	Pros	Cons
1	If your schema is referencing components from multiple namespaces then this approach gives a consistent way of referring to the components (i.e., you always qualify the reference).	Schemas that have no targetNamespace must be designed so that the XMLSchema components (element, complexType, sequence, etc.) are qualified. If you adopt this approach to designing your schemas then in some of your schemas you will qualify the XMLSchema components and in other schemas you won't qualify the XMLSchema components. Changing from one way of designing your schemas to another way can be confusing.
2	Schemas that have no targetNamespace must be designed so that the XMLSchema components (element, complexType, sequence, etc.) are qualified. This approach will work whether your schema has a targetNamespace or not. Thus, with this approach you have a consistent approach to designing your schemas —always namespace-qualify the XMLSchema components.	If your schema is referencing components from multiple namespaces then for some references you will namespace-qualify the reference, whereas other times you will not (i.e., when you are referencing components in the targetNamespace). This variable use of namespace qualifiers in referencing components can be confusing.
3	1. Schemas that have no targetNamespace must be designed so that the XMLSchema components (element, complexType, sequence, etc.) are qualified. With this approach, all your schemas are designed in a consistent fashion. 2. If your schema is referencing components from multiple namespaces then this approach gives a consistent way of referencing components (i.e., you always qualify the reference).	Very cluttered: being very explicit by namespace qualifying all components and all references can be annoying when reading the schema.

This is a human readability issue as it does not affect the semantics of the XML schema at all. The cleanest and simplest (but not the most compact) approach is Case 2.

If your schema has a suggested prefix (like `wsdl:` for the WSDL namespace) then use that prefix otherwise use "`tns`" which stands for Target Name Space.

Localize Namespace vs. Expose Namespaces

The namespaces of the elements and the attributes that a schema defines may be localized within the schema itself. The alternative to this would be to expose the namespaces in instance documents. We will compare and contrast the two approaches.

A typical schema will reuse elements and types from multiple schemas, each with different namespaces. A schema, then, may comprise components from multiple namespaces. Thus, when a schema is designed the schema designer must decide whether or not the origin (namespace) of each element should be exposed in the instance documents.

A binary switch attribute in the schema is used to control the hiding/exposure of namespaces: by setting `elementFormDefault="unqualified"` the namespaces will be hidden (localized) within the schema, and by setting `elementFormDefault="qualified"` the namespaces will be exposed in instance documents.

All Schemas must have a consistent value for `elementFormDefault`.

Be sure to note that `elementFormDefault` applies just to the schema that it is in. It does not apply to schemas that it includes or imports. Consequently, if you want to hide namespaces then **all** schemas involved must have set `elementFormDefault="unqualified"`. Likewise, if you want to expose namespaces then all schemas involved must have set `elementFormDefault="qualified"`.

Requirements for Localizing the Schema

There are two requirements on an element for its namespace to be localized (hidden) from instance documents:

1. The value of `elementFormDefault` must be "unqualified".
2. The element must not be globally declared. For example:

```
<?xml version="1.0"?>
    <xsd:schema ...>
       <xsd:element name="book">
       ...
    </xsd:schema ...>
```

The element `book` can never have its namespace hidden from instance documents, regardless of the value of `elementFormDefault`.

`book` is a *global* element (i.e., an immediate child of `<schema>`) and therefore must always be qualified. To enable namespace hiding, the element must be a local element.

> Whenever you create a schema, make two copies of it. The copies should be identical, except that in one copy you set `elementFormDefault="qualified"`, whereas in the other copy you set `elementFormDefault="unqualified"`. If you make two versions of all your schemas then people who use your schemas will be able to implement either design approach—hide (localize) namespaces, or expose namespaces. Note that exposing namespaces helps in avoiding name collisions.

Advantages of Localizing Component Namespaces within the Schema

The instance document is simple. It's easy to read and understand. There are no namespace qualifiers cluttering up the document, except for the one on the document element. The knowledge of where the schema got its components is irrelevant and localized to the schema.

Advantages of Exposing Namespaces in Instance Documents

If your company spends the time and money to create a reusable schema component, and makes it available to the marketplace, then you will most likely want recognition for that component.

Another case where it is desirable to expose namespaces, is when processing instance documents. Oftentimes, when processing instance documents the namespace is required to determine how an element is to be processed (e.g., "if the element comes from this namespace then we'll process it in this fashion, if it comes from this other namespace then we'll process it in a different fashion"). If the namespaces are hidden, then your application is forced to do a lookup in the schema for every element. This will be unacceptably slow.

Global vs. Local Declaration

A component (`element`, `complexType`, or `simpleType`) is "global" if it is an immediate child of `<schema>`, whereas it is "local" if it is not an immediate child of `<schema>`, i.e., it is nested within another component.

Below is a snippet of an XML instance document. We will explore the different design strategies using this example.

```
<Book>
   <Title>SOA and POA</Title>
   <Publisher>Packt</Publisher >
</Book>
```

Russian Doll and Salami Slice Designs

The Russian Doll design approach has a schema structure mirroring the instance document structure, e.g., declare a `Book` element and within it declare a `Title` element followed by a `Publisher` element:

```
<xsd:element name="Book">
   <xsd:complexType>
       <xsd:sequence>
          <xsd:element name="Title" type="xsd:string"/>
          <xsd:element name="Publisher" type="xsd:string"/>
        </xsd:sequence>
   </xsd:complexType>
</element>
```

The instance document has all its components bundled together. Likewise, the schema is designed to bundle together all its element declarations.

In Salami Slice design, we disassemble the instance document into its individual components. In the schema, we define each component (as an element declaration), and then assemble them together:

```
<xsd:element name="Title" type="xsd:string"/>
   <xsd:element name="Publisher" type="xsd:string"/>
   <xsd:element name="Book">
       <xsd:complexType>
           <xsd:sequence>
              <xsd:element ref="Title"/>
              <xsd:element ref="Publisher"/>
           </xsd:sequence>
       </xsd:complexType>
   </xsd:element>
```

Note how the schema declared each component individually (`Title`, and `Publisher`) and then assembled them together in the creation of the `Book` component.

Best Practice

Use Salami Slice design when your task requires that you make available to instance document authors the option to use element substitution. When minimizing size and coupling of components is of utmost concern then use the Russian Doll design.

Element vs. Type

When should an item be declared as an element versus when should it be defined as a type? XML Schema provides two constructs for tagging data in XML instances elements and types; which construct to employ is not always clear. While data can often be stored as attributes or elements, there are a number of constraints on attributes that generally impact one's decision.

Consider this example:

Should `Title` be declared as an element?

```
<xsd:element name="Title">
     ...
</xsd:element>
```

or as a type?

```
<xsd:complexType name="Title">
     ...
</xsd:complexType>
```

Best Practice

When in doubt, make it a type. You can always create an element from the type, if needed. With a type, other elements can reuse that type.

Example: If you cannot decide whether to make `Title` an element or a type, then make it a type:

```
<xsd:complexType name="Title">
...
</xsd:complexType>
```

If you decide later that you need a `Title` element, you can create one using the `Title` type:

```
<xsd:element name="Title" type="Title"/>
```

If the item is not intended to be an element in instance documents then define it as a type.

Example: If you will *never* see this in an instance document:

```
<Title>
<>
...
</Title>
```

then define `Title` as a `complexType`. If the item is intended to be used as an element in instance documents and other elements are to be allowed to substitute for the element, then it must be declared as an `element`.

Suppose that we would like to enable instance document authors to use interchangeably the vocabulary `Title` and `BookName`,

```
<xsd:Title>
...
</xsd:Title>
...
<xsd:BookName>
...
</xsd:BookName>
```

To enable this substitutable-tag-name capability, `Title` and `BookName` must be

declared as elements, and made members of a `substitutionGroup`:

```
<xsd:element name="Title">
...
</xsd:element>
<xsd:element name="BookName" substitutionGroup="Title"/>
```

Zero, One, or Many Namespaces

In a project where multiple schemas are created, should we give each schema a different `targetNamespace`, or should we give all the schemas the same `targetNamespace`, or should some of the schemas have no `targetNamespace`?
In a typical project, many schemas will be created. The schema designer is then confronted with this issue: "shall I define one t`argetNamespace` for all the schemas, or shall I create a different `targetNamespace` for each schema, or shall I have some schemas with no `targetNamespace`?"

Here are the three design approaches for dealing with this issue:

1. Heterogeneous Namespace Design: Give each schema a different `targetNamespace`.
2. Homogeneous Namespace Design: Give all schemas the same `targetNamespace`.
3. Chameleon Namespace Design: Give the "main" schema a `targetNamespace` and give no `targetNamespace` to the "supporting" schemas (the no-namespace supporting schemas will take on the `targetNamespace` of the main schema, just like a Chameleon). When a schema uses Chameleon components those components become part of the including Schema's `targetNamespace`, just as though the schema author had typed the element declarations and type definitions inline. If the schema includes multiple no-namespace schemas then there will be a chance of name collisions. In fact, the schema may end up not being able to use some of the no-namespace schemas because their use results in name collisions with other Chameleon components.

Best Practices

When you are reusing schemas that *someone else* created you should `<import>` those schemas, i.e., use the Heterogeneous Namespace design. It is a bad idea to copy those components into our namespace, for two reasons:. soon your local copies would get out of sync with the other schemas, and you lose interoperability with any existing applications that process the other schema's components.

Use the Heterogeneous Namespace Design

1. When there are multiple elements with the same name (avoid name collision).
2. When there is a need to visually identify in instance documents the origin/lineage of each element/attribute. In this design, the components come from different namespaces, so you have the ability to identify in instance documents that "element A comes from schema X".

Use the Homogeneous Namespace Design

1. When all of your schemas are conceptually related.
2. When there is no need to visually identify in instance documents the origin of each element/attribute. In this design, all components come from the same namespace, so you lose the ability to identify in instance documents that "element A comes from schema X".

Use the Chameleon Design

1. With schemas that contain components that have no inherent semantics by themselves.
2. With schemas that contain components that have semantics only in the context of an including schema.
3. When you don't want to hard-code a namespace to a schema, rather you want including schemas to be able to provide their own application-specific namespace to the schema.

Using XSL for Transformation

With the growing popularity of XML as a medium to interact with different systems, more and more organizations are turning to XML to solve their interoperability issues. And with architects trying to achieve a clear demarcation between display and business logic, XSL is gaining importance. XSL is basically an XML document tree conforming to a schema that is applied to a XML tree to produce an output tree.

XSL can be used to define how an XML file should be displayed by transforming the XML file into a format that is recognizable to a browser. One such format is HTML. Normally XSL does this by transforming each XML element into an HTML element.

XSL can also add completely new elements into the output file, or remove elements. It can rearrange and sort the elements, test and make decisions about which elements to display, and a lot more.

For more information on using XSL to transform XML documents, read the XML tutorial at `http://www.w3schools.com/xsl/`.

This section discusses a list of the best practices to be followed when writing XSL stylesheets.

xsl:import and xsl:include

The `xsl:include` instruction is similar to the `include` instruction available in several programming languages. The XSLT processor replaces it with the contents of the stylesheet named in the `href` attribute. For example, the following stylesheet includes the `attach.xsl` stylesheet for transformation processing.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     version="1.0">
  <xsl:include href="attach.xsl"/>
  <xsl:template match="po_name">
```

```
    <p><xsl:apply-templates/></p>
  </xsl:template>
</xsl:stylesheet>
```

If `attach.xsl` looks like this,

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     version="1.0">
  <xsl:template match="emphasis">
   <b><xsl:apply-templates/></b>
  </xsl:template>
</xsl:stylesheet>
```

the XSLT processor will treat:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     version="1.0">
  <xsl:include href="attach.xsl"/>
  <xsl:template match="po_name">
   <p><xsl:apply-templates/></p>
  </xsl:template>
</xsl:stylesheet>
```

as:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     version="1.0">
  <xsl:template match="emphasis">
   <b><xsl:apply-templates/></b>
  </xsl:template>
  <xsl:template match="po_name">
   <p><xsl:apply-templates/></p>
  </xsl:template>
</xsl:stylesheet>
```

The included stylesheet must still be a complete and well formed XML document.

An included stylesheet may in turn include other stylesheets, and they may include other stylesheets. There's no limit to the levels of inclusion that you can use, although the more you do it, the more complexity you have to keep track of.

The `xsl:include` element can be included anywhere you want in a stylesheet, as long as it's a top-level element, that is, a child of the `xsl:stylesheet` element that makes up the main body of the stylesheet.

Using `xsl:include` doesn't change XSLT's approach to multiple template rules that apply to the same node. If the XSLT processor can't find one template that is more specific than another for a particular source tree node, it's an error. Using `xsl:include` does increase the chance of this error happening, especially if you include stylesheets that include other stylesheets, because it's harder to keep track of the full collection of template rules being grouped together.

The `xsl:import` instruction is similar to `xsl:include` except that instructions in the imported stylesheet can be be overridden by instructions in the importing stylesheet and in any included stylesheet. For example, the following `makehtml2.xsl` stylesheet tells the XSLT processor to import the `attach.xsl` stylesheet.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     version="1.0">
  <xsl:import href="attach.xsl"/>
  <xsl:template match="po_name">
   <p><xsl:apply-templates/></p>
  </xsl:template>
</xsl:stylesheet>
```

If `attach.xsl` looks like this,

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     version="1.0">
  <xsl:template match="po_name">
   <h1><xsl:apply-templates/></h1>
  </xsl:template>
</xsl:stylesheet>
```

the XSLT processor will treat:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     version="1.0">
  <xsl:import href="attach.xsl"/>
  <xsl:template match="po_name">
   <p><xsl:apply-templates/></p>
  </xsl:template>
</xsl:stylesheet>
```

as:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     version="1.0">

  <xsl:template match="po_name">
   <p><xsl:apply-templates/></p>
  </xsl:template>

</xsl:stylesheet>
```

The `attach.xsl` stylesheet's `"po_name"` template rule was ignored while importing because of the existing template rule with the same match.

Whether you want to use `xsl:import` or `xsl:include` depends on whether you want to override some of the templates that are defined in the imported stylesheet: if you do, then use `xsl:import`, otherwise, use `xsl:include`. Use `<xsl:import>` to import common, general-purpose rules into a stylesheet designed to handle the specific transformation. If you can help it, don't `xsl:import` any more `xsl` than you need.

Securing XML Documents

In the last section, we discussed various best practices for designing your XML documents in web services. However, the web services model brings into the system unique security challenges because the business data in the form of XML documents may be required to travel across untrusted networks and has the chance of being manipulated by external systems.

Throughout the entire business transaction, different classes of users and systems need access to the entire business transaction. If any part of this chain is compromised, the whole business application deployed as a service will fail.

Web services are inherently about how to share the process of computing across a distributed network of systems. Web services' communication channel being XML, messages are text-based, readable, and self describing.

XML Security Threats

All the components in web services are described in XML. SOAP and all the WS -Security specifications are XML formats. Hence it just makes sense for expressing security data in XML format. Fortunately, there has been no need to invent new cryptography technologies for XML. The XML security standards have used existing cryptography directly. XML-based data transfer has emerged as the

standard for organizations to exchange business data. As with all communications over the public Internet, XML-based transfers have their own set of vulnerabilities to confront. Like any other document exchange, XML document exchange must support the usual security measures which are Confidentiality, Integrity, Authenticity, and Non-Repudiation. The following list illustrates some specific XML security threats:

- **Schema Altering**—Manipulation of WS schema to alter the data processed by the application.
- **XML Parameter Tampering**—Injection of malicious scripts or content into XML parameters
- **Coercive Parsing**—Injection of malicious content into the XML
- **Oversized Payload**—Sending oversized files to create an XDoS attack
- **Recursive Payload**—Sending mass amounts of nested data to create an XDoS attack against an XML parser
- **XML Routing Detours**—Redirecting sensitive data within the XML path
- **External Entity Attack**—An attack on an application that parses XML input from suspicious sources using an incorrectly configured XML parser

These threats pose potentially serious problems to developers creating applications, components, and systems that depend on XML data. The solution for the above problems is XML Encryption.

XML Encryption

XML Encryption provides end-to-end security for applications that require secure exchange of structured data. XML itself is the most popular technology for structuring data, and therefore XML-based encryption is the natural way to handle complex requirements for security in data interchange applications.

XML Encryption is a process for encrypting and decrypting parts of XML documents. Most of today's encryption schemes use transport-level techniques that encrypt an entire request and response stream between a sender and receiver, offering zero visibility into contents of the interchange to intermediaries. Content-level encryption converts document fragments into illegible ciphertext, while other elements remain legible as plaintext.

Some features of XML encryption are:

1. The ability to encrypt a complete XML file
2. The ability to encrypt a single element of an XML file
3. The ability to encrypt only the contents of an XML element
4. The ability to encrypt binary data within an XML file

Encrypting an XML File

Here's a short sample XML file that can serve to demonstrate XML encryption:

```
<?xml version='1.0'?>
<POInfo xmlns='http://packtpub.com/payments'>
  <Name>FJ</Name>
  <Amount>125.00</Amount>
  <CreditCardNumber>1234-5678-4564-4321</CreditCardNumber>
  <Date>July 6, 2006</Date>
</POInfo>
```

When you encrypt an entire XML file, the process simply replaces the root element (`<POInfo>` in the sample) with an `<EncryptedData>` element that contains the encryption details, including the encrypted content.

Here is how the encrypted file will look:

```
<?xml version="1.0" encoding="UTF-8"?>
<xenc:EncryptedData
   xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
   Type="http://www.w3.org/2001/04/xmlenc#Element">
   <xenc:EncryptionMethod
      Algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc"
      xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"/>
   <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
      <xenc:EncryptedKey
         xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
         <xenc:EncryptionMethod
            Algorithm="http://www.w3.org/2001/04/xmlenc#kw-tripledes"
            xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"/>
         <xenc:CipherData
            xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
            <xenc:CipherValue
               xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
               MKeT0ZmHFLwnZaSXO+oZSxlSJ5/BqvblqG76B3nOMU0=
            </xenc:CipherValue>
         </xenc:CipherData>
      </xenc:EncryptedKey>
   </ds:KeyInfo>
   <xenc:CipherData
      xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
      <xenc:CipherValue
         xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
          +M/Tamk/62Lut4HqLpU/es9sdhnNTTpasbeszN8GN8EAJZsX0vvClcKEW
          UAgIdbvyJpprQ+jUIiWJKTz1X3L6VAefHqO963pU3bzmGMo
```

```
            pHLqS1Eg7iAPFhKV1PJclyswyyepEjyu+bOgqzgGnS1XA0/V
            NP7kLK70rB2Zb0DSbaCi+7HjTNGWF9YKtPIP5bvrs5xw+x
            HnKO++2EuqzK+deD7mCu8w6sG9vmRCrUR99Mx1QDZon9a2962ZD
            FSwoIJKg5I83GzOU+RObBBUme+yTf7UWybEiwtHp5ZgvuaQYJA=
        </xenc:CipherValue>
    </xenc:CipherData>
</xenc:EncryptedData>
```

Encrypting a Single Element

To encrypt a single element of an XML file, you specify the desired child element, rather than the root element of the input file as the element to encrypt. The following snippet shows the results of encrypting only the `<CreditCardNumber>` element of the sample file.

```
<?xml version="1.0" encoding="UTF-8"?>
<POInfo xmlns="http://jeffhanson.com/payments">
   <Name>John Doe</Name>
   <Amount>125.00</Amount>
   <xenc:EncryptedData
      xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
      Type="http://www.w3.org/2001/04/xmlenc#Element">
      <xenc:EncryptionMethod
         Algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc"
         xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"/>
      <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
         <xenc:EncryptedKey
            xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
            <xenc:EncryptionMethod
               Algorithm=
                 "http://www.w3.org/2001/04/xmlenc#kw-tripledes"
               xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"/>
            <xenc:CipherData
               xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
               <xenc:CipherValue
                 xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
                  6zhAcEW7KIKrbSjEOkXDrVkmws5zhQQLDO4YYW+RfRY=
               </xenc:CipherValue>
            </xenc:CipherData>
         </xenc:EncryptedKey>
      </ds:KeyInfo>
      <xenc:CipherData
         xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
         <xenc:CipherValue
            xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
```

```
                JqsRmdSoS+PXqCe80Y8zNiQ49sHTLNaAgHX1Ja7d+u9fv
                TFBrkBMK7C7EHsQTglZ3yT9yCZDuFnjBoQTLULKqOy71Qw
                EPRPObtYLPIJgy1vUdNrw47uDmJ/R5r/B0SH37HN8mfNv
                i50zPt1qPxxRwA==
            </xenc:CipherValue>
        </xenc:CipherData>
    </xenc:EncryptedData>
    <Date>July 6, 2005</Date>
</POInfo>
```

Notice that the encryption process replaced the `<CreditCardNumber>` tag and its contents with an `<EncryptedData>` tag, while leaving the siblings of the `<CreditCardNumber>` element unaltered.

This type of encryption can be performed using XML Signature and Encryption. The interested reader may look up the implementation at the Apache site (`http://xml.apache.org/security/`).

Best practices for XML encryption, can be summarized as follows:

- It is good to have standard element tags for representing encrypted elements within the XML documents. This will enable parsers to better understand encrypted elements and data during the validation process.
- It is necessary to provide means for encrypting only the desired elements within an XML document instead of encrypting the whole document. This will pave the way for incorporating several confidential data elements that are intended for different recipients within a single XML document.
- There should be standard mechanisms for exchanging the secret keys used for encryption and decryption processes.
- The standard should allow encryption of different parts of the document with different keys, so that multiple recipients can decrypt only those portions that are intended for them.
- The standards should be adaptable to both ASCII and binary data.
- The standards should be adaptable to different cryptographic algorithms.
- The standards should work along with other XML security standards and specifications.

SSL versus XML Encryption

Why do we need exclusive standards and methodology for XML encryption when we already have established technologies like Secure Socket Layers (SSL) and Transport Security Layer (TSL), which also use cryptographic concepts to secure communications?

Although SSL and TSL are good for securing communications across two parties, they are not suitable for multi-party interactions, which is a typical characteristic of XML/web service interactions. Also, SSL and TSL do not have the capacity to encrypt only specific parts of the document or to encrypt different portions of the document using different keys—which are critical to XML encryption.

SSL guarantees security of payload during the transit. However, when the payload reaches a node, it is unprotected. Also, SSL is a point-to-point security mechanism where as XML encryption guarantees end-to-end security.

XML Signatures

XML Signature is a W3C recommendation that defines XML syntax for digital signatures. It is used by various web technologies such as SOAP, SAML, and others.

XML signatures will enable a sender to cryptographically sign data, and the signatures can then be used as authentication credentials or a way to check data integrity. This also guarantees non-repudiation.

XML signatures can be applied to any XML resource, such as XML, an HTML page, binary-encoded data such as a GIF file, and XML-encoded data. The primary feature of the XML digital signature is its ability to sign only specific portions of the XML document just like the encryption example we discussed in the previous section.

XML signatures can be broadly classified into three types:

1. Enveloped signature—An enveloped signature is the signature applied over the XML content that contains the signature as an element. The Signature element is excluded from the calculation of the signature value.
2. Enveloping signature—An enveloping signature is the signature applied over the content found within an Object element of the signature itself. The object or its content is identified through a Reference element by way of a Uniform Resource Identifier (URI) fragment identifier or transform.
3. Detached signature—A detached signature is the signature applied over the content external to the Signature element, and it can be identified by way of a URI or a transform. The signed XML resource can be present within the same document as the Signature element, or it can be external to the XML document.

Guidelines for Securing Your Services

Security testing, which is important for any software application, is even more crucial for web services. Web services' security architecture not only depends on the standard security measures, it also depends on the service scope and scale of deployment. For instance, security can either be enforced in the application server itself, or as a separate security appliance that can virtualize the service by sitting in the middle between the service and its consumers.

The points that you need to consider for securing your web services are:

1. Find suitable security architecture—Decide whether to implement the security on the transport layer or on the message layer. TLS (Transport Layer Security) is a mature technology so both standards and tools have already been developed. It also provides a good transition path for engineers who are somewhat familiar with transport-level security but are new to web services. On the other hand, TLS has inherent limitations that make it inappropriate when you desire an end-to-end security. Fortunately, message layer security provides an alternative solution for situations where TLS's limitations are troublesome.
2. Follow the standards religiously—publicly available, commonly used, well-analyzed cryptographic algorithms are the best choice, simply because they've already undergone a great deal of research and scrutiny since they were adopted by the industry.
3. Filter your XML—XML requires complex processing to ensure that transactions are known to be good before they penetrate deep into the enterprise. XML filtering offers a variety of functionality as complex rule sets can be built around network-level information, message size, and message content.
4. Prevent XML Denial-of-Service (XDos)—To protect against XDoS, implement reasonable constraints for all incoming messages. With the use of an XML security gateway as a proxy, you can configure simple settings on message size, frequency, and connection duration.

Securing your web services is a vital aspect of ensuring a successful deployment. When deployed externally for consumption by partners or customers, only secure web services can provide a justifiable integration solution. An interested reader may refer to the WS-Security standard for creating secured web services. WS-Security is a communications protocol originally developed by IBM, Microsoft, Verisign, and Forum Systems. It contains specifications on how integrity and confidentially can be enforced on web services messaging. It describes how to attach signature and encryption headers to SOAP messages. It also describes how to attach security tokens, including binary tokens such as X.509 certificates and Kerberos tickets, to XML messages.

XML Streaming and DOM

There are two programming models for working with XML documents i.e. document *streaming* and the *document object model* (DOM).

The *DOM* model involves creating in-memory objects representing an entire document tree and the complete document state. Once in memory, DOM trees can be navigated freely and parsed arbitrarily, and provide maximum flexibility. However, the cost of this flexibility is a potentially large memory footprint and significant processor requirements, as the entire representation of the document must be held in memory as objects for the duration of the document processing. This may not be an issue when working with small documents, but memory and processor requirements can escalate quickly with document size.

S*treaming* refers to a model in which XML documents are transmitted and parsed serially at application run time, often from dynamic sources whose contents are not precisely known beforehand. Moreover, stream-based parsers can start generating output immediately. While providing a smaller memory footprint, reduced processor requirements, and higher performance in certain situations, the primary trade-off with stream processing is that you can only see the document state at one location at a time in the document.

Streaming models for XML processing are particularly useful when your application has strict memory limitations, as in the case of hand held devices (PDAs, Smart phones) that run J2ME, Windows Mobile, and Symbian OS, or when your application needs to simultaneously process several requests, as with an application server.

Pull Parsing versus Push Parsing

Streaming *pull parsing* refers to a model in which a client application calls methods on an XML parsing library when it needs to interact with an XML document; the client only gets (pulls) XML data when it explicitly asks for it.

Streaming *push parsing* refers to a model in which an XML parser sends (pushes) XML data to the client as the parser encounters elements in an XML document; the parser sends the data whether or not the client is ready to use it at that time.

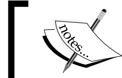
Pull parsing provides several advantages over push parsing when working with XML streams:

- With pull parsing, the client controls the application thread, and can call methods on the parser when needed. By contrast, with push processing, the parser controls the application thread, and the client can only accept invocations from the parser.

- Pull parsing libraries can be much smaller and the client code to interact with those libraries much simpler than with push parsing libraries, even for more complex documents.
- Pull clients can read multiple documents at one time with a single thread.

What is StAX?

The StAX project was started by BEA with support from Sun Microsystems, and the JSR 173 specification. The primary goal of the StAX API is to give "parsing control to the programmer by exposing a simple iterator-based API. This allows the programmer to ask for the next event (pull the event) and allows state to be stored in procedural fashion." StAX was created to address limitations in the two most prevalent parsing APIs, SAX and DOM.



Note: On Microsoft's .Net platform, the equivalent to this is the `System.Xml.XmlReader` class.

StAX and Other JAXP APIs

As an API in the JAXP family, StAX can be compared, among other APIs, to SAX, TrAX, and JDOM. Of the latter two, StAX is not as powerful or flexible as TrAX or JDOM, but neither does it require as much memory or processor load to be useful, and StAX can, in many cases, outperform the DOM-based APIs.

StAX offers features that are beneficial in many cases; these include:

- StAX-enabled clients are generally easier to code than SAX clients. While it can be argued that SAX parsers are marginally easier to write, StAX parser code can be smaller and the code necessary for the client to interact with the parser simpler.
- StAX is a bidirectional API, meaning that it can both read and write XML documents. SAX is read only, so another API is needed if you want to write XML documents.
- SAX is a push API, whereas StAX is pull. The trade-offs between push and pull APIs apply here.

The following table compares the JAXP APIs.

Feature	StAX	SAX	DOM	TrAX
API Type	Pull, Streaming	Push, Streaming	In memory tree	XSLT Rule
Ease of Use	High	Medium	High	Medium
XPath	No	No	Yes	Yes
CPU & Memory Consumption	Good	Good	Differs	Differs
Forward Only	Yes	Yes	No	No
Read XML	Yes	Yes	Yes	Yes
Write XML	Yes	No	Yes	Yes
Create, Read, Update, and Delete.	No	No	Yes	No

Performance Considerations

It is important to consider performance when processing XML documents. XML document processing—handling the document in a pre- or post-processing stage to an application's business logic—may adversely affect application performance because such processing is potentially very CPU, memory, and input/output or network intensive.

There are factors with XML document processing that affect performance. Often, the physical and logical structures of an XML document may be different. An XML document may also contain references to external entities. These references are resolved and substituted into the document content during parsing, but prior to validation. Given that the document may originate on a system different from the application's system, and external entities—and even the schema itself—may be located on remote systems, there may be network overhead affecting performance. To perform the parsing and validation, external entities must first be loaded or downloaded to the processing system. This may be a network intensive operation, or require a great deal of input and output operations, when documents have a complex physical structure.

In summary, XML processing is potentially CPU, memory, and network intensive, for these reasons:

- It may be CPU intensive. Incoming XML documents need not only to be parsed but also validated and they may have to be processed using APIs that may themselves be CPU intensive. It is important to limit the cost of validation as much as possible without jeopardizing the application processing and to use the most appropriate API to process the document.

- It may be memory intensive. XML processing may require creating large numbers of objects, especially when dealing with document object models.
- It may be network intensive. A document may be the aggregation of different external entities that during parsing may need to be retrieved across the network. It is important to reduce as much as possible the cost of referencing external entities.

Following are some guidelines for improving performance when processing XML documents. In particular, these guidelines examine ways of improving the CPU, memory, and input/output or network consumption.

Limit Parsing of Incoming Documents

In general, it is best to parse incoming XML documents only when the request has been properly formulated. In the case of a web service application, if a document is retrieved as a source parameter from a request to an endpoint method, it is best first to enforce security and validate the meta information that may have been passed as additional parameters with the request.

In a more generic messaging scenario, when a document is wrapped inside another document (considered an envelope), and the envelope contains meta-information about security and how to process the inner document, you may apply the same recommendation: Extract the meta-information from the envelope, then enforce security and validate the meta-information before proceeding with the parsing of the inner document. When implementing a SAX handler and assuming that the meta-information is located at the beginning of the document, if either the security or the validation of the meta-information fails, then the handler can throw a SAX exception to immediately abort the processing and minimize the overall impact on performance.

Use the Appropriate API

It's important to choose the most appropriate XML processing API for your particular task. In this section, we look at the different processing models in terms of the situations in which they perform best and where their performance is limited.

In general, without considering memory consumption, processing using the DOM API tends to be slower than processing using the SAX API. This is because the DOM may have to load the entire document into memory so that the document can be edited or data retrieved, whereas SAX allows the document to be processed as it is parsed. However, despite its initial slowness, it is better to use the DOM model when the source document must be edited or processed multiple times.

When using higher-level technologies such as XSLT, keep in mind that they may rely on lower-level technologies like SAX and DOM, which may affect performance, possibly adversely.

Choosing Parser

Each parser and stylesheet engine implementation is different. For example, one might emphasize functionality, while another performance. A developer might want to use different implementations depending on the task to be accomplished. Consider using JAXP, which not only supports many parsers and stylesheet engines, but also has a pluggability feature that allows a developer to swap between implementations and select the most effective implementation for an application's requirements. If you are using Microsoft's .Net platform, look up System.Xml namespace for XML parsing classes.

Reduce Validation Cost

Not only is it important, but validation may be required to guarantee the reliability of an XML application. An application may legitimately rely on the parser's validation so that it can avoid double-checking the validity of document contents. Validation is an important step of XML processing, but keep in mind that it may affect performance.

Although you must validate external incoming XML documents, you can exchange freely—that is, without validation—internal XML documents or already validated external XML documents. In short, you need to validate only at the system boundaries, and you may use validation internally only as an assertion mechanism during development. You may turn validation off when in production and looking for optimal performance.

In other words, when you are both the producer and consumer of XML documents, you may use validation as an assertion mechanism during development, then turn off validation when in production. Additionally, during production validation can be used as a diagnostic mechanism by setting up validation so that it is triggered by fault occurrences.

Referencing External Entities

An XML document may be the aggregation of assorted external entities, and these entities may need to be retrieved across the network when parsing. In addition, the schema may also have to be retrieved from an external location. External entities, including schemas, must be loaded and parsed even when they are not being validated to ensure that the same information is delivered to the application regardless of any subsequent validation. This is especially true with respect to default values that may be specified in an incoming document schema.

There are two complementary ways to reduce the cost of referencing external entities:

1. **Caching using a proxy cache:** You can improve the efficiency of locating references to external entities that are on a remote repository by setting up a proxy that caches retrieved, external entities. However, references to external entities must be URLs whose protocols the proxy can handle.
2. **Caching using a custom entity resolver:** SAX parsers allow XML applications to handle external entities in a customized way. Such applications have to register their own implementation of the `org.xml.sax.EntityResolver` interface with the parser using the `setEntityResolver` method. The applications are then able to intercept external entities (including schemas) before they are parsed.

Dynamically Generated Documents

Dynamically generated documents are typically assembled from values returned from calls to business logic. Generally, it is a good idea to cache dynamically generated XML documents to avoid having to re-fetch the document contents, which entails extra round trips to a business tier. This is a good rule to follow when the data is predominantly read only, such as catalogue data. Furthermore, if applicable, you can cache document content (DOM tree or JAXB content tree) in the user's session on the interaction or presentation layer to avoid repeatedly invoking the business logic.

However, you quickly consume more memory when you cache the result of a user request to serve subsequent, related requests. When you take this approach, keep in mind that it must not be done to the detriment of other users. That is, be sure that the application does not fail because of a memory shortage caused by holding the cached results. To help with memory management, use soft references, which allow more enhanced interaction with the garbage collector to implement caches.

Using XML Judiciously

XML documents can enhance web service interoperability: Heterogeneous, loosely coupled systems can easily exchange XML documents because they are text documents. However, loosely coupled systems must pay the price for this ease of interoperability, since the parsing that these XML documents require is very expensive. This applies to systems that are loosely coupled in a technical and an enterprise sense.

Contrast this with tightly coupled systems. System components that are tightly coupled can use standard, non-document-oriented techniques (such as RMI or .Net Remoting) that are far more efficient in terms of performance and require far less coding complexity. Fortunately, with technologies such as JAX-WS and JAXB you can combine the best of both worlds. Systems can be developed that are internally tightly coupled and object oriented, and that can interact in a loosely coupled, document-oriented manner.

Generally, when using XML documents, follow these suggestions:

- Rely on XML protocols, such as those implemented by JAX-WS and others, to interoperate with heterogeneous systems and to provide loosely coupled integration points.
- Avoid using XML for unexposed interfaces or for exchanges between components that should be otherwise tightly coupled.

Summary

This chapter discussed various design anomalies that may arise while designing XML schemas. As mentioned in this chapter, there are several ways to design a schema that suits your architecture requirements. The suggestions given in this chapter are meant for consideration only when you already know your system very well.

Some of the broad categories covered in this chapter are enumerated as follows:

1. Design recommendations for architecting domain-specific XML Schemas
2. Tips for designing XML schemas with examples
3. Using XSL effectively for translating Infosets from one form to another
4. Securing XML documents with encryption and digital signature
5. XML serialization and the differences between SAX, DOM, and StAX
6. Some of the performance issues concerned with using XML as a medium of application integration

4

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.

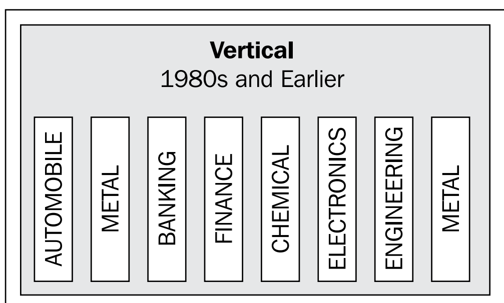
- Interoperability Challenges in Web Services: An SOA might use several web services deployed on disparate platforms and technologies. These web services must interoperate with each other. Several specifications have come up to achieve this interoperability. In this section, you will study these specifications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

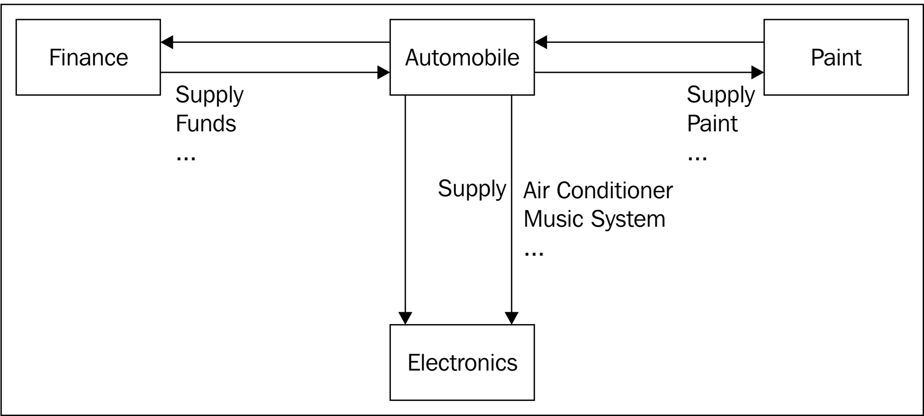
Over the last several decades, corporations have developed applications based on a wide range of systems and technologies. Though these systems are component-ized, the integration of these systems poses a challenge due to their heterogeneity. The need for integration arises due to globalization and a 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:

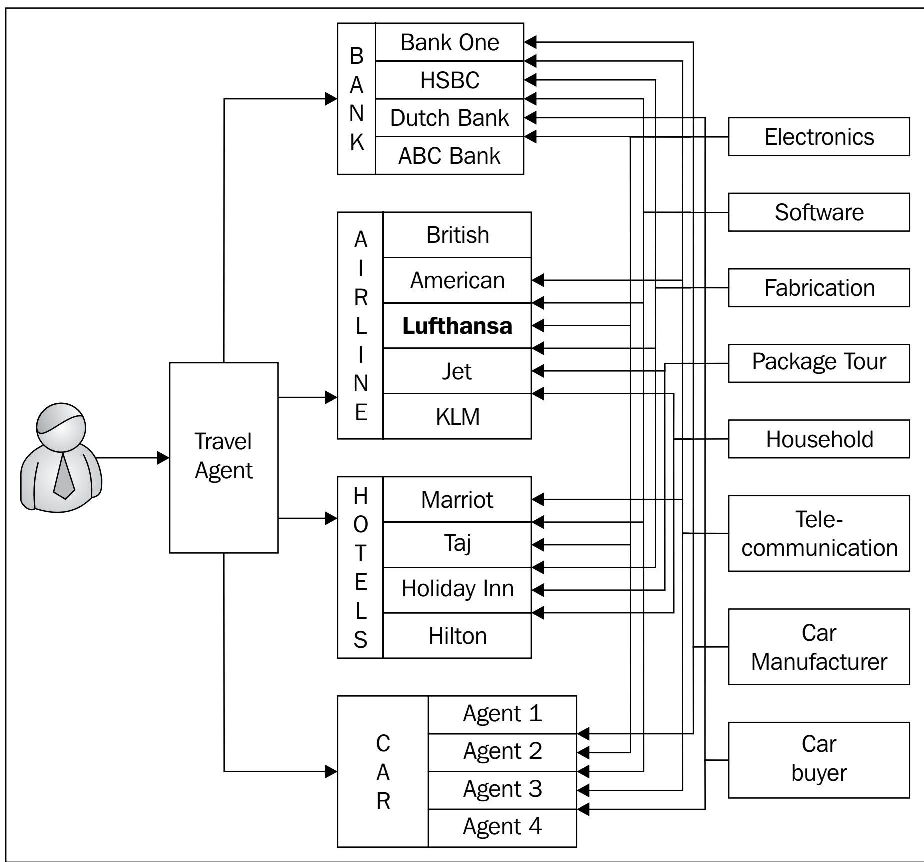


In the late '80s and early '90s, we saw the need for business applications to grow horizontally to cooperate with business partners. The industry saw the evolution of B2B (Business-to-Business) collaborations through components now spreading across several industry verticals. These components were now 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.

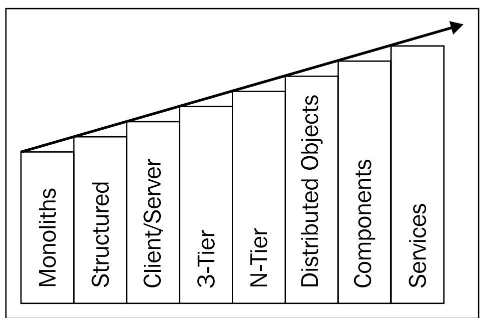
Such interactions are depicted in the following figure of today's complex IT requirements.



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.

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.

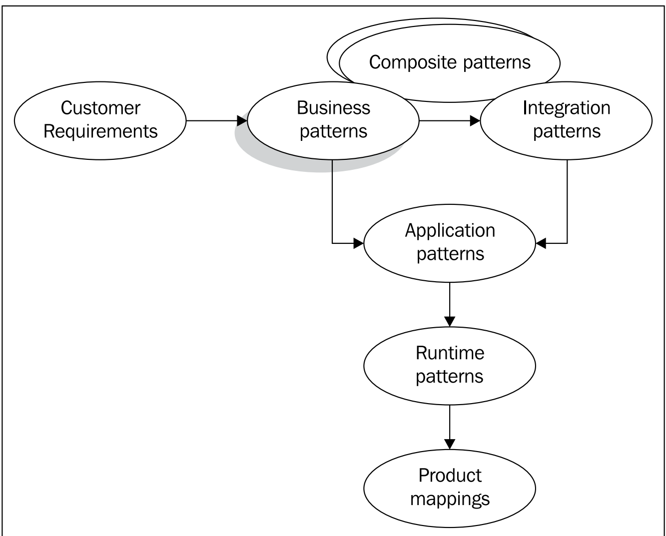


In the early years of computing, we had only monolithic applications running on stand-alone machines. From the monolithic systems of early '60s, the industry saw the development of structured, client/server, 3-tier, N-tier, distributed systems, and finally the service-oriented architectures of the modern 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

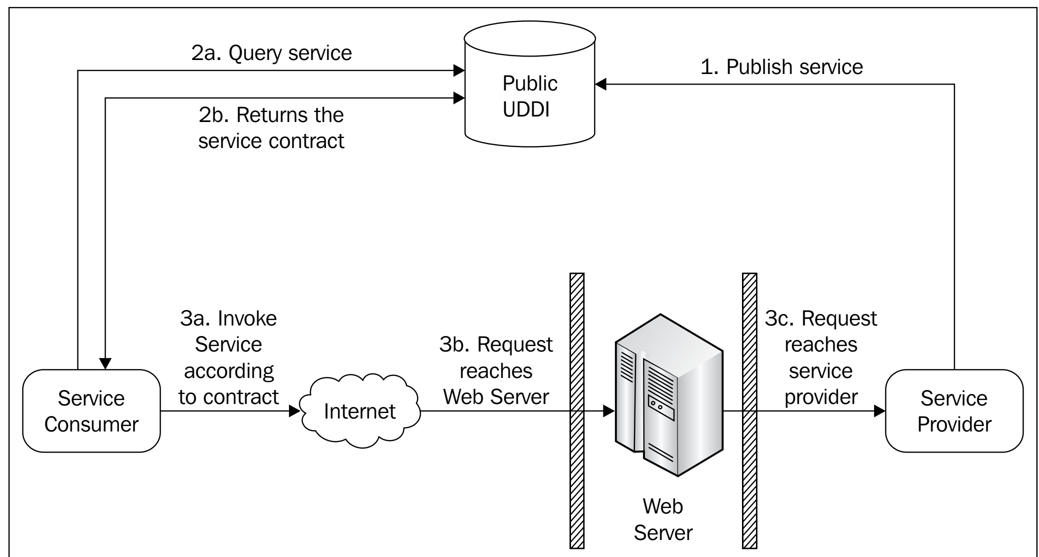
The Web Services technology plays an important role when applying the concepts of Service-Oriented Architectures. The web services technology is based on open standards such as:

- 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.

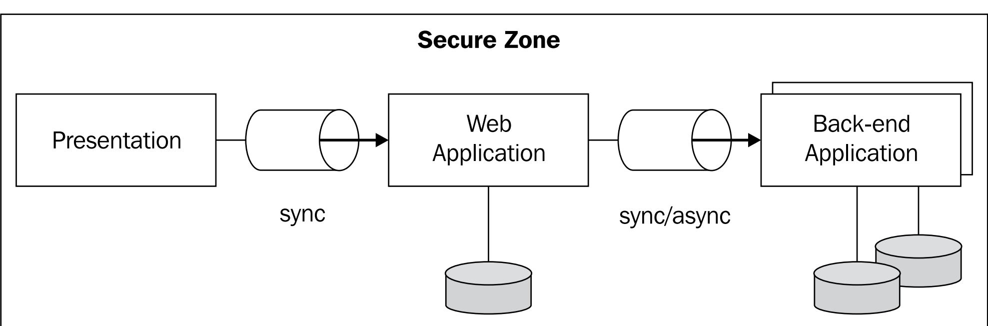
Patterns

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.

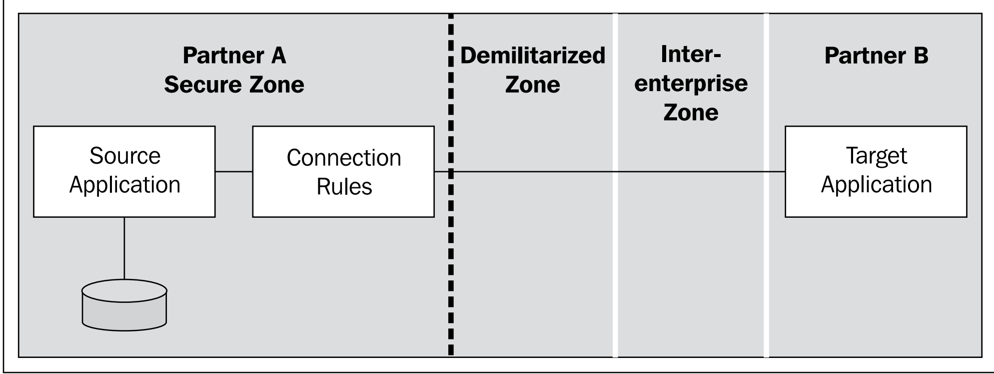
Guidelines

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 patterns 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

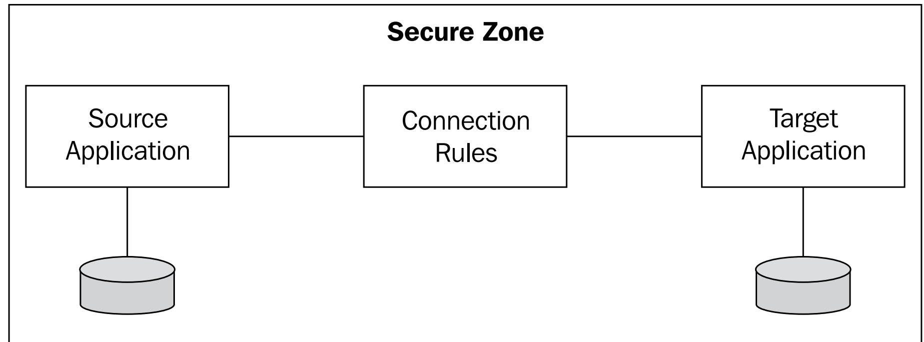
When applied to the Extended Enterprise domain, these are classified as:

- 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.

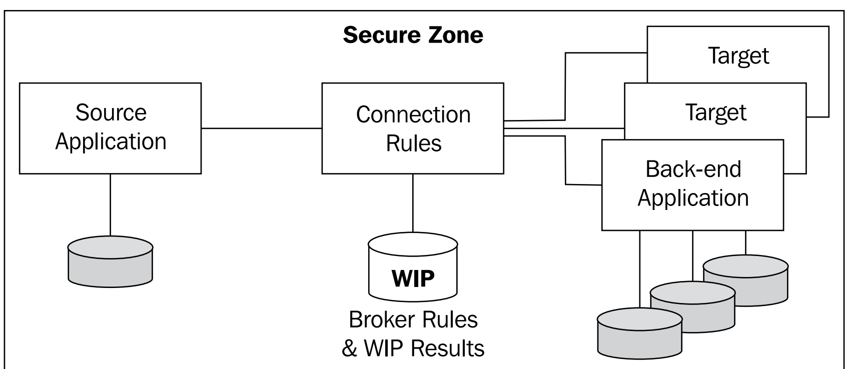
Guidelines

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.

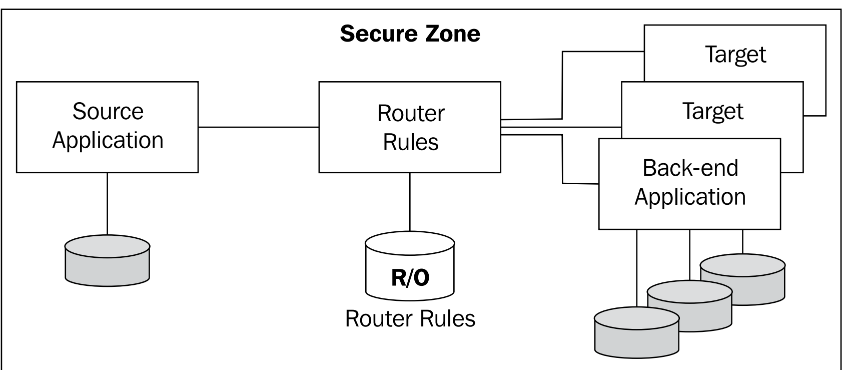
Broker Application Pattern

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 interactiond 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.

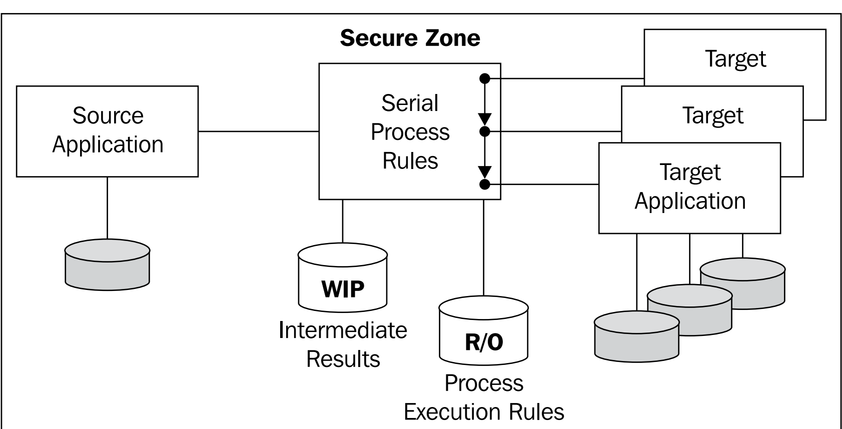
In the case of an Extended Enterprise domain, this pattern defines the interaction between a source application and the multiple target partner applications.

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.



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

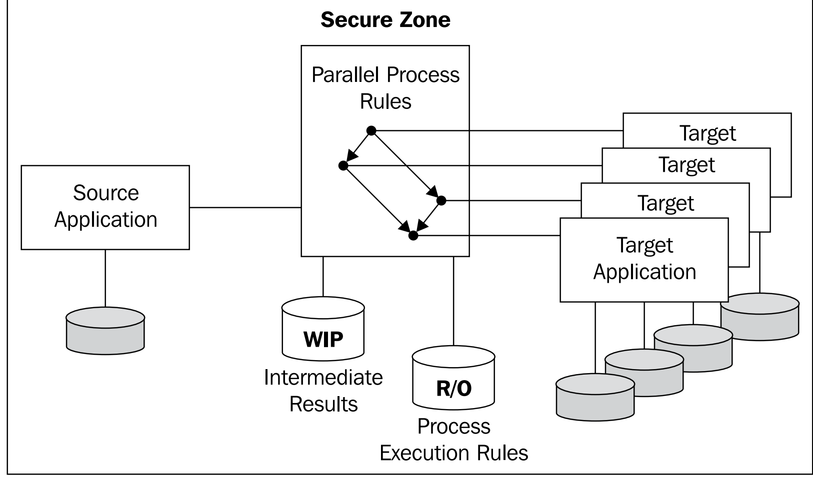
The Serial Process application pattern fits perfectly in the SOA paradigm. The businesses provide autonomous services. A consumer requires a series of services to be executed in a desired sequence. The orchestration flow may be defined at run time. The consumer requests a certain kind of business service 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.

The practical implementation of this pattern is seen in the case of orchestration servers based on BPEL (Business Process Execution Language) discussed in depth in Chapter 5. The reader may also refer to a Packt book on BPEL (*Business Process Execution Language for Web Services, Second Edition,* ISBN 978-1-904811-81-7).

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.

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.

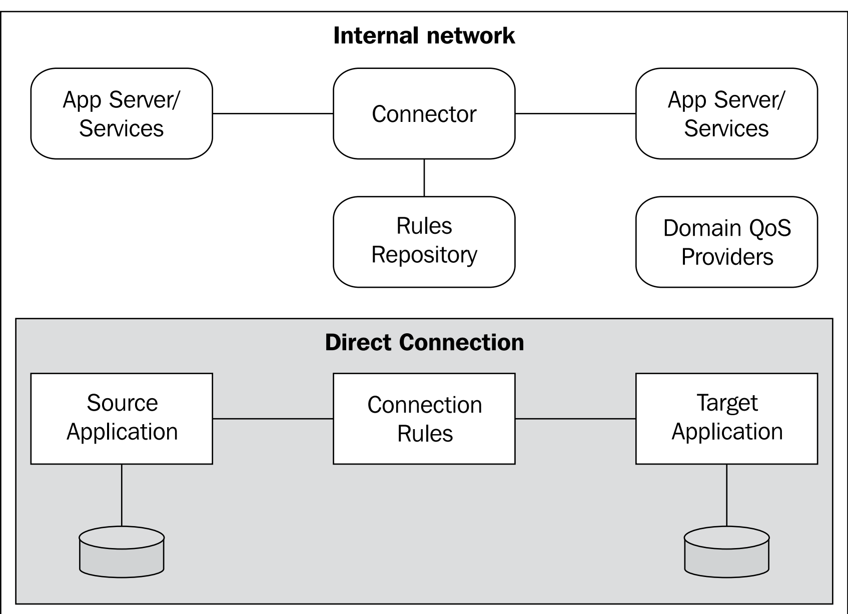
Path Connector

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.

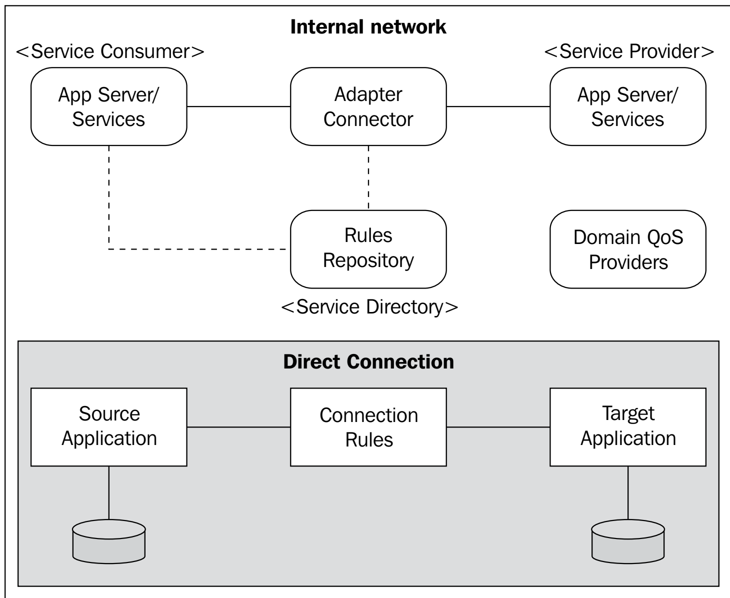
This pattern provides a direct connection between a service consumer and a service provider with the help of connectors. The pattern is classified into the following categories depending on the connector used:

- Single Adapter connector
- Coupling Adapter connector
- Service Bus

Each of these is discussed further.

Direct Connection Pattern using Single Adapter

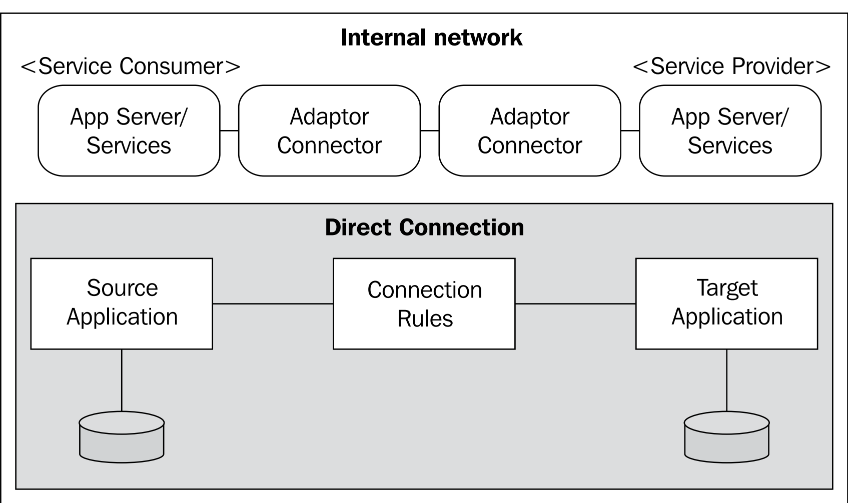
This pattern is depicted in the following figure.



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

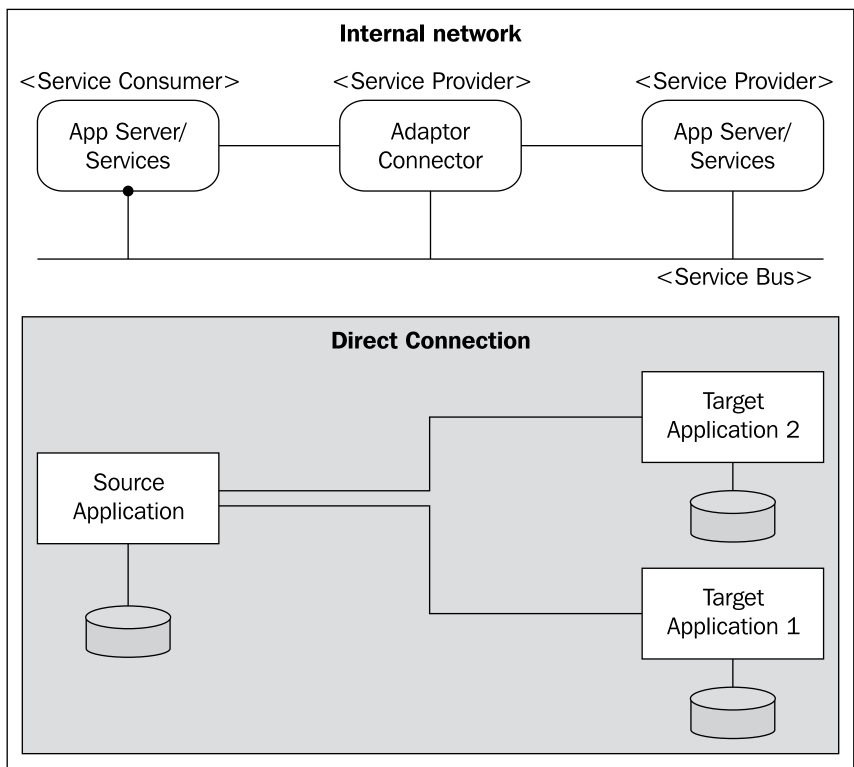
This pattern is depicted in the following figure.



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

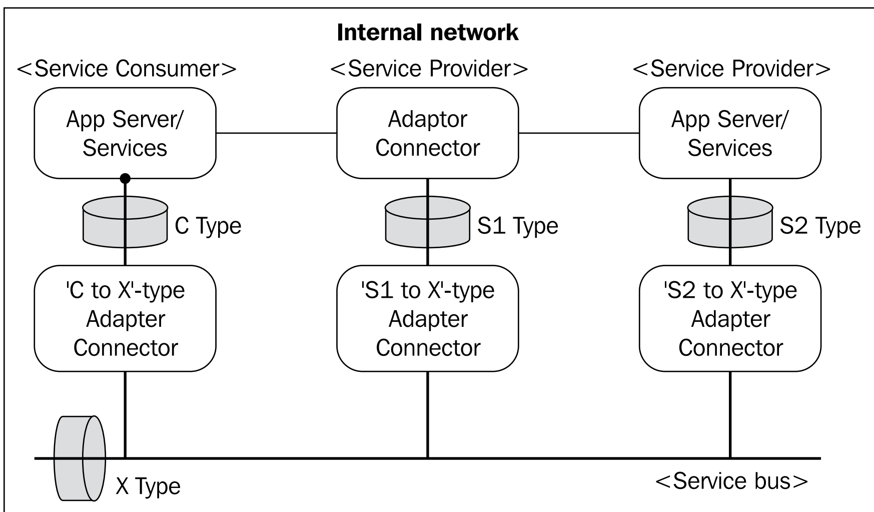
This pattern is depicted in the following figure.



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.

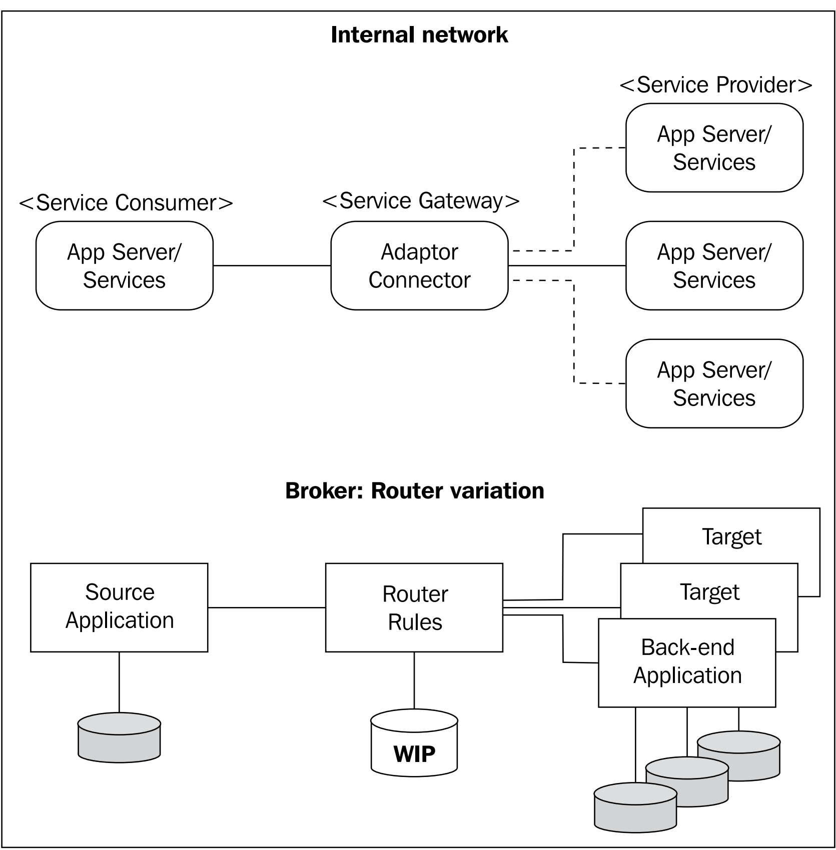


Here, the service bus is said to be of **X-type** as each of the application services connects to this X-type connector. An example of such an X-type service bus could be an HTTP service bus or a JMS (Java Messaging Service) service bus. The X-type adapter connectors bridge the service consumers and 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.

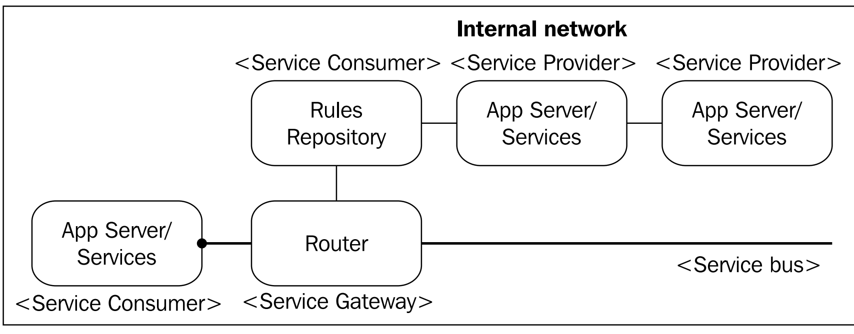
Runtime Patterns for Broker

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 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/wsdl/http/"
              xmlns:soapenc=
                       "http://schemas.xmlsoap.org/soap/encoding/"
              xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
                  style="document"/>
    <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>
        <wsdl:operation name="oneWayOperation">
            <soap:operation
                soapAction="http://www.mycompany.com/InputMessage"
                style="document"/>
            <wsdl:input>
                <soap:body use="literal"/>
            </wsdl:input>
        </wsdl:operation>
    </wsdl:binding>

    <wsdl:service name="MyWebService">
        <wsdl:port name="MyWebServiceSoap" binding="tns:
MyWebServiceSoap">
            <soap:address
                location="http://localhost/WebApplication1/
MyWebService.asmx"/>
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

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.

```
<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:

```
<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.

```
<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 `wsdl:binding` tag.

```
<wsdl:binding name="MyWebServiceSoap"
              type="tns:MyWebServicePortType"
```

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.

```
<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>
```

8. For one-way operation, make modifications similar to the previous bulleted item in the code lines below:

```
<wsdl:operation name="oneWayOperation">
    <soap:operation
        soapAction="http://www.mycompany.com/InputMessage"
        style="document"/>
    <wsdl:input>
        <soap:body use="literal"/>
    </wsdl:input>
</wsdl:operation>
```

9. Finally, modify the following line in the service tag to specify the URL for your service.

```
<soap:address location="http://www.yourCompany.com/
               WebApplication1/MyWebService.asmx"/>
```

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.

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:
    - A DESCRIPTION SHOULD NOT contain the namespace declaration `xmlns:xml=http://www.w3.org/XML/1998/namespace`.
- 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 `wsdl:part` elements bound to the envelope's corresponding `soapbind:body` element.
    - In a doc-literal description where the value of the parts attribute of `soapbind:body` is an empty string, the corresponding ENVELOPE MUST have no element content in the `soap:body` element.
    - In an rpc-literal description where the value of the parts attribute of `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 `name` attribute of the corresponding `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.

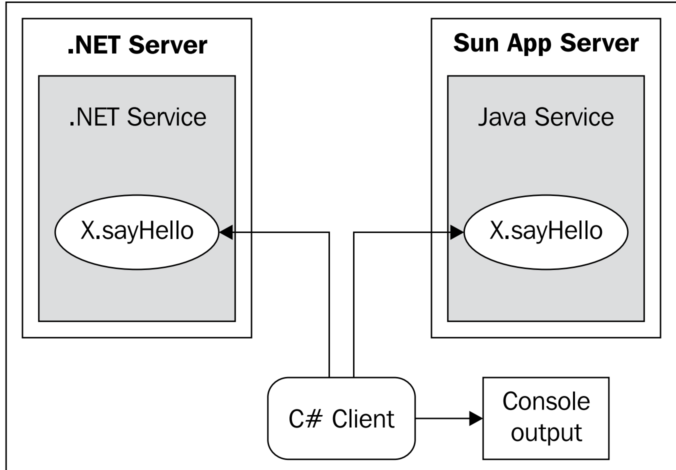
Having considered the various aspects of implementing SOA, the requirements for creating interoperable web services and various standards for interoperability, we will now take a practical approach to learning by demonstrating the creation of interoperable .NET and J2EE web services in the next section.

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.

```
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.

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

```
        </xsd:schema>
    </types>
    <message name="SayHello">
        <part name="parameters" element="tns:SayHello"/>
    </message>
    <message name="SayHelloResponse">
        <part name="parameters" element="tns:SayHelloResponse"/>
    </message>
    <portType name="Hello">
        <operation name="SayHello">
            <input message="tns:SayHello"/>
            <output message="tns:SayHelloResponse"/>
        </operation>
    </portType>
    <binding name="HelloPortBinding" type="tns:Hello">
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
style="document"/>
        <operation name="SayHello">
            <soap:operation soapAction=""/>
            <input>
                <soap:body use="literal"/>
            </input>
            <output>
                <soap:body use="literal"/>
            </output>
        </operation>
    </binding>
    <service name="HelloService">
        <port name="HelloPort" binding="tns:HelloPortBinding">
            <soap:address location="http://DRSARANG:8080/Hello/
HelloService"/>
        </port>
    </service>
</definitions>
```

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.

```
using System;
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;

[WebService(Namespace = "http://tempuri.org/")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
public class Service : System.Web.Services.WebService
{
    public Service () {

        //Uncomment the following line if using designed components
        //InitializeComponent();
    }

    [WebMethod]
    public string SayHello() {
        return "\nHello 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:

```
http://localhost:20278/NetService/Service.asmx?WSDL
```

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

```
<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">
<wsdl:input message="tns:SayHelloSoapIn"/>
<wsdl:output message="tns:SayHelloSoapOut"/>
</wsdl:operation>
</wsdl:portType>
   <wsdl:binding name="ServiceSoap" type="tns:ServiceSoap">
<soap:binding transport="http://schemas.xmlsoap.org/soap/http"/>
   <wsdl:operation name="SayHello">
<soap:operation soapAction="http://tempuri.org/SayHello"
style="document"/>
   <wsdl:input>
```

```
<soap:body use="literal"/>
</wsdl:input>
   <wsdl:output>
<soap:body use="literal"/>
</wsdl:output>
</wsdl:operation>
</wsdl:binding>
   <wsdl:binding name="ServiceSoap12" type="tns:ServiceSoap">
<soap12:binding transport="http://schemas.xmlsoap.org/soap/http"/>
   <wsdl:operation name="SayHello">
<soap12:operation soapAction="http://tempuri.org/SayHello"
style="document"/>
   <wsdl:input>
<soap12:body use="literal"/>
</wsdl:input>
   <wsdl:output>
<soap12:body use="literal"/>
</wsdl:output>
</wsdl:operation>
</wsdl:binding>
   <wsdl:service name="Service">
   <wsdl:port name="ServiceSoap" binding="tns:ServiceSoap">
<soap:address location="http://localhost:20278/NetService/Service.
asmx"/>
</wsdl:port>
   <wsdl:port name="ServiceSoap12" binding="tns:ServiceSoap12">
<soap12:address location="http://localhost:20278/NetService/Service.
asmx"/>
</wsdl:port>
</wsdl:service>
</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.

```
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.

5

BPEL and the Process-Oriented Approach to Integration

Service-oriented architectures reveal their full potential only after we have introduced the process layer. The process layer provides support for executing business processes that are composed out of services. The process-oriented approach to SOA (and integration) introduces several important changes to software architecture. On one side, it fosters the separation of process and business logic, where processes are developed in specialized languages, such as BPEL (Business Process Execution Language). On the other side, composition of services into processes is flexible, requires less development time and effort compared to "classic" approaches using languages such as Java or C#, and allows easier modifications in the future. In this way, the process-centric view of SOA allows software composition out of services. It introduces a new development model—programming-in-the-large instead of programming-in-the-small.

In this chapter, we will look at the process-oriented approach to SOA-based integration. We will look at:

- Process-oriented integration architectures
- Service composition with orchestration and choreography
- Complexity of business services, their identification, and their lifecycle
- Executable business processes
- BPEL for service composition
- What we can do with BPEL
- Executable and abstract processes
- BPEL and other process languages
- Details of writing BPEL processes, including process interface, partner links, partner link types, variables, handlers, fault handlers, event handlers, compensation handlers, and scopes
- Developing an example BPEL process

Process-Oriented Integration Architectures

The main objective of IT has traditionally been support for business operations. Initially, IT focused on support of business functions, such as accounting, warehouse management, etc. This resulted in so-called stove-pipe applications. Recently the focus has been shifted towards end-to-end support for business processes. Companies have started to realize that:

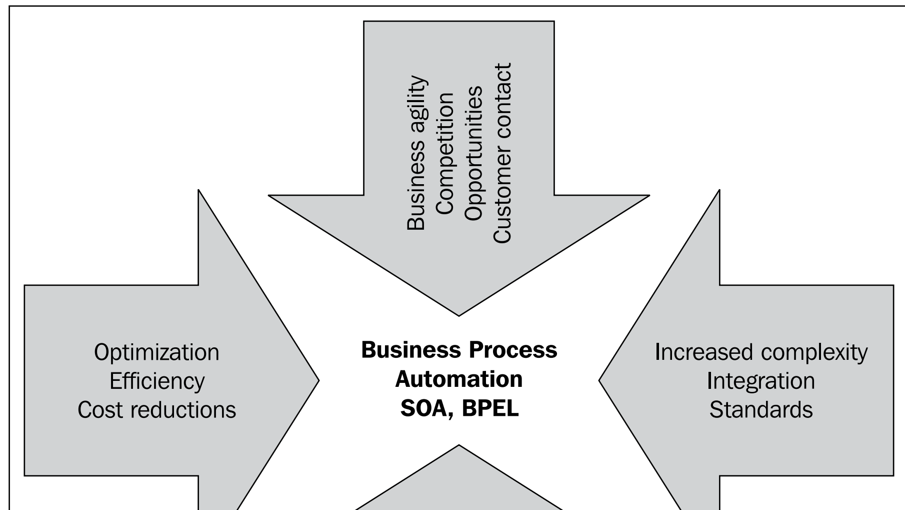
- It is important to automate business processes using IT. Automating the provisioning process for example can considerably improve the efficiency. This holds true for all business processes as well as for technical processes.
- It is important to optimize business process execution through measuring the efficiency of existing processes and improving them. In an automated provisioning process, for example, it would be possible to measure the time for each process activity and identify those activities that require the longest time to fulfill. In this way a company can focus on optimization of most critical activities and improve the overall performance of business processes.

The process-oriented approach to SOA enables companies to optimize business processes and discharge their employees from all routine activities that can be done by software. In this way companies can become more agile and responsive to market opportunities and customer needs, etc. All this leads to overall improvement of the companies' competitive position in the market.

There are four major categories of forces that influence business process automation through the use of information technologies. These are:

- Business aspects, such as agility, competition, new opportunities, customer demands, and contact
- Organizational aspects, such as the need for optimizations, improvements of efficiency, and cost reductions
- Increased complexity, integration demands, and standards
- Introduction of new technologies, such as Web 2.0, new devices, and architectures.

This is shown in the following figure:



Although companies have taken care of their processes in the past—most of them have used some sort of BPM (Business Process Management) approach—there has been a considerable gap between the business processes as designed on paper, and their execution in the information systems. In other words, companies have drawn their business processes on paper, often on a relatively high-level of abstraction, and also thought of how the processes could be optimized. Companies also had software and applications that supported the execution of these processes. However, this picture had several shortcomings:

- Usually, business processes were only partially supported by applications. Often several different applications supported the activities of a process. This required users (employees) to switch between different applications to get things done.
- Several activities of a process had to be done manually as there was no support in software for them.
- The order of activities had usually not been documented very well. Often it was in the heads of the employees only.

- It was difficult to achieve a clear and comprehensive picture about how business processes worked, how long activities took to fulfill, and where the bottlenecks were. Even worse, often companies were unable to get an overview of the processes running, identify in which activity they currently were, who was involved and responsible for fulfilling the activity, etc. All this gave companies (and their management) very few quantitative data about process execution.

The above-mentioned shortcomings in existing applications have become an important factor, which has slowed down the performance of companies. This is particularly true because this semantic gap between business processes and IT has had other consequences:

- The complexity of applications has been growing considerably, which has had a consequence on the ability to make changes and modifications, and on the reliability of software.
- As changes have been difficult to make, it usually took too long to implement new functionalities or adapt existing functions.
- Because of such high complexity, a lot of effort has been put into maintenance, which has been related to high costs of operating and running.

SOA addresses these problems. The real value of SOA is that it enables agile change and reassembly of services. It allows us to apply changes is small steps, rather than monoliths, which results in faster adaptation of applications to business needs.

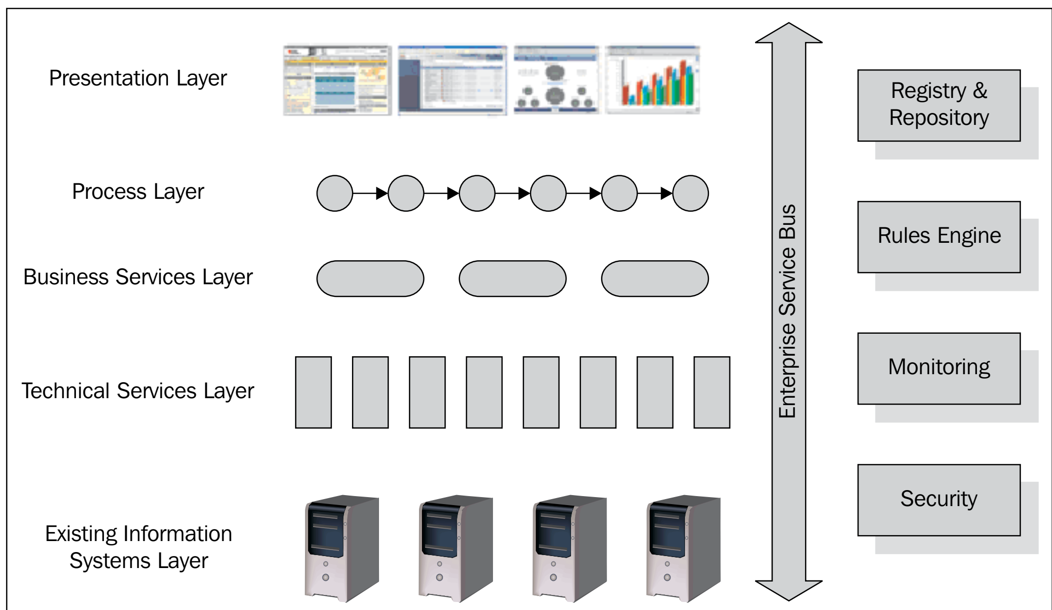
This is why SOA introduces layered architecture. It consists of several layers:

- Presentation layer
- Process layer
- Business services layer
- Technical services layer
- Existing information systems layer

In addition, SOA introduces the following infrastructure elements:

- Enterprise service bus
- Registry and repository
- Rules engine
- Business activity monitoring
- Security

These elements are shown in the following figure. In this chapter, we will focus particularly on processes and business services.



SOA, as you have learned in the previous chapters, has solutions for the previously mentioned problems. In the next section, we will look at service composition.

Service Composition

In SOA, composition of services is the concept with which we provide support for business processes in a flexible and relatively simple way. Services are composed in a particular order and introduce a set of rules and an order in which a business process executes. Business processes in SOA are a composition of service invocations in a certain order with rules that influence the execution and other constructs, such as parallel invocations, transformations of data, dependencies, and correlations. Another important aspect of business processes are fault handlers, event handlers, and compensation handlers.

Service composition enables us to modify business processes quickly and therefore provide support to changed requirements faster and with less effort. Only when we reach the level of service composition can we realize all the benefits of SOA.

To be successful with service composition, we first have to understand what a business service is. We have already seen that the notion of service is integral to designing service-oriented architectures. Even more important than technical characteristics of services is the way we design the service and the granularity of operations a service exposes through the business interface. Understanding what a business service is becomes important for composing such services into processes and realizing all the advantages SOA is promising:

- Loose coupling
- Easier changes and reduced maintenance costs
- Faster development and improved flexibility
- Resilience to change in the future

Business services in an enterprise are those activities that are performed in order to fulfill a request, whether internal or external. Business services are a reaction to customer requests and always perform an action that leads to a business result. Activities that have not been requested by customers are not considered business services.

To be usable in business processes, business services should expose operations that provide business results to internal or external customers. From the business perspective, services must expose operations that make business sense, not technical sense. Web services can be business services, if they are designed according to the above-mentioned rules. In other words, not all web services are business services. Business services are the centerpiece of a composition of services into business processes. Such composition can be done in two different ways: orchestration or choreography.

Orchestration and Choreography

Service composition can follow two different patterns, orchestration or choreography.

Orchestration follows the notion of a central process, which takes control over the involved business services and coordinates the execution of different operations on the services involved in the process. The involved services do not know and do not need to know that they are involved in an orchestrated process. Only the central coordinator of the orchestration knows this, so the orchestration is centralized with explicit definitions of operations and the order of invocation of business services. This centralized orchestration is called an executable business process.

Choreography, on the other hand, does not rely on a central coordinator. Rather, each business service involved in the choreography knows exactly when to execute its operations and whom to with interact. Choreography is a collaborative effort focused on the exchange of messages in business processes. All participants of the choreography need to be aware of the business process: operations to execute, messages to exchange, and the timing of message exchanges.

Complexity of Business Services

In SOA, business services can have different complexities. The complexity can range from simple services to complex services that are composed of different underlying services. Such composed services can reach the complexity of business processes. The separation between a service and a process is not straightforward and is related to the context in which a service or process is used. For example, a business process can be orchestrated from several services. Such a process can then be used in another, even more complex business process where it can represent only one business service.

The separation of business services and processes is therefore context driven. Therefore, from the technology perspective, the differentiation is also not obvious. Services and processes expose their operations in the same way, using the same interfaces in WSDL (Web Services Description Language).

Business services can be of different complexity:

- Discrete services are services that expose operations that are discrete. Discrete operations are those that complete within a single interaction. They are executed as a whole, without parts. Discrete services can be synchronous or asynchronous. Examples of discrete services are posting messages, returning quotes, getting information (such as an invoice, purchase order, etc.) from back-end application systems, etc.
- Composite services or orchestrated services are services that expose operations that require more than one interaction to complete.

Identifying Business Services

Business services can be identified in two different ways:

- Top-down or
- Bottom-up

In the top-down approach, business services are identified as activities of business processes. This approach is process-centric and requires good knowledge of the process itself. Therefore, it helps if the business process has been modeled in advance. Modeling business processes can be done in various different ways, using notations such as EPC (Event Process Chain) or BPMN (Business Process Modeling Notation), and using different methodologies, such as ARIS (Architecture of Integrated Information Systems). ARIS is one of the enterprise modeling methods. It is widely used for analyzing processes and taking a holistic view of process design, management, workflow, and application processes.

Irrespective of the notation and methodology, it is necessary to be familiar with the process to identify the services. Services identified using the top-down approach are usually composite business services and have to be further decomposed into discrete business services. Discrete business services then map to technology services, which are either exposed from existing systems or developed from scratch.

The bottom-up approach does not rely on the process. Rather it looks at the existing applications, tries to figure out what functionality is available, and how this functionality could be exposed for application access. This approach is directly opposite to the top-down approach. The functionalities exposed from existing systems are usually technology oriented, thus services in the first step are not business services yet. Usually, several technology services have to be combined into discrete business services. Such discrete business services can in the next step be grouped together (composed) into composite or orchestrated services.

In the real world, we will usually use a combination of both approaches and we will develop some business services out of process requirements, and some we will expose from existing systems or develop on our own. Such a combination of top-down and bottom-up approaches is usually called the inside-out approach.

No matter which approach we use, we always have to be aware to develop services with the appropriate granularity, to focus on business services with business operations, and to foster loose-coupling and reuse of services across different processes.

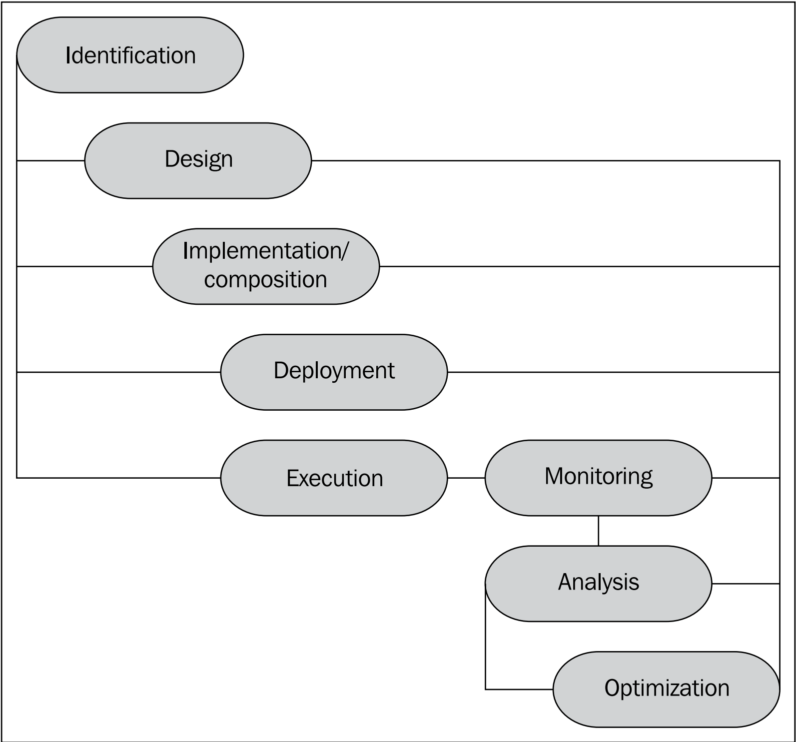
Development Lifecycle

The development lifecycle of services and composed processes usually consists of the following stages:

- Identification
- Design
- Implementation/composition

- Deployment
- Execution
- Monitoring
- Analysis and optimization

The relations between the various stages are shown in the following figure:



In the identification stage, we identify the business logic and flows, and develop a unified view on activities and back-end systems that are involved into the realization of a business service (either discrete or composite). Activities, events, tasks, roles, data in information flows, and functional logic are identified and a joint picture is obtained.

The design stage is focused on the definition of sound service architecture. It is focused on a single service, on the interaction of the service with other services, and on the overall service architecture. SOA is about developing a long-term architecture; therefore each service has to positively contribute to the overall architecture.

The implementation/composition phase realizes the design from the previous stage. It uses languages and technologies together with techniques to implement the service. In SOA, for the design and implementation stage, composition techniques are used. These techniques compose, decompose, combine, and transform existing artifacts or services to develop a new composite service.

The deployment phase is not only related to the actual deployment of the service to the application server, but also relates to versioning, change management, and notifications. In the real-world environment, we will often be faced with many different services that will evolve over time. This means that the implementation will change and that the interface will be modified from time to time. Versioning and related change management takes care that various clients will use the appropriate versions and that clients will not fail if there is a change in the contract of behavior of a service. Notification takes care of notifying the different clients (users) of a service that a change has occurred.

Execution of services is related to run-time behavior, through instance management, but also through logging, reliability, security, and other QoS aspects a service needs to have. An SOA platform can provide huge support in these activities. At the same time, an execution environment also has to provide administrative tools that simplify administration and related tasks.

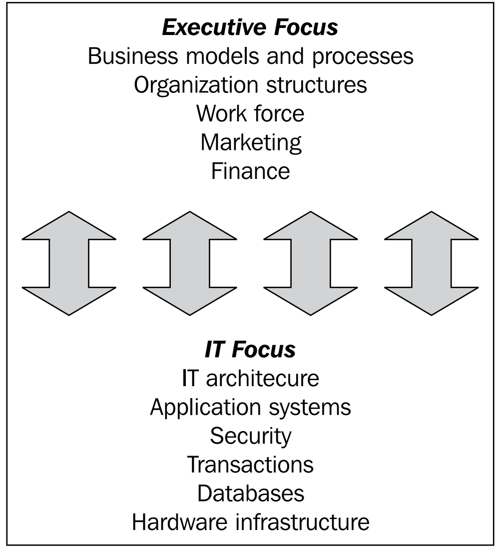
Monitoring is particularly important, as it provides the possibility to observe the run-time behavior. When we think about services as business entities, such monitoring can provide quantitative data about service execution times, number of requests, level of fulfillment, and other business activity monitoring results.

The analysis and optimization stage uses the results from the monitoring stage to analyze the service behavior and performance, and applies changes. If we talk about composed business services, at a certain level, such services represent business processes. Then we can talk about optimization of processes. We also have to consider that each service that we optimize also results in the optimization of all other composed services that reuse this service.

In the next section, we will look at executable business processes and will discuss the relations between composed business services and business processes.

SOA and Executable Business Processes

The SOA approach to service composition into business processes minimizes the semantic gap between the business processes on one side and the IT (applications and software) on the other side. The approach of service composition into executable business processes provides an opportunity to develop processes that can actually be executed. This means that processes are not "nice pictures" anymore, but are represented as code. This is schematically shown in the following figure:



Because business processes become executable code, we need a language for them. Although processes could be defined in traditional programming languages, such as Java or C#, SOA introduces new language for this purpose: BPEL (Business Process Execution Language). BPEL differs considerably from Java, C# and similar languages. In contrast to them (also called programming-in-the-small languages), BPEL is a programming-in-the-large language, specifically designed for service composition and focused on business processes. Before we dig into the details of BPEL, let us first discuss the other aspects of executable business processes.

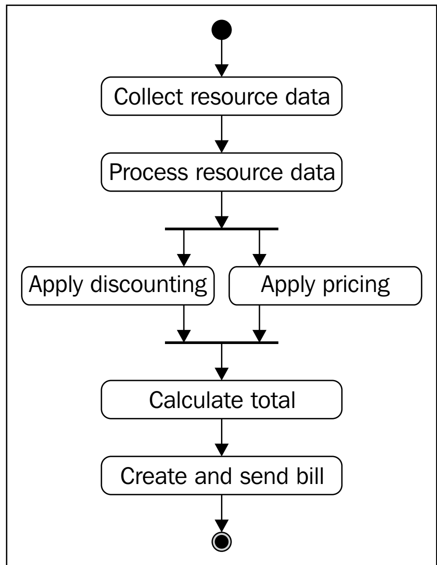
In SOA, business processes do not differ from composite business services. From the outside, both expose their operations through WSDL interfaces. In the bottom-up approach, we could say that we compose services until the aggregate services provide support for the whole business processes. Business processes are defined as collections of activities through which services are invoked.

For the clients, an executable business process looks like any other service. In real-world scenarios, we will usually create two kinds of business processes: those that will contain services from within enterprise only, and those that will consume services provided by several companies. An executable business process in SOA is a collection of coordinated service invocations and related activities that produces a business result, either within a single organization or across several.

Example Business Process

Consider an example business of a Telco operator for billing. This example, although simplified, defines the billing process as a series of activities.

The billing process first collects the resource data for a specific customer. Then it processes the resource data. Next, the pricing and discounting are applied. These last two activities can be applied in parallel. Finally, the total sum for the bill is calculated, and the bill is created and sent. The process flow is shown in the following figure:



We will model the activities of this process as business services. In our example, we will define three business services:

- The Resource Service will handle resource data collection and resource data processing.
- The Rating Service will handle record rating including pricing and discounting.
- The Billing Service will handle calculation of the total sum and bill creation and delivery.

Seen from the perspective of the business process, it is completely irrelevant how the three business services are implemented. We have already mentioned that business services will typically use web services technologies, and describe the interface in WSDL. The business process also does not care how the web service is implemented. It could be an exposed legacy system, a newly developed service, a service that connects to the database and uses stored procedures, etc. Typically, the three business services will be composite services and will probably reuse other, lower-level services. At this level of abstraction, this is not important.

In a similar way, the client of our business process will access the process through a WSDL interface. To the client, our process will look like any other service. The client will not care whether the process is implemented through composition of other services, or in some other way. This stimulates reuse and further composition.

Real-world business processes will usually be much more complicated than our example. Usually, they will contain several services and invoke their operations either in sequence or in parallel. They will also contain flow logic, handle faults, take care of transactions, perform message correlation, etc.

BPEL for Service Composition

To implement business processes, like the one in our example, we will use a special language, called BPEL (Business Process Execution Language, also WS-BPEL or BPEL4WS).

We could implement processes using one of the well-known programming languages, such as Java or C# for service composition. However, composition of services differs in some ways from traditional programming. With composition, we compose services into larger processes (or composite services). It refers to representation of the high-level state transition logic of a system. Traditional programming languages have not been designed for these purposes. Using them for business process composition is too complex, because developers have to deal with too many low-level issues, instead of focusing on orchestration. Therefore, such attempts usually result in inflexible solutions, particularly because there is no clear separation between the process flow and the business logic, which should not be tightly coupled.

Composition of services into processes also has some other specific requirements. One of them is support for several concurrent process instances. Another is support for processes that execute for a longer period of time (days, weeks, even months). For such long-running processes, it is particularly important to provide a transactional semantic, which is often done using compensating transactions. Finally, there is the need for correlation, event handling, asynchronous operation invocations, callbacks, and several other things. In all the mentioned topics, BPEL is superior to traditional programming languages and is therefore the de facto choice for developing business processes in SOA.

What We Can Do with BPEL

BPEL is the language for defining business processes that compose services and externalize their functionality as services. BPEL is an XML-based language and is based on WSDL, XML Schema, and XPath. From a historical perspective, BPEL can be seen as a convergence of IBM WSFL (Web Services Flow Language) and Microsoft XLANG.

To define a business process collaboration, activities have to be defined, and message exchange with involved services has to be specified. WSDL provides the basic technical description and specifications for messages that are exchanged, but does not go into the details of collaboration and interactions. BPEL therefore defines all those aspects that are not defined in WSDL. It is used to describe complex compositions of multiple services, which usually consist of several messages exchanged in a well-defined order. In such complex compositions, synchronous and asynchronous messages can be combined; interactions are usually long running, and often involve state information that has to be preserved. An important aspect is also the ability to describe how to handle faults and other exceptional situations.

The most important part of BPEL is activities (constructs) for invoking services. Synchronous and asynchronous invocations of services are supported in a straightforward way. The same holds true for callbacks. BPEL allows sequential or parallel service invocations. It provides fault-handling mechanisms, which is very important for developing robust business processes that need to react to failures in a smart way. BPEL also provides support for long-running process and compensation, which allows undoing partial work done by a process that has not finished successfully, and for message correlation, event handling, and other specific features. Listed below are the most important features that BPEL provides:

- Describing the logic of business processes through composition of services
- Composing larger business processes out of smaller processes and services
- Handling synchronous and asynchronous (often long-running) operation invocations on services, and managing call-backs that occur at later times

- Invoking service operations in sequence or parallel
- Selectively compensating completed activities in case of failures
- Maintaining multiple long-running transactional activities, which are also interruptible
- Resuming interrupted or failed activities to minimize work to be redone
- Routing incoming messages to the appropriate processes and activities
- Correlating requests within and across business processes
- Scheduling activities based on the execution time and define their order of execution
- Executing activities in parallel and defining how parallel flows merge based on synchronization conditions
- Structuring business processes into several scopes
- Handlling message-related and time-related events

BPEL also offers constructs, such as loops, branches, variables, assignments, etc. These constructs are very similar to the ones in traditional programming languages and allow us to define business processes in an algorithmic way. BPEL is a specialized language for business process compositions. It offers constructs that make the definition of processes relatively simple. Still, it is less complex than traditional programming languages.

Executable and Abstract Processes

BPEL can be used to define executable business processes. Such processes can be executed on a BPEL Engine. This is usually a BPEL process server. Today, there is a variety of commercial and open-source BPEL servers available. Please refer to the book Business Process Execution Language for Web Services by Poornachandra Sarang, Benny Mathew, and Matjaz Juric from Packt Publishing for more information.

Executable BPEL processes comprise a set of existing services and specify the exact order of activities. With BPEL executable processes, we are able to specify the exact algorithm of service composition in a relatively simple and straightforward way, and execute it.

In most cases, BPEL is used to define executable business processes.

For the client, the executable BPEL process is nothing else but a web service that is a composition of existing services. The interface of the BPEL process is defined in WSDL and specifies a set of port types, through which it provides operations, like any other web service. To invoke an executable BPEL process, we have to invoke the resulting web service.

BPEL can also be used for defining abstract processes. Abstract processes are not executable, because they do not define all the details of the process. Rather, they specify the public message exchange between parties or the externally observable aspects of process behavior. The description of the externally observable behavior of a business process may be related to a single service or a set of services. It might also describe the behavior of a participant in a business process.

Using BPEL for defining abstract processes is quite rare in real-world projects. This is mainly because not many tools support BPEL abstract processes. There are two major scenarios for using abstract processes: to describe the behavior of a service without knowing exactly in which business process it will take part; and to define collaboration protocols among multiple parties, and precisely describe the external behavior of each party. In both cases, such an abstract process can be seen as a template to define executable processes.

BPEL and Other Process Languages

Although BPEL has become the de facto standard for specifying business processes in SOA, it is not the only language for this purpose. BPEL also complements with some other languages and technologies.

Predecessors of BPEL include the following languages:

- XLANG and the new version XLANG/s from Microsoft
- WSFL (Web Services Flow Language) from IBM
- WSCL (Web Services Conversation Language) from HP, submitted to W3C
- BPSS (Business Process Specification Schema), part of the ebXML framework

Languages for Choreography

The languages that address choreography are to some extent related to BPEL abstract processes. These languages that can be seen as complementary to BPEL, include:

- WSCI (Web Services Choreography Interface), co-developed by Sun, SAP, BEA, and Intalo and submitted to W3C
- WS-CDL (Web Services Choreography Description Language), at the time of writing a W3C Working Draft

Modeling Notations

Finally, there are business process modeling notations. These are used to develop process models on a higher abstraction level. Usually, process models in these languages are the basis for implementation of processes in BPEL. The most important modeling notations are:

- EPC (Event-Driven Process Chain) and eEPC (Extended Event-Driven Process Chain)
- BPMN (Business Process Modeling Notation)

BPMN defines how business process models in BPMN can be automatically translated to executable BPEL processes. This makes them very interesting for using within SOA. We will not go into the details of process modeling in this book.

In the next sections, we will focus on executable BPEL processes and will demonstrate how to develop, deploy, and use a BPEL process.

Writing BPEL Processes

Each BPEL process consists of an activity. Activities are steps of the process. BPEL activities can be primitive, such as web service invocation, variable assignment, fault indication, etc. BPEL activities can also be structured. A structured activity can represent a sequence, a parallel flow, a decision, loop, etc.

BPEL is in XML language, therefore it uses the XML syntax to write code. In this book, we will show BPEL code; however, if you are using a graphical BPEL designer, you might prefer to use the visual development, where you can drag and drop various activities into the BPEL process without writing the code directly.

The most important primitive BPEL activities are:

- `<invoke>`, used to invoke operations on web services
- `<receive>`, used to wait for an incoming message request, for example from the client to start the business process, or for waiting for the response from a web service
- `<reply>` to generate a synchronous response, most often as a result of a completed process
- `<assign>` for assigning variables and partner links
- `<sequence>` to define activities that will be invoked in an ordered sequence
- `<flow>` to define activities that will be invoked in parallel
- `<switch>` to implement branches

- `<while>` to define loops
- `<pick>` to select one of a number of alternative paths
- `<throw>` to indicate faults
- `<faultHandler>` to define the fault handlers; this is a set of activities that executes when a fault occurs
- `<scope>` to group activities into logical scopes
- `<wait>` to delay the execution of a process for a specific period or time
- `<terminate>` to terminate the entire process

Process Interface

We have already mentioned that for the clients, BPEL processes look like any other web service. In other words, BPEL processes are exposed as web services. This means that we need to define a WSDL interface for each BPEL process.

A client will usually invoke an operation on the BPEL process to start it. With the BPEL process WSDL, we specify the interface for this operation. We also specify all message types, operations, and port types a BPEL offers to other partners. Most often a BPEL process will define only one operation, which will start the process. Starting a BPEL process results in creation of a new process instance.

Similar as any other web service, BPEL process operation can be synchronous or asynchronous. In other words, a BPEL process can be synchronous or asynchronous. Synchronous request/reply processes are those where clients send requests and wait for replies. We decide on a synchronous process if a process does not require much time to process, therefore it is reasonable for the sender (client) to wait for the reply.

Asynchronous processes do not require the client to wait for the result. This is reasonable for processes that requires a longer time to finish, or where it is difficult to foresee how long the processing will last. Asynchronous processes do not block the client for the duration of the operation. Results from asynchronous processes are usually sent back to the client using callbacks. Callbacks usually need to be related to original requests. This is called message correlation.

Synchronous and asynchronous processes differ in two main characteristics:

- How they return results—both synchronous and asynchronous processes first wait for the initial message, using a `<receive>` or `<pick>` activity. Both also invoke other web services, either synchronously or asynchronously. A synchronous BPEL process returns it result after the process has completed, using a `<reply>` activity at the end of the process. An asynchronous BPEL process uses an `<invoke>` activity to invoke the call-back port type. An asynchronous BPEL process, however, does not need to return anything.

- How they declare the interface—the type of the operation (synchronous/asynchronous) is defined in the WSDL interface for the BPEL process. A synchronous operation specifies `<input>` and `<output>` messages (and optional `<fault>` message), while asynchronous behavior usually requires two port types, one for the initial invocation, and the other for the callback. Please notice that the callback port type is implemented by the client.

Partner Links

Probably the most important activity of BPEL processes is invocation of services. At the same time, a BPEL process is a web service itself and receives invocations from its clients. In simple cases, a BPEL process will have only one operation and a client will invoke this operation to start the process.

In BPEL all links to web services—this is all web services that a BPEL process invokes and all clients that invoke operations on the process—are called partner links. Each BPEL process has at least one client partner link, because there has to be a client that invokes the BPEL process to start it.

A BPEL process may also have some invoked partner links. Invoked partner links are links to web services, that are called from the BPEL process using the `<invoke>` activity. Most likely, each BPEL process will have several invoked partner links, although this is not necessary. If a BPEL process does not invoke any related web service, it might have zero invoked partner links. Such a BPEL process is called a discrete process.

If a BPEL process uses asynchronous web services, then an invoked partner might become a client partner link. This happens with asynchronous operations that return results using callbacks. The process invokes an operation on the invoked web service. Later the service can invoke the call-back operation on the process to return the requested data.

Partner Link Types

To describe the roles of all involved web services, clients, and the process itself from an independent perspective, BPEL introduces partner link types. They allow us to model relationships as a third party. We are not required to take a certain perspective; rather we just define roles. A partner link type must have at least one role and can have at most two roles. It has one role if we use synchronous communication or if there is no callback from an asynchronous invocation. Otherwise, it has two roles. For each role, we must specify a port type, which is used for interaction.

Partner link types are not part of the BPEL process specification document. They are defined in the WSDLs of participating web services and in the WSDL of the BPEL process. This is reasonable, because partner link types belong to the service specification and not the process specification. Partner link types use the WSDL extensibility mechanism, so they can be a part of a WSDL document.

Variables

Variables are used to store messages that are exchanged between BPEL process partners (web services) or to hold data that relates to the state of the process. Messages are exchanged as operations are invoked. When the business process invokes an operation and receives the result, we often want to store that result for subsequent invocations, to use the result as is or extract certain data. We store the result in a BPEL variable.

Variables can also hold data that relates to the state of the process, but will never be exchanged with partners. Specifically, variables can store WSDL messages, XML Schema elements, or XML Schema simple types. Each variable has to be declared before it can be used. When we declare a variable, we must specify the variable name and type. To specify type, we have to specify one of the following attributes:

- `messageType`: A variable that can hold a WSDL message
- `element`: A variable that can hold an XML Schema element
- `type`: A variable that can hold an XML Schema simple type

The declaration of variables is gathered within the `<variables>` element.

Handlers

BPEL processes have three type of handlers:

- Fault handlers are invoked when a fault occurs during the run time of a BPEL process.
- Event Handlers allow a process to react to external events. These can be message events, which are related to the invocation of operations on the BPEL process by the clients; or alarm events, which can occur at a certain time or represent a duration.
- Compensation handlers define the compensating activities to undo the partial results of a process that did not complete successfully.

Fault Handlers

SOA is based on the concept of loose coupling. The communication between web services is done over Internet connections that may or may not be highly reliable. Web services could also raise faults due to logical errors and execution errors arising from defects in the infrastructure. Therefore, BPEL business processes will need to handle faults appropriately. Faults in BPEL can arise in various situations. A fault can occur when:

- A BPEL process invokes a synchronous web service operation. The operation might return a WSDL fault message, which results in a BPEL fault.
- A BPEL process can explicitly signal (throw) a fault.
- A fault can be thrown automatically, when a certain execution criterion has not been fulfilled (for example, when a join failure has occurred due to an unfulfilled join condition. For more information about join conditions please refer to the book *Business Process Execution Language for Web Services, 2nd Edition,* published by Packt Publishing, 2006).
- The BPEL process server can encounter an error condition at run time or in network communications, etc. BPEL defines several standard faults that are thrown automatically.

When a fault occurs within a business process (this can be a WSDL fault, a fault thrown by the BPEL process, or any other type of fault), it means that the process may not complete successfully. The process can complete successfully only if the fault is handled within the BPEL process (inside a scope). A business process can handle a fault through one or more fault handlers. Within a fault handler, the business process defines custom activities that are used to recover from the fault and recover the partial (unsuccessful) work of the activity in which the fault has occurred.

Event Handlers

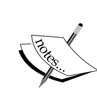
A BPEL process may have to react to certain events. In most business processes, we will need to react to two types of events:

- **Message events**: These are triggered by incoming messages through operation invocation on port types.
- **Alarm events:** These are time related and are triggered either after a certain duration or at a specific time.

Event handlers allow a BPEL process to react to events that occur while the business process executes. In other words, we do not want the business process to wait for the event (and do nothing else but wait). Instead, the process should execute, and still listen to events and handle them whenever they occur. If the corresponding events occur, event handlers are invoked concurrently with the business process. Typical usage of event handlers is to handle a cancellation message from the client.

Compensation Handler

Compensation, or undoing steps in the business process that have already completed successfully, is one of the most important concepts in business processes. The goal of compensation is to reverse the effects of previous activities that have been carried out as part of a business process that is being abandoned. Compensation is related to the nature of most business processes, which are long running and use asynchronous communication with loosely coupled partner web services. Business processes are often sensitive in terms of successful completion because the data they manipulate is sensitive. Because they usually span multiple partners (often multiple enterprises) special care has to be taken that business processes either fully complete their work or that the partial (not fully completed) results are undone—compensated.



Compensation differs from fault handling. In fault handling, a business process tries to recover from an activity that could not finish normally because an exceptional situation has occurred. The objective of compensation on the other hand, is to reverse the effects of a previous activity or a set of activities that have been carried out successfully as part of a business process that is being abandoned.

To define compensation activities, BPEL provides compensation handlers. Compensation handlers gather all activities that have to be carried out to compensate another activity. Compensation handlers can be defined:

- For the whole process
- For the scope
- Inline for the `<invoke>` activity

Compensation handlers can be invoked only after the activity that is to be compensated has completed normally. If we try to compensate an activity that has completed abnormally, nothing will happen because an `<empty>` activity will be invoked.

Now that we have become familiar with fault, event, and compensation handlers, let us look at another important BPEL construct—scopes.

Scopes

Scopes provide a way to divide a complex business process into hierarchically organized parts—*scopes*. Scopes provide behavioral contexts for activities. They allow us to define different fault handlers for different activities (or sets of activities). In addition to fault handlers, scopes also provide a way to declare variables that are visible only within the scope. Scopes also allow us to define local correlation sets, compensation handlers, and event handlers.

Each scope has a primary activity. This is similar to the overall process structure where we have said that a BPEL process also has a primary activity. The primary activity, often a `<sequence>` or `<flow>`, defines the behavior of a scope for normal execution. Fault handlers and other handlers define the behavior for abnormal execution scenarios.

With scopes we can also control concurrency. We will need such control if more than one process instance uses shared variables concurrently. This can occur, for example, if we use an event handler through which we react to an event while the main process is executing. Scopes that require concurrency control are called **serializable scopes**. In serializable scopes, access to all shared variables is serialized; in other words, concurrency is prohibited. This guarantees that there will be no conflicting situations if several concurrent scopes access the same set of shared variables. Conflicting operations are in this case all read/write and write-only activities, such as assignments, storing incoming messages in variables, etc.

Overview of BPEL Activities

The following table provides an overview of different BPEL activities with a brief description. Look at the table to get an idea what a BPEL process can include. For more information please refer to *Business Process Execution Language for Web Services* from Packt Publishing:

BPEL Activity	Description
`<assign>, <copy>`	Copy data from one variable to another. Construct and insert new data using expressions and literal values. Copy partner link endpoint references.
`<catch>, <catchAll>`	Specified within fault handlers to specify faults that are to be caught and handled. The `<catchAll>` activity is used to catch all faults.
`<compensate>`	To invoke a compensation handler. The `<compensate>` activity has an optional scope attribute that can be used to specify the compensation handler to be invoked.

BPEL Activity	Description
`<compensationHandler>`	Compensation handlers are used to define compensation activities. Compensation handlers gather all activities that have to be carried out to compensate another activity.
`<correlations>`, `<correlation>`	To associate a correlation set (a collection of key data fields) with an activity. Correlation can be used within the `<receive>`, `<reply>`, `<invoke>`, and `<onMessage>` activities.
`<correlationSets>`, `<correlationSet>`	A correlation set is a set of properties shared by messages and used for correlation. It is used to associate a message with a business process instance.
`<empty>`	An activity that does nothing.
`<eventHandlers>`	Event handlers react to events that occur while the business process is executing. When these events occur, the corresponding event handlers are invoked. Event handlers can be specified for the whole process as well as for each scope.
`<faultHandlers>`, `<faultHandler>`	Fault handlers are used to react to faults that occur while the business process activities are executing. They can be specified for the global process or each scope, or inline for `<invoke>` activities. Multiple `<catch>` activities can be specified within the fault handler for specific faults. You need to specify at least one `<catch>` activity. You can optionally specify the `<catchAll>` activity.
`<flow>`	Provides concurrent execution of enclosed activities and their synchronization.
`<invoke>`	To invoke the web service operations provided by partners.
`<links>`, `<link>`	Synchronization dependencies in concurrent flows are specified using links.
`<onAlarm>`	This activity is used in the `<pick>` and `<eventHandlers>` activities to specify the occurrence of alarm events.
`<onMessage>`	Used in `<pick>` and `<eventHandlers>` activities to specify the occurrence of message events.
`<partnerLinks>`, `<partnerLink>`	A business process interacts with services that are modeled as partner links. Each partner link is characterized by a `<partnerLinkType>`.
`<partnerLinkType>`, `<role>`	A partner link type characterizes the relationship between two services. It defines roles for each of the services in the conversation between them and specifies the port type provided by each service to receive messages. Partner link types and roles are specified in the WSDL.

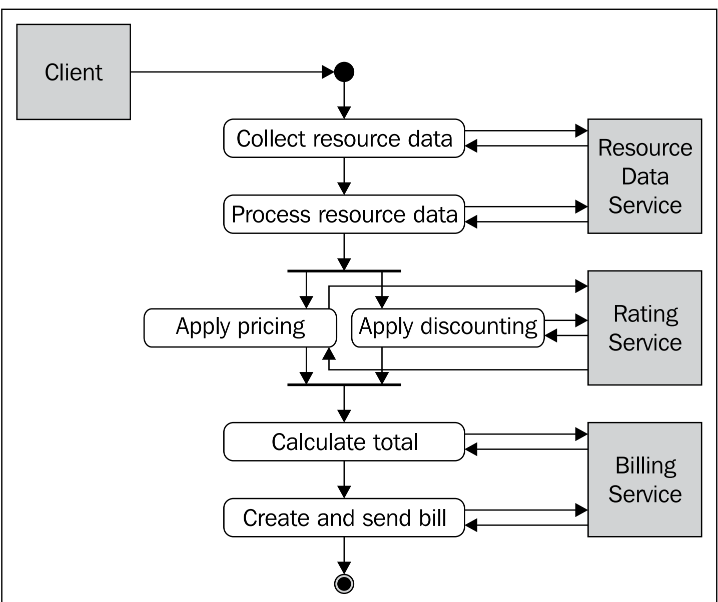
BPEL Activity	Description
`<partners>`	The partner element is used to represent the capabilities required from a business partner. A partner is defined as a set of partner links.
`<pick>`	To wait for the occurrence of one of a set of events and then perform an activity associated with the event.
`<process>`	This is the root element of each BPEL process definition.
`<property>`	Properties are used to create globally unique names and associate them with data types (XML Schema types). They are defined in WSDL.
`<propertyAlias>`	Property aliases are used to map global properties to fields in specific message parts. Property aliases are defined in the WSDL.
`<receive>`	To receive requests in a BPEL business process to provide services to its partners.
`<reply>`	To send a response to a request previously accepted through a `<receive>` activity. Responses are used for synchronous request/reply interactions.
`<scope>`	Scopes define behavior contexts for activities. They provide fault handlers, event handlers, compensation handlers, data variables, and correlation sets for activities.
`<sequence>`	To define activities that need to be performed in a sequential order.
`<source>`	To declare that an activity is the source of one or more links.
`<switch>, <case>`	To express conditional behavior. It consists of one or more conditional branches defined by `<case>` elements, followed by an optional `<otherwise>` element. The case branches of the switch are considered in alphabetical order.
`<target>`	To declare that an activity is the target of one or more links.
`<terminate>`	To immediately terminate a business process instance.
`<variables>,` `<variable>`	Variables are used to hold messages that constitute the state of a business process. A variable may be of the WSDL message type, an XML Schema simple type, or an XML Schema element.
`<wait>`	To specify a delay for a certain period of time or until a certain deadline is reached.
`<while>`	To define an iterative activity. The iterative activity is performed until the specified Boolean condition no longer holds true.

In the next section, we will develop an example process to demonstrate how to develop processes in BPEL.

Developing an Example BPEL Process

In this section, we will develop the Telco billing process that we introduced earlier in this chapter. Let's repeat what the process does: it first collects the resource data for a specific customer. Then it processes the resource data. Next, the pricing and discounting are applied. Finally, the total sum for the bill will be calculated, and the bill will be created and sent.

Our process together with access to web services is shown in the following figure:



We will develop an asynchronous BPEL process. To develop it we will go through the following steps:

- First, we will get familiar with the involved services. Here we will:
    - Study the WSDL interface of each service
    - Add partner link types
- Second, we will define the WSDL interface for the process.
- Third, we will write the process logic.
- Fourth, we will add fault handlers and other handlers (event handlers, compensation handler) if necessary.

Services Used in the Process

Understanding the interfaces of the services that we will call from the BPEL process is crucial. Therefore, it is important to study the interfaces and understand the behavior of the services and their operations. In our example, we will use the following three services:

- The Resource Data Service, which will handle resource data collection and resource data processing.
- The Rating Service, which will handle record rating including pricing and discounting.
- The Billing Service, which will perform calculation of the total sum and bill creation with delivery.

The web services and the BPEL process example can be downloaded from `http://www.packtpub.com`.

For this example, we have developed the three web services. In real-world applications, we would perform similarly. We would develop the three services; however, we would consider how to connect these services to existing systems in order to reuse the functionality inside existing systems. Reusing functionality through exposing it as web services is a very important step in designing SOA. Here it is important to do this in several layers. Usually, we develop technical services that directly expose the functionality of existing systems. On top of technical services, we develop business services. From the BPEL process, we always invoke business services. This approach ensures loose-coupling and improves flexibility.

To study the involved services, we will have to look at the WSDL definitions. WSDL specifies the operations and port types web services offer, the messages they accept, and the types they define.

Resource Data Service

The Resource Data Service provides two business operations:

- Operation `CollectData` is used to collect call data for a specific customer.
- Operation `ProcessData` is used to process the call data for a specific customer and order it by different tariff rates.

Both operations are asynchronous. We see this from the WSDL `<portType>` definition, where we only see `<input>` messages:

```
<definitions name="ResourceData"
             targetNamespace="http://packtpub.com/ResourceData"
             xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:client="http://packtpub.com/ResourceData"
             xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/
partner-link/">
   ...

   <portType name="ResourceData">
      <operation name="CollectData">
         <input message="client:CollectDataRequestMessage"/>
      </operation>
      <operation name="ProcessData">
         <input message="client:ProcessDataRequestMessage"/>
      </operation>
   </portType>
```

Results are returned by using callbacks. This means that a second `<portType>` is defined in the WSDL. This callback port type is implemented by the client. It includes callbacks for both operations: `CollectDataOnResult` and `ProcessDataOnResult`. It also includes the `OnFault` operation, which is used to report possible faults on the service side that would prevent the service from fulfilling the request. In asynchronous callback scenarios, this is the usual way to signal faults. The code excerpt below shows the callback port type definition:

```
   <portType name="ResourceDataCallback">
      <operation name="CollectDataOnResult">
         <input message="client:CollectDataResponseMessage"/>
      </operation>
      <operation name="ProcessDataOnResult">
         <input message="client:ProcessDataResponseMessage"/>
      </operation>
      <operation name="OnFault">
         <input message="client:FaultMessage"/>
      </operation>
   </portType>
```

CollectData Operation

To understand the operation, for the messages that have to be sent as input for the operation, and for the messages that are send as results (in callbacks), we have to look at the `<message>` definitions. For the `CollectData` operation, we look at the `CollectDataRequestMessage` message. The `CollectDataOnResult` uses the `CollectDataResponseMessage`:

```
<message name="CollectDataRequestMessage">
   <part name="payload" element="client:CollectDataRequest"/>
</message>
<message name="CollectDataResponseMessage">
   <part name="payload" element="client:CollectDataResponse"/>
</message>
```

We can seen that both messages have single parts, called payloads. They use XML elements to specify the payload. To get familiar with the structure of the messages, we have to look at the corresponding XML Schema.

This web service uses an external XML Schema, which is good, because it decouples the data formats from the interface specifications. WSDL imports the schema:

```
<types>
   <schema xmlns="http://www.w3.org/2001/XMLSchema">
      <import namespace="http://packtpub.com/ResourceData"
                 schemaLocation="ResourceData.xsd" />
   </schema>
</types>
```

From the schema, we can see that the `CollectData` operation uses the following XML element for input. The input consists of the customer ID:

```
<element name="CollectDataRequest">
   <complexType>
       <sequence>
         <element name="CustomerID" type="string"/>
       </sequence>
   </complexType>
</element>
```

The output for the `CollectData` operation consists of a list of call items. Each item has date and time information, duration, and tariff ID. Here, a named complex type is defined first. This is because we will reuse this type as the input for another operation:

```
<complexType name="CollectedData">
      <sequence>
              <element name="Item" maxOccurs="unbounded">
                  <complexType>
                      <sequence>
                          <element name="DateTime" type="dateTime"/>
                          <element name="Duration" type="int"/>
                          <element name="TariffId" type="string"/
                     </sequence>
                  </complexType>
             </element>
        </sequence>
</complexType>
<element name="CollectDataResponse" type="tns:CollectedData"/>
```

ProcessData Operation

In a similar way, we would look at the `ProcessData` operation. The `ProcessData` operation takes as input the output from the `CollectData` operation and processes the call items to group them by different tariffs. Both relevant messages are:

```
<message name="ProcessDataRequestMessage">
    <part name="payload" element="client:ProcessDataRequest"/>
</message>
<message name="ProcessDataResponseMessage">
    <part name="payload" element="client:ProcessDataResponse"/>
</message>
```

The XML schema for the `ProcessDataRequest` and `ProcessDataResponse` elements is shown below. We can see that the `ProcessDataRequest` uses the same type as the output for the `CollectData` operation:

```
<element name="ProcessDataRequest" type="tns:CollectedData"/>

<element name="ProcessDataResponse">
    <complexType>
        <sequence>
          <element name="ProcessedData" maxOccurs="unbounded">
                 <complexType>
                    <sequence>
                      <element name="TariffId" type="string"/>
```

```
                        <element name="TotalDuration" type="int"/>
                      </sequence>
                  </complexType>
          </element>
      </sequence>
  </complexType>
</element>
```

OnFault Operation

Finally, let's look at the `OnFault` operation. This operation will be used throughout the web services and the BPEL process to signal faults. Standardizing fault signaling makes sense, because it is much easier to develop fault handling capabilities if we have a unified way to report faults. Shown below is the message declaration and the element structure, where we can see that the fault has a code and a description:

```
<message name="FaultMessage">
   <part name="fault" element="client:FaultElement"/>
</message>
 <element name="FaultElement">
         <complexType>
             <sequence>
                 <element name="Code" type="string"/>
                 <element name="Description" type="string"/>
             </sequence>
         </complexType>
 </element>
```

Next, let's look at the Rating Service.

Rating Service

The Rating Service also provides two business operations:

- Operation `ApplyPricing` is used to apply tariff prices on the call durations, gathered by the Resource Data Service.
- Operation `ApplyDiscounting` is used to apply discounting on the call data.

Both operations are asynchronous. We see this from the WSDL `<portType>` definition, where we only see `<input>` messages:

```
<portType name="Rating">
   <operation name="ApplyPricing">
      <input message="rd:ProcessDataResponseMessage"/>
   </operation>
   <operation name="ApplyDiscounting">
```

```
        <input message="rd:ProcessDataResponseMessage"/>
      </operation>
</portType>
```

Results are returned by using callbacks. Similar to the previous service, a callback port type is defined. It includes callbacks for both operations: `ApplyPricingOnResult` and `ApplyDiscountingOnResult`. It also includes the `OnFault` operation, which is used to report possible faults. The code excerpt below shows the callback port type definition:

```
<portType name="RatingCallback">
   <operation name="ApplyPricingOnResult">
      <input message="client:PricingResponseMessage"/>
   </operation>
   <operation name="ApplyDiscountingOnResult">
      <input message="client:DiscountingResponseMessage"/>
   </operation>
   <operation name="OnFault">
      <input message="rd:FaultMessage"/>
   </operation>
</portType>
```

ApplyPricing Operation

Let us look at the messages that are used by the `ApplyPricing` operation. We can see that for input the operation uses the output from the Resource Data Service, more exactly, from the `ProcessData` operation. This means that we only have to look at the output message:

```
<message name="PricingResponseMessage">
    <part name="payload" element="client:PricingResponse"/>
</message>
```

The XML Schema for the `PricingReponse` element is shown below. We can see that this is a list of call items with added pricing information:

```
<element name="PricingResponse">
      <complexType>
          <sequence>
              <element name="ProcessedData" maxOccurs="unbounded">
                 <complexType>
                     <sequence>
                       <element name="TariffId" type="string"/>
                         <element name="TotalDuration" type="int"/>
                              <element name="PriceBeforeDiscount"
```

```
                                    type="float"/>
                            </sequence>
                        </complexType>
                    </element>
                </sequence>
            </complexType>
    </element>
```

ApplyDiscounting Operation

The `ApplyDiscounting` operation is similar to `ApplyPricing`, except that it applies discounting to the call items. As input, the operation uses the output from the Resource Data Service. This means that we only have to look at the output message:

```
<message name="DiscountingResponseMessage">
    <part name="payload" element="client:DiscountingResponse"/>
</message>
```

The XML Schema for the `DiscountingResponse` element is shown below. We can see that this is a list of call items with added pricing information:

```
<element name="DiscountingResponse">
      <complexType>
          <sequence>
              <element name="ProcessedData" maxOccurs="unbounded">
                     <complexType>
                         <sequence>
                            <element name="TariffId" type="string"/>
                            <element name="TotalDuration" type="int"/>
                            <element name="Discount" type="float"/>
                         </sequence>
                     </complexType>
              </element>
          </sequence>
      </complexType>
</element>
```

We will not look at the `OnFault` operation again, because it is the same as for the Resource Data Service. Next, let us look at the Billing Service.

Billing Service

The Billing Service provides two business operations too:

- Operation `CalculateTotal` calculates the final price for combined tariff call durations.
- Operation `CreateSendBill` creates the bill (whether electronically or on paper) and sends it to the customer.

As with the previous service, both operations in the Billing Service are asynchronous too. We see this from the WSDL `<portType>` definition, where we only see `<input>` messages:

```
<portType name="Billing">
    <operation name="CalculateTotal">
       <input message="client:CalculateTotalRequestMessage"/>
    </operation>
    <operation name="CreateSendBill">
      <input message="client:CreateSendBillRequestMessage"/>
    </operation>
</portType>
```

Results are returned by using callbacks. The callback port type defines callbacks for both operations: `CalculateTotalOnResult` and `CreateSendBillOnResult`. It also includes the `OnFault` operation, which is used to report possible faults. The code excerpt below shows the callback port type definition:

```
<portType name="BillingCallback">
   <operation name="CalculateTotalOnResult">
      <input message="client:CalculateTotalResponseMessage"/>
   </operation>
   <operation name="CreateSendBillOnResult">
      <input message="client:CreateSendBillResponseMessage"/>
   </operation>
   <operation name="OnFault">
      <input message="rd:FaultMessage"/>
   </operation>
</portType>
```

CalculateTotal Operation

The `CalculateTotal` operation takes as its input the results from both previous operations in the Rating Service and returns the final pricing. Please notice that the combination of results of `ApplyPricing` and `ApplyDiscounting` operations is achieved using an input message with two parts, as shown in the code excerpt below:

```
<message name="CalculateTotalRequestMessage">
   <part name="pricing" element="rt:PricingResponse"/>
   <part name="discounting" element="rt:DiscountingResponse"/>
</message>
<message name="CalculateTotalResponseMessage">
   <part name="payload" element="client:CalculateTotalReponse"/>
</message>
```

The XML Schema for the `CalculateTotal` element `CalculateTotalReponse` is shown below. We can see that this is a list of call items with added final pricing information:

```
<element name="CalculateTotalReponse">
    <complexType>
        <sequence>
            <element name="ProcessedData" maxOccurs="unbounded">
                    <complexType>
                        <sequence>
                            <element name="TariffId" type="string"/>
                            <element name="TotalDuration" type="int"/>
                            <element name="PriceAfterDiscount"
                                     type="float"/>
                        </sequence>
                    </complexType>
            </element>
        </sequence>
    </complexType>
</element>
```

CreateSendBill Operation

The `CreateSendBill` operation creates the bill and sends it to the customer. Let's look at the input and the output messages:

```
   <message name="CreateSendBillRequestMessage">
        <part name="bill" element="client:CalculateTotalReponse"/>
       <part name="customer" element="client:CreateSendBillRequest"/>
   </message>
   <message name="CreateSendBillResponseMessage">
                <part name="payload" element="client:
CreateSendBillResponse"/>
   </message>
```

We can see that the input message `CreateSendBillRequestMessage` has two parts. The first part, bill, is actually the output from the previous `CalculateTotal` operation. The second part, customer, includes the customer ID, which is used to retrieve the address and to specify how the bill will be delivered (electronically, on paper, etc.).

The XML Schema for the `CreateSendBillRequest` and `CreateSendBillResponse` elements is shown below:

```
<element name="CreateSendBillRequest">
    <complexType>
        <sequence>
            <element name="CustomerID" type="string"/>
        </sequence>
    </complexType>
</element>
<element name="CreateSendBillResponse">
    <complexType>
      <sequence>
         <element name="SendDateTime" type="dateTime"/>
      </sequence>
    </complexType>
</element>
```

The `OnFault` operation is the same as for the Resource Data Service and Rating Service. With this, we have concluded the study of the involved web services. In the next step, we will add partner link types.

Adding Partner Link Types to the Service's WSDL

In the next step, we will add partner link types to the service WSDL definitions. Usually, the services that we will invoke from the BPEL process will not have the partner link types defined. To add them, we have two options:

- Add the partner link types directly into the WSDL
- Wrap the original WSDL into a new document, import the original WSDL, and add partner link types

Both approaches are quite straightforward. The first is applicable if we own and control the service. In real-world scenarios, where we will reuse existing services, the second approach is more realistic.

As all three services are asynchronous, our partner link types will have two roles. Let's look at all three definitions.

Partner Link Type for Resource Data Service

The partner link type for the Resource Data Service specifies the `ResourceDataProvider` and `ResourceDataRequester`, and connects both roles with the appropriate port types, as shown in the code excerpt below:

```
<plnk:partnerLinkType name="ResourceData">
   <plnk:role name="ResourceDataProvider">
      <plnk:portType name="client:ResourceData"/>
   </plnk:role>
   <plnk:role name="ResourceDataRequester">
      <plnk:portType name="client:ResourceDataCallback"/>
   </plnk:role>
</plnk:partnerLinkType>
```

Partner Link Type for Rating Service

The partner link type for the Rating Service specifies the `RatingProvider` and `RatingRequester` and connects both roles with the appropriate port types, as shown in the code excerpt below:

```
<plnk:partnerLinkType name="Rating">
    <plnk:role name="RatingProvider">
       <plnk:portType name="client:Rating"/>
    </plnk:role>
    <plnk:role name="RatingRequester">
       <plnk:portType name="client:RatingCallback"/>
    </plnk:role>
</plnk:partnerLinkType>
```

Partner Link Type for Billing Service

The partner link type for the Billing Service specifies the `BillingProvider` and `BillingRequester` and connects both roles with the appropriate port types, as shown in the code excerpt below:

```
<plnk:partnerLinkType name="Billing">
   <plnk:role name="BillingProvider">
      <plnk:portType name="client:Billing"/>
   </plnk:role>
   <plnk:role name="BillingRequester">
      <plnk:portType name="client:BillingCallback"/>
   </plnk:role>
</plnk:partnerLinkType>
```

With this we have prepared everything necessary to start developing the BPEL process. Therefore, in the next section, we will define the WSDL for the BPEL process.

Define a WSDL Interface for the BPEL Process

We already know that BPEL processes are exposed as web services. For the client there is no difference between a BPEL process and a web service, as they both use a WSDL interface description to define the contract for the client.

The Billing Process will be asynchronous; therefore, it will have to define two port types. The first port type will be to initiate the process and the second to return results, or to alternatively signal faults. The input to the process will be the customer identification. The output from the process will be the date and the time at which the bill has been created and sent.

To define the WSDL for the BPEL process, let's start with the header:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="BillingProcess"
             targetNamespace="http://packtpub.com/BillingProcess"
             xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:client="http://packtpub.com/BillingProcess"
             xmlns:bl="http://packtpub.com/Billing"
             xmlns:rt="http://packtpub.com/Rating"
             xmlns:rd="http://packtpub.com/ResourceData"
             xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/
partner-link/">
```

The port type for process initiation will define a single `Initiate` operation:

```
   <portType name="BillingProcess">
      <operation name="Initiate">
         <input message="client:BillingProcessRequestMessage"/>
      </operation>
   </portType>
```

The callback port type will define two operations. The `OnResult` operation will be used to signal process competition and return the date and time of bill creation and sending. The `OnFault` operation will be used to signal faults to the clients. The `OnFault` operation uses the same message type and element definition as are used by the web services. The code excerpt below shows the callback port type definition:

```
   <portType name="BillingProcessCallback">
       <operation name="OnResult">
          <input message="client:BillingProcessResponseMessage"/>
       </operation>
```

```
    <operation name="OnFault">
       <input message="rd:FaultMessage"/>
    </operation>
</portType>
```

As the BPEL process will reuse several messages from the web services, we will import all three services' WSDL definitions:

```
<import location="http://localhost:8888/orabpel/default/
  ResourceData/1.0/ResourceData?wsdl"
        namespace="http://packtpub.com/ResourceData"/>
<import location="http://localhost:8888/orabpel/default/
  Rating/1.0/Rating?wsdl"
        namespace="http://packtpub.com/Rating"/>
<import location="http://localhost:8888/orabpel/default/
  Billing/1.0/Billing?wsdl"
        namespace="http://packtpub.com/Billing"/>
```

Next, we will define the input (request) and the output (response) messages:

```
<message name="BillingProcessRequestMessage">
   <part name="payload" element="rd:CollectDataRequest"/>
</message>

<message name="BillingProcessResponseMessage">
   <part name="payload" element="bl:CreateSendBillResponse"/>
</message>
```

The XML Schema for both elements has been imported through the WSDL import from the involved web services. We do not show it again.

Finally, we have to define the partner link types for the BPEL process. As the process is asynchronous, we have to define two roles, `BillingProcessProvider` and `BillingProcessRequester`:

```
<plnk:partnerLinkType name="BillingProcess">
    <plnk:role name="BillingProcessProvider">
       <plnk:portType name="client:BillingProcess"/>
   </plnk:role>
   <plnk:role name="BillingProcessRequester">
       <plnk:portType name="client:BillingProcessCallback"/>
   </plnk:role>
</plnk:partnerLinkType>
```

With this, we have concluded the WSDL definition of the BPEL process. In the next section, we will start writing the BPEL process logic.

Writing the BPEL Process Logic

The BPEL process will define the order of the activities, that have to be performed to do the billing. The BPEL process first waits for the incoming message, which creates a new process instance and starts the execution of the process. This incoming message is usually the client request. Then a series of activities occur, either sequentially or in parallel. These activities include:

- Operation invocations on web services
- Callbacks are received from web services
- Loops and branches, which influence the flow of the process

Before we start writing our billing process, let's have a quick look again at the activity diagram that we have shown earlier in this chapter.

The billing process involves the following parties:

- The client, which will invoke the BPEL process
- The BPEL process itself
- The Resource Data web service
- The Rating web service
- The Billing web service

The client will start the BPEL process by sending an input message that includes the customer identification. A new process instance will be generated. Then the `CollectData` operation on the Resource Data Service will be invoked. The output from this operation will be used as the input for the second operation, the `ProcessData` operation that will group the calls by tariff rates. Next, the pricing and discounting will be applied in parallel. For this two operations will be called on the Rating Service: `ApplyPricing` and `ApplyDiscounting`. Next, the total bill sum will be calculated. This is achieved invoking the `CalculateTotal` operation on the Billing Service. Finally, the bill will be created and sent to the customer. For this final activity, the `CreateSendBill` operation on the Billing Service is invoked.

Now, we are ready to start writing the BPEL process definition. Each BPEL definition contains at least four main parts:

- The initial `<process>` root element with the declaration of namespaces
- The definition of partner links, using the `<partnerLinks>` element
- The declaration of variables, using the `<variables>` element
- The main body where the actual business process is defined, which is usually a `<sequence>` that specifies the flow of the process

Process Declaration

First, let us look at the overall BPEL process outline. Each BPEL process will have at least the following parts:

- Declaration of partner links
- Declaration of variables
- Process sequence

Later in this chapter, we will also add fault handlers and other elements. For now, let us stick with the simple process outline, shown below:

```
<process name="BillingProcess" ... >
   <partnerLinks>
      <!-- The declaration of partner links -->
   </partnerLinks>
   <variables>
      <!-- The declaration of variables -->
   </variables>
   <sequence>
      <!-- The definition of the BPEL business process main body -->
   </sequence>
</process>
```

We start writing the BPEL process with the `<process>` declaration, which specifies the process name and declares various XML namespaces that will be used. The declaration is shown below. The BPEL activity namespace must be `http://schemas.xmlsoap.org/ws/2003/03/business-process/`:

```
<?xml version = "1.0" encoding = "UTF-8" ?>
<process name="BillingProcess"
         targetNamespace="http://packtpub.com/BillingProcess"
         xmlns="http://schemas.xmlsoap.org/ws/2003/03/
          business-process/"
         xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/
          business-process/"
         xmlns:xsd="http://www.w3.org/2001/XMLSchema"
         xmlns:ns1="http://packtpub.com/Billing"
         xmlns:ns2="http://packtpub.com/Rating"
         xmlns:ns3="http://packtpub.com/ResourceData"
         xmlns:client="http://packtpub.com/BillingProcess">
...
```

Next, we will define the partner links.

Defining Partner Links

Partner links define different parties that interact with the BPEL process. Each partner link is related to a specific `partnerLinkType` that characterizes it. We have defined partner link types earlier in this chapter. Each partner link also specifies up to two attributes:

- `myRole`: Indicates the role of the business process itself
- `partnerRole`: Indicates the role of the partner

The first partner link is called `Client` and is characterized by the `BillingProcess` partner link type. The `Client` invokes the business process:

```
...
   <partnerLinks>
        <partnerLink name="Client" partnerLinkType="client:
BillingProcess"
                     myRole="BillingProcessProvider"
                     partnerRole="BillingProcessRequester"/>
...
```

Next, we need to specify the partner links for the three involved web services. As we will invoke two different asynchronous operations on each service, it makes sense to declare two different partner links for each web service. The reason is that asynchronous operations require callback message correlation. This correlation is achieved using WS-Addressing.

For the Resource Data Service, we will define the following two partner links: `CollectResourceData` and `ProcessResourceData`. The definitions are shown below:

```
...
        <partnerLink myRole="ResourceDataRequester" name=
                                                 "CollectResourceData"
                     partnerRole="ResourceDataProvider"
                     partnerLinkType="ns3:ResourceData"/>

        <partnerLink myRole="ResourceDataRequester" name=
                                                 "ProcessResourceData"
                     partnerRole="ResourceDataProvider"
                     partnerLinkType="ns3:ResourceData"/>
...
```

For the Rating Service, we will define the following two partner links: `Pricing` and `Discounting`. The definitions are shown below:

```
...
        <partnerLink myRole="RatingRequester" name="Pricing"
                     partnerRole="RatingProvider"
                     partnerLinkType="ns2:Rating"/>
        <partnerLink myRole="RatingRequester" name="Discounting"
                     partnerRole="RatingProvider"
                     partnerLinkType="ns2:Rating"/>
...
```

For the Billing Service, we will define the following two partner links: `BillingTotal` and `BillSend`. The definitions are shown below:

```
...
        <partnerLink myRole="BillingRequester" name="BillingTotal"
                     partnerRole="BillingProvider"
                     partnerLinkType="ns1:Billing"/>
        <partnerLink myRole="BillingRequester" name="BillSend"
                     partnerRole="BillingProvider"
                     partnerLinkType="ns1:Billing"/>
...
```

Next, we will declare variables.

Declaring Variables

Variables are used to store messages and to reformat and transform them. We usually need a variable for every message sent to the partner services and received from the partner services. We also need a variable for the initial input message, which initiates the process, and one for the final output message. And then, there is also the variable for the faults.

In our example, we will structure the BPEL process into several scopes. We already know that scopes can declare local variables. We could define local variables for input and output messages for each scope. However, our process uses output messages from some operations as input messages to other operations. That is why we will declare output (result) messages globally. We will declare input messages locally to scopes later.

We can see that we will need two variables, one for the input and one for the output message. Then, we will need six variables to store the results from all six operation invocations (on the three services). Finally, we will need one variable to store faults.

For each variable, we have to specify the type. We can use a WSDL message type, an XML Schema simple type, or an XML Schema element. In our example, we use WSDL message types for all variables:

```
<variables>
    <variable name="InputVariable"
              messageType="client:BillingProcessRequestMessage"/>
    <variable name="OutputVariable"
              messageType="client:BillingProcessResponseMessage"/>
    <variable name="CollectData_Result"
              messageType="ns3:CollectDataResponseMessage"/>
    <variable name="ProcessData_Result"
              messageType="ns3:ProcessDataResponseMessage"/>
    <variable name="ApplyPricing_Result"
              messageType="ns2:PricingResponseMessage"/>
    <variable name="ApplyDiscounting_Result"
              messageType="ns2:DiscountingResponseMessage"/>
    <variable name="CalculateTotal_Result"
              messageType="ns1:CalculateTotalResponseMessage"/>
    <variable name="CreateSendBill_Result"
              messageType="ns1:CreateSendBillResponseMessage"/>
    <variable name="OnFault_Variable" messageType="ns3:
              FaultMessage"/>
</variables>
```

Now, we are ready to start writing the main process definition.

Writing the Process Definition

The process definition is the heart of each BPEL process. It specifies the activities that have to be carried out. The process main body can contain only one top-level activity. Usually, this is a `<sequence>` that allows us to define several activities that will be performed sequentially. Other possibilities for this activity include `<flow>`, through which several activities can be performed concurrently. We can also specify `<while>` to indicate loops, or `<scope>` to define nested activities. However, we usually use `<sequence>` and nest other activities within the sequence.

Within the sequence, we first specify the input message that starts the business process. We do this with the `<receive>` construct, which waits for the matching message. In our case, this is the `BillingProcessRequestMessage` message. Within the `<receive>` construct, we specify the variable name and the partner link, and not the message directly.

To start our Billing process, we link the message reception with the Client partner link, and wait for the `Initiate` operation to be invoked on port type `BillingProcess`. We store the received message into the `InputVariable` variable:

```
<sequence name="main">
    <receive name="receiveInput" partnerLink="Client"
             portType="client:BillingProcess" operation="Initiate"
             variable="InputVariable" createInstance="yes"/>
```

This receive activity will create a new process instance. This is why we specify the `createInstance` attribute and set it to yes.

We will structure our process in six scopes. Scopes provide a way to divide a complex business process into hierarchically organized parts. For our Billing process, we will use one scope for each activity. We will define the following scopes:

- CollectResourceData
- ProcessResourceData
- ApplyPricing
- ApplyDiscounting
- CalculateTotal
- CreateSendBill

Let us now look at these six scopes into more detail.

CollectResourceData Scope

In this scope, we will invoke the Resource Data Service, the `CollectData` operation. For this, we will use the `<invoke>` activity. Before this, we have to prepare the input for this operation. Looking at the WSDL, we can see that we have to send a message that consists of the customer ID. We can construct such a message by copying the data from the initial input variable. To do the copying, we will use the `<assign>` activity. We will also need to declare a local variable to store the input message. This is shown in the code excerpt below:

```
<scope name="CollectResourceData">
   <variables>
       <variable name="CollectData_InputVariable"
                 messageType="ns3:CollectDataRequestMessage"/>
   </variables>
   <sequence>
       <assign name="PrepareInputForRD">
           <copy>
               <from variable="InputVariable" part="payload"
```

```
                      query="/ns3:CollectDataRequest/ns3:CustomerID"/>
                         <to variable="CollectData_InputVariable"
                             part="payload"
                      query="/ns3:CollectDataRequest/ns3:CustomerID"/>
                     </copy>
                 </assign>
```

Next, we will invoke the `CollectData` operation:

```
                 <invoke name="CallRD"
                         partnerLink="CollectResourceData"
                         portType="ns3:ResourceData"
                         operation="CollectData"
                         inputVariable="CollectData_InputVariable"/>
```

As this is an asynchronous operation, we will also need to wait for the callback. We wait for the callback using a `<receive>` actvitiy:

```
                 <receive portType="ns3:ResourceDataCallback"
                          operation="CollectDataOnResult"
                          variable="CollectData_Result"
                          partnerLink="CollectResourceData"
                          name="Callback" createInstance="no"/>
```

ProcessResourceData Scope

We can already see that we will follow the same pattern in basically all scopes:

- First, we declare the scope.
- Then we declare the local variable to hold the input message for operation invocation on the web service.
- Then we prepare the input variable using `<assign>` and `<copy>` activities.
- Next we invoke the operation on the web service using an `<invoke>` activity.
- Finally, we wait for the callback using a `<receive>` activity.

Let us look at the code for the `ProcessResourceData` scope:

```
         <scope name="ProcessResourceData">
             <variables>
                 <variable name="ProcessData_InputVariable"
                          messageType="ns3:ProcessDataRequestMessage"/>
             </variables>
             <sequence>
                 <assign name="PrepareInputForRD">
                     <copy>
```

```
                    <from variable="CollectData_Result"
                         part="payload"
                          query="/ns3:CollectDataResponse"/>
                    <to variable="ProcessData_InputVariable"
                        part="payload"
                        query="/ns3:ProcessDataRequest"/>
                </copy>
            </assign>
            <invoke name="CallRD"
                    partnerLink="ProcessResourceData"
                    portType="ns3:ResourceData"
                    operation="ProcessData"
                    inputVariable="ProcessData_InputVariable"/>
            <receive name="Callback"
                    portType="ns3:ResourceDataCallback"
                    operation="ProcessDataOnResult"
                    variable="ProcessData_Result"
                    partnerLink="ProcessResourceData"
                    createInstance="no"/>
        </sequence>
    </scope>
```

ApplyPricing and ApplyDiscounting Scopes

The next two scopes also follow the steps described in the previous section. The difference is, however, that the `ApplyPricing` and `ApplyDiscounting` operations will be called in parallel. To achieve parallel invocation in BPEL, we use the `<flow>` activity.

We will group both scopes under the `<flow>` activity and thus achieve that both scopes will execute simultaneously. This is often the case in business processes and is very useful. When we close the `</flow>` activity, all parallel flows are synchronized. The code excerpt below shows both scopes and the `<flow>` activity:

```
<flow name="Flow">
    <sequence name="Sequence">
        <scope name="ApplyPricing">
           <variables>
                <variable name="ApplyPricing_InputVariable"
                   messageType="ns3:ProcessDataResponseMessage"/>
            </variables>
            <sequence name="Sequence">
                <assign name="PrepareInput">
                   <copy>
```

```
                        <from variable="ProcessData_Result"
                              part="payload"
                             query="/ns3:ProcessDataResponse"/>
                        <to variable="ApplyPricing_InputVariable"
                            part="payload"
                            query="/ns3:ProcessDataResponse"/>
                    </copy>
                </assign>
                <invoke name="InvokePricing" partnerLink="Pricing"
                        portType="ns2:Rating"
                        operation="ApplyPricing"
                      inputVariable="ApplyPricing_InputVariable"/>
                <receive name="Callback"
                         portType="ns2:RatingCallback"
                         operation="ApplyPricingOnResult"
                         variable="ApplyPricing_Result"
                         partnerLink="Pricing"
                         createInstance="no"/>
            </sequence>
        </scope>
    </sequence>
    <sequence name="Sequence">
        <scope name="ApplyDiscounting">
            <variables>
                 <variable name="ApplyDiscounting_InputVariable"
                    messageType="ns3:ProcessDataResponseMessage"/>
            </variables>
            <sequence name="Sequence">
                <assign name="PrepareInput">
                    <copy>
                        <from variable="ProcessData_Result"
                              part="payload"
                              query="/ns3:ProcessDataResponse"/>
                        <to
                         variable="ApplyDiscounting_InputVariable"
                            part="payload"
                            query="/ns3:ProcessDataResponse"/>
                    </copy>
                </assign>
                <invoke name="InvokeDiscounting"
                        partnerLink="Discounting"
                        portType="ns2:Rating"
                        operation="ApplyDiscounting"
                  inputVariable="ApplyDiscounting_InputVariable"/>
```

```
                <receive name="Callback"
                         portType="ns2:RatingCallback"
                         operation="ApplyDiscountingOnResult"
                         variable="ApplyDiscounting_Result"
                         partnerLink="Discounting"
                         createInstance="no"/>
            </sequence>
        </scope>
    </sequence>
</flow>
```

CalculateTotal and CreateSendBill Scopes

Now that we are already familiar with the structure of the scopes, let us have a quick look at the final two scopes. The `CalculateTotal` scope is shown below:

```
<scope name="CalculateTotal">
    <variables>
        <variable name="CalculateTotal_InputVariable"
                  messageType="ns1:CalculateTotalRequestMessage"/>
    </variables>
    <sequence name="Sequence">
        <assign name="PrepareInputForBilling">
            <copy>
                <from variable="ApplyPricing_Result"
                      part="payload"
                      query="/ns2:PricingResponse"/>
                <to variable="CalculateTotal_InputVariable"
                    part="pricing"
                    query="/ns2:PricingResponse"/>
            </copy>
            <copy>
                <from variable="ApplyDiscounting_Result"
                      part="payload"
                      query="/ns2:DiscountingResponse"/>
                <to variable="CalculateTotal_InputVariable"
                    part="discounting"
                    query="/ns2:DiscountingResponse"/>
            </copy>
        </assign>
        <invoke name="InvokeBilling" partnerLink="BillingTotal"
                portType="ns1:Billing"
                operation="CalculateTotal"
                inputVariable="CalculateTotal_InputVariable"/>
        <receive name="Callback" partnerLink="BillingTotal"
```

```
                 portType="ns1:BillingCallback"
                 operation="CalculateTotalOnResult"
                 variable="CalculateTotal_Result"
                 createInstance="no"/>
    </sequence>
</scope>
```

Finally let us look at the `CreateSendBill` scope:

```
<scope name="CreateSendBill">
    <variables>
        <variable name="CreateSendBill_InputVariable"
                  messageType="ns1:CreateSendBillRequestMessage"/>
    </variables>
    <sequence name="Sequence">
        <assign name="PrepareInputCSB">
            <copy>
                <from variable="InputVariable" part="payload"
                    query="/ns3:CollectDataRequest/ns3:CustomerID"/>
                <to variable="CreateSendBill_InputVariable"
                    part="customer"
                 query="/ns1:CreateSendBillRequest/ns1:CustomerID"/>
            </copy>
            <copy>
                     <from variable="CalculateTotal_Result"
                      part="payload"
                      query="/ns1:CalculateTotalReponse"/>
                <to variable="CreateSendBill_InputVariable"
                    part="bill"
                    query="/ns1:CalculateTotalReponse"/>
            </copy>
        </assign>
        <invoke name="InvokeCSB" partnerLink="BillSend"
                portType="ns1:Billing"
                operation="CreateSendBill"
                inputVariable="CreateSendBill_InputVariable"/>
        <receive name="Callback" partnerLink="BillSend"
                 portType="ns1:BillingCallback"
                 operation="CreateSendBillOnResult"
                 variable="CreateSendBill_Result"
                 createInstance="no"/>
    </sequence>
</scope>
```

Returning a Callback to the Client

We have now almost finished our BPEL process. We only need to return the result to the client. Out Billing BPEL process is asynchronous and returns the result through a callback. The content of the result is the date and time that specifies when the bill has been sent to the customer.

We will return the callback using the `<invoke>` activity. We will call the Client partner link, but otherwise the `<invoke>` will be identical to the invokes that we have used for web services earlier in this chapter. Again, before we can do the `<invoke>`, we have to prepare the result, using the `<assign>` activity:

```
    <assign name="PrepareCallback">
        <copy>
            <from variable="CreateSendBill_Result" part="payload"
                  query="/ns1:CreateSendBillResponse/ns1:SendDateTime"/>
            <to variable="OutputVariable" part="payload"
                query="/ns1:CreateSendBillResponse/ns1:SendDateTime"/>
        </copy>
    </assign>
    <invoke name="callbackClient" partnerLink="Client"
           portType="client:BillingProcessCallback" operation="OnResult"
           inputVariable="OutputVariable"/>
   </sequence>
  </process>
```

With this, we have concluded the development of our Billing BPEL process. We would now be ready to deploy and test the process. However, before we do that, let us add fault and event handlers to the process.

Adding a Fault Handler

Fault handling and signaling is an important aspect of business processes. A BPEL process can handle a fault through one or more fault handlers. Within a fault handler, the process defines custom activities that are used to recover from the fault and recover the partial (unsuccessful) work of the activity in which the fault has occurred, or to signal the fault to the client. In BPEL, faults are not automatically propagated to the clients.

The fault handlers are specified before the first activity of the BPEL process, after the partner links and variables. Fault handlers can be specified globally for the whole process, or for each scope individually.

In our Billing process, we will for the sake of simplicity define a simple global fault handler. The fault handler will catch all faults using `<catchAll>`. It will signal the fault to the client and terminate the process execution. This is the simplest fault handler. In real-world scenarios, you might want to define a more sophisticated fault handler that will catch separate faults individually. Such a fault handler could first try to recover from the fault and signal the fault to the client only after all other possibilities have been exhausted. Usually, you will also want to define a custom fault handler for each scope.

For this example, we will, however, make do with the simple fault handler shown in the code excerpt below:

```
<faultHandlers>
    <catchAll>
        <sequence name="Sequence">
          <invoke name="SignalFault" partnerLink="Client"
            portType="client:BillingProcessCallback"
            operation="OnFault" inputVariable="OnFault_Variable"/>
          <terminate name="Terminate"/>
        </sequence>
    </catchAll>
</faultHandlers>
```

Adding an Event Handler

Business processes often need to react on certain events that occur while the business process executes. For this purpose, BPEL provides event handlers. If the corresponding events occur, event handlers are invoked concurrently to the business process. A typical usage of event handlers is to handle a cancellation message from the client. Another usage is to handle fault notifications. This is exactly the case with our example. Please recall that each callback port type defines an `OnFault` operation that is used to signal faults. We can use an event handler to react to such fault messages.

An event handler is not the only way to react on fault messages. Another option would be to use a `<pick>` activity instead of `<receive>` to wait for callbacks. For more details, please refer to *Business Process Execution Language for Web Services* published by Packt Publishing.

To define an event handler for the `CollectResourceData` scope, we would define it after the variables section:

```
<eventHandlers>
    <onMessage portType="ns3:ResourceDataCallback"
               operation="OnFault" variable="OnFault_Variable"
               partnerLink="CollectResourceData">
        <throw name="ThrowFault"
               faultName="client:ResourceDataFault"
               faultVariable="OnFault_Variable"/>
    </onMessage>
</eventHandlers>
```

This event handler is very simple. It waits for the `OnFault` operation and throws a corresponding fault (using a `<throw>` activity). This fault is then cached by the fault handler that we have defined in the previous section. In real-world examples, you might want to define more complex event handlers.

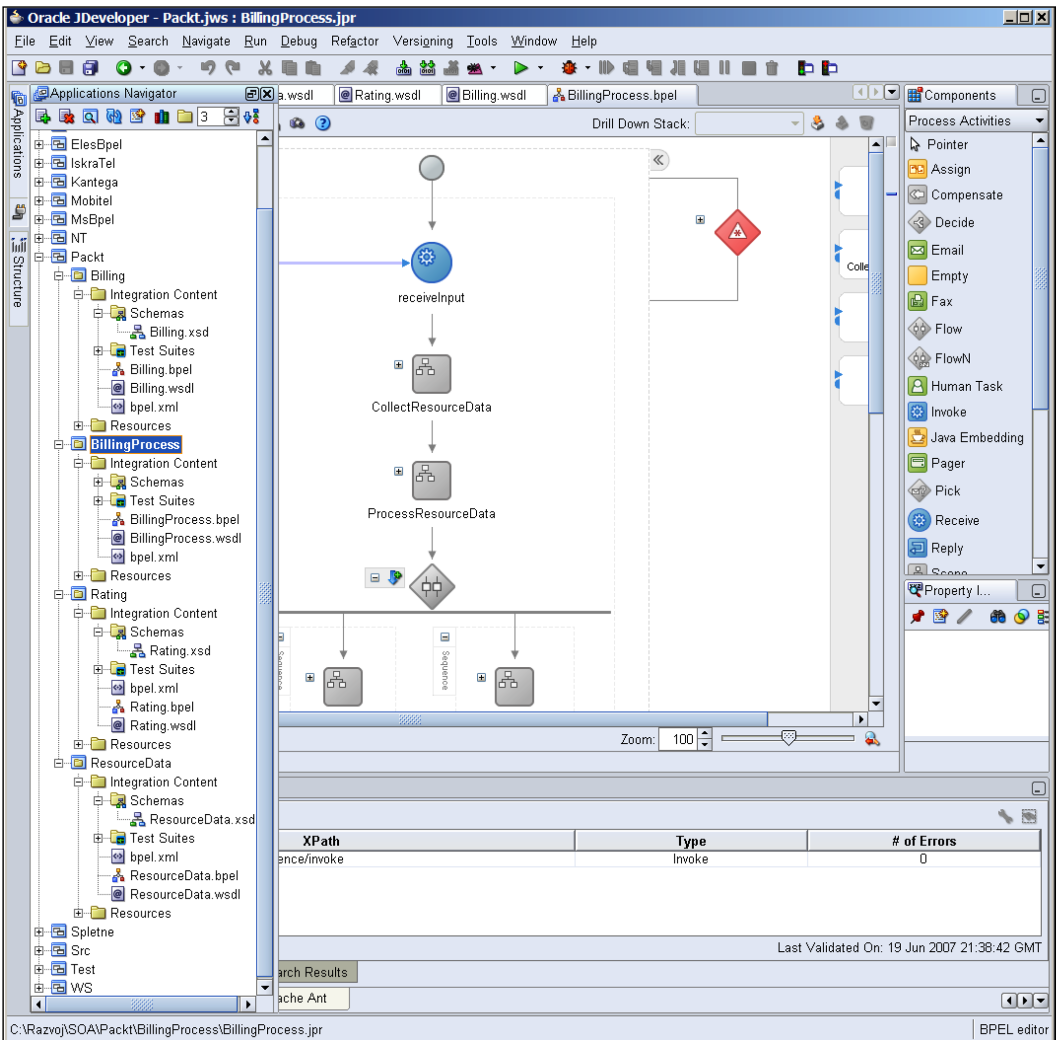
With this, we have concluded the development of the Billing BPEL process. We are now ready to deploy and run the process.

Deploy and Run the Process

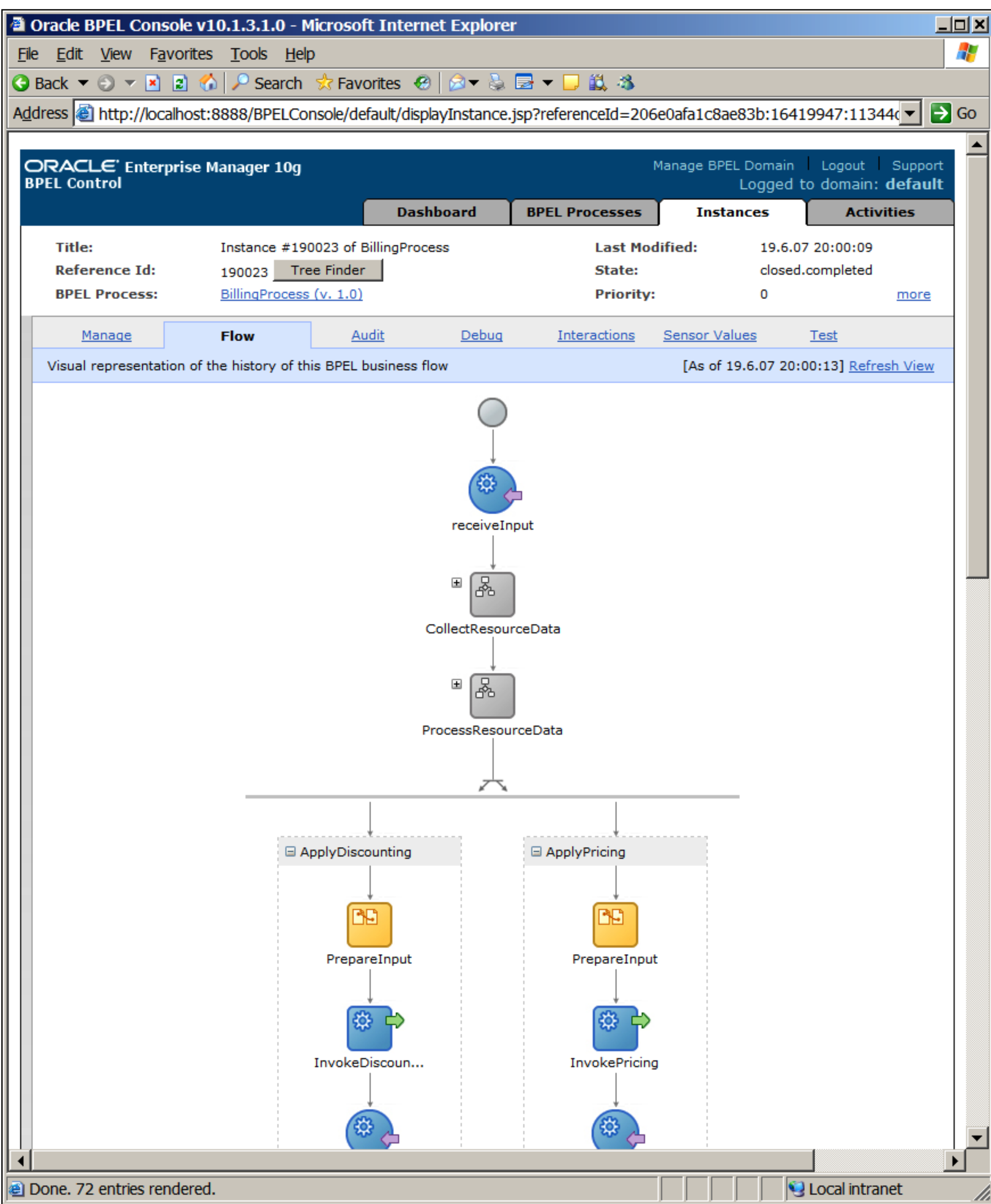
To deploy and test the process, we first have to select the BPEL process server. BPEL is portable, which means that we can deploy this process on any BPEL server that is compliant with the specification. BPEL servers provide a run-time environment for executing BPEL business processes. BPEL is strongly related to web services and to the modern software platforms that support web service development, particularly to Java Enterprise Edition and Microsoft .NET. Most often BPEL servers leverage Java Enterprise Edition or .NET application server environments. The majority of BPEL servers are commercial products from companies such as Oracle, IBM, Microsoft, BEA, Software AG, SAP, etc. There are also several open-source BPEL servers available. For more information, please refer to the vendor websites.

Deployment and execution of BPEL processes is product-related. In this book, we do not go into the details of different products. To give you an idea of how to deploy and run the process, we provide two screenshots of the Oracle SOA Suite.

To develop and deploy the process we could use Oracle JDeveloper BPEL Designer, which provides a nice and convenient environment for graphical development and deployment of BPEL processes, as shown in the following screenshot:



After successful deployment, we can activate the process from the BPEL Console and observe the following visual flow representation:



We can see that the process has completed successfully.

Summary

In this chapter, we have become familiar with the BPEL language and a process-oriented approach to integration. We have discussed the characteristics of the process-oriented integration architectures. A central concept to process-centric SOA in service composition. We have explained the ideas and discussed the differences between orchestration and choreography. We have also talked about identifying business services and service lifecycle, where we have identified various stages.

Then we have explained the role of executable business processes, which reduce the semantic gap between business and IT. We have introduced the most important technology—BPEL. We have explained characteristics of BPEL and identified the differences between executable and abstract processes. We have overviewed the basic BPEL concepts, described how to invoke web services synchronously and asynchronously, and discussed the role of WSDL. We have seen that BPEL processes can be synchronous or asynchronous too. Web services with which a BPEL process interacts are called partner services. Therefore, we have explained the concepts of partner link types and partner links. We have also explained variables and discussed the importance of fault, event, and compensation handlers, and scopes. Finally, we have developed a sample Billing process and demonstrated how to develop a BPEL process efficiently.

6

Service- and Process-Oriented Approach to Integration Using Web Services

Building on the introduction to SOA and the foundations in Web Services and XML, in this chapter we will discuss the more evolved SOA infrastructure option—Enterprise Service Bus. This is a technology for the infrastructure that spans service provider, communication infrastructure, transformation, and routing services and the integration client applications/business processes.

In Chapter 2, we saw how SOA fits into the Integration space and the various technology options available for SOA, including Web Services and ESB. An underlying driver in all technologies is the pervasive usage of XML and the sheer simplicity it offers to exchange data and information between systems. This is further fuelled by the strong support for the Web Service standards in all technologies. In this standards context, the difference between the various options will be in their abilities to handle different services topologies and any additional services-centric functionality that they offer.

In the current global business environment, as enterprises spread across the world, the demands on IT have changed. With increasing complexity of businesses and business relationships, it is imperative that the IT infrastructure scales up with them. This will require additional facilities in the infrastructure such as reliable and scalable communications and high availability of services, and manageability of the services infrastructure. As the services become more widespread in any enterprise, hosting and accessing services will place demands far beyond the connectivity that Web Services offer, or abilities to host the services that Application platforms such as Java EE (hitherto called J2EE) and .NET provide.

In SOA infrastructure, normally a high level of emphasis is placed on the connectivity and service description protocols. While the protocols and the services execution are necessary, the infrastructure that connects the service clients to the service execution environment is equally important, even if there may not be too many standards or specifications in this space. SOA as a concept will work just as well with any of the technologies. However, in a widely distributed enterprise, a higher level of performance and scalability may be expected from the platforms.

Providing a well performing services environment in such distributed enterprises will involve an operating infrastructure that both provides the services runtime and connects the various service providers, service consumers, and processes. Once there is an infrastructure that connects the various ends in the enterprise services environment, one could look at more capabilities such an infrastructure could provide beyond the basic services connectivity.

The Enterprise Service Bus is one such Infrastructure. A technology for the infrastructure that spans service provider and the integration client applications/ consumer ends. In this chapter, we will discuss the Enterprise Service Bus in more detail.

From Just Services to an Enterprise Bus

Market dynamics today causes the landscape of any enterprise to be very dynamic. Even in an organization located physically in a single location with a single data center, this co-location may not remain so for long given the immense M&A activity underway in businesses today. New companies would be acquired, and existing companies merged to create new entities. This results in very fluid organizations. The services infrastructure should be in a position to adapt to this fluidity. The peer-to-peer (P2P) integration solutions such as simple Web Services are probably best suited for organizations with not too many systems talking to each other with low volume of services traffic, in a single or just few locations. The moment that these assumptions are no longer valid, peer-to-peer (P2P) services infrastructure such as Web Services or EAI may not suffice. They are likely to face these constraints:

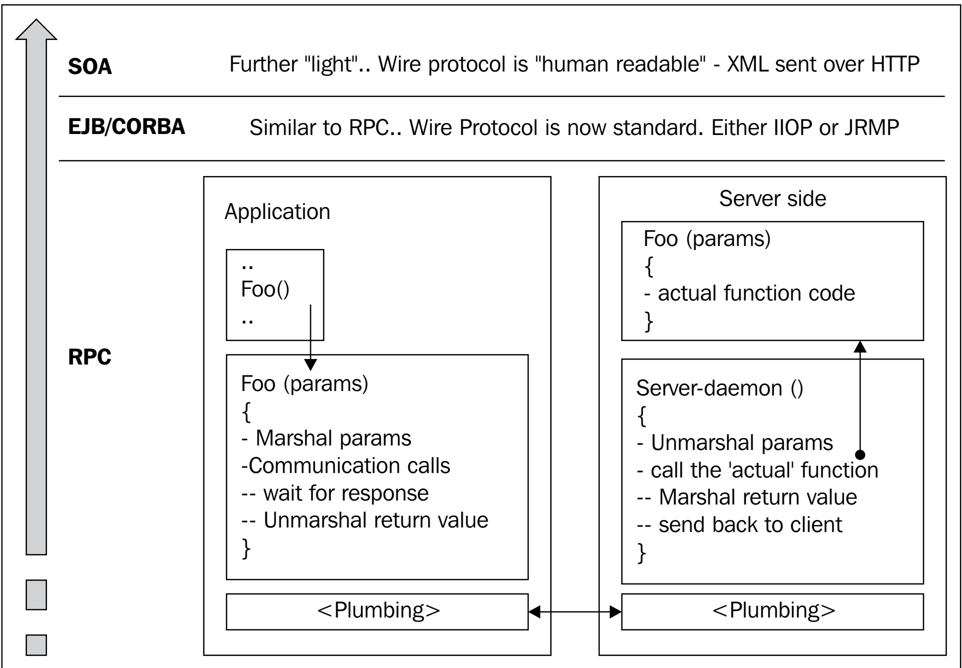
- It will be difficult to assure reliability across WANs, as network performance may be unpredictable.
- RPCs, being synchronous in nature, may face failures/timeout issues servicing service requests due to high latencies.
- In simple point to point connectivity, each request gets its own connection from the origin system to the provider system, which could result in a large number of connections in high services traffic environments.
- Scalability and reliability could come under stress.

"Services" or business components used over the network have come to mean Web Services, mostly. Web Services, as we know today, emerged rapidly with XML and SOAP over HTTP, gaining mainstream traction in IT.

The advent of XML opened up a very different front in approaches to integrating applications—which hitherto was done either using shared databases or EDI solutions. Now XML was a simple, elegant, and portable mechanism to represent data, with XML Schemas providing a powerful mechanism to describe and validate the structure of XML documents. Soon, SOAP followed—a brilliant middleware approach built around XML, using XML to not only represent business data, but also to represent the actual middleware plumbing metadata.

XML is used to represent information that identifies the specific service method to be executed, and the input parameters to that request. Web Services is now the suite of standards and approaches that are built around SOAP for systems to talk to each other.

The most common use of Web Services is for simple services (shown in the following figure)—the point-to-point interactions to "execute" remote services. A service provider makes its service available, and a service consumer accesses the service—by directly connecting to the service provider, say, via SOAP over HTTP. Here the focus is more on connectivity and service access, and not so much on the scalability of the solution space and flexibility in the infrastructure between the service provider and the integration application that accesses the service.



Web services, and the traditional middleware systems such as Java EE (EJB) and .NET, focused on a client application accessing services provided by the Application Server; mostly in a P2P mode. Though distributed services, wherein the application services are distributed across multiple servers and instances, were supported, the fundamental access model was still point-to-point. Even if one of the server instances were to access another server instance, the lowest-level connection is again point-to-point, with one server instance communicating with another.

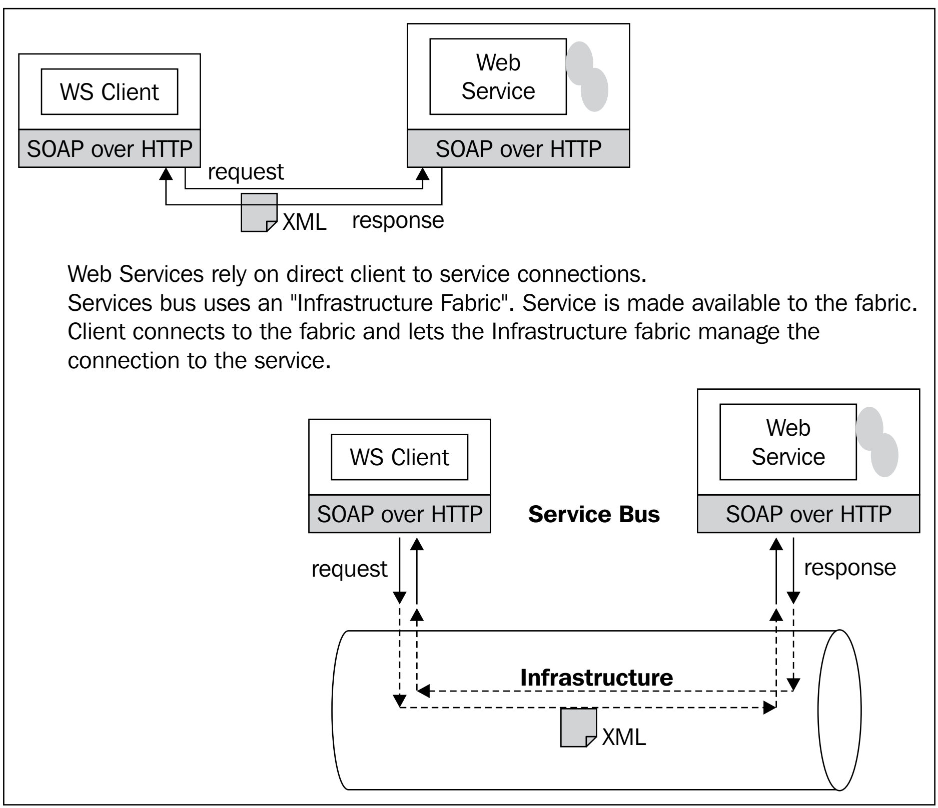
Consider a simple example, say an Order Processing System that is a Java EE application needs to access a Warehouse application, which is another Java EE application. In this case, an Enterprise Java Bean (EJB) component in the Order application will access another EJB in the Warehouse application. Though order application is a Java EE application itself, for the Warehouse application it is just another client, just like a simple Java program that may access this EJB. The assumption is a simple point-to-point connectivity. Every accessing program/application must be aware of exactly where and how the "service providing" application can be accessed. The coupling is very tight—from protocols through the physical address to which the communication sockets are established.

As integration requirements rapidly spread beyond simple homogenous application environments, the environment is rapidly becoming heterogeneous and distributed. In contemporary IT architectures, there is a wide range of application platforms and environments. An added complexity is introduced when they are physically and geographically distributed. In such environments, the spaghetti links that we discussed in Chapter 2 for integrating the large number of solution islands become a major performance and scalability challenge.

To address such wide-scale integration scenarios, there need to be additional infrastructure components that sit in between the two service ends. This would involve services being "available" on this infrastructure and accessed from this infrastructure. This infrastructure will extend across the organization and all existing solution islands can be made available on this common infrastructure. The business processes and integration applications will also be on the same infrastructure as "consumers" of the available services. If such infrastructure leverages the XML-based wire protocols that are popular in Web Services, then the services requests flow through the infrastructure as XML documents. As a large number of services become available and get accessed, this infrastructure will start looking like a "bus" transporting these XML documents. A bus, that enables "services"—a Service Bus.

The service bus provides the necessary abilities to host services, provide effective communication infrastructure where the notion of services is well understood as a first-class-artifact, and abilities to access the services through this infrastructure. Additionally, the bus can provide the abilities to manage the services and their access. This might include providing abstractions of services such that the location

of the services is hidden from the consuming applications and process engines. The infrastructure will take on the responsibility of locating and routing the request to the right location. The difference between Services and a Service Bus is highlighted in the following figure.



One important point to note is that even though the service bus provides additional capabilities, the capabilities only add to the existing service mechanism between service providers and consumers. The service description and request/response representation protocol can remain the same. Additional capabilities are built on this basic protocol. These protocols are well defined by the various Web Services standards. They are also very widely accepted and understood.

The application's view of the bus is based entirely on the omni-present Web Services standards. The services are described using WSDL, and the service requests may be in SOAP. It is just that the bindings used in the SOAP interactions may be different from simple HTTP or even JMS—even though standard bindings are preferable. The bindings specific to the middleware platform, enable the infrastructure to provide value additions in the service bus.

This "Service bus" provides the required infrastructure abstractions. So the applications can still be written assuming the Web Services-like application interface abstractions. Given the flexibility provided in the binding layer of Web Services via the Bindings, the communication layer need not be just HTTP or JMS connections. This layer can be replaced, transparent to the services and the integration applications, by a more powerful distributed services "bus" framework.

We Need Web Services and More

Why do simple Web Services and environments like EAI not provide a service bus? Both of these are inherently peer-to-peer models—Web Services in a pure point-to-point interaction model, and EAI in a hub-and-spoke model where one consuming end can access one or more legacy back ends, but each links in a point-to-point mode. While this serves the basic peer-to-peer connectivity and integration requirement very well, it doesn't provide for the enterprise-grade services scenario—where typically a plethora of connections would exist across a multitude of systems dispersed across geographies. In such widely distributed environments, a different higher performing and more scalable and reliable approach may be needed.

Web Services initially started as a point-to-point approach for integration, where one application can talk to another application in an SOA environment using a simple direct HTTP connection as the primary communication channel. SOAP provides the representation of the request and response that are sent on this channel. Over the years, WS standards have evolved. The WS standards provide a very powerful mechanism to describe any service, regardless of the protocol to be used for communication and also the bindings used to represent the inputs/outputs.

In Web Services, the focus is primarily on connecting two ends—peer-to-peer. All the standards in this space are directed towards this primary objective. In general, though, the interactions in enterprises are not simply P2P, they are more many-to-many. There are many systems and applications in the enterprise, and Business processing will require each application talking to one or more applications—with each application providing services that are accessed by one or more other applications/systems.

In such a widely distributed services environment, loose binding between the various applications is very important to get a successful organization-wide integration. The basic abstractions for the loose coupling are well provided by the various Web Services standards. Beyond this, even the infrastructure layers need to be loosely bound. This is where Web Services, EAI, and Application Platforms like Java EE and .NET fail to provide the needed flexibility.

Added to this, is an interesting natural interaction model between departments in an enterprise. In distributed organizations, the natural interaction between departments and locations is to send "business documents" to a mailbox and get those documents "processed". The business documents could be forms or documents that describe a business requirement or inputs. This is well understood by end users and business analysts, as it fits into the natural interaction models within an organization. In an automated interaction, these documents may very well be XML documents. The widespread acceptance of XML as a document representation format, with most organizations having well defined schemas to represent business document structures and document exchanges already happening in XML lays a very good foundation for alternative services interaction models.

In effect, flip the view from services "invoked" with documents passed in as arguments, to "sending" a document to an address or endpoint and getting the document "processed". This is a subtle, but very clear difference, resulting in a fundamentally different approach to the services infrastructure.

Enter Enterprise Service Bus (ESB)

The Enterprise Service Bus, as discussed in Chapter 2, is a very strong technology option for high-end enterprise-wide SOA environments. ESB inherently provides the service-bus functionality described above.

There are simple Web Service-based integration options, and there are massively distributed integration platforms. The distributed services platform, while supporting Web Service standards, will have a highly scalable runtime and communication environment. The latter is an Enterprise Services platform and the former is just a simple point-to-point services solution.

ESB Architecture

ESB, as a technology for Enterprises Services, provides the necessary infrastructure to host/access the services and provides the required service bus functionality such as scalable communications and mediation and control of the services. While there are no authoritative definitions of ESB yet, there is a common ground that is rapidly evolving: a defined set of platform attributes with a clearly emerging architecture, from hosting or enabling access to services through mediation services and process engines.

Defining ESB

An ESB makes it easy to dynamically connect, mediate, and control services and their interactions. An Enterprise Service Bus, as the name suggests, is an enterprise-wide distributed common infrastructure for hosting, managing, and accessing the services.

The defining concepts of an Enterprise Service Bus are:

- An ESB provides an **enterprise-grade** service execution environment:

  Services need to run in a services operating environment that will ensure reliability, fault-tolerance, and security of the services. These will be inherent properties of the service binding, provided for by the ESB platform. Services themselves delegate the security, reliability, and communication to the ESB infrastructure; they need not implement low-level communications themselves. In cases where the actual service is running in an external SOA-aware service container, the ESB may delegate some of this processing to such containers.

- **Services** on the ESB are all first-class citizens:

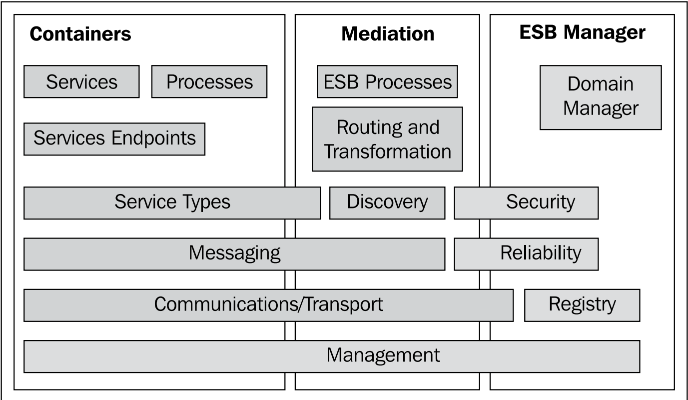
  Services are not just protocol handling layers that help in un-marshaling the requests coming over protocols such as SOAP. In ESB, Services are broadly available and configured for mediated interaction with any other service or processes. The only programming required is the actual service implementation. No other code is needed either to access or to control the services. This would be provided by the ESB infrastructure. Though Web Services will naturally be supported, the ESB will also provide connectivity to a broad range of technologies such as Java EE and .NET components, bespoke applications, and legacy MOM systems.

- An ESB provides a **service bus** to make services broadly available for use across the enterprise:

  The bus is essentially a framework that provides a transparent entity on which services will be available. The service consumers, using this same entity, also access the services. The bus is a topology of "nodes" that forms this transparent and uniform "single" services framework. The bus will be able to scale to connect and host distributed applications and infrastructure services in an arbitrarily large deployment. Other integration technologies typically support hub-and-spoke or star topologies and usually rely on a single central broker. While these may be sufficient for managing a few resources inside a single LAN, they are unsuitable for a broad-scale SOA deployment across WANs and extended enterprises. The bus provides a logically continuous service infrastructure that enables other capabilities such as mediation of the service requests, transformation and routing of data, and control of the services environment.

The Enterprise Service Bus will provide the complete infrastructure from Service Containers and Communication Infrastructure to the Control and Management of the Services. Services may be hosted directly in the ESB environment or it may just provide the plumbing to access the services running in external legacy systems or application servers. Some of the key constituents of an Enterprise Service Bus are:

- XML-based backbone
- Framework to host and run services, or access services running externally
- Highly scalable communication ensuring quality of services
- Capability to connect heterogeneous systems
- Privision of mediation and control of services
- Support for content based routing and transformations
- Support for "orchestration" of services (can integrate with external process engines as well)
- Framework for events and notifications

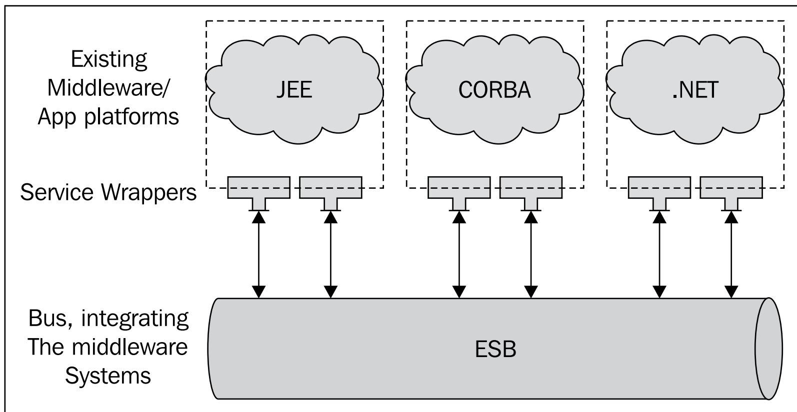


While, at the time of writing this book, there are still no common "standard" definitions of ESB, clearly ESB is evolving into a tangible infrastructure for SOA. Some critics show ESB systems to be just a mechanism to host services and discover the services, based on a registry of services and Web Services-based standards for defining service interfaces and actual access mechanisms. Now, surely, and ESB is not just a service repository, and it is also not just a framework that supports standard services runtime where there are some standards for services assembly and deployment. ESB is far more than that (see the above figure of the functional overview of ESB)—it is a middleware integration fabric for services. ESB provides an Enterprise Services platform in a Bus that connects all the services available in the enterprise to the business processes and services that need to "consume" these services—an enterprise-wide middleware for Services.

Middleware for Middleware Technologies

Extending the existing services platforms such as Web Services and filling in the missing links in the platform, the emphasis is on the infrastructure that "connects" various services and the consumers in the enterprise on a single infrastructure platform. The services themselves may typically have back-end applications that may be hosted on traditional middleware systems such as Java EE and .NET.

Are Java EE and .NET middleware platforms or just application servers? Surely, they are more application platforms with their primary function being to host and run applications. However, they also provide distributed processing functionality via being able to access the functional layer from other applications or application servers. This aspect of the platform is conventional middleware functionality, and when ESB infrastructure connects such platforms, a few technologists refer to ESB as a Middleware for Middleware systems (see the following figure of middleware for middleware technologies).



The services in ESB essentially conform to a common services model. The service implementation will hide the specific platform dependencies of the back-end applications that it may be accessing. Regardless of the platforms behind the service layer, the services themselves will be available in a common way on the ESB. The services in an ESB can be considered to be available on an ESB "address" often also called a service endpoint. Any consumer of the services needs to connect to the underlying framework in a standard way. Once connected, the consumer can access any service in the domain using the same "connection". In essence, each consumer and producer will "talk" only to the framework directly. The framework/runtime will take care of "delivering" the request to the relevant endpoint—very much like normal Messaging systems such as MS MQ, Websphere MQ, and TIBCO. Sonic ESB is one such product that supports this abstraction out of the box—built entirely on an underlying robust messaging infrastructure.

The middleware functionality of the platform is driven by the aspect that all services are available in a common manner. Any consumer of the service need only "connect" to the bus, and once connected can access any services that is available on the bus. While not necessary, ESB platforms typically do this by leveraging a message platform at the lowest level. A service invocation translates to a message being composed and "sent" to that endpoint. Unlike in simple integration solutions, a service or process that needs to access multiple services need not establish connections to each of the service endpoints—an optimal situation for extensive and distributed SOA environments.

Taking the notion of middleware systems a little further, is it likely that there will be multiple ESB platforms in an organization that need to be integrated as well? This is quite possible and will be challenge in the absence of ESB standards. The basic interoperability of ESB systems will itself be possible through simple gateways or adapters. However, ensuring that the benefits from the ESB platform over other SOA infrastructure continue in this integrated multiple ESB environment may be a challenge.

As often pointed out, there certainly is a risk of "locking" the application to one vendor's ESB leading to a likely need for integrating multiple "proprietary ESB systems", but this risk can easily be mitigated by ensuring the vendors have a commitment to the various standards that are emerging in the SOA space.

When considering applications on ESB, it may help to look for these attributes:

- The key application "assets" in an ESB environment are the service implementations/wrappers and the Business Process. Rest of the plumbing is a pure runtime configuration and setup. The latter can be vendor specific, the former should, to the extent possible, be vendor neutral.
- When defining services consider ESB vendors supporting standards.
- In the Java world, consider standards such as Java Business Integration (JBI) and WSDL2. If a Java service is written as per JBI or even just WSDL2, the application artifact (asset) may still be "portable" to other ESB/SOA runtimes with minimal effort. (In the .NET world, as it is a homogenous single-vendor technology environment, there will be no issues with respect to standards.)
- Business processes should be in standard languages (such as BPEL). At the least, the ESB vendor should support "exporting" the Business Process to BPEL. Then, even the Business Process becomes portable.

On a related note, there is a lot of talk and buzz around the standards to extend WS to support reliable messaging etc., but these may exist in localized implementations. To extend this to the extended enterprise, a common communication infrastructure like HTTP over Internet that supports Reliable Messaging and Routing will need to first come up. Even messaging systems such as JMS (the Java Messaging Service, a standard in the Java space) may not suffice for an extended enterprise, as there is no good interoperability across JMS implementations yet—just as there is no interoperability to other platforms such as .NET. Until then the solutions will be limited to those supported by a single vendor that can be used in closed environments alone. Outside this environment, gateways will need to be used (either via regular WS/HTTP or other such mechanisms).

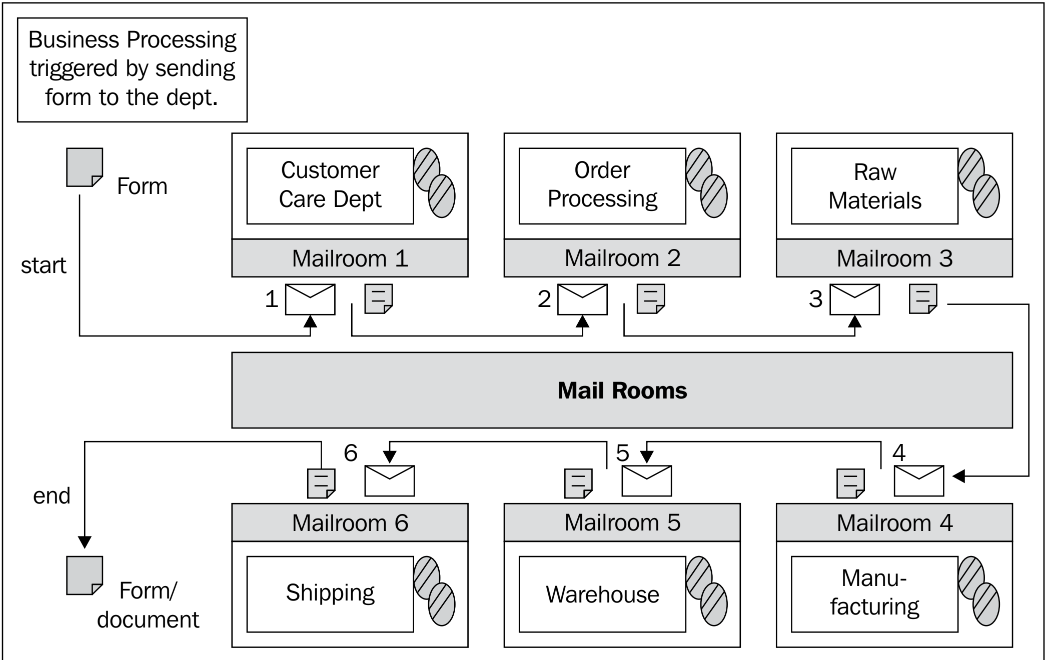
Modeling the Enterprise Document Flows

ESB being the infrastructure sitting in the middle, between the service and its consuming integration applications, it is well placed to provide intermediary functions; functions that could be provided when the request for a service is on its "way" to the actual service running in its execution environment. Now, here, it is interesting to draw parallels between ESB and the way enterprises process business documents—on how the business processes are document centric. Documents flow through the business interactions, among various departments of an organization.

In an organization, all departments work towards serving the organization's business purpose. The business processes that span departments typically will involve a business document or data that is exchanged between departments to get processed. The business document could be, say, a manufacturing work order or a shipping request. When the document is received by a department, the information in the document is processed as per the established business processes and flows in the organization. The function of a department is here modeled by the structure of the information to be sent to the department, and an "address" to which the business document must be sent. Prior to automation, this "document" would have been a paper-based form and the address would be a mail stop or an address. The "integration" of these departments happened via the information flows. Say, a PO is given to the manufacturing department, which will manufacture per the details in the PO. In short, in enterprises business documents flow and get processed (see the following figure).

With automation coming in place, the departments each have their own business application that helps with the departmental business processes and functioning. And more recently, with the Web Services and SOA catching up, all these various departmental applications and solutions are now getting integrated organization-wide. The same document flow between departments could now be an electronic exchange of data between automated business applications, with the data sent to an "electronic address" such as the URL of service.

When integrating the departments, in homogenous technology environments, such as when all applications are either Java EE or .NET, there could be technology-specific mechanisms to access the various departmental services—for example, order processing department could explicitly "invoke" the processPO method in the manufacturing application—but this is a very tight integration where there is a binding to technology, service, method name and parameters, and the actual programming constructs to invoke the service. This, however, does not simulate the "normal" flow across departments in any organization, where data flow is the focus, rather than the "procedure invocation".



The loosely connected departments can be better integrated by keeping the internal details hidden—including what method to invoke. The intuitive integration lays emphasis on the information flows; the assumption being when a document comes in, it gets processed. Exactly what the process internals are, should be immaterial to the originating point. Surprisingly, this was the approach to integration in the days of EDI. Even if the technology was very minimal, the focus in EDI is on the data flowing rather than the actual processing.

The difference between data being "sent" to get processed and "invoking" a process with data as "input arguments" is very subtle. At the end of both modes, the "processing" of the "data" is what is expected. It is just that in one case the data coming in as a single document is the focus, and in the other case the method invocation is the focus. The crux here is that in loosely coupled environments, the "contract" between the departments is easily described if it is just the structure of the data and a location to deliver the data—and not in a tighter bound method with input parameters. This is consistent with a pre-integration organizational structure, where forms or printed documents "flow" between departments and get processed. Such loose coupling serves the significant purpose of keeping the binding between the departments to the minimum.

Extending this further, often there are cases where a single form gets processed in a "workflow" mode. Initially, it contains the input data that triggers the process. Say, it is an insurance claim form in an insurance company's claim process. This form goes from department to department, where it gets processed and annotated with additional information notes and decisions. For example, an investigative officer will validate the genuineness of the claim, and an appraising officer might verify the value of the claim, and so on. The form goes from department to department until it is paid and settled, when it gets filed. In effect, the document flows and gets processed. The specific steps in the processing, and "mail stops" along the way are dictated by established business processes in the organization.

Imagine if this were how business processes could be executed in an SOA environment! Business data is captured as a business document, and the document flows from one application to another through the service containers of the SOA infrastructure. As it flows through the SOA infrastructure, it gets updated with additional information at each step. And the sequence of processing, defined by the business processes, is also "attached" to the same document—like a processing schedule or an itinerary.

This idea of a document flowing and getting processed forms the basis for several additional ESB mediation functions such as transformations and content-based routing, that we will discuss later in this chapter.

Service: Procedure Centric or Document Centric?

Should services be procedure centric or document centric?

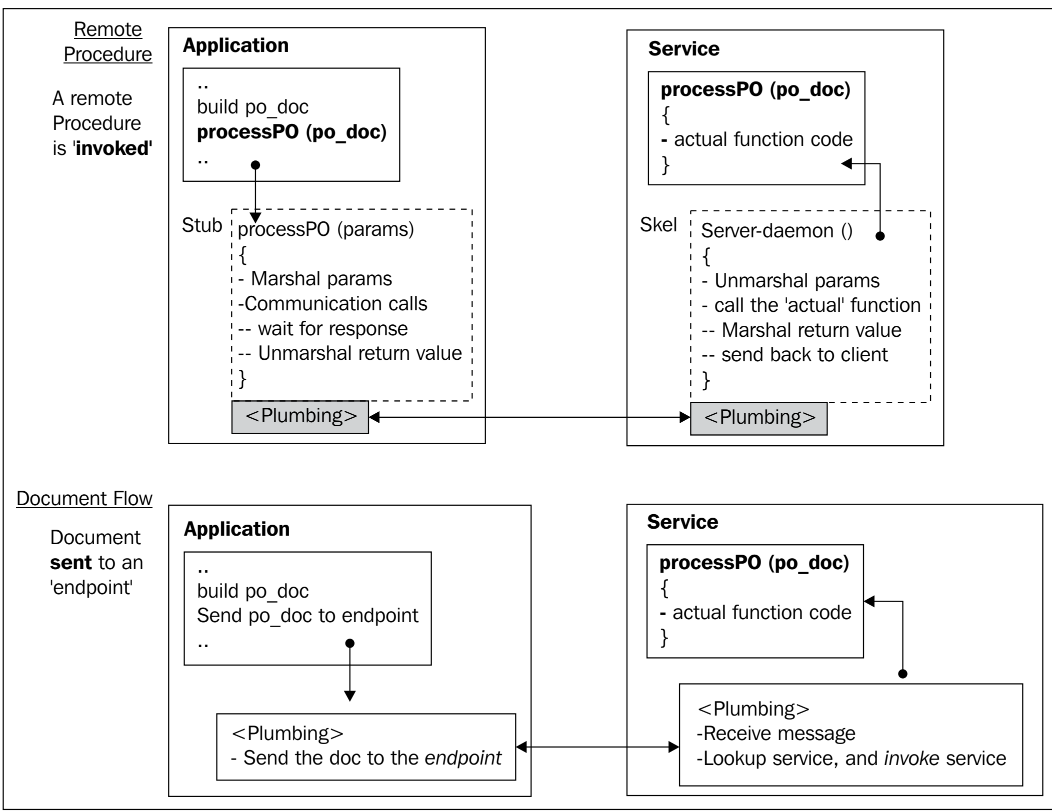
Services in the Web Services framework allow two modes: RPC (Remote Procedure Call) Invocation or Doc-literal, with clear, albeit subtle, differences. The difference between the two modes is about how a service on the "bus" can be accessed. There are two abstractions:

- Access the service as a traditional RPC—wherein the service is invoked and the input parameters sent, and as per the service interface there is a known response that is expected. The caller waits for the response and then proceeds.
- The service is triggered by sending ONE document, of probably a well-defined structure (say a schema or a WSDL type), which is processed and a response document is returned. This is also called doc-literal.

Now, do you think these don't look too different? Consider the example illustrated in the following figure, a service that expects a customer name and a Purchase Order as inputs. Regardless of how the inputs are sent (as a single document or as RPC parameters), the input required to execute the service will remain the same. For the service to be executed it must be triggered and the inputs provided—in the example here, the service must have both the customer name and the PO. In the RPC model, the application (client) invokes the service by passing in these two arguments. In the Document approach (Doc-literal—a SOAP term), the client has to put the customer name and the PO in a single XML document and send it across to an "address"—typically a Web Services URL. At that address, there is a service that expects the single document as input.

It may seem like it is just a subtle play on who does what, between the client application, the middleware (SOA infrastructure), and the actual service implementation! Skimming the surface, the difference is essentially in the application design abstractions in massively distributed environments—which are the typical target scenarios for SOA and ESB (incidentally, ESB is a technology for SOA infrastructure). In such environments, assuming there is a service that is defined, what will be the usage scenarios? In the RPC-type definition, the client application will need to know the various arguments, its types and the order of the arguments. In the Doc-Literal-type definition, there is a single document that is expected. Any required information needs to be modeled in this XML document.

Clearly in the latter case, the information binding needed to understand and use the service (on the client application side) is simpler. The name and IP/port are always needed, but to construct the input, in the latter case it is just a document with a well defined structure. In most SOA scenarios, these will be business documents, probably already well defined and understood within the entire organization. In the former case, regardless of the information model of the Business data/documents in the organization, one will need to know exactly how many and of what type are the input parameters, and extract the data from existing Business data/documents and then invoke the service.



The RPC model is well suited in application environments where data exists in a relatively arbitrary and less structured form. Here, each application is defining its own form and structure. The doc-literal approach is more contemporary and aligned with the emerging Services-Oriented IT infrastructure. In organizations that are consciously and actively working towards an integrated "service-oriented" IT infrastructure, there is a strong push towards well defined business data structures and schema definitions. These will be followed and used throughout the organization for the inter-departmental IT systems' interactions. Business Documents

are composed and "sent" to the required department's electronic address. In such scenarios, the application architecture to enable defining and using services becomes quite simplified:

- Locate the service endpoint (binding, IP, port)
- Compose the business document needed by the service, and send the document to the endpoint

The service definition also follows the same model. Based on the service functionality needed, identify a well-defined business document as its input, and in the service implementation, process that document by extracting any needed arguments from it. So, a well known set of business documents and schemas now becomes the key component of service descriptions. With this in place, service definitions and consumption become very simple and we will need very little information to either describe a service or to use a service.

XML Fits in Nicely in the Bus

XML is omnipresent today. So, it is little surprise that XML is at the core of most ESB functions as well, from the wire representation of requests to modeling all information flows as XML documents.

In ESB, Services are essentially business-functions that could either run self-contained (say, their logic is in Java) or are "wrappers" for back-end legacy processing—like say an SAP or Baan or on a mainframe or even Java EE/.NET. The primary objective of ESB here is to enable simple "integration" of various IT systems in the enterprise: enable systems to "talk" to each other and also enable simple business processes that "orchestrate" these services to provide some aggregate business logic. (For more complex business processes, full-fledged Orchestration servers may be used or even made available as part of the ESB platform. If ESB vendors themselves provide all the power of such Orchestration servers or support standards such as BPEL remains to be seen.)

At the center is the communication of data between systems and services. Now, while the communication could be based on a good Messaging backbone, the actual messages that flow may be service requests, most likely represented in SOAP, which is an XML document format. A further amplification of XML usage is that the service parameters are also in general one or more XML documents. The request parameters and responses also are XML documents.

In effect, information flows between systems as XML. Service requests and responses also are using XML as the exchange mechanism. And soon there are a large number of IT applications/solutions in the organization that talk to each other and integration is widely used—such that even simple operations performed on any system may involve accesses through ESB to multiple other systems, kind of like how widely prevalent DB usage is today.

Once XML becomes the key form for information flows through the systems, then additional needs immediately crop up, say like XML transformations. For example, a Purchase Order in the ERP system may have different fields, when compared to how the OrderProcessingApp understands a PO. And when a PO is sent from one to the other, somewhere it must be transformed from one structure to the other, so XSLT or XQuery comes into play. And once XML documents are flowing, one can quickly think of cases where the "processing" of the document is "attached" to the document, rather than passing the document as an input to a "process". In the mode where processing is "attached" to the document, what actually happens is that the ESB infrastructure handles the document and its processing through the various ESB containers/nodes, passing the document from one processing "step" to the other. In such a processing environment, smart "routing" comes into play. This is basically content-based routing of the document—wherein, based on the document contents, the document is sent to a particular processing "step" (service endpoint or container).

ESB Services: Built on Documents/Messages

Doc-literal opens up a new abstraction for services. Tracking the document-flow based enterprise that we discussed above, unlike the conventional middleware systems, where the remote processing is generally triggered via procedure-like semantics, the document flow abstraction is central to ESB systems. The services now take on a different form—abstracted behind endpoints that serve as an "address", to which a well-defined document needs to be "sent" for the service to be invoked.

Drawing upon the Web Services' doc-literal approach, ESB services are typically built around documents or messages being sent to a service "address".

Abstracting the Service Location with Endpoints

In describing the service, apart from the service interface, the bindings and the "address" are also key components. This address is normally the specific IP and port of the Web Service. In ESB, this is an abstract "address". The service can have the semantics of Web Services, with a well-defined interface that may include a set of related operations. Each operation has an expected input document and a return document. The fault responses are also defined for each operation. The bindings will provide the necessary information to "construct" a SOAP document, at run time, when the service needs to be accessed. The endpoint may be represented as a URL in a WSDL that may define the service.

At the time of writing this book, there is no standard definition of ESB endpoints. The basic service endpoint can be defined using W3C definitions. This, however, doesn't distinguish the various types possible in ESB. Based on common ESB use cases, endpoints could be of these types:

- Service
- Process
- Message Router
- Transformation
- Simple Message Destination (with no other value-added ESB handling)

In ESB systems, if a service execution essentially involves a document or message that is sent to an endpoint, then there is a lot value-added smart processing possible as a result of such usage:

- The documents could be "smartly" routed. Based on routing rules defined, the document could be sent to different endpoints based on its content.
- Documents could be transformed based on business requirements. Transformations can be for the structure, or even for content.
- The document could be processed in the ESB fabric as a process (ESB Itineraries)—the document gets sent to one endpoint, which actually represents a Business Process (and not just a service). This process will involve multiple steps (service invocations)—the business document along with process state data "flows" through the fabric from one service container to the other (per the process definition—itinerary), at each step getting "processed" by a service—as opposed to the less-efficient alternative model for the same, wherein one process engine "invokes" services in the bus in a hub and spoke model—each service request originating from the process engine.

Documents—Route, Transform, and Process

As documents or messages "flow" in an enterprise and "get processed" along the way—by the IT solutions of various departments—additional possibilities crop up.

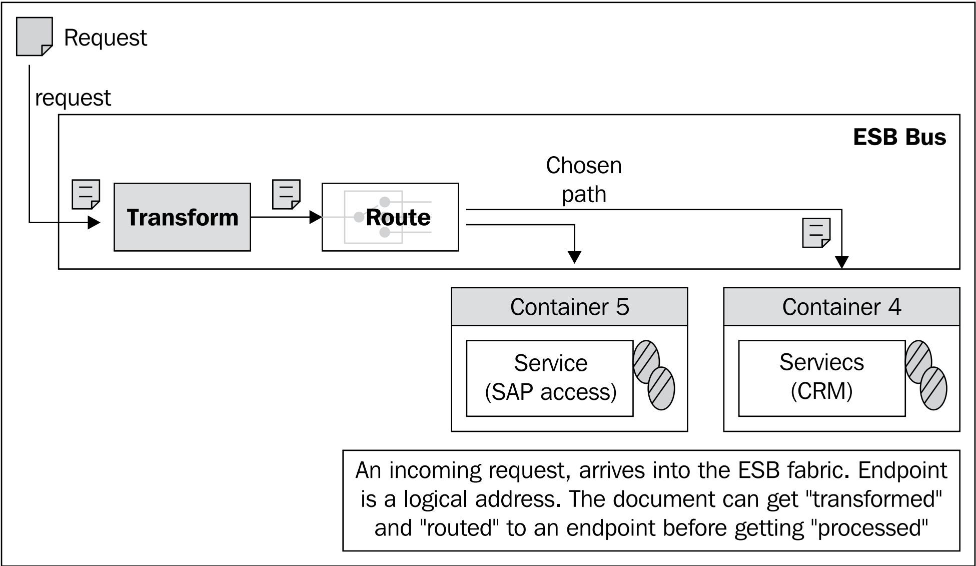
Given that each IT solution would evolve by itself, regardless of other departments in the organization, it is quite likely that the information models used in the various solutions are different from each other. For example, the Purchase Order as represented in the Order-Processing system may be different from the PO represented in the ERP system. There may be additional or a reduced set of fields in the PO. So when exchanging data between these systems, the information from one system will need to be "transformed" to the form understood by the other system.

Such transformation could be done by the integration application—it extracts info from the first system, converts it to the required format, and then sends it to the second system. This is fine in P2P-based solutions, where the service-consuming application is tightly coupled to the service-providing application. However, in more loosely coupled integration environments such as the Service bus (ESB), such tight couplings are not desirable.

This is where more transparent transformation comes in. Each application is aware of its data formats alone. And in the integration environment, say in a Business Process, the required transformations are performed independently. In a more evolved SOA-based enterprise, it is quite likely that there is a corporate type repository, where all information models are described. In such cases, the PO in the Order system and the ERP system will be described in the type library and it can extend a bit further, to also include canned transformations that will transform the PO from the Order-system format to the ERP format.

A good Service Bus can leverage such type libraries and further add value to the platform by helping with the transformations, both during design time (by helping with the XSLT/XQuery needed for the transformation) as well as during run time (by picking up the latest transformation definitions from the central type library).

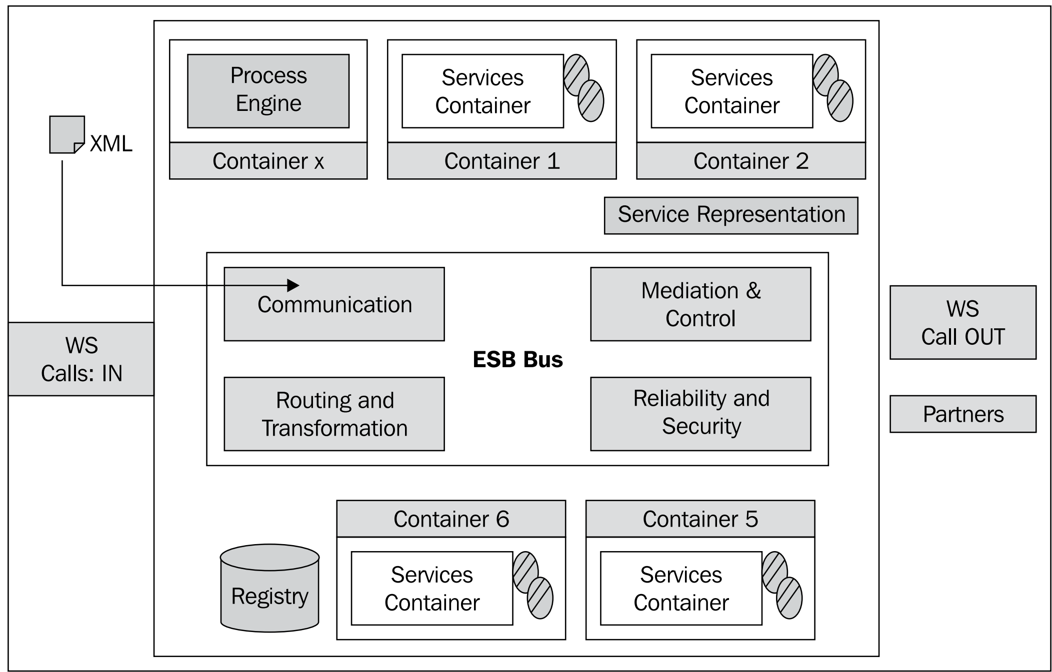
Once transformed, the document needs to be delivered to the service. As the document is now being managed by the ESB fabric, it is possible that the document can now be "routed" to a specific processing endpoint based on the contents in the document. The routing could be based on application-defined rules. (See the following figure).



One could also conceivably look at documents flowing from step to step, and getting processed much like a document processing schedule—a sequence of "mail-stops". The document "workflow" will not be too different from the business processes—with inputs, process flow control, steps, decision points, and an exit. This gets into the bus, forming the input here and the sequence of steps are attached as an "itinerary" to the document. It "flows" through the ESB system getting processed as needed.

ESB Infrastructure Components

ESB is an architecture approach that addresses most of the infrastructure complexities of SOA—from representing the services and executing the services to mediating between the services, the actual communication between the service client or process engine, and the service implementation. Broadly, the Services Runtime Environment and the Bus form two key components of an ESB infrastructure. And then, there are a set of additional services and capabilities, as shown in the following figure of ESB infrastructure.



Common ESB terminology includes terms such as Services, Service Types, Service Containers, Service Descriptions, Transformations, Content-based Routing, Bindings and Binding Components, Service Interfaces, Service Registries and extensive XML handling.

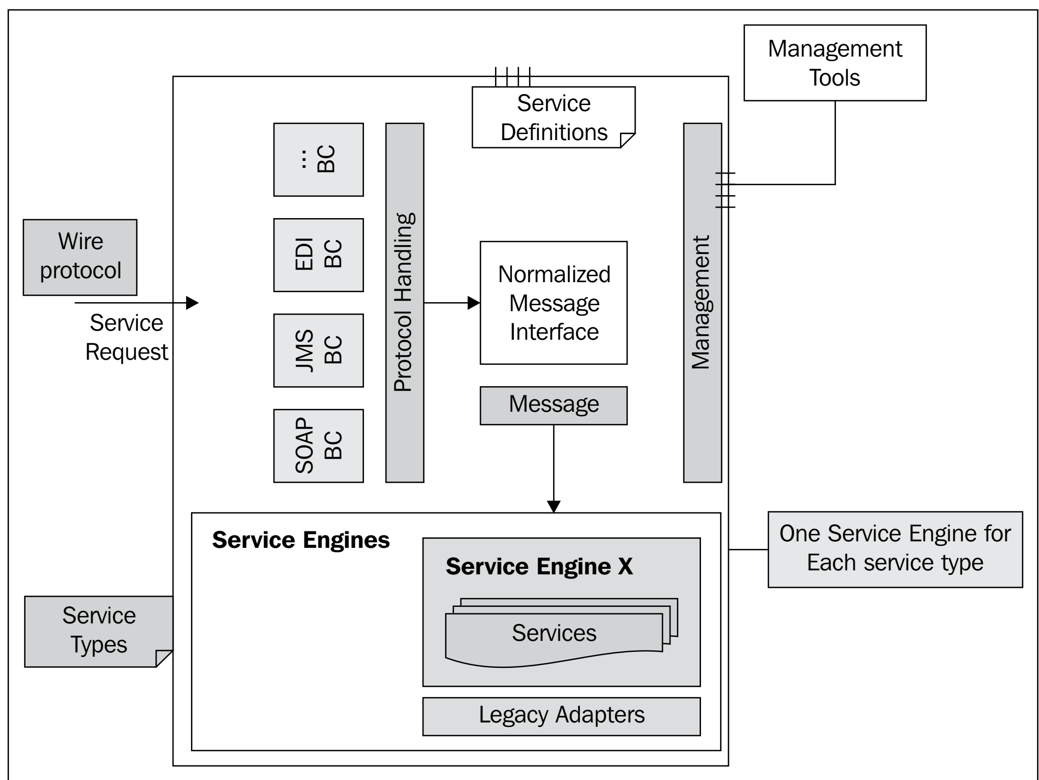
Services are the lowest-level individual components in an ESB infrastructure. Every business function that needs to be executed on the bus will be a service. Services have a handling or processing logic within, to provide their business functionality. Services could be custom written, or could be service instances of a well-defined type—often referred to as a Service Type, and have the handling logic written once and multiple services configured that all use this same logic.

Service types define a specific "behavior". Each service type will have a definitive interface and processing semantics. The Service type is like an abstract definition, and the Service instances are the concrete instances of that "behavior" that are accessible by the consumers of the service.

Service containers in an ESB environment, shown in the following figure, provide the execution environment for the services, with a set of containers forming the ESB domain. Service containers may be provided by the ESB platform, or could be application servers that are "plugged" into the ESB platform. Services can be configured to run on one or more containers for scalability and load distribution. Services are generally of a given service type. Containers provide the necessary middleware plumbing that includes the communication layers, protocol handling, ESB services such as transformation and routing, security, and management functions. The service containers use Binding Components or Protocol Handlers to abstract the protocol handling from the services. This also enables service containers to transparently support multiple protocols—say invoking services via HTTP or a more native Messaging protocol.

Services, in an application-server context, may be application functional-tier components such as Java EE's Enterprise Java beans or .NET services. When such functionality is available in an ESB environment, it will need to be "wired" into the ESB platform. This is typically done by using "glue" services, which are essentially wrapper-services with the logic needed to access the functionality in the application server. The ESB services are generally implemented in a common way per some platform integration semantics defined by the ESB platform. In the Java world, this would be implementing a well-defined Java interface provided by the ESB platform. Services in ESB are independent of the specific functionality they provide or any back-end systems they may access—regardless of whether it is more contemporary application servers or any legacy systems. (Now, here they are quite conceivable that the application servers may themselves provide this glue without needing any "wrappers". This may happen as more standards evolve in the service definition and runtime space.)

Services need definition for consuming applications to understand the interface and the operation details. These definitions use standard interface languages such as WSDL. The primary description is the service's behavioral contract—on the operations it provides, the inputs and outputs from each operation, the message type definitions for the messages/parameters used in inputs and outputs. The specifics are—what URL to use, the protocol to use for communication, the specific bindings—on how each input and output is represented on the wire. These very exact definitions are very important to ensure that both ends of a service invocation speak the same language. The request is represented by the invoking end in exactly the same manner as how it is understood by the service-providing end.



The **Registry** of services provides the capability to discover services available on the bus. ESB implementations may use standards such as UDDI or go proprietary. Registries become an even more relevant function in widely distributed integration environments such as those where ESB plays an important part. In an ESB environment, the registry of services is an important enabler for mediation of services. This will be useful for application developers to query on the services that are currently available on the platform. This is also equally useful for the functioning

of ESB infrastructure—for the validations and execution at run time. The mediation and control services will extensively rely on the registry for keeping track of the various containers, service types, and service instances. Further, each service instance may be available on multiple containers—with the same access "endpoint" but multiple instances of the same endpoint available to provide for higher availability and concurrent load.

The Security, Transformation, and **Routing** services are provided by the ESB infrastructure. ESB infrastructure is inherently built to handle XML extensively. Apart from handling XML for the basic communication protocols, such as SOAP, XML documents are dealt with in the process of handling services and service requests.

XML handling will include transformations (XSLT) and XML Querying (XQuery). ESB environments could have an inbuilt XML Database as well—though such functionality can be equally effective with external XML databases that are available on the bus. Either way, having it on the bus enables more transparent access to the XML database when processing XQuery.

The Communication layer of ESB enables the notional Distributed "Service Bus". The transparent enterprise backbone should aim at leveling the distributed nature of the organization, by hiding the specific locations and connectivity of each department or organizational component/application. All the systems and locations should be available on a homogenous communication backbone.

Mediation and Control is a primary function of the bus infrastructure that helps abstract the exact location of the services and the mechanics of the services access. This involves late binding of the service physical address and protocols, also strong support in managing these services and their deployment and configuration. The services framework on the bus should abstract the specific location of the service instances. The whole service producer-consumer relationship should be based on a well-defined contract (possibly in a WSDL document) and an abstract location such as a named "endpoint" or a simple URL. At the endpoint or the URL, there should be the ESB plumbing that dynamically locates the service on the fly and directs the request to the service.

This indirection immediately enables additional infrastructure capabilities—with respect to mediation and control. As the location of the service is transparent, and only a logical "address" is available outside, the service location can be very dynamic. As load increases, the service can be started on additional service-containers (servers)—ensuring high scalability. When any container goes down for any reason, the next request will be routed to one of the other instances running—ensuring high uptime.

Mediation essentially is the system plumbing that sits between the consumer (that originated the service request) and the actual service—along the way, providing value-added system services such as validation, security, transformation of the data inputs, smart business-rules-based request routing, and handling any mismatch in the invocation abstractions or data models (as there would be in the case of a simple Web Service request made to a non WS services framework in the back end).

The ESB Communication Infrastructure connects all the service containers on the ESB platform to each other and to other infrastructure components of the platforms.

Communication and Interoperability are sometimes confused with each other. Interoperability across systems and services is a functionality provided by ESB platforms. A good ESB platform should allow Legacy Applications, Java EE, .NET, Databases, etc., to be part of an orchestrated business processes. Now, this has nothing directly to do with what the actual ESB runtime is, and what specific transport it uses as its backbone. The interoperability requirement ends just with stating that it should be able to connect the last mile to the service endpoint in its native transport—which any good ESB platform provides. From the service wrappers, through connectors, it would be able to connect to any back end and use any last-mile transport. Once a service requests lands in the ESB runtime fabric, how it delivers the request to the service container and service wrapper that access the last mile is purely transparent to the actual transport used by the legacy back end.

With the primary purpose of ESB being the ability to provide and consume services, the business processes are the most important "consumer" of services. These will need to run in Process Engines. In ESB, they would be like any other Orchestration Server, with an additional attribute that these may be in a services container in the ESB domain. Basically, processes in the process engine are also available as a "service". The process could be running on any orchestration engine, including BPEL engines. The requirement is that this engine be available on the bus, just as any other service is.

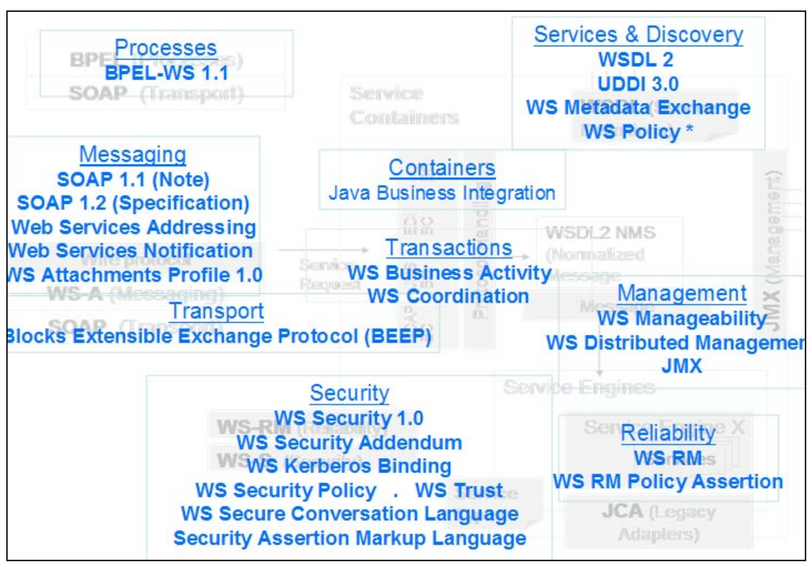
Built on Web Services Standards

ESB draws heavily upon the standards in the Web Services space, even though some of the extensions such as intelligent routing or mediation still do not have any standards foundation yet. Many of the standards discussed in Chapter 2 will be relevant to ESB as well.

In ESB infrastructure, the key pieces in the infrastructure are Service Containers, Service Representation, Communication Frameworks, Wire Protocols, Transformation, Routing, Mediation and Control, Process Engines, and ESB Itineraries. Here there are parts that belong in the applications realm and some in the infrastructure realm. Service representations, business processes, and definition of transformations are part of the SOA application—its "source code". ESB implementations support standards in such areas that form SOA Applications.

In ESB, service representations and business process definitions rely on standards. Services are represented using WSDL, and processes by and large are based on BPEL. Additionally, WS-RM and WS-S are used for security and reliability—specifically for the SOA applications to specify the security and RM requirements.

Where there is no standardization yet, is the internal guts of the ESB—communication protocols and some of the emerging functionality such as mediation and control. The ESB Itineraries is essentially an internal execution mechanics like say another process engine. The process that runs as an itinerary may be coded in BPEL as well.



Referring again to the figure on ESB Infrastructure, of the areas listed there the ESB Bus functionality is the area with no standards in place. The service representations, bindings and protocols have well defined standards. While ESB systems do not have to support all the standards, many vendors do support all standards that come into play in application interaction layers—like service description—or in interoperability and communication layers—like reliability and security.

Description and Discovery: Like any reusable programming construct, the services also need a well-defined mechanism to describe the service details and also provide for "discovering" the services from a common services registry. Potential users may find information sufficient to enable their access and execution. The focus of these specifications and standards is the definition of a set of services supporting the description and discovery of businesses, organizations, and other web services providers; the web services they make available; and the technical interfaces that may be used to access those services.

Reliability: Given the widely distributed nature of services and consumers in an SOA environment, the reliability of the interactions between the service consumers and the providers becomes critical. It is not possible to solve business issues if the participants are unable to be sure of the completion of message exchanges. Reliable messaging, which allows messages to be delivered reliably between distributed applications in the presence of software component, system, or network failures, is therefore critical to Web Services.

Transactions: Transactions are a fundamental concept in building reliable distributed applications. A web-service environment requires coordination behavior provided by a traditional transaction mechanism to control the operations and outcome of an application. The conventional transaction models, the ACID semantics and two-phase commit protocols, assume a much tighter coupling with the operating environment for the resources participating in the transaction. In ESB, given the loosely coupled and very widely distributed nature of the transactions, approaches beyond conventional transaction and two-phase commit protocols are needed. Some of the emerging standards such as WS-Coordination, WS-AtomicTransaction, and WS-BusinessActivity are steps in this direction.

Security: An elaborate set of specifications are either already defined or in the process of being defined that addresses various aspects of security in an SOA environment. Several WS-* standards are in the works. Using these security specifications, applications can engage in secure communication designed to work with the general Web Services framework. Standards include WS-Security, WS-SecurityPolicy, WS-Trust and Security Assertion Markup Language.

Business Processes: A business process specifies the sequence of steps involved in a business operation. This includes the execution order of operations from a collection of web services, the data flowing/shared across these service invocations and the handling of exceptions. In extended business processes, the definition also includes which partners are involved and how they are involved in the business process, and other issues involving how multiple services and organizations participate. BPEL specifies business processes and how they relate to Web Services.

Transports: BEEP, the Blocks Extensible Exchange Protocol (formerly referred to as BXXP), is a framework for building application protocols. It has been standardized by IETF and does for Internet protocols what XML has done for data. MTOM is another specification that is gaining support.

Messaging: The messaging standards and specifications are intended to give a framework for exchanging information in a decentralized, distributed environment. SOAP defines the primary protocol for representing the requests and responses. WS-Addressing defines a standard mechanism to identify WS addresses, used in asynchronous operations and for notifications and acknowledgements. In the

Java world, Java Messaging Service provides a good programming abstraction for working with messages, though the standard falls short of specifying interoperability across different vendor's JMS implementations.

Management: Web services manageability is defined as a set of capabilities for discovering the existence, availability, health, performance, usage, as well as the control and configuration of a web service within the Web Services architecture. As web services become pervasive and critical to business operations, the task of managing and implementing them is imperative to the success of business operations.

Events: What about events? Are events also important? Particularly in the sense of EDA–Event Driven Architecture as proposed by Gartner? ESB and various Web Service standards do support building distributed applications that rely on events and notifications. The extent of usage remains to be seen.

Service Containers—The Primary Tier of the Bus

The ESB platform provides the services middleware and runtime. The middleware is the primary function of ESB that enables accessing services on the bus—from a variety of back-end systems and platforms. The service runtime provides a generic mechanism to run the services. In the minimum, this would be light-weight "wrappers" that enable accessing back-end systems. In more evolved usage scenarios, this could enable full-fledged application functionality that is run in this environment. This latter use case is in some ways replacing what Application Servers do today.

The functionality to enable running services is realized by the infrastructure component—Service Containers. These are essentially a run-time environment to run the individual services. In any ESB domain, there will be multiple Service Containers. Each container will run multiple services. Services may be of multiple service types. Depending on the capabilities of the ESB platform, services may also be clustered—the same service running on multiple containers for high availability and higher load handling.

Services have an "external" view in terms of their contract and addressability—on how they can be accessed. Then, the actual service implementation and its runtime artifacts—the classes/files/binaries, libraries, configuration etc. The service's internal implementation is often abstracted in a reusable service type. Service types include clear definition of the following:

- The interface for the service
- The actual implementation "code"
- Packaging, distribution and deployment onto the ESB platform
- Service instances can be configured of the service types.

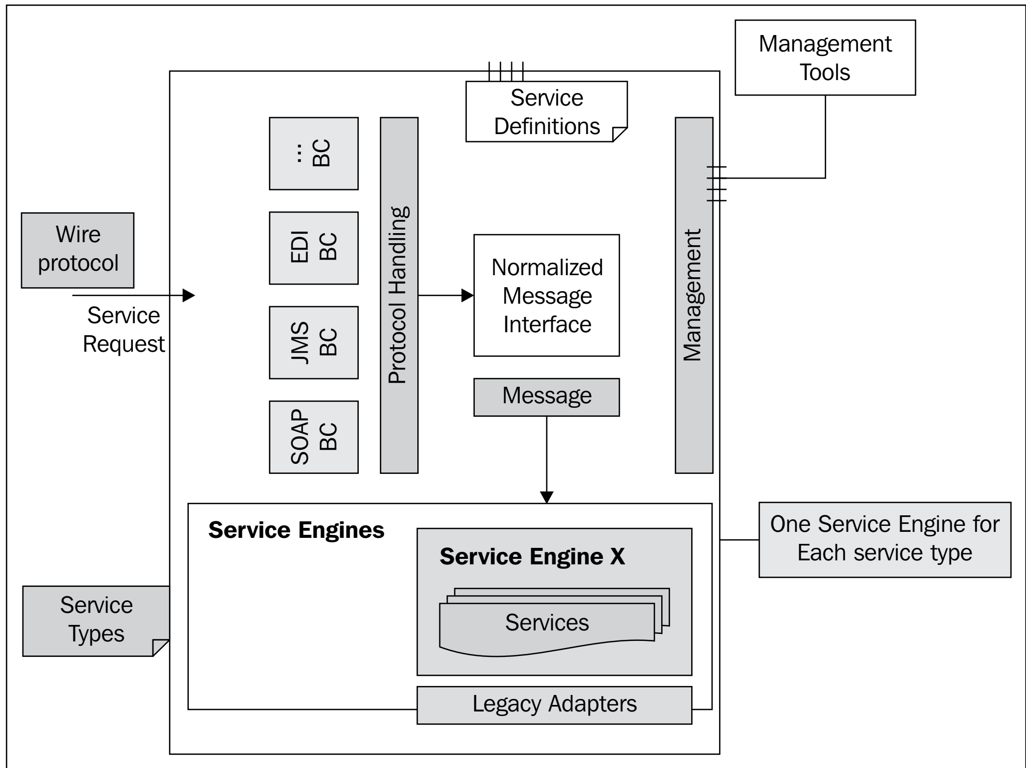
Services that are accessed are service instances of a given service type. While service instances are the actually accessible and executable services, the service types enable the capability for reuse. We have one definition for a service type of all artifacts and its deployment configuration, out of which multiple logically different services can be configured, by creating instances, each with some different environment or initialization parameters, thus providing a different business services, out of the same development artifacts (in the service type).

The key steps inside a Service Container, in processing a service request will be:

- Protocol layer (understand the "bindings"): The first step in a service execution cycle is to understand the wire protocol and "un-marshal" the request. For example if the wire protocol used is SOAP, then the SOAP protocol must be understood in order to extract the vital request details including the service being accessed and the input parameters. These protocol adapters are also referred to as Binding Components.
- Extract Request details: The request inputs or messages are a key part of any service request. Most middleware technologies represent this in proprietary manners. There are also emerging standards such as WSDL2's Normalized Messages that have a standardized representation for any service request's inputs.
- Service instance: A service instance is a named instance that provides a specific Business Service. These are the end services in any SOA environment. These would form a "step" in any Business Process that uses the services. Each service is explicitly addressable. Each service is described by its interface and binding details. The interface defines the "contract" exposed by the service. Binding gives the specific invocation details such as protocols and physical wire representations of the input parameters and such.
- Service types: A service type can be broadly described as a class of services, with a common behavior and probably a common execution mechanism. In the spirit of reuse, these are made available as a service type.
- Service Execution Engines: Each service type would have a service engine that embodies all the execution logic required to run any instance of this service type. Any service will be executed in a service engine that can execute service instances of that type.

Inside the Container

Service containers, sitting on the peripheries of the ESB platform, form the leaf nodes of the Service Bus. They host the services provided by the applications. This is one clear part of an ESB platform where the SOA application comes in contact with the infrastructure. The services are defined by the SOA developers and deployed and configured onto the ESB domain and the individual containers. Considering that there is the application code involved, it is highly desirable that this be based on standards. This ensures that the application is not locked into a single vendor. Some of the key constituents of a Service Container are highlighted in the following figure.



Service Containers will include mechanisms to define service types, configure service instances, describe the service interface and access details, manage the services and the container, and handle various wire protocols and connectivity to legacy systems.

In the case of Services, the service implementation is possibly in Java, C#, or any other language that is supported by the platform. The services platform could be an Application Server as well—so long as the ESB platform can access the service running in the application server. Given the portability of the language, Java is probably a better cross-platform option today, while C# would be a good choice on Windows-centric environments.

Services could either be written to be fully self contained, or could be written in a more generic manner to allow configuring multiple services with a similar behavior. This generic behavior is referred to as a Service Type. To configure any service, the service type that has all the required execution handling for a given service is to be written and deployed. The service type essentially defines a "class" of services. This definition includes the necessary processing code. Once the service type is defined and available on the platform, the individual service instances need to be defined. Depending on the service type, the service instance configuration may be different. Say for an SAP-ABAP access service, the specific SAP access handling is probably built into the service type. Configuring the individual service instances may involve essentially giving say an SAP ABAP function name. Each SAP function can now be defined as an explicitly named service, even though the actual handling logic is defined once for the whole class in the service type.

In implementing the handling for a service, one key aspect that comes into play is the back-end connectivity—given that ESB essentially solves the enterprise integration problem today. The specifics of what back end is connected to and how, is completely internal to the service implementation. The ESB platform may be expected to provide sufficient enabling pieces to help with this connectivity. Here again, there are integration standards at play such as JCA.

With the service definition in place, the immediate next requirement is the service description—needed to be able to communicate to consumers of the services. In a loosely coupled environment such as an enterprise-wide ESB platform, the definition has to be common across all services regardless of service type and the specific containers on which they run. WSDL describes the services. WSDL2, the current version of the standard, has a better definition of the wire formats, normalized messages, interfaces, and bindings. The NMS part of WSDL2 defines the input documents.

WSDL provides a clear structure for defining the operations and interfaces of a service. WSDL2 also defines the notion of a Normalized Message Service. The service implementation will expect just a Normalized Message as an input. This further simplifies the service or service type implementation. The service code will work off the simple document that comes in as the input. Likewise, the service implementation code need not have any handling for the protocol as well. The Service Container performs the protocol handling such as processing and un-marshaling the SOAP request.

WSDL2 also includes the detailed definition of the protocol bindings. The protocols are the actual wire protocols of the service requests. The primary input to any service invocation will be a document. The structure of the document will be well described by its schema.

Service containers will need a Service Engine to execute the services. This will be aware of the Service Type and any other common handling that the Services may expect. The Service Engine will be the component in the Services Container that will provide the interfaces with the various other components in the Service Container such as the protocol handler, adapters, management framework, and other system resources made available to the services by the container.

Being complex infrastructure components, Service containers need good management solutions. This will include configuring the containers, deploying Service Types, configuring services, monitoring and managing the services, and managing the start and stop of containers and services.

Why Standards?

Talking about the standards, one will realize that all good ESB solutions have to provide support for the various WS-* standards rapidly emerging—not much value in recreating anything new here. More importantly, from an SOA customer's standpoint, the security and flexibility (to choose different vendors) that standards provide is a very important factor. This is the interface/contact layer of the whole solution—be it WSDL for describing services, WS-R for specifying reliability requirements, WS-S for security, or soon JBI for service containers in the Java world! By 'interface/contact' layer, I mean any part of the solution that an application's artifact/code comes in direct touch with.

Under this layer, what the infrastructure does is totally transparent to the application's code. But as an infrastructure decision, this is one of the most important factors! The specific approach for the infrastructure determines a lot of the performance, reliability, and scalability abilities. And taking it a step further, this is what an ESB vendor must provide. Else, it is just a bunch of WS-* API handling layers. In short, both the standards-based application interface layers, and the high-performing and reliable infrastructure plumbing underneath are important in a platform choice. The former ensures that the application is not locked into a single vendor's platform, and the latter ensures that the platform chosen scales and delivers the organizational computing need.

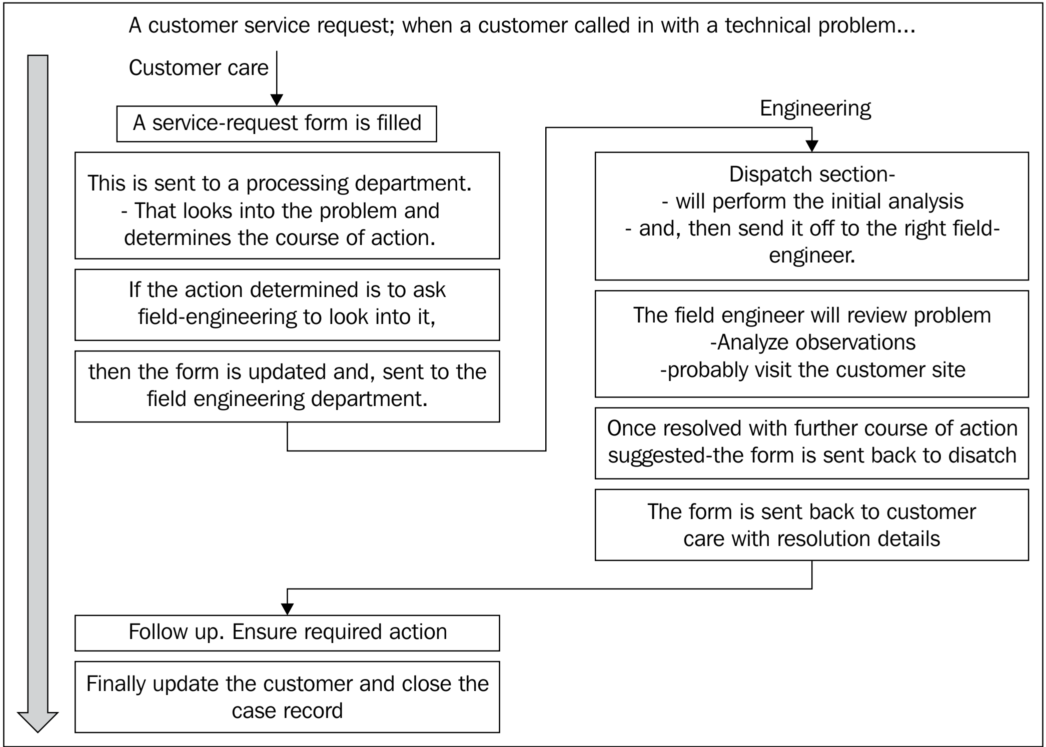
It is often argued whether ESB needs a backbone infrastructure or if it could leverage existing infrastructure. ESB is about a Services backbone. Now if there is an ESB solution that does not include a backbone, then what exactly does it consist of? It is more of a simple integration solution that enables services. The communication backbone enables other things beyond just services' access—like mediation, control, and routing services. In the absence of this layer, one will need a proxy service to perform these functions—in a rather inefficient manner.

External View of Services: Documents Sent to Abstract "Endpoints"

In ESB, a service is identified by a document of a specific structure (schema) that is to be sent to an address or an "endpoint". As discussed in Chapter 2, SOA defines the notion of a service, from a normal procedure call with parameters or arguments, to a document "sent" to a specific place for "processing". The document is defined by its schema, and the address by an "endpoint".

An endpoint is a location. That location may be a simple URL that directly represents the host and port at which the service is available or may be an infrastructure component that will redirect the request to the right location after performing the systems plumbing.

This is not too different from the mail flow of some years or decades back—where forms were filled up and sent to various "departments" for processing. There is a clearly defined business process, even if not well documented, on the processing steps for any form. The form gets sent to the next "mail-stop". At that mail-stop there is someone or a department that processes the form, updates the form or appends the form with additional information or status or reference, and sends it across to the next department. At that department the same form is processed, with the department being fully "informed" of the original inputs and any additional information or statuses that were added to the form as it got processed until that point. The continues until the end of the Business Process, where the form may get eventually "closed". A customer care call is an example of this, shown in the following figure.



This is a potential real-life example of a workflow. While there are many ways for the information to flow, and there could be many ways each department may store its info, the requirement for the customer-service work request above is that all the relevant info up to that point for the work request be available at all points.

In the absence of centralized IT solutions, each department probably maintains the information in its own ledgers and probably maintains copies of each work request processed, but the main work request that contains all additional processing info, status, and observations flows to the next processing step.

In SOA-based environments, this is probably not too different. The services being integrated are generally departmental services—a service provided by a departmental self-contained application. As these are departmental applications, other departments may not have direct access to the data maintained in these systems. To perform cross-departmental functions such as above, a simple approach could be to expose services that provide access to data as needed. That would mean that a lot of the data-access logic would now need to reside in the Business Processes. An alternative, a better one, is to exactly "mimic" the manual processes that might have existed prior to IT—where the "form" along with all information updates "flows".

At each step in the process, it is essentially a "document" that is sent to a "location". Each functional processing has a well-defined location or endpoint and a well-defined document structure it might expect. Just send the document to that location, and it gets processed. Along with this, the document itself could contain the processing sequence or the itinerary for that document's processing—which department/location does what when.

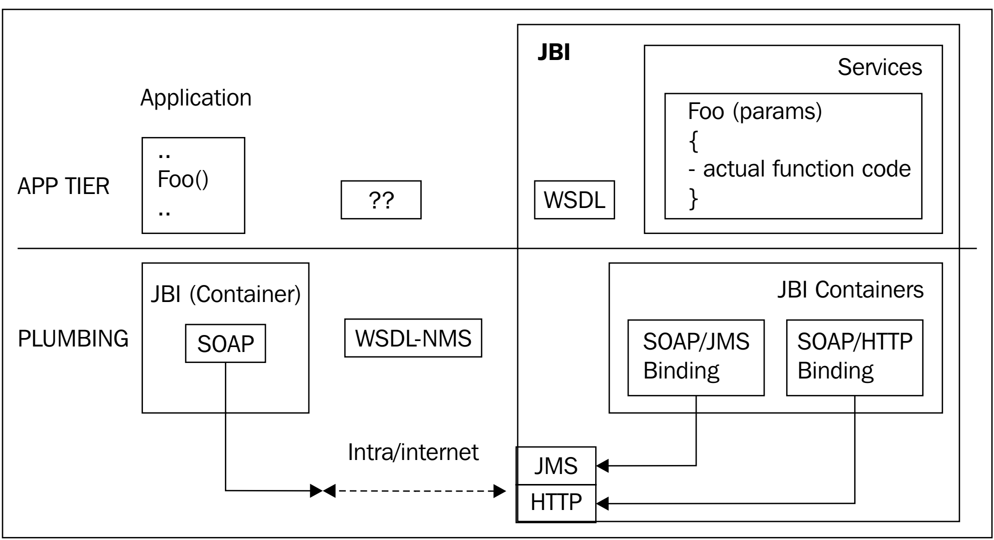
An endpoint is an abstract named entity that represents the location at which a service is available. The endpoint is represented by a name defined using a simple string. The internal meaning of this name, and how it maps to a physical address, is internal to the ESB implementation. From an SOA application viewpoint, a service is available at that logical address, represented by an endpoint.

An endpoint translates to a physical address, by the configuration process supported by the ESB implementation. In Messaging system-based implementations, an endpoint may translate to a Message destination—Queue or Topic. The configuration of a service will then involve setting up the service type that will include getting the required files, classes/JARs/libraries, and other execution-time requirements to be made available in the service runtime environment. Create a service instance of this type. Create a Message destination, say a Queue. Create a new endpoint that is attached to this destination. Finally, associate this endpoint with the created service instance. When using this service, a message is sent to the endpoint, which internally will translate into a Message sent to the Message system destination—the new Queue that was created and attached to the endpoint.

JBI—A Standard Container to "host" Services

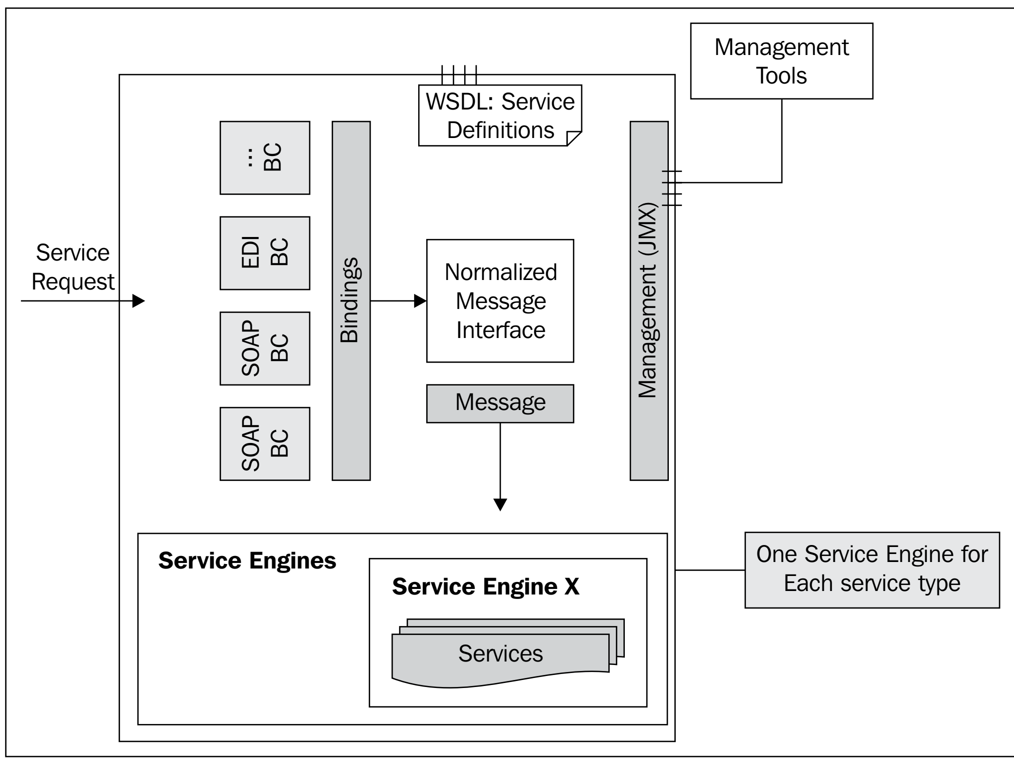
As we saw in Chapter 2, while there are several standards to describe services and represent service requests and responses on the "wire", the standards for actually hosting services is still emerging. JBI (Java Business Integration) is evolving as a standard framework for hosting and running services in the Java space. Having a standard framework to run services is important to avert vendor lock-ins. Standards-based services containers would mean that the services are built and deployed in a standard way, using mechanisms described by the standards to deploy the service engines and service types.

Service implementation and service invocation are both directly dependent on the chosen platform for hosting the services. Given that SOA is just a concept, just about any server-side infrastructure could be used to host the services. These include custom Java/C++ programs, Java EE, .NET, and CORBA. The mechanics of invoking a service are shown in the following figure.



JBI (Java Business Integration) is an attempt from the Java Community, through the JCP, to solve the services container problem. A generic framework is being evolved to provide a runtime environment for the services. A Service can be written in a generic way, and be hosted on a JBI-compliant container from any vendor. This is not possible in a simple WS environment. WS standards only describe how a request is represented over the wire they don't provide for any mechanism to "host" services. JBI tries to fill this gap. See the following figure.

JBI architecture is modeled very closely on the actually processing involved in any service container. Given the flow of a service request and execution, JBI constituents are modeled very closely on these lines.



The JBI Constituents are:

Service Engines: JBI envisions specific runtime handling for various service types. Service types could be a "class" of services with the actual execution of such services abstracted by the service type's implementation. The service itself will then be a simple artifact. An example could be an XML-Transformation service. Each XSLT file may be a service instance. But the engine to run the XSLTs could be common, represented by a service type. When setting up a service, the service type is chosen first, and then any additional information such as the name of the XSLT file is provided as service properties. The end result is a specific transformation service. Say, if there is an XSLT to extract an address from an order (Order-to-Addr), I can set up a service instance using this XSLT file for extracting the billing address from an order and call the service instance ExtarctBillingAddressFromOrder. This will be a service of type TransformationService, with the Order-to-Addr.xslt as the XSLT file to be used. The TrasformationService type expects this as one of the service properties.

Binding Components: The binding components are the protocol adapter layers. There is one binding component per protocol. Say, one for SOAP, and one for JCA.

Normalized Message Service (NMS): Normalized messages play a key role in separating the service implementations from the Containers. Any service implementation can expect the request inputs/parameters to arrive in a standard manner via the Normalized Messages, regardless of which protocol the request came on, or which binding component processed it. Regardless of any other mechanics of the services container, the inputs arrive at the service execution layer in the exact same manner. This is very important to ensure portability of a service implementation—the service engine. The Normalized Message Router routes the message to the right execution engine.

Services: Services are the specific service instances. Services are the actual end service instances expected by any application or business processes in the SOA environment. Each service (instance) serves a specific business purpose or functionality. Each service will expect the inputs per the NMS, and the service implementation will rely on a specific service engine for its execution.

Service Definition: Each service in the system will need to be described, to enable the "consumption" of the service. The services in JBI will be described via the WSDL2 standards. WSDL2 improves upon earlier version of WSDL by clearly separating the interface and binding parts of any service definition.

Management: In the Java world, the omnipresent management standard is JMX (Java Management Extension). JBI also specifies the use of JMX for the management of the JBI Service containers.

Communication Infrastructure

ESB provides communication infrastructure as a core part of the platform. ESB is not about just two systems talking to each other. That is the turf of simple integration solutions like Adapters and "SOAP over HTTP"-type P2P invocation mechanisms. ESB is more about large-scale enterprise-wide services infrastructure involving a large number of service containers, invocations, transformations, routing, and orchestration, all in a highly scalable manner with good business responses. Applications and processes using services could become as common as DB access today is. As SOA takes off as a paradigm with good infrastructure support, the number of SOA service invocations will increase. Performing any user operation in an application may involve several back-end SOA service calls. This surely runs the risk of choking the networks. Such large-scale integration is practically feasible only with a very good high-performance backbone. Simple P2P integration mechanisms such as WS-over-HTTP will not suffice at all! Good ESB implementations will become imperative!

ESB systems, even though they are built around XML and Web Services, overcome the limitations of typical Web Service platforms by providing significant communication layers. Technologies such as messaging, under the covers, enable this scalability. SOAP over Messaging is one of the accepted bindings that enable supporting Standard Web Services as the applications' view of the services and their consumption, but the infrastructure could still transparently use Messaging to provide the lower-level communication. Messaging is at the same level as HTTP here, and is at a layer below the main WS abstractions.

In effect, WS and Messaging are complementary in ESB. WS over JMS actually adds significantly to scalability. Now, we are not talking about a simple WS request made over the network; we are talking about an enterprise-wide integration scenario, where many applications/systems in the enterprise talk to many other applications—enterprise wide, geographically distributed, over both high-speed LANs and not-so-high-speed WANs. Unlike simple Web Services, the ESB platform manages the communication.

Bus Services—Mediation, Transformations, and Process Flows

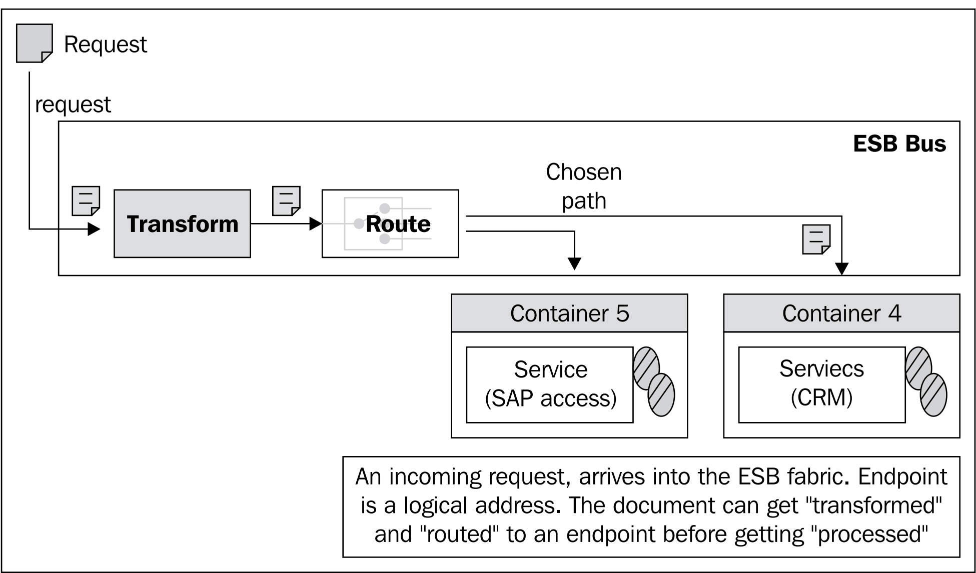
The Service Bus provides critical integration handling capabilities en route to a service's execution from a requesting application or process. These mediation functions, provided for by intermediary components of the ESB, include protocol handling, routing, transformation, security, and management.

ESB services and bus functionality hover around this aspect of "document flows" in an organization, which in the ESB context can also be seen as a message flow. With ESB modeled on such flows, the information now flows as documents or messages rather than as procedure requests. With ESB now acting as an intermediary between the service requester and the service provider, additional infrastructure services are possible in this mode, while propagating the request to the service implementation. The infrastructure has clear "insight" into the message contents unlike simple P2P services interaction. This is useful in providing services such as routing, automatic transformation, security, quality of service assurances, etc. All of these can happen transparently, without interfering with the actual service's execution or imposing any additional programming requirements on the service.

Why Mediation?

Mediation provides the transparency needed for the service consumers on exactly where the consumed services are and how they are executed. This is essential to ensure a loose coupling. Mediation is provided by the ESB platform, which acts as an intermediary between the service provider and the consumer—along the way providing key platform services. Without an intermediary, a requester has to know the service address and access it directly. With the ESB, the requester delegates the mechanics of making the call to the ESB. This includes handling the details of locating the service, invoking the service, or routing the message. This makes services and the code that invokes them simpler and easier to maintain.

In addition to the late discovery of the service, additional infrastructure functions would also need to be provided by some component. These include such functions as protocol handling, security, transactions, and data format transformations. Most of these will need a functional component between the client integration application and the service being accessed. The following figure shows the pre-processing in the bus prior to executing the service.



Mediation essentially is the system plumbing that sits between the consumer (that originated the service request) and the actual service. Along the way there can be value-added system services such as validation, security, transformation of the data inputs, smart business-rules-based request routing, and handling any mismatch in the invocation abstractions or data models (as would be in the case of a simple Web Service request made to a non-WS services framework in the back end).

Mediation functions eliminate the need for additional infrastructure and protocol handling functions to be built into the integration applications. The more contemporary SOA infrastructure provides this in a more generic manner. For instance, handling data model mismatches isdone more declaratively and across the enterprise. Global transformation rules are defined, where a given source document can be automatically transformed to specific target documents. When a service request is made, the mediation layer checks the document type expected by the actual service implementation and if the input document is of a different type, then applies the transformation rules specified to convert the input document to the form expected by the actual service.

This is done transparently even to the service deployment and configuration process. When defining the service there is no need to neither code for nor mention all document types that may be expected as input and what the transformation needed is. The various document types and transformations are globally defined and the rules applied automatically by the mediation layer.

Likewise, protocol mismatch is another common integration issue. The request arrives in a given protocol, but the service is deployed in a different protocol. An example is any ESB implementation such as Sonic ESB. Here the high-performing back end is JMS-based using proprietary wire protocols in the back-end bus. However, the request may come in from outside the ESB domain as a standard web service request. Now the service running in the ESB domain will expect the requests per its specific internal protocol. The service definition and the expected inputs are as per the specific formats.

Now, if the same service also has to be accessed from outside the domain as web services, and if to support this the programmers are asked to "code" the web service access as well, then the value from the platform is greatly diminished. The expectation is that the "same" service or process is "available" as a web service with some facilities provided by the SOA infrastructure—such services would normally be provided by the mediation layer. Based on some rules and protocol handling definitions, a SOAP request could be converted to an internal protocol and likewise on the response path, the internal response is converted to a SOAP response and sent back to the caller.

Physical Address Indirection Enables Mediation

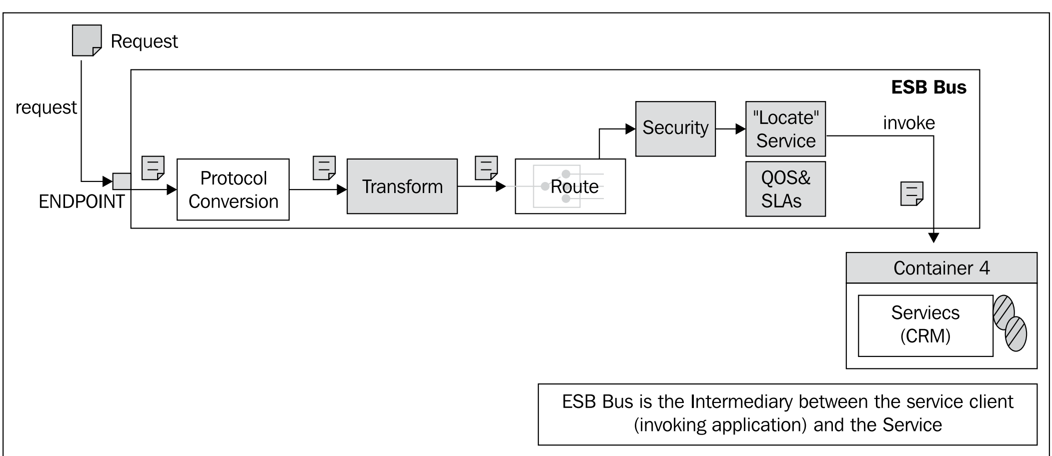
Service lifecycle is the transitions from getting a service deployed and configured to starting and stopping the service. Service lifecycle typically determines the availability of a service. In hard-wired accesses, where the service specific connection details are exposed, anytime this service goes down, the service-becomes unavailable. If this can be abstracted behind a logical "address" or "endpoint", then the service lifecycle can be managed without always affecting its availability.

The services framework on the bus should abstract the specific location of the service instances. The whole service producer-consumer relationship should be based on a well-defined contract (possibly in a WSDL document) and an abstract location such as a named "endpoint" or a simple URL. At the endpoint or URL, the ESB plumbing will dynamically locate the service on the fly and direct the request to the service.

This indirection only enables mediation and control. As the location of the service is transparent with only a logical "address" available outside, the service location can be very dynamic. As load increases, the service can be started on additional service-containers (servers)—ensuring high scalability. When any container goes down for any reason, the next request is routed to one of the other instances running—ensuring high uptime.

Infrastructure Mediation

Infrastructure mediation is functionality that is void of any business logic or application inputs. It is about more environmental considerations such as the protocols, security, and service registries (see the following figure).



Protocol handling: Multiple protocols will exist in an enterprise. Even so, individual services may be available only on specific protocols. In this environment, requests may arrive on a certain protocol while the actual service is available on some other—such protocol mismatches can be handled transparently by the mediation layer when passing the request to the service. This may need support for multiple communication abstractions, such as event-driven publish-and-subscribe, synchronous and asynchronous invocation, and others.

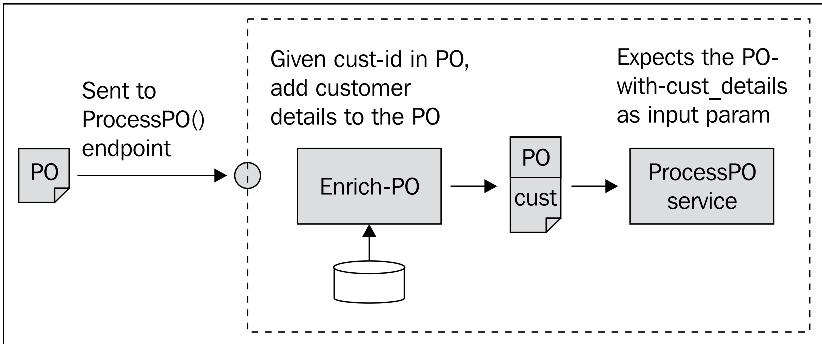
Request routing and version resolution: This is the physical indirection handling, where the physical location of a service is determined by the mediation layer, and the request routed to the container running that service. In doing so, the infrastructure also can manage versions of the service artifacts. This will allow more advanced service upgrade management. There can be non-infrastructure routing possible such as subject-based, content-based, and even itinerary-based routings.

Security: The Mediation layer can enforce security policies regarding service usage. This requires additional infrastructure for defining and managing these policies, and for managing the identity of service requesters. Or the ESB may just hook into existing security infrastructure for identification, authorization, access control, and others as required.

Quality of service: The ESB can persist requests to message queues and retry service operations when failures occur, implement failover to alternative servers, and other steps to ensure that otherwise unreliable networks and services can be made to provide the quality of service required by the requester. More sophisticated implementations will interact with the environment to provision additional service instances to handle increased request volume, so that SLAs can still be met.

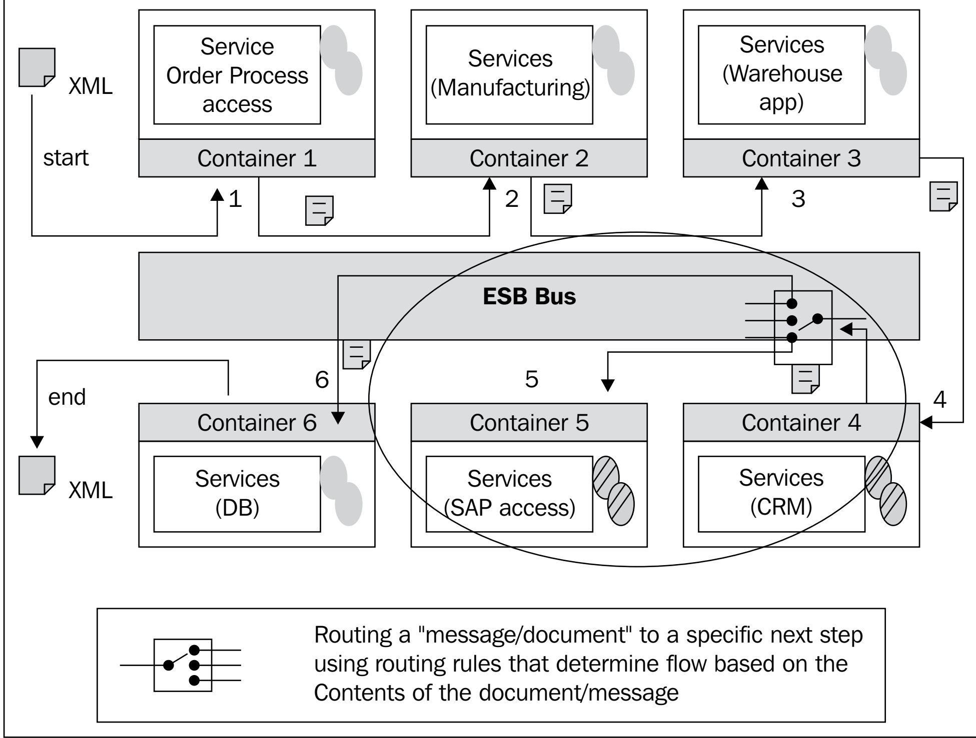
Service registry: The service registry will hold the information on the services, their current physical locations, and status information. This is used for service discovery and routing. When maintaining a namespace (service discovery), the ESB may extend the service metadata; this requires a specification (such as WSDL), to enable services to be classified to ease searching for reuse.

Message enrichment: This is a special case of semantic mapping, where the input data is tagged with additional information fetched from a database or other sources of information. Such enrichment enables the input document or message to emerge from the bus richer with data than when it arrived. In the example shown in the following figure, a service implementation expects a PO to include customer details, and if a PO arrives without the details, this message could be appended with the customer details loaded from the database.



Intelligent Content-Based Routing

One defining attribute of ESB is that documents and messages flow through the bus and get processed along the way. One such processing is the content-based routing that the Bus provides. This is like conditional single-step itineraries—meaning, based on the content of the document it may take different paths. Say the document is a customer-order, and based on the part type, either it has to go to the manufacturing system (that is used for managing the operations of building 'custom built' products) or to the warehouse system (that is used for pre-built stocked products). Normally, some application would execute this check and logic. But in a smart document-flow-based organization, it is conceivable that content-based routing is possible—where routing rules are set up on various document types—and whenever the "router" is reached, it inspects the contents and applies the routing rules and picks a path. This can happen, independent of any other steps in the processing of the document or any other process-flow semantics. Routing happens in the ESB fabric, as shown in the following figure.



The routing rules are platform specific at this point. Each vendor has its own mechanism and possibly language—even in cases where pluggable rules engines are used, there are no portable standards. This will include support for accessing the input documents and the contents in the documents, and using this in defining the rules. For each rule defined, there will be an "endpoint" that is specified to which the input document or message is to be sent when the rule evaluates to true at run time. This rule evaluation will happen in the ESB fabric in the intermediary layers—either using native rules engines or embeddable external rules engines.

Content-based routing (CBR), under the hood, is essentially a simple rules engine. The CBR will get an input document(s), and the rules are based on the contents of this document(s), and will evaluate to a "destination" to which the input document(s) need to be routed or forwarded.

Transformation Services

As documents flow through the ESB system, getting processed along the way, it is quite possible that there is a need to transform the document structure along the way. One common need for this is when a service that is being accessed expects the information in a format that is different from the input document.

There may be several different types of transformation, like request, response, format mismatch, explicit conversion, extract parts from input document, and more. The document that flows and gets processed is essentially a superset document—it may need to be subsetted for each service along the way.

In a hub-spoke model, these transformations are explicit steps in a process, before the step that needs the modified document is executed. In an ESB system, the transformation is available as a system service, and so can be executed any place—it doesn't have to run in a Process Server alone, it can happen anywhere, and in-place!

The Need to Transform

Once we accept that documents "flow" in an enterprise and "get processed" along the way by the IT solutions of various departments, then additional requirements crop up.

Given that each IT solution would evolve by itself, regardless of other departments in the organization, it is quite likely that the information models used in the various solutions are different from each other. For example, the purchase order as represented in the Order-Processing system may be different from the PO represented in the ERP system. There may be additional or a reduced set of fields in the PO. So when exchanging data between these systems, the information from one system will need to be "transformed" to the form understood by the other system.

Such transformation could be done by the integration application: extract info from the first system, convert to the required format, and then send it to the second system. This is fine in P2P-based solutions, where the service-consuming application is tightly coupled to the service-providing application. However, in more loosely coupled integration environments such as the Service bus (ESB), such tight couplings are not desirable.

This is where more transparent transformation comes in. Each application is aware of its data formats alone, and in the integration environment, say in a Business Process, the required transformations are performed independently. In a more evolved SOA-based enterprise, it is quite likely that there is a corporate type repository, where all information models are described. In such cases, the PO in the Order system and the ERP system will be described in the type library. And it can extend a bit further, to also include canned transformations that will transform the PO from the Order-system format to the ERP format. In such cases, the ESB environment can automatically transform the document to the format needed before actually sending the document to the service that is being invoked.

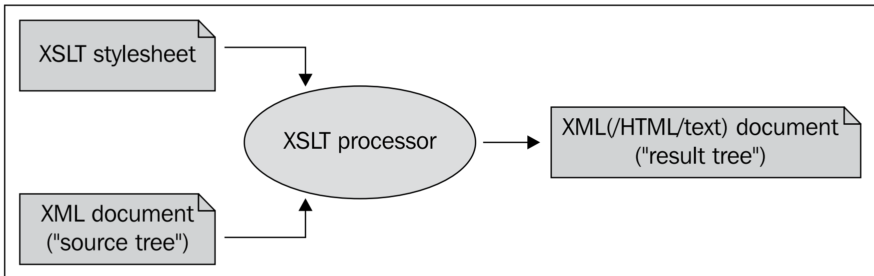
A good Service Bus can leverage such type libraries and further add value to the platform by helping with the transformations, both during design time (by helping with the XSLT/XQuery needed for the transformation) as well as during run time (by picking up the latest transformation definitions from the central type library).

The transformations may be explicit transformations programmed in the application, or implicit auto-transformations based on type libraries. Type libraries are corporate schema repositories that provide additional information on the various XML document schemas used in the organization. This information will include conversion or transformation rules to get the data from other schema types to the required schema type. This will help automate mapping the information from the message or document to what the service requires, and so on all without the service implementer or the consumer application needing to code for any of this.

Transformation using XSLT

XSLT is part of XSL (eXtensible Stylesheet Language). XSL consists of two parts: XSL Transformations XSLT and XSL Formatting Objects.

XSLT files, often called stylesheets, are themselves XML documents defining a transformation for a class of XML documents. The basic design is that XSLT is declarative and based on pattern-matching and templates (see the following figure). XSL Formatting Objects is the "second half" of XSL. It is an *XML vocabulary* for specifying formatting in a more low-level and detailed way than is possible with HTML and CSS.



XQuery for More Complex XML Document Manipulation

With XML being omnipresent in enterprises, and more so in an ESB bus, other XML-centric processing quickly emerges as a need. As documents flow through the bus and get processed, there could be cases where the documents need to be saved in a document repository or database for subsequent retrieval and processing. Once documents get into a repository, when a business need to retrieve and process arises, there will be use cases where multiple documents need to be retrieved, possibly based on some query criteria, and then processed.

XQuery is a standard mechanism available to realize the XML querying and handling. A set of XML documents could be retrieved, and then filters and rules applied to the set of documents, and finally the output document(s) composed from this set by extracting and massaging data from the XML documents as needed.

ESB systems should support XML querying within the infrastructure. This will pre-suppose that there is the ability to store XML documents in a repository or database. XQuery could be a step in a business process, where this step returns multiple documents based on its rules and transformation definition. The process will then use each of these documents in its subsequent steps—probably in a loop with the same set of steps repeated for each document. In effect, we have now converted a business process into a batch processing mechanism. The process may still get its input document, which may represent the criteria for extracting the required XML documents and then processing the resultant set as defined in the process definition.

ESB Processes: Extending the WS Process Model

Business Processes are an inherent part of any SOA-based infrastructure. One primary purpose of embarking upon SOA-based infrastructure would be to enable the integration of the several systems in the organization to provide automated business processes that span accesses to these systems. The most common processes approach is a hub-and-spoke model where the processes run in a process engine. This is very much possible in ESB where the process engine is running in a container

on the bus. Additionally, ESB enables an alternative model that closely mimics the document-flows-and-gets-processed model in manual organizational business process flows that we discussed earlier in this chapter. These are often referred to as "ESB Processes".

Processes—n the "Fabric"

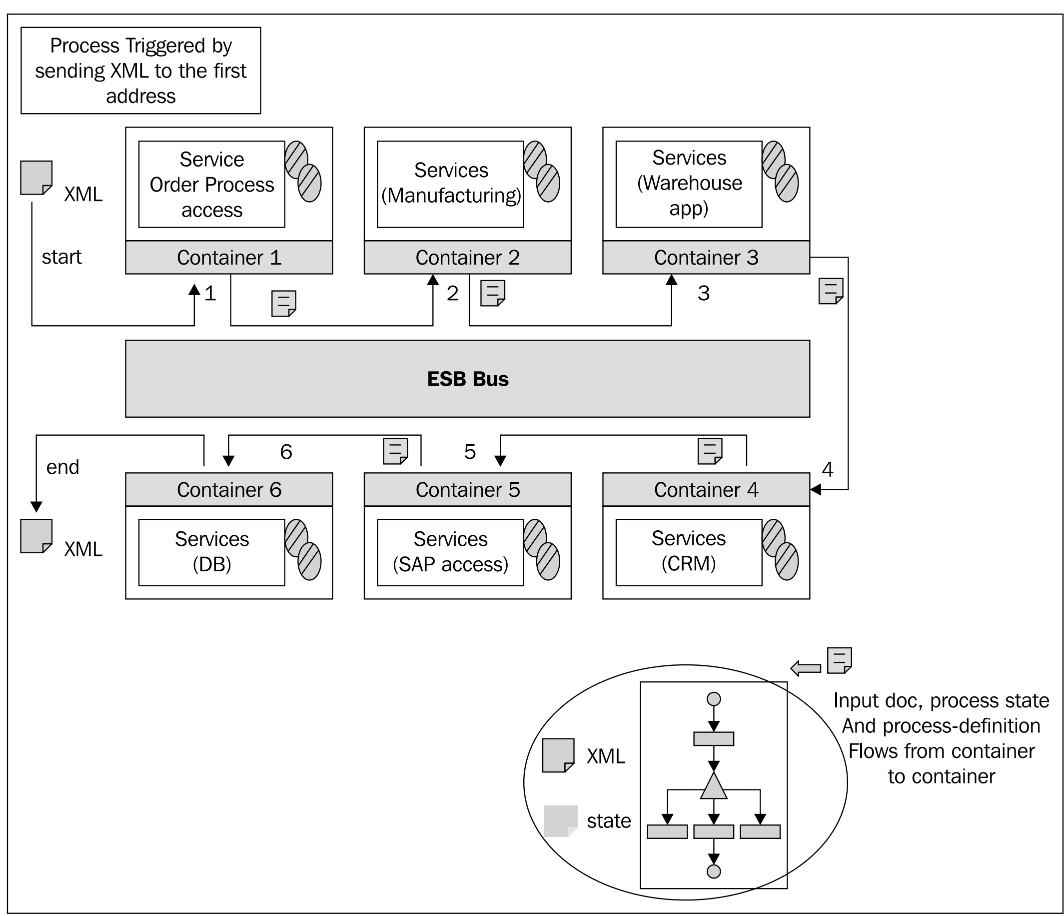
The distributed process execution model can be extended further where the process view is flipped. From being a sequence of steps, it is a document flowing through a sequence of service executions—with the same document as "input" to all the services in the flow path.

Document Itineraries are ESB Processes

Once a document arrives at an endpoint in an ESB system, the document is processed by a Service—a service may be a business logic chunk or a process. Considering the document flow model of enterprises, the enterprise workflows involving a form or a document getting processed by various departments as it flows through the organization can be automated. In ESB, these are called Itineraries. When there is a document that needs to be processed, the processing steps are attached to this document—much like a business process.

The document and the process along with the state information on what has transpired so far with that document "hops" from one container to another in the bus as it gets processed by various steps of the process definition that is attached to the process. From an application standpoint, the process definition is all that matters and the expected end behavior is the same regardless of whether this process runs in a process server/engine or runs as an ESB Itinerary on the bus. The difference is only in the internal execution.

On the lines of a document flowing and getting processed in an organization, the ESB processes have a process state that "flows" and gets processed in the "fabric". This is shown in the following figure. Instead of the process running in a process server, the process input document and process state "hop" from one service container to another to execute the steps of the process. After executing one step, the infrastructure inspects the process and identifies where the next step must execute, and "sends" the process input and state to that container, where the next step is executed and the state updated. And that container sends the process to the next "step".



Consider a work-order query process, where an Order Processing application is accessed to get a work-order detail; this work order is checked for build status from the Manufacturing application, then there is a Shipping status check in the warehouse application, followed by an update to a customer record in the CRM application and then SAP is accessed for the shipping and logistics module and finally a DB update is performed before the order status details are sent out of the process exit. Now each of the six steps is a service in the SOA system, each running on a different container—each container containing services accessing a certain back-end/legacy system. The input document "flows" from one container to the other and gets processed. Each container "sends" the document to the next step. And the last step sends the output document out of the process environment.

Here the process essentially runs in the ESB fabric, as opposed to a normal process that runs in a process engine or orchestration server. In the two cases though, the process programming definition is not different, other than possibly a restriction that there be only one document input in one case, and no such restriction in the other. Otherwise, the process has inputs, processing steps, and flow constructs in both cases, possibly written in a business process programming language such as BPEL. In effect, the application developer's view of the process is independent of the specific process execution environment and mode. Whether it runs in a Process Server or runs as an ESB process in a distributed fabric, the process definition remains the same, say, a BPEL program.

The runtime differences are modeled into the deployment step that the platform may support, that will be used when taking the project artifacts such as a BPEL program and configuring the run-time behavior—say at this point choosing to have the process be available as an ESB process as opposed to having it run in the Process engine, assuming both options are available in the chosen ESB platform.

Itineraries—Processing Schedules for Business Documents

ESB Processes may be represented at run time as a process document with a processing itinerary attached to the document—and this is "executed" in the ESB environment with the document being sent along with its state and itinerary from one step to the other. The document or message, and the itinerary, "flow" and get processed as per the itinerary.

In traditional organizations, there is always a concept of a mail flow where a document or docket has a flow path or itinerary. Based on the document type, this gets set and then flows per that process, with each recipient along the way "processing" the document and adding any additional info or status/updates to the docket/document/form as it flows. ESB Processes follow this same model.

ESB Processes or Itineraries are a key component of the process in the fabric model, wherein, along with the input document, a schedule of processing is attached. The schedule is the process definition—the series of steps that need to be executed as part of the processing. The series of steps can also be seen as a processing "itinerary"—the order of steps that will be executed.

All the steps will get the input document and additional process state that flows along with the document and the itinerary. The process engine is now a distributed engine, where there is an itinerary-handling component in each container. It executes the service step to be run in that container and does the necessary housekeeping to locate the next step in the process (itinerary) and ship the request including the input document, process state, and the itinerary (process definition) to the service container that can run the next step.

ESB Process vs. Orchestrated Process

ESB Processes or Itineraries are still an emerging concept. Processes have so far been known to run only in an Orchestration server or a central process engine. Most vendor implementations support this model. A few ESB vendors are now beginning to support a distributed process execution framework wherein the processes run in the fabric. As Orchestration servers run in a hub-and-spoke model, there will be a performance overhead due to the additional network roundtrips involved with each request and response flowing as opposed to the ESB process model where the requests and responses are always local within the container and the network flows are essentially a single hop between requests where the process state and other information flows between containers.

Orchestration servers, however, do provide for centralized housekeeping, which will allow for tracking all process instances. Work lists or user interactions amidst process steps are another capability that is easily provided by Orchestration servers. For an ESB process to support this functionality, it will need a centralized administrative server that will be notified of process state transitions and work list requests. This will take away some of the performance benefits from running the processes in the fabric in a distributed environment.

Security and Transactions

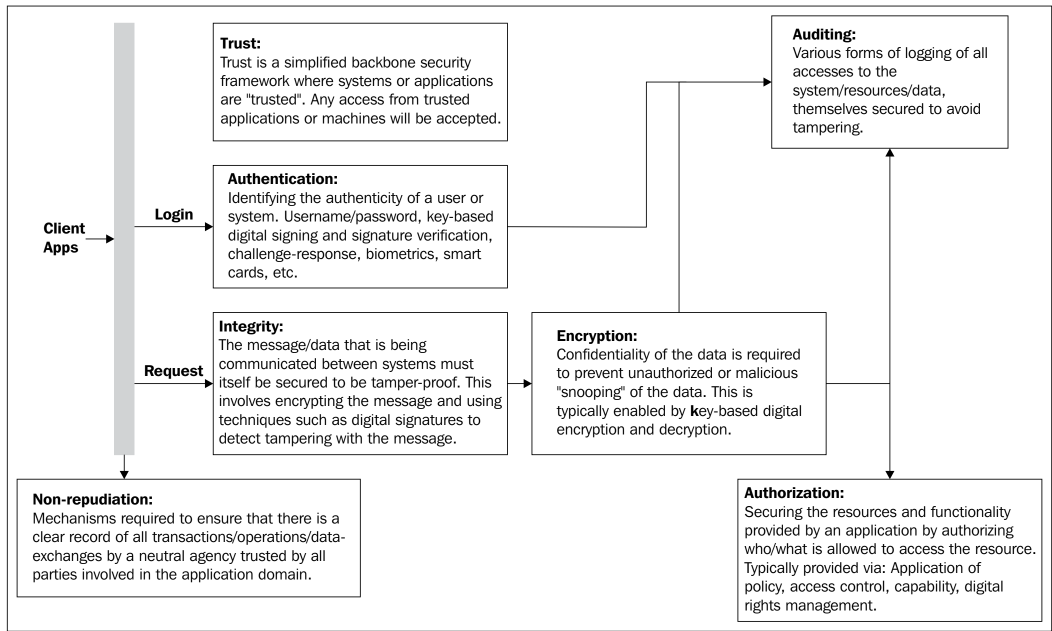
Continuing the discussions from Chapter 2 on the security and transactions standards in the SOA space, we will now see how these apply to ESB. The requirements are essentially the same in an ESB environment as well. After all, ESB is a technology for SOA. Security and transactions are realized by a combination of handling built into the application and capabilities provided by the infrastructure. The infrastructure capabilities are available out-of-the-box for use by the SOA applications. In some cases, the usage is realized by explicitly programming for the security or transaction handling.

The ideal case, though, is when this is available declaratively, without needing any programming in the services or the processes. The required behavior is triggered by the deployment-time setup or the run-time configurations. We will now see how ESB provides for the security and transactions per the relevant Web Service standards.

Security Considerations in Integration Architecture

SOA-based integration environments are back-end systems that provide programmatic access to other systems—and not for end users as in a normal web application. Security models in such environments will be a little different from normal authentication that web applications will require, ranging from simple trust-based access, where unlimited access is provided for specific applications or

network addresses to more complex security tokens. Apart from secure communications, the infrastructure will need to be able to secure the services being built, have mechanisms to get user login credentials, and be able to propagate the authentication details across the ESB domain, between service containers—and along the way, manage the privacy and trust, as shown in the following figure.



In a simple integration environment, which is essentially in a P2P mode, the security provided for by the legacy application can be directly supported in the integration application, without necessarily requiring any common security infrastructure across all legacy systems.

Building on Application Platforms Security (.NET & Java EE)

When the services infrastructure is built on Application Platforms, the security infrastructure provided by the platforms could be used. Both the popular application platforms, Java EE and .NET, provide extensive services frameworks ranging from Secure Socket communications (SSL), to elaborate authentication and authorization frameworks.

The Java EE platform provides for a highly evolved Security framework via JAAS (Java Authentication and Authorization Services) and other constituent specifications. This is in addition to the very flexible declarative application security model for both Web and ESB application modules.

When developing Web Services in Java EE, the service implementations will either be Servlets, or Session Beans or Message Driven Beans. All the three have security semantics that could be defined even when they are to be accessed as a Web Service.

For Web Services built on the .NET platform, the .NET security infrastructure can be used. This includes the capabilities provided by ASP.NET and IIS.

Since Web Services are part of ASP.NET, and these are hosted by the ASP.NET runtime, the security support provided by ASP.NET will also be available for Web Services. ASP.NET works with IIS and the Windows operating system in order to implement the security services. In ASP.NET, most of the IIS settings have been replaced with configuration files. However, security settings made in IIS are still valid in some areas because IIS is still handling the process of accepting users' requests. In fact, whenever IIS receives requests for some ASPX page, ASMX Web Service, or any other resource that is associated with ASP.NET, it uses the IIS applications mappings to send the request to ASP.NET.

ESB Security—Built on WS-Security

Security includes the message privacy/integrity to ensure tamper-proof communications and the authentication and authorization for access to the bus and the services. The security in ESB systems builds on prevailing Web Services security standards.

WS-Security and Related Specifications

In Chapter 2, we discussed the security standards that are coming up. All aspects of the security infrastructure discussed there are relevant to ESB infrastructure. The security standards that are coming up to address various aspects of the security infrastructure include WS-Security, WS-Trust, and WS-SecurityPolicy.

WS-Security includes authentication, integrity (via Digital Signatures) and privacy (via Encryption). WS-Trust provides a framework for trusted exchange of messages, where the sending entity is trusted by the receiving entity. The web service security model defined in WS-Trust relies on services specifying the requirements and an incoming message will have to prove a set of claims (e.g., name, key, permission, capability, etc.). If a message arrives without having the required proof of claims, the service will ignore or reject the message. A service can indicate its required claims and related information in its policy as described by WS-Policy and WS-PolicyAttachment specifications.

Security across the Enterprise

ESB in many respects adopts the various Web Services standards. Security in an ESB platform can be both native and driven by standards. ESB security has two distinct manifestations:

- Security for accessing services and processes from within the ESB domain (internal accesses)
- Security for accesses to the service or processes from outside the ESB domain (say, as standard web services)

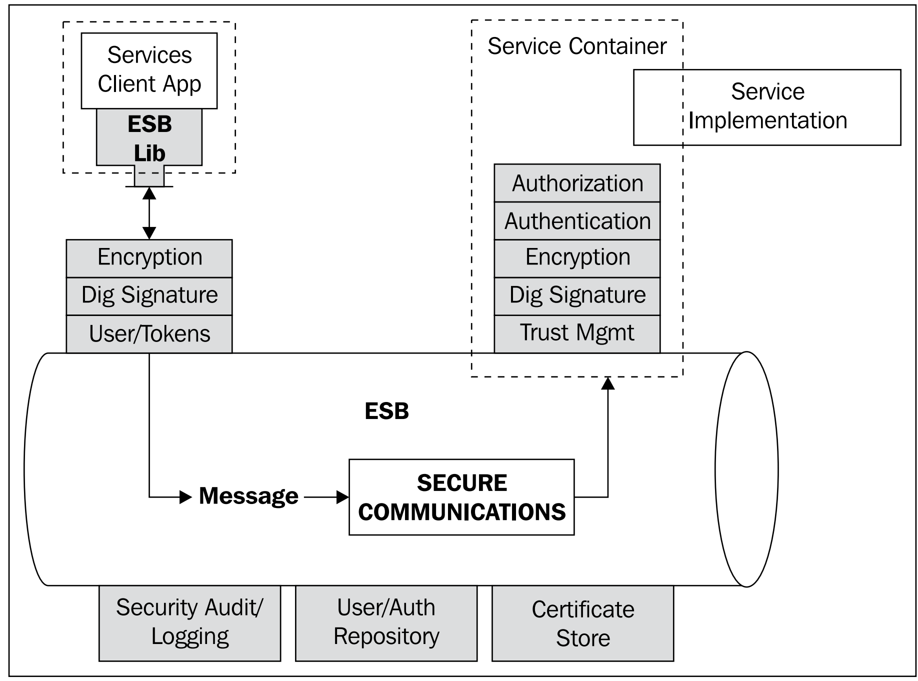
Security and reliability for external accesses should be provided by a standards-compliant platform using the WS-S and WS-RM standards. Internal security also should be defined by using the same standards. In effect, the contract of a service includes its security and reliability assertions that the ESB platform will honor in both its internal and in its external accesses.

Additional internal accesses could leverage any proprietary security infrastructure. For example, in JMS-based ESB implementations, the JMS security infrastructure could be used as is. This should not be preferred by SOA application developers though, as this introduces a potential vendor lock-in.

One important consideration for applications is that the security should be non-programmatic. This ensures that the application program artifacts such as the services and business processes themselves are not locked into any vendor-specific support. The security should be declaratively available. Preferably the requirements should be specified in standard ways such as using WS-Security in the service WSDLs.

ESB can support security at multiple levels including transport layer, messaging layer, service containers, and Web Services gateways with the Domain server at the core of security services.

The transport layer security assures the privacy and integrity of the communications. These can be controlled by the SOA integration application via the WS-Security policy assertions. The ESB platform will honor the assertions and ensure that all communications conform to these assertions. While the specification of the integrity requirements is based on the standards, the actual implementation is platform specific and is built into the lowest communication layers.



Message-layer security will rely on the messaging platform—most ESB solutions may rely on a messaging platform. Messaging platforms, either based on the Java standards or otherwise, have a clearly defined notion of connections, sessions, and the actual sending/receiving of messages. Security is built at all levels. The user credentials are generally provided when establishing connections to the messaging platform. The integrity and privacy is built into the transport layer of the messaging solution, the need for which is specified as a connection property. Access control is built into the destinations. For each destination, the allowed users and roles are specified, which are used to authorize access to any destination, either for sending messages or receiving messages. When an ESB platform is built on an underlying messaging solution, the well-defined security infrastructure of the messaging platform can be leveraged. This will involve delegating a lot of the security handling to the messaging layer. For instance, all access control could be delegated to the message layer's access control for the destinations—wherein, a service endpoint is translated into a messaging destination.

A more elaborate services-level security is possible in an ESB given the extensive mediation infrastructure that ESB platforms include. This could involve complete support for WS-Security including integrity via digital signatures, confidentiality via encryption, and user authentication using passwords or security tokens. Unlike the messaging layer security support, this layer could be based on more generic transport and communication layer-independent approaches. When a service is set up with WS-S policy assertions, the ESB platform will enforce all security assertions even when accessed from outside the ESB domain via regular SOAP over HTTP.

Transaction Semantics for Enterprise Integration

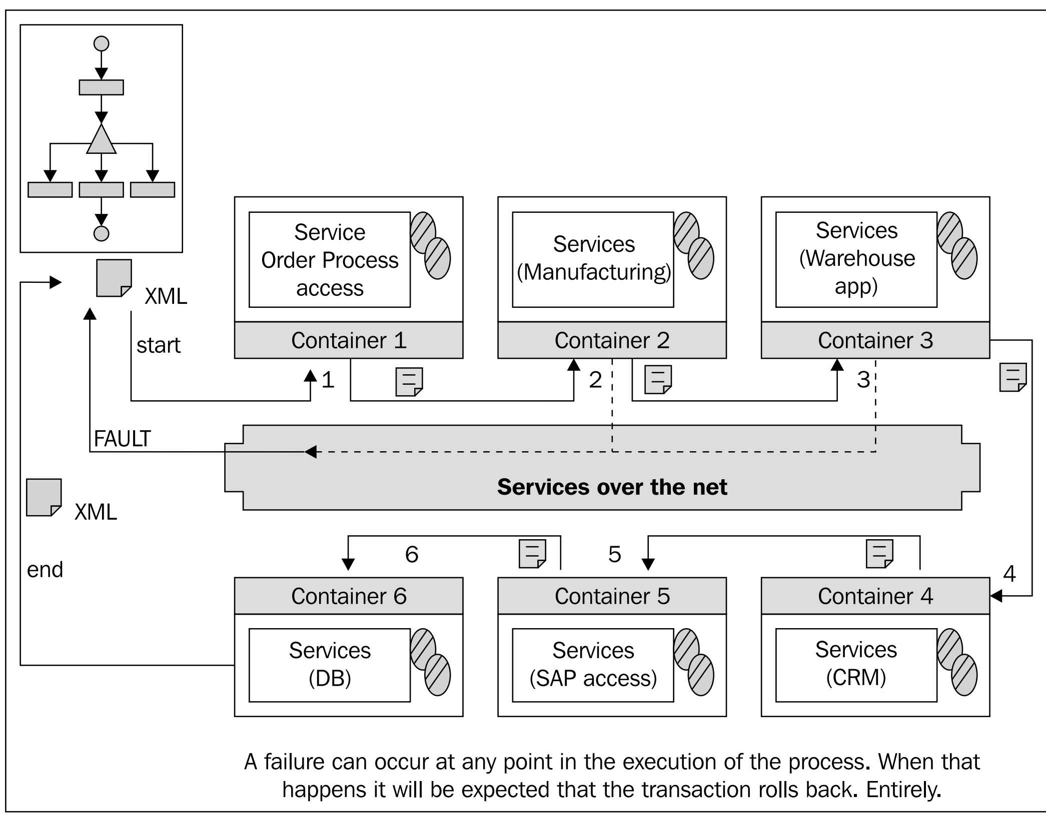
Business applications typically perform aggregate operations wherein one business function is performed by invoking multiple operations underneath. Due to various validations, business logic, and rules, operations could fail. Say when reserving inventory for an order, there might be insufficient inventory quantities. When while executing a series of operations an operation fails, the expectation is that the whole business operation has to fail, and all the state and data in the system be restored to what it was prior to the start of the business function.

We are quite familiar with this requirement in monolithic applications that work off a database. Database Servers and database access mechanisms provide very good support for this via transaction boundaries and abilities to commit or roll back a transaction. With the broad penetration of Application Servers and platforms, inherently applications now started accessing multiple data repositories (databases or the resources). This brought forth the need for distributed transactions. Platforms such as Java EE and .NET have implicit support for distributed transactions—mostly built on the established technologies such as the two-phase commit.

In an SOA or ESB environment, the business function is essentially a business process, and the operations accessed as part of this process are the various services available on the platform. Once these business processes start servicing serious business requirements, it will be inevitable that this will come with a strong need for transactions.

In the Web Services world, the connectivity to the various systems was the focus. Most accesses might be local to one system or a tightly knit domain that probably has a good distributed transaction backbone, and only few sporadic accesses happen to external Web Services. Here the need for tight transactions might not be high; and any transaction semantics can be explicitly programmed for. For example, the compensating operation for a reserveInventory (part number, quantity) function would be a function unReserveInventory (partNumber, quantity). The former function reserves a specified quantity of inventory. The latter function releases the given quantity—explicitly, as part of the rollback operation.

Now, with SOA platforms becoming more widespread and starting to serve as first tier application platforms, the transaction requirements change. The expectation may be that the same declarative transaction capabilities that are available in database and application-server platforms, where transaction semantics can be got without actually programming for rollback operations or compensating business operations, will be available here as well. This is easier said than done. Today, even ESB systems rely on long-running transaction models discussed below.



In a widely distributed setup such as a typical SOA environment, where the service containers may be geographically distributed over a WAN, the infrastructure for transactions becomes a little bit of a challenge (see the above figure). Even so, the minimal transaction semantics expected will be:

- Atomic transactions
- Long-running transactions

Atomic transactions are similar to the database or Application Platform transactions, where a transaction boundary starts, say at the beginning of a Business process, and at any point the process ends the transaction is committed. At commit, all operations performed as part of the transaction will commit their changes, honoring the ACID properties. In a distributed setup such as in SOA, this will be performed using the two-phase commit (2-PC) protocol, where each service or resource that is participating in the transaction will need to support the protocol. Here, the COMMIT now involves a two-step process, first PREPAREd and the COMMITted.

In the PREPARE step, that is propagated to all the service/resource containers, the changes that were made as part of that transactions are taken to an intermediate persisted state in such a way that the changes still remain private to that transaction, but can survive a crash of the service/resource/container. Once all the participating services have confirmed this state, then the transaction coordinator will issue a final COMMIT on each of the resources. Should any of the services/resources come up with an error on the PREPARE state, then all the services are instructed to roll back. Should, after the PREPARE and when COMMITS are underway, any of the services /resources go down or crash, then on its restart the transaction coordinator will "recover" incomplete transactions and issue a COMMIT on those.

Now, in a widely distributed services environment, it is quite likely that some operations take much longer to complete. Say in a Business Process where there is a process step for the "partner confirmation of an order build". This step will complete only when the partner actually manufactures the required part. This may take days, weeks, or months depending on the nature of the part being manufactured. Such transactions are called Long-Running Transactions.

Conventional transaction models will fail or may be sub-optimal in Long-Running Transactions—as they rely heavily on locking resources to ensure concurrency integrity. And for long running, locking resources is a bad model as this will grossly restrict other business processes that need access to the same data/resource from executing. In such cases, a different transaction model is be needed.

A common approach for these is to have the notion of a "compensating" business operation. For every operation performed, there will be its complementary pair that will provide the "compensating" function. Like, if reserveInventory () is a service, there might be an unreserveInventory () service as its compensating operation. These are much more involved than regular transactions.

In normal atomic transactions, most of the housekeeping and transaction management is managed by the infrastructure. The application need only bother about the transaction boundaries—start, commit, and rollback. In a long-running transaction environment though, the compensating functions are to be implemented and made available by the application. Further, the semantics of such compensating operations are also extremely tied-in to the business processing involved. Say, for the unreserveInventory () operation, the call must be made for the exact same part with the exact same inventory count as was used for the original reserveInventory () operation request. The onus for ensuring such integrity now rests with the application developer.

Transaction Strategies for EAI and B2B

Enterprise Integration using any B2B solutions or EAI essentially involves a hub-and-spoke model, where there is a single central server instance that manages all interactions with the service back ends. In this mode, the distributed transactions involving multiple resources and legacy applications can be managed by having the transactions coordinated out of the EAI server instance. Being managed by a single transaction manager, which is most likely running along with the EAI server instance, existing transaction manager solutions will fit in very well. Any transaction manager and existing two-phase commit protocols can be used to provide the transaction boundaries across access to various services from the EAI platform.

In simple B2B solutions, conventional transaction management solutions will not work. This is primarily because, over an unreliable WAN or internet network, having a tight wire protocol for transaction management is not practical. Here, alternative transaction management solutions such as those that depend on compensating business transactions are warranted. In such solutions, unlike the normal commit and rollback, undoing a transaction's operations is realized by invoking "compensating" business operations. The compensating operations are regular business operations that perform the converse of another business operation.

Distributed Transactions and Web Services

Web Services are designed to be available over intranet or Internet, over high bandwidth networks or loosely coupled networks. When the services are built on EAI or Application platforms, the transaction capabilities of the platforms come into play. These will generally support the atomic transactions semantics, either simple local transactions or per the two-phase commit protocols for distributed transactions across multiple resources. In a more involved distributed-services infrastructure such long-running transactions also can be supported.

Web Service Transaction Standards

Transaction requirements in an SOA platform are addressed by WS-* specifications. The Web Services Transactions specifications describe an extensible coordination framework (WS-Coordination) and specific coordination types for both short-lived ACID transactions (WS-AtomicTransaction) and long-running business processes (WS-BusinessActivity). This provides transaction solutions for the short-lived ACID-enabled business-process coordination within the enterprise as well as the more extended long-term transaction coordination across the partners and the supply chain.

The WS-* standards provide a framework for transaction boundaries to span heterogeneous service environments. In the case of ESB, this becomes a little simplified as there is a homogenous layer introduced by the service containers. The WS transactions specifications are designed to enable interoperability of distributed, recoverable transactions.

The WS-Transaction specifications are defined by the Web Service Interoperability Organization (WS-I Organization), which is an industry-wide effort at standardizing how Web Services are requested and delivered. This includes WS-Coordination, WS-AtomicTransaction, and WS-BusinessActivity.

WS-Coordination: This specification is a generic extensible framework for describing protocols that coordinate the actions of distributed applications. Such coordination protocols are used to support a number of applications, including those that need to reach consistent agreement on the outcome of distributed activities. The distributed activities could be transaction coordination, audit logging, or any other distributed activity that requires central coordination and housekeeping. The WS coordination is generally itself a service in the platform, thus ensuring easy access.

This specification describes a definition of the structure of context and the requirements for propagating context between cooperating services—a mechanism for an individual Coordination service to participate in another Coordination service enables extending the Coordination domain. In effect one SOA or ESB domain, could integrate its coordination space into that of another ESB domain that has its own Coordination service.

WS-AtomicTransaction: This specification provides the definition of the atomic transaction coordination type that is to be used with the extensible coordination framework described in the WS-Coordination specification. The specification defines three specific agreement coordination protocols for the atomic transaction coordination type: completion, volatile two-phase commit, and durable two-phase commit. Developers can use any or all of these protocols when building applications that require consistent agreement on the outcome of short-lived distributed activities that have the all-or-nothing property.

WS-BusinessActivity: This specification provides the definition of the business activity coordination type that can be used to effect compensating business activities for long-lived transactions. The specification defines two specific agreement coordination protocols for the business activity coordination type: one involving the participants to drive the coordination and the other involving a transaction coordinator. Developers can use any or all of these protocols when building applications that require consistent agreement on the state or outcome of long-running distributed activities.

Architecture: The architecture for WS-transaction is not too different from that for conventional transactions. There is a Transaction Coordinator, and transaction-enabled services. Each transaction-enabled service will need to register with the transaction coordinator. As part of this registration, the compensating business operation is also specified. Each business process will register the process instance with the transaction coordinator and, on successful completion, will again notify the transaction coordinator. The TC will do the needful to ensure all services that participated in the transaction have completed their parts. Likewise, if a fault occurred then the process will notify the TC of the same, and the TC will do the needful to roll back all services that participated in the transaction. The rollback may involve triggering the corresponding compensating business operations.

Realizing Transactions in ESB

The transaction models in Web Services-related transaction specifications form the core of typical ESB transaction support. These are extended to leverage the infrastructure capabilities like the underlying messaging infrastructure.

ESB Transactions Built on the Messaging Layer

ESB platforms may provide support for the emerging WS-* standards or could provide transaction support more natively. The primary use case for transactions in an ESB system might be local to the enterprise and it could be unlikely that they will be long lived. Hence regular atomic transactions with a commit and rollback model might suffice most cases. There is the likely standard WS-AtomicTransaction-based implementation.

With messaging being the actual transport layer for the services, the more proprietary solutions may leverage the transaction handling available at the Messaging layer.

Process Driven Local Transaction Semantics

In any SOA application, processes are the primary access points for services. The services are orchestrated in a process. The processes perform the aggregate business functions that involve access to multiple services in the enterprise. It is quite likely that these processes themselves form the transaction boundaries. In cases where the process accesses the services in a local process space, there could be transaction optimizations possible where the distributed transaction now becomes managed from a single process space.

This could use regular two-phase commits, with an in-process transaction manager that manages the transaction coordination and the resource interactions per the two-phase-commit protocols.

Reliability, Scalability, and Management

ESB enables enterprise-grade reliability and scalability through its inherently distributed platform. Having services abstracted behind "endpoints" as opposed to explicit IP and port and specific URLs of each container is the primary enabler for high-end reliability and scalability.

While reliability is described in terms of the availability and service delivery guarantees, scalability is the ability to service higher loads without disrupting the environment or needing major reconfiguration. The reliability is required to ensure that each request to a service is delivered accurately and predictably. And scalability ensures that a large number of such requests are handled by the ESB platform.

Unlike simple Web Services or even platforms such as EAI, ESB is inherently a distributed infrastructure. It is not single-server-centric as in EAI or handling libraries as in simple Web Services. Reliability can be built into the transport and messaging layers of the ESB infrastructure.

Reliability Concepts

Reliability is essentially at the level of reliable communications and high availability of the whole ESB platform. Reliable communication is at a base platform level where there is an expectation that all communications are guaranteed to complete and be failsafe. Additionally, reliability with regard to communication expectations in a widely distributed ESB domain may also be specified by the SOA application. There are standard mechanisms available like WS-ReliableMessaging that allow specifying the reliability expectations.

High platform availability is an attribute of the platform. The only expectation from the application is that the platform be available all the time. Sufficient redundancies via clustering and fail-over have to be built.

There have been several analyses and studies that touched upon how widely Services will be used in the rapidly evolving SOA landscape. The inter-application and cross-application accesses will increase drastically with:

- SOA becoming more easily available
- Services exposed by all applications and legacy systems easily accessible from other applications
- Business processes easily composable from these services

This is much like how DB accesses have become omnipresent in applications today. Even today, if when processing a sales order, the programmer could easily access the customer database maintained by the CRM application to get the latest info on the customer, he or she would do that. But given that such accesses are expensive today, most often there would be a discreet "upload" of customer info from the CRM application to the Order Processing Application—with the info duplicated. And then, there are the Business processes.

The BPM tools have gained in popularity as Business Processes help integrate the IT-Solution Islands that currently exist in enterprises. And these solutions are all interconnected in the functioning of the enterprises. In the absence of any integration solutions, the end user is the integration point: the user accesses multiple applications from multiple windows, and performs the "Business Process".

With SOA, both ends of the integration are targeted well. Each legacy solution/ application can easily expose services, and Business Processes can be composed and executed easily. Unlike traditional BPM solutions that are more focused on just the "process" and service access is more from the process environment, in ESB, services are accessed from a generic framework that is agnostic to the processes or accesses. In effect, there are no synchronous invocations of services. The service invocations are always dependent on typical asynchronous mechanisms such as messaging. This imposes potential reliability issues that need to be addressed by the ESB infrastructure.

Reliable Messaging Basics

What constitutes Reliable Messaging? In short it has to ensure that a message is delivered. This is not exactly a new concept. It has been in existence since the EDI days. EDI in fact solved this very well for information exchanged as files. It had sufficient redundancies and communication protocols and extensive acknowledgement and confirmation mechanism to ensure reliable message delivery.

Reliable messaging could include:

- Ensuring messages are delivered
- Ensuring non-duplicate message delivery
- Ensuring a sequence of messages
- Ensuring sufficient bookkeeping to help with non-repudiation

To ensure the above, additional management of the communication will be needed in the communication layers of the ESB. This includes the notion of sequence IDs, message numbers, acknowledgements of communication packets, and persisting messages to ensure delivery even when intermediate infrastructure components fail or crash.

Sequences—Each message set sent from a source to a destination is given a sequence reference ID. The messages sent from the source to the destination are scoped using a sequence. Sequences are distinguished using a unique identifier (a URI) for each sequence. The sequences only ensure that the messages are delivered in the same sequence as that they were emanated at the source. Sequences are controlled declaratively without the service implementation or the integration application that is accessing the services having to program for managing sequences. The sequences are generated and processed by the communication layers of the ESB infrastructure.

Message Numbers—Messages form the lowest level of a communication. In an ESB, each communication packet typically represents one service invocation. Every message sent in the context of a sequence has a unique identifier in the context of the sequence. This identifier is a serially incremented number, with each message in the sequence getting the next number in the sequence. This identifier is a monotonically increasing integer number, starting with 1 and increasing by exactly 1 for each message. Managing the message numbers would be a responsibility of the communication layers of the ESB infrastructure. The source generates the message numbers within a sequence and the destination ensures that the messages are delivered in the right sequence.

Acknowledgements—One important consideration in ensuring reliable communication is to acknowledge the receipt of messages. An acknowledgement is an indication that a message was successfully delivered to the destination. Messages are acknowledged using acknowledgement ranges. Any missing acknowledgement at the source is an indication of a missed message. This is then processed as per the retry specifications at the source. An acknowledgement does not necessarily indicate that the message was processed. This responsibility rests with the application, and will need to use the Fault and Error response mechanism provided by the service bus to report such processing errors.

Message persistence becomes a necessity when needing to ensure reliable message delivery even when any of the intervening infrastructure components crash. Such durability considerations are beyond the basic wire protocol for the communications. As mentioned above (in the discussion about acknowledgements), WS-RM ensures transfer, not processing. Persistence requirements have to do with the storing of the message on the destination until it is processed, and are thus the responsibility of the implementation.

Since persistence is a common aspect of reliable systems, an implementation of WS-RM would typically provide it (at least, as an option). If provided, a typical implementation would only acknowledge transfer after the transferred message was persistently buffered.

It is interesting to note that because persistence is not related to the wire protocol, applications can be programmed with the same simplified communication error-handling model regardless of the persistence capabilities of the system.

WS-Standards for Reliability

Web Services are dependent on communication, even if primarily peer-to-peer, and as with everything else around Web Services the vendor community has come together and defined the ReliableMessaging standards. Any compliant Web Services processing stack, both on the client side and on the service provider side, can process reliable messaging protocols. WS-ReliableMessaging defines the reliability semantics that can be expected and processed in a Web Services environment and the WS-RM Policy specification describes the language to describe the ReliableMessaging requirements.

WS-ReliableMessaging: This specification describes a protocol that allows messages to be delivered reliably between distributed applications in a communication using the SOAP-based protocols. WS-RM is described in a transport-neutral manner allowing it to be implemented using different network transport technologies. To provide for reliable messaging in Web Services, a SOAP binding is defined within this specification.

WS-RM Policy: Reliable Messaging involves the two ends involved in a services communication describing and understanding the requirements well. For example if sequences have to be used or if acknowledgements are required, accordingly, the infrastructure on both ends will ensure the corresponding handshakes and processing happens when sending and receiving the messages. WS-RM Policy is a specification that describes the policy assertions that leverages the WS-Policy framework to enable an RM Source and Destination to describe their requirements for a given reliable message exchange.

The Policy Assertions supported are:

- Spec Version
- Sequence Creation
- Sequence Expiration
- Inactivity Timeout
- Retransmission Interval
- Acknowledgement Interval

The specification also specifies a good mechanism to handle RM faults.

WS-Addressing: WS-Addressing defines a mechanism for identifying generic Web Services addresses that could be used for replies and fault responses. Basically this is a variant of a Web Service, which is meant as an address to send an arbitrary response or fault or message back to a well defined location. WS-Addressing provides a mechanism to describe such locations.

Reliable messaging is one area of Web Services where there is no unanimity yet among the technology vendors and other stake holders. There are standards in place like:

WS-ReliableMessaging: WS-Reliable Messaging is a standard spearheaded by Microsoft, BEA, IBM, and TIBCO. **WS-Reliability** is encouraged by Sun, Sonic, Fujitsu, Hitachi, NEC, and others. WS-Reliability is part of OASIS RM TC. There are efforts underway to try and get the best out of both.

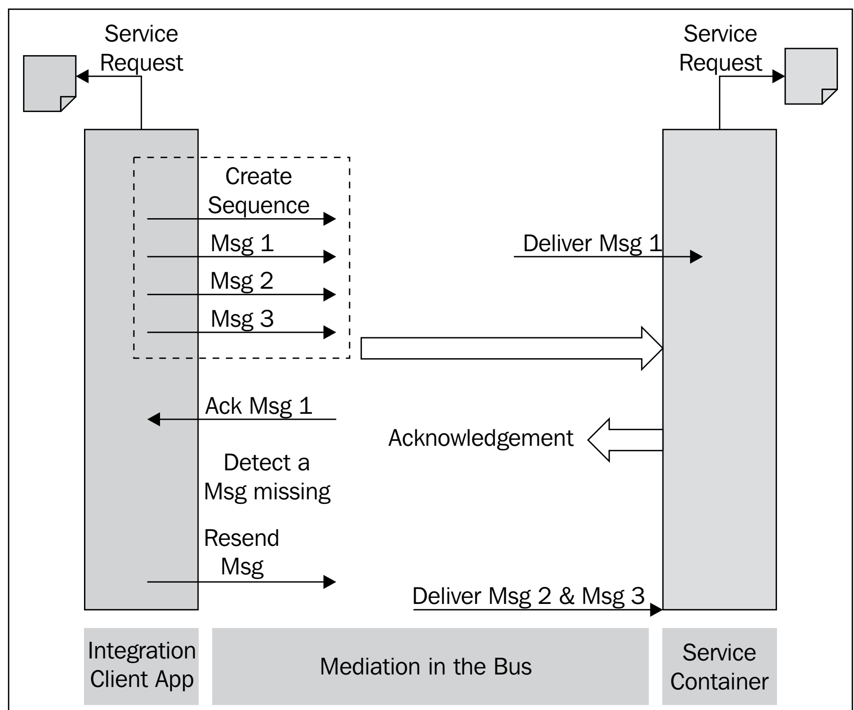
Achieving Reliable Communication through ESB

In an enterprise scenario, the reliability and scalability of an ESB platform (service providers, service consumers, and the ESB communication layers) is extremely critical. Scalability is an attribute of the platform and configured by the deployer. Reliability, though, is an attribute that could be expected by itself from the platform, and also an application could have reliability expectations to service SLAs. The reliability expectations will be specified by the application developer using standards

such as WS-ReliableMessaging. The ESB platform will honor these expectations. Additionally, an ESB platform should support the scalability expectations from any deployment.

While reliability expectations can be specified in a standard manner by the SOA applications, ensuring reliability itself is an internal implementation aspect of the ESB platforms. One common implementation approach for ESB platforms is to have a high-performance messaging backbone as its lowest level infrastructure. Messaging platforms offer message delivery guarantees. An ESB services platform built on such messaging platforms can leverage the message delivery guarantees for service communication reliability as well.

The messaging layers of ESB platforms have a high level of message delivery reliability already built in, as shown in the following figure. This would be available even when WS-RM-based assertions are not set. The WS-RM becomes more important when the same service needs to be accessed from outside of the ESB domain as regular web service. Such accesses will happen through an HTTP gateway that is normally a part of ESB platforms.



Message platforms have very well defined delivery semantics that include persistence of messages when needed. The message layer includes mechanisms to handle acknowledgements, once-and-only-once delivery semantics, built-in retry mechanisms when a delivery fails, persistence of messages to survive crashes, and such. These will all be implicitly available for the services and service access available on the ESB infrastructure that is built on a messaging platform.

High Availability in ESB—Leveraging the Messaging Platform

ESB platforms being built on messaging backbones have leveraged the high availability capabilities of the messaging system. Most commercial Messaging platforms like TIBCO, Sonic, and IBM MQ Series have strong high availability built into the platform. Some messaging solutions even support failing-over a session when there is an active transaction.

High availability would include ensuring the availability of all parts of the ESB infrastructure: the messaging layer, including the message brokers, client tiers, and the communications among clients, brokers, and destinations. This is in addition to guaranteeing the messaging and reliability semantics such as delivery-exactly-once message reliability under both normal and failure conditions.

The high availability solutions may also include hardware and OS-level capabilities such as RAID, OS clustering software, or third-party HA frameworks in the messaging layer. The expectation from the platform is that the in-process operations/transactions continue to their processing destinations without any costly rollback or recovery time.

Given that ESB platforms leverage the "document flows and gets processed" paradigm, the key infrastructure components will be the documents and their delivery to their destinations. Documents here are delivered as messages in the messaging layer. Any service request is now broken down to a document-type and a delivery address. If these addresses can be logical, with physical resolution itself resolved via late binding just at the time of the actual execution of the request, then this "hiding of the actual address" allows for some creative high availability solutions.

Location Transparency at the Core

In ESB, the services are configured on endpoints. Endpoints have a binding to a specific message destination. The discovery of the specific message destination is made very late at the time of actual invocation of a service, automatically by the infrastructure. The process or a service consumer invokes a service only by using its logical service endpoints. Only when the infrastructure needs to actually "ship" the request to the service, are the actual message destination and other specific details located and used to actually communicate.

The infrastructure would use some domain service or manager in the ESB to locate the "physical address" of a service endpoint—the "logical address" that the process or integration application uses. The address in the ESB will be a destination for the message/document to be delivered. The actual address of any service is registered at the time a service instance is started. And whenever the service needs to to be accessed, the same registry is used to get the physical address for the endpoint.

Smart high availability handling is built into this resolution process. The possible location transparency abilities include:

- The service can be available on any container at any point in time.
- If the current container goes down, a new one could be started and service made available.
- When the load increases, additional containers can be configured.
- The implementation of the service can change and it can be redeployed. This can be done without affecting any of the processes or integration application that access this service.

In such cases, the domain manager, would keep track of the various containers and the services running in each container. (The message/document address is essentially the lowest-level resolution for any service. This is again an infrastructure function, to map a service to a destination/address). Each time a service is accessed, the logical service name is resolved to a physical location and the request is sent to that address. This forms the basis for ESB High Availability.

Multiple, Configurable Interaction Models

Given the location transparency of the services, the interaction models between services and between process and services is extremely loosely coupled. The minimal coupling that exists is also based on interaction configurations. When a service is configured, the endpoint is defined and its bindings to the underlying physical layers such as messaging are also configured. The service consumers though, use only the logical endpoint for the services. This ensures that the service consumers are agnostic to the physical bindings of the service. And for the service itself, at any point the actual runtime or execution environment for the service can be changed—declaratively by just changing the bindings—without in any way impacting the consumers and applications that access the service. The service consumption is configurable in that it needs only the endpoint. And the service itself is also configurable by virtue of the bindings being modifiable at any point.

Even the actual service artefacts are also configurable. When configuring a service, the service artifacts such as class names and JARs are provided. And this could also be modified at any point, again, without affecting any of the consumers and integration applications.

In short, an ESB platform provides for multiple configurable interaction models.

Apart from the location transparency, by abstracting the transport ESB provides for other mediated interaction models including synchronous and asynchronous invocation, publish-and-subscribe-based multi-cast invocation, intelligent routing, and stateful services orchestration.

The location transparency and the flexible interaction models form the basis for the scalability and load balancing capabilities in ESB, even as this builds on the message-oriented middleware models.

Scalability and Performance of ESB

Services and processes in ESB platforms are abstracted behind the ESB domain services. Any consumer of ESB services or processes will only connect to the "domain" and not directly to specific services or processes. This indirection will form the basis for the Reliability and Scalability offerings from an ESB platform.

The location transparency offers Scalability possibility in ESB environments. This could also result in Utility Computing where service can be made available on demand on any container. This requires a high level of configurable service definitions and interactions that ESB anyway offers. While SOA seems to have got a very firm footing in IT organizations, it remains to be seen how the high scalability that Utility Computing offers will be available in ESB. With the advent of SOA, there is a new framework possibly in the making that will accelerate Utility Computing. But one impeding factor, though, is that while SOA provides a very powerful abstraction to provide a common integration fabric for the organization, it still doesn't take away the actual business application execution environments that sit behind this fabric. This is a problem that cannot easily be solved in the case of IT environments with very many application infrastructure platforms (like Java EE, .NET, mainframes, ERP apps, custom applications and more). With the ESB's neutral services framework and the location transparency, this is made possible, though. Someday, ESB will also make possible services grids—very highly scalable and dynamically configurable and reconfigurable services environments.

Something to think about—Application Grids:

- A grid of generic App Server cells (Java EE or .NET)—Each cell is an out-of-the-box Java EE server—with any OS and VM vendor. The application server grid component will run on each cell. Cells can go down and come up at will—grid will be unaffected.
- The work units that run on the grid will be Java EE applications. The grid presents one logical processing environment with a bunch of loosely coupled cells.
- Load distribution and fail-over will be dynamically managed by the grid.
- No single point of failure in the grid. The application server will manage the grid abstraction. Applications available on the grid Distribution are strategy specifiable (what application is available on how many nodes).
- Scheduling and Provisioning managed by the grid.

Implicit Service/framework components:

- Grid-aware naming service
- Application repository—what and where
- Distributed Administration service—for remote administration
- Cluster service—self-organizing, app management, failover++

Missing Pieces:

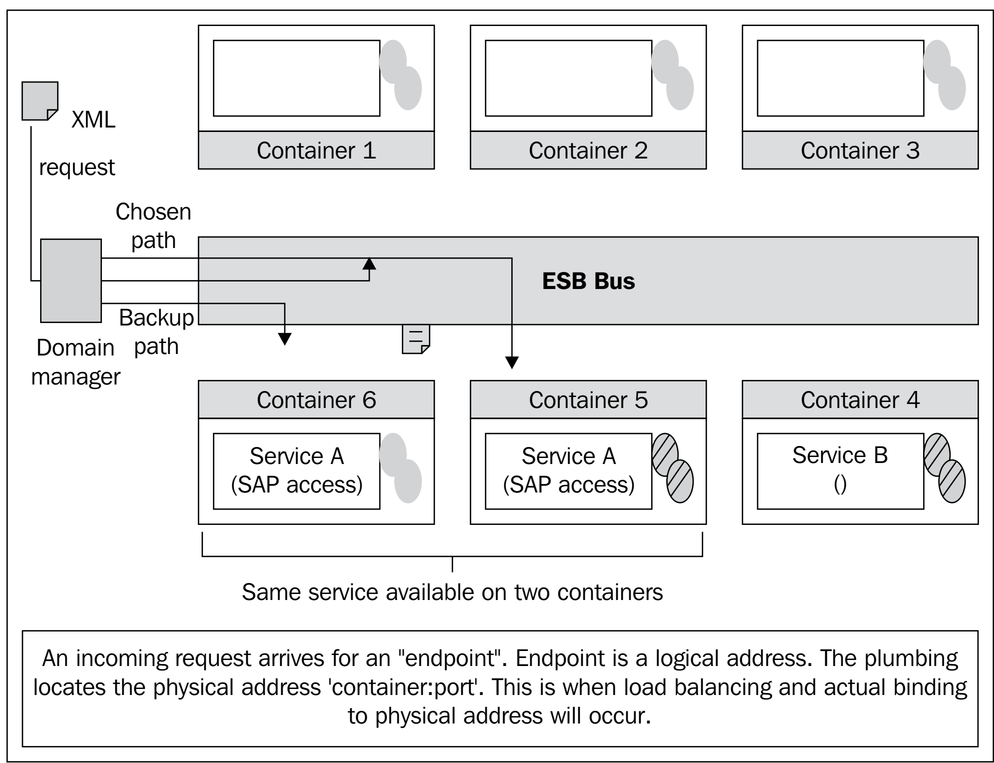
- Smart Provisioning
- Ease of Management/Administration of the Grid
- Binary management of all nodes on the grid
- We need a Federated Naming Service and a Federated App naming service

Load Balancing and Load Scaling in ESB

We have seen how the abstract address of a service in an endpoint-driven service ESB environment helps with balancing load for any given service. The service can be configured on multiple containers and allow taking on higher number of requests for the service.

ESB offers another interesting possibility with regard to parallel processing. ESB systems being message (document flow)-oriented, a service request is essentially a message sent to the service's endpoint. And at that endpoint, there will be some infrastructure component that waits for the response message from the service. This would mean that by sending multiple messages (for service requests) at any given time, concurrent execution of services is processing. Enabling parallel processing of parts of a given process is as shown in the following figure.

The parallel branches can be triggered one after the other and then the process could wait for the response messages from the invoked services. If a program is managing the concurrent parallel execution of the parts of a common process, then this program will need to have the logic for collating the responses and determining when the process could continue in its path—after ensuring that all parallel parts have completed. In processes engines, this could become more implicit. The process engine when detecting parallel branches in the defined process can trigger the service requests from each branch and then wait on the responses.



With Load Balancing as the basic functionality, a complete scalable infrastructure is made possible in an ESB. This includes reliable guaranteed message delivery, clustering of the various individual components such as the domain server or the services registry, continuous and high availability of the whole platform, high performance built into every component of the infrastructure with the most critical components of service containers and the messaging and communications infrastructure at the core. Additionally high-performance federated messaging domains play a key role in ensuring reliable and performing interactions in a multi-geography widely distributed enterprise.

Looking at one of the leading players in the space, some of the functionality provided is:

- **Continuous availability**: Provides high availability and transactional fault tolerance. Completely transparent to services, in-process transactions continue without recovery process delay or rollback. Eliminates operational risk of lost data without expensive RAID, OS clustering software, or third-party HA frameworks in the messaging layer.
- **High performance**: Industry-leading high-throughput and low-latency communications, including high-volume/high-availability scenarios (durable, persistent) and high QoS scenarios (durable, persistent, transacted). Provides fast service response at a lower cost of hardware for a given throughput requirement.
- **Clustering**: Scales service throughput and ensures constant response time by load balancing over clustered servers. Also, allows deployments to scale to support large numbers of messages, users, and applications.
- **Dynamic Routing (DRA)**: Supports global service namespace across large, distributed deployments. Routes data and process flows across clusters and sites without manual gateway reconfiguration. The specific load-balancing algorithms are specific to the ESB platforms. Typically, they will support round-robin balancing or available-resources-based balancing.

Control and Management of ESB

ESB being an infrastructure platform targeted at enterprise usage, control and management of the platform are necessary functionality that would be expected. Managing ESB infrastructure includes managing the domain, containers, and services, and monitoring and configuring the reliability, security, and performance of the platform.

Broadly, the management and control involves:

- Configuring the domain and the containers
- Managing the lifecycle of service types, services and processes, from deployment through configuring and monitoring the service and process instances
- Monitoring the business processes and services activity and operational statistics
- Securing the platform
- Tuning and optimizing the platform

The manageability starts from setting up and configuring the domain and containers to be able to set up service types and service instances and managing the lifecycle of a container or a service. As the ESB platform is still emerging with any standards required yet to be defined, in general ESB will follow the Web Services standards, including management. With Java being one of the implementation choices for building ESB infrastructure, Java management standards including JMX may also be relevant. Other existing distributed management technologies such as SNMP will also bring managing Web Services and ESB into their fold. Let us take a quick overview of the relevant management standards and then understand the specifics of managing an ESB platform.

Controlling and Monitoring the Services

Control is essentially the management power made available to the users and systems administrators. Given the inherently distributed nature of SOA applications, if to this mediation power is also made available, then this puts quite some challenge in managing the services, processes and the instances. This will require that the service containers be configured remotely. It will require that a given container configuration be "replicated" quickly to create more "instances" of the containers to take on additional load. It will require that the services and processes be configured separately, and declaratively be made available on multiple containers. And over and above everything, it will require that all the administration be done remotely.

Good "Control" implementations will rely on a logical deployment model that defines the containers, their operating environment, and the service to be made available on the containers. Once defined, these can then be "bound" to physical runtime environments without any additional configuration. Once bound, all the services and operating environment defined in the logical model are available in the physical container.

The model could be extended to individual services as well, where the services are configured once. They are given a name and an endpoint and any initialization or environment parameters are defined once. Once this service with a name is defined, instances can be deployed onto any number of containers.

In addition to the goal to automate management as much as possible, it will also be necessary to enable humans to investigate problems, find root causes, and take action to correct the issues that are discovered. Where the management infrastructure is automated, it codifies the rules that drive simpler diagnosis/action scenarios in a form that is machine readable. This infrastructure may also log and/or warehouse request content for later analysis or auditing.

Management Information Standards

Three generic multi-platform management standards are Java Management Extensions (JMX), SNMP (Simple Network Management Protocol), and Distributed Management Task Force (DMTF). SNMP is a standard to be aware of given its very wide acceptance in the management domain over the years and also extensive existing management infrastructure that uses SNMP—though SNMP may not have anything specific for the management of Web Services and distributed services over ESB.

The Java Management Extensions (JMX) technology is a technology standard developed by the Java Community Process for management and monitoring of infrastructure and applications using Java to build the management solutions. JMX is domain independent and can be used to build management solutions for platforms ranging from legacy systems to those that may come up in the future. JMX management architecture provides for three layers—Instrumentation, Agent, and Remote Management. Instrumentation is the lowest level of management operation layer that collects data and manages any administered unit or entity. Agent is a proxy for the management tools and front ends to work with the instrumentation tier. Remote Management is an extension or mechanism for tools to be able to access the agents and the instrumentation remotely.

The managed resources or entities such as applications, devices, or services, are instrumented using Java objects called Managed Beans (MBeans). MBeans expose their management interfaces, composed of attributes and operations, through a JMX agent for remote management and monitoring. The main component of a JMX agent is the MBean server. MBean server is the repository of all MBeans—kind of like the boot-strap for the management domain. A JMX agent includes a set of pre-defined services for handling MBeans.

Remote Management is provided via Protocol Adaptors. Protocol adaptors and standard connectors make a JMX agent accessible from remote management applications outside the agent's Java Virtual Machine (JVM).

As with most other aspects of Web Services, OASIS is spearheading the standards definition activity in the area of Web Services management. Interestingly, there are two very different views to Web Services and Management:

1. Management of the Web Services
2. Management of any distributed infrastructure/applications/domains using Web Services

The OASIS Web Services Distributed Management TC is defining two sets of specifications, one for each of these aspects: Web Services Distributed Management: Management Using Web Services (MUWS) and Web Services Distributed Management: Management of Web Services (MOWS) specifications.

Another organization spearheading management technologies is DMTF (Distributed Management Task Force), via its working group formed out of the member companies, it develops various documents, guidelines, and standards specifications for Distributed Management. These include CIM (Common Information model) and Web-Based Enterprise Management (WBEM). These are standards for distributed management using Web Services—though the view is one of managing any distributed infrastructure, and there is no specific description of capabilities for managing web services themselves. However, as these are by and large domain independent, the same approach could be used for managing web services and their operating environments and infrastructure as well. DMTF is adopted by many distributed application infrastructure vendors including Oracle, Microsoft, and Sun.

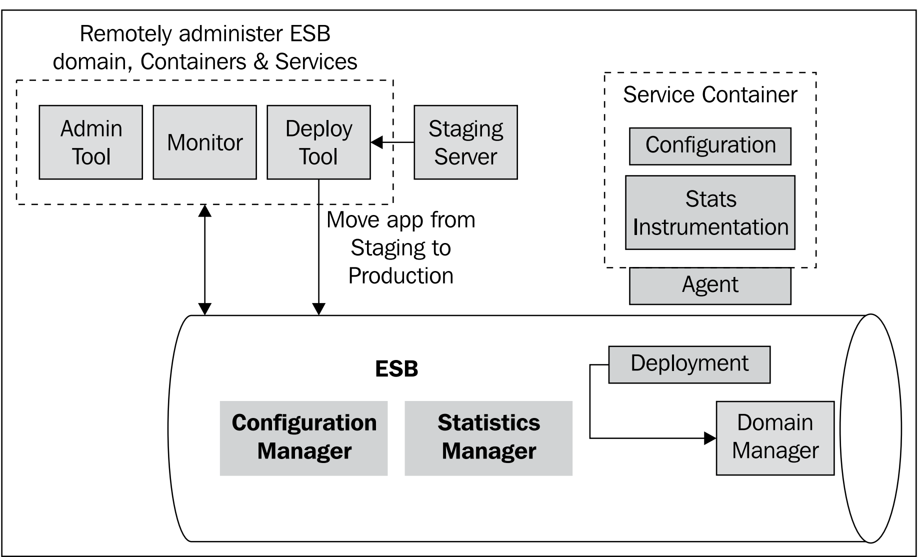
CIM is a common data model of an implementation-neutral schema for describing overall management information in a network/enterprise environment. WBEM is a set of management and Internet standard technologies developed to unify the management of enterprise computing environments. WBEM is built on other DMTF standards including CIM. The Common Information Model (CIM) provides a common definition of management information for systems, networks, applications and services, and allows for vendor extensions. CIM-XML is an extension of CIM, and as a part of WBEM, provides the Representation of CIM using XML. WBEM Discovery using Service Location Protocol (SLP) and WBEM Universal Resource Identifier (URI) mapping are two standards that provide a way for applications to identify and interact with WBEM management systems, capitalizing on existing standards and protocols to enable rapid development and deployment of management solutions.

Infrastructure for ESB Management

ESB is a distributed services infrastructure. To be able to deliver such a platform, a distributed management framework will be an inherent part of such platform. The platform will include a domain manager and the containers that will need to be configured and managed by the management solution that is part of the platform. The management solution will include tools and agents. The tools part offers the management front end for the users and administrators, and the management agents and utilities will be part of each container and infrastructure component that provides the necessary management functions pertinent to that component.

Some of the common management functions that can be expected from an ESB platform will include centralized management support, remote administration of the domain and the containers, configuration of the components, deployment of the application artifacts, configuration of the services, monitoring, and quditing and logging. While there are no mandated standards yet, the more the standards in any ESB platform, the greater the flexibility in managing the platform. Best-of-breed tools from other vendors could be used for managing the platform and also enterprise-wide multi-infrastructure management tools such as Tivoli or Unicenter could be used to administer the ESB platform along with the other infrastructure such as database servers in the organization.

Let's look at some of the management functions that will be expected from an ESB platform, highlighted in the following figure.



Centralized Configuration: Built-in framework for managing ESB infrastructure and services. Supports management of a large deployment from a single console. This starts from being able to remotely configure the various components of an ESB infrastructure. Once configured, it manages the containers and service lifecycles from starting and stopping through configuring and optimizing.

Monitoring: Information on the status of various parts of the infrastructure and the service containers and services is imperative to effective upkeep of the ESB platform. This includes the availability stats and statistics on various parts and their accesses. Number of active sessions on the platform, number of services accessed, average service times, uptime of various containers and components and such would form the monitoring data of an ESB platform.

Service Interactions: Configurable control of service interaction. This allow modification of data and process flow without re-coding or shutdown of running services. It provides flexibility to adapt SOA to changing business requirements.

Distributed, Dynamic Deployment: Supports distributed deployment of services and their configuration. It provides the ability to independently scale, reconfigure, and redeploy individual services without disrupting other operations.

Staged Deployment: In an enterprise-wide services platform, updating the services and various application artifacts without bringing the whole platform down becomes a major functional requirement. This will need good support for taking an application into production through a staging area. Utilities to simplify this process will be necessary to support deployment and migration of ESB services and processes from development to test and deployment environments. This solves the problem of service and process upgrade management for large-scale SOA deployments.

Centralized Auditing and Logging: Central logging and auditing of services, faults, process status, etc. This provides the ability to monitor and diagnose behaviors of complex distributed systems.

Security and Reliability: Managing Security involves configuring the secure communication channels, security certificates, and user repositories.

Management standards that ESB could use depend on the specific technologies used in the platform. In Java-based environments, the underlying management infrastructure is most likely based on JMX. Additionally, to fit into generic SOA monitoring and QoS solutions, support for simple Web Services management standards may be required.

Application Development Considerations

In this chapter, we have taken a comprehensive look at various aspects of an ESB infrastructure. For an organization adopting SOA, both the infrastructure and the application are equally important.

Integration Application Constituents

Solutions on ESB, or for that matter any SOA application, essentially involve developing the services and the business processes that use the services. In some cases, there could be custom integration applications that may access services from outside of business processes. The list of typical application artifacts includes:

- **Service Types**: A generic handling or program provided for a class of services. This is defined once, and multiple services can be configured. Each service of a given type is differentiated by some initialization properties or parameters for that service. These parameters are provided when defining the service.
- **Services**: Services are the business functions that are made available on the bus for consumption by the integration applications and business processes. These services may be self contained, or an instance of a reusable service type.
- **Service Interfaces (WSDL)**: Each service in ESB will have a well-defined interface. These may be standard interfaces such as WSDL or could be proprietary platform-specific interfaces. The service name, its operations, and the signature (inputs, outputs, faults), its protocol bindings and the access address/IP-port/URL.
- **Adapters**: Services typically may access other applications or systems in the enterprise. This needs adapters to be able to connect to these systems. Adapters provide a common API or interface using which many different systems and application platforms such as mainframes, SAP, BAAN, Java EE, etc. could be accessed.
- **Document Schemas (XSD)**: In an XML-centric environment, applications will also benefit by using XML to describe elements, provide information and transfer information. It is preferable that all business data is represented in XML. In this mode, the structure of these XML documents needs to be described. This is done using XSDs. Any project will have a set of well defined schemas that is used to define service interfaces and to describe the documents that are interchanged between various application components.
- **Transformations**: XSLT and Xquery; as we saw, ESB environments have powerful support for document transformations in the bus. Whether the documents are transformed in the bus or explicitly in a process, this will require the definition of the transformation rules. The standard way of doing this is using XSLT or XQuery.
- **Content-Based Routing**: These are like mini-process files. These define the routing rules based on the content of a document. The CBR files are typically configured on endpoints themselves.
- **Business Processes**: Next to services, the Business Processes are the application artifacts that hold direct business processing logic and rules. The business processes are the primary purpose of any SOA environment. Everything else essentially enables this ultimate goal.

- **Deployment Configuration Map/Plan**: Based on the application artifacts and the physical distribution of the organization, a deployment configuration needs to be defined. Unlike other artifacts, this is not a physical project "file". This is just a description of how all the project artifacts will be taken into the runtime production environment.

ESB—Application Design Approach

ESB being a technology instance for SOA, all the SOA design approaches discussed in Chapter 2 will come in handy here as well.

Some of the common SOA design patterns include:

- **Business Patterns**: The interaction between users, businesses, and data. Includes Information Aggregation, Process workflow, and Extended-Enterprise (Business-to-business) patterns.
- **Integration Patterns**: Tie multiple Business patterns together. This includes the Data Aggregation and Process aggregation (workflow) patterns.
- **Composite Patterns**: Commonly occurring combinations of Business Patterns. Examples include Buy-side-hub, Sell-side-hub and other E-Commerce patterns.
- **Application Patterns**: Conceptual layout describing how the application components and the data within the business pattern and integration pattern interact. Includes Asynchronous Services, Publish-Subscribe Services, Synchronous Services, Component Services, and Serial Service Orchestration.
- **Runtime Patterns**: Logical middleware structure that supports an Application pattern. Depicts the middleware nodes, their roles and the interfaces between the nodes.

ESB patterns, while building on the SOA patterns, have few specific considerations. The primary consideration is that ESB is based on a document-flow-based paradigm towards Services, Service Orientation, and Processes. Given this, design of any ESB based application will need to completely ingrain this into the design psyche—that there are documents and document flows. This warrants that any design exercise should start with a complete understanding and design of the data and data flows. Analyze the data in various systems and services and model the data via well described schemas.

In an ESB application, all Components are Services with a WSDL interface, either Custom Services or Generic Services—to ensure reuse with all services maintained in a Central Repository of Service. The application must use Routing and Transformation Logic, which also must be implemented as generic reusable

services. Like any other SOA application, ESB applications will also have a Services Orchestration Layer above the ESB layer to automate business processes—but in ESB the processes may run in the "fabric" as opposed to running in a central Orchestration Server or Process Engine. This will require few additional design considerations.

Data Transformations: Once there are well-defined data structures that capture the business data in various systems and departments in the organization, given that departmental systems existed in "isolation" first, this could very well mean that similar business data is represented differently in different departments or systems. This begets the need for common "translation" of data from one department's representation to another department's.

Considering that integration applications essentially start with existing enterprise applications and solutions that need to be integrated into enterprise wide business processes, getting the existing applications onto the ESB platform would be the first task. To enable this, there are design patterns such as the *Service On-Ramp* and *Service Off-Ramp* patterns.

The ESB process model, where the process runs in the fabric as opposed to running in an orchestration server or process engine, offers interesting design possibilities. In this mode, while there may be a well-defined definition of the process much like what is needed in a process server, the process does have a strong document-flow-based processing flavor. In such processes, the focus is a "single" document that is supposed to be flowing through the service steps and getting updated. From a solution design standpoint, this approach is different from a conventional "procedure"-like approach to business processes, where the process expects inputs, has process state and variables to manipulate through the process steps, and at the end there is a return/output from the process. In the ESB process case, the design is essentially around a document that needs to "flow" through various steps.

Security and Transaction Considerations: Ensure that there is very little programmatic handling of both. Use the standard approaches where possible to ensure that there is no vendor lock-in. Integration applications in an SOA environment should ideally separate the logical security model from the physical security. The SOA artifacts, processes, and services shouldn't be programmed for specific security and transaction handling. The security and transactions should be managed outside of the actual service program code, either declaratively as configuration options when setting up the service and process instances, or via the more standard approach of specifying the Security and Reliable Messaging assertions via WS-S and WS-RM.

Comparing ESB with Other Technologies

ESB provides the flexible connectivity and description that Web Services offers, the legacy systems access that EAI provides, and the manageability and performance that Application Platforms provide. Beyond this, ESB provides the enterprise-wide high-performing massively distributed platform, services mediation, and control in such a distributed environment.

ESB—Improves upon Web Services

ESB and Web Services are very complementary. ESB never contradicts Web Services; it only extends it. XML-based wire exchanges are designed for remote access across heterogeneous systems; WS-* standards are used for reliability and security.

XML-Based Data Easily Exchanged

As we saw earlier in this chapter, XML-based data interchange leverages the extensively well-defined XML document structure and semantics via the plethora of widely accepted standards in this space, and by the omnipresent XML-handling utilities and infrastructure. Both ESB and regular Web Services rely quite extensively on XML, both to represent the business data and also for the internal plumbing data interchanges. Requests and responses are typically represented in XML. The service interfaces are represented in XML. System configuration data is represented in XML. The actual middleware exchanges are represented in XML (SOAP). XML is everywhere.

Designed for Remote Access, Across Heterogeneous Platforms

Both ESB and Web Services are inherently designed for access from remote environments—the service and its consumer will be on different machines, and often accessible via a WAN with no assumptions whatsoever on the operating environments on either of the two ends. ESB takes this a step further by even abstracting the ends themselves. Unlike regular web services where the connection is directly established from the client (consumer) to the machine where the service is running, in ESB the connection is established to some intermediary (the ESB mediation layer). And the mediation then performs value-added infrastructure services before routing the request to the right service endpoint, which could again be anywhere in the network (accessible on the ESB domain).

Leverage WS-RM for Reliable Interoperability

WS-RM is a rapidly emerging standard that defines reliable communications. Given that most service accesses may happen via unreliable links in the WAN or still worse over the public Internet, specifying and actually providing the reliability semantics becomes imperative. An aggravating factor is cases where a series of documents/ messages are required to perform a business process, and there are sequencing expectations. Reliability could also mean time criticality—where if a message doesn't arrive by a certain time, there may be a certain "staleness" to the data and the business processes might warrant that that request not be processed. So, some timeout mechanisms need to be specified. In case of communication failure, retries could be attempted. This again is to be specified based on the business expectations from the system. It may be OK to retry just once, or a few times or several times. This is a business requirements-based decision.

WS-RM provides for specifying all of these as "expectations" from the service. The requesting end (consumer) and the service end, it they support WS-RM, will ensure that these reliability requirements are met, with sufficient handling when they fail—like aborting the request and notifying the requester of the failure. Both ESB and WS platforms fully support WS-RM.

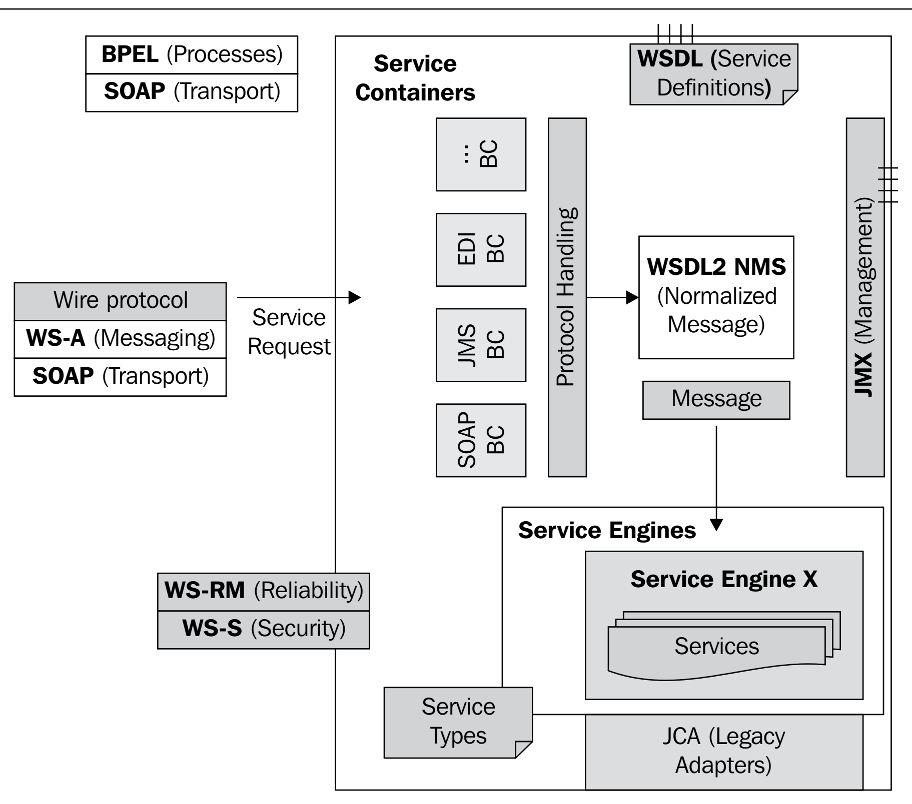
Security using WS-S

While ESB platforms may have own proprietary security models, they do fully support the WS-Security standards. This ensures that Web Services and ESB both have a common security model.

Where does ESB better Web Services?

Interestingly, the application's view of a simple Web Services-based integration application and an elaborate ES-based application environment need not be too different. As the business processes may be defined per the BPEL standard, and services described using WSDL, standards based adapters used to connect to legacy systems, and so on. The difference essentially is in the configuration, deployment, manageability and, the internal scalability and performance.

Web Services, being more about the wire protocols, do not have any definition of a services container as such. There is no notion of a set of service containers forming a "bus". This is where ESB steps in and provides this abstraction—without in any way modifying the base application models using BPEL and WSDL. ESB provides the constructs to configure and manage the environment and more importantly, provides a simple mechanism to deploy the services and processes and monitor them. In the Java world, there are standards that are leveraged to build the service containers, as shown in the following figure.



Given that ESB is a platform that has control of all parts of an interaction between the service consumer or process and the service itself, mediation services are possible. In a simple Web Services model, this is not possible, as there is no infrastructure component sitting between the service consumer and the service. Therefore, value-added mediation and control capabilities such as late binding of service physical address, transformation of data on the wire, or content-based routing are not possible in Web Services—while they are inherently supported in ESB platforms.

In Web Services, as there is no intermediary platform as such, there will not be any internal scalability or performance capabilities. ESB being based on a high performance communication backbone, with elaborate service containers and mediation framework can provide a high level of scalability and performance.

Higher Level of Abstraction in ESB

One aspect of services is to abstract out as much of the actual implementation as possible to ensure that the service can be easily be reconfigured, redeployed, and internal implementations changed without affecting any of the "consumers" of the service. This is possible by having a well-defined contract via its WSDL interface and having a simple endpoint URL that is NOT the actual service. The URL should point to a plumbing infrastructure component that does the actual routing. Conventional web services do not support the model; the endpoint is generally directly accesses the service implementation. That makes a hard binding to the deployed and configured service instance. ESB systems typically abstract the actual endpoint behind an elaborate back-end service bus.

This makes the service consumers and service endpoints very loosely coupled—providing a much greater flexibility. This is not that easily possible in simple Web Services, where the binding port directly refers to the service itself, and not to any abstract logical endpoint or address.

EAI: Cannot Span Integration Brokers

EAI solutions are based on Integration brokers. The primary purpose of EAI platforms is to provide simple connectivity to legacy systems. These are typically deployed inside a single company—inside the firewall. Services and processes cannot seamlessly span integration brokers. While some EAI vendors now do talk about federation of Integration Brokers, they would still suffer from the primary focus being on programmatic connectivity to legacy systems than to provide access to services.

Application Servers: Hub-and-Spoke Model Limits Scalability

Application platforms such as Java EE and .NET are primarily targeted to host applications and data- and UI-intensive business applications. Such applications are quite likely to be providing the services that are integrated in an SOA environment. The application platforms are exceptionally good for hosting business logic in a component model and serving web pages. Scalability is well addressed within the context of one server instance via clusters. But these form a very tightly coupled cluster that will most likely depend on intense back-end housekeeping communications either among the cluster nodes or with a data repository.

In a widely distributed enterprise, the application platform instances cannot form a functional single cluster. This would mean that there will be multiple application server instances in the enterprise, and for this to provide a single SOA environment, the instances will need to interact with each other. Application servers do not do a good job at providing a good communication infrastructure for such interactions.

Further, there is no single services namespace that Application Servers offer. Each server instance may have its own services registry. Though technically there could be shared registries, again owing to protocols that were more designed for server LANs, these would not work very well over a WAN.

The Application Platforms have an elaborate deployment and configuration process to get an application up and running. This also means that updating the Services would require disruptive re-configuration and re-deployment.

ESB—Helps Avoid Vendor Lock-Ins

In an ESB environment, there is the infrastructure provided by the vendor and then there is the application. Typical ESB application artifacts will include: services (most likely wrappers that access some legacy system), XML schemas and transformations, and processes. A few off-the-shelf service types may be available like the transform service or a database service as in the case of Sonic ESB. While there are standards to describe a service and to invoke a service, there are not standards yet on implementing services. In the Java world, standards such as JBI are emerging that may help; but this has to gain some traction. So, vendor lock-ins can occur essentially in the implementation of services and processes, wherein the same services and processes will not work in another SOA environment.

This lock-in for the services can be minimized with some good practices. The services' implementation programming code may be in popular languages such as Java, C#, or VB.NET. If designed well, a thin layer that implements any vendor-specific interfaces will alone have the vendor lock-in. When porting from one vendor's platform to another, this thin layer alone will need to be re-written. The rest of the application artifacts can be reused as is.

It is often said that by writing Web Services they become reusable and can be made to run on any platform. One has to understand that just by having a WSDL that describes a service, a Web Service does not just happen. In addition to the plumbing that processes the SOAP requests, there will be the actual service implementation that is sitting behind the plumbing. And this will be platform dependent.

The other major application artifact, the Business Process, has much stronger standards traction. Either the vendor already supports these standards or can at the least be easily exportable to standard BPEL. So vendor lock-in can be kept very minimal here. Again, in the absence of absolute support for standards, good practices will need to be established to try to avoid using vendor-specific capabilities provided in the proprietary process models.

The XML schemas and transformations used in an application are all absolutely standard. Though vendors may support some extensions, say to the standard XSLT, these can easily be avoided to ensure that there is no lock-in. These are probably the easiest to migrate from one vendor's platform to another vendor's.

Of course, in the absence of key standards in the ESB platforms, today there is no out-of-the box portability from one ESB vendor to another. Come to think of it, there are no common definitions of what an ESB is. Even so, with ESB, a good amount of it is possible today. More would be possible in the next few releases as WSDL2 and JBI gain more traction. Even today, with some simple design considerations, the ESB application can be kept as vendor neutral as possible, minimizing the porting effort needed when moving onto a different vendor's.

Standards being the key enabler to minimize vendor lock-ins, it is interesting to note the power of standards here. Just as Open Source does offer alternatives to commercial vendors, Standards enable each commercial vendor being an alternative to the others. To the users and enterprises, both offer the same benefit of rapid "commoditization" of the space. The standards adoption is driven largely by the uptake from the consumers (developers and infrastructure decision makers). So, if there is a push on standards from the developers and IT managers, all ESB vendors will rapidly adopt the same. At least in the case of ESB, there are two key standards that have decent traction—JBI (with WSDL2) and BPEL. This would make it possible for a complete ESB application to move from one ESB platform to another—just as Java EE did in the Application Server space.

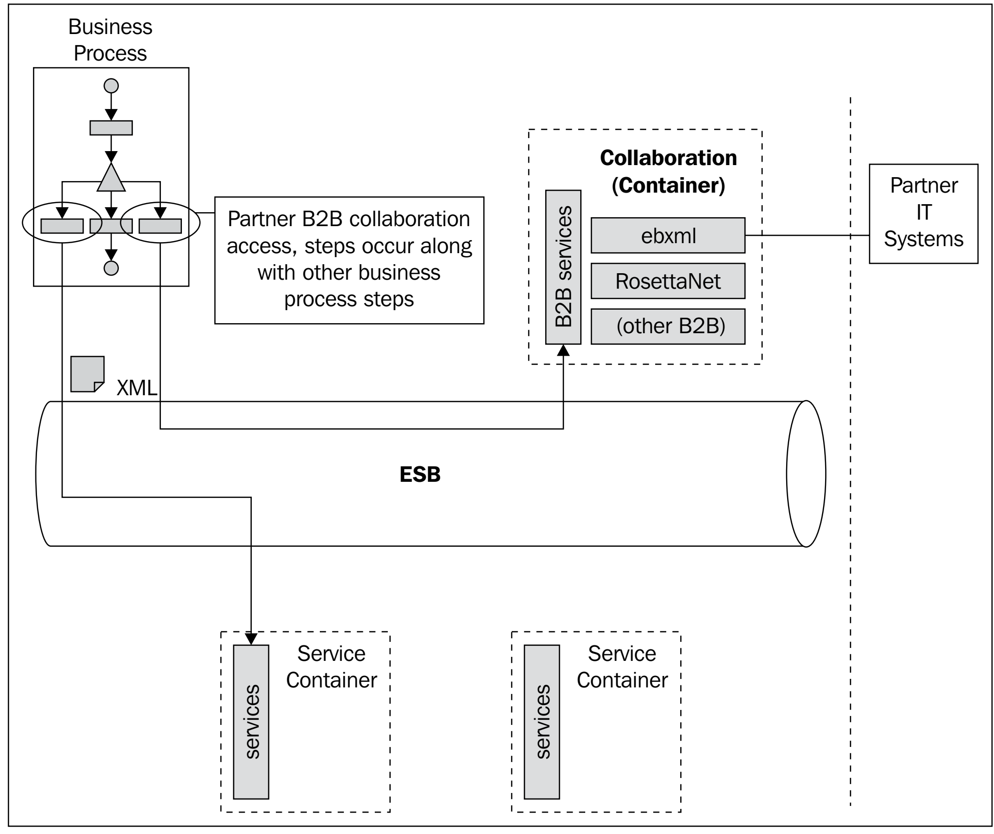
Messaging Platforms: ESB Extends the Message Model

ESB platforms typically rely on a messaging-based communication backbone. Messaging platforms are used for delivering messages to destinations and they do not prescribe nor assume the nature of processing involved on the messages. ESB systems, however, have the service execution as their primary objective. That there is a message underneath is incidental and internal to the platform. The ESB infrastructure interacts with the messaging layer and will direct the request to the service by invoking the service operation requested. The messaging layer here is just a means to the end objective—of providing a services infrastructure to host and invoke services.

Extending ESB to Partners

With the highly simplified connectivity that the Internet ushered in, enterprises are rapidly engaging with partners and vendors electronically. In this landscape, once an enterprise adopts ESB for integrating the various systems and solutions in the enterprise, it would be a very logical next step to get even the partner interactions into the same enterprise integration fold. In determining operational needs for partner interactions, there are three levels and areas to include. The first level is the overall operational approach to solving large enterprise-level interactions, then enterprise to small business interactions, and small business to small business interactions.

The typical partner interactions are document-centric rather than remote-procedure-centric. ESB would fit in very well in such partner interactions by its document/message-based interaction models, as shown in the following figure. This can easily be extended to support the conversational interactions that are common in B2B space. Here, any partner interaction is defined by business processes and agreements between the partners. These will typically involve a series of exchanges, spread over an extended interval, for any given trade transaction.



Just looking at a business-to-business interaction, there may be multiple technologies that may be available ranging from the EDI and AS2 type interactions, to Web Services, to a more modern Collaboration Framework such as RosettaNet and ebXML. While Web Services provide a simple interactive point-to-point solution that can address the case of atomic exchanges between two partners, when there are multiple partners and complex business interactions and collaborations to be modeled, simple Web Services will be found lacking. In such cases, more evolved electronic business platforms such as ebXML will be needed. These provide secure, reliable business-to-business exchanges through open eBusiness architecture. They provide for well-defined interchange framework, that supports clearly and unambiguously describing the interactions, including the document structures for the exchanges, the protocols for the interactions and the sequence of interactions in any business conversation.

ebXML as a representative B2B technology includes:

- Business Processes—defined as models in UMM, scripted in XML
- Business Messages—content agnostic - exchanged using ebMS
- Collaboration Protocol Profile and Agreement—specifies parameters for businesses to interface with each other—expressed in XML
- Messaging Layer—moves the actual XML data between trading partners—ebMS
- Core components—library of pre-defined business vocabulary artifacts
- Collaboration Registry—provides a "container" for process models, vocabularies, assembly templates, partner profiles and discovery

One aspect in these interactions is that they are focused on documents flowing between partners. This fits in very nicely with ESB. In ESB messages and documents flow through the bus and get processed. This can easily be extended to include documents being sent to a partner as a part of a bigger business process. Now if the ESB process environment can be made aware of the B2B specifics, such as the specific interaction types in an ebXML or RosettaNet exchange, then the steps in a business process can be more integrated into the ESB. Like for example, as part of an order processing step, can actually include B2B calls to a supplier to supply parts, and the process can actually wait for the confirmation of shipment before proceeding further, say with a build work order handling step.

Summary

In this chapter, we saw how ESB provides a concrete infrastructure for SOA, extending the simple services model to include a robust service bus with extensive mediation functionality.

ESB extends the omnipresent P2P model of Web Services. We saw how ESB provides the middleware functions for SOA via its communication backbone, mediation services, and its services infrastructure. ESB leverages Web Services standards where possible—though standards are yet to come up in the ESB service runtimes space. Until then there will be limitations in terms of portability of applications from one vendor's ESB platform to another.

In summary, ESB is pre-built SOA infrastructure, with enterprise-grade capabilities, generalized support for variety of integration tasks, implementing and reinforcing architectural best practices, suitable for individual projects or massively distributed integration projects. ESB allows enterprises to integrate applications across the extended enterprise using a standards-based, service-oriented architecture (SOA).

Index

A

alarm events 233
API
choosing 156
application development
about 346
application design approach 348, 349
ESB, extending to partners 356, 357
ESB versus other technologies 350-354
integration application 346
vendor lock-ins, avoiding 354, 355
application integration
about 26
graphical representation 27
need for 5
application servers
about 42
aspects 42
asynchronous processes 230

B

B2B versus EAI
about 188
interface design 189
service registry 189
best practices, integration process
about 48
incremental development 49
iterative development 48
prototyping 50
reuse 50
bottom-up approach
about 17
disadvantages 20
BPEL
about 223, 226
abstract processes 227, 228
executable processes 227, 228
features 226, 227
languages for choreography 228
modelling notations 229
service composition 225
uses 226
BPEL activities
overview 235-238
BPEL processes, writing
about 229
BPEL activities overview 235-237
handlers 232
partner links 231
partner link types 231
process interface 230
scopes 235
variables 232
BPEL process example, developing
about 238
ApplyDiscounting operation 245
ApplyPricing operation 244
asynchronous BPEL process 238
Billing Service 246, 247
CalculateTotal operation 247
CollectData operation 241, 242
CreateSendBill operation 247, 248
event handler, adding 264
fault handler, adding 263
graphical representation 238
OnFault operation 243
partner links, defining 254
partner link types, adding to WSDL 248, 249

process, declaring 253
process, deploying 265-268
process, running 265-268
ProcessData operation 242
process definition, writing 256
process logic, writing 252
Rating Service 243, 244
Resource Data Service 239
services used 239
variables, declaring 255
WSDL interface, defining 250, 251
BPEL process logic
ApplyDiscounting scope 259
ApplyPricing scope 259
CalculateTotal scope 261
callback, returning to client 263
CollectResourceData scope 257
CreateSendBill scope 261
partner links, defining 254, 255
process, declaring 253
process definition, writing 256
ProcessResourceData scope 258
variables, declaring 255
writing 252
bus 75
business integration
need for 5
business processes composing, services used
advantages 91
B2B collaboration 92
multi party business process 91, 92
simple business processes 90, 91
simple integration applications 89
business process integration
about 28
graphical representation 28
business services
complexity 219
development lifecycle 220-222
identifying 219, 220
business to business integration 29
bus services
about 307
ESB processes 315
infrastruction mediation 310
intelligent content-based routing 312, 313
mediation, need for 308, 309
physical address indirection 309
transformation services 313

C

central managed integration project
about 13
integration architecture, defining 14, 15
integration documentation development 15
integration documentation maintenance 15
integration infrastructure 15
integration technologies 15
chameleon design 143
compensation handlers 234
about 234
defining 234
current system
about 8
common applications 8
shortcomings 6, 7
custom integration application
uses 64, 65

D

data-level integration
about 25
graphical representation 25
database access technologies 37
Document Object Model 153
dynamically generated documents 158

E

element attribute, variables 232
elements
<variables> 232
Enterprise Service Bus
about 269
application development 346
architecture 275
bus services 307
defining 276
management 330
reliability 330

scalability 330
security 319
service containers 296
Enterprise Service Buses 45
ESB. *See* **Enterprise Service Bus**
ESB architecture
about 275
application attributes 279, 280
document centric service 283
documents 287, 289
enterprise document flows, modelling 280, 282
ESB, defining 276
ESB, defining cocepts 276
infrastructure components 289-293
key constituents 277
middleware for middleware 278, 279
procedure centric service 283, 284
service locations with endpoints, abstracting 286
services 286-289
web services standards 293-296
XML in ESB 285
ESB infrastructure components
about 289
communication and interoperability 293
communication layer 292
mediation and control 292
registry 291
routing 292
security 292
service containers 290
services 290
service types 290
transformation 292
ESB processes
about 316
document itineraries 316, 317
itineraries 318
versus orchestrated processes 319
event handlers
about 233
alarm events 233
message events 233
event managing 233
expose namespaces
advantages 138
versus localize namespace 137
extensibility mechanism, WSDL 232
external entities
referencing 158

F

fault handlers
about 233
fault causes 233

H

handlers, BPEL process
compensation handlers 234
event handlers 233
fault handlers 233
types 232
heteregeneous namespace design 142
homogeneous namespace design 142

I

infrastructure mediation
about 310
message enrichment 311
protocol handling 310
quality of service 311
requesting routing 311
security 311
service registry 311
version resolution 311
integration
challenges 6
custom integration application 64, 65
difficulties in 6
effective information systems 9
existing applications, replacing 9, 10
information access with low latency 12
need for 5
requirements 11
single data input 11
strategies 11
with XML 125
integration architecture
about 16
advantages 23, 24

approaches 16
bottom-up approach 17-21
steps 16
top-down approach 21, 22
integration infrastructure
about 30
brokering 32
business intelligence 33
communication 31
horizontal layer services 30
lifecycle 34
management 35
naming 35
routing 32
rules 36
scalability 35
security 34
transactions 34
transformation 33
vertical layer services 31
integration process
about 46
activities 50, 51
best practices 48
ingration patterns 52
milestones, defining 46, 47
phases 50, 51
steps, choosing 46, 47
integration technologies
about 36
application servers 42, 43
database access technologies 37
Enterprise Service Buses 45, 46
message oriented middleware 37
Object Request Brokers 41, 42
remote procedure calls 39, 40
Transaction Processing monitors 40
web services 43, 45
integration types
about 24
application integration 26, 27
business process integration 28
business to business integration 29, 30
data-level integration 25
presentation integration 29
interoperability challenges in web services 195

J

Java Business Integration. *See* **JBI**
Java EE and .NET integration
.NET web service, deploying 209, 210
.NET web service, developing 208
Java web services, deploying 206
Java web services, developing 205
test client, developing 211, 212
WSDL for Java web service 206, 207
JAXP APIs
features 155
JBI
about 304
binding components 306
management 306
normalized message service 306
service definition 306
service engines 305
services 306

L

localize namespaces
advantages 138
requirements 137, 138
versus expose namespace 137

M

management of ESB
infrastructure 344-346
managing events 233
mediation
infrastructure mediation 310
need for 308
physical adress indirection 309
message events 233
message oriented middleware 37

N

namespaces
default namespaces 133

O

Object Request Brokers 41

P

parsing
for incoming documents 156
parser, choosing 157
push parsing 153
partner links
about 231
types 231
POA
concepts 119
focusing on processes first 118
principles 119
services first 117
services orchestration 117, 118
transition to 116
POA architecture
about 214
process automation influencing factors 214, 215
POA principles
about 119
analysts to programmers 120
infrastructure 123
processes first 119, 120
process standards 122
software development roles, changing 121, 122
top-down design, using processes 120
portability. *See* **services designing, for portability**
presentation integration 29
processes for portability. *See* **services designing, for portability**
process interface
about 230
asynchronous processes 230
synchronous processes 230
pull parsing
versus push parsing 153

R

reliability
about 330
achieving 334, 336
concepts 330
configurable interaction model 337
location transparency 336, 337
messaging basics 332, 333
messaging platform, leveraging 336
multiple interaction model 337
WS-standards 333
RPC 39
runtime patterns
for broker 188
Russian Doll design approach 139

S

scalability
about 338
ESB performance 338
load balancing 339, 340, 341
load scaling 339, 340, 341
schema 137
scopes, BPEL process
about 235
serializable scopes 235
security, ESB
about 319
application platform security 320
distributed transactions 327
in integration architecture 319
process driven local transction semantics 330
transactions, realising 329
transactions built on messaging layer 329
transaction semantics 324-326
transaction strategies 327
WS-security 321, 323
WS tranastion standards 328
security, SOA
.NET security 96
about 93
in Java infrastructure 95
loosely coupled services 96
middleware systems security 94
security challenges 93
transactions 98, 99
web services security 96, 97, 98
web services security, specifications 97
serializable scopes 235

service containers
about 296
communication infrastructure 306, 307
JBI 304
services, external views 301-303
service types definitions 296
standards, need for 300
structure 298, 299
services. *See also* **web ser XE**
business processes, composing 89
securing guidelines 152
services compostion
about 217
business services, identifying 219, 220
business services complexity 219
choreography 218
development lifecycle 220-222
orchestration 218
services designing, for portability
about 110
adoption considerations 111, 112
business data, as XML 113, 114
infrastructure independence, designing for 114
new applications 114
processes in BPEL 114
SOA/POA 114
think services 112
vendor independence, designing for 114
SOA
about 6
concepts 66
custom integration application 64, 65
executable business processes 222, 223
executable business processes, example 224, 225
inverted view 65
need for 5
principles 66
reusable services 65
security 93
simple integration processes 65
transition to POA 116
web services 76
XML 76
SOA architecture
about 71
asynchronous messages 74
bus 75
communication infrastructure 75
graphical representation 72
layers 216
messaging abstractions 73
process engines 73
Quality of Service 75
service implementation 73
service invocation 73
service registries 74
services abstractions 73
synchronous messages 74
SOA infrastructure
communication 100
communication bus 104
communication bus, advantages 104
communication infrastructure 103, 104
component services types 101
distributed SOA environment, managing 106
execution engines 101, 102
MOM 105
options 107
reliability 106
scalability 106
service containers 101, 103
service execution 100
XML backbone 105
SOA infrastructure options
about 107
application platforms 108
ESB 110
integration platforms 109
simple messaging based custom infra 109
technology standards overview 107
web services 108
SOA principles
component-based services 68-70
paradigm shift 66, 67
service orientation 67
services, consuming 71
SSL
versus XML Encryption 150
StAX
about 154
features 155

JAXP APIs 154
synchronous processes 230

T

top-down approach 21
Transaction Processing Monitors 40
transactions. S*ee* security, SOA; S*ee* security, ESB
transformation services
about 313
need for transformation 313, 314
transformation, XSLT used 314
XML manipulating, XQuery used 315
type attribute, variables 232

U

UDDI
about 83
uses 83
Universal Description Discovery and Integration. S*ee* UDDI

V

validation cost
reducing 157
variables
element attribute 232
messageType attribute 232
type attribute 232
variables, BPEL process
about 232
attributes 232

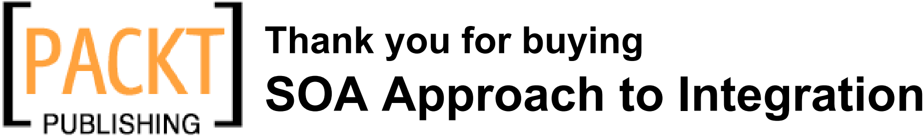
W

web services. S*ee also* services
about 43, 79
application platforms 88
B2B versus EAI 188
containers for hosting 84
interoperability challenges 195
JAVA EE 88
security 96
service description, WSDL used 83
specifications 44
system-to-system interaction 81, 82
UDDI 83
web services standards
about 293
description 294
discovery 294
reliability 295
WSDL interoperable definitions
validating 194, 195
writing 190-194
WS tranastion standards
about 328
Architecture 329
WS-AtomicTransaction 328
WS-BusinessActivity 329
WS-Coordination 328

X

XML documents, securing
about 146
service securing guidelines 152
XML security threats 146
XML Encryption
about 147
best practices 150
single element, encrypting 149, 150
versus SSL 150
XML file, encrypting 148
XML Signatures 151
XML for integration
about 125
designing tips 131
domain specific XML schemas 125
domain specific XML schemas, recommendations 126
incoming XML documents, fragmenting 131
processing models, choosing 129, 130
schemas, mapping 129
XML documents, receiving 126
XML documents, sending 127
XML documents, validating 127, 128
XML in middleware
about 76
midleware mechanics for services 76, 77

services invoking, XML-based mechanism used 77, 79
services over the web via SOAP 79
XML schemas, designing tips
about 132
chameleon design 143
default namespace 133-137
designing cases 136
expose namespace 137
global declaration 139
heterogeneous namespace design 142
homogeneous namespace design 142
local declaration 139
multiple schema namespace problem 141
type declaration 140
XML Signatures 151
XML streaming
push parsing 153
XSL for transformation
about 143
import instruction 143-146
include instruction 143-146



Packt Open Source Project Royalties

When we sell a book written on an Open Source project, we pay a royalty directly to that project. Therefore by purchasing SOA Approach to Integration, Packt will have given some of the money received to the Apache Synapse Project.

In the long term, we see ourselves and you—customers and readers of our books—as part of the Open Source ecosystem, providing sustainable revenue for the projects we publish on. Our aim at Packt is to establish publishing royalties as an essential part of the service and support a business model that sustains Open Source.

If you're working with an Open Source project that you would like us to publish on, and subsequently pay royalties to, please get in touch with us.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `authors@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

About Packt Publishing

Packt, pronounced 'packed', published its first book "Mastering phpMyAdmin for Effective MySQL Management" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.PacktPub.com`.