



From Technologies to Solutions

Service Oriented Java Business Integration

Enterprise Service Bus integration solutions for Java developers

Binildas C. A.

[PACKT]
PUBLISHING

Service Oriented Java Business Integration

Enterprise Service Bus integration solutions for
Java developers

Binildas C. A.



BIRMINGHAM - MUMBAI

Service Oriented Java Business Integration

Copyright © 2008 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 author, 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: March 2008

Production Reference: 1040308

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

ISBN 978-1-847194-40-4

www.packtpub.com

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

Credits

Author

Binildas C. A.

Project Manager

Abhijeet Deobhakta

Reviewers

Rajesh R V

Rajesh Warriar

Project Coordinator

Aboli Mendhe

Acquisition Editor

Bansari Barot

Indexers

Hemangini Bari

Monica Ajmera

Development Editor

Ved Prakash Jha

Proofreader

Angie Butcher

Technical Editor

Della Pradeep

Production Coordinator

Shantanu Zagade

Aparna Bhagat

Editorial Team Leader

Mithil Kulkarni

Cover work

Shantanu Zagade

About the Author

Binildas C. A. provides Technical Architecture consultancy for IT solutions. He has over 13 years of IT experience, mostly in Microsoft and Sun technologies. Distributed Computing and Service Oriented Integration are his mainstream skills, with extensive hands-on experience in Java and C#.NET programming. Binil holds a BTech. degree in Mechanical Engineering from College of Engineering, Trivandrum (www.cet.ac.in) and an MBA in Systems Management from Institute of Management, Kerala (www.imk.ac.in). A well-known and a highly soughtafter thought leader, Binil has designed and built many highly scalable middle-tier and integration solutions for several top-notch clients including Fortune 500 companies. He has been previously employed by multiple IT consulting firms including IBS Software Services (www.ibsplc.com) and Tata Consultancy Services (www.tcs.com) and currently works for Infosys Technologies (www.infosys.com) as a Principal Architect where he heads the J2EE Architects group servicing Communications Service Provider clients.

Binil is a Sun Certified Programmer (SCJP), Developer (SCJD), Business Component Developer (SCBCD) and Enterprise Architect (SCEA), Microsoft Certified Professional (MCP) and Open Group (TOGAF8) Certified Enterprise Architecture Practitioner. He is also a Licensed Zapthink Architect (LZA) in SOA. Besides Technical Architecture Binil also practices Enterprise Architecture.

When not in software, Binil spends time with wife Sowmya & daughter Ann in 'God's Own Country', Kerala (www.en.wikipedia.org/wiki/Kerala). Binil does long distance running and is a national medalist in Power Lifting. You may contact Binil at biniljava@yahoo.co.in or binil_christudas@infosys.com.

Acknowledgement

First and Foremost, I would thank God who has always showered his choicest blessings on me. I thank Him for all that he has done for me.

I would like to thank PACKT and everyone I worked with—Priyanka Baruah, Bansari Barot, Patricia Weir, Aboli Mendhe, Bhushan Pangaonkar, Ved Prakash Jha, Della Pradeep and others. They worked very hard with the aggressive schedule I proposed on this book, and I truly do appreciate their contributions.

Next, I'd like to thank the Technical Reviewers of our book, Rajesh R. V. and Rajesh R. Warriar. Without them, you wouldn't see the text as it appears here. Thank you for your thorough review of the scripts and testing of the code. Your reviews were very objective in pointing out issues and helping me to come up with the even better chapters.

This book would have never been possible if were it not for the excellent colleagues at IBS (other than the reviewers) whom I have worked with and learned from. The most important of them are Prasanth G Nair, Shyam Sankar S, Sherry CK, Jayan P and Amritha Mathew M. Thanks are due to VK Mathews for providing all of us the opportunity. I would also like to thank Rajasekhar C. and Ajmal Khan who provided me the right challenges at the right time.

Special thanks are due to Dr. Srinivas Padmanabhuni, Principal Researcher, Web Services/SOA Centre of Excellence, Software Engineering and Technology Labs, Infosys for his guidance and help.

I would like to thank my wife and best friend Sowmya for being a constant source of inspiration in my life. Also my sweet little daughter Ann, I remember all those moments when you were so desperate to play with me and to go out for 'dinner' with pappa and amma, but I could not look beyond my laptop screen. Both of you are my angels, thanks for everything, especially for being in my life.

A massive thanks must go to my Mom and Dad – Azhakamma and Christudas – who have supported their wayward son through good and lean times, and have given everything and asked for nothing. I thank the rest of my family for their support and friendship – my sister Binitha and family, my wife's mother, Pamala, and father Hubert for their unconditional encouragement and love in helping me find the energy to complete this book.

I would also like to thank Ramavarma R for supporting me through his consultancy, MacJavaB.

There were many others who played their part too. Most important of them are the creators of the frameworks that have inspired me to write this book. Thank you for the JBI specification and special thanks to the ServiceMix developer and user community.

Last, it's necessary to thank you, the reader, for choosing to buy this book. I understand that you have a specific intention in choosing to read this book and I hope I take only the minimum required time from your busy schedules to serve your requirements.

About the Reviewers

Rajesh R V received his Computer Engineering degree from the University of Cochin, India. He joined the JEE community during the early days of EJB (1.0) and fully dedicated his time in core technical activities in and around Java, JEE. During the course as a solution architect, he has worked on many large scale mission critical projects, including the New Generation Passenger Reservation System (400+ Man Years) and Next Generation Cargo Reservation System (300+ Man Years), in the Airline domain. Rajesh is also Sun Certified Java Enterprise Architect and BEA Certified Weblogic Administrator.

Rajesh is currently working with Emirates Airlines IT Division based in Dubai and his work is mainly in Consulting, Application Framework development, Technology Evaluations and SOA related topics.

All my thanks goes to my wife Saritha for supporting me and loving me even though I booked a lot of personal time to review this book.

Rajesh Warrior, currently working as one of the lead system architects in Emirates Group IT, has around 10 years experience in the industry working with companies like Sun Microsystems. He has been responsible for architecting and designing many mission critical enterprise applications using cutting edge technologies. He is currently working as an architect and mentor for the new generation cargo system for the emirates airlines, developed completely using JEE.

Table of Contents

Preface	1
Chapter 1: Why Enterprise Service Bus	7
Boundary-Less Organization	8
Multiple Systems	8
No Canonical Data Format	8
Autonomous, but Federated	9
Intranet versus Internet	10
Trading Partners	10
Integration	11
Enterprise Application Integration	12
Integration Architectures	12
Point-to-Point Solution	13
Hub-and-Spoke Solution	13
Enterprise Message Bus Integration	15
Enterprise Service Bus Integration	16
Enterprise Service Bus versus Message Bus	17
Similarities and Differences	17
Maturity and Industry Adoption	19
Making the Business Case for ESB	20
How many Channels	20
Volatile Interfaces	22
New Systems Introducing Data Redundancy	22
Service Reuse	23
System Management and Monitoring	23
Enterprise Service Bus	23
Service in ESB	23
Abstraction beyond Interface	24
Service Aggregation	25
Service Enablement	26
Service Consolidation	26

Service Sharing	27
Linked Services	28
Virtualization of Services	29
Services Fabric	30
Summary	30
Chapter 2: Java Business Integration	31
SOA—the Motto	31
Why We Need SOA	32
What is SOA?	32
SOA and Web Services	33
Service Oriented Integration (SOI)	36
JB1 in J2EE—How they Relate	36
Servlets, Portlets, EJB, JCA, and so on	37
JB1 and JCA—Competing or Complementing	37
JB1—a New Standard	38
JB1 in Detail	39
JSR 208	39
JB1 Nomenclature	40
Provider—Consumer Contract	42
Detached Message Exchange	44
Provider—Consumer Role	45
Message Exchange	47
Service Invocation	47
Message Exchange Patterns (MEP)	47
In-Only MEP	48
Robust In-Only MEP	48
In-Out MEP	50
In-Optional-Out MEP	52
ESB—Will it Solve all Our Pain Points	55
Summary	56
Chapter 3: JB1 Container—ServiceMix	57
ServiceMix—Under the Hood	58
Salient Features	58
ServiceMix Architecture	58
Architecture Diagram	58
Normalized Message Router Flows	59
Other ESBs	63
Mule	63
Celtix	63
Iona Artix	64

PEtALS	64
ChainBuilder	64
Installing ServiceMix	65
Hardware Requirements	65
OS Requirements	65
Run-time Environment	65
Installing ServiceMix in Windows	66
Installing ServiceMix in Unix	67
Configuring ServiceMix	67
Starting ServiceMix	67
Stopping ServiceMix	67
Resolving classpath Issues	67
ServiceMix Components—a Synopsis	68
Standard JBI Components	68
Lightweight JBI Components	69
Your First JBI Sample—Binding an External HTTP Service	70
Servlet-based HTTP Service	71
Configure the HTTP Service in ServiceMix	74
Run ServiceMix Basic JBI Container	76
Run a Client against ServiceMix	78
What Just Happened in ServiceMix	78
Spring XML Configuration for ServiceMix	79
Summary	81
Chapter 4: Binding—The Conventional Way	83
Binding—What it Means	83
Binding	84
Endpoints	84
Apache SOAP Binding	84
A Word about Apache SOAP	85
Apache SOAP Format and Transports	85
RPC and Message Oriented	86
Binding Services	86
Sample Bind a Stateless EJB Service to Apache SOAP	88
Sample Scenario	88
Code Listing	89
Running the Sample	91
Deploying the EJB	91
Bind EJB to SOAP	92
Run the Client	93
What Just Happened	94
How the Sample Relates to ServiceMix	96

Summary	97
Chapter 5: Some XFire Binding Tools	99
Binding in XFire	100
XFire Transports	100
JSR181 and XFire	101
Web Service Using XFireConfigurableServlet	101
Sample Scenario	101
Code Listing	102
Running the Sample	104
Web Service using XFire Spring XFireExporter	106
Sample Scenario	106
Code Listing	106
Running the Sample	109
Web Service Using XFire Spring Jsr181 Handler	109
Sample Scenario	109
Code Listing	110
Running the Sample	113
XFire Export and Bind EJB	113
Sample Scenario	114
Code Listing	115
Running the Sample	119
XFire for Lightweight Integration	120
Summary	121
Chapter 6: JBI Packaging and Deployment	123
Packaging in ServiceMix	124
Installation Packaging	124
Service Assembly Packaging	125
Service Unit Packaging	126
Deployment in ServiceMix	126
Standard and JBI compliant	126
Lightweight	127
Packaging and Deployment Sample	127
Phase One—Component Development	128
Phase Two—Component Packaging	129
Running the Packaging and Deployment Sample	132
Summary	134
Chapter 7: Developing JBI Components	135
Need for Custom JBI Components	135
ServiceMix Component Helper Classes	136
MessageExchangeListener	137

TransformComponentSupport	137
Create, Deploy, and Run JBI Component	140
Code HttpInterceptor Component	140
Configure HttpInterceptor Component	141
Package HttpInterceptor Component	142
Deploy HttpInterceptor Component	143
Build and Run the Sample	144
Summary	145
Chapter 8: Binding EJB in a JBI Container	147
Component versus Services	147
Coexisting EJB Components with Services	148
Indiscrimination at Consumer Perspective	148
Binding EJB Sample	149
Step One—Define and Deploy the EJB Service	149
Step Two—Bind EJB to ServiceMix	150
Step Three—Deploy and Invoke EJB Binding in ServiceMix	155
Step Four—Access WSDL and Generate Axis-based	
Stubs to Access EJB Outside Firewall	156
Reconciling EJB Resources	160
Summary	160
Chapter 9: POJO Binding Using JSR181	161
POJO	161
What are POJOs	161
Comparing POJO with other Components	162
ServiceMix servicemix-jsr181	162
JSR 181	162
servicemix-jsr181	162
servicemix-jsr181 Deployment	163
servicemix-jsr181 Endpoint	164
POJO Binding Sample	164
Sample Use Case	164
POJO Code Listing	166
XBean-based POJO Binding	166
Deployment Configuration	167
Deploying and Running the Sample	169
Access WSDL and Generate Axis-based Stubs to	
Access POJO Remotely	169
Accessing JBI Bus Sample	173
Sample Use Case for Accessing JBI Bus	175
Sample Code Listing	177

Build, Deploy, and Run the Sample	179
Summary	179
Chapter 10: Bind Web Services in ESB—Web Services Gateway	181
Web Services	181
Binding Web Services	182
Why Another Indirection?	182
HTTP	182
ServiceMix's servicemix-http	183
servicemix-http in Detail	183
Consumer and Provider Roles	184
servicemix-http XBean Configuration	185
servicemix-http Lightweight Configuration	188
Web Service Binding Sample	189
Sample Use Case	189
Deploy the Web Service	190
XBean-based servicemix-http Binding	193
Deploying and Running the Sample	193
Access WSDL and Generate Axis Stubs to Access the Web Service Remotely	194
Summary	198
Chapter 11: Access Web Services Using the JMS Channel	199
JMS	199
Web Service and JMS	200
Specifications for Web Service Reliable Messaging	200
SOAP over HTTP versus SOAP over JMS	201
JMS in ServiceMix	203
ServiceMix-jms	203
Consumer and Provider Roles	204
servicemix-jms XBean Configuration	204
servicemix-jms Lightweight Configuration	206
Protocol Bridge	207
Web Service in the JMS Channel Binding Sample	208
ServiceMix Component Architecture for the JMS Web Service	209
Deploy the Web Service	210
XBean-based servicemix-jms Binding	211
XBean-based servicemix-eip Pipeline Bridge	212
XBean-based servicemix-http Provider Destination	212
Deploying the Sample and Starting ServiceMix	213
Test Web Service Using JMS Channel	214

Summary	219
Chapter 12: Java XML Binding using XStream	221
Java XML Binding	222
JAXB	223
XStream	223
ServiceMix and XStream	225
XStream in a Normalized Message Router Sample	226
Sample Use Case	226
Code HTTPClient	228
Unmarshalling to Transfer Objects	228
HttpInterceptor Component	230
XStreamInspector Component	232
Configure Interceptor and Inspector Components	232
Package Interceptor and Inspector Components	234
Deploy Interceptor and Inspector Components	234
Build and Run the Sample	235
Summary	236
Chapter 13: JBI Proxy	237
Proxy—A Primer	238
Proxy Design Pattern	238
JDK Proxy Class	239
Sample JDK Proxy Class	240
ServiceMix JBI Proxy	243
JBI Proxy Sample Implementing Compatible Interface	244
Proxy Code Listing	245
XBean-based JBI Proxy Binding	246
Deployment Configuration	247
Deploying and Running the Sample	247
JBI Proxy Sample implementing In-Compatible interface	248
Proxy Code Listing	248
XBean-based JBI Proxy Binding	250
Deployment Configuration	251
Deploying and Running the Sample	251
Invoke External Web Service from the ServiceMix Sample	252
Web Service Code Listing	252
Axis Generated Client Stubs	253
XBean-based JBI Proxy Binding	256
Deployment Configuration	258
Deploying and Running the Sample	258
Proxy and WSDL Generation	259

Summary	260
Chapter 14: Web Service Versioning	261
Service Versioning—A Means to SOA	261
Services are Autonomous	262
Change is the Only Constant Thing	262
All Purpose Interfaces	262
SOA Versioning—Don't Touch the Anti-Pattern	263
Types can Inherit—Why not My Schemas	265
If Not Versions, Then What	265
Strategy to Version Web Service	265
Which Level to Version	266
Version Control in a Schema	266
targetNamespace for WSDL	267
Version Parameter	267
Web Service Versioning Approaches	268
Covenant	268
Multiple Endpoint Addresses	269
Web Service Versioning Sample using ESB	270
Sample Use Case	270
Configure Components in ESB	274
Deploy and Run the Sample	285
Web Service Versioning Operational Perspective	287
Summary	287
Chapter 15: Enterprise Integration Patterns in ESB	289
Enterprise Integration Patterns	289
What are EAI Patterns?	290
EAI Patterns Book and Site	290
ServiceMix EAI Patterns	291
Why ServiceMix for EAI Patterns?	291
ServiceMix EAI Patterns Configuration	293
EAI Patterns—Code and Run Samples in ESB	294
Content-based Router	294
Notation	294
Explanation	295
Illustrative Design	295
Sample Use Case	296
Sample Code and Configuration	297
Deploy and Run the Sample	301
Content Enricher	303
Notation	303
Explanation	303

Illustrative Design	303
Sample Use Case	304
Sample code and configuration	305
Deploy and Run the Sample	307
XPath Splitter	308
Notation	309
Explanation	309
Illustrative Design	309
Sample Use Case	310
Sample Code and Configuration	311
Deploy and Run the Sample	312
Static Recipient List	313
Notation	314
Explanation	314
Illustrative Design	314
Sample Use Case	315
Sample Code and Configuration	316
Deploy and Run the Sample	318
Wiretap	319
Notation	319
Explanation	320
Illustrative Design	320
Sample Use Case	320
Sample Code and Configuration	321
Deploy and Run the Sample	323
Message Filter	323
Notation	324
Explanation	324
Illustrative Design	324
Sample Use Case	325
Sample Code and Configuration	326
Deploy and Run the Sample	328
Split Aggregator	329
Notation	329
Explanation	329
Illustrative Design	330
Sample Use Case	330
Sample Code and Configuration	331
Deploy and Run the Sample	332
Pipeline	334
Notation	334
Explanation	335
Illustrative Design	335
Sample Use Case	336
Sample Code and Configuration	337
Deploy and Run the Sample	339
Static Routing Slip	339
Notation	340

Table of Contents

Explanation	340
Illustrative Design	340
Sample Use Case	341
Sample Code and Configuration	342
Deploy and Run the Sample	344
Summary	345
Chapter 16: Sample Service Aggregation	347
Provision Service Order—Business Integration Sample	347
Solution Architecture	348
JB1-based ESB Component Architecture	350
Understanding the Message Exchange	351
Deploying and Running the Sample	365
Summary	366
Chapter 17: Transactions, Security, Clustering, and JMX	367
Cross Cutting Concerns—Support Inside ServiceMix	368
Transactions	368
Security	371
Clustering	373
JMX	377
Sample Demonstrating Transaction	377
Sample Use Case	378
Configure Transaction in ServiceMix	379
Deploy and Run the Sample	382
Sample demonstrating Security	383
Sample Use Case	384
Configure Basic Authorization in servicemix-http	384
Deploy and Run the Sample	387
Sample Demonstrating Clustering	388
Sample Use Case	389
Configure ServiceMix Cluster	391
Deploy and run the sample	395
Sample demonstrating JMX	397
Enable JMX in ServiceMix Application	397
Initialize JMX Console—MC4J	398
Retrieve WSDL through JMX	400
Summary	402
Index	403

Preface

You're all in the business of software development. Some of you are architects and developers while few others are technology managers and executives. For many of you, ESB is encroaching and JBI is still an unknown – a risk previously avoided but now found to be inescapable. Let us tame these buzzwords in the context of SOA and Integration.

While you do the day to day programming for solving business problems, you will be generating business code as well as business integration code. The traditional Java/J2EE APIs provide you with enough tools and frameworks to do the business coding. The business code will help you to implement a business service such as creating orders or finding products. On the contrary, business integration code wires together multiple applications and systems to provide seamless information flow. It deals with patterns of information exchange across systems and services, among other things. This is where the new Java API for Integration – Java Business Integration (JBI) seeks attention.

JBI provides a collaboration framework which has standard interfaces for integration components and protocols to plug into, thus allowing the assembly of Service Oriented Integration (SOI) frameworks following the Enterprise Service Bus (ESB) pattern. JBI is based on JSR 208, which is an extension of Java 2 Enterprise Edition (J2EE). The book first discusses the various integration approaches available and introduces ESB, which is a new architectural pattern which can facilitate integrating services. In doing so, we also introduce ServiceMix, an Apache Open Source Java ESB. Thus for each of the subsequent chapters, we limit our discussion to a major concern which we can address using ESB and then also showcase this with samples which you can run using ServiceMix. If you are a person with a non-Java background, the book still benefits you since most of the integration wiring happens in XML configuration files.

The aim of this book is to prepare an architect or developer for building integration solutions using ESB. To that end, this book will take a practical approach, emphasizing how to get things done in ServiceMix with code. On occasions, we will delve into the theoretical aspects of ESB, and such discussions will always be supplemented with enough running samples. The book, thus, attempts to distill some of the knowledge that has emerged over the last decade in the realm of Java Integration. Quite different from the other books in the related topics, you don't need a 4GB RAM or some heavy, vendor specific IDE/Workshops to run the samples. Instead, get set with the latest JDK and a text editor and few other lightweight frameworks including Tomcat and you are ready to go. Enough about the hype, supplement with what you've heard with some substance and code.

Happy Reading!

What This Book Covers

Chapter 1 begins with a quick tour on Enterprise Integration and the associated issues so that you can better understand the problem which we are trying to solve, rather than following a solution for an unknown problem. We also introduce Enterprise Service Bus (ESB) architecture and compare and contrast that with other integration architectures.

Chapter 2 introduces Java Business Integration (JBI) and inspects the need for another standard for Business Integration, and also looks into the details on what this standard is all about.

Chapter 3 introduces ServiceMix, which is an open source ESB platform in the Java programming language, built from the ground up with JBI APIs and principles. It runs through a few other ESB products also.

Chapter 4 looks at how we have been binding services locally or remotely even before the ESB became popular. The chapter will demonstrate how tunneling using conventional J2EE tools will help to expose even technology-specific API services. An example of such a service is an EJB service to be exposed through firewalls over HTTP so that the service becomes technology agonistic.

Chapter 5 introduces XFire, which is a new generation Java SOAP framework to easily expose web services. Here we demonstrate the integration capabilities of the XFire. Then we can do integration using XFire within the JBI Architecture also.

Chapter 6 teaches you JBI packaging and deployment. After going through this chapter the reader will be able to build, package, and deploy integration artifacts as standard JBI packages into the JBI container.

Chapter 7 teaches you how to create your own components and deploy them onto the JBI container. This chapter visits the core API from the ServiceMix as well as from the JBI specification which will function as useful helper classes using which you can develop integration components quickly.

Chapter 8 shows you how to bind Enterprise Java Beans components to the ESB. EJB is the Distributed Component paradigm in the Java-J2EE world and the industry has a lot invested in this technology. Coexisting services with those components will help you to reuse those existing investments so that we can continue building new systems based on higher levels of SOA maturity.

Chapter 9 shows POJO Binding using JSR181 to the ESB. POJO components can be easily exposed as WSDL-compliant services to the JBI bus.

Chapter 10 illustrates how to bind the web services to the ServiceMix ESB, thus creating a web services gateway at the ESB layer.

Chapter 11 looks at how Java Message Service (JMS), which is a platform dependent messaging technology, can increase the QOS features of web services by making web services accessible through the JMS channel.

Chapter 12 introduces Java XML binding and XStream, which is a simple open source library to serialize the Java objects to XML and back again. We will show the XStream integration with ESB demonstrating streaming of data across the bus.

Chapter 13 visits the JDK Proxy classes and then explains the JBI Proxy in detail with examples. We show a practical use of the JBI Proxy – Proxying the external web services in the JBI bus.

Chapter 14 explains versioning in the SOA context and explains various strategies and approaches to versioning services. It also explains and demonstrates a versioning sample leveraging the targetNamespace attribute. Versioning services, especially versioning of web services, is a topic of heated discussion in many forums and sites.

Chapter 15 explains that the EAI patterns are nuggets of advice made out of aggregating basic Message Exchange Pattern elements to solve frequently recurring integration problems. We can look at EAI patterns as design patterns for solving integration problems. This chapter will demonstrate many of the common EAI patterns.

Chapter 16 looks into a sample use case. One of the useful applications of ESB is to provide a "Services Workbench" wherein which we can use various integration patterns available to aggregate services to carry out the business processes.

Chapter 17 visits a few selected QOS features such as Transactions, Security, Clustering, and Management which can impact the programming and deployment aspects using ESB.

What You Need for This Book

To install and run most of the samples mentioned in this book all you need is the following:

- Latest Java SDK (JDK) installed in your machine.
- Apache Open Source Enterprise Service Bus—ServiceMix.
- Apache Ant build tool.
- A simple text editor, like Textpad.
- Any other software required is mentioned which is downloadable free from the net.

Who is This Book for

This book is aimed at Java developers and integration architects who want to become proficient with Java Business Integration (JBI) standard. They are expected to have some experience with Java and to have developed and deployed applications in the past, but need no previous knowledge of JBI. The book can also be useful to anyone who has been struggling to understand ESB and how it differs from other architectures and to understand its position in SOA.

This book primarily targets IT professionals in the field of SOA and Integration solutions—in other words, intermediate to advanced users. You are likely to find the book useful if you fall into any of the following categories:

- A programmer, designer or architect in Java who wants to learn and code in JBI or ESB.
- A programmer, designer or architect who doesn't normally code in Java can still benefit from this book, since we 'assemble integration components' using XML with little to no Java code.
- An IT Manager or an Officer who knows well about SOA or SOI but want to see something in code (you can adorn your flashy presentations with some live code too).

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 three styles for code. Code words in text are shown as follows: "For example, inside the `<component>` tag you can configure properties on the component."

A block of code will be set as follows:

```
<target name="run">
  <java classname="HttpInOutClient" fork="yes" failonerror="true">
    <classpath refid="classpath"/>
    <arg value="http://localhost:8912/EsbServlet/hello/" />
    <arg value="HttpSoapRequest.xml" />
  </java>
</target>
```

Any command-line input and output is written as follows:

```
cd ch03\Servlet
ant
```

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: "The components being initialized, in our case, include **trace**, **timer**, **httpGetData**, and **httpReceiver**."



Important notes appear in a box like this.



Tips and tricks appear like this.

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.

Downloading the Example Code for the Book

Visit http://www.packtpub.com/files/code/4404_Code.zip, to directly download the example code.

The downloadable files contain instructions on how to use them.

Errata

Although we have taken every care to ensure the accuracy of our contents, 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

Why Enterprise Service Bus

Today's enterprise is not confined to the physical boundaries of an organization. Open systems and an open IT infrastructure strives to provide interoperability not only between platforms, runtimes, and languages, but also across enterprises. When our concerns shift from networked systems to networked enterprises, a whole lot of opportunities open up to interact with enterprise applications. Whether it is for trading partners to collaborate through their back-end systems, or for multichannel deployments where consumers can use a whole lot of user agents like web and mobile handsets, the opportunities are endless. This also introduces the issues and concerns to be addressed by network, integration, and application architects. Today, companies that have been built through mergers or rapid expansions have **Line of Businesses (LOB)** and systems within a single enterprise that were not intended to interact together. More often than not these interactions fail and are discredited.

Let's begin with a quick tour of enterprise integration and the associated issues so that we can better understand the problem which we are trying to solve, rather than follow a solution for an unknown problem. At the end of this chapter, you should be in a position to identify what we are trying to aim with this book. We also introduce Enterprise Service Bus (ESB) architecture, and compare and contrast it with other integration architectures. Then we can better understand how Java Business Integration (JBI) helps us to define ESB-based solutions for integration problems.

In this chapter we will cover the following:

- Problems faced by today's enterprises
- Enterprise Application Integration (EAI)
- Different integration architectures
- ESB
- Compare and contrast service bus and message bus
- Need for an ESB-based architecture

Boundary-Less Organization

Jack Welch, of General Electric (GE), invented the boundary-less organization. Meaning that, organizations should not only be boundary-less, but should also be made permeable. "Integrated information" and "integrated access to integrated information" are two key features that any enterprise should aim for, in order to improve their organizational business processes. Boundary-less doesn't mean that organization has no boundaries; it means that there's a smooth and efficient flow of information across boundaries. While enterprise portals and web services give a new face for this emerging paradigm, the skeleton of this "open enterprise" is provided by an open IT infrastructure. The flesh is provided by emerging trends in Enterprise Application Integration (EAI) practice.

Let us briefly visit a few selected issues faced by today's enterprises.

Multiple Systems


In a home business or in small scale ventures, we start up with systems and applications in silos which cater to the entire setup. When the business grows, we add more verticals, which are functionally separated entities within the same organization. It goes without saying that, each of these verticals or LOB will have their own systems. People ask why they have different systems. The answer is because they are doing functionally different operations in an enterprise and hence they need systems carrying out different functionality for them. For example, an HR system will manage and maintain employee related data whereas the marketing relations will use Customer Relationship Management (CRM) systems. These systems may not be interconnected and hence impede information flow across LOBs. Adding to this, each LOB will have their own budgeting and cost constraints which determine how often systems are upgraded or introduced.

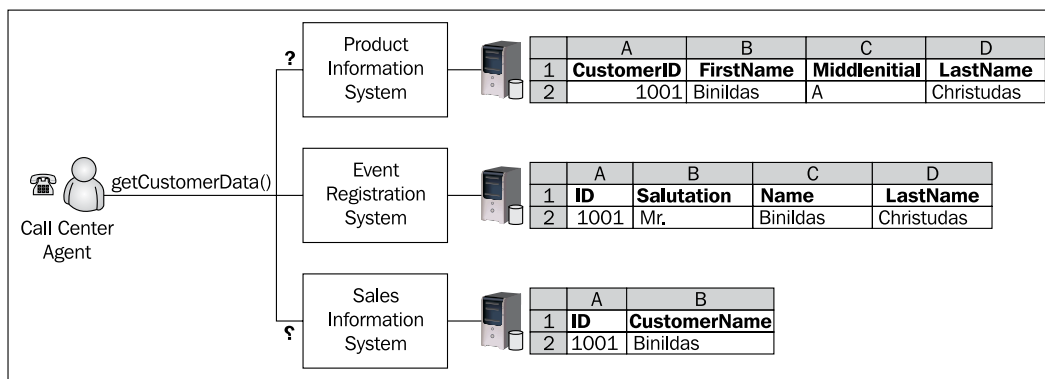
No Canonical Data Format

Multiple LOBs will have their own systems, and hence their own data schemas, and a way of accessing the data. This leads to duplication of data, which in turn leads to multiple services providing views for the same entity in different ways.

Let's consider the example shown in the following figure. There is one LOB system, Product Information System, which will keep track of customers who have registered their interest for a particular product or service. Another LOB system, Event Registration System, will provide membership management tools to customers, for a sales discount program. Sales Information System will have to track all customers who have registered their interest for any upcoming products or services. Thus, we can see there is no unified view of the Customer entity at the enterprise-level. Each

LOB will have its own systems and their own view of Customer. Some will have more attributes for the Customer, while others a few. Now the question is which system owns the Customer entity? Rather, how do we address the data stewardship issue (This is represented in the figure using the symbols "?" and "§")?

 **Data stewardship** roles are common when organizations attempt to exchange data, precisely and consistently, between computer systems, and reuse data-related resources where the steward is responsible for maintaining a data element in a metadata registry.

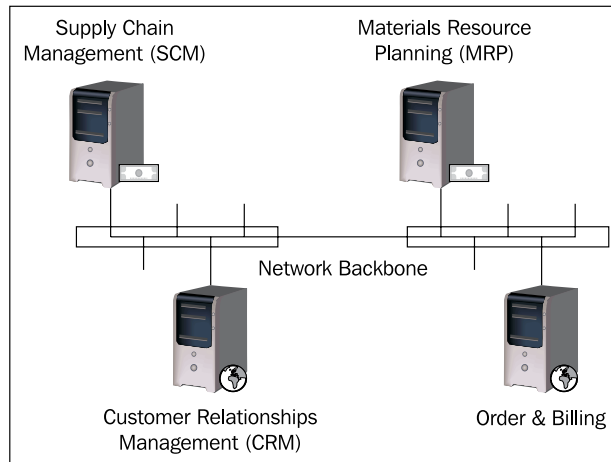


Autonomous, but Federated

Now the question is how long can these systems remain isolated? Rather, can we bring all these systems under a central, single point of control? The fact is, systems cannot remain isolated, and also they cannot be controlled centrally. This is because every system needs data from every (or most) other systems. Can we then integrate everything together into a single big system so that we don't have the problem of integration at all? The question seems tempting, but this is analogous to attempting to boil sea water.

Different departments or verticals within the same organization need autonomy and so do their systems. Without the required level of autonomy, there is no freedom which will hinder innovation. Constant innovation is the key to success, in any walk of life. So, departments and their systems need to be autonomous. But, since they require each other's data, they need to integrate. This leads to the necessity for a farm of autonomous systems, which are all federated to the required level to facilitate information flow.

The following figure represents systems that are autonomous, but federated to facilitate information flow:



Intranet versus Internet

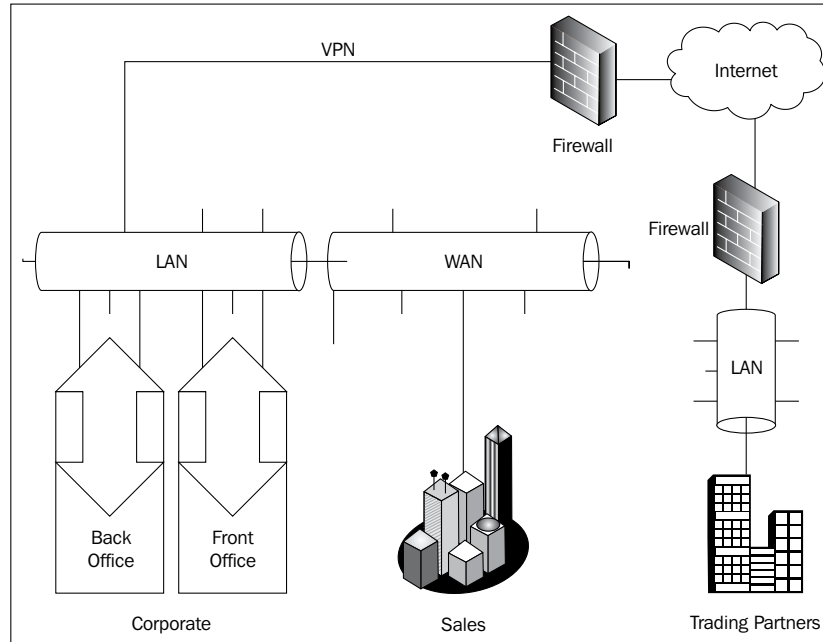
Integrating different functional departments within an organization is not a big hurdle, since everything is under the control of the organization. But the picture changes the moment the organization grows beyond a certain limit. Twenty first century organizations are growing beyond a single location; many of them are truly global organizations. They have operations around the globe, in multiple countries, with multiple legal, technical, and operational environments. The good news is that we have technologies like Internet, Wide Area Networks, and Virtual Private Networks for connectivity. This means global organizations will have their systems federated across the globe. All these systems will evolve due to constant innovation and business changes. They have to integrate across the firewall, and not every protocol and format can be easily traversed across the firewall.

Trading Partners

Organizations conduct business with other organizations. An online retailer's system has to partner with its wholesaler's inventory systems. These systems are not federated in any manner due to the fact that they belong to multiple organizations. There exists no federation due to the competitive nature between organizations too. But they have to collaborate; otherwise there is no business at all. So, information has to flow across organizational systems in the Internet. This is what we mean by a permeable organization boundary which allows for constant information exchange

on 24 X 7 bases, with adequate **Quality of Service (QOS)** features. Thus the necessity is to extend the enterprise over the edge (firewall), and this activity has to happen based on pre-plans and not as an afterthought.

The following figure explains a trading partners system that is not federated but the information flow is through the Internet:



Integration

Knowingly or unknowingly, applications and systems have been built over decades in silos, and it is the need of the day for these applications and systems to interchange data. At various levels depending upon the level of control, the data can be interchanged at multiple layers, protocols, and formats. There seems to be a general trend of "Technology of the day" solutions and systems in many organizations. Neither this trend is going to end, nor do we need to turn back at them. Instead, we need to manage the ever changing heterogeneity between systems.



Application and system portfolio entropy increases with technology innovation.

I don't know if there is a global movement to bring standardization to innovation in technology. This is because the moment we introduce rules and regulations to innovation, it is no longer an innovation. This statement is made, even after acknowledging various world wide standard bodies' aim and effort to reduce system and application entropy. A few years back, Common Object Request Broker Protocol's (CORBA) promise was to standardize binary protocol interface so that any systems could interoperate. If we look at CORBA at this point, we can see that CORBA has not attained its promise, that doesn't mean that we need to throw away all those systems introduced during the 90's because we cannot integrate.

Enterprise Application Integration

David S. Linthicum defined EAI as:

The unrestricted sharing of data and business processes among any connected applications and data sources in the enterprise.

This is a simple, straightforward definition. In my entire career, I have been fortunate enough to participate in much new generation IT system development for domains such as Airline, Healthcare, and Communications. Most of the time, I've been writing either adapters between systems, or negotiating and formalizing data formats between desperate systems. I know this is not because the former system's architects haven't put a long term vision to their systems in the angle of interoperability, but because systems have to evolve and interoperate in many new ways which were not foreseen earlier. This pushes integration providers to define new software pipes across applications. When we start this activity it might be elegant and straight forward, but sooner than later we realize that our integration pipes have no central control, administration, or management provisions.

Integration Architectures

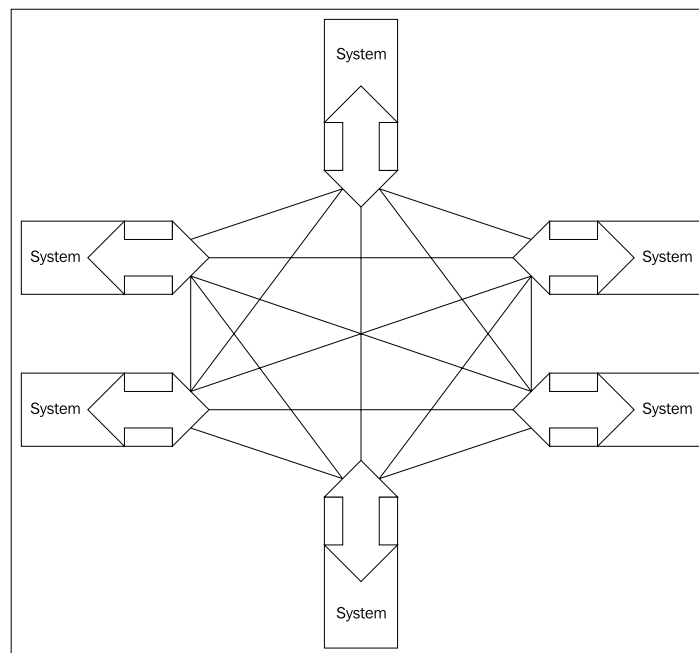
The first and foremost step in understanding integration architectures is to understand the different topologies existing in integration arena, and to appreciate the vital difference between them. If one can understand the true difference, half the job is already done. Understanding the difference will enable the integration architect to attach prescriptive architecture for a given integration problem. Let us now understand the basic integration architectures that are listed as follows:

- Point-to-Point solution
- Hub-and-Spoke solution
- Enterprise Message Bus Integration
- Enterprise Service Bus Integration

Point-to-Point Solution

EAI has traditionally been done using point-to-point integration solutions. In point-to-point, we define an integration solution for a pair of applications. Thus we have two end points to be integrated. We can build protocol and/or format adapters, or transformers at one or either end. This is the easiest way to integrate, as long as the volume of integration is low. We normally use technology-specific APIs like FTP, IIOP, remoting or batch interfaces, to realize integration. The advantage is that between these two points, we have tight coupling, since both ends have knowledge about their peers.

The following figure is the diagrammatic representation of the point-to-point integration:



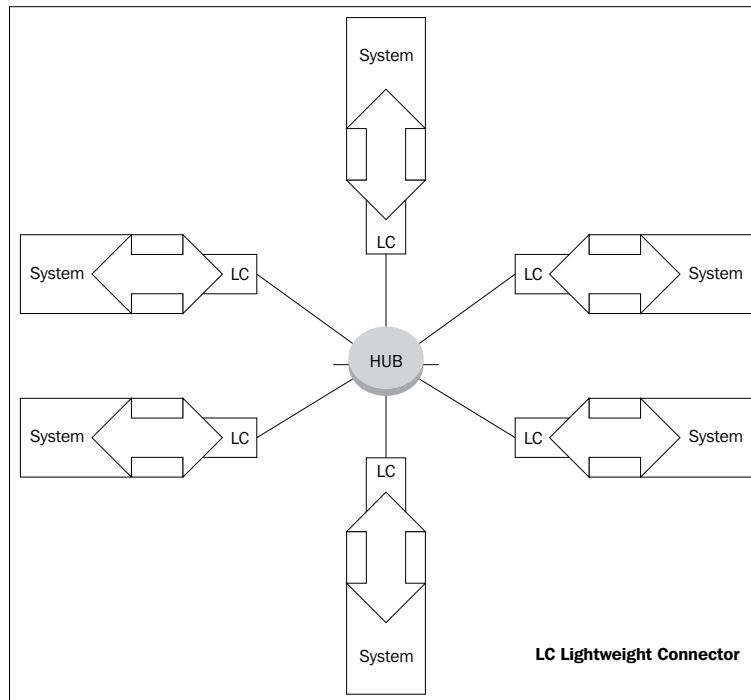
Hub-and-Spoke Solution

Hub-and-spoke architecture is also called the **message broker** and is similar to point-to-point architecture in that it provides a centralized **hub** (broker) to which all applications are connected. When multiple applications connect in this manner, we get the typical hub-and-spoke architecture. Another distinguishing feature of the hub-and-spoke architecture is that each application connects with the central hub through lightweight connectors. The **lightweight connectors** facilitates for application integration with minimum or no changes to the existing applications.

Message transformation and routing takes place within the hub. This architecture is a major improvement to the point-to-point solution since it reduces the number of connections required for integration. Since applications do not connect to other applications directly, they can be removed from the integration topology by removing from the hub. This insulation reduces disruption in the integration setup.

There is a major drawback with the hub-and-spoke architecture. Due to the centralized nature of the hub, it is a single point of failure. If the hub fails, the entire integration topology fails. A solution to this problem is to cluster the hub. A **cluster** is a logical grouping of multiple instances running simultaneously and working together. But clustering is not the right solution to the problem of single point of failure. This is due to the fact that the very point in having a hub-and-spoke architecture is to have a single point of control. This drawback may be offset to some extent by the fact that most of the clustering solutions provide central management and monitoring facilities, which will provide some form of centralized control.

The following figure is the diagrammatic representation of the hub-and-spoke integration:

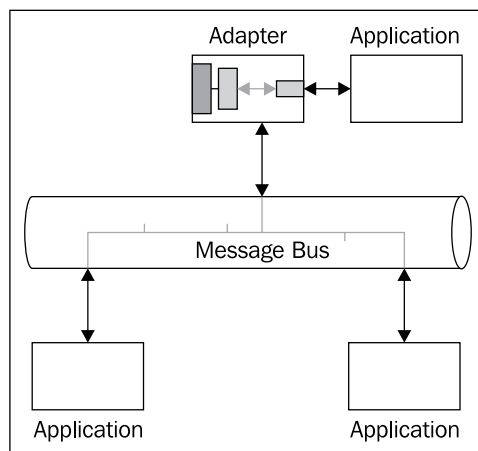


Enterprise Message Bus Integration

While the hub-and-spoke architecture makes use of lightweight connectors for applications to integrate together through a central hub, many a times the integrating applications need to interact in a decoupled fashion, so that they can be easily added or removed without affecting the others. An enterprise message bus provides a common communication infrastructure, which acts as a platform-neutral and language-neutral adapter between applications.

This communication infrastructure may include a message router and/or Publish-Subscribe channels. So applications interact with each other through the message bus with the help of request-response queues. If a consumer application wants to invoke a particular service on a different provider application, it places an appropriately formatted request message on the request queue for that service. It then listens for the response message on the service's reply queue. The provider application listens for requests on the service's request queue, performs the service, and then sends (if) any response to the service's reply queue.

We normally use vendor products like IBM's Websphere MQ (WMQ) and Microsoft MQ (MSMQ), which are the best class message queue solution to integrate applications in the message bus topology. As shown in the following figure, sometimes the applications have to use adapter which handle scenarios such as invoking CICS transactions. Such an adapter may provide connectivity between the applications and the message bus using proprietary bus APIs, and application APIs. The Message bus also requires a common command structure representing the different possible operations on the bus. These command sets invoke bus-level primitives which includes listening to an address, reading bytes from an address, and writing bytes to an address.

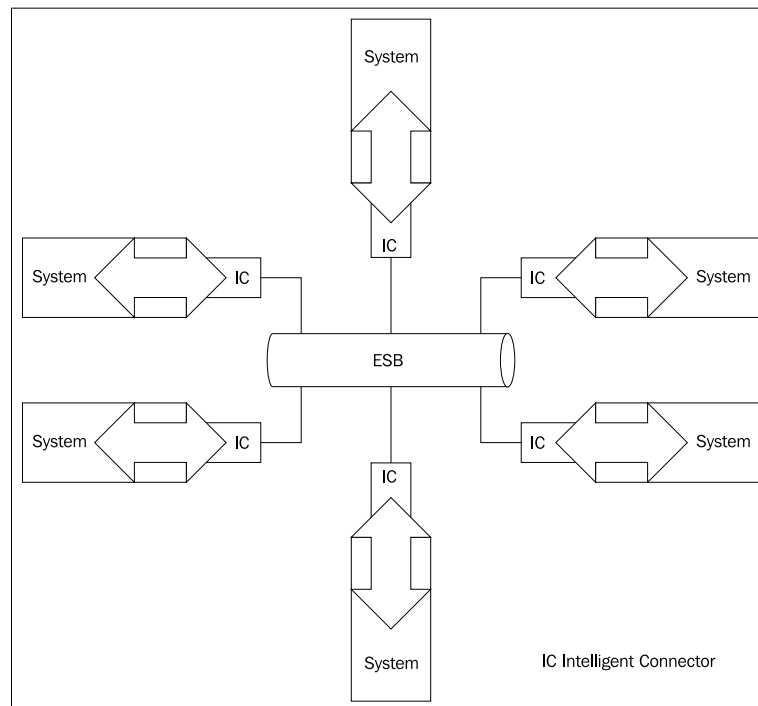


Enterprise Service Bus Integration

The service bus approach to integration makes use of technology stacks to provide a bus for application integration. Different applications will not communicate directly with each other for integration; instead they communicate through this middleware Service Oriented Architecture (SOA) backbone. The most distinguishing feature of the ESB architecture is the distributed nature of the integration topology. Most ESB solutions are based on Web Services Description Language (WSDL) technologies, and they use Extensible Markup Language (XML) formats for message translation and transformation.

ESB is a collection of middleware services which provides integration capabilities. These middleware services sit in the heart of the ESB architecture upon which applications place messages to be routed and transformed. Similar to the hub-and-spoke architecture, in ESB architecture too, applications connect to the ESB through abstract, **intelligent connectors**. Connectors are abstract in the sense that they only define the transport binding protocols and service interface, not the real implementation details. They are intelligent because they have logic built-in along with the ESB to selectively bind to services at run time. This capability enhances agility for applications by allowing late binding of services and deferred service choice.

The following figure is the diagrammatic representation of ESB integration:



The above qualities of ESB provides for a true open enterprise approach. As we have discussed above, ESB is neither a product nor a separate technology; instead, ESB is a platform-level concept, a set of tools, and a design philosophy. What this means is, if we just buy a vendor product and install it in our IT infrastructure, we cannot say that we have ESB-based integration architecture. Instead ESB-based integration solutions are to be designed and built in the "ESB way". Tools and products help us to do this.

A list of major features and functionalities supported by an ESB will help us to understand the architecture better, which are listed as follows:

- Addressing and routing
- Synchronous and asynchronous style
- Multiple transport and protocol bindings
- Content transformation and translation
- Business process orchestration
- Event processing
- Adapters to multiple platforms
- Integration of design, implementation, and deployment tools
- QOS features like transactions, security, and persistence
- Auditing, logging, and metering
- Management and monitoring

Enterprise Service Bus versus Message Bus

Let's leave alone the point-to-point and the hub-and-spoke architectures, since it is rather easy to understand them and you have been doing them in one way or another. But when we discuss about ESB and message bus, we need to understand the similarities and differences between these two topologies.

Similarities and Differences

Let us first see how the message bus and the service bus are alike. In fact, both of them are aimed to solve the problem of integration and provide a common communication infrastructure, which acts as a platform-neutral and language-neutral adapter between applications. So mediation is the prime functionality provided by

both the architectures. Applications can integrate each other in a loosely coupled manner through this mediator, so that they will not be dependent on each other's interfaces. Last but not the least, using either the message bus or the service bus architecture, you can implement SOA-based integration solutions!

The last statement might be hard to digest – at least some of you might have thought that one is purely SOA-based while the other is not. But the fact is that both the message bus and the service bus helps enterprises to attain an SOA ecosystem, if architected properly, but neither of them by itself will automatically transfer existing non-SOA architecture into SOA-based architecture.

Now, what is the difference between a message bus and a service bus?

Before we dig into the differences let me give you a word of caution. Even though the main differences between a message bus and a service bus will be listed as follows, they may not be very obvious in the first read. Hence, I recommend the reader to go through the subsequent chapters and samples too, and get a feel of how we do things in the "ESB way", and then revisit the given differences.



The main difference between enterprise message bus and enterprise service bus architecture is in the manner in which the consumer or the client application interact with the messaging middleware.

More easily said, than understood! OK, let us worry less (I didn't say "let us worry not"!), understand more.

Service description and service discovery are two key elements to attain higher levels of SOA maturity. Only if something is described, can it be discovered. This is where a service registry is going to add value, there you can register a service description so that some other interested party can discover it.

Let us now come back to our message bus. We earlier saw that message bus applications interact with each other with the help of request-response queues. If you have ever worked in any messaging solution (like JMS) before, then you will appreciate the fact that queues are addressed by themselves, which will not give you any further information about the underlying service. Information on the operations available, or the messaging patterns to expect, or the schema for the types exchanged is never available at the queue-level. In other words, services are not described in a message bus.

What is available is just the messaging counterparts for the *put* and *get* primitives so that messages can be *sent to* and *received from* the message bus. So consumer or client applications should have pre-awareness of the format of the messages exchanged. This implies everything has to be known before hand – rather, static binding or compile-time binding becomes the norm.

Let us now consider the service bus. We said earlier that many ESB solutions are based on WSDL technologies, and they use XML formats for message translation and transformation. This by itself gives us lot of opportunities for service description and discovery. All the (minimum) details about the service will be available from the WSDL. Hence, message exchange patterns and message formats are readily available. Moreover, the consumer or client applications can discover a matching service from the ESB, given a set of messaging capabilities (WSDL capabilities) they are looking for. I used the term matching service because in an ideal ESB architecture the consumer is looking for capabilities which match their abstract service expectations (interface). It is the role of the ESB to route the requests to any concrete service implementation behind the bus which matches the requested interface.

The next big difference is the type of approach used in each architecture. In service bus architecture we used a standard-based approach. When services are WSDL-based, it brings a time tested and well adopted industry standard to integration. This has a big pay off when compared to traditional Message Oriented Middleware (MOM), because in the message bus architecture the adapters provide connectivity using proprietary bus APIs and application APIs. So, whereas in the pre-ESB world, we have been using CORBA IDL (Interface Definition Language), or Tuxedo FML (Field Manipulation Language), or COM/DCOM Microsoft IDL, and CICS common Area (COMMAREA) as the service definition language, we now use WSDL as the interface in standard-based ESB architectures.

Maturity and Industry Adoption

Having looked at a few of the similarities and differences between service bus and message bus, it is time now to look at realities which exist in the industry today. We agreed that an ESB can do lot many things, and for that matter a message bus can too. We then talked about the differences a service bus has to offer.

How mature is the technology today to address these differences? Have we started practical implementations of service bus in a big way yet? The answer to this question is neither yes nor no. The reason is that necessity runs before standards. Rather, when you agree that you need description and discovery along with other features for a service bus-based architectures, the question is, will the existing standards like Universal Description Discovery and Integration (UDDI) alone will help to achieve this? Maybe we need a simple and standard way to specify a pair of request-reply queues along with a HTTP URL (mechanism to specify HTTP URL is already available) in the WSDL itself. This way a consumer or client application can interact in the 'MOM way' through the ESB. Maybe we also need a simple and, again, a standard way to find and invoke a concrete service at the ESB, matching an abstract service interface.

These standards are evolving and are at different stages of adoption. Similar is the case of support for these capabilities across multiple vendors' solutions. Hence, the tail piece is that it is time now to gather all our experience in message bus based architectures, and leverage it to define and demonstrate service bus-based architecture. So, how do we decide that we need an ESB-based Architecture? Read on the next section and you will come to know.

Making the Business Case for ESB

Just like any one of you, I am not satisfied yet with enough reasons for why to use ESB. Hence, this section is going to explain more about the value proposition ESB is going to bring to your IT infrastructure.

There are a few concerns we need to address when we do point-to-point or a similar style of integration:

- How many channels do we need to define for complete interoperability?
- How easy it is to change a system interface, while still maintaining interoperability?
- How do we accommodate a new system to the IT portfolio?
- How much we can reuse system services in this topology?
- Where do we plug-in system management or monitoring functionality?

Let us address these issues one by one.

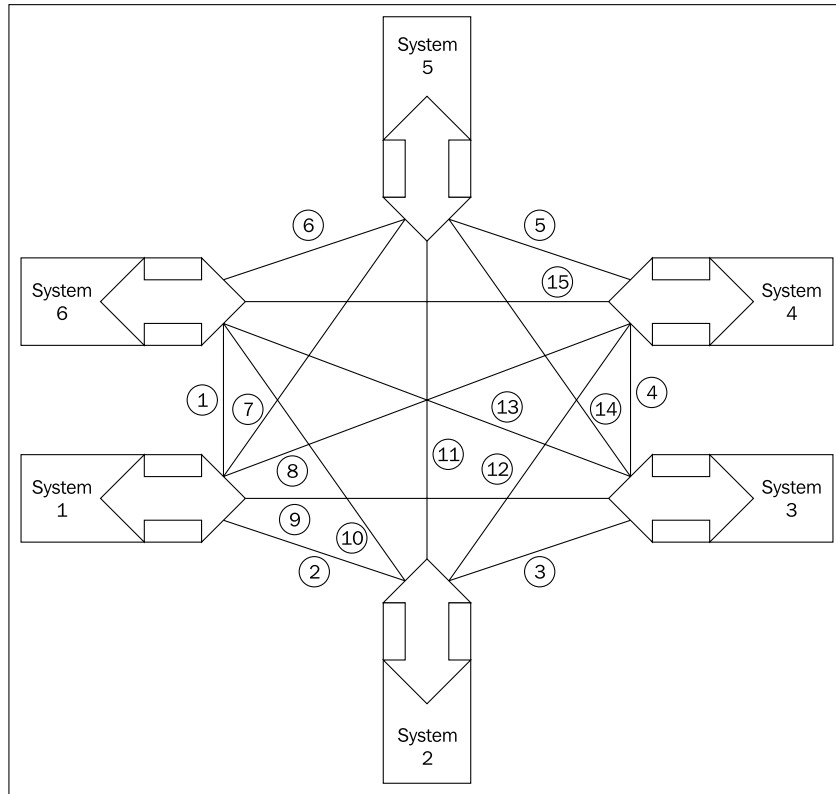
How many Channels

Let us go back to the point-to-point integration scenario and assume that we have four separate in-house systems, one for each of the four separate departments (HR, Accounts, R&D, and Sales). The operational problem the business faces is to interoperate these systems. In the point-to-point integration, we need to have independent channels of connection between each pair of systems. These channels are static, strongly typed, and without much intelligence for dynamic routing or transformation. The advantage is that it is rather easy to build the solution.

As shown in the next figure, if there are six systems (nodes) to be interconnected, we need at least thirty separate channels for both forward and reverse transport. If we add one more node to the IT portfolio, the number of channels to be defined goes up from thirty to forty two. This means, as time passes and as more and more systems and applications are added to the organization, the number of interconnections or channels to be defined between these systems rises exponentially, in the order of two.

We can generalize the number of channels (N_c) required for complete interconnection for n separate systems as:

$$N_c = n^2 - n$$



This number is still manageable for small organizations with a small number of systems, but experience has shown that this is going to be a nightmare for mid-sized and large-sized organizations. It is customary for such organizations to have more than fifty or hundred systems. For complete interoperability in a hundred system scenario, we need to define around ten thousand separate channels! Leave alone the definition, how do we maintain such a huge network of interconnection?

Perhaps, every system in an organization needn't interoperate with every other system. Again, experience has shown that only half of them need to interoperate fully, thus bringing down the figure to five thousand. What this means is, we have to build five thousand adapters to define channels to interoperate for these systems. Still this number is not manageable.

Volatile Interfaces

In a perfect world, we can somehow build five thousand separate adapters to interconnect a hundred systems by fifty percent, and then hope to relax. But the real scenario is far from perfect. The imperfection arises out of constant innovation and evolution, and this forces change even to system interfaces. An `int` data type in C++ is of two bytes whereas the same `int` in Java is of four bytes. We have already taken care of this data impedance by building adapters. But, what if on one fine morning, the C++ application starts spitting `float` instead of `int`? The existing adapter is going to fail, and we need to rework on the adapter in this channel.

The good news is, within an organization this can be controlled and managed since we have access to both the ends of the adapter. The scenario worsens when the interface changes in a B2B scenario where systems between two separate organizations interact. As we know already, this is going to happen. So, we have to build agile interfaces.

New Systems Introducing Data Redundancy

Adding new systems is an after effect of business expansion, but there are multiple risks associated with it. First, we need to integrate new systems with the existing systems. Secondly, many a time, new systems introduce data redundancy. This is because the same data might get entered into the network through multiple interfaces. Similar is the problem of the same data duplicated at different parts of the organization. This contradicts the requirements of Straight-Through Processing (STP). STP aims to enter transactional data only once into the system. This data can flow within or outside the organization. To achieve STP, there should be mechanism for seamless data flow, also in a flexible manner.

Data redundancy can be managed if and only if there is a mechanism to consolidate data in the form of Common Information Model (CIM). For example, the NGOSS's Shared Information/Data (SID) model from Telecom Management Forum is the industry's only true standard for development and deployment of easy to integrate, flexible, easy to manage OSS/BSS components. The SID provides an information or data reference model and a common information or data vocabulary from a business and systems perspective. Not every domain or organization has access to a readymade information model similar to this.

In such a scenario, ESB can play a vital role by providing the edge of the system view by wrapping and exposing available network resources. Thus, ESB provides hooks for a "leave-and-layer" architecture which means that instead of altering existing systems to provide a standards-based services interface they are wrapped with a layer that provides the services interface.

Service Reuse

Code and component reuse is a familiar concept amongst designers and developers. We have Gang of Four patterns to prescribe design solutions for recurring problems. But with the invention of SOA, we have what is called the Service Oriented Integration (SOI) which makes use of Enterprise Integration Patterns (EIP). SOI speaks about the best ways of integrating services so that services are not only interoperable but also reusable in the form of aggregating in multiple ways and scenarios. This means, services can be mixed and matched to adapt to multiple protocols and consumer requirements.

In code and component reuse, we try to reduce copy and paste reuse and encourage inheritance, composition, and instance pooling. A similar analogy exists in SOI where services are hosted and pooled for multiple clients through multiple transport channels. ESB does exactly this job in the best way integration world has ever seen. For example, if a financial organization provides a credit history check service, an ESB can facilitate reuse of this service by multiple business processes like a Personal Loan approval process or a Home Mortgage approval process.

System Management and Monitoring

Cross cutting concerns like transactions or security or even Service Level Agreement (SLA) management and monitoring are just a few of a set of features apart from business functionality that any enterprise class service ecosystem has to provide. The ESB architectures provides enough hooks to attach similar services onto the bus separate from normal business functionality. This ensures that these cross cutting functionality can be applied to all services and also enable us to manage, monitor, and control them centrally.

Enterprise Service Bus

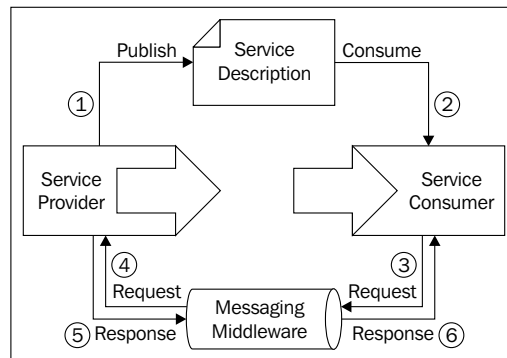
Having understood the various integration problems, which can be addressed in multiple ways, of which ESB is of the prime concern and is central to our discussion in this book, we will try to understand the ESB better here.

Service in ESB

It is time now to understand "Service" in ESB. Why don't we call it Enterprise Component Bus?

Services are exposed functionalities which are clearly defined, self contained, and which will not depend on the state and context of peer services. As shown in the following figure, the service has a **service description** (similar to WSDL) and it is this description which is exposed. It is never necessary to expose the implementation details, and in fact, we shouldn't expose them. It is this hidden nature of the implementation which helps us to replace services with similar services. Thus, service providers expose the service description and service consumers find them.

Once the service description is found service consumers can bind to it. Moreover, it is during the time of actual binding, we need more information like transport protocols. As we know, we can handle with transport details with the help of protocol adapters. Intelligent adapters are required if we need to route messages based on content or context. These adapters are of a different kind like splitters and content based routers. ESB is all about a run-time platform for these kinds of adapters and routers, which are specifically designed for hosting and exposing services.



Abstraction beyond Interface

Almost all OOP languages support interface-driven development and hence this is not a new concept. Separating interface from implementation abstracts the Service Consumer from the implementation details from a Service Provider perspective. But, what about abstracting out this interface itself? If the Service Consumer interacts directly with the Service Provider, both these parties are tightly coupled and it may be difficult to change the interface. Instead, a service bus can effectively attach itself to a service registry. This means, a consumer can even be unaware of the service interface.

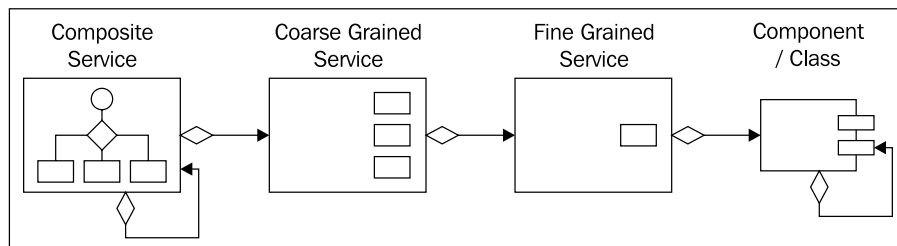
Now, just in time, the consumer can selectively choose an appropriate interface from a registry, then bind to a suitable implementation through an ESB, and invoke the service at run time. Such kinds of flexibility come at the cost of design time or compile time type checking, which can be error prone during service invocation using conventional tools. An ESB with its set of tools and design primitives helps developers to effectively address this.

Service Aggregation

The key value of aggregation, here, is not so much in being able to provide core services, which developers have to build anyway. It is to facilitate an arbitrary composition of these services in a declarative fashion, so as to define and publish more and more composite services. Business Process Management (BPM) tools can be integrated over ESB to leverage service aggregation and service collaboration. This facilitate reuse of basic or core (or fine grained) services at the business process-level. So, granularity of services is important which will also decide the level of reusability.

Coarse Grained or Composite Services consume Fine Grained Services. Applications which consume Coarse Grained Services are not exposed to the Fine Grained Services that they use. Composite Services can be assembled from Coarse Grained as well as Fine Grained Services.

To make the concept clear, let us take the example of provisioning a new VOIP Service for a new customer. This is a Composite Service which in turn calls multiple Coarse Grained Services such as *validateOrder*, *createOrVerifyCustomer*, and *checkProductAvailability*. Now, *createOrVerifyCustomer*, the Coarse Grained Service in turn call multiple Fine Grained Services like *validateCustomer*, *createCustomer*, *createBillingAddress*, and *createMailingAddress*.



As is shown in the above figure, a Composite Services is an aggregate of other Composite Services and/or Coarse Grained Services. Similarly, Coarse Grained Services are again aggregates of other Fine Grained Services. Whereas, Fine Grained Services implement minimum functionality and are not aggregates of other services. ESB here acts as a shared messaging layer for connecting applications and other services throughout the enterprise, thus providing a workbench for service aggregation and composition.

Service Enablement

Services can be designed but if left alone they will sit idle. The idea behind true SOA is not just service definition or implementation, but also service enablement. By service enablement, we mean enabling, configuring, tuning, and attaching QOS attributes to services, so that they are accessible, as per SLA irrespective of formats or transport protocols.

Using today's techniques like Aspect Oriented Programming (AOP), Annotations and byte code instrumentation, service enablement can be done without any dependency on the services framework. An ESB does exactly this. It is to be noted that a JBI-based ESB will do this API-less. It is not completely API-less, in the sense that there is always a dependency on an XML configuration or annotation. The differentiating factor is that this dependency is externalized so that they are not cemented to the service implementation.

Service Consolidation

Service consolidation is an important step to eliminate redundant information and data. As a part of service consolidation, we may have to first consolidate IT infrastructure and data assets in the network. There should be clear ownership for these two entities, so that we know who or which applications are changing which sectors of the data assets. Once this is done, the next step is to do application portfolio analysis and do direct mapping between services and data operations (business-level CRUD).

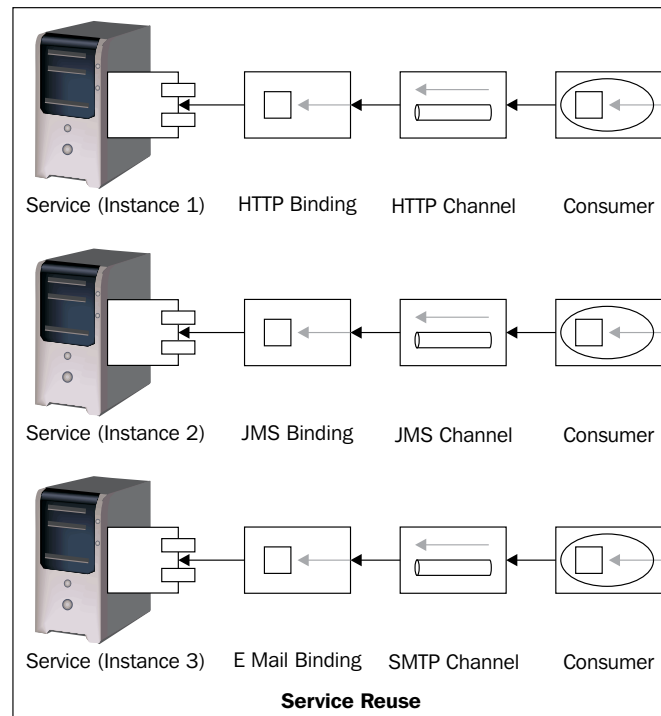
A point to be noted here is that I am not talking about exposing simple Create, Read, Update, and Delete operations to be exposed as services, since they are not (coarse grained) services at all. They are just functions to alter the constants of data model and if we expose them as services, we are trying to expose our internal data modeling, which is going to impact internal agility. Instead, what we have to expose is our coarse grained business operations (processes) which carry out useful business functions on behalf of client applications. When we do that, we can map which services need to be invoked to finish an online retailing experience, and which other services are responsible for realizing the retail booking in the back office systems.

Thus services are named, labeled, and all the more indexed. And you look at the index for a particular service similar to the way you look at a book index, to find a particular subject. Thus, we effectively pool services and consolidate unused capacity formerly spread over many connected systems and servers.

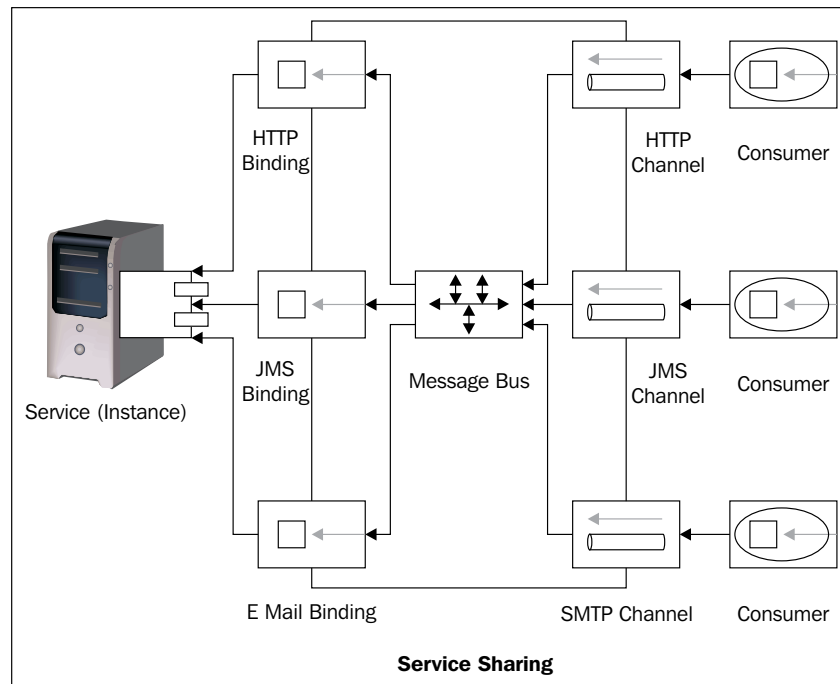
Service Sharing

Message bus facilitates service sharing as compared to service reuse. In fact service sharing is one of the key aspects of SOA.

In service reuse, the same service is hosted in different deployment environments due to various reasons. Even though the component or the code behind the service is reused here, there is significant overhead in terms of deployment, management, and maintenance. The Service reuse scenario is as shown in the following figure:



Service sharing is, in contrast, the true sharing of a service itself. This means the service hosting environment is configured for different scenarios and consumers access the same service instance through multiple channels or formats. SOA web services can be leveraged to achieve this, and this is further enabled using a message bus as shown in the following figure:



Linked Services

Microsoft SQL Server has the ability to query remote database objects as if they were local to the server. Linked servers are useful in deployments, where the data can be split by functional area into databases with very little coupling as a means of scale out strategy. This means that a scaled out database can look like a single large database to an application. Similar to this, ESB can be seen as a means to link services. Binding to remote end points is synonymous to linking to remote databases—an application code can interact with the linked services as if they were local to the invoking code.

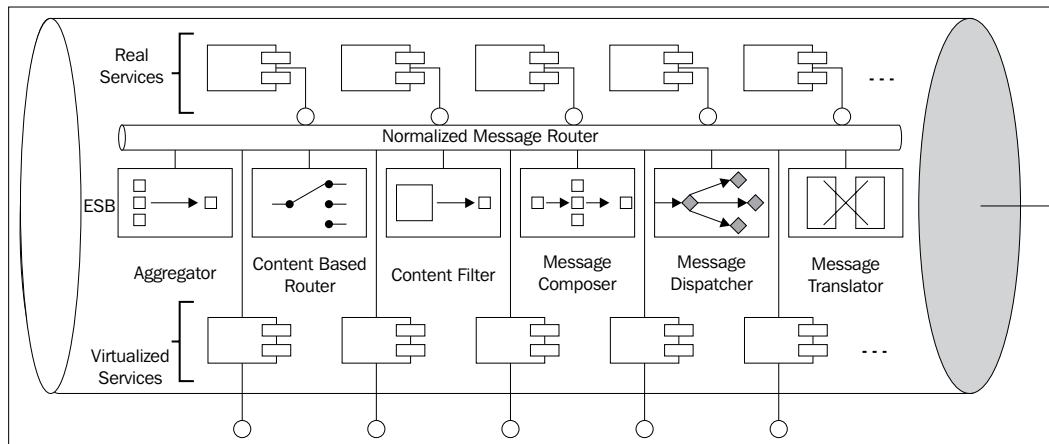
Virtualization of Services

By service virtualization, I mean dividing the organization's information assets into "Virtual Services" without regard to the transport or binding protocol of the actual implementation components. Corporate network assets (information) can reside in multiple places like in a centralized mainframe or distributed across multiple systems implemented using DCOM, RMI, IIOP, remoting, or SOAP protocols.

Service wrapping and service aggregation are two methods by which we can virtualize them to be exposed in a consistent way to external or internal clients. Thus, virtualization is closely related to "black box" reuse of services by providing a different interface at the same abstraction-level. The ESB abstracts behind the scene addressing, translations, and aggregation.

Well-defined interfaces facilitate development of interacting software systems not only in different LOBs, but also at different times, thus providing a suitable abstraction for any consumers of the service. But abstraction has a negative side too—software systems, components, or services designed for one interface will not work with those designed for another interface. This, in fact, is an interoperability issue which can be a concern, especially in a world of networked computers where the trend is to move data and information freely.

Service virtualization is a way to circumvent this issue by mapping the visible interface of a service onto the interface and resource of an underlying, possibly different, real service.



This figure depicts the concept of Service Virtualization, where we make use of core ESB features like Aggregation, Routing, and Translation to republish the existing service assets. These are explained in detail in subsequent chapters.

Services Fabric

We, integration people, need to learn lessons from the way network architects organize their network channels. When using fiber, network architects have three network topologies. For the fibre channel option, we have three network topologies via any from the following list:

- Point-to-point
- Arbitrated loop
- Fabric

In the fabric topology, there are one or more hardware switches in addition to ports on each device to network. Comparatively, in ESB we have intelligent adapters. Multiple clients can hit the ESB fabric, and intelligent adapters at the ESB edge are responsible for dynamic binding of client requests to appropriate concrete service implementations. Thus, ESB services fabric functions as the virtualization layer. In other words, the ESB provides infrastructure capabilities and application plug-in points for an SOA fabric for the whole enterprise.

Summary

Information integration is as old as Information Technology (IT) itself. We have been doing it in multiple ways and using multiple technology stacks. But, what is more important is that, at the end of the day, the integrated solution should be solving more existing problems and introducing less new problems. This is where selecting and choosing appropriate integration architecture is most important in the IT landscape.

We have seen what ESB architecture is and how it is different from other peer integration architectures. So far so good, and today, you can read any number of documents on ESB from the Internet. We, as architects and developers now, also need to implement them in our code, not just in sketches in white papers and presentations. How can we do that using Java?

In the next chapter (Java Business Integration), we will see exactly what Java has to offer in defining ESB architectures.

2

Java Business Integration

Integration has been an area for specialists for years, since no standards exist across vendor products. This increases the Total Cost of Ownership (TCO) to implement and maintain any integration solution. Even though integration is a necessary evil, CIOs and IT managers postpone decisions and actions, and sometimes go for ad-hoc or temporary solutions. Any such activity will complicate the already confused stove pipes and it is the need of the hour to have standardization. Here we are going to inspect the need of another standard for business integration, and also look into the details of what this standard is all about.

So we will cover the following in this chapter:

- Service oriented architecture in the context of integration
- Relationship between web services and SOA
- Service oriented integration
- J2EE, JCA, and JBI – how they relate
- Introduction to JBI
- JBI Nomenclature – main components in JBI
- Provider-consumer roles in JBI
- JBI Message Exchange Patterns (MEP)

SOA—The Motto

In Chapter 1, we went through integration and also visited the major integration architectures. We have been doing integration for many decades in proprietary or ad-hoc manner. Today, the buzz word is SOA and in the integration space, we are talking about Service Oriented Integration (SOI). Let us look into the essentials of SOA and see whether the existing standards and APIs are sufficient in the integration space.

Why We Need SOA

We have been using multiple technologies for developing application components, and a few of them are listed as follows:

- Remote Procedure Call (RPC)
- Common Object Request Broker Architecture (CORBA)
- Distributed Component Object Model (DCOM)
- .NET remoting
- Enterprise Java Beans (EJBs)
- Java Remote Method Invocation (RMI)

One drawback, which can be seen in almost all these technologies, is their inability to interoperate. In other words, if a .NET remoting component has to send bytes to a Java RMI component, there are workarounds that may not work all the times.

Next, all the above listed technologies follow the best Object Oriented Principles (OOP), especially hiding the implementation details behind interfaces. This will provide loose coupling between the provider and the consumer, which is very important especially in distributed computing environments. Now the question is, are these interfaces abstract enough? To rephrase the question, can a Java RMI runtime make sense out of a .NET interface?

Along these lines, we can point out a full list of doubts or deficiencies which exist in today's computing environment. This is where SOA brings new promises.

What is SOA

SOA is all about a set of architectural patterns, principles, and best practices to implement software components in such a way that we overcome much of the deficiencies identified in traditional programming paradigms. SOA speaks about services implemented based on abstract interfaces where only the abstract interface is exposed to the outside world. Hence the consumers are unaware of any implementation details. Moreover, the abstract model is neutral of any platform or technology. This means, components or services implemented in any platform or technology can interoperate. We will list out few more features of SOA here:

- Standards-based (WS-* Specifications)
- Services are autonomous and coarse grained
- Providers and consumers are loosely coupled

The list is not exhaustive, but we have many number of literature available speaking on SOA, so let us not repeat it here. Instead we will see the importance of SOA in the integration context.

SOA and Web Services

SOA doesn't mandate any specific platform, technology, or even a specific method of software engineering, but time has proven that web service is a viable technology to implement SOA. However, we need to be cautious in that using web services doesn't lead to SOA by itself, or implement it. Rather, since web services are based on industry accepted standards like WSDL, SOAP, and XML; it is one of the best available means to attain SOA.

Providers and consumers agree to a common interface called Web Services Description Language (WSDL) in SOA using web services. Data is exchanged normally through HTTP protocol, in Simple Object Access Protocol (SOAP) format.

WSDL

WSDL is the language of web services, used to specify the service contract to be agreed upon by the provider and consumer. It is a XML formatted information, mainly intended to be machine processable (but human readable too, since it is XML). When we host a web service, it is normal to retrieve the WSDL from the web service endpoint. Also, there are mainly two approaches in working with WSDL, which are listed as follows:

- Start from WSDL, create and host the web service and open the service for clients; tools like `wsdl2java` help us to do this.
- Start from the types already available, generate the WSDL and then continue; tools like `java2wsdl` help us here.

Let us now quickly run through the main sections within a WSDL. A WSDL structure is as shown here:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace=
    "http://versionXYZ.ws.servicemix.esb.binildas.com" ...>
  <wsdl:types>
    <schema elementFormDefault="qualified"
      targetNamespace="http://version20061231.ws.
        servicemix.esb.binildas.com"
      xmlns="http://www.w3.org/2001/XMLSchema">
    </schema>
  </wsdl:types>
  <wsdl:message name="helloResponse">
    <!-- other code goes here -->
```



```
</wsdl:message>
<wsdl:portType name="IHelloWeb">
  <!-- other code goes here -->
</wsdl:portType>
<wsdl:binding name="HelloWebService20061231SoapBinding"
  type="impl:IHelloWeb">
  <!-- other code goes here -->
</wsdl:binding>
<wsdl:service name="IHelloWebService">
  <wsdl:port binding="impl:HelloWebService20061231SoapBinding"
    name="HelloWebService20061231">
    <wsdlsoap:address
      location="http://localhost:8080/AxisEndToEnd20061231/
        services/HelloWebService20061231"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

We will now run through the main sections of a typical WSDL:

- **types:** The data types exchanged are expressed here as an XML schema.
- **message:** This section details about the message formats (or the documents) exchanged.
- **portType:** The PortType can be looked at as the abstract interface definition for the exposed service.
- **binding:** The PortType has to be mapped to specific data formats and protocols, which will be detailed out in the binding section.
- **port:** The port gives the URL representation of the service endpoint.
- **service:** Service can contain a collection of port elements.

Since JBI is based on WSDL, we will deal with many WSDL instances in the subsequent chapters.

SOAP

In web services, data is transmitted over the wire in SOAP. SOAP is an XML-based messaging protocol. SOAP defines a set of rules for structuring messages that can be used for simple one-way messaging and is useful for performing RPC-style interactions. Even though SOAP is not tied to any particular transport protocol, HTTP is the popular one.

A SOAP-encoded RPC dialogue contains both a request message and a response message. Let us consider a simple service method that takes a `String` parameter and returns a `String` type:

```
public String hello(String param);
```

The respective SOAP request is shown as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <ns1:hello soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/
      encoding/"
      xmlns:ns1="http://AxisEndToEnd.axis.apache.
        binildas.com">

      <in0 xsi:type="soapenc:string"
        xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
        Binil
      </in0>
    </ns1:hello>
  </soapenv:Body>
</soapenv:Envelope>
```

Similar to the request, the SOAP response is shown below:

```
<?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <ns1:helloResponse soapenv:encodingStyle="http://schemas.xmlsoap.
      org/soap/encoding/"
      xmlns:ns1="http://AxisEndToEnd.axis.apache.
        binildas.com">

      <helloReturn xsi:type="soapenc:string"
        xmlns:soapenc="http://schemas.xmlsoap.org/soap/
          encoding/">

        Return From Server
      </helloReturn>
    </ns1:helloResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

As is the case with WSDL, we will be dealing with many instances of SOAP requests and responses in our discussions in the coming chapters. Hence, let us not delve into the intricacies of WSDL and SOAP, as there are many text books doing the same. Let us continue with our discussion on integration.

Service Oriented Integration (SOI)

In Chapter 1, we identified ESB as an architectural pattern different from traditional integration architectures, which support standards-based services for integration. Gartner originally identified ESB as the core component in an SOA landscape. Gartner says:

SOA will be used in more than 80% of new mission-critical applications and business processes by 2010

Another market strategy study, conducted by Gartner in SOA reveals that:

The ESB category grew by 160% from year 2004 to year 2005

As is true with SOA, ESB-based integration has been increasingly using web services standards and technologies for service description, hosting, and accessing. The advantage is that we not only reap all the benefits of ESB architecture (explained in Chapter 1), but also make sure that the services exposed at the bus are complying with industry standards. This means consumers and providers makes use of existing toolsets and frameworks to interact with the ESB. Most of the current SOA toolsets support web services and related technologies. This opens up a whole lot of opportunities in the integration space too, which leads to SOI. When we virtualize services in the ESB, it is very critical to provide a uniform ecosystem for all kind of services, whether it is written in Java, or .NET, or C++.

If the services are as per the web service standards, ESB can provide the mediation services which will help to interconnect these services. Externalizing mediation logic out of services to an ESB will remove in-code coupling between services. To sum up, our current SOA infrastructure still exists which hosts services and takes care of service management and governance. Also an ESB provides SOI in the form of mediation, which provides communication (along with other features) between services.

JB1 in J2EE—How they Relate

The Java 2 Enterprise Edition (J2EE) platform provides containers for client applications, web components based on servlets and Java Server Pages (JSP), and Enterprise JavaBeans (EJB) components. These J2EE containers provide deployment and run-time support for application components. They also provide a federated view of the platform-level services, provided by the underlying application server for the application components. In this section, we will look at where JB1 is positioned in the J2EE stack.

Servlets, Portlets, EJB, JCA, and so on

The world of Java is fabulous. Java speaks about portable code (`.class` files), portable data (JAXP and XML), portable components (EJB), and now portable services (JAX-WS). However, until now, there was no standard way by which we could do business-level integration across application servers from multiple vendors.

Business integration can be done with business components or with business services. We know that EJB components can interoperate across multiple vendor's application servers, and similar is the case with web services. We have been doing integration within our traditional J2EE programming paradigm in the pre-ESB era. The challenge of accessing, integrating, and transforming data has largely been done by developers using manual coding of J2EE components like POJO, EJB, JMS, and JCA. These J2EE APIs are optimized for business purpose programming, and not for integration. The after effect is that the J2EE programmers have to use low-level APIs to implement integration concerns.

Programmers create integration code having hard coded dependencies between specific applications and data sources, which are neither efficient nor flexible. Hence, a way to do integration in a loosely coupled manner and to port the integration solution as a whole is missing. Components and services are only a part of the integration solution. It also includes the strategy to integrate, the patterns adapted to route and transform messages and similar artifacts. All these differ for different vendor's products and JBI is trying to bring standardization across this space.

JBI and JCA—Competing or Complementing

The J2EE Connector Architecture (JCA) defines standards for connecting the J2EE platform to heterogeneous EIS systems. Examples of EIS systems include CICS, IMS, TPF systems, ERP, database systems, and legacy applications not written in the Java programming language. The JCA enables the integration of EISs with application servers and enterprise applications by defining a developer-level API and a vendor-level SPI. The SPIs enables an EIS vendor to provide a standard resource adapter for its EIS. The resource adapter plugs into an application server, thus providing connectivity between the EIS, the application server, and the enterprise application.

If an application server vendor has extended its system to support the JCA, it is assured of seamless connectivity to multiple EISs. So following JCA, an EIS vendor needs to provide just one standard resource adapter, which has the capability to plug-in to any application server that supports the JCA. He can now be assured that his EIS will plug into the J2EE continuum.

JCA-based adapters are usually used to integrate compliant ESBs with EIS. This means, by using JCA, an EIS vendor can be assured that his EIS can integrate with compliant ESBs. JCA provides the most efficient way of resource pooling, thread pooling, and transaction handling on JMS and other resource adapters. We have already seen the similarities between a message bus and a service bus. Along those lines, we can understand how important is the MOM technology such as JMS. Hence, the importance JCA has got in integration.

We also need to understand one subtle difference between JBI and JCA here. JCA is designed to support the traditional request-response model, but fails to support complex long running transactions and integration scenarios. Much back-end integration does not always fit well with the synchronous request-response model and JBI has got something else to offer here. JBI is based on a mediated message exchange pattern. That is, when a component sends a message to another component, the message exchange is mediated by the JBI infrastructure. As we will see shortly, JBI supports multiple message exchange patterns. Moreover, JBI-based ESBs will give more functionalities such as service composition, BPM, etc., which makes sense in handling back-end integration and long running transactions.

JBI—a New Standard

Every organization would like to leverage existing IT staff to run their integration effort similar to their daily IT development and operations. However, integration is a different art than normal software development and that is why we have integration architects. That doesn't mean integration experts have to be experts in vendor X's product, and are back to novice-level in vendor Y's product. If this has to happen, we need to do similar activities across multiple vendor's products.

As we already discussed, integration is not that simple as it involves integration architectures, integration patterns, and MOM expertise. We need to build the extra intelligent adapters around services and endpoints. These adapters are to work in tandem with the platform provider to give low-level features like communication, session, state, transport, and routing. Thus, business integration is not limited by just application programming, but involves components and engineering processes at the application and platform interface-level. Hence, to make integration artifacts portable, even the integration vendors (who deal with platform services) have a big part to play. This is where we need standardization as components and adapters should be portable across several vendor products.

Let me try to explain this point a bit further, as this is the prime concern in today's integration initiatives. Since the inception of J2EE, we have been using multiple J2EE containers from different vendors. Most of these J2EE containers also have an integration stack, which can be either plugged into their existing J2EE containers or

can be run as standalone. Websphere Business Integration (WBI) message broker and BEA Aqualogic Service Bus (ALSB) are just a few of them in the list. It is true, that most of these integration stacks too support standard-based integration, following SOA principles and patterns. What about the integration artifacts as such?

When I say integration artifacts, I refer to any integration libraries available, or any custom configuration the developers do to the above libraries, the packaging and deployment artifacts they generate. In other words, if we create a `.jar` or `.ear` file containing some integration solution in ALSB; can we port it to WBI later? I leave this question open for the user community of these frameworks to answer. So, what has happened here? As always, the evolution of standards lags behind the evolution of technology. Hence, even before JBI, we had multiple programming paradigms to solve integration issues, but then they were tied to the vendor's environment. The promise of JBI is to bring portability across different JBI containers for the integration artifacts.

JBI in Detail

In Chapter 1, we discussed ESB architecture which can facilitate the collaboration between services. JBI provides a collaboration framework which provides standard interfaces for integration components and protocols to plug into, thus allowing the assembly of SOI frameworks.

JSR 208

JSR 208 is an extension of J2EE, but it is specific for JBI Service Provider Interfaces (SPI). SOA and SOI are the targets of JBI and hence it is built around WSDL. Integration components can be plugged into the JBI environment using a service model based on WSDL. Service composition is a major target in ESB architecture and the JBI environment aggregates multiple service definitions in the WSDL form into the message infrastructure.

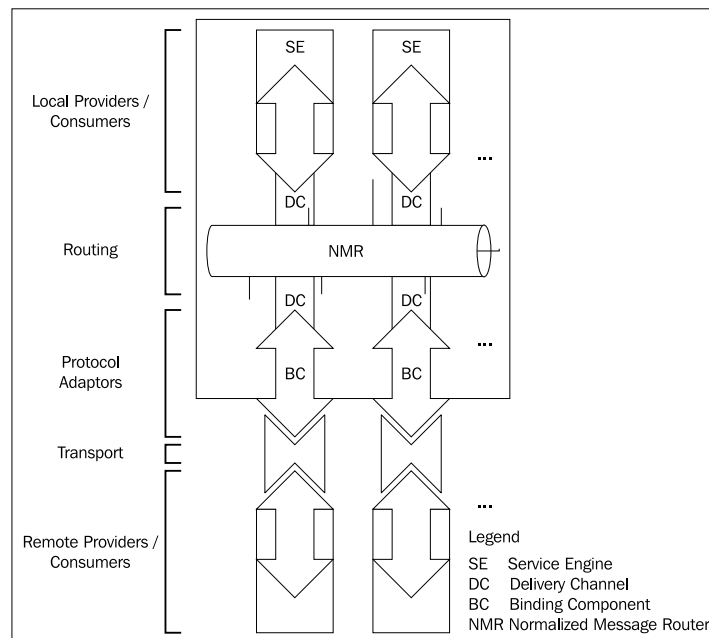
In the context of a larger service composition, we have multiple partners (service providers or service consumers) and the metadata for interaction of these individual partners are termed as the **business protocol**. The metadata of choreography played by a business process in a business protocol is termed as the **abstract business process**. Partner processes interact with each other by looking at abstract business process; it is the ESB's job to realize this abstract business process. JSR aims to make this abstract business process definition portable. This means the wiring details between components in a service composition scenario can be extracted into a separate layer and it is this layer which we are speaking about JSR 208. Thus, JSR 208 or JBI is a foundation for realizing SOI.

JSR 208 mandates the following for JBI components:

- **Portable:** Components are portable across JBI implementations.
- **Manageable:** Components can be managed in a centralized manner.
- **Interoperable:** Components should be able to provide service to and consume services from other components, despite the fact that they come from different sources and transport protocols.

JBI Nomenclature

This section helps us to understand the major components in JBI architecture and their roles with reference to the following figure:



The major components are explained here:

- **JBI environment:** A JBI environment is a single Java Virtual Machine (JVM) where we can deploy integration artifacts. This JBI can be a standalone ESB or an ESB embedded in the JVM of an application server. In the latter case, even an EJB component deployed in an application server can function as a provider or consumer to the ESB, thus further narrowing down the bridge between traditional J2EE application servers and the relatively new ESB.

- **Service Engine (SE):** SEs are service providers or service consumers deployed locally within a JBI environment. They provide the actual business logic like transformation. Transformation service can be done with the help of an XSLT engine by using a stylesheet. Another engine may use JCA to give a data access service, or Business Process Execution Language (BPEL), or even custom logic to integrate legacy code like that in CICS or mainframe.
- **Binding Components (BC):** BC provide communications protocol support and they are normally bound to components deployed remotely from the JBI run time. In fact, nothing prevents a user from defining a binding for a local service in the case where it closely resembles SE. Thus BC provides remote access to services for remote service providers and consumers.



The distinction between SE and BC is important for various pragmatic reasons. Mainly, it separates service logic from binding protocol. This facilitates reusability.

- **JBI Container:** Similar to the container in an application server, a JBI environment by itself is a JBI container. This container hosts SE and BC. The interesting part is a SE is again a container for engine specific entities like XSLT engine, stylesheets, rules engines, and scripts. Thus, a JBI environment is a container of containers whereas a service engine is a container for hosting WSDL defined providers and consumers local to the JBI.
- **Normalized message:** A normalized message consists of two parts – the actual message in XML format, and message metadata which is also referred to as the message context data. The message context data helps to associate extra information with a particular message, as it is processed by both plug-in components and system components in the bus.
- **Normalized message router (NMR):** The nerve of the JBI architecture is the NMR. This is a bus through which messages flow in either direction from a source to a destination. The specialty of this router is that messages are always in a normalized format, irrespective of the source or destination. NMR thus acts as a lightweight messaging infrastructure which facilitates actual message exchange in a loosely coupled fashion. NMR provides varying QOS functionalities and the three levels of message delivery guarantee provided by NMR are listed as follows:
 - **Best effort:** Message may be delivered only once or more than once, or even the message can be dropped.

- **At Least Once:** Message has to be delivered once or more, but cannot be dropped. Hence, duplicates can exist.
- **Once and only once:** It is guaranteed that messages will be delivered once and only once.
- **Pluggable components:** The loosely coupled feature of ESB is due to the pluggable architecture for service components. There are two broad categories of pluggable components called SE and BC. Pluggable components can play the role of service provider, service consumer, or both. Pluggable components are connected to the NMR through a Delivery Channel.
- **Service providers and service consumers:** There are mainly two roles played by pluggable components within an ESB, the service provider and service consumer. Components can act as both a provider and a consumer at the same time. Service definition is available in the form of WSDL and this is the only shared artifact between providers and consumers. Since endpoint information is not shared between providers and consumers in the ESB, loosely coupled integration is facilitated.

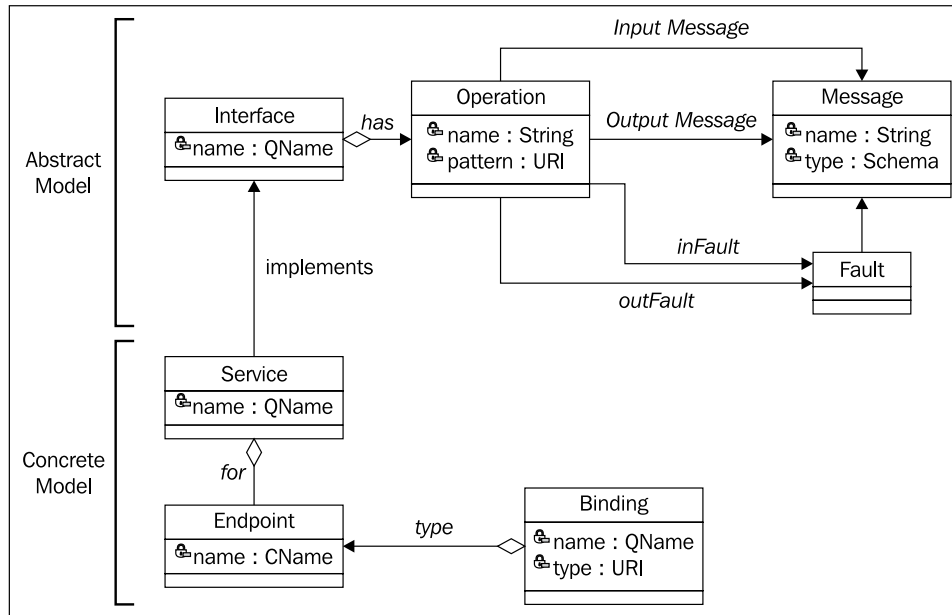
It is to be noted that a single service definition may be implemented by multiple providers. Similarly, a consumer's service definition of interest might be provided by multiple providers too, in ESB architecture. Moreover, the role of both provider and consumer can be played by the component either directly or through a proxy for a remote service.

- **Delivery Channel (DC):** DC connects a message source and a destination. The messaging infrastructure or the NMR is the bus for message exchange for multiple service providers and consumers. Hence when a service provider has got information to communicate, it doesn't just fling the message into the NMR, but adds the message to a particular DC. Similarly, a message consumer doesn't just pick it up at random from the messaging system. Instead, the consumer receives the message from a particular DC. Thus DCs are logical addresses in the ESB.

Provider—Consumer Contract

In the JBI environment, the provider and consumer always interact based on a services model. A service interface is the common aspect between them. WSDL 1.1 and 2.0 are used to define the contract through the services interface.

The following figure represents the two parts of the WSDL representation of a service:



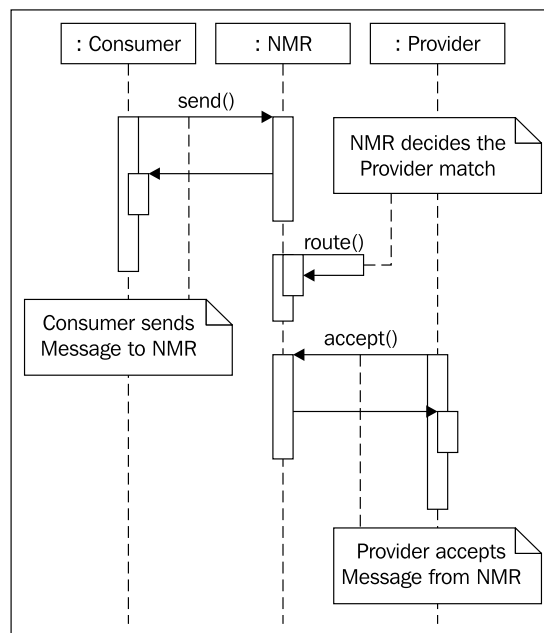
In the **Abstract Model**, WSDL describes the propagation of a message through a type system. A message has sequence and cardinality specified by its Message Exchange Pattern (MEP). A Message can be a Fault Message also. An MEP is associated with one or more messages using an Operation. An Interface can contain a single Operation or a group of Operations represented in an abstract fashion – independent of wire formats and transport protocols.

An Interface in the Abstract Model is bound to a specific wire format and transport protocol via Binding. A Binding is associated with a network address in an Endpoint and a single Service in the concrete model aggregates multiple Endpoints implementing common interfaces.

Detached Message Exchange

JBIs-based message exchange occurs between a Provider and Consumer in a detached fashion. This means, the Provider and Consumer never interact directly. In technical terms, they never share the same thread context of execution. Instead, the Provider and Consumer use JBI NMR as an intermediary. Thus, the Consumer sends a request message to the NMR. The NMR, using intelligent routers decides the best matched service provider and dispatches the message on behalf of the Consumer. The Provider component can be a different component or the same component as the Consumer itself. The Provider can be an SE or a BC and based on the type it will execute the business process by itself or delegate the actual processing to the remotely bound component. The response message is sent back to the NMR by the Provider, and the NMR in turn passes it back to the Consumer. This completes the message exchange.

The following figure represents the JBI-based message exchange:

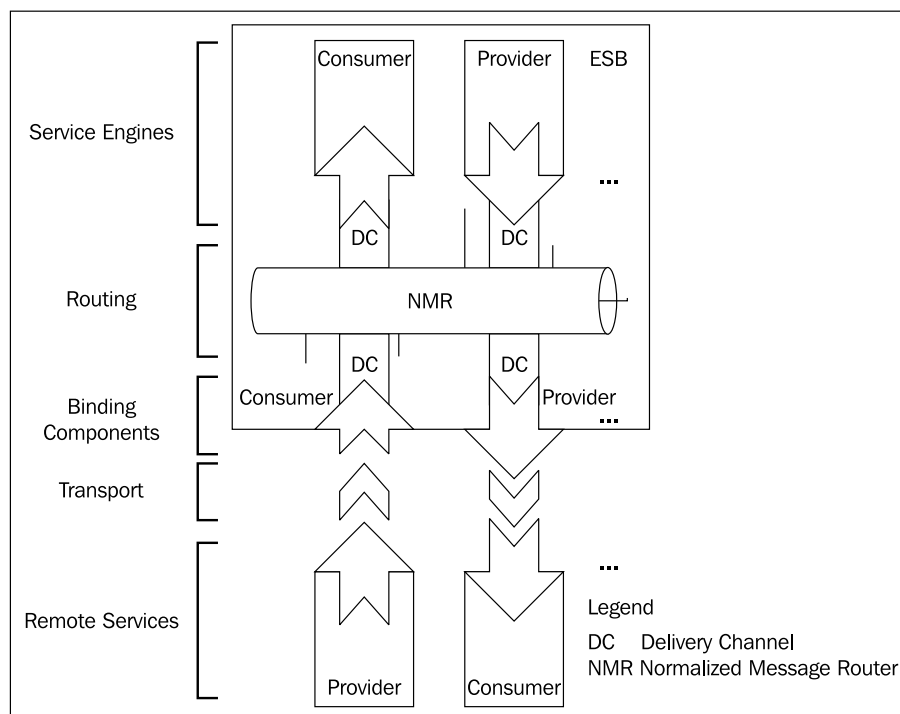


There are multiple patterns by which messages are exchanged, which we will review shortly.

Provider—Consumer Role

Though a JBI component can function as a Consumer, a Provider, or as both a Consumer and Provider, there is clear cut distinction between the Provider and Consumer roles. These roles may be performed by bindings or engines, in any combination of the two. When a binding acts as a service Provider, an external service is implied. Similarly, when the binding acts as a service Consumer, an external Consumer is implied. In the same way, the use of a Service Engines in either role implies a local actor for that role.

This is shown in the following figure:



The Provider and Consumer interact with each other through the NMR. When they interact, they perform the distinct responsibilities (not necessarily in the same order).

The following is the list of responsibilities, performed by the Provider and Consumer while interacting with NMR:

1. *Provider*: Once deployed, the JBI activates the service provider endpoint.
2. *Provider*: Provider then publishes the service description in WSDL format.
3. *Consumer*: Consumer then discovers the required service. This can happen at design time (static binding) or run time (dynamic binding).
4. *Consumer*: Invokes the queried service.
5. *Provider and Consumer*: Send and respond to message exchanges according to the MEP, and state of the message exchange instance.
6. *Provider*: Provides the service by responding to the function invocations.
7. *Provider and Consumer*: Responds with status (fault or done) to complete the message exchange.

During run-time activation, a service provider activates the actual services it provides, making them known to the NMR. It can now route service invocations to that service.

```
javax.jbi.component.ComponentContext context ;
// Initialized via. AOP
javax.jbi.messaging.DeliveryChannel channel = context.
                                     getDeliveryChannel();
javax.jbi.servicedesc.ServiceEndpoint serviceEndpoint = null;
if (service != null && endpoint != null)
{
    serviceEndpoint = context.activateEndpoint
                       (service, endpoint);
}
```

The Provider creates a WSDL described service available through an endpoint. As described in the Provider-Consumer contract, the service implements a WSDL-based interface, which is a collection of operations. The consumer creates a message exchange to send a message to invoke a particular service. Since consumers and providers only share the abstract service definition, they are decoupled from each other. Moreover, several services can implement the same WSDL interface. Hence, if a consumer sends a message for a particular interface, the JBI might find more than one endpoint conforming to the interface and can thus route to the best-fit endpoint.

Message Exchange

A message exchange is the "Message Packet" transferred between a consumer and a provider in a service invocation. It represents a container for normalized messages which are described by an exchange pattern. Thus message exchange encapsulates the following:

- Normalized message
- Message exchange metadata
- Message exchange state

Thus, message exchange is the JBI local portion of a service invocation.

Service Invocation

An end-to-end interaction between a service consumer and a service provider is a service invocation. Service consumers employ one or more service invocation patterns. Service invocation through a JBI infrastructure is based on a 'pull' model, where a component accepts message exchange instances when it is ready. Thus, once a message exchange instance is created, it is sent back and forth between the two participating components, and this continues till the status of the message exchange instance is either set to 'done' or 'error', and sent one last time between the two components.

Message Exchange Patterns (MEP)

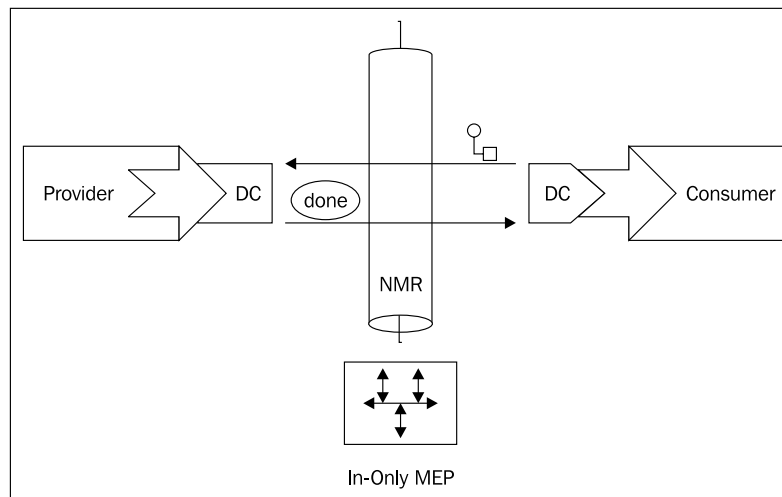
Service consumers interact with service providers for message exchange employing one or more service invocation patterns. The MEP defines the names, sequence, and cardinality of messages in an exchange. There are many service invocation patterns, and, from a JBI perspective, any JBI-compliant ESB implementation must support the following four service invocations:

- **One-Way:** Service consumer issues a request to the service provider. No error (fault) path is provided.
- **Reliable One-Way:** Service consumer issues a request to the service provider. Provider may respond with a fault if it fails to process the request.
- **Request-Response:** Service Consumer issues a request to the service provider, with expectation of response. Provider may respond with a fault if it fails to process request.
- **Request Optional-Response:** Service consumer issues a request to the service provider, which may result in a response. Both consumer and provider have the option of generating a fault in response to a message received during the interaction.

The above service invocations can be mapped to four different MEPs that are listed as follows.

In-Only MEP

In-Only MEP is used for one-way exchanges. The following figure diagrammatically explains the In-Only MEP:



In the In-Only MEP normal scenario, the sequence of operations is as follows:

- Service Consumer initiates with a message.
- Service Provider responds with the status to complete the message exchange.

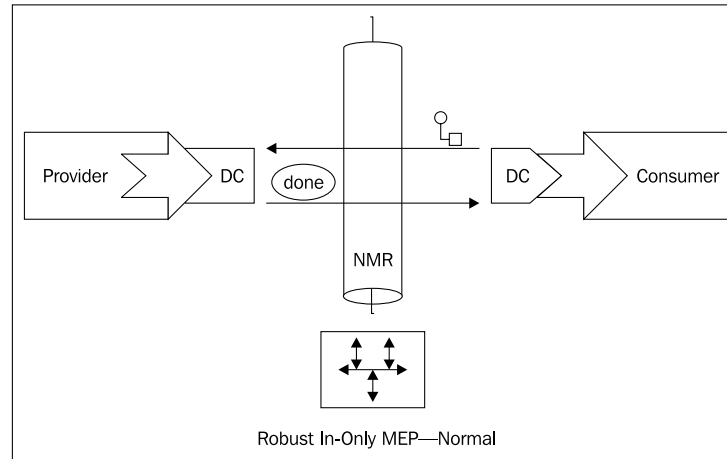
In the In-Only MEP normal scenario, since the Consumer issues a request to the Provider with no error (fault) path, any errors at the Provider-level will not be propagated to the Consumer.

Robust In-Only MEP

Robust In-Only MEP is used for reliable, one-way message exchanges. It has got two usage scenarios – the normal scenario and the fault scenario.

- **Normal scenario:** In the Robust In-Only MEP in the normal (without any fault) scenario, the sequence of message exchanges is similar to that of In-Only MEP. The difference comes to play only in the case where there is an error at the Provider-level, which will be described as the next item.

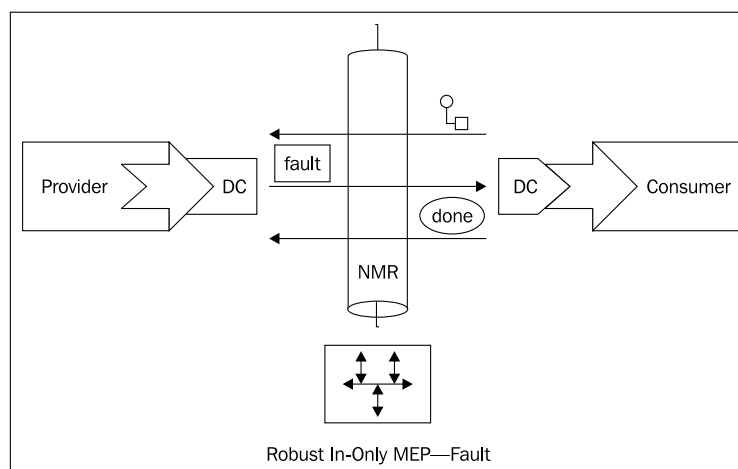
The following figure explains the Robust In-Only MEP – Normal scenario:



In the Robust In-Only MEP – Normal scenario, the sequence of operations is as follows:

- Service Consumer initiates with a message.
- Service Provider responds with the status to complete the message exchange.
- **Fault scenario:** In the Robust In-Only MEP in the fault scenario, the Consumer issues a request to the Provider and the Provider will respond with a fault instead of the normal response.

The following figure explains the Robust In-Only MEP – Fault scenario:



In the Robust In-Only MEP – Fault scenario, the sequence of operations is as follows:

- Service Consumer initiates with a message.
- Service Provider responds with a fault.
- Service Consumer responds with the status to complete the message exchange.

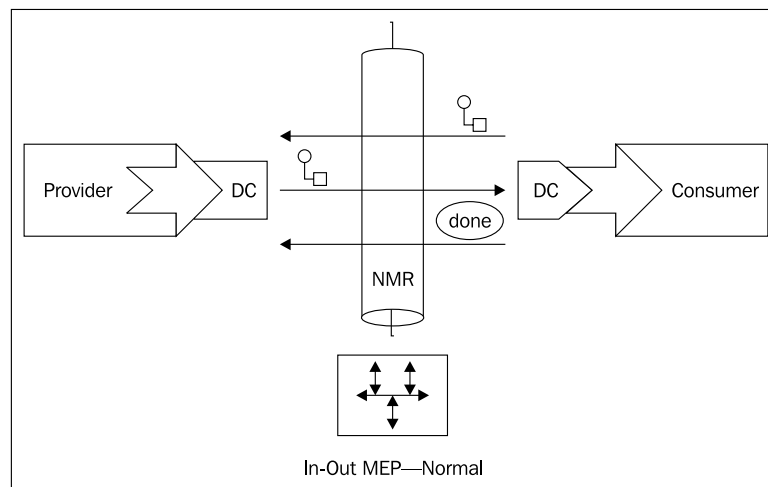
So, what you need to note is that, in the Robust In-Only MEP – Normal scenario, the exchange is terminated when the Provider responds with status to complete the message exchange, whereas in the Robust In-Only MEP – Fault scenario, the exchange is terminated when the Consumer responds with the status to complete the message exchange. The status to complete the message exchange even in the case of fault scenario, brings robustness to it.

In-Out MEP

In-Out MEP is used for a request-response pair of service invocations. Here the Consumer issues a request to the Provider, with expectation of a response. It has got two usage scenarios – the normal scenario and the fault scenario.

- **Normal scenario:** In the In-Out MEP in the normal (without any fault) scenario, the Consumer issues a request to the Provider, with expectation of a response. The Provider responds with the normal response.

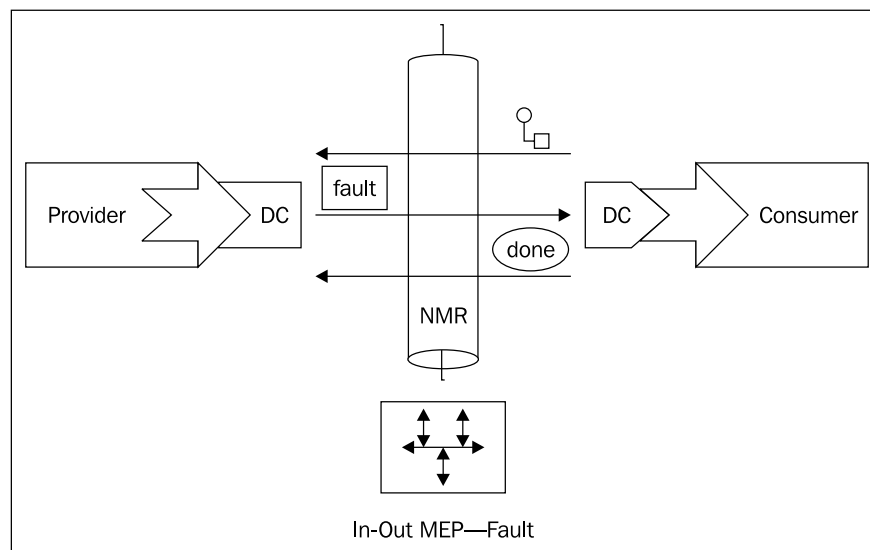
The following figure explains the In-Out MEP – Normal scenario:



In the In-Out MEP – Normal scenario, the sequence of operations is as follows:

- Service Consumer initiates with a message.
 - Service Provider responds with a message.
 - Service Consumer responds with the status to complete the message exchange.
- **Fault scenario:** In the In-Out MEP in the fault scenario, the Consumer issues a request to the Provider with expectation of a response and the Provider will respond with a fault instead of the normal response.

The following figure explains the In-Out MEP – Fault scenario:



In the In-Out MEP – Fault scenario, the sequence of operations is as follows:

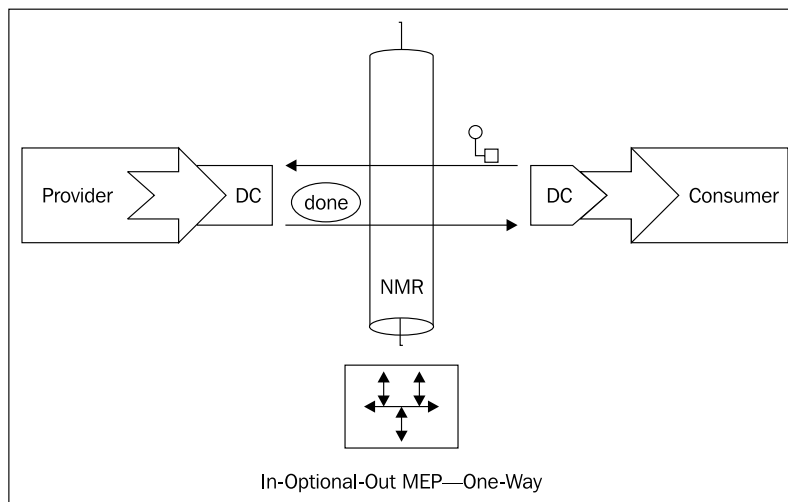
- Service Consumer initiates with a message.
- Service Provider responds with a fault.
- Service Consumer responds with the status to complete the message exchange.

In-Optional-Out MEP

In the **In-Optional-Out MEP**, the service Consumer issues a request to the service Provider, which may or may not result in a response. Both the Consumer and the Provider have the option of generating a fault in response to a message received during the interaction.

- **One-Way:** In the In-Optional-Out MEP – One-Way scenario, the service Consumer issues a request to the service Provider. The Provider neither returns any response, nor generates any fault.

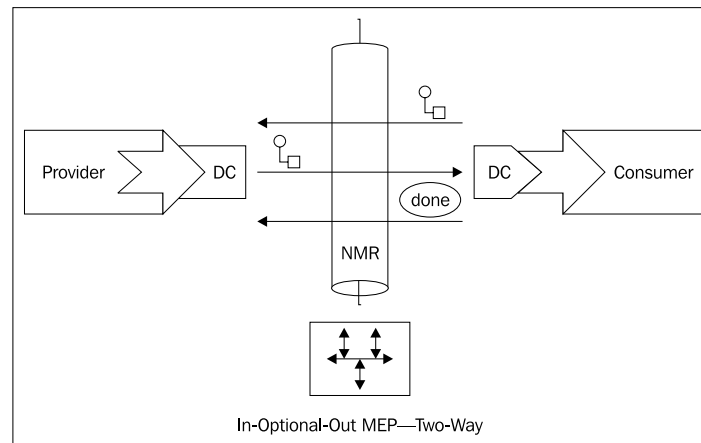
The following figure explains the In-Optional-Out MEP – One-Way scenario:



In the In-Optional-Out MEP – One-Way scenario, the sequence of operations is as follows:

- Service Consumer initiates with a message.
- Service Provider responds with the status to complete the message exchange.
- **Two-Way:** In the In-Optional-Out MEP – Two-Way scenario, the service Consumer issues a request to the service Provider. The Provider then returns a response.

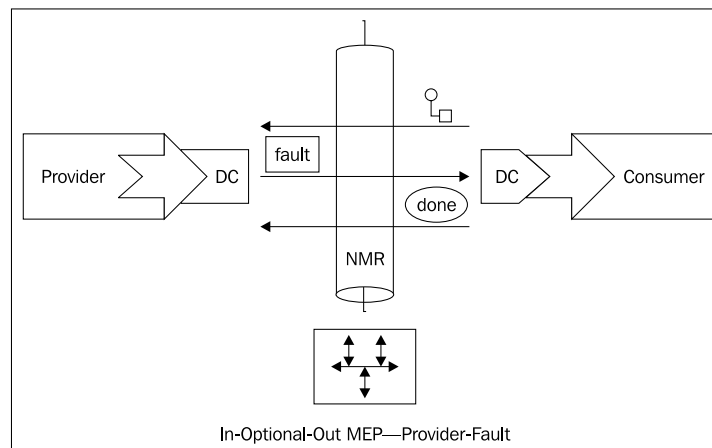
The following figure explains the In-Optional-Out MEP – Two-Way scenario:



In the In-Optional-Out MEP – Two-Way scenario, the sequence of operations is as follows:

- Service Consumer initiates with a message.
 - Service Provider responds with a message.
 - Service Consumer responds with the status to complete the message exchange.
- **Provider-Fault:** In the In-Optional-Out MEP – Provider-Fault scenario, the service Consumer issues a request to the service Provider. The Provider generates a fault instead of the normal response.

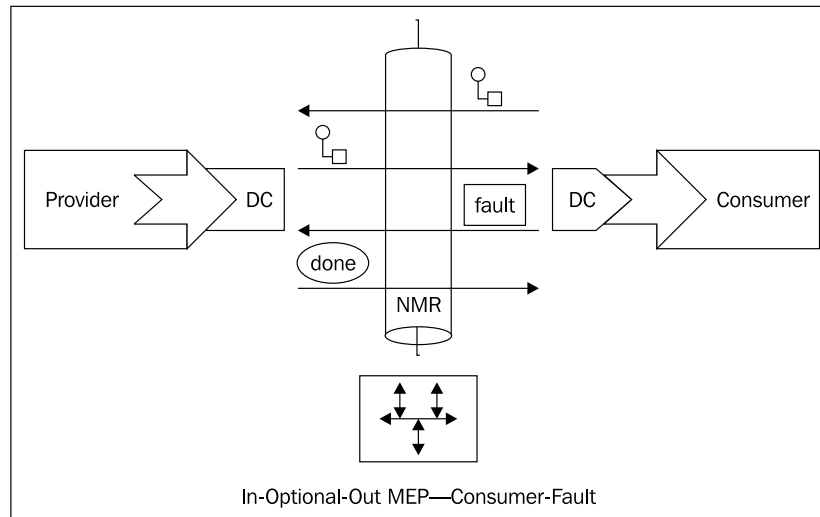
The following figure explains the In-Optional-Out MEP – Provider-Fault scenario:



In the In-Optional-Out MEP – Provider-Fault scenario, the sequence of operations is as follows:

- Service Consumer initiates with a message.
 - Service Provider responds with a fault.
 - Service Consumer responds with the status to complete the message exchange.
- **Consumer-Fault:** In the In-Optional-Out MEP – Consumer-Fault scenario, the service Consumer issues a request to the service Provider. The Provider then returns a response. The Consumer generates a fault while accepting the response.

The following figure explains the In-Optional-Out MEP – Consumer-Fault scenario:



In the In-Optional-Out MEP – Consumer-Fault scenario, the sequence of operations is as follows:

- Service Consumer initiates with a message.
- Service Provider responds with a message.
- Service Consumer responds with a fault.
- Service Provider responds with the status to complete the message exchange.

When considering a MEP, we always consider the service provider's point of view. Thus, a message targeted towards the provider in an In-Only MEP is the 'In' part of the MEP. On the contrary, if a message is targeted towards the consumer, it is in fact targeted out from the provider, and hence is the 'Out' part of the MEP.

Thus, depending upon the role of the component in the message exchange, the appropriate part or message is created, initialized, and sent to the delivery channel. For an In-Out scenario, the typical steps are as follows:

```
javax.jbi.messaging.InOut inout = createInOutExchange
    (new QName(addressNamespaceURI, addressLocalPart), null, null);
inout.setProperty("correlationId", id);
// set other properties
javax.jbi.messaging.NormalizedMessage nMsg = inout.createMessage();
// nMsg.setProperty(Constants.PROPERTY_SSN_NUMBER, ssnNumber);
// set other properties
inout.setInMessage(nMsg);
send(inout);
```

ESB—Will it Solve all Our Pain Points

In Chapter 1, we introduced ESB and also looked into what JBI has got to offer here. If you are familiar with SOA principles, one subtle fact, which is evident now is that ESB or JBI are not an end by themselves, but a means towards an end (which is SOA). An ESB is not required to build an SOA, nor is JBI required for ESB or SOA. However, all of them have something in common using JBI—we can build standard components to be deployed into ESB architectures. Thus, JBI by itself is one of the ways by which we can attain SOA. There is also a caveat to this—just following JBI or ESB will not guarantee that you attain SOA. Increasingly, you will hear requests from your project stakeholders to implement an ESB without considering SOA as a whole, such that they want immediate solutions. It is technically feasible to build ESB, which act as pipes interconnecting systems, but the success of such ESB architectures without considering the SOA landscape, which it is supposed to be a part of, will be difficult to measure.

Summary

JBIs the new integration API introduced in the J2EE world. It is a great enabler for SOA because it defines ESB architecture, which can facilitate the collaboration between services. It provides for loosely coupled integration by separating out the providers and consumers to mediate through the bus.

The NMR provides a common integration channel through which the messages flow. Services are published in the bus using the WSDL standard. Providers and consumers are the different roles taken by the integration components with respect to the bus, when plugged into the JBI bus. Message exchange takes place through different MEPs, each providing different levels of reliability.

The next chapter will introduce a JBI container. Be ready to wet your hands with some code too – to build and deploy your first JBI sample.

3

JBIs Container—ServiceMix

The first two chapters introduced ESB and JBI respectively, with much theory and less of code. Just like me, you too might have come across a lot of white papers and point of views on the above two technologies. However, when we want to actually implement them into our technical architectures, we need working code demonstrating both.

Just like any other Java APIs, servlet or EJB, for a JBI API too, we need concrete implementation. There are many implementations supporting ESB architectures available in the market today. A couple of them can interoperate with the JBI API whereas a few others tackle JBI on the side of quite a different programming model. ServiceMix is an open-source ESB platform in Java programming language, built from the ground up with JBI APIs and principles.

Due to the reason quoted in the first paragraph, we will use ServiceMix to better understand JBI and ESB. Hence, most of the subsequent chapters in this book will introduce new scenarios or patterns in business integration and then try to solve them in the JBI way with code samples in ServiceMix. What you need to keep in mind is that each of these integration techniques can be used as standalone, or in integration with other scenarios and patterns to bring out an integration solution using ESB architectural blueprints.

We will cover the following in this chapter:

- Introduction to ServiceMix – the JBI container
- Few other ESB products available in the market
- Where to download and how to install ServiceMix
- First JBI sample

ServiceMix—Under the Hood

ServiceMix is based on SOA and Event Driven Architecture (EDA), and hence provides a platform for SOI. Let us look more into this JBI container.

Salient Features

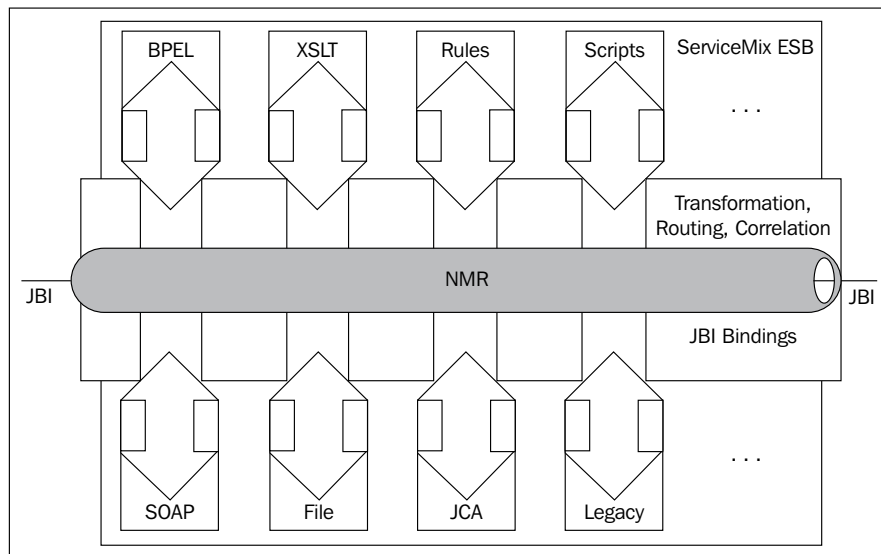
ServiceMix is built based on the JBI (JSR 208) specification and hence components are portable across ESB containers. ServiceMix is lightweight and can be run standalone or embedded in other containers. Since it is JBI-compliant, ServiceMix can itself be plugged into another JBI-compliant ESB. ServiceMix has integrated Spring support, and hence we can integrate and wire services and components in Spring-like configuration files. The JBI standard speaks about standard deployment unit formats in the form of service unit and service assembly. This means we can hot deploy any JBI-compliant BPEL engine (or set of BPEL files into a BPEL engine), rule engine, transformation engine, routing engine (even though ServiceMix does its own routing), scripting engine, or other integration component (such as specific JBI binding components) directly into ServiceMix.

ServiceMix Architecture

JB1 is a specification and vendors are free to implement the JBI container in their own ways, just like there are multiple EJB containers brought out by multiple vendors. Each vendor's container has to respect the JBI APIs if it is to be J2EE-compliant. ServiceMix is a JBI container and hence is compliant with the JBI APIs, but has got its own features and way of routing messages. Let us look more into the internal architecture of ServiceMix.

Architecture Diagram

JB1 compliancy is the main feature of the ServiceMix architecture. This means, as shown in the following architecture diagram, ServiceMix is an open architecture. So, any JBI-compliant component can fit into the architecture. Moreover, the architecture is also scalable from a deployment perspective. Most hub-and-spoke architecture suffers from the single point of failure. However, ServiceMix, due to the open nature, can also collaborate with any other ESB, effectively plugging its routing capabilities into other ESB cores, thus generating a federation of service containers.



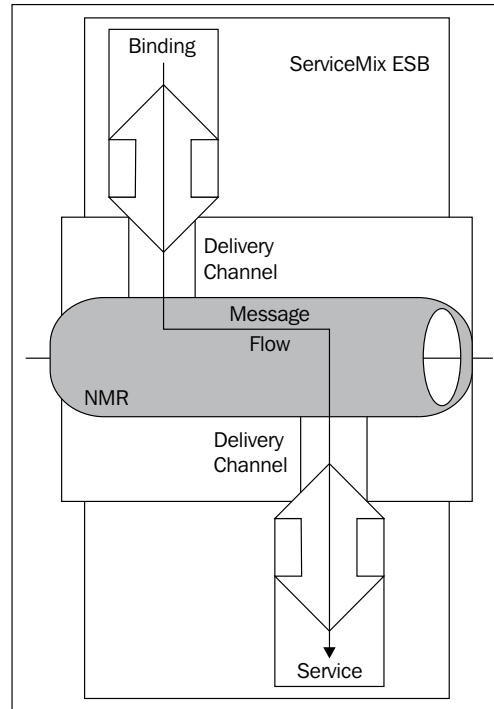
Normalized Message Router Flows

Components plugged into ServiceMix, whether they are local or remote to it, interact with each other through the NMR. Moreover, ServiceMix is based on MOM principles. Hence, messages are exchanged between components through the NMR using a suitable MEP. To exchange messages, different message dispatch policies can be adopted which will decide the QOS of the message exchanges. Each of these policies is abstracted out as different NMR flows in ServiceMix. Depending upon the specific use case you want to implement using ServiceMix, the actual NMR flow can be specified. It is also to be noted that different NMR flows will exhibit different capabilities in terms of cross cutting concerns such as message brokering and message buffering.

ServiceMix, at present, provides four NMR flow types:

- **Straight through (ST) flow:** STP is analogous to the B2B trading process for capital markets and payment transactions done electronically. This is done without needing to rekey or manually intervene the STP flow. A message exchange is routed ST from the source to the destination. There is no staging or buffering of messages en route. This kind of flow is preferred for cases where the ServiceMix container is deployed with simple flows (no state) or is embedded. Since there is no staging or buffering, latency will be as low as possible.

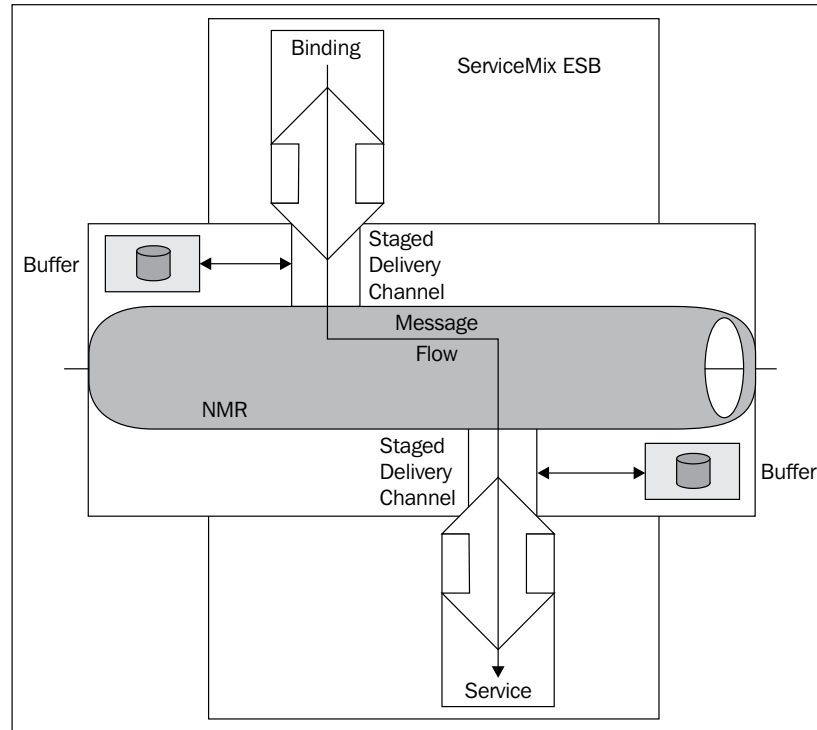
The following figure represents a ST flow:



- **Staged Event Driven Architecture (SEDA) flow:** SEDA decomposes a service into multiple stages, where each stage is an event-driven service component that performs some aspect of processing the request. Each stage contains a small, dynamically sized thread pool to drive its execution. SEDA provides nonblocking I/O primitives to eliminate the most common sources of long blocking operations.

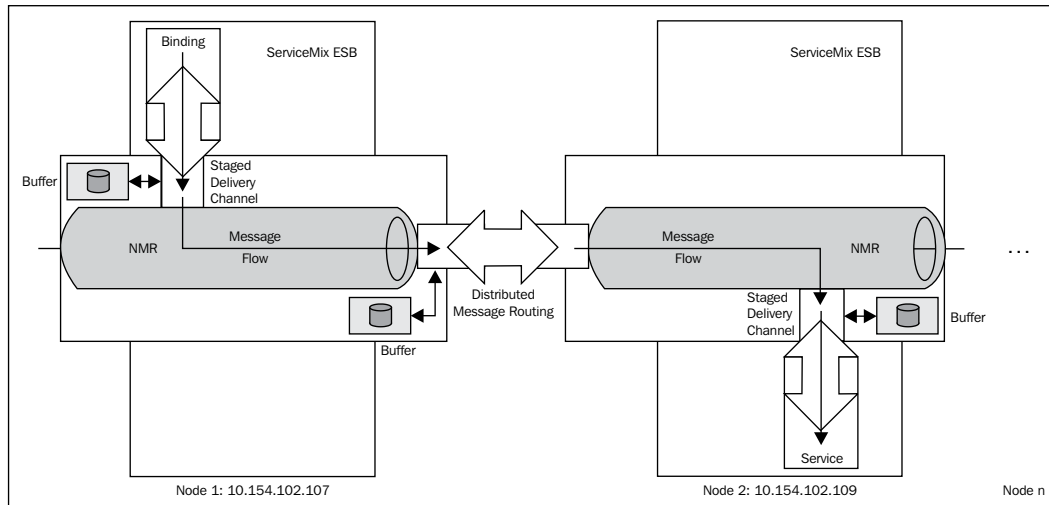
In the ServiceMix SEDA flow, we have a simple event staging between the internal processes in the NMR broker. SEDA is the default flow in ServiceMix and is suited for general deployment, as the additional staging can buffer exchanges between heavily routed to components (where state maybe used), for example.

The following figure represents a SEDA flow:



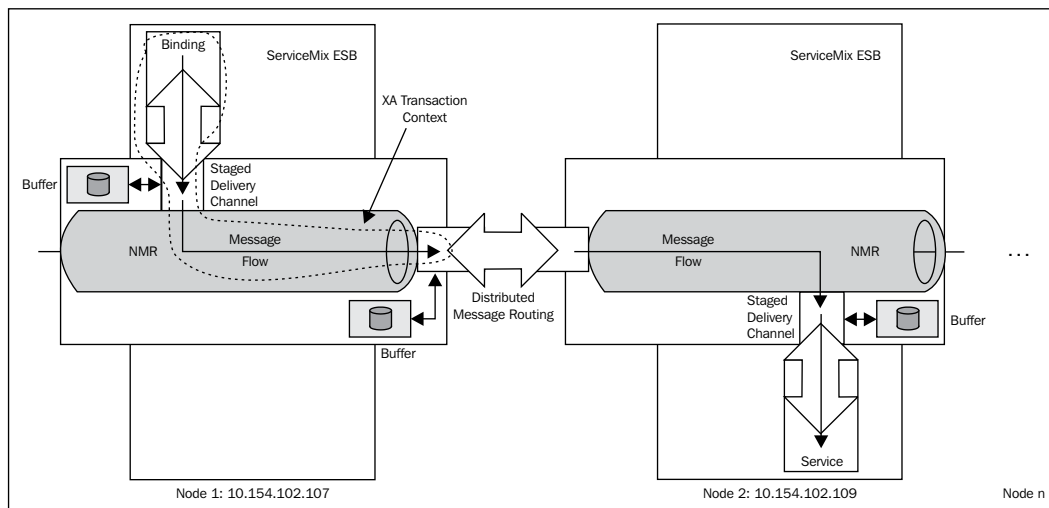
- Java Message Service (JMS) flow:** In the ServiceMix JMS flow, we can leverage the tested and proven methodology of MOM to address scalability or failover. Using JMS flow, multiple ServiceMix containers can collaborate in a cluster or otherwise, to provide component and service replication. When we deploy a component either as a POJO or as an archive component into a JMS flow configured ServiceMix container, all the containers in the cluster are notified of the deployment. The JMS NMR flow can handle automatic routing (and failover) of message exchange(s) between multiple container instances.

The following figure represents a JMS flow:



- JCA Flow:** The ServiceMix JCA NMR flow is very much similar to the JMS flow. In fact, ServiceMix uses JMS sessions as the underlying mechanism so that multiple ServiceMix containers can collaborate in a cluster. In addition, JCA provides support for **XA transactions** when sending and receiving JB1 exchanges.

The following figure represents a JCA flow:



Other ESBs

There are quite a few ESB frameworks available in the industry today and we will list a few of them here for completeness of our discussion.

Mule

Mule is defined as a lightweight messaging framework functioning as an ESB. This ESB features a distributed object broker, which can handle interactions between different systems, applications, components, and services, irrespective of the transport protocols and binding technologies. Mule provides a Universal Message Object (UMO) API (inside `org.mule.umo` package), a way for components to interact without needing to know about the protocol or delivery mechanisms of information passed between them.

Mule can host standard POJOs, which can be managed from containers such as Spring, Pico, and Plexus or from the classpath, or any other source. Mule has a JBI interface which will compliment and not compete with ServiceMix. Mule and JBI functions differently in the way they exchange message formats.

JBI is XML and WSDL centric whereas Mule makes no assumptions about the message type, so that you can easily use Strings, binaries, MIME, XML, objects, streams, or a mixture without any extra development. To stream all Mule supported formats through a JBI-compliant container, suitable tunneling may need to be done or binary like messages need to be transformed into XML format. Thus, JBI brings WSDL centric standardization, whereas Mule provides more options in terms of flexibility.

Celtix

Celtix is an ESB which is JBI-compliant. Leveraging Celtix, developers can build service engines and binding components. Since Celtix is JBI-compliant, the service engines and binding components too are JBI-compliant, hence can be deployed into another third-party JBI container. Similarly, Celtix is also a full fledged JBI-compliant container. This means any third-party JBI-compliant service engines and binding components can be deployed into a Celtix ESB container.

Even though Celtix is JBI-compliant, JBI objectives are slightly different from that of Celtix. The main differences are listed as follows:

- A JBI-compliant service engine or binding component is essentially a black box and there is little or no control over the message flow within them.

- The normalized message format specified by JBI is XML and hence exchange of non XML-based message formats, like that of CORBA, will be difficult and at the very least will require the binary formatted messages to be transformed into an XML format.
- For binding components and service engines in the role of provider, WSDL is the contract in JBI. WSDL may not be appropriate in all scenarios. For example, in the case of legacy integration involving proprietary API calls.
- Object references and callbacks are not specified in JBI.

Celtix, being JBI-compliant, is also trying to bridge the above mentioned gaps, which are yet to be proven as gaps by industry based on real integration scenarios.

Iona Artix

Artix ESB service enables integration existing servers directly at the endpoints without the need for a separate integration server, thus facilitating macro-level integration. Artix is a platform independent infrastructure product for building Java, C/C++, and mainframe web services. All features can be configured dynamically providing maximum flexibility for implementing a scalable, fully distributed SOA. Artix ESB provided tools include Artix Designer, Code generators, and Management Console to make creating, testing, and deploying services easy. The Artix ESB container is a server that allows you to run service plug-ins, either those that come with Artix ESB or custom components, and provides threading, resource management, and network management services. However, Artix technical specification doesn't speak about JBI, nor do they claim that Artix components interoperate with other JBI containers.

PEtALS

PEtALS is an open-source JBI platform. PEtALS is based on JSR-208 and provides lightweight and packaged integration solutions, by providing a solid backbone for your enterprise information system. PEtALS is based on ESB for data exchange. PEtALS has a strong focus on distribution and clustering by basing its message exchange middleware on JORAM (an open-source JMS implementation).

ChainBuilder

ChainBuilder is JBI-compliant with the advantage that it hides the complexities of JBI behind a GUI. These GUI capabilities and configuration editors enables the point-and-click mapping of non-XML formatted messages such as variable, fixed, and X12 EDI formats. The GUI has an Eclipse-based plug-in, which will enable drag-and-drop functionality for ESB components through wizards.

Installing ServiceMix

I agree that we need to start doing, rather than reading. This section is intended to help the reader get started with ServiceMix. We will try out few examples without looking much into the rationale behind doing so, since the same has been covered through other chapters. Moreover, a single "Getting Started" section alone is not sufficient to help a reader new to ESB to appreciate all aspects of it; hence, we will not attempt that in this chapter alone.

There are multiple ways to get started but obviously developers like us have to code something, build, deploy, and get that running without too much hassle. That is exactly what we will do in this section. Let's do that now.

Hardware Requirements

As of ServiceMix 3.0.x, we need the following minimum hardware:

- 31 MB of free disk space for the ServiceMix 3.0.x binary distribution

OS Requirements

The following flavors of ServiceMix installs are available:

- **Windows:** Windows XP SP2, Windows 2000
- **Unix:** Ubuntu Linux, Powerdog Linux, MacOS, AIX, HP-UX, Solaris, or any Unix platform that supports Java

Run-time Environment

ServiceMix require the following run-time environment settings:

- Java Developer Kit (JDK) 1.6.x (<http://java.sun.com/javase/downloads/index.jsp>).
- The `JAVA_HOME` environment variable must be set to the parent directory where the JDK is installed.

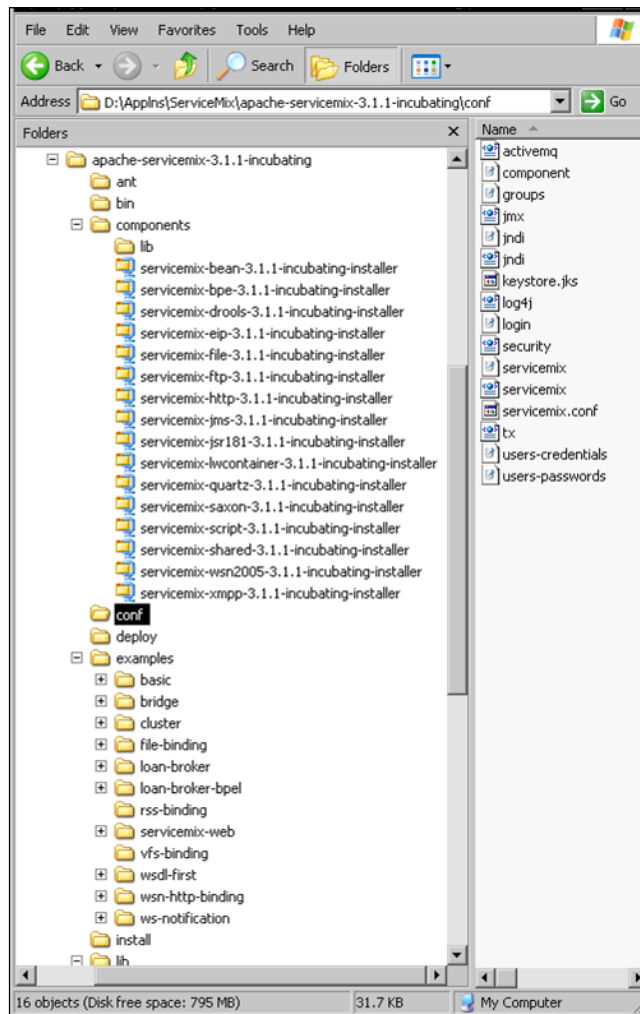
For example, `C:\Yourfilepath\jdk.1.6.0_04`.


- Apache Ant and/or Maven.

Installing ServiceMix in Windows

The steps for installing ServiceMix in Windows are listed as follows:

- Click the **ServiceMix 3.1.1 Release** link under the **Latest Releases** section of the download page available at <http://incubator.apache.org/servicemix/download.html>.
- Download the binary distribution of your choice. For example, `apache-servicemix-3.1.x.zip`; `apache-servicemix-3.1.1-incubating` is the preferred release.
- Extract this ZIP file into a directory of your choice. It is advised not to have illegal characters or blank spaces in the installation path directories.



[ As of this writing, apache-servicemix-3.1.1-incubating is the stable release. You can download this version of the ServiceMix binary from: <http://incubator.apache.org/servicemix/servicemix-311.html>]

Installing ServiceMix in Unix

To install ServiceMix in Unix, follow similar procedures as for Windows, except download the binary distribution with file name similar to `apache-servicemix-x.x.x.tar.gz`.

Configuring ServiceMix

In the ServiceMix installation home directory, there is a sub-directory named `conf`. This hosts all major configuration files for ServiceMix. `servicemix.properties` here contains the port specifications, especially the `rmi.port`. You can change something here, if the default ports are engaged or for some other reason.

Starting ServiceMix

To start ServiceMix, go to `SERVICEMIX_HOME\bin` (where `SERVICEMIX_HOME` is the parent directory where you have extracted the binary ZIP) and then execute `servicemix.bat`. Now, ServiceMix is running with a basic configuration, but no components.

While starting ServiceMix, working directories get created relative to the current directory from where you are starting ServiceMix.

Stopping ServiceMix

For both Windows and Unix installations, you can terminate ServiceMix by typing `CTRL-C` in the command shell or console in which ServiceMix is started.

Resolving classpath Issues

When we expand the binary ZIP folder of ServiceMix installation, all the required libraries are not extracted and included in the correct folder. In ServiceMix, the required Java libraries should be placed in `SERVICEMIX_HOME\lib` or `SERVICEMIX_HOME\lib\optional` folder. So any missing libraries can be added to these two folders. Some libraries are already available in `SERVICEMIX_HOME\components` or `SERVICEMIX_HOME\components\lib` folder, and we can extract the libraries from these places to `SERVICEMIX_HOME\lib` or `SERVICEMIX_HOME\lib\optional` folders as and when required.

The following are the two most common errors that happen when starting ServiceMix and the required libraries are not found in the correct place:

- Caused by: `java.lang.ClassNotFoundException:...`
- Caused by: `org.springframework.beans.factory.BeanDefinitionStoreException: Unrecognized xbean namespace mapping:...`

It is very important that when either of the above errors or some similar "**libraries not found**" error happens, you either follow what we have already explained or download any dependent library (.jar files) from third-party websites, and place them in the `SERVICEMIX_HOME\lib` or `SERVICEMIX_HOME\lib\optional` folder.

ServiceMix Components—a Synopsis

As is shown in the ServiceMix architecture diagram, ServiceMix ships standard JBI components and lightweight JBI components.

Standard JBI Components

ServiceMix standard JBI components are fully JBI-compliant, and support the JBI packaging and deployment model. They are placed in the folder `%SERVICEMIX_HOME%\components`, and the major components are listed as follows:

- `servicemix-bean`: For mapping POJO beans to JBI exchanges.
- `servicemix-bpe`: BPEL engine based on a Sybase donation to the Apache ODE project.
- `servicemix-camel`: Camel provides a full set of Enterprise Integration Patterns both from Java code and Spring XML.
- `servicemix-drools`: Provides integration with Drools rules engine.
- `servicemix-eip`: Provides Enterprise Integration Patterns.
- `servicemix-file`: This binding component provides integration with the File system.
- `servicemix-ftp`: An FTP binding component.
- `servicemix-http`: A HTTP binding component.
- `servicemix-jms`: Provides integration with messaging middleware through JMS.
- `servicemix-jsr181`: Service Engine to expose annotated POJOs as services.
- `servicemix-lwcontainer`: This component is a container to which we can deploy lightweight components.

- `servicemix-quartz`: Service Engine used to schedule and trigger jobs using the Quartz scheduler.
- `servicemix-saxon`: Service Engine for integrating XSLT / XQuery engines.
- `servicemix-script`: Helps to integrate scripting engines with JBI.
- `servicemix-wsn2005`: Service Engine implementing Oasis WS-Notification specification.
- `servicemix-xmpp`: Helps to communicate with XMPP (Jabber) servers through the JBI bus.

Lightweight JBI Components

ServiceMix lightweight components are not packaged and deployed as per JBI specification. This is because the components used here are lightweight components that activate a single JBI endpoint and they do not support JBI deployments. Instead, they can be configured using the Spring configuration.

Listed as follows are the major lightweight components:

- **Cache**: Cache component can cache service invocations to avoid unnecessary load.
- **Component helper classes**: These components make it easy to write new JBI components.
- **Drools**: Drools can be used to do rules-based routing.
- **Email**: Provides support for MIME email sending.
- **File**: It can write messages to files in a directory, or poll files, or directories to send messages to JBI.
- **FTP**: Provides integration to FTP via the Jakarta Commons Net library.
- **Groovy**: Helps to use Groovy scripts as endpoints, transformers, or services.
- **HTTP**: Helps to invoke requests on remote HTTP servers and to expose JBI components over HTTP.
- **Jabber**: Can integrate with Jabber network via the XMPP protocol.
- **JAX WS**: Uses the Java API for XML-based web services to invoke a web service or to host a Java-based web service and expose it over multiple protocols.
- **JCA**: Provides a very efficient way of thread pooling, transaction handling, and consumption on JMS and other resource adapters.
- **JMS**: Helps to send and receive JMS messages.

- **Quartz:** Used to schedule and trigger jobs using the Quartz scheduler.
- **Reflection:** For In-Only and In-Out JBI components, we can create dynamic proxies, which when invoked dispatch the messages into the JBI container.
- **RSS:** Supports RSS and Atom via the Rome library.
- **SAAJ:** Provides integration with SOAP with attachments for Java (SAAJ) and Apache Axis.
- **Scripting:** Helps to script In-Only or In-Out message exchanges using a JSR 223 compliant scripting engine such as JavaScript, Jython, or Groovy.
- **Validation:** Used to validate document schema using JAXP 1.3 and XMLSchema or RelaxNG.
- **VFS:** Provides integration with file systems, jars/zips/bzip2, temporary files, Samba (CIFS), WebDAV, HTTP, HTTPS, FTP, and SFTP.
- **WSIF:** Provides a way to call web services, hiding the details of how the service is provided.
- **XFire:** Provides integration with XFire SOAP stack.
- **XSLT:** Can do XSLT transformation for one normalized message to another.
- **XSQL:** Use Oracle tool for turning SQL queries into XML and for taking XML and inserting or updating into a database.

The above lists are not exhaustive, and the number of components added to the list is increasing day by day. Readers are advised to refer to the ServiceMix website for updated information. We will look at a few amongst the above standard and lightweight JBI components in samples or otherwise, as we walk through different chapters in this text.

Your First JBI Sample—Binding an External HTTP Service

I hope all readers, whether novice or professional, can understand and appreciate what we mean by saying a HTTP service, otherwise most probably they will not be reading a text book like this. Hence, I will choose a HTTP service for demonstration purposes here.

Servlet-based HTTP Service

A HTTP service is a network service accessible through the HTTP protocol. Servlets are the simplest components available in Java with which we can build HTTP service, and hence we will use that here. Assuming that the reader is familiar with servlets and how to deploy a HTTP service in their favorite web container such as Tomcat, only the major steps are listed here. For further details the reader is encouraged to refer to any available servlet technology books.

As the first step, we will build and deploy a very simple servlet. The servlet code is given as follows:

```
public class WelcomeServlet extends HttpServlet
{
    public static final String XML_CONTENT =
        "<?xml version='1.0'><Name>Binil's Servlet wishes you</Name>";
    public void init(ServletConfig config) throws ServletException
    {
        super.init (config);
    }
    public void doGet (HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException
    {
        doPost (request, response);
    }
    public void doPost (HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException
    {
        System.out.println("WelcomeServlet.doPost...");
        response.setContentType("text/xml");
        response.setContentLength(XML_CONTENT.length());
        PrintWriter out = response.getWriter();
        out.println (XML_CONTENT);
        out.flush();
    }
}
```

As shown in the code, the servlet simply spits out some XML content to the response stream, without even looking at the contents of the request. Equally simple is the `web.xml` file and it is also shown here:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app>
  <display-name>WAP Examples</display-name>
  <description>WAP Examples.</description>
```

```
<servlet>
  <servlet-name>WelcomeServlet</servlet-name>
  <servlet-class>
    com.binildas.esb.servicemix.servlet.WelcomeServlet
  </servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>WelcomeServlet</servlet-name>
  <url-pattern>/hello/*</url-pattern>
</servlet-mapping>
</web-app>
```

As a first step, if you haven't done it before, edit `examples.PROPERTIES` (provided along with the code download for this chapter), and change the paths there to match your development environment.

Now to build the web component, change the directory to `ch03\Servlet` and execute ant script.

```
cd ch03\Servlet
ant
```

Here, we assume that you have installed the latest version of Apache Ant build tool and the `bin` folder of that is in your path environment variable. This will generate the web archive (`EsbServlet.war`) and place it in the `dist` folder inside `ch03\Servlet` which can be deployed in the `webapps` folder of Tomcat (or any other relevant web server) and restart the server.

The HTTP service would have been exposed by now and the same can be accessed using your favorite browser with the following URL:

```
http://localhost:8080/EsbServlet/hello/
```

We can now write `Client` code similar to what is shown in the following code to test the HTTP service:

```
public class HttpInOutClient
{
  private static String url = "http://localhost:8080/
                              EsbServlet/hello/";
  private static String fileUrl = "HttpSoapRequest.xml";
  protected void executeClient()throws Exception
  {
    InputStream inputStream =
      ClassLoader.getResourceAsStream(fileUrl);
    byte[] bytes = new byte[inputStream.available()];
    inputStream.read(bytes);
  }
}
```

```

        inputStream.close();
        URLConnection connection = new URL(url).openConnection();
        connection.setDoOutput(true);
        OutputStream os = connection.getOutputStream();
        os.write(new String(bytes).getBytes());
        os.close();
        BufferedReader in =
            new BufferedReader(new InputStreamReader(connection.
                                                    getInputStream()));

        String inputLine;
        while ((inputLine = in.readLine()) != null)
        {
            System.out.println(inputLine);
        }
        in.close();
    }
    public static void main(String[] args) throws Exception
    {
        if (args.length == 2)
        {
            url = args[0];
            fileUrl = args[1];
        }
        HttpInOutClient httpInOutClient = new HttpInOutClient();
        httpInOutClient.executeClient();
    }
}

```

Note that we are testing the HTTP service by sending arbitrary XML content as a request, even though, any character stream will be acceptable as request. This is because, later we will bind this service to the ServiceMix ESB and during that time messages have to be routed through the NMR. For this, XML is the valid normalized message.

To run the above client, make sure (edit, if required) the run target in the `ch03\Servlet\build.xml` file matches the following code:

```

<target name="run">
    <java classname="HttpInOutClient" fork="yes" failonerror="true">
        <classpath refid="classpath"/>
        <arg value=" http://localhost:8080/EsbServlet/hello/" />
        <arg value="HttpSoapRequest.xml" />
    </java>
</target>

```


Now, executing `ant run` will send a message to the HTTP service:

```
cd ch03\Servlet
ant run
```

Configure the HTTP Service in ServiceMix

Before I start describing the binding of services to ServiceMix, I would like to put out one word of caution—the binding components used in this chapter are ServiceMix lightweight components. They are deprecated as of now, and hence for any production configurations we have to use standard JBI components, which will be covered in detail in subsequent chapters. Still I am using these components because they are simple, straightforward, intuitive enough, and easy to configure for a novice user.

All the ServiceMix specific bindings are done in `ch03\HttpBinding\servicemix.xml` and let us look straight into that in the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xbean.org/schemas/spring/1.0"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:sm="http://servicemix.apache.org/config/1.0"
       xmlns:lb="http://servicemix.apache.org/demos/gettingstarted">
  <bean id="propertyConfigurer"
        class="org.springframework.beans.factory.
              config.PropertyPlaceholderConfigurer">
    <property name="location"
              value="classpath:servicemix.properties" />
  </bean>
  <import resource="classpath:activemq.xml" />
  <import resource="classpath:jmx.xml" />
  <import resource="classpath:jndi.xml" />
  <import resource="classpath:security.xml" />
  <import resource="classpath:tx.xml" />
  <sm:container name="jbi"
                monitorInstallationDirectory="false"
                createMBeanServer="true"
                useMBeanServer="true">
    <sm:activationSpecs>
      <sm:activationSpec componentName="trace" service="lb:trace">
        <sm:component>
          <bean xmlns="http://xbean.org/schemas/spring/1.0"
                class="org.apache.servicemix.components.util.
                              TraceComponent" />
        </sm:component>
      </sm:activationSpec>
    </sm:activationSpecs>
  </sm:container>
</beans>
```

```

</sm:activationSpec >
<sm:activationSpec componentName="timer"
    service="lb:timer"
    destinationService="lb:httpGetData">
  <sm:component>
    <bean class="org.apache.servicemix.components.quartz.
        QuartzComponent">
      <property name="triggers">
        <map>
          <entry>
            <key>
              <bean class="org.quartz.SimpleTrigger">
                <property name="repeatInterval"
                    value="5000" />
                <property name="repeatCount"
                    value="0" />
              </bean>
            </key>
            <bean class="org.quartz.JobDetail">
              <property name="name"
                  value="My Example Job" />
              <property name="group"
                  value="ServiceMix" />
            </bean>
          </entry>
        </map>
      </property>
    </bean>
  </sm:component>
</sm:activationSpec>
<sm:activationSpec componentName="httpGetData"
    service="lb:httpGetData"
    destinationService="lb:trace">
  <sm:component>
    <bean xmlns="http://xbean.org/schemas/spring/1.0"
        class="org.apache.servicemix.
        components.http.HttpInvoker">
      <property name="url"
          value="http://localhost:8080/EsbServlet/
              hello/" />
    </bean>
  </sm:component>
</sm:activationSpec>
<sm:activationSpec componentName="httpReceiver"

```

```
        service="lb:httpReceiver"
        endpoint="httpReceiver"
        destinationService="lb:httpGetData">
    <sm:component>
        <bean class="org.apache.servicemix.
            components.http.HttpConnector">
            <property name="host" value="127.0.0.1"/>
            <property name="port" value="8912"/>
        </bean>
    </sm:component>
</sm:activationSpec>
</sm:activationSpecs>
</sm:container>
</beans>
```

Let us not look at the complexities of the above configuration until we actually run the service. So we will defer discussion on the above code until we see the code in action.

Run ServiceMix Basic JBI Container

Before we bring up ServiceMix, your web server should be up and running with the web application generated in the previous section deployed successfully.

We now need to prepare ServiceMix with a few extra libraries. For that, do the following:

- Copy %SERVICEMIX_HOME%\components\lib\servicemix-components-3.1.1-incubating.jar to %SERVICEMIX_HOME%\lib\optional.
- Open %SERVICEMIX_HOME%\components\servicemix-http-3.1.1-incubating-installer.zip and copy the following .jars to %SERVICEMIX_HOME%\lib\optional:
 - jetty*.jar
 - commons-codec*.jar
 - commons-httpclient*.jar
- Copy quartz.jar from the Quartz distribution (<http://www.opensymphony.com/quartz/>) to %SERVICEMIX_HOME%\lib\optional.

Now, to get ServiceMix up with components configured in the XML configuration shown above, change directory to ch03\HttpBinding\ and execute %SERVICEMIX_HOME%\bin\servicemix servicemix.xml.

What you see in the following screenshot is the contents of my ServiceMix console:

```

D:\binil\com\java\com\binildas\esb\ServiceMix\Text\ESBUsingJBI-ServiceMix\ch03\H
ttpBinding>D:\Applns\ServiceMix\apache-servicemix-3.1.1-incubating\bin\servicem
i
x servicemix.xml
Starting Apache ServiceMix ESB: 3.1.1-incubating

Loading Apache ServiceMix from file: servicemix.xml
INFO - ConnectorServerFactoryBean - JMX connector available at: service:jmx
:rmi:///jndi/rmi://localhost:1099/jmxrmi
INFO - JBIContainer - ServiceMix 3.1.1-incubating JBI Contain
er <jbi> is starting
INFO - JBIContainer - For help or more informations please se
e: http://incubator.apache.org/servicemix/
WARN - ManagementContext - Failed to start rmi registry: internal
error: ObjID already in use
WARN - ManagementContext - Failed to start jmx connector: connecto
r:name=rmi
INFO - ComponentMBeanImpl - Initializing component: #SubscriptionMa
nager#
INFO - DeploymentService - Restoring service assemblies
INFO - ComponentMBeanImpl - Initializing component: trace
INFO - ComponentMBeanImpl - Initializing component: timer
INFO - SimpleThreadPool - Job execution threads will use class lo
ader of thread: main
INFO - RAMJobStore - RAMJobStore initialized.
INFO - StdSchedulerFactory - Quartz scheduler 'DefaultQuartzSchedule
r' initialized from default resource file in Quartz package: 'quartz.properties'

INFO - StdSchedulerFactory - Quartz scheduler version: 1.4.5
INFO - ComponentMBeanImpl - Initializing component: httpGetData
INFO - ComponentMBeanImpl - Initializing component: httpReceiver
2008-01-18 16:09:22.449::INFO: Logging to STDERR via org.morthay.log.StdErrLog
INFO - QuartzScheduler - Scheduler DefaultQuartzScheduler_$_NON_
CLUSTERED started.
2008-01-18 16:09:22.574::INFO: jetty-6.0.1
2008-01-18 16:09:22.683::INFO: Started SocketConnector @ 127.0.0.1:8912
INFO - JBIContainer - ServiceMix JBI Container <jbi> started
INFO - TraceComponent - Exchange: InOnly[
  id: ID:10.10.10.10-1178c7b3577-4:0
  status: Active
  role: provider
  service: {http://servicemix.apache.org/demos/gettingstarted}trace
  endpoint: trace
  in: {<?xml version="1.0" encoding="UTF-8"?><Name>Binil's Servlet wishes you</Name>
me>
} received IN message: org.apache.servicemix.jbi.messaging.NormalizedMessageImpl
@4f71a3{properties: {Date=Fri, 18 Jan 2008 10:39:23 GMT, Content-Length=60, Cont
ent-Type=text/xml, Server=Apache-Coyote/1.1}}
INFO - TraceComponent - Body is: {<?xml version="1.0" encoding="
UTF-8"?><Name>Binil's Servlet wishes you</Name>
-

```

The first part of the console output is showing ServiceMix initializing the components, which we have configured previously in `servicemix.xml`. The components being initialized, in our case, include **trace**, **timer**, **httpGetData**, and **httpReceiver**. Towards the latter part of the output, we can see some XML messages. In fact, what we have done here by bringing ServiceMix up is, we have triggered a message flow through various components configured to the external HTTP service (in Tomcat or so), retrieved the XML message and sent to the console output.

You have successfully invoked an external HTTP service through ServiceMix ESB!

Run a Client against ServiceMix

You can repeat sending messages through ServiceMix ESB by running the client code packaged in the web application code base. To do this, change the directory to `ch03\Servlet`.

To run the client code against ServiceMix, make sure (edit, if required) the run target in the `ch03\Servlet\build.xml` is as shown in the following code:

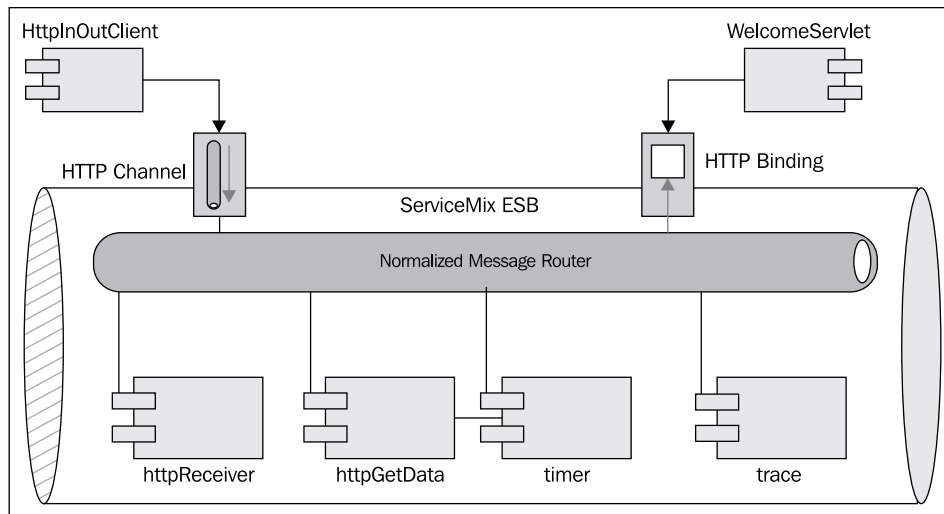
```
<target name="run">
  <java classname="HttpInOutClient" fork="yes" failonerror="true">
    <classpath refid="classpath"/>
    <arg value="http://localhost:8912/EsbServlet/hello/" />
    <arg value="HttpSoapRequest.xml" />
  </java>
</target>
```

Now, executing the following command will send another message through ServiceMix ESB to the HTTP service.

```
ant run
```

What Just Happened in ServiceMix

The following figure shows how we configured various components to the **ServiceMix ESB** in `servicemix.xml`:



As we know, all message exchange happens through the NMR. The description of various components follows:

- **httpReceiver:** Here, we configure `org.apache.servicemix.components.http.HttpConnector` class to listen to a particular port (8912 in our case), connected to the NMR using a HTTP channel. This means an external client (such as `HttpInOutClient` in our case) can send HTTP requests to the NMR through this component.
- **httpGetData:** `org.apache.servicemix.components.http.HttpInvoker` performs HTTP client invocations on a remote HTTP site. As described earlier, we have `EsbServlet.war` deployment containing `WelcomeServlet` providing HTTP service in the remote site. Thus, in effect the `HttpInvoker` functions as a binding component for the remote service.
- **timer:** `org.apache.servicemix.components.quartz.QuartzComponent` is a Quartz component for triggering components when timer events fire. In our case, we have configured `repeatCount` property with a value of zero, which means the trigger will happen only once.
- **trace:** `org.apache.servicemix.components.util.TraceComponent` is a simple tracing component, which can be placed inside a pipeline to trace the message exchange through the component.

Obviously, there are multiple paths which we can define using various combinations of these components. We have configured a typical one in `servicemix.xml` so as to enable the client to send messages through this pipeline to the remote HTTP service. When the client sends a message to the ServiceMix ESB, it reaches the NMR through the `httpReceiver` component. The `destinationService` for `httpReceiver` is `httpGetData`, hence the message is routed to `httpGetData`. However, `httpGetData` is an invoker to the remote HTTP Service. This ends up in invoking the remote HTTP service passing the message parameters. The service gets invoked and any response is routed back through a similar pipeline back to the client.

Spring XML Configuration for ServiceMix

ServiceMix uses XML configuration files. From 2.0 onwards, ServiceMix use the XBean library to do the XML configuration. Thus, the simplest way to start using ServiceMix to wire together JBI components is via Spring and the XML configuration file mechanism from Spring.

We will now look at the details of how we have configured components together. First, we introduce a few new XML tags for JBI configuration such as `container` and `activationSpecs`, but apart from that, you can use all the regular Spring configuration tags—`beans`, `bean`, `property`, and `value`. For example, inside the `<component>` tag you can configure properties on the component. A component can have `<bean>` tag with nested `<bean>` tag and so on. This allows you to mix and match regular Spring configuration of POJOs with the ServiceMix JBI Spring configuration mechanism.

The table shown in the following lists out a few key attribute details for the JBI container element. The ServiceMix website will give details on the elements and attributes which are not described here. Also the bean wiring details in Spring style and all, are not described here, since we assume that the reader is familiar with that. If not, any other text book on Spring will help you here.

Sr . No.	Key Attribute Name	Attribute Type	Description
1	name	String	Denotes name of the JBI container. This has to be unique if it is running in a cluster configuration. The default name is defaultJBI
2	useMBeanServer	Boolean	If set to true, ServiceMix will try to find an MBeanServer from the MBeanServerFactory if one is not supplied
3	createMBeanServer	Boolean	If set to true, ServiceMix will create it's own MBeanServer if one is not supplied to the container or found from an MBeanServerFactory
4	rmiPort	Int	This is the port used for the rmi registry (and thus, the JMX connector) and the default is to 1099.

The ServiceMix component maybe given following names:

- ComponentName
- Service
- Endpoint
- DestinationService

We may also use the following in a component to specify its routing:

- `destinationService`
- `destinationInterface`
- `destinationOperation`

The `PropertyPlaceholderConfigurer` reads `SERVICEMIX_HOME/conf/servicemix.properties` file, which contains values for commonly used environment entries such as `rmi.port` or `jmx.url`.

As we look into more samples in subsequent chapters, you will better understand the mechanism of wiring services in the JBI bus and how we can route messages through these services.

Summary

Just like an EJB component lives in an EJB container, a JBI component lives in a JBI container. A JBI container provides hosting facilities for a JBI component and control the life cycle of the component. Multiple vendors provide their own implementations of the JBI container.

`ServiceMix` is the one we have seen here in detail, which we will be continuing in this book for the purpose of showing JBI using examples. Just like the scenario you have already seen in this chapter of binding an external HTTP service to the JBI container, we can also bind many other protocols and formatted components to the JBI container, thus providing a Service Workbench at the JBI container-level. However, the interesting part to be noted is that even before JBI, we have been doing binding and integration of different kinds of services in multiple ways.

The next chapter is devoted to re-look at how we have been doing things, without a full fledged JBI container. This will help you to get a broader picture of the *as is* state of JBI; from there we will dig more into JBI and related services.

4

Binding— The Conventional Way

Binding services locally or remotely is not an innovation brought by ESB or JBI, but we have been doing it in multiple ways all through the years. In this chapter, we will explore the meaning of "Binding" and then look at how we can bind a remotely available service (in the form of an EJB component) to a middle-tier and expose it through a firewall friendly channel like HTTP. By the end of the chapter, you will appreciate how an extra indirection with a suitable tunneling will help to expose even technology specific API services, so that the service becomes technology agonistic.

We will cover the following in this chapter:

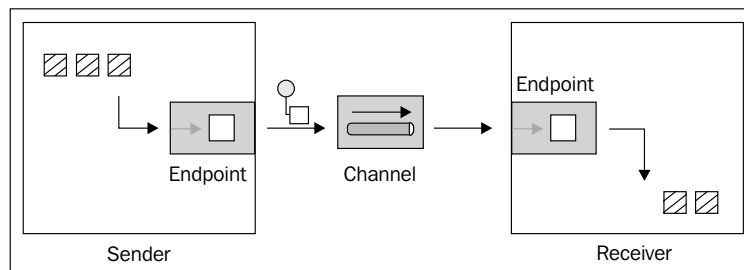
- Meaning of binding
- Apache SOAP binding
- Binding a stateless EJB service to Apache SOAP
- Running the sample

Binding—What it Means

We will consider the very basic requirement of two applications or two services interacting—to exchange messages. Applications share data in the form of messages. One of the applications sends messages while the other receives it. Messages are exchanged between a sender application and a receiver application over a messaging channel. Let us look at binding in this context.

Binding

Applications connect to the messaging channel through a message Endpoint. The process of connecting an application or service to a suitable Endpoint is called 'binding'. In more technical terms, a binding will define how `PortType` (abstract interface of the service) is bound to a particular transport protocol and an encoding schema. A binding interaction involves a service requester and provider. When an application uses the service description to create a message to be sent to the service provider, we are binding to the service.



Endpoints

Since multiple applications or services interact with each other through a messaging channel, they have to deal with multiple transport mechanisms and message formats. Endpoints do the functionality of converting messages from one format to another. Thus the rest of the application knows little about message formats, messaging channels, or any other details of communicating with other applications when they exchange messages. Since endpoints do this message normalization functionality, a message endpoint code is custom to both the application and the messaging system's client API.

When we write a program to a messaging API such as JMS, we're developing endpoint code. This involves either developing low-level plumbing code by hand or using appropriate client APIs and run-time tools to automatically generate code. Later, we will see that a whole lot of endpoints are available, as off the shelf components along with message bus products like ServiceMix. In this chapter, we will look at raw forms of binding service. This example help us to understand what we actually mean by binding.

Apache SOAP Binding

As said earlier, without using a JBI or an ESB framework, let us see binding in action using the Apache open-source SOAP stack.

A Word about Apache SOAP

Apache SOAP is an implementation based on the SOAP submission to W3C (World Wide Web Consortium). This submission has been produced by the XML Protocol Working Group, which is part of the Web Services Activity. IBM Alphaworks first brought a Java reference implementation of the SOAP 1.1 specification, which is contributed to form the Apache SOAP project.

SOAP is a lightweight protocol for the exchange of information in a decentralized, distributed environment. SOAP is based on XML and consists of three parts – an envelope (containing and optional Header and a mandatory Body) that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined data types, and a convention for representing remote procedure calls and responses.

Apache SOAP Format and Transports

Apache SOAP supports three encoding styles:

1. **XMI encoding:** This (available when using Java 1.2.2) provides support for automatic marshalling and unmarshalling of arbitrary objects.
2. **SOAP encoding:** Built-in support is provided for encoding/decoding primitive types like Strings and Integer, arbitrary JavaBeans (using reflection) and one-dimensional arrays of these types. For other types, the user can hand write encoders/decoders and register with the XML-SOAP run time.
3. **Literal XML encoding:** Allows us to send XML elements (DOM org. w3c.dom.Element objects) as parameters by embedding the literal XML serialization of the DOM tree.

Apache SOAP supports messaging and RPC over two transports – HTTP and SMTP. As per the SOAP specification, all SOAP implementations should support SOAP XML payload over HTTP Transport. As optional features, implementations are free to support other transport bindings like JMS, FTP, and SMTP. Vendors can even support proprietary transport bindings for whatsoever reason they have, so there should be nothing which prevents one from not exposing XML SOAP even over radio waves as a transport mechanism!

RPC and Message Oriented

An Apache SOAP service or binding can be RPC or message oriented.

If it is RPC oriented, the run time expects a strict, SOAP formatted request (and response too). The run time will process the SOAP envelope, dispatch the RPC method call request to the appropriate service implementation class and to the appropriate method. Forward the response back to the SOAP client.

If it is a message oriented, request and response, they are transported in a document style – we have freedom to embed arbitrary formats of message inside the SOAP envelope. That means the run time will pass through the SOAP body to the service implementation, and it is the duty of the service implementation to process the contents of the SOAP Envelope.

Binding Services

Apache SOAP utilizes deployment descriptors in the form of XML files to provide information to the SOAP run time about the services that should be made available to clients. They can provide a wide array of information such as the Uniform Resource Names (URN) for the service (which is used to route the request when it comes in), method and class details, if the service is being provided by a Java class or the method. Moreover, the EJB JNDI name, if the service is being provided by an EJB. The exact contents of the deployment descriptor depend upon the type of artifact which is being exposed via SOAP.



URNs are intended to serve as persistent, location-independent resource identifiers



Apache SOAP supports the following artifacts as services to be bound to the run time:

- Standard Java classes
- EJBs
- Bean Scripting Framework (BSF) supported script

The service element is the root element of the deployment descriptor and is shown in the following code:

```
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
            id="urn:ejbhello">
    <!--other code goes here -->
</isd:service>
```

The service element contains an `id` attribute which is used to specify the name of the service. SOAP clients use the value of the `id` attribute to route requests to the service. We will use the URN syntax to specify the name of the service as `urn:ejbhello`. The service element can also contain an optional `type` and an optional `checkMustUnderstands` attribute.

The optional `checkMustUnderstands` will mandate whether the server should understand a particular SOAP header in the message. If the receiver cannot recognize the element, it must fail when processing the header. The server will throw a SOAP fault, if there are SOAP headers in a SOAP request message that have the `mustUnderstand` attribute set to `true`.

The `provider` element is the most important element which contains attributes and sub-elements that specify the details of the artifact that is implementing the service. The `provider` element, attributes, and sub-elements are shown in the following code:

```
<isd:provider type="org.apache.soap.providers.StatelessEJBProvider"
  scope="Application" methods="hello">
  <isd:option key="JNDIName"
    value="sample-statelessSession-TraderHome"/>
  <isd:option key="FullHomeInterfaceName"
    value="samples.HelloServiceHome" />
  <isd:option key="ContextProviderURL" value="t3://localhost:7001" />
  <isd:option key="FullContextFactoryName"
    value="weblogic.jndi.WLInitialContextFactory" />
</isd:provider>
```

The `type` attribute of the `provider` element tells the run time which provider implementation has to be used at run time. The available providers are:

- `java`
- `script`
- `org.apache.soap.providers.StatelessEJBProvider`
- `org.apache.soap.providers.StatefulEJBProvider`
- `org.apache.soap.providers.EntityEJBProvider`

The nested elements within the `provider` elements are specific to the type of artifact used to define the service and are self explanatory.

Sample Bind a Stateless EJB Service to Apache SOAP

In this section, we will walk through a sample binding scenario which will help us to understand binding of services.

Sample Scenario

The sample will make use of the following run-time setups:

- Apache SOAP in Tomcat
- EJB container in BEA Weblogic

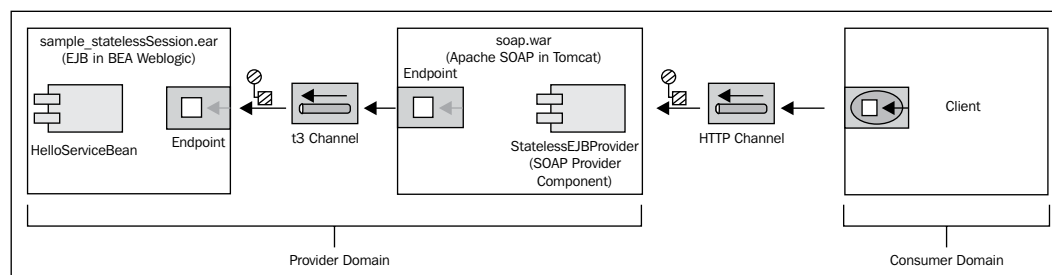
A stateless session EJB (**HelloServiceBean**) is deployed in BEA Weblogic Server which exposes a single method, shown as follows:

```
public String hello(String phrase)
```

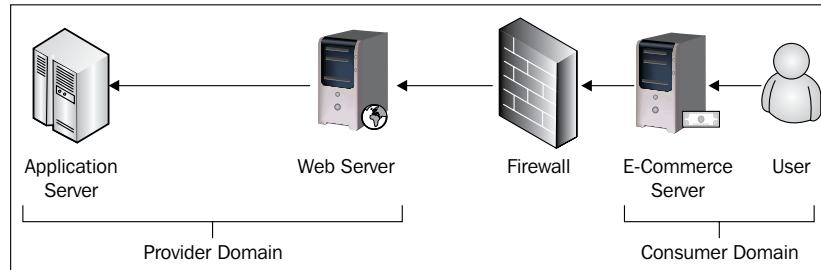
HelloServiceBean exposes an Endpoint. This Endpoint connects to a Weblogic specific t3 channel. t3 channel can pass through t3 protocol, for object service access. Let us assume that we have a requirement of accessing the above EJB service through a HTTP channel (for some reason like a firewall restriction between the server and the client). We can solve this problem by making use of a web server infrastructure.

Even though we can use different web server setups (even Weblogic's web container can be used for this) to achieve this, we will use Apache Tomcat for our demo. Tomcat will host Apache SOAP as an internal web application. We will create a service binding inside the Apache SOAP run time in Tomcat to bind the EJB service. This means, we can chain a HTTP channel using a suitable binding to the t3 channel. Doing so will allow clients to speak the HTTP language to service binding in Tomcat through HTTP channel, and Tomcat will in turn translate the HTTP protocol to t3 protocol and pass through the message to the t3 channel. So that the message will ultimately reach the EJB component hosted in Weblogic application server.

The entire scenario is represented in the following figure:



The deployment for the sample requirement will translate to that shown in the following figure:



Code Listing

In this section, we will walk through the main code used for the demo. All the code discussed here is in the folder `ch04\AxisSoapBindEjb`:

- **Session EJB:** The `HelloServiceBean.java` class is shown here:

```

public class HelloServiceBean implements SessionBean
{
    public String hello(String phrase)
    {
        System.out.println("HelloServiceBean.hello
                           {" + (++times) + "}...");
        return "From HelloServiceBean : HELLO!! You just said : "
               + phrase;
    }
}
  
```

The `HelloServiceBean.java` is a simple stateless session EJB and hence is straightforward.

```

<weblogic-ejb-jar>
  <weblogic-enterprise-bean>
    <ejb-name>statelessSession</ejb-name>
    <enable-call-by-reference>True</enable-call-by-reference>
    <jndi-name>sample-statelessSession-TraderHome</jndi-name>
  </weblogic-enterprise-bean>
</weblogic-ejb-jar>
  
```


We will deploy the EJB in Weblogic server (even though the same EJB can be deployed in any compatible EJB server like Websphere or JBoss) and hence we require the `weblogic-ejb-jar.xml` as shown above. The most important thing to note here is the `jndi-name` whose value we have configured as `sample-statelessSession-TraderHome`.

- **Apache SOAP Binding:** The `DeploymentDescriptor.xml` contains entries as shown in the following code:

```
<?xml version="1.0"?>
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
             id="urn:ejbhello">
  <isd:provider type="org.apache.soap.providers.
                StatelessEJBProvider"
                scope="Application"
                methods="hello">
    <isd:option key="JNDIName"
                value="sample-statelessSession-TraderHome"/>
    <isd:option key="FullHomeInterfaceName"
                value="samples.HelloServiceHome" />
    <isd:option key="ContextProviderURL"
                value="t3://localhost:7001" />
    <isd:option key="FullContextFactoryName"
                value="weblogic.jndi.WLInitialContextFactory" />
  </isd:provider>
  <isd:faultListener>
    org.apache.soap.server.DOMFaultListener
  </isd:faultListener>
</isd:service>
```

Here, we can note that we have given the value `sample-statelessSession-TraderHome` for the key `JNDIName`. So, here we are referring to the EJB service we deployed previously. We have configured `weblogic.jndi.WLInitialContextFactory`, which can create initial contexts for accessing the WebLogic naming service.

We have configured `org.apache.soap.providers.StatelessEJBProvider` for binding the EJB service to speak SOAP protocol. `StatelessEJBProvider` implements the two methods defined in `org.apache.soap.util.Provider`, that is shown in the following code:

```
public interface Provider
{
    public void locate(DeploymentDescriptor dd, Envelope env,
                      Call call, String methodName, String targetObjectURI,
                      SOAPContext reqContext) throws SOAPException ;
}
```

```

    public void invoke(SOAPContext req, SOAPContext res)
                        throws SOAPException ;
}

```

A provider's function is split into two pieces – locating the service (which also includes any verification that the service can be executed at all and by the client), and actually running it. It's up to the `invoke` method to call the actual service up to grab and format the response into a SOAP response object.

During initialization, `StatelessEJBProvider` will do the following to get a remote reference to the EJB component:

```

EJBHome home = (EJBHome) PortableRemoteObject.narrow(context.
    lookup(jndiName), Class.forName(homeInterfaceName));
Method createMethod = home.getClass().
    getMethod("create", new Class[0]);
remoteObjRef = (EJBObject) createMethod.invoke((Object) home,
    new Object[0]);

```

During actual method invocation `StatelessEJBProvider` will execute the following code:

```

Method m = MethodUtils.getMethod (remoteObjRef,
    methodName, argTypes);
Bean result = new Bean (m.getReturnType (),
    m.invoke (remoteObjRef, args));

```

Running the Sample

As a first step and if you haven't done it before, edit `examples.PROPERTIES` provided along with the code download for this chapter and change the paths there to match your development environment. The code download for this chapter also includes a `README.txt` file, which gives detailed steps to build and run the samples.

Running the sample involves multiple steps, as shown as follows:

Deploying the EJB

For deploying the EJB, we have to follow general EJB deployment steps as specified in the Weblogic documentation.

Since you are building your first EJB sample in this book, we will spend a little more time to help you to deploy your EJB sample. First, change directory to the EJB samples directory.

```
cd ch04\AxisSoapBindEjb\ejb
```

Now, set the environment variables for the build console as per the Weblogic documentation and then use the ant script provided along with the Weblogic server bundle. For that, do the following:

```
%wl.home%\samples\domains\examples\setExamplesEnv.bat
%wl.home%\server\bin\ant
```

Assuming that you are using the Weblogic 8.x version, the above steps should have built the EJB by now. If you use a different version of the EJB server, or if you use a different vendor's EJB server, refer to the documentations there to make changes to the build exercise.

We can even test whether our EJB deployment went fine by executing a test client, like:

```
cd ch04\AxisSoapBindEjb\ejb
%wl.home%\server\bin\ant run
```

Bind EJB to SOAP

We will use the Apache SOAP implementation for this demonstration. From the Apache SOAP distribution, copy the `soap.war` file and deploy that to the `webapps` folder of your favorite web server (Apache Tomcat). You also need to copy the following files and make them available in the `lib` folder of your web server:

- `%wl.home%\server\lib\weblogic.jar`
- `%wl.home%\samples\server\examples\build\clientclasses\sample_statelessSession_client.jar`

Now restart your web server.

Apache SOAP provides `ServiceManagerClient` using which we can register the remote EJB binding to the SOAP environment. The following ant task does exactly this:

```
<target name="deploy" depends="compile">
  <java classname="org.apache.soap.server.ServiceManagerClient"
        fork="true" >
    <arg value="http://localhost:8080/soap/servlet/rpcrouter"/>
    <arg value="deploy"/>
    <arg value="DeploymentDescriptor.xml"/>
    <classpath>
      <path refid="classpath"/>
    </classpath>
  </java>
</target>
```

This ant target can be invoked by typing:

```
cd ch04\AxisSoapBindEjb\SoapBind
ant deploy
```

The `ServiceManagerClient` in the above deploy target takes three parameters – the URL to the SOAP server, the command `deploy`, and the file containing your deployment descriptor. The first parameter specifies the URL of the Apache SOAP RPC router, which will help in routing client requests to the requested service. The second parameter specifies the operation that the `ServiceManagerClient` should do. The `deploy` operation registers the service using the deployment information located in a file specified by the third parameter. In our case, the file is called `DeploymentDescriptor.xml` and it contains the deployment descriptor for the `hello Service`.

Once you have executed the `deploy` command, you can execute the `list` command. You should now see output listing `urn:ejbhello`, which is the unique ID of your service. You can also view this service from the web admin tool by going to `http://localhost:8080/soap/admin/index.html` and selecting the `List` button. The `list` ant target to execute the `list` command is as follows:

```
<target name="list">
  <java classname="org.apache.soap.server.ServiceManagerClient"
        fork="true" >
    <arg value="http://localhost:8080/soap/servlet/rpcrouter"/>
    <arg value="list"/>
    <classpath>
      <path refid="classpath"/>
    </classpath>
  </java>
</target>
```

We can verify whether the deployment was successful by typing the following:

```
ant list
```

Run the Client

The client creates a `Call`, sets parameters, and invokes the service binding as shown in the following code:

```
URL url = new URL (args[0]);
Call call = new Call ();
call.setEncodingStyleURI (Constants.NS_URI_SOAP_ENC);
call.setTargetObjectURI ("urn:ejbhello");
call.setMethodName ("hello");
Vector params = new Vector ();
params.addElement (new Parameter("phrase", String.class,
                                "what's your name?", null));

call.setParams (params);
Response resp = call.invoke (url, " ");
Parameter result = resp.getReturnValue();
```

The run ant target will execute the client code.

```
<target name="run">
  <java classname="samples.ejb.SoapTest" fork="true" >
    <arg value="http://localhost:8080/soap/servlet/rpcrouter"/>
    <classpath>
      <path refid="classpath"/>
    </classpath>
  </java>
</target>
```

To run the client, type:

ant run

What Just Happened

If everything went fine, you would see something similar to the result as shown in the following screenshot:



```
Command Prompt
D:\binil\com\java\com\binildas\esh\ServiceMix\Text\ESBUsingJBI-ServiceMix\ch04\AxisSoapBindEjb\SoapBind>ant deploy
Buildfile: build.xml

clean:
[delete] Deleting directory D:\binil\com\java\com\binildas\esh\ServiceMix\Text\ESBUsingJBI-ServiceMix\ch04\AxisSoapBindEjb\SoapBind\build

init:
[mkdir] Created dir: D:\binil\com\java\com\binildas\esh\ServiceMix\Text\ESBUsingJBI-ServiceMix\ch04\AxisSoapBindEjb\SoapBind\build

compile:
[javac] Compiling 1 source file to D:\binil\com\java\com\binildas\esh\ServiceMix\Text\ESBUsingJBI-ServiceMix\ch04\AxisSoapBindEjb\SoapBind\build
[javac] Note: D:\binil\com\java\com\binildas\esh\ServiceMix\Text\ESBUsingJBI-ServiceMix\ch04\AxisSoapBindEjb\SoapBind\SoapTest.java uses unchecked or unsafe operations.
[javac] Note: Recompile with -Xlint:unchecked for details.

deploy:
BUILD SUCCESSFUL
Total time: 1 second
D:\binil\com\java\com\binildas\esh\ServiceMix\Text\ESBUsingJBI-ServiceMix\ch04\AxisSoapBindEjb\SoapBind>ant list
Buildfile: build.xml

list:
[java] Deployed Services:
[java] urn:ejbhello

BUILD SUCCESSFUL
Total time: 0 seconds
D:\binil\com\java\com\binildas\esh\ServiceMix\Text\ESBUsingJBI-ServiceMix\ch04\AxisSoapBindEjb\SoapBind>ant run
Buildfile: build.xml

run:
[java] Done. result : From HelloServiceBean : HELLO!! You just said :what's your name?

BUILD SUCCESSFUL
Total time: 0 seconds
D:\binil\com\java\com\binildas\esh\ServiceMix\Text\ESBUsingJBI-ServiceMix\ch04\AxisSoapBindEjb\SoapBind>
```

The SoapTest has actually created a SOAP request and routed it to the service binding within SOAP run time. The format of this SOAP request is as follows:

```
POST /soap/servlet/rpcrouter HTTP/1.0
Host: localhost:8080
Content-Type: text/xml; charset=utf-8
Content-Length: 458
SOAPAction: ""
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV=
    "http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema
        -instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SOAP-ENV:Body>
    <ns1:hello xmlns:ns1="urn:ejbhello"
      SOAP-ENV:encodingStyle=
        "http://schemas.xmlsoap.org/soap/encoding/">
      <phrase xsi:type="xsd:string">
        what&apos;s your name?
      </phrase>
    </ns1:hello>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

A HTTP-based SOAP endpoint is identified by a URL. This URL, in our case, in the above request is `http://localhost:8080/soap/servlet/rpcrouter`. `SOAPMethodName` is an optional header, which uniquely identifies the method name. For HTTP transport, SOAP messages are sent over `POST` method. The SOAP message body will contain the XML formatted content required for the SOAP run time to invoke the requested method.

In the SOAP content, the root element is an element whose namespace-qualified tag name (`ns1:hello` in our case) matches the optional `SOAPMethodName` HTTP header. This redundancy is provided to allow any HTTP infrastructure intermediaries like proxies, firewalls, web server software, to process the call without parsing XML, at the same time also allowing the XML payload to stand independent of the enclosing HTTP message.

Upon receiving this request, the `StatelessEJBProvider` comes into action. the `StatelessEJBProvider` unmarshalls the SOAP Body content and retrieves the method parameters. It then invokes the EJB service, which is remotely bound to the SOAP run time. The results are then packaged back to a SOAP response body and sent back to the client.

The SOAP response is shown in the following code:

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Set-Cookie: JSESSIONID=929DAB0C201F82C9B3F575C8276692A1; Path=/soap
Content-Type: text/xml;charset=utf-8
Content-Length: 523
Date: Thu, 26 Oct 2006 08:20:23 GMT
Connection: close

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV=
    http://schemas.xmlsoap.org/soap/envelope/
    xmlns:xsi="http://www.w3.org/2001/
        XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SOAP-ENV:Body>
    <ns1:helloResponse xmlns:ns1="urn:ejbhello"
      SOAP-ENV:encodingStyle=
        "http://schemas.xmlsoap.org/soap/encoding/">
      <return xsi:type="xsd:string">
        From HelloServiceBean :
        HELLO!! You just said :what&apos;s your name?
      </return>
    </ns1:helloResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

As you can see above, the SOAP response doesn't have any SOAP specific HTTP headers. The payload will contain an XML document that contains the results of the operation invoked. The results will be inside an element with the name matching the method name suffixed by Response.

How the Sample Relates to ServiceMix

In this sample, we saw how to use conventional tools like an EJB container, Apache SOAP toolkit, and Apache Tomcat web container to bind an external EJB service remotely into SOAP run time and expose the service over a HTTP transport channel. Later, we will see that the same functionality can be done without all these hassles and deployment steps, but with just few component configurations in ServiceMix. The idea behind this example is to make the reader familiar with the technical case behind the need for service binding and also to reckon that service binding is not a new concept, but something we have been doing for a long time.

Summary

In this chapter, we discussed a simple binding scenario by which we can make an EJB service accessible across the firewall to the outside world. If you have an Apache SOAP implementation with you, you can also try out the samples and see the results. Similar to what we did in this sample, at times we need to tunnel or redirect services through intermediaries either due to protocol or format mismatches or due to some other reasons.

All these activities are concerns apart from your business logic coding and need to be addressed at your application framework-level. In this chapter too, we have performed it in the same manner, but we have done a lot of steps to configure the web server with Apache SOAP. Later, when you go through the JBI and ESB samples you will realize that the same functionality can be achieved by selecting and configuring suitable JBI components.

ESB implementations like ServiceMix make use of new generation SOAP frameworks which bind binary services such as EJB and POJO to the JBI bus to make it SOA-compliant.

In the next chapter, we will delve more into such a SOAP framework namely XFire and look at some very useful mediation features of it. In the later chapters, we will leverage similar features from the JBI bus straightaway.

5

Some XFire Binding Tools

JBIs advocates that XML data based on a WSDL model should be flowing through the NMR. Hence, a reference to an appropriate SOAP framework capable of understanding WSDL and generating WSDL-compliant formatted data is important in any JBI discussion.

ServiceMix has the best integration with XFire, which is the new generation Java SOAP framework. Since the API is easy to use and supports standards, XFire makes SOA development much easier and straightforward. It is also highly performance oriented, since it is built on a low memory StAX (Streaming API for XML) model. Currently, XFire is available in version 2.0 under the name CXF. In this chapter, we will not discuss any JBI specific binding methods; instead we will concentrate on XFire and look at how we can use the same for integration solutions.

Once we appreciate this, we will better understand what part of the integration functionality can be done using XFire within the JBI architecture.

We will cover the following in this chapter:

- Binding in XFire
- Web service using XFireConfigurableServlet
- Web service using XFire Spring XFireExporter
- Web service using XFire Spring Jsr181 handler
- XFire Export and bind EJB
- XFire for lightweight integration

Binding in XFire

In XFire terms, bindings are ways to map XML to Java objects. Currently, XFire supports the following bindings:

- **Aegis:** Aegis is the default XFire binding which maps XML to POJOs. It supports code first development where you write your service in POJOs and Aegis will generate the XML schema or WSDL for you. It has a flexible mapping system so that you can control how your POJOs are bound.
- **JAXB:** JAXB is the reference implementation of the JAXB specification. XFire integrates with JAXB 1.0 and 2.0.
- **XMLBeans:** XMLBeans is an Apache project which takes the richness, features, and schema of XML and maps these features as naturally as possible to the equivalent Java language and typing constructs.
- **Message:** The MessageBinding has special semantics to allow you to work with XML streams and fragments very easily. The MessageBinding takes the XMLStreamReader from the request and provides it directly to your classes.
- **Castor:** Castor provides marshalling and unmarshalling of XML and Java objects which doesn't require recompilation of the Java code if the mapping definition changes. Hence systems where the service layer is being developed independently from the business layer can benefit much using Castor.

A detailed description of the above frameworks is beyond the scope of this book, but we need to look at what XFire has to offer and how this is going to help us in binding services and components to the ServiceMix ESB.

XFire Transports

XFire provides multiple transport channels for communications and is built on an XML messaging layer. The main transport mechanisms are listed as follows:

- **HTTP:** Standard XML in SOAP format over HTTP
- **JMS:** Asynchronous and reliable way of sending messages
- **XMPP or Jabber:** An asynchronous messaging mechanism for SOAP

A Service Registry is the central part of the XFire messaging infrastructure. Users can send messages through any of the available transport channels. The transport looks at the service being invoked and passes a MessageContext and service name off to the XFire class. The XFire class looks up the service from the Service Registry and then invokes the appropriate handler. Thus XFire transport is responsible for managing incoming and outgoing communications using a particular wire protocol, which is shown as abstract in the method signature in the following code:

```
public interface Transport extends ChannelFactory, HandlerSupport
{
    Binding findBinding(MessageContext context, Service service);
}
```

JSR181 and XFire

Java Web Services Metadata (WSM, based on JSR 181) is built over Java Language Metadata technology (JSR 175). The intention behind JSR181 is to provide an easy to use syntax for describing web services at the source-code-level for the J2EE platform. Thus, WSM leverages the metadata facility in Java to web services. In other words, a Java web service is just a Plain Old Java Object (POJO) with a few annotations. The annotations can describe the web service name, details about the methods that should be exposed in the web service interface, parameters, their types, the bindings, and other similar information. XFire is continuously adding support for JSR181 and this chapter also gives an introduction to this, with working examples.

We will walk through four different but related working samples. All of them demonstrate capabilities of XFire. The part to notice is the simplicity with which we can get things done with XFire.

Web Service Using XFireConfigurableServlet

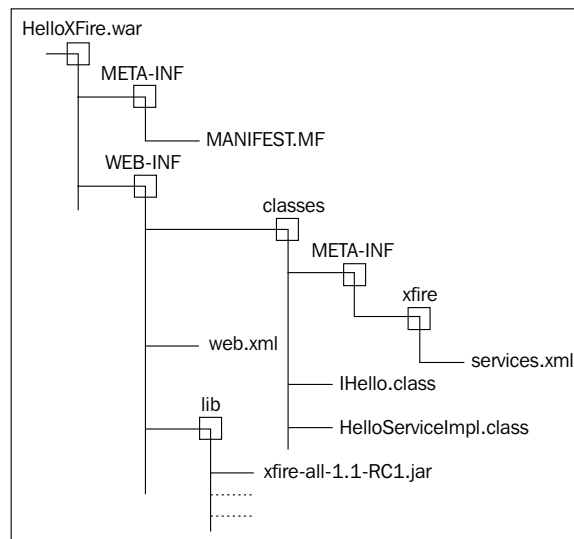
Web services are to a great extent in the spirit of SOA. Most web services frameworks internally uses a SOAP stack for transport and format handling. Since XFire is a SOAP stack capable of easily building web services, let us look into one sample doing that here.

Sample Scenario

Our aim here is to expose a POJO as web service using XFire—`org.codehaus.xfire.transport.http.XFireConfigurableServlet`.

Code Listing

XFireConfigurableServlet expects a service definition in the form of an xml file called `services.xml`. XFire by itself is a web-based application; hence we are going to package the sample application as a standard web archive.



We will now look at the contents of the individual artifacts that make up the web archive:

1. **IHello.class:** IHello is a simple Java interface, declaring a single method `sayHello`.

This is shown in the following code:

```
public interface IHello
{
    String sayHello(String name);
}
```

2. **HelloServiceImpl.class:** HelloServiceImpl implements the IHello interface and has some verbose code printing out details into the server console.

The following code demonstrates it:

```
public class HelloServiceImpl implements IHello
{
    private static long times = 0L;
    public HelloServiceImpl()
    {
        System.out.println("HelloServiceImpl.HelloServiceImpl()...");
    }
}
```

```

    }
    public String sayHello(String name)
    {
        System.out.println("HelloServiceImpl.sayHello(" +
                           (++times) + ")");
        return "HelloServiceImpl.sayHello : HELLO! You just said: "
               + name;
    }
}

```

3. **services.xml**: Here we specify the details of our web services, using the `serviceClass` and `implementationClass` elements. `serviceClass` specifies the Java interface, which hosts the method signature whereas `implementationClass` specifies the class which implements the method. All the methods in `serviceClass` will be exposed as web services. If the `implementationClass` doesn't implement any interface, the `serviceClass` element can have the `implementationClass` itself as its value. `services.xml` is placed within the `WEB-INF\classes\META-INF\xfire` directory, so that the XFire run time can set up the service environment. The name and namespace elements can have any valid XML names as the values.

It is shown in the following code:

```

<beans xmlns="http://xfire.codehaus.org/config/1.0">
    <service>
        <name>Hello</name>
        <namespace>myHello</namespace>
        <serviceClass>IHello</serviceClass>
        <implementationClass>HelloServiceImpl</implementationClass>
    </service>
</beans>

```

4. **web.xml**: The main part in `web.xml` is configuring the `XFireConfigurableServlet` as shown here:

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app>
    <servlet>
        <servlet-name>XFireServlet</servlet-name>
        <display-name>XFire Servlet</display-name>
        <servlet-class>
            org.codehaus.xfire.transport.http.XFireConfigurableServlet
        </servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>XFireServlet</servlet-name>
        <url-pattern>/servlet/XFireServlet/*</url-pattern>
    </servlet-mapping>
</web-app>

```

```
<servlet-name>XFireServlet</servlet-name>
<url-pattern>/services/*</url-pattern>
</servlet-mapping>
</web-app>
```

Any request of a URL with the pattern `/services/*`, will be routed to the `XFireServlet`, which will in turn do the magic of SOAP handshaking.

Running the Sample

As a first step and if you haven't done it before, edit `examples.PROPERTIES` provided along with the code download for this chapter, and change the paths there to match your development environment. If your web server doesn't include Xalan (Xalan is an XSLT processor for transforming XML documents) libraries, download Xalan for Java, and transfer `xalan*.jar` from the download to the `libraries` folder of your web server (`${tomcat.home}/lib`). The code download for this chapter also includes a `README.txt` file, which gives detailed steps to build and run the samples.

To build the sample, change directory to `ch05\01_XFireServletWebService` and execute `ant`, as shown here:

```
cd ch05\01_XFireServletWebService
ant
```

This will generate the `war` file, which in turn contains all required libraries extracted out from the XFire installation folder. Folder `dist` will contain `HelloXFire.war`, which should be deployed in the `webapps` folder of Tomcat (or any other relevant web server). Now, restart the server.

The web service would have been exposed by now and the service definition can be accessed using the following URL:

```
http://localhost:8080/HelloXFire/services/Hello?WSDL
```

We can now write a `Client` code to test the previous web service as shown in the following code:

```
public class Client
{
    private static String serviceUrl = "http://localhost:8080/
                                      HelloXFire/services/Hello";
    public static void main(String[] args) throws Exception
    {
        if (args.length > 0)
        {
            serviceUrl = args[0];
        }
    }
}
```

```

        Client client = new Client();
        client.callWebService("Sowmya");
    }
    public String callWebService(String name) throws Exception
    {
        Service serviceModel = new ObjectServiceFactory().create(
                                                    IHello.class);
        XFire xfire = XFireFactory.newInstance().getXFire();
        XFireProxyFactory factory = new XFireProxyFactory(xfire);
        IHello client = null;
        try
        {
            client = (IHello) factory.create(serviceModel, serviceUrl);
        }
        catch (MalformedURLException e)
        {
            log("WsClient.callWebService(): EXCEPTION: " +
                e.toString());
        }
        String serviceResponse = "";
        try
        {
            serviceResponse = client.sayHello(name);
        }
        catch (Exception e)
        {
            log("Client.callWebService(): EXCEPTION: " + e.toString());
            serviceResponse = e.toString();
        }
        return serviceResponse;
    }
}

```

At the client side, we perform the following steps:

- Create a service model which contains the service specification from the interface `IHello`.
- Get XFire instance using `XFireFactory`.
- Retrieve a proxy factory instance for XFire.
- Using the proxy factory, we can now get a local proxy for the remote web service using the service model and the service endpoint URL.
- Now invoke the remote web service using the proxy.

To run the client, assuming that you have already compiled the client while building the sample, execute `ant` as follows:

```
ant run
```


Web Service using XFire Spring XFireExporter

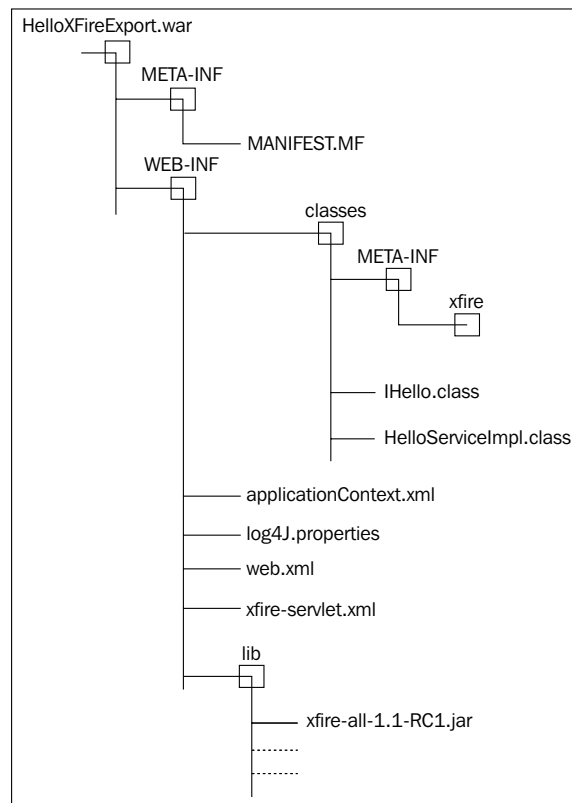
Having seen how to expose a POJO as web service using the XFire class, `org.codehaus.xfire.transport.http.XFireConfigurableServlet`, our next aim is to do the same using a different approach.

Sample Scenario

Here again, our aim is to expose a POJO as web service using XFire Spring support class `org.codehaus.xfire.spring.remoting.XFireExporter`. Here, the XFire class `XFireExporter` is internally leveraging Spring's remoting framework and as such depends on the Spring libraries.

Code Listing

The artifacts in the code listing is shown in the following figure:



We have few more artifacts to be packaged in this scenario and the packaging too is slightly different as shown in the above figure. Since we are using the same classes (IHello and HelloServiceImpl) as used in the previous example here too, they are not repeated here. We will look at the other artifacts in detail here:

1. **IHello.class:** This is same as in the previous example.
2. **HelloServiceImpl.class:** This is same as in the previous example.
3. **applicationContext.xml:** The HelloServiceImpl class is configured in Spring's applicationContext as shown here:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>
    <bean id="helloBean" class="HelloServiceImpl"/>
</beans>
```

4. **xfire-servlet.xml:** In xfire-servlet.xml, the main part is the configuration of the web controller that exports the specified service bean as an XFire SOAP service endpoint. Typically, we will do this in the servlet context configuration file. As given in next section, since the dispatcher servlet is named xfire, this file should be called xfire-servlet.xml.

This is shown in the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>
    <bean class="org.springframework.web.servlet.
        handler.SimpleUrlHandlerMapping">
        <property name="urlMap">
            <map>
                <entry key="/HelloService">
                    <ref bean="hello"/>
                </entry>
            </map>
        </property>
    </bean>
    <bean id="hello"
        class="org.codehaus.xfire.spring.remoting.XFireExporter">
        <property name="serviceFactory">
            <ref bean="xfire.serviceFactory"/>
        </property>
        <property name="xfire">
            <ref bean="xfire"/>
        </property>
        <property name="serviceBean">
```

```
        <ref bean="helloBean"/>
    </property>
    <property name="serviceClass">
        <value>IHello</value>
    </property>
</bean>
</beans>
```

We have configured helloBean as the serviceBean here. In fact, helloBean is coming from the applicationContext.xml configuration.

5. **web.xml:** The feature to be noted in web.xml, is the DispatcherServlet configuration which provides the locations of where your Spring beans are. The contextConfigLocation tells Spring where to find the applicationContext.xml files.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
            /WEB-INF/applicationContext.xml
            classpath:org/codehaus/xfire/spring/xfire.xml
        </param-value>
    </context-param>
    <listener>
        <listener-class>
            org.springframework.web.util.Log4jConfigListener
        </listener-class>
    </listener>
    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
    <servlet>
        <servlet-name>xfire</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>xfire</servlet-name>
        <url-pattern>/services/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Running the Sample

To build the sample, change directory to `ch05\02_XFireExportWebService` and execute `ant` as follows:

```
cd ch05\02_XFireExportWebService
ant
```

This will generate the `war` file, which in turn contains all required libraries extracted out from the XFire installation folder. Folder `dist` will contain `HelloXFireExport.war` which should be deployed in the `webapps` folder of Tomcat (or any other relevant web server). Now, restart the server.

The web service would have been exposed by now and the service definition can be accessed using the following URL:

```
http://localhost:8080/HelloXFireExport/services/HelloService?WSDL
```

We can now write a `Client` similar to what we have seen in the previous example to test the web service.

To run the client, assuming that you have already compiled the `Client`, while building the sample, execute `ant` as follows:

```
ant run
```

Web Service Using XFire Spring Jsr181 Handler

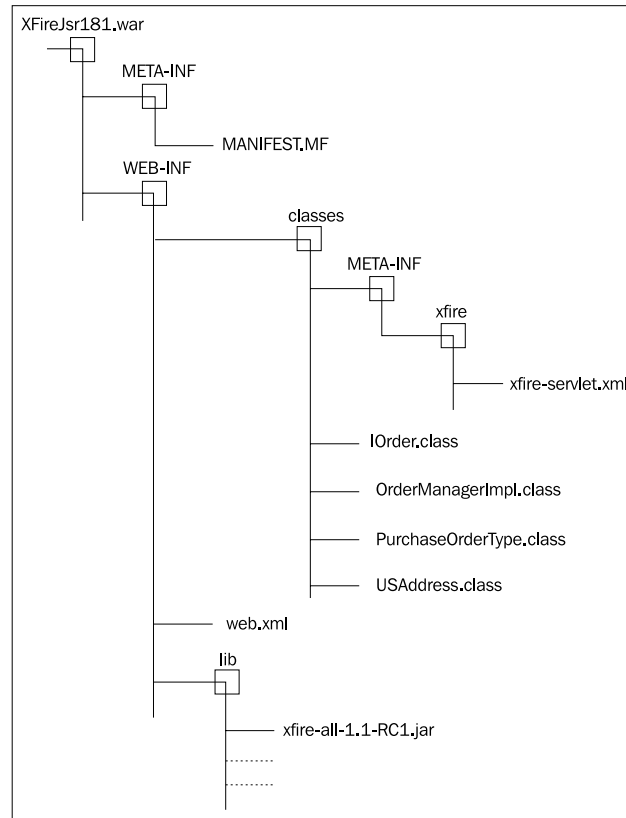
JSR 181 defines an annotated Java syntax for programming web services and is built on the Java Language Metadata technology (JSR 175). It provides an easy to use syntax to describe web services at the source-code-level for the J2EE platform. It aims to make it easy for a Java developer to develop server applications that conform both to basic SOAP and WSDL standards. In this sample, we will increase the complexity of the web service by including a Transfer Object (TO) as a parameter. We will also see JSR 181 annotations work behind the scenes to generate the plumbing required to expose web services.

Sample Scenario

Here again, our aim is to expose a POJO as web service and use JSR 181 annotation support for this. Both the service interface and the service implementation will be annotated.

Code Listing

The artifacts in the code listing is shown in the following figure:



We will list out the different artifacts which make up this example in the following:

1. **IOrder.class:** This is the service interface and is web service annotated. Since all methods in the interface get exported by default, you don't need to define the `@WebMethod` annotation in the interface. Annotations like `@WebResult` too, are totally optional and let you control how the WSDL looks like.

The service interface, `IOrder.class`, is shown in the following code:

```
import javax.jws.WebService;
import javax.jws.WebResult;
@WebService
public interface IOrder
{
    @WebResult(name="PurchaseOrderType")
    public PurchaseOrderType getPurchaseOrderType(String orderId);
}
```

2. **OrderManagerImpl.class:** As you can see here, OrderManagerImpl is the service implementation class and is web service annotated. We use the @WebService annotation with some parameters like web service name as that is used on the client-side to decorate the class and tell the jsr181 processor that there is an interface to export the web service methods.

```
import javax.jws.WebService;
@WebService(serviceName = "OrderService",
            endpointInterface = "IOrder")
public class OrderManagerImpl implements IOrder
{
    public PurchaseOrderType getPurchaseOrderType(String orderId)
    {
        PurchaseOrderType po = new PurchaseOrderType();
        po.setOrderDate( getDate() );
        USAddress shipTo = createUSAddress( "Binil Das",
                                           "23 Hidden Range",
                                           "Dallas",
                                           "TX",
                                           "17601" );
        USAddress billTo = createUSAddress( "Binil Das",
                                           "29K Colonial Creast",
                                           "Mountville",
                                           "PA",
                                           "17601" );

        po.setShipTo(shipTo);
        po.setBillTo(billTo);
        return po;
    }
}
```

It is again worth noting the parameter or return type here, since it is no longer simple Java type, but a custom class (complex type).

3. **web.xml:** If you compare this example with the first one in this chapter, you will notice the difference here. Instead of XFireConfigurableServlet we are going to use org.codehaus.xfire.spring.XFireSpringServlet as the controller servlet. Then you specify the url-pattern so that such pattern URLs can be routed to XFireSpringServlet.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app>
    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
```

```
        classpath:org/codehaus/xfire/spring/xfire.xml,
        /WEB-INF/classes/META-INF/xfire/xfire-servlet.xml
    </param-value>
</context-param>
<servlet>
    <servlet-name>XFireServlet</servlet-name>
    <servlet-class>
        org.codehaus.xfire.spring.XFireSpringServlet
    </servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>XFireServlet</servlet-name>
    <url-pattern>/servlet/XFireServlet/*</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>XFireServlet</servlet-name>
    <url-pattern>/services/*</url-pattern>
</servlet-mapping>
</web-app>
```

4. **xfire-servlet.xml:** Next, we have to define the Spring applicationContext for XFire called `xfire-servlet.xml`. The mechanism here is to define Spring's `SimpleUrlHandlerMapping` which in turn makes use of the XFire Spring `Jsr181HandlerMapping`. Once that is done, you only need to define your web service POJOs like the one seen with the `id` as `annotatedOrder`.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>
    <bean id="webAnnotations"
        class="org.codehaus.xfire.annotations.
            jsr181.Jsr181WebAnnotations"/>
    <bean id="handlerMapping"
        class="org.codehaus.xfire.spring.
            remoting.Jsr181HandlerMapping">
        <property name="typeMappingRegistry">
            <ref bean="xfire.typeMappingRegistry"/>
        </property>
        <property name="xfire">
            <ref bean="xfire"/>
        </property>
        <property name="webAnnotations">
            <ref bean="webAnnotations"/>
        </property>
    </bean>
    <bean id="annotatedOrder" class="OrderManagerImpl"/>
    <bean class="org.springframework.web.servlet.handler.
        SimpleUrlHandlerMapping">
        <property name="urlMap">
            <map>
                <entry key="/">
```

```
        <ref bean="handlerMapping"/>
      </entry>
    </map>
  </property>
</bean>
<import resource="
  classpath:org/codehaus/xfire/spring/xfire.xml"/>
</beans>
```

Running the Sample

To build the sample, change directory to `ch05\03_XFireJsr181BindWebService` and execute `ant` as follows:

```
cd ch05\03_XFireJsr181BindWebService
ant
```

This will generate the `war` file, which in turn contains all required libraries extracted out from the XFire installation folder. Folder `dist` will contain `XFireJsr181.war`, which should be deployed in the `webapps` folder of Tomcat (or any other relevant web server). Now, restart the server.

The web service would have been exposed by now and the service definition of the same can be accessed using the following URL:

```
http://localhost:8080/XFireJsr181/services/OrderService?WSDL
```

We can now write a `Client` similar to what we have seen in the previous few examples to test the web service.

To run the client, assuming that you have already compiled the `Client` while building the sample, execute `ant` as follows:

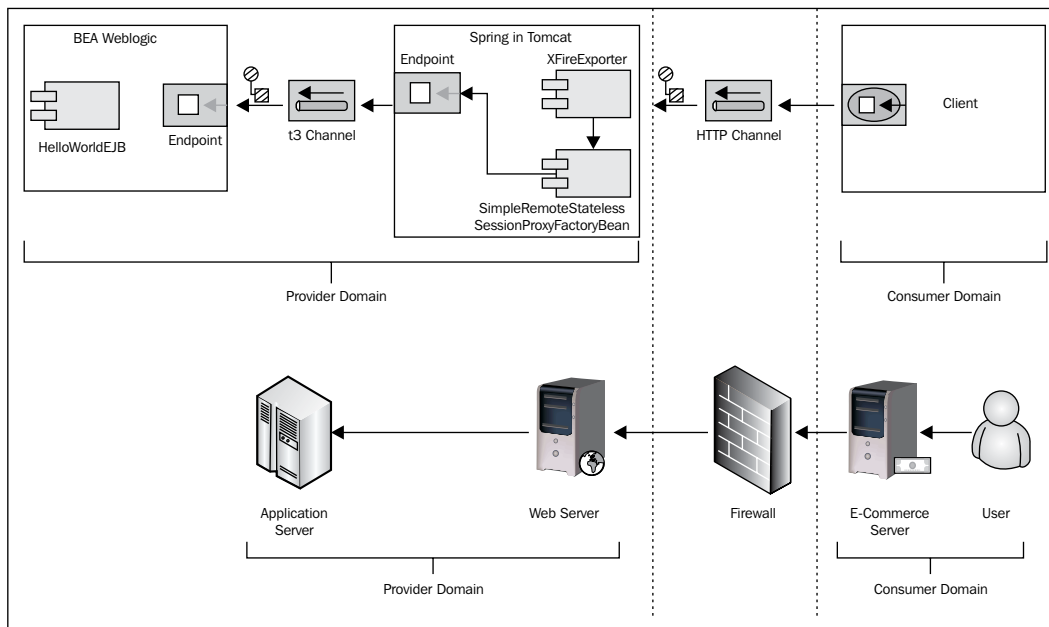
```
ant run
```

XFire Export and Bind EJB

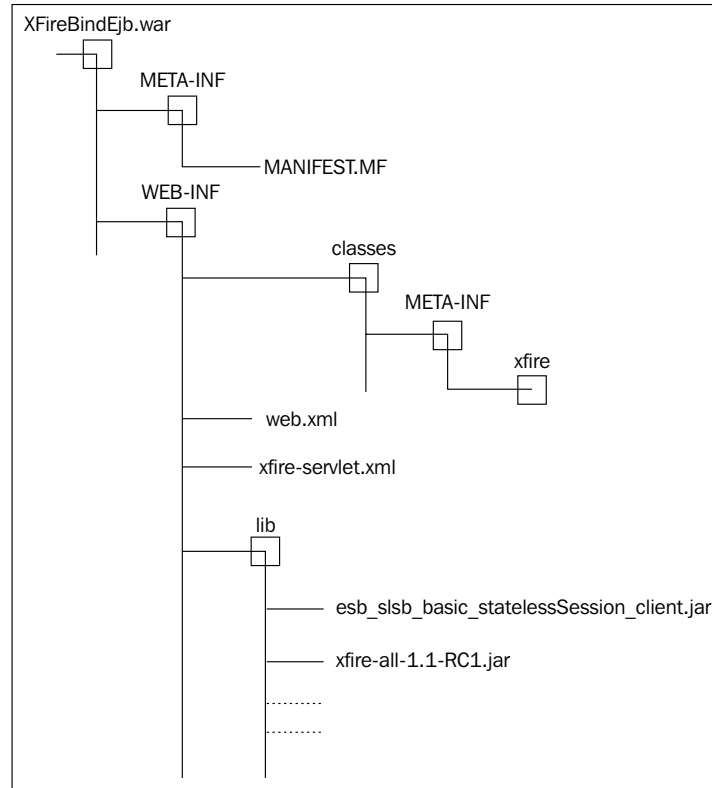
In Chapter 4, you have already seen the sample of how to expose a stateless EJB service deployed in an application server. Hence, the consumer can use the SOAP protocol to invoke the EJB service through a HTTP channel. Let us again do a similar demonstration here, but with XFire now. Once you complete this example you will better appreciate the similarity between XFire mechanisms and the process of binding.

Sample Scenario

The scenario is to expose an EJB component service to external clients through a HTTP channel. The difference between the EJB and SOAP sample in Chapter 4 (*Binding – The Conventional Way*) is that here we will use XFire classes for service binding. Also, the three previous samples in this chapter have demonstrated the power of XFire in exposing web services. But this sample is more towards binding an external service, which, as you will see very shortly, is more similar to the binding activity we do using a JBI stack.



Code Listing



The code for this sample is organized into two, in two separate folders, one for the EJB part and another for the XFire binding. We will walk through the main classes only, in the following:

1. **HelloWorldBI.class:** HelloWorldBI is the Business Interface (BI) class for the stateless enterprise Java session bean, hence very simple and straightforward. This is demonstrated as follows:

```

public interface HelloWorldBI
{
    String sayHello(int num, String s) throws IOException;
}

```

2. **HelloWorldEJB.class:** HelloWorldEJB is the EJB implementation class. We will again use Weblogic libraries to make coding our EJB simpler; hence we use Weblogic's GenericSessionBean as the base class, which will have default implementations for the EJB interface. You may want to deploy the EJB into a different application server in which case tweaking the code and configuration files should be a trivial exercise.

This is shown in the following code:

```
public class HelloWorldEJB extends GenericSessionBean
    implements HelloWorldBI
{
    public String sayHello(int num, String s)
    {
        System.out.println("sayHello in the HelloWorldEJB has "+
            "been invoked with arguments " + s + " and " + num);
        String returnValue = "This message brought to you by the "+
            "letter "+s+" and the number "+num;
        return returnValue;
    }
}
```

3. **xfire-servlet.xml:** For readers who are familiar with Spring, xfire-servlet.xml is self explanatory but for the interest of others, we will explain the main aspects, using the following code:.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>
    <import resource="classpath:org/codehaus/xfire/spring/
        xfire.xml"/>
    <bean id="jndiTemplate"
        class="org.springframework.jndi.JndiTemplate">
        <property name="environment">
            <props>
                <prop key="java.naming.factory.initial">
                    weblogic.jndi.WLInitialContextFactory
                </prop>
                <prop key="java.naming.provider.url">
                    t3://localhost:7001
                </prop>
            </props>
        </property>
    </bean>
    <bean id="sessionEjbProxy"
        class="org.springframework.ejb.access.
            SimpleRemoteStatelessSessionProxyFactoryBean">
        <property name="jndiName">
```

```
        <value>esb-statelessSession-TraderHome</value>
    </property>
    <property name="jndiTemplate">
        <ref bean="jndiTemplate"/>
    </property>
    <property name="resourceRef">
        <value>false</value>
    </property>
    <property name="lookupHomeOnStartup">
        <value>false</value>
    </property>
    <property name="businessInterface">
        <value>
            examples.webservices.basic.statelessSession.HelloWorldBI
        </value>
    </property>
</bean>
<bean id="webService"
    class="org.codehaus.xfire.spring.remoting.XFireExporter">
    <property name="style">
        <value>rpc</value>
    </property>
    <property name="use">
        <value>encoded</value>
    </property>
    <property name="serviceFactory">
        <ref bean="xfire.serviceFactory"/>
    </property>
    <property name="xfire">
        <ref bean="xfire"/>
    </property>
    <property name="serviceBean">
        <ref bean="sessionEjbProxy"/>
    </property>
    <property name="serviceInterface">
        <value>
            examples.webservices.basic.
            statelessSession.HelloWorldBI
        </value>
    </property>
</bean>
<bean class="org.springframework.web.servlet.handler.
    SimpleUrlHandlerMapping">
```

```
<property name="urlMap">
  <map>
    <entry key="/InvokeService">
      <ref bean="webService"/>
    </entry>
  </map>
</property>
</bean>
</beans>
```

xfire-servlet.xml defines the Spring applicationContext for XFire. The first part is defining a `jndiTemplate` pointing the application server context. `jndiTemplate` provides methods to lookup and bind objects. We are using `weblogic.jndi.WLInitialContextFactory` here, but this needs to be changed to suit your application server environment. Since our EJB is a remote stateless session bean, we can use the Spring provided `SimpleRemoteStatelessSessionProxyFactoryBean` class for producing proxies to look up and access services.

The `jndiTemplate` property supplies details to look up whereas the `businessInterface` property specifies the business interface implemented by the service. This much is enough to get a handle on the remote service and the next part is to export the service from within the XFire context, which we can do using the XFire Spring `XFireExporter`. Next the `SimpleUrlHandlerMapping` will route any requests of URL pattern, `/InvokeService`, to the above exported service.

4. **web.xml:** web.xml is as shown in the following code, which is again similar to that in the previous example.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/xfire-servlet.xml
      classpath:org/codehaus/xfire/spring/xfire.xml
    </param-value>
  </context-param>
  <listener>
    <listener-class>
      org.springframework.web.util.Log4jConfigListener
    </listener-class>
  </listener>
  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
```

```

</listener>
<servlet>
  <servlet-name>xfire</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>xfire</servlet-name>
  <url-pattern>/services/*</url-pattern>
</servlet-mapping>
</web-app>

```

5. **build.xml:** The `build.xml` requires special attention in this case because we explicitly refer to the EJB client jar. We generate this as a part of the EJB build process and later, during XFire export, we include this client jar too along with the war file as the XFire run time requires the business interface, home, and remote stubs of EJB, and other similar helper classes during service invocation.

The `build.xml` is shown as follows:

```

<target name="copyjars">
  <copy todir="${tomcat.home}/lib">
    <fileset dir="${bea.home}/weblogic812/server/lib">
      <include name="weblogic.jar"/>
    </fileset>
  </copy>
  <copy todir="${build.dir}/WEB-INF/lib">
    <fileset dir="${bea.home}/weblogic812/samples/server/
      examples/build/clientclasses">
      <include name="${ejb.client.jar}"/>
    </fileset>
  </copy>
</target>

```

Running the Sample

Running the sample involves multiple steps that are listed as follows:

- **Deploying the EJB:** To deploy the EJB, we have to follow general EJB deployment steps as specified in Weblogic documentation.

First, change directory to the EJB samples directory as shown here:

```
cd ch05\04_XFireExportAndBindEjb\ejb
```

Now set the environment variables for the build console as per the Weblogic documentation and then use the ant script provided along with the Weblogic server bundle. For that, do the following:

```
%wl.home%\samples\domains\examples\setExamplesEnv.bat
%wl.home%\server\bin\ant
```

We can even test whether our EJB deployment went fine by executing a test client, as shown here:

```
%wl.home%\server\bin\ant run
```

- **XFire export and Bind EJB:** In this step, we create the web application in .war format and deploy it in the web server. Change directory to `ch05\04_XFireExportAndBindEjb\XFireBind` and execute ant to build the web application, as shown here:

```
cd ch05\04_XFireExportAndBindEjb\XFireBind
ant
```

This will generate the war file, which in turn contains all required libraries extracted out from the XFire installation folder. Folder `dist` will contain `XFireBindEjb.war`, which should be deployed in the `webapps` folder of Tomcat (or any other relevant web server). Now, restart the server.

The web service would have been exposed by now and the service definition can be accessed using the following URL:

```
http://localhost:8080/XFireBindEjb/services/InvokeService?WSDL
```

We can now write a `Client` similar to what we have seen in the previous few examples to test the web service.

To run the client, assuming that you have already compiled the client, while building the sample, execute ant as follows:

```
ant run
```

XFire for Lightweight Integration

Almost all literature that we have been reading about XFire, is for deploying and accessing web service in a lightweight manner. Since this text is speaking on SOI, we are interested in the SOI aspects of XFire. We have demonstrated this through the examples in this chapter. Later, when we look at the `ServiceMix` examples, we will see the real power of XFire, especially the XFire Proxy classes.

Summary

Web services provide us with a means to attain SOA-based architectures. With the growing number of technology frameworks available today, it is easy to deploy web services.

XFire is a SOAP stack with which we can quickly and easily expose web services. As seen in this chapter, a combination of technology stacks like XFire and Spring can help to bind external services, including external EJB services. Once bound, these services can be re-published as web services so that they become firewall friendly. This is similar to JBI binding of services providing a mediation layer for service integration, more of which we will see in later chapters.

We now have had enough of traditional bindings and bindings using XFire. With this background it is time now to delve deep into JBI and related services. We will continue with more JBI discussions in the next chapter by understanding the JBI packaging and deployment model.

6

JBIs Packaging and Deployment

Small things matter; whether packaging and deployment are smaller concerns compared to design and development, is still under dispute. However, one thing is clear, that a standard way of packaging and deployment promotes cross-platform and cross-vendor portability of components, whether it is our familiar `.jar`, `.war`, and `.rar` files or the new JBI archive defined by the JBI specification.

ServiceMix is a container for JBI components. At the same time the ServiceMix JBI container by itself is a JBI component. This characteristic enables ServiceMix to be deployed as a standard JBI component into another vendor's ESB container, provided the host ESB container supports JBI components. This is similar to a `java.awt.Frame` which is a component that you can include in a container. At the same time, the `Frame` by itself is a container which can contain other components. ServiceMix is analogous to our `Frame` example.

When we say a ServiceMix container is a JBI component, it means that a host container (like OpenESB from Sun) can make use of almost every ServiceMix component, whether the component is a standard JBI component or a lightweight component (the difference between these two is explained later). The promise of this model is that the developer-created ServiceMix components can be reused in any other JBI container.

This chapter deals with the details of how we package and deploy JBI components into ServiceMix. We will look into the following:

- Installation, service assembly, and service unit packaging
- Standard versus lightweight JBI components in ServiceMix
- A packaging and deployment sample

Packaging in ServiceMix

ServiceMix, being JBI-compliant, follows the JBI packaging schema. JBI specification defines standard packaging for both installation of components and deployment of artifacts to those components that function as containers for other components.

Installation Packaging

JBI talks about two types of JBI components for installation—a JBI component and a `shared-library` for use by such components. The JBI installation package is a ZIP archive file. This ZIP archive has contents that are opaque to JBI except for the so called installation descriptor that must be named and located as follows:

`/META-INF/jbi.xml`

The `jbi.xml` must conform to either the component installation descriptor schema or the `shared-library` installation descriptor schema. A sample installation descriptor is shown in the following code:

```
<?xml version="1.0" encoding="utf-8"?>
<jbi version="1.0" xmlns="http://java.sun.com/xml/ns/jbi"
    xmlns:sam="http://www.binildas.com/esb/sample">
  <component type="service-engine">
    <identification>
      <name>sample-engine-1</name>
      <description>An example service engine</description>
      <sam:TypeInfo part-number="012AB490-578F-114FAA">
        BPEL:2.0:XQuery:1.0:XPath:2.0:XPath:1.0
      </sam:TypeInfo>
    </identification>
    <component-class-name description="sam">
      com.binildas.esb.Sample1
    </component-class-name>
    <component-class path>
      <path-element>Sample1.jar</path-element>
    </component-class path>
    <bootstrap-class-name>
      com.binildas.esb.Sample1Bootstrap
    </bootstrap-class-name>
    <bootstrap-class path>
      <path-element>Sample1.jar</path-element>
    </bootstrap-class path>
    <shared-library>slib1</shared-library>
    <sam:Configuration version="1.0">
      <sam:ThreadPool size="10"/>
    </sam:Configuration>
  </component>
</jbi>
```

```

        <sam:Queue1 size="50"/>
      </sam:Configuration>
    </component>
  </jbi>

```

The `jbi.xml` can include extension elements. If you observe the sample descriptor above there are two extension elements nested between the `Configuration` elements. These extension elements provide component specific information, using an XML namespace that is outside that of document elements (<http://www.binildas.com/esb/sample> in our case). They are `ThreadPool` and `Queue1`. JBI implementations and component implementations may use these extension elements to provide extra, component specific information for the use of the component, component tooling, or both. For example, you can think of an IDE which will help in JBI development and deployment, using which you can configure the characteristics of the extension elements (like the number of threads in the pool, in our sample).

Service Assembly Packaging

A Service Assembly (SA) deployment package contains opaque (to JBI) deployment artifacts, and a deployment descriptor. This deployment descriptor is named `jbi.xml` and provides information for the JBI deployment service to process the contents of the deployment pack appropriately. The SA then contains one or more Service Unit (SU) archives, all contained within a ZIP archive file. A sample deployment descriptor referring to two SUs is shown in the following code:

```

<?xml version="1.0" encoding="UTF-8"?>
<jbi xmlns="http://java.sun.com/xml/ns/jbi" version="1.0">
  <service-assembly>
    <identification>
      <name>soap-demo</name>
      <description>Soap demo</description>
    </identification>
    <service-unit>
      <identification>
        <name>engine-su</name>
        <description>Contains the service</description>
      </identification>
      <target>
        <artifacts-zip>engine-su.zip</artifacts-zip>
        <component-name>servicemix-jsr181</component-name>
      </target>
    </service-unit>
    <service-unit>
      <identification>
        <name>binding-su</name>
        <description>Contains the binding</description>

```

```
</identification>
<target>
  <artifacts-zip>binding-su.zip</artifacts-zip>
  <component-name>servicemix-http</component-name>
</target>
</service-unit>
</service-assembly>
</jbi>
```

This deployment descriptor refers to two different SUs namely `engine-su` and `binding-su`. These two SU archives, plus the descriptor itself, are combined into a single ZIP archive to form the SA. The deployment descriptor is placed inside the ZIP archive in a directory structure as given in the following, which will be explained with figures later in this chapter:

```
/META-INF/jbi.xml
```

Service Unit Packaging

The SA contains opaque (to JBI) deployment artifacts called SUs, which are again ZIP archive files. These archives contain a single JBI-defined descriptor file:

```
/META-INF/jbi.xml
```

The `jbi.xml` descriptor provides information about the services, which are statically provided and consumed as a result of deploying the SU to its target component.

The ServiceMix JBI container also supports XBean-based deployment—we can deploy SUs containing a file named `xbean.xml`. Since, at SA-level, the SUs are opaque to JBI deployment, even the SA containing XBean-based SUs can be ported across JBI containers.

Deployment in ServiceMix

JB1 components, by themselves, can act as JBI containers. Adding more artifacts to installed components is called **deployment**. ServiceMix supports two modes of deployment—standard and JBI complaint, and lightweight.

Standard and JBI compliant

Using this mode, we can install components at run time and deploy SAs onto them. These components are JBI specification compliant and hence they are JBI containers for other components too. They can accept SA deployments and are implemented using the `servicemix-common` module. Since they are JBI compliant, they are packaged as ZIP archive files with a `jbi.xml` descriptor.

Examples of a few ServiceMix standard JBI components are shown in the following list:

- `servicemix-jsr181`
- `servicemix-drools`
- `servicemix-http`
- `servicemix-jms`

We can also configure the above mentioned standard components, to be used in a static deployment mode using the `servicemix.xml` configuration file.

We can deploy lightweight components in this mode. To do that, lightweight components must be deployed to the `servicemix-lwcontainer`.

Lightweight

Lightweight components are POJO components implementing the required JBI interfaces. They don't follow standard JBI packaging; hence, do not support SU deployments. Due to this reason, they cannot normally be deployed at run time. In case we need to deploy them at run time, we can deploy them on the `servicemix-lwcontainer` (lightweight container). Lightweight components normally inherit the `ComponentSupport` class directly or indirectly, and are mainly in the `servicemix-components` module. We normally use the `servicemix.xml` static configuration file when we want to run a single application for testing purposes, or when we embed ServiceMix in a web application, for example.

A Few examples are shown in the following list:

- JDBC component
- Quartz component
- XSLT component

Packaging and Deployment Sample

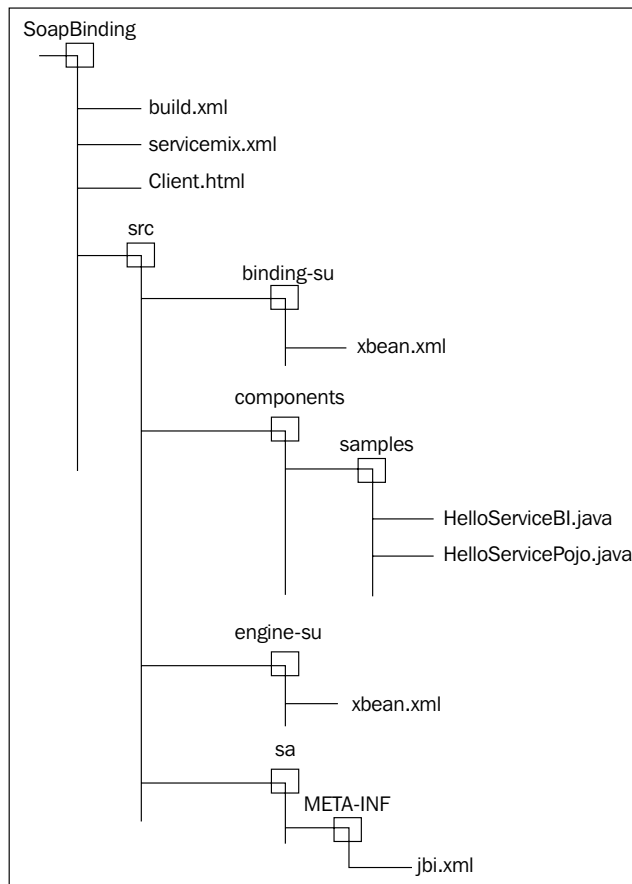
We will now create SUs and SAs, and deploy them into the ServiceMix JBI container. The whole process can be divided into two phases, with multiple steps in each phase.

The two phases in the process are the following:

- Component development
- Component packaging

Phase One—Component Development

This phase includes coding and building the code base. We have the code base split up into multiple folders each representing separate artifacts, which go into the final SAs as shown in the following figure.



The components folder is supposed to contain lightweight (or POJO) components, but in our case, we have a simple service class (`HelloServicePojo`) implementing a BI (`HelloServiceBI`), this is shown in the following code:

```
public interface HelloServiceBI
{
    String hello(String phrase);
}
public class HelloServicePojo implements HelloServiceBI
{
    private static long times = 0;
```

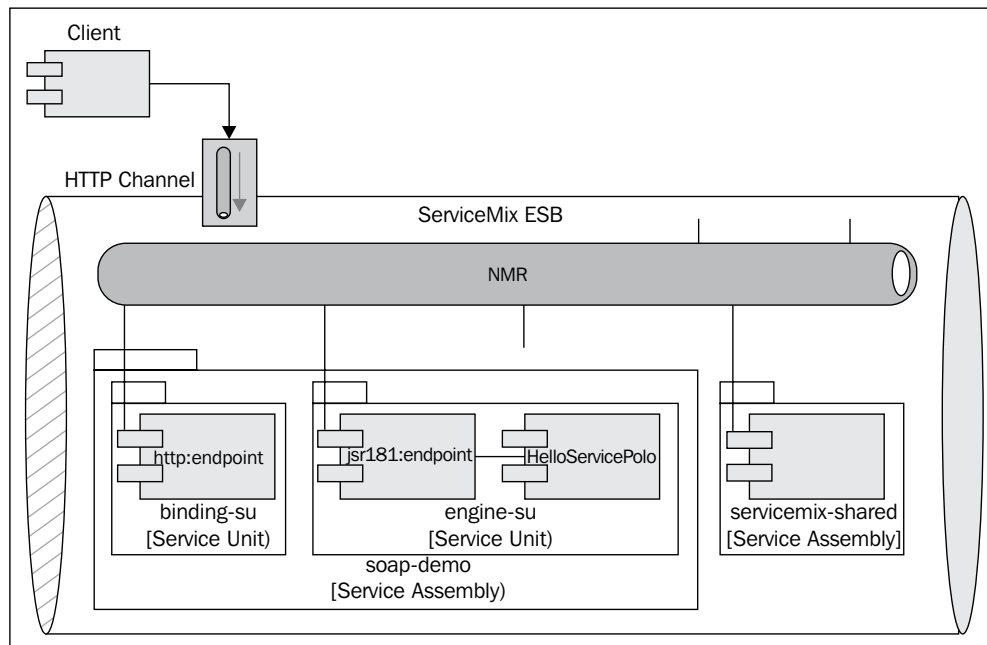
```

public String hello(String phrase)
{
    System.out.println("HelloServiceBean.hello{
        " + (++times) + "}...");
    return "From HelloServiceBean :HELLO!! You just said :" + phrase;
}
}

```

Phase Two—Component Packaging

We will now have two Service Units—one a SE and the other a BC. Both these Service Units will be packaged into a single Service Assembly so that we can deploy them into the ESB. The servicemix-shared Service Assembly provides common services and hence we will also deploy that too into the ESB. The full setup is as shown in the following figure:



We will now look into the packaging of individual SUs and SAs.

The engine-su (Service Engine Service Unit) is based on `xbean.xml`, which leverages `jsr181:endpoint` to expose our service class as a web service. This is shown in the following code:

```
<?xml version="1.0"?>
<beans xmlns:jsr181="http://servicemix.apache.org/jsr181/1.0"
       xmlns:demo="http://binildas.com/esb/sample">
  <classpath>
    <location>./</location>
  </classpath>
  <jsr181:endpoint.pojoClass="samples.HelloServicePojo"
                annotations="none"
                service="demo:hello-service"
                endpoint="hello-service"
                serviceInterface="samples.HelloServiceBI" />
</beans>
```

The binding-su (Binding Component Service Unit) is also based on `xbean.xml`, which leverages `http:endpoint` in the consumer (consumer to NMR) role. This is shown in the following code:

```
<?xml version="1.0"?>
<beans xmlns:http="http://servicemix.apache.org/http/1.0"
       xmlns:demo="http://binildas.com/esb/sample">
  <http:endpoint service="demo:hello-service"
                endpoint="hello-service"
                role="consumer"
                locationURI="http://localhost:8192/Service/"
                defaultMep="http://www.w3.org/2004/08/wsdl/in-out"
                soap="true" />
</beans>
```

The two Service Units can be now packaged into a single Service Assembly. So, these two SUs are configured in the `.xml` file shown as follows:

`/META-INF/jbi.xml`

The contents of `jbi.xml` file are as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<jbi xmlns="http://java.sun.com/xml/ns/jbi" version="1.0">
  <service-assembly>
    <identification>
      <name>soap-demo</name>
      <description>Soap demo</description>
    </identification>
    <service-unit>
```

```

    <identification>
      <name>engine-su</name>
      <description>Contains the service</description>
    </identification>
    <target>
      <artifacts-zip>engine-su.zip</artifacts-zip>
      <component-name>servicemix-jsr181</component-name>
    </target>
  </service-unit>
</service-unit>
<service-unit>
  <identification>
    <name>binding-su</name>
    <description>Contains the binding</description>
  </identification>
  <target>
    <artifacts-zip>binding-su.zip</artifacts-zip>
    <component-name>servicemix-http</component-name>
  </target>
</service-unit>
</service-assembly>
</jbi>

```

As is evident, the Service Assembly encapsulates two Service Units – engine-su and binding-su.

Thus, the Service Unit is a ZIP archive that will contain your application's Java class files and the servicemix.xml configuration file. The Service Unit can also contain a jbi.xml file which provides information about services statically provided and consumed. The name of the ZIP archive that we create here is later referred to from the jbi.xml file of the enclosing Service Assembly.

Building and packaging can be automated using the ant build tool with the help of build.xml, shown in the following code:

```

<project name="jms-binding" default="setup" basedir=".">
  <target name="build-components" depends="init">
    <javac srcdir="${comp.src.dir}" destdir="${comp.build.dir}">
      <classpath refid="javac.classpath"/>
    </javac>
  </target>
  <target name="build-engine-su" depends="build-components">
    <zip destfile="${build.dir}/engine-su.zip">
      <fileset dir="${comp.build.dir}"/>
      <fileset dir="${su.engine.src.dir}"/>
    </zip>
  </target>
</project>

```

```
</target>
<target name="build-binding-su">
  <zip destfile="${build.dir}/binding-su.zip">
    <fileset dir="${su.binding.src.dir}"/>
  </zip>
</target>
<target name="build-sa"
  depends="build-engine-su,build-binding-su">
  <zip destfile="${build.dir}/soap-demo-sa.zip">
    <fileset dir="${build.dir}" includes="engine-su.zip"/>
    <fileset dir="${build.dir}" includes="binding-su.zip"/>
    <fileset dir="${sa.src.dir}"/>
  </zip>
</target>
<target name="setup" depends="build-sa">
  <mkdir dir="${install.dir}"/>
  <mkdir dir="${deploy.dir}"/>
  <copy todir="${install.dir}">
    <fileset dir="${servicemix.home}/components"
      includes="*jsr181*"/>
    <fileset dir="${servicemix.home}/components"
      includes="*http*"/>
    <fileset dir="${servicemix.home}/components"
      includes="*servicemix-shared*"/>
  </copy>
  <copy file="${build.dir}/soap-demo-sa.zip"
    todir="${deploy.dir}"/>
</target>
</project>
```

In the setup target we can see that we copy the `jsr181` component, `http` component, and `servicemix-shared` from the ServiceMix installation path to our target install directory. These are standard JBI components onto which we are deploying the SU artifacts. When we do that, if ServiceMix is already running, it will detect the file present there and start it.

Running the Packaging and Deployment Sample

As a first step and if you haven't done it before, edit `examples.PROPERTIES` provided along with the code download for this chapter and change the paths there to match your development environment. The code download for this chapter also includes a `README.txt` file, which gives detailed steps to build and run the samples.

Now we need to build the samples. For this, change directory to `ch06\SoapBinding` and then execute `ant`, as shown here:

```
cd ch06\SoapBinding
ant
```

This will build the entire sample.

Now we are going to run this example as standalone. That is, ServiceMix will be started and then the SOAP demo is deployed and run. This is done by executing the `servicemix.xml` file found in the topmost folder (`ch06\SoapBinding`).

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:sm="http://servicemix.apache.org/config/1.0">
  <bean id="propertyConfigurer"
        class="org.springframework.beans.factory.
              config.PropertyPlaceholderConfigurer">
    <property name="location"
              value="classpath:servicemix.properties" />
  </bean>
  <import resource="classpath:jmx.xml" />
  <import resource="classpath:jndi.xml" />
  <import resource="classpath:security.xml" />
  <import resource="classpath:tx.xml" />
  <sm:container id="jbi"
                rootDir="./wdir"
                installationDirPath="./install"
                deploymentDirPath="./deploy"
                flowName="seda">
    <sm:activationSpecs>
    </sm:activationSpecs>
  </sm:container>
</beans>
```

We can bring up ServiceMix by running the following commands:

```
cd ch06\SoapBinding
%SERVICEMIX_HOME%\bin\servicemix servicemix.xml
```

When we start ServiceMix, the JBI container is configured using the above `servicemix.xml` file.

To run the demo, there is a `Client.html` file provided again in the top folder. The Client will send the following request:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV=
                    "http://schemas.xmlsoap.org/soap/envelope/"
                    xmlns:xsi=
                    "http://www.w3.org/2001/XMLSchema-instance"
                    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

```
<SOAP-ENV:Body>
  <ns1:hello xmlns:ns1="http://binildas.com/esb/sample"
    SOAP-ENV:encodingStyle=
      "http://schemas.xmlsoap.org/soap/encoding/">
    <phrase xsi:type="xsd:string">what&apos;s your name?
    </phrase>
  </ns1:hello>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The client sends the above SOAP request to `http://localhost:8192/Service/`. As, we already configured ServiceMix `http:endpoint` listening in port 8192, the SOAP request will be intercepted and sent to the JBI NMR. `http:endpoint` has the consumer role when it sends a SOAP request to the NMR and it will be routed to `http:endpoint` listening in port 8192. This service is the JSR181 binding component for our service implementation class, which when invoked will carry out the business functionality and any response is returned through a similar channel back to the client.

Summary

We use multiple archive formats for various J2EE components — `.jar`, `.war`, `.ear`, and `.rar` are few amongst them. Now, JBI specification recognizes `.zip` as a valid archive format for JBI components. SUs and SAs are packaged as valid `.zip` files and are deployed into JBI compliant containers. In this chapter, we have seen how to write code from scratch, package it into standard JBI formats, and deploy it to the ESB run time. We will follow similar packaging for most of our samples in the subsequent chapters.

The next chapter will teach how to custom code JBI components on our own, so that they can take part in the message exchanges happening through the JBI bus.

7

Developing JBI Components

Until now, you have been assembling JBI components which are pre-built and available along with the ServiceMix installation in your own way. The aim of this chapter is to get started in developing your own components and deploying them into the ServiceMix JBI container. In this chapter, we will visit the core API from ServiceMix as well as from the JBI specification, which will function as useful helper classes using lightweight components. We will also create our own JBI component and deploy it into the JBI bus.

Developing JBI components in ServiceMix requires the developer to write a bit of plumbing code, which may or may not be so appealing. To make the developer's life easy, ServiceMix supports Spring-based POJO classes and their configuration. ServiceMix also provides many component helper classes. It is up to the developer whether to make use of these helper classes, or develop their own components or code against the core APIs of ServiceMix and JBI, though there is no reason why anyone shouldn't be using the helper classes.

So, we will cover the following in this chapter:

- Need for custom JBI components
- ServiceMix component helper classes
- Create, deploy, and run JBI components
- Build and run a sample

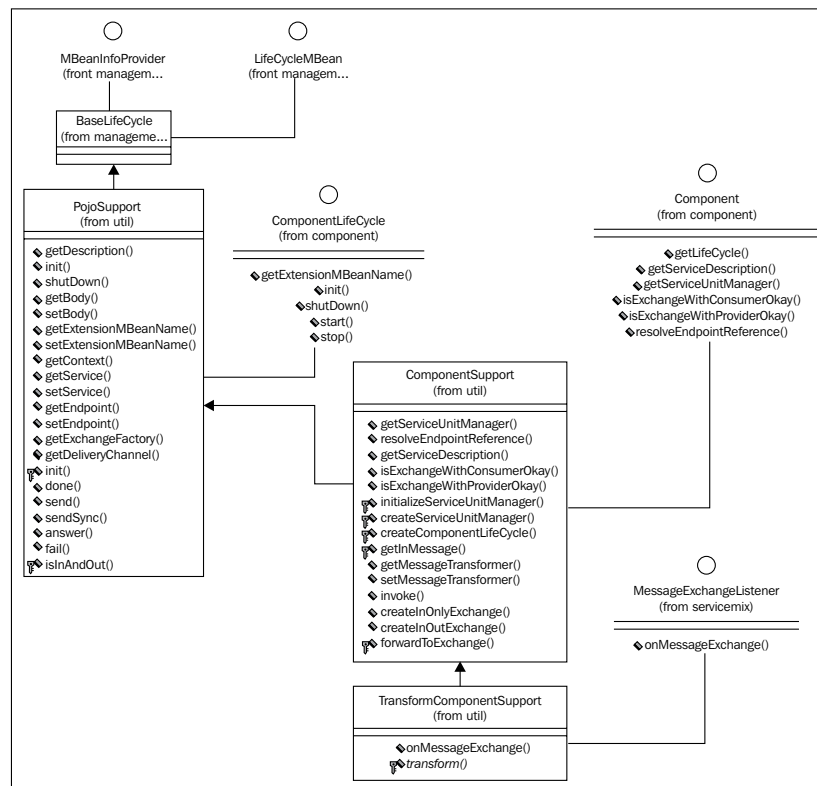
Need for Custom JBI Components

We have already seen many ServiceMix components, both JBI compliant and lightweight. Many of them are available inside the `components` folder in the ServiceMix installation. A natural query in this context is "Why does one need to develop custom components in ServiceMix?". Often I see myself as a lazy programmer, trying to reuse codes, components, or services rather than hand code them every time.

Going by that trend, I am also compelled to look at ServiceMix with an aim to reuse anything and everything, and not to hand code anything. The proposition looks fine, but the truth is that the world is not ideal all the time. In an ideal world, we will have all the ServiceMix components, tools, and hooks available, so that we can assemble or reconfigure the components to suit our needs. This makes sense in a pure integration initiative where we only integrate, and do not develop anything—after all, JBI and ServiceMix is all about integration. However, since the world is far from ideal and all of us will have our own integration problems and scenarios. Often, we will have to custom code integration wrappers, message orchestration logic, or error handling logic. This logic can be coded into components created out of ServiceMix helper classes and easily deployed into a ServiceMix container.

ServiceMix Component Helper Classes

ServiceMix provides a lot of plumbing code in the form of reusable component helper classes. Most of these classes are abstract base classes from which the developer can derive their own classes to put custom logic. The main classes of interest and their relationship are shown in the following figure:



MessageExchangeListener

MessageExchangeListener is a ServiceMix package interface, very similar to the JMS MessageListener. This interface is shown in the following code:

```
package org.apache.servicemix;
import javax.jbi.messaging.MessageExchange;
import javax.jbi.messaging.MessagingException;
public interface MessageExchangeListener
{
    /**
     * MessageExchange passed directly to the listener
     * instead of being queued
     *
     * @param exchange
     * @throws MessagingException
     */
    public void onMessageExchange(MessageExchange exchange)
        throws MessagingException;
}
```

When our custom components implement this interface, we will be able to receive new message exchanges easily, rather than writing a new thread. The default JBI asynchronous dispatch model is where a thread is used per JBI component by the container. However, when the component implement the MessageExchangeListener interface, the ServiceMix container will detect the use of this interface and be able to perform immediate dispatch.

TransformComponentSupport

TransformComponentSupport contains a default implementation for the onMessageExchange method. The code is more explanatory than a textual description of the sequences. Hence, let's reproduce that in the following code:

```
public void onMessageExchange(MessageExchange exchange)
{
    if (exchange.getStatus() == ExchangeStatus.DONE)
    {
        return;
    }
    else if (exchange.getStatus() == ExchangeStatus.ERROR)
    {
        return;
    }
    try
    {
```



```
InOnly outExchange = null;
NormalizedMessage in = getInMessage(exchange);
NormalizedMessage out;
if (isInAndOut(exchange))
{
    out = exchange.createMessage();
}
else
{
    outExchange = getExchangeFactory().createInOnlyExchange();
    out = outExchange.createMessage();
}
boolean txSync = exchange.isTransacted() && Boolean.TRUE.
    equals(exchange.getProperty(JbiConstants.SEND_SYNC));
copyPropertiesAndAttachments(exchange, in, out);
if (transform(exchange, in, out))
{
    if (isInAndOut(exchange))
    {
        exchange.setMessage(out, "out");
        if (txSync)
        {
            getDeliveryChannel().sendSync(exchange);
        }
        else
        {
            getDeliveryChannel().send(exchange);
        }
    }
    else
    {
        outExchange.setMessage(out, "in");
        if (txSync)
        {
            getDeliveryChannel().sendSync(outExchange);
        }
        else
        {
            getDeliveryChannel().send(outExchange);
        }
    }
}
```

```

        exchange.setStatus(ExchangeStatus.DONE);
        getDeliveryChannel().send(exchange);
    }
}
else
{
    exchange.setStatus(ExchangeStatus.DONE);
    getDeliveryChannel().send(exchange);
}
}
catch (Exception e)
{
    try
    {
        fail(exchange, e);
    }
    catch (Exception e2)
    {
        logger.warn("Unable to handle error: " + e2, e2);
        if (logger.isDebugEnabled())
        {
            logger.debug("Original error: " + e, e);
        }
    }
}
}
}

```

The above code calls `transform(exchange, in, out)`. By doing so, it is invoking the abstract method in `TransformComponentSupport`. The abstract method is reproduced as follows:

```

protected abstract boolean transform(MessageExchange exchange,
                                   NormalizedMessage in, NormalizedMessage out)
                                   throws Exception;

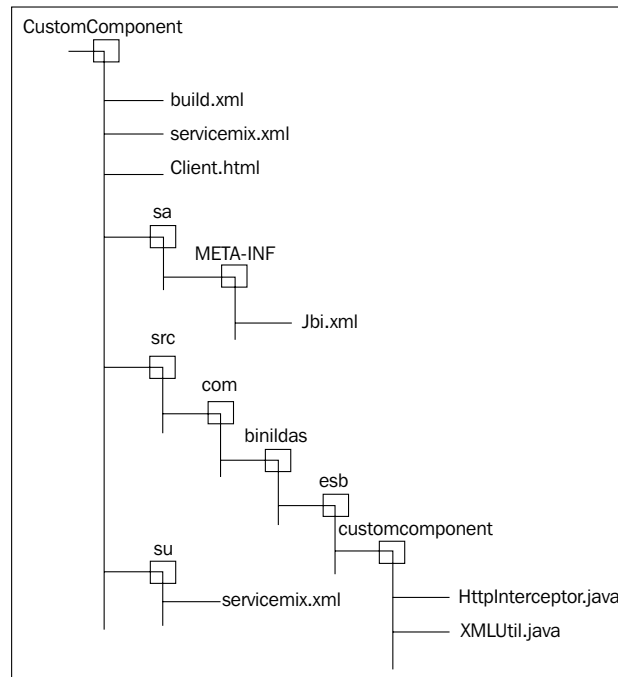
```

Hence, it is easy to just implement the `transform` method in the custom code and let the helper class work behind the scenes for all message exchanges.

`ComponentSupport` and `PojoSupport` are the other two helper classes in the hierarchy.

Create, Deploy, and Run JBI Component

This section covers from end to end, the creation, deployment, and running of a sample JBI component in ServiceMix. Following the basics demonstrated in this chapter, readers can code JBI components to suit their own requirements. We will also see many such components in the examples in coming chapters. The code for this sample is kept in `ch07\CustomComponent` folder.



Code HttpInterceptor Component

`HttpInterceptor`, as the name implies, will intercept messages coming in the HTTP channel. It then prints out the message to the console, and sends some message back through the response channel. To quickly code our component, we will extend the ServiceMix component helper class namely `TransformComponentSupport`.

This is demonstrated in the following code:

```
public class HttpInterceptor extends TransformComponentSupport
{
    public HttpInterceptor() {}
    protected boolean transform(MessageExchange exchange,
        NormalizedMessage in, NormalizedMessage out) throws
        MessagingException
```

```

    {
        NormalizedMessage copyMessage = exchange.createMessage();
        getMessageTransformer().transform(exchange, in, copyMessage);
        Source content = copyMessage.getContent();
        System.out.println("HttpInterceptor.transform02.
                           content = " + content);
        String contentString = null;
        if (content instanceof DOMSource)
        {
            contentString = XMLUtil.node2XML(((DOMSource)
                                             content).getNode());
            System.out.println("HttpInterceptor.transform03.
                               contentString = " + contentString);
        }
        out.setContent(new StringSource("<?xml version=\"1.0\"
                                         encoding=\"UTF-8\"?>
                                         <Response>Response From Server</Response>"));
        return true;
    }
}

```

As we described earlier, we will implement the abstract transform method in TransformComponentSupport. The Transform method will have a reference to the Message Exchange and also to the in and out Normalized Messages. We first copy the contents from the in message and print that out to the console. Then, we create a new StringSource with some message from the server, sending that to the out Normalized Messages. The code is as simple as that, since most of the plumbing has already been done for you by the base class methods.

Configure HttpInterceptor Component

As, we can configure the HttpInterceptor component as SU, we will do the configuration of this component as a SU in the servicemix.xml file kept at ch07\CustomComponent\suservicemix.xml.

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:sm="http://servicemix.apache.org/config/1.0"
       xmlns:demo="http://www.binildas.com/esb/customcomponent">
    <classpath>
        <location>./</location>
    </classpath>
    <sm:serviceunit id="jbi">
        <sm:activationSpecs>
            <sm:activationSpec componentName="interceptor"
                               endpoint="interceptor"
                               service="demo:interceptor">

```

```
<sm:component>
  <bean class=
    "com.binildas.esb.customcomponent.HttpInterceptor"/>
  </sm:component>
</sm:activationSpec>
</sm:activationSpecs>
</sm:serviceunit>
</beans>
```

This is different from the XBean-based packaging example, that we saw in the previous example. As, we are using `servicemix.xml` to specify the SU. Thus, we can see that there are multiple ways by which we can configure and package our SUs.

Package HttpInterceptor Component

In Chapter 6, we saw how to package and deploy components in ServiceMix. We will follow the same packaging methodology here. We will create an SU and then package it into an SA.

We have already seen the SU configuration, let us now look into the SA configuration as follows:

`\sa\META-INF\jbi.xml`

The content of the `Jbi.xml` file is shown in the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<jbi xmlns="http://java.sun.com/xml/ns/jbi" version="1.0">
  <service-assembly>
    <identification>
      <name>InterceptorAssembly</name>
      <description>Interceptor Service Assembly</description>
    </identification>
    <service-unit>
      <identification>
        <name>Interceptor</name>
        <description>Interceptor Service Unit</description>
      </identification>
      <target>
        <artifacts-zip>Interceptor-su.zip</artifacts-zip>
        <component-name>servicemix-lwcontainer</component-name>
      </target>
    </service-unit>
  </service-assembly>
</jbi>
```

Here, we package the `Interceptor Service Unit` into the SA. The main point to be noted in the `SA jbi.xml` is the `target` element of the `service-unit`. Here, we specify that the SU artifact (that is, `Interceptor-su.zip`) is to be deployed into the `servicemix-lwcontainer` target container.

Deploy HttpInterceptor Component

To deploy the `HttpInterceptor` component, we have a `servicemix.xml` file in the `topmost` folder to start the `ServiceMix` container.

The content of the `servicemix.xml` file is shown in the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:sm="http://servicemix.apache.org/config/1.0"
       xmlns:demo="http://www.binildas.com/esb/customcomponent">
  <bean id="propertyConfigurer"
        class="org.springframework.beans.factory.config.
              PropertyPlaceholderConfigurer">
    <property name="location"
              value="classpath:servicemix.properties" />
  </bean>
  <import resource="classpath:jmx.xml" />
  <import resource="classpath:jndi.xml" />
  <import resource="classpath:security.xml" />
  <import resource="classpath:tx.xml" />
  <sm:container id="jbi"
                rootDir="./wdir"
                installationDirPath="./install"
                deploymentDirPath="./deploy"
                flowName="seda"
                monitorInstallationDirectory="true"
                createMBeanServer="true"
                useMBeanServer="true">
    <sm:activationSpecs>
      <sm:activationSpec componentName="httpReceiver"
                        service="bt:httpBinding"
                        endpoint="httpReceiver"
                        destinationService="demo:interceptor">
        <sm:component>
          <bean class="org.apache.servicemix.components.
                    http.HttpConnector">
            <property name="host" value="127.0.0.1"/>
            <property name="port" value="8912"/>
          </bean>
        </sm:component>
      </sm:activationSpec>
    </sm:activationSpecs>
  </sm:container>
</beans>
```

The first thing to note here is that we have `installationDirPath="./install"` for the JBI container. This is where we place our JBI components, which are to be installed by ServiceMix. Hence, they can act as containers for components like our `HttpInterceptor`.

It is also worth noting that we have an `HttpConnector` configured with our sample, and `HttpInterceptor`, as the `destinationService` so that we have a way to test our sample too.

Build and Run the Sample

As a first step and if you haven't done it before, edit `examples.PROPERTIES` provided along with the code download for this chapter and change the paths there to match your development environment. The code download for this chapter also includes a `README.txt` file, which gives detailed steps to build and run the samples.

The `build.xml` is as usual but we will list the major differences here.

As, we want `HttpInterceptor SU` to be deployed into `servicemix-lwcontainer` while building the code base, we will also copy the `servicemix-lwcontainer` from ServiceMix installation to `installationDirPath`.

```
<copy todir="${install.dir}" overwrite="true">
  <fileset dir="${servicemix.home}/components"
    includes="*lwcontainer*" />
  <fileset dir="${servicemix.home}/components" includes="*shared*" />
</copy>
```

Execute ant to build the sample as shown as follows:

```
cd ch07\CustomComponent
ant
```

We can bring up ServiceMix by running the following commands:

```
cd ch07\CustomComponent
%SERVICEMIX_HOME%/bin/servicemix servicemix.xml
```

When we start ServiceMix, the JBI container is configured using the above `servicemix.xml` file. To run the demo, there is a `Client.html` file provided again in the top folder.

Summary

In this chapter, we looked at the core API from ServiceMix as well as from the JBI specification, which will function as useful helper classes using which we can develop lightweight components quickly.

We have also custom coded a JBI component and deployed it into the JBI bus. You may not want to always custom code components. Many times, JBI components will be available as off-the-shelf-libraries. Such components can take part in the message exchanges through the ESB and can provide integration with external services like CICS and CORBA, for example. If by any chance you want to create your own JBI components, then you can follow the guidelines presented in this chapter as a starting point.

As we have covered the standard JBI packaging and deployment model in Chapter 6, we now have enough toolsets to delve deep into JBI and ESB. We will continue our journey by looking at using Spring beans to Spring-wrap an EJB service onto the JBI bus in the next chapter. By doing so, we will expose EJB as a WSDL compliant service across firewalls – yes really, we are going to do that! Hence, don't scrap your existing investments in EJB till you cover the next chapter.

8

Binding EJB in a JBI Container

EJB is the distributed component paradigm in the Java-J2EE world. EJB proved not to be a push button solution for programming problems. Still, a few of the promises (of course, reality too) – distributed transaction propagation, component-based deployment model, and interface-based design, proved to be really useful. Today, we have been talking about lightweight containers and aspect-based programming, and whether EJB still holds the crown is something which has to be answered on a case by case basis. Being neither a proponent nor an opponent of EJB, one thing I have to admit is that the industry has a lot invested in this technology. Scraping all these investments and implementing alternate solutions is surely not a topic for our discussion, at least in this text book. For our SOI-based discussion, perhaps, it is more interesting to look at how to reuse those existing investments. Hence, we can continue building newer system based on higher levels of SOA, maturity coexisting with old functionality. In clearer terms, coexisting services and components.

We will cover the following in this chapter:

- Component versus services.
- Indiscrimination at consumer perspective.
- Stepwise binding EJB sample.
- Reconciling EJB resources.

Component versus Services

In a chapter like this, it makes sense to discuss the difference between components and services.

Components are first-class deployable units packaged into standard artifacts. Components live in some container; there is component-container interaction for component lifecycle management and event notification. Thus, components are physical units. Components will also assume explicitly about the protocol and format of data sent over the wire, since most of the time they are technology-dependent.

Services on the other hand have URL addressable functionality, accessible over the network. At least from the consumer perspective, there are no lifecycle activities associated with the service, since services are stateless and idempotent. Between any two service invocations, there is no state maintained at the provider-level (of course, there are stateful services too). Thus services are virtual distributed components, which are neutral about the protocol and format of sending messages. By neutral, we mean it doesn't matter what the underlying technology or platform is as long as the provider and consumer agrees to a common protocol and format.

Coexisting EJB Components with Services

We need to devise a mechanism by which we can allow coexistence of components and services. By coexistence, we mean a consumer should be able to consume a web service and an EJB service without any difference. Traditional component developers, who also know how to develop a web service, will first wonder how this can be possible. Both have completely different consumption mechanisms – a web service is character-oriented whereas an EJB service is binary-oriented, just one of the many mismatches.

ESB Binding Components (BC) comes to the rescue. By judicious combination of BCs, even an EJB component can be hooked to the NMR so that any consumer, whether they are in a different technology like .NET or Mainframe, can interoperate. Of course, this is not something new, since we have been doing this for years using CORBA or COBOL Copy Books, but never in a standard way. Now JBI promises the same so that interoperability of our components across multiple JBI containers is no longer a dream.

The advantage of this kind of interoperability is multifold. First, even an organization with less SOA maturity implementations can quick start their enterprise efforts, and when they do so they can reuse existing IT assets. This is different from the Bing-Bang approach, which will reduce the overall risk considerably. Secondly, developers will be ready to accept the change since change is not abrupt – change will still harmonize their past intellectual spending in terms of time and effort for developing all those past (EJB) components.

Indiscrimination at Consumer Perspective

When we want to coexist components and services together, technical indiscrimination is very important. As a consumer can now use the same set of tools and methodologies to access functionality, whether it is a (web) service or an EJB component. If you have programmed an EJB or web service before, you will agree that the contract models are different. In the case of an EJB, the consumer-provider contract is the EJB home and EJB remote interface (better is the case with the CORBA world, where the

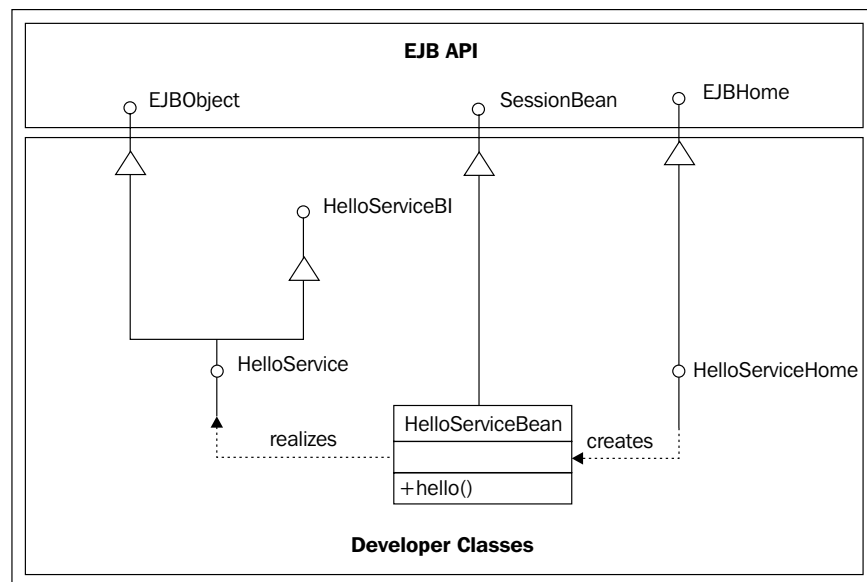
interface is the IDL). Moreover, these interfaces don't make any sense to any consumer who doesn't speak Java. However, in a web service the contract is the WSDL, which is supposed to be interpreted by any consumer. To indiscriminate, we will need to provide the same interface as the web service and see how easy it is to do with the help of an ESB.

Binding EJB Sample

We will not spend too much time describing how to write and deploy an EJB component, since there are a lot of books and resources available which will do just that. However, we will spend some time looking at how we can use Spring beans to Spring-wrap an EJB service. More time will be spent on actual binding of EJB and related discussion. As usual, we will do this sample in a step by step manner.

Step One—Define and Deploy the EJB Service

The EJB service we implement is very simple; the classes and interfaces involved are shown in the following figure:



We need to abstract out the interface from all EJB specific details; hence, we have followed the BI pattern to define the interface. HelloServiceBI is the BI, which is void of any EJB specific API.

```
package samples;
public interface HelloServiceBI
{
    String hello(String phrase) throws java.io.IOException;
}
```

As a first step and if you haven't done it before, edit `examples.PROPERTIES` provided along with the code download for this chapter and change the paths there to match your development environment. The code download for this chapter also includes a `README.txt` file, which gives detailed steps to build and run the samples.

For the sample deployment, we will use BEA Weblogic server's example domain. Hence, all necessary deployment descriptors for the Weblogic EJB container are provided in the respective folders. In case you need to deploy the EJB into a different vendor's EJB container, change the deployment descriptors accordingly.

Execute the following scripts in the command prompt to bring up the server first:

```
cd %BEA_HOME%/weblogic812/samples/domains/examples
%BEA_HOME%/weblogic812/samples/domains/examples/startExamplesServer
```

To build and deploy the EJB, in a different command prompt change directory to:

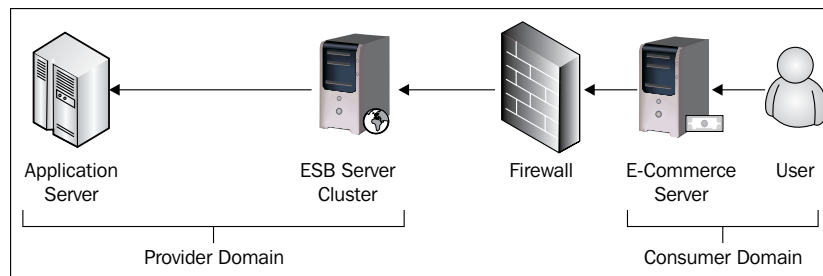
```
ch08\BindEjb\01_Ejb
%BEA_HOME%/weblogic812/samples/domains/examples/setExamplesEnv
%BEA_HOME%/weblogic812/server/bin/ant
```

This will build, package, and deploy the EJB module into the Weblogic server. It will also create a client jar containing all the client-side stubs, which is required later for accessing the service. In case you need to test whether your deployment went fine, there is a client provided which you can execute by typing the following code:

```
ch08\BindEjb\01_Ejb
ant run
```

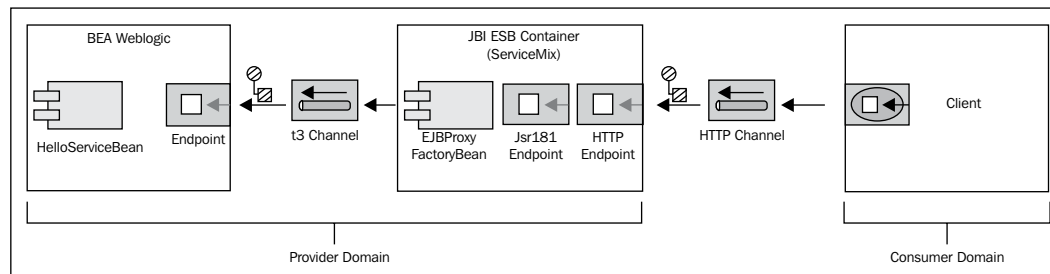
Step Two—Bind EJB to ServiceMix

Once the EJB service is up and running, we now want to expose the service across firewalls in a technology neutral format and protocol. By doing so, we can make the EJB service consumable by other B2B partners. The deployment diagram for an ESB-based solution for the sample scenario is shown in the following figure:



Here, the Application Server will host our EJB component services. The ESB server (cluster) in front of the Application Server will host the required BCs to proxy the EJB service behind the scenes as firewall-friendly services, consumable by other B2B clients.

In Chapter 6, you have already seen how we can leverage the `servicemix-jsr181` standard JBI component as a container for custom POJO classes, so that the methods defined in a POJO are exposed as services. We need to follow a similar approach to bind an EJB service too to the JBI bus, but the configuration is slightly different. The sample binding scenario inside the bus is represented in the following figure:



Let us now bind the EJB service to the JBI bus using the `servicemix-jsr181` component. `servicemix-jsr181` can be configured in the lightweight mode too. Let us look into the `servicemix.xml` file for that, as shown in the following code:

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:sm="http://servicemix.apache.org/config/1.0"
  xmlns:eip="http://servicemix.apache.org/eip/1.0"
  xmlns:http="http://servicemix.apache.org/http/1.0"
  xmlns:jsr181="http://servicemix.apache.org/jsr181/1.0"
  xmlns:my="http://binildas.com/esb/bindejb">
  <bean id="propertyConfigurer"
    class="org.springframework.beans.factory.config.
      PropertyPlaceholderConfigurer">
  
```

```
<property name="location"
          value="classpath:servicemix.properties" />
</bean>
<import resource="classpath:jmx.xml" />
<import resource="classpath:jndi.xml" />
<import resource="classpath:security.xml" />
<import resource="classpath:tx.xml" />
<sm:container id="jbi"
              useMBeanServer="true"
              createMBeanServer="true">
  <sm:activationSpecs>
    <sm:activationSpec>
      <sm:component>
        <http:component>
          <http:endpoints>
            <http:endpoint
              service="my:HelloServiceBIService"
              endpoint="HelloServiceBI"
              role="consumer"
              defaultOperation="hello"
              targetService="my:jsrEjbEP"
              targetEndpoint="jsrEjbEP"
              locationURI="http://localhost:8192/Services/
                          HelloWebService"
              soap="true"
              defaultMep="http://www.w3.org/2004/08/wsdl/
                          in-out" />
            </http:endpoints>
          </http:component>
        </sm:component>
      </sm:activationSpec>
      <sm:activationSpec componentName="jsrEjbBC"
                        service="my:jsrEjbBC"
                        endpoint="jsrEjbBC">
        <sm:component>
          <jsr181:component>
            <jsr181:endpoints>
              <jsr181:endpoint annotations="none"
                                service="my:jsrEjbEP"
                                endpoint="jsrEjbEP"
                                serviceInterface="samples.
                                                  HelloServiceBI">
                <jsr181:pojo>
                  <bean class="org.springframework.ejb.
                        access.SimpleRemoteStatelessSession
                        ProxyFactoryBean">
                    <property name="jndiName"
                              value=
                                "sample-statelessSession-
                                TraderHome"/>
                  </bean>
                </jsr181:pojo>
              </jsr181:endpoint>
            </jsr181:endpoints>
          </jsr181:component>
        </sm:component>
      </sm:activationSpec>
    </sm:activationSpecs>
  </sm:container>
</sm:container>
```

```

        <property name="businessInterface"
            value="samples.
                HelloServiceBI"/>
        <property name="jndiTemplate">
            <ref bean="jndiTemplate"/>
        </property>
    </bean>
</jsr181:pojo>
</jsr181:endpoint>
</jsr181:endpoints>
</jsr181:component>
</sm:component>
</sm:activationSpec>
</sm:activationSpecs>
</sm:container>
<bean id="jndiTemplate"
    class="org.springframework.jndi.JndiTemplate">
    <property name="environment">
        <props>
            <prop key="java.naming.factory.initial">
                weblogic.jndi.WLInitialContextFactory
            </prop>
            <prop key="java.naming.provider.url">
                t3://localhost:7001
            </prop>
        </props>
    </property>
</bean>
</beans>

```

We will now discuss this code section by section.

JNDI context is the starting point in getting a reference to an EJB resource. In the previous step, we have deployed our EJB service to the Weblogic server. To access WebLogic's JNDI tree, you need to establish a standard JNDI InitialContext representing the context root of the server's directory service. For this, the `PROVIDER_URL` property specifies the URL of the server, whose JNDI tree we want to access and the `INITIAL_CONTEXT_FACTORY` property contains the fully qualified class name of WebLogic's context factory.

If you need to access WebLogic's JNDI tree, it is advised to establish the context using the class name `weblogic.jndi.WLInitialContextFactory`, so that we can use the Weblogic specific `t3` protocol which is optimized for accessing services. We define these details by configuring the `org.springframework.jndi.JndiTemplate` bean.

The next step is to define our POJO class. The POJO class here is not a service class, but a client-side proxy to the remote EJB service. There's a lot of back end plumbing happening here when we use `SimpleRemoteStatelessSessionProxyFactoryBean`, courtesy of the Spring AOP framework. The POJO bean definition creates a proxy for the remote stateless EJB, which implements the business method interface. The EJB remote home is cached on startup, so there's only a single JNDI lookup. Each time the EJB is invoked, the proxy invokes the corresponding business method on the EJB. All the context details necessary for contacting the server are retrieved from the previous `jndiTemplate` bean.

Once our POJO bean is ready, it is just a matter of wrapping that up in the `servicemix-jsr181` as shown in the configuration above. When you configure the `jsr181:endpoint`, you can also specify the `serviceInterface`. This will hold the class name of the interface to expose as a service and this abstraction will help the binding to be exposed in standard ways (using WSDL), which we will explore shortly.

We will appreciate the fact that EJB components are remotable, but they normally expose binary interfaces. Binary protocols are not platform agonistic, nor are they firewall-friendly. Hence, EJB components are normally preferred on a LAN where we have perfect control of the network, so that nothing hinders marshalling bytes across domains. If we have to make them firewall-friendly, we can tunnel them through a different protocol, which by itself is firewall-friendly. Weblogic and other similar servers provide easy tunneling options but still the mechanism is not a standard approach. The second method is to wrap EJB with web services. This is also made easy nowadays by application servers, it is just a matter of enabling options during deployment time.

When we have an ESB, an option available to us is to leverage the protocol or format conversion capability of the ESB to make EJB invocations firewall-friendly. `ServiceMix servicemix-http` is exactly what we need here. By connecting a `servicemix-http` channel in serial and in front of the `servicemix-jsr181`, we can make the EJB component service available outside the firewall as a normal web service! What more, we can even run our entire normal web services client toolkit to retrieve WSDL, and then to generate client-side proxy classes so that it is easy to invoke functionality. The `targetService` and the `targetEndpoint` attributes of `servicemix-http` point back to the respective names of the `servicemix-jsr181`, so that we connect the components together.

Step Three—Deploy and Invoke EJB Binding in ServiceMix

For this sample we will follow the lightweight deployment model.

ch08\BindEjb\02_BindInEsb contains all the necessary files for binding, building, and deploying the artifacts into a ServiceMix container. To build the ESB binding codebase and deploy the sample, change directory to ch08\BindEjb\02_BindInEsb, which contains a top-level build.xml file. The notable section in the build.xml is shown in the following code:

```
<target name="copy-components"
        depends="init"
        description="Build components">
    <copy todir="${servicemix.home}/lib/optional" overwrite="true">
        <fileset dir="${client.classes.dir}"
            includes="sample_statelessSession_client.jar" />
    </copy>
    <copy todir="${servicemix.home}/lib/optional" overwrite="true">
        <fileset dir="${wl.home}/server/lib" includes="weblogic.jar" />
    </copy>
</target>
```

Here, we first copy the EJB client jar from the Weblogic directory to ServiceMix's optional library folder. Other than this, we don't have any explicit referral to EJB classes in the ServiceMix binding above. What you're seeing is Spring's success in abstracting away the client side of the clunky EJB contract. However, the home and remote interfaces are still required—Spring is using them under the hood, just as you would if you had to write the JNDI lookup, EJB home, and create calls yourself. What Spring is doing behind the scenes is making a JNDI lookup for the home, then calling the `create()` method on the home and enabling you to work with that. That is why we require the EJB client jar in the classpath.

We have seen previously, how we configured `jndiTemplate` by explicitly referring the `weblogic.jndi.WLInitialContextFactory`. This is included in `weblogic.jar` and hence we include that jar in the ServiceMix's optional library folder too. In case you are using a different EJB server and you use a different `java.naming.factory.initial` class, then include the respective jars in ServiceMix's optional library folder too. To build, execute ant as shown in the following code:

```
cd ch08\BindEjb\02_BindInEsb
ant
```

Now, bring up the ServiceMix container by executing the `servicemix.xml` file contained in the same folder.

```
%SERVICEMIX_HOME%/bin/servicemix servicemix.xml
```

The `Client.html` file provided again in the same folder can be used to send messages to test the deployed service.

Step Four—Access WSDL and Generate Axis-based Stubs to Access EJB Outside Firewall

As discussed above, we can now access the WSDL auto-generated by ServiceMix out of the earlier EJB binding. The WSDL can be accessed by pointing your browser to the following URL:

```
http://localhost:8192/Services/HelloWebService/?wsdl
```

or

```
http://localhost:8192/Services/HelloWebService/main.wsdl
```

The WSDL is reproduced in the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:ns1="http://io.java"
    xmlns:soap11="http://schemas.xmlsoap.org/soap/
        envelope/"
    xmlns:soap12="http://www.w3.org/2003/05/
        soap-envelope"
    xmlns:soapenc11="http://schemas.xmlsoap.org/soap/
        encoding/"
    xmlns:soapenc12="http://www.w3.org/2003/05/
        soap-encoding"
    xmlns:tns="http://binildas.com/esb/bindejb"
    xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/
        soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://binildas.com/esb/bindejb">
  <wsdl:types>
    <xsd:schema attributeFormDefault="qualified"
        elementFormDefault="qualified"
        targetNamespace="http://io.java"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <xsd:complexType name="IOException"/>
    </xsd:schema>
    <xsd:schema attributeFormDefault="qualified"
        elementFormDefault="qualified"
```

```

        targetNamespace="http://binildas.com/esb/bindejb"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="hello">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element maxOccurs="1" minOccurs="1" name="in0"
                    nillable="true" type="xsd:string"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="helloResponse">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element maxOccurs="1" minOccurs="1" name="out"
                    nillable="true" type="xsd:string"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="IOException" type="ns1:IOException"/>
</xsd:schema>
</wsdl:types>
<wsdl:message name="helloResponse">
    <wsdl:part element="tns:helloResponse" name="parameters">
    </wsdl:part>
</wsdl:message>
<wsdl:message name="IOException">
    <wsdl:part element="tns:IOException" name="IOException">
    </wsdl:part>
</wsdl:message>
<wsdl:message name="helloRequest">
    <wsdl:part element="tns:hello" name="parameters">
    </wsdl:part>
</wsdl:message>
<wsdl:portType name="jsrEjbEPPortType">
    <wsdl:operation name="hello">
        <wsdl:input message="tns:helloRequest" name="helloRequest">
        </wsdl:input>
        <wsdl:output message="tns:helloResponse" name="helloResponse">
        </wsdl:output>
        <wsdl:fault message="tns:IOException" name="IOException">
        </wsdl:fault>
    </wsdl:operation>
</wsdl:portType>
<wsdl:binding name="HelloServiceBIBinding"
    type="tns:jsrEjbEPPortType">
    <wsdlsoap:binding style="document"
        transport="http://schemas.xmlsoap.org/soap/
        http"/>

    <wsdl:operation name="hello">
        <wsdlsoap:operation soapAction=""/>

```

```
<wsdl:input name="helloRequest">
  <wsdlsoap:body use="literal"/>
</wsdl:input>
<wsdl:output name="helloResponse">
  <wsdlsoap:body use="literal"/>
</wsdl:output>
<wsdl:fault name="IOException">
  <wsdlsoap:fault name="IOException" use="literal"/>
</wsdl:fault>
</wsdl:operation>
</wsdl:binding>
<wsdl:service name="HelloServiceBIService">
  <wsdl:port binding="tns:HelloServiceBIBinding"
    name="HelloServiceBI">
    <wsdlsoap:address location="http://localhost:8192/Services/
      HelloWebService/">
  </wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

We will now use Apache Axis tools to auto-generate client-side stubs and binding classes using which we can write a simple Java client class, to access the service through the HTTP channel. When we do this, the interesting point is that we are accessing an EJB service behind the scenes, but using the SOAP format for request and response, that too through a firewall-friendly HTTP channel. The Axis client classes are placed in the directory `ch08\BindEjb\03_AxisClient`.

To do that, we have to use the `wsdl2java` ant task. Let us first declare the task definition and execute that task to generate stub classes, as shown in the following code:

```
<taskdef name="wsdl2java"
  classname="org.apache.axis.tools.ant.wsdl.Wsdl2javaAntTask"
  loaderref="axis" >
  <classpath refid="classpath"/>
</taskdef>
<target name="wsdl2java">
  <java classname="org.apache.axis.wsdl.WSDL2Java"
    fork="true"
    failonerror="true">
    <arg value="-o"/>
    <arg value="${src}"/>
    <arg value="-x"/>
    <arg value="http://io.java"/>
    <arg value="HelloWebService.wsdl"/>
    <classpath>
      <path refid="classpath"/>
      <pathelement location="${build}"/>
    </classpath>
  </java>
</target>
```

The task will extract the details from the WSDL and generate the following client-side artifacts in the `ch08\BinEjb\03_AxisClient\src` folder:

```
com\binildas\esb\bindejb\HelloServiceBIBindingStub.java
com\binildas\esb\bindejb\HelloServiceBIService.java
com\binildas\esb\bindejb\HelloServiceBIServiceLocator.java
com\binildas\esb\bindejb\JsrEjbEPPortType.java
```

Remember that the above WSDL from where the Axis tool generated the client-side artifacts is, in fact, the WSDL retrieved from the ServiceMix ESB. The Client java class can be written against these generated files as follows:

```
public class Client
{
    private static String wsdlUrl = "http://localhost:8192/
        Services/HelloWebService/main.wsdl";
    private static String namespaceURI = "http://binildas.com/
        esb/bindejb";
    private static String localPart = "HelloServiceBIService";
    protected void executeClient(String[] args)throws Exception
    {
        HelloServiceBIService helloServiceBIService = null;
        JsrEjbEPPortType jsrEjbEPPortType = null;
        if(args.length == 3)
        {
            helloServiceBIService = new HelloServiceBIServiceLocator(
                args[0], new QName(args[1], args[2]));
        }
        else
        {
            helloServiceBIService = new HelloServiceBIServiceLocator(
                wsdlUrl, new QName(namespaceURI, localPart));
        }
        jsrEjbEPPortType = helloServiceBIService.getHelloServiceBI();
        log("Response From Server : " + jsrEjbEPPortType.hello(
            "Binil"));
    }
    public static void main(String[] args)throws Exception
    {
        Client client = new Client();
        client.executeClient(args);
    }
}
```

To build the entire Axis client codebase, assuming that both the EJB server and the ServiceMix container is up and running, change directory to `ch08\BindEjb\03_AxisClient`, which contains a top-level `build.xml` file. Execute `ant` as shown in the following commands:

```
cd ch08\BindEjb\03_AxisClient
ant
```

This will generate the required Axis client-side stubs and compile the client classes. Now to run the client, execute the following command:

```
ant run
```

Reconciling EJB Resources

This chapter demonstrated how to engineer with ServiceMix, hence we can consider EJB component services equivalent to normal web services so the same rules can be applied from a consumer perspective. This will have greater impact in today's solution infrastructure. This is because in the last one decade or so, we saw the rise of EJB component-based programming. Setting aside (if) any drawbacks, EJB gave us a lot of support for cross cutting concerns like transaction management and instance pooling. For the same reason, lots of solution artifacts are still available and remaining hosted in the form of EJB services. Using an ESB, we are no longer forced to throw away all those investments, instead leverage them in a services ecosystem environment. I am sure I will have an easy time convincing my CTO that I don't need to scrap all my EJB investments, instead I can reuse them in the best possible manner. I hope you will also enjoy the same.

Summary

ESB as a service fabric supports integrating multiple services and component types. Hence, from the consumer perspective, they see a pure services interface with all SOA qualities. This chapter demonstrated how the same principles helped us to expose an EJB service as a firewall-friendly web service. The notable thing here is the ease with which an ESB framework does this – and that is where the right tools will help to solve even non trivial problems easily. So, instead of holding the ESB hammer and looking at every IT problem as a nail, use ESB and JBI to solve appropriate integration problems following SOI guidelines alone. Most component frameworks available today allow even POJO components to be exposed as services so that they can be consumed remotely. This is a lightweight approach compared to the traditional EJB programming paradigm.

In the next chapter, we will look into this concept to understand how we can expose an annotated POJO as services in the ESB.

9

POJO Binding Using JSR181

First things first and simple things foremost! I should have introduced the POJO binding as the first example due to the simplicity in the name "POJO". However, there is another component associated with the POJO binding, which is the `servicemix-jsr181` component that we need to learn together. This is why we delayed the POJO binding until now. We can now move on.

We will cover the following in this chapter:

- Overview on POJO
- JSR 181 and `servicemix-jsr181` component
- A POJO binding sample demonstrating POJO as services
- A second sample to demonstrate accessing the JBI bus directly at programming API-level

POJO

Christopher Richardson in his cover story "What Is POJO Programming" says:

Fortunately, there's now a much better way to build Enterprise Java applications: Plain Old Java Objects (POJOs), which are classes that don't implement infrastructure framework-specific interfaces, and non-invasive frameworks such as Spring, Hibernate, JDO, and EJB 3, which provide services for POJO.

What are POJOs

POJOs depend on basic or core Java and don't implement any other API's. They neither contain any framework specific callback methods, nor depend on any external methods, which they need to call. Instead, any third-party framework can make use of a POJO class and expose it or utilize it in their own way. For example,

an O-R mapping framework can use a POJO model class and persist in a relational database. Similarly, a service generation framework can expose all the public methods in a POJO service class as services. A TO assembler can get and set values from a POJO TO to create a coarser grained, POJO TO graph.

Comparing POJO with other Components

POJOs, being significantly lightweight, will have their own advantages and disadvantages when compared to other counterpart components such as EJBs and servlets.

The main advantage is that the POJOs can be deployed and run in the simplest of the Java run times available, and nothing prevents you from deploying them into a full-fledged container infrastructure. This will help you to leverage the best of both the worlds – the lightweight deployment model and the full-fledged container infrastructure functionalities.

At the same time, the downside of POJOs is that they are not remotable. Remoting is the mechanism by which functionality can be accessed from a remote node, through a network. EJB, servlet, and JSP. These components are inherently remotable. How interested would a reader be if we could make a POJO remotable – without much hassle? Hold on, we will do that in this chapter.

ServiceMix servicemix-jsr181

ServiceMix's `servicemix-jsr181` component is built based on the JSR 181 specification.

JSR 181

JSR 181 defines an annotated Java syntax for programming web services and is built on the Java Language Metadata technology (JSR 175), to provide an easy way to use syntax to describe web services at the source-code-level for the J2EE platform. It aims to make it easy for a Java developer to develop the server applications that conform both to basic SOAP and WSDL standards. The WSM for the Java platform is built upon the JSR 175 and hence requires a JDK installation supporting Java metadata.

servicemix-jsr181

`servicemix-jsr181` component is a JBI standard SE. It can expose annotated POJO as services. `servicemix-jsr181` internally uses XFire to expose POJOs as services. `servicemix-jsr181` links the JBI channel to XFire transport using the following steps:

- `servicemix-jsr181` first creates a `DefaultXFire` and registers the `JbiTransport` to the transport manager of XFire.
- Then it uses XFire's `ObjectServiceFactory` to create an XFire service.
- `servicemix-jsr181` then configures the service and registers it in the XFire service registry.

The above steps link the JBI channel to XFire transport. However, all these are the background plumbing abstracted out by the `servicemix-jsr181`, so that the developer won't see all these complexities. We have already discussed the power of XFire and worked out a few samples. Again we will leverage XFire, but now in the form of a standard JBI component itself, for binding.

As per the ServiceMix documentation, `servicemix-jsr181` supports the following features:

- no annotations
- jsr181 annotations
- commons-attributes annotations
- aegis binding
- jaxb2 binding
- xmlbeans binding
- wsdl auto generation
- MTOM or attachments support

servicemix-jsr181 Deployment

XBean-based deployment can be used to deploy `servicemix-jsr181`, both in the provider and consumer roles.

The first step is to declare the necessary jsr181 namespace elements, shown as follows:

```
<beans xmlns:jsr181="http://servicemix.apache.org/jsr181/1.0"
      xmlns:test="http://binildas.com/esb/jsrpojo">
</beans>
```

Whether it is a simple POJO binding or an EJB binding, we need to include all the required interface and implementation classes to the respective classpaths. We can do this by including the required class files or jar archives in the SU and reference them using the following tags in the `xbean.xml` configuration file:

```
<classpath>
  <location>.</location>
</classpath>
```

The path value specified for the `location` tag is relative to the unzipped SU. For example, if `JsrBind` is the SU name and we have a class called `test.EchoService`, then we need to package the class within the folder `test`, relative to the top-level of `JsrBind` archive. Similarly, you can also add Java archives (`.jar`) containing class files by explicitly specifying them as follows:

```
<classpath>
    <location>lib/test.jar</location>
</classpath>
```

servicemix-jsr181 Endpoint

`servicemix-jsr181` endpoints can be configured in multiple formats using Spring bean configurations. The multiple formats are listed in the following list:

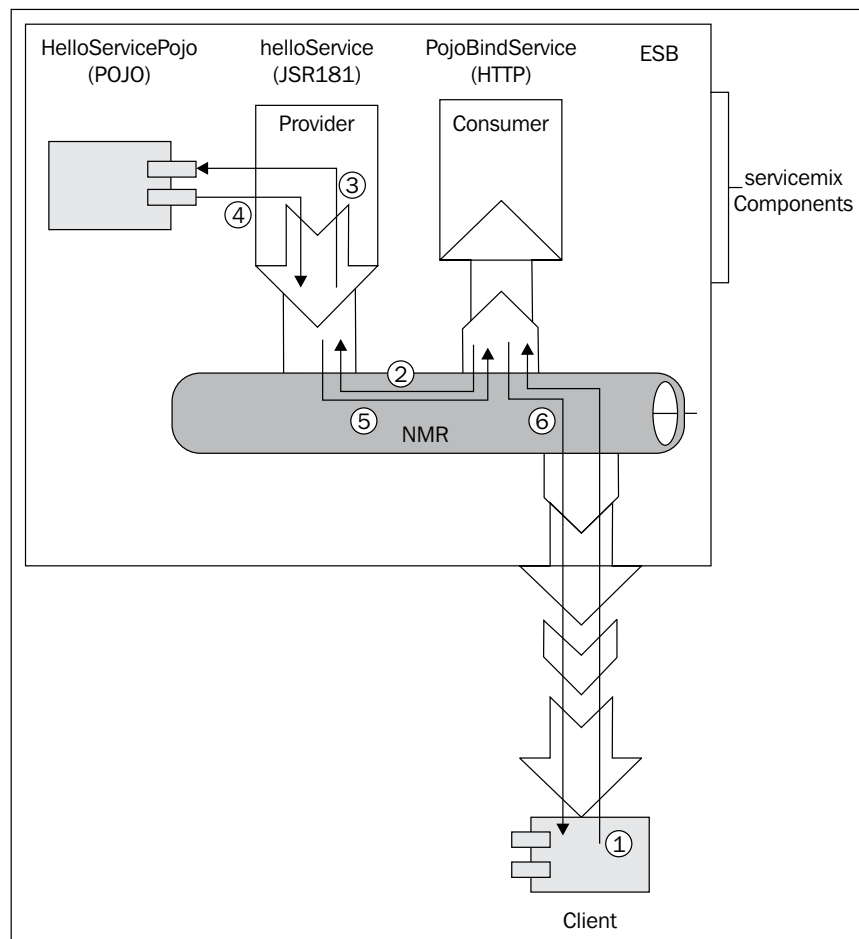
1. `<jsr181:endpoint annotations="none" service="test:helloService">`
 `<jsr181:pojo>`
 `<bean class="samples.HelloServicePojo">`
 `</bean>`
 `</jsr181:pojo>`
 `</jsr181:endpoint>`
2. `<bean id="helloPojo" class="samples.HelloServicePojo" />`
 `<jsr181:endpoint pojo="#helloPojo" />`
3. `<jsr181:endpoint pojoClass="samples.HelloServicePojo"`
 `annotations="none" />`

POJO Binding Sample

From the entire discussion we have had in this chapter, let us build a POJO binding sample. The codebase for the sample is located in the folder `ch09\Jsr181BindPojo`.

Sample Use Case

In this POJO binding sample use case, we will integrate POJO components with ServiceMix JBI components so that they will take part in message exchanges with the NMR. The POJO class used in the sample can be replaced with your code performing transformation, depending upon your business scenario. In our sample here, we will use a set of components integrated in the ESB as shown in the following figure:



A simple POJO class alone is enough to demonstrate the binding sample. However, we will also expose POJO as a normal web service, access WSDL (yes, really), and run an Axis client against the WSDL to generate client stubs to access the POJO service remotely. To facilitate all this, we will first integrate our POJO class (`HelloServicePojo`) with a `servicemix-jsr181` component. Now as we know, HTTP is an easy and simple channel to access services. Hence, let us also integrate a `servicemix-http` in the consumer role to the NMR so that our test clients can have an easy channel to send messages. The components are integrated as shown in the above figure. When the client sends a message, the message-flow across the NMR through various JBI components are marked by numbers in sequence.

POJO Code Listing

Let us first define a BI. Note that the BI is not mandatory for POJO binding, but here we are using best (interface-based programming) practices. Thus, `HelloServiceBI` is the BI.

The interface `HelloServiceBI` is shown in the following code:

```
public interface HelloServiceBI
{
    String hello(String phrase) throws java.io.IOException;
}
```

In Java RMI or EJB programming, the business methods are usually marked to throw a `java.rmi.RemoteException`. We can generalize this exception scenario by replacing the `RemoteException` with the `java.io.IOException`. We will follow this convention by marking our business methods as throwing `IOException`. There is absolutely nothing wrong if we don't mention the exception in the `throws` clause too, for our samples here. Now, `HelloServicePojo` will implement the above interface. Moreover as you can see, this class qualifies to be a hundred percent POJO class.

The `HelloServicePojo` class is shown as follows:

```
public class HelloServicePojo implements HelloServiceBI
{
    private static long times = 0;
    public String hello(String phrase)
    {
        System.out.println("HelloServiceBean.hello
                           {" + (++times) + "} ");
        return "From HelloServiceBean : HELLO!! You just said :
               " + phrase;
    }
}
```

XBean-based POJO Binding

Using XBean, we will now configure the POJO to be deployed onto the standard `servicemix-jsr181` JBI component.

The `xbean.xml` is as shown in the following code:

```
<beans xmlns:jsr181="http://servicemix.apache.org/jsr181/1.0"
        xmlns:test="http://binildas.com/esb/jsrpojo">
    <classpath>
        <location>.</location>
```

```

</classpath>
<jsr181:endpoint annotations="none"
    service="test:helloService"
    serviceInterface="samples.HelloServiceBI">
    <jsr181:pojo>
        <bean class="samples.HelloServicePojo">
        </bean>
    </jsr181:pojo>
</jsr181:endpoint>
</beans>

```

The above configuration will expose `HelloServicePojo` as a service on the JBI bus, so that from now on any JBI component can exchange messages with this component.

To easily demonstrate accessing the service, we will also bind the HTTP channel to the JBI bus. Then a simple HTML client can interact with the bus sending the messages. The HTTP binding is shown in the following code:

```

<beans xmlns:http="http://servicemix.apache.org/http/1.0"
    xmlns:test="http://binildas.com/esb/jsrpojo">
    <classpath>
        <location>.</location>
    </classpath>
    <http:endpoint service="test:PojoBindService"
        endpoint="PojoBindService"
        role="consumer"
        targetService="test:helloService"
        locationURI="http://localhost:8081/services/
            PojoBindService"
        soap="true"
        defaultMep="http://www.w3.org/2004/08/wsdl/
            in-out"/>
</beans>

```

Deployment Configuration

For deployment, we will package the relevant artifacts for the JSR POJO binding into a ZIP archive named `jsrbind-su.zip`. We need to deploy this archive onto the `servicemix-jsr181` component and that is done through the `jbi.xml` as shown in the following code:

```

<?xml version="1.0" encoding="UTF-8"?>
<jbi xmlns="http://java.sun.com/xml/ns/jbi" version="1.0">
    <service-assembly>
        <identification>

```

```
        <name>JsrBindAssembly</name>
        <description>JsrBind Service Assembly</description>
    </identification>
    <service-unit>
        <identification>
            <name>JsrBind</name>
            <description>JsrBind Service Unit</description>
        </identification>
        <target>
            <artifacts-zip>jsrbind-su.zip</artifacts-zip>
            <component-name>servicemix-jsr181</component-name>
        </target>
    </service-unit>
</service-assembly>
</jbi>
```

As said earlier, we also have a HTTP binding to easily access the POJO service. Hence, we have another `jbi.xml`, which will specify how to deploy the HTTP artifacts onto the `servicemix-http` standard JBI component. This is shown in the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<jbi xmlns="http://java.sun.com/xml/ns/jbi" version="1.0">
    <service-assembly>
        <identification>
            <name>HttpBindAssembly</name>
            <description>HttpBind Service Assembly</description>
        </identification>
        <service-unit>
            <identification>
                <name>JsrProxy</name>
                <description>HttpBind Service Unit</description>
            </identification>
            <target>
                <artifacts-zip>httpbind-su.zip</artifacts-zip>
                <component-name>servicemix-http</component-name>
            </target>
        </service-unit>
    </service-assembly>
</jbi>
```

Deploying and Running the Sample

As the first step and if you haven't done it before, edit `examples.PROPERTIES` provided along with the code download for this chapter and change the paths there to match your development environment. The code download for this chapter also includes a `README.txt` file, which gives detailed steps to build and run the samples.

To build the entire codebase and deploy the sample, change directory to `ch09\Jsr181BindPojo`, which contains a top-level `build.xml` file. Execute `ant` as shown in the following command:

```
cd ch09\Jsr181BindPojo
ant
```

Now, bring up the ServiceMix container by executing the `servicemix.xml` file contained in the same folder.

```
%SERVICEMIX_HOME%/bin/servicemix servicemix.xml
```

The `Client.html` provided again in the same folder can be used to send messages to test the deployed service.

Access WSDL and Generate Axis-based Stubs to Access POJO Remotely

You can now access the WSDL auto-generated by ServiceMix out of the earlier POJO binding. The WSDL can be accessed by pointing your browser to the following URL:

```
http://localhost:8081/services/PojoBindService/?wsdl
```

or

```
http://localhost:8081/services/PojoBindService/main.wsdl
```

Even though the WSDL here has nothing fancy about it, the important aspect is that this WSDL is auto-generated by the JBI bus, out of the `serviceInterface` (the BI, discussed earlier) configured in the `jsr181:endpoint`. Let us make sure that the WSDL is very similar to any WSDL we generate out from a normal web service. Let us list the WSDL here:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:ns1="http://io.java"
                  xmlns:soap11="http://schemas.xmlsoap.org/soap/
                               envelope/"
```



```

        xmlns:soap12="http://www.w3.org/2003/05/
            soap-envelope"
        xmlns:soapenc11="http://schemas.xmlsoap.org/soap/
            encoding/"
        xmlns:soapenc12="http://www.w3.org/2003/05/
            soap-encoding"
        xmlns:tns="http://binildas.com/esb/jsrpojo"
        xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/
            soap/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://binildas.com/esb/jsrpojo">
<wsdl:types>
  <xsd:schema attributeFormDefault="qualified"
    elementFormDefault="qualified"
    targetNamespace="http://binildas.com/esb/jsrpojo"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="hello">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element maxOccurs="1" minOccurs="1" name="in0"
            nillable="true" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="helloResponse">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element maxOccurs="1" minOccurs="1" name="out"
            nillable="true" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="IOException" type="ns1:IOException"/>
  </xsd:schema>
  <xsd:schema attributeFormDefault="qualified"
    elementFormDefault="qualified"
    targetNamespace="http://io.java"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:complexType name="IOException"/>
  </xsd:schema>
</wsdl:types>
<wsdl:message name="helloRequest">
  <wsdl:part element="tns:hello" name="parameters">
    </wsdl:part>
</wsdl:message>
<wsdl:message name="helloResponse">

```

```

        <wsdl:part element="tns:helloResponse" name="parameters">
        </wsdl:part>
    </wsdl:message>
    <wsdl:message name="IOException">
        <wsdl:part element="tns:IOException" name="IOException">
        </wsdl:part>
    </wsdl:message>
    <wsdl:portType name="helloServicePortType">
        <wsdl:operation name="hello">
            <wsdl:input message="tns:helloRequest" name="helloRequest">
            </wsdl:input>
            <wsdl:output message="tns:helloResponse"
                name="helloResponse">
            </wsdl:output>
            <wsdl:fault message="tns:IOException" name="IOException">
            </wsdl:fault>
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="PojoBindServiceBinding"
        type="tns:helloServicePortType">
        <wsdlsoap:binding style="document"
            transport="http://schemas.xmlsoap.org/
                soap/http"/>

        <wsdl:operation name="hello">
            <wsdlsoap:operation soapAction=""/>
            <wsdl:input name="helloRequest">
                <wsdlsoap:body use="literal"/>
            </wsdl:input>
            <wsdl:output name="helloResponse">
                <wsdlsoap:body use="literal"/>
            </wsdl:output>
            <wsdl:fault name="IOException">
                <wsdlsoap:fault name="IOException" use="literal"/>
            </wsdl:fault>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="PojoBindService">
        <wsdl:port binding="tns:PojoBindServiceBinding"
            name="PojoBindService">
            <wsdlsoap:address location="http://localhost:8081/
                services/PojoBindService"/>

        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>

```

We will now use the Apache Axis tools to auto-generate the client-side stubs and the binding classes using which we can write a simple Java client class to access the service through a HTTP channel. The Axis client classes are placed in the directory `ch09\Jsrl81BindPojo\03_AxisClient`.

To do that, we have to use the `wsdl2java` ant task. Let us declare the task definition and execute that task to generate the stub classes.

```
<taskdef name="wsdl2java"
    classname="org.apache.axis.tools.ant.wsdl.Wsdl2javaAntTask"
    loaderref="axis" >
    <classpath refid="classpath"/>
</taskdef>
<target name="wsdl2java">
    <java classname="org.apache.axis.wsdl.WSDL2Java"
        fork="true" failonerror="true">
        <arg value="-o"/>
        <arg value="{src}"/>
        <arg value="-x"/>
        <arg value="http://io.java"/>
        <arg value="http://localhost:8081/services/PojoBindService/
                                main.wsdl"/>
        <classpath>
            <path refid="classpath"/>
            <pathelement location="{build}"/>
        </classpath>
    </java>
</target>
```

The task will extract WSDL from the specified location and generate the following client-side artifacts:

```
com\binildas\esb\jsrpojo\HelloServicePortType.java
com\binildas\esb\jsrpojo\PojoBindService.java
com\binildas\esb\jsrpojo\PojoBindServiceBindingStub.java
com\binildas\esb\jsrpojo\PojoBindServiceLocator.java
```

The Client Java class can be written against these generated files as follows:

```
public class Client
{
    private static String wsdlUrl = "http://localhost:8081/services/
                                PojoBindService/main.wsdl";
    private static String namespaceURI = "http://binildas.com/esb/
                                jsrpojo";
    private static String localPart = "PojoBindService";
```

```

protected void executeClient(String[] args)throws Exception
{
    PojoBindService pojoBindService = null;
    HelloServicePortType helloServicePortType = null;
    if(args.length == 3)
    {
        pojoBindService = new PojoBindServiceLocator(args[0],
            new QName(args[1], args[2]));
    }
    else
    {
        pojoBindService = new PojoBindServiceLocator(wsdlUrl,
            new QName(namespaceURI, localPart));
    }
    helloServicePortType = pojoBindService.getPojoBindService();
    log("Response From Server : " + helloServicePortType.
        hello("Binil"));
}
public static void main(String[] args)throws Exception
{
    Client client = new Client();
    client.executeClient(args);
}
}

```

To build the entire Axis client codebase, assuming that ServiceMix is up and running, change directory to `ch09\Jsr181PojoAccessBus\04_AxisClient`, which contains a top-level `build.xml` file. Execute ant as follows:

```

cd ch09\ Jsr181PojoAccessBus
ant

```

This will generate the required Axis client-side stubs and compile the client classes. Now to run the client, execute the following command:

```

ant run

```

Accessing JBI Bus Sample

In the previous sample, we have seen how to bind a POJO to JBI and access the service again using the standard web service access mechanisms. Occasionally, your components may also want to access the JBI bus directly at programming API-level, and we can do that using the numerous APIs provided by ServiceMix and JBI.

The preferred mechanism to access JBI from your component is to first get the `ComponentContext` implementation and from there traverse the other JBI APIs.

```
<jsr181:endpoint annotations="none" service="some:Service"
                 serviceInterface="some.Interface">
  <jsr181:pojo>
    <bean class="some.Pojo">
      <property name="context" ref="context" />
    </bean>
  </jsr181:pojo>
</jsr181:endpoint>
```

Correspondingly, we will now have our `Pojo` (Oh, yes; our POJOs are getting bulkier here!) class into which the JBI container can inject a reference of the `ComponentContext`. The `pojo` class is reproduced in the following code:

```
public class Pojo
{
  private javax.jbi.component.ComponentContext context;
  public void setContext(javax.jbi.component.ComponentContext
                        context)
  {
    this.context = context;
  }
}
```

`ComponentContext` is the JBI API using which we can access the underlying `DeliveryChannel`. This is shown in the following code:

```
public interface ComponentContext
{
  public DeliveryChannel getDeliveryChannel() throws
                        MessagingException;
}
```

`DeliveryChannel` can be used to send messages to the JBI bus. It is to be noted that only `sendSync()` is allowed for active JBI exchanges (but you have to use `send()` for DONE or ERROR status exchanges). This is shown in the following code:

```
public interface DeliveryChannel
{
  public MessageExchange accept() throws MessagingException;
  public void send(MessageExchange exchange) throws
                MessagingException;
  public boolean sendSync(MessageExchange exchange)
                throws MessagingException;
  public boolean sendSync(MessageExchange exchange, long timeout)
                throws MessagingException;
}
```

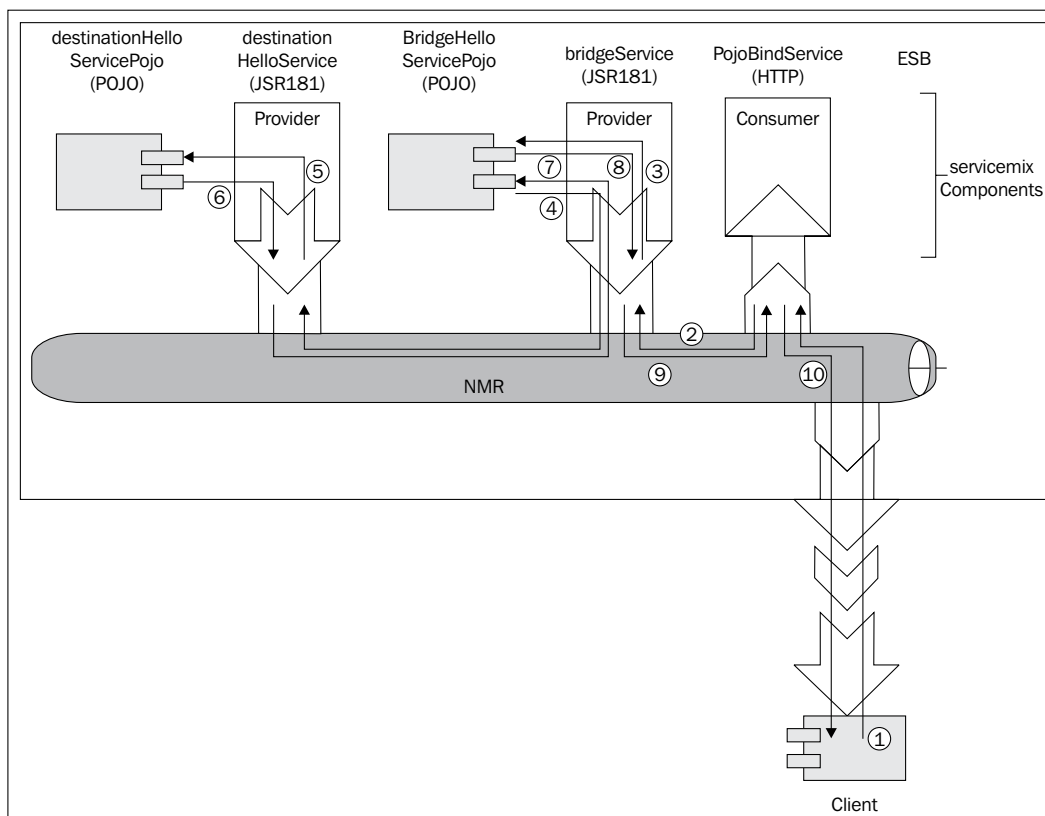
Sample Use Case for Accessing JBI Bus

We will have another full sample for POJO binding to demonstrate how to access the JBI bus from within a POJO. As described above, we can use `ComponentContext` to access the `DeliveryChannel` to send messages to the JBI bus. Another way to interact with the JBI bus is to use the client API. Let us define a sample use case to access the JBI bus.

We will have the following components to implement the sample use case:

- Client
- HTTP binding
- JSR POJO Bridge
- JSR POJO Destination

These components are integrated as shown in the following figure. When the client sends a message, the message flow through the NMR through various JBI components is marked by numbers in sequence.



The above components will exchange messages in the following sequence:

1. Client sends message to HTTP binding.
2. HTTP binding routes messages to destination through JBI bus.
3. JSR POJO Bridge being the destination for the above message, will accept the message.
4. The JSR POJO Bridge will perform the following steps to send the next message to the JBI bus.
 - a. It first creates a reference of `ServiceMixClient`.

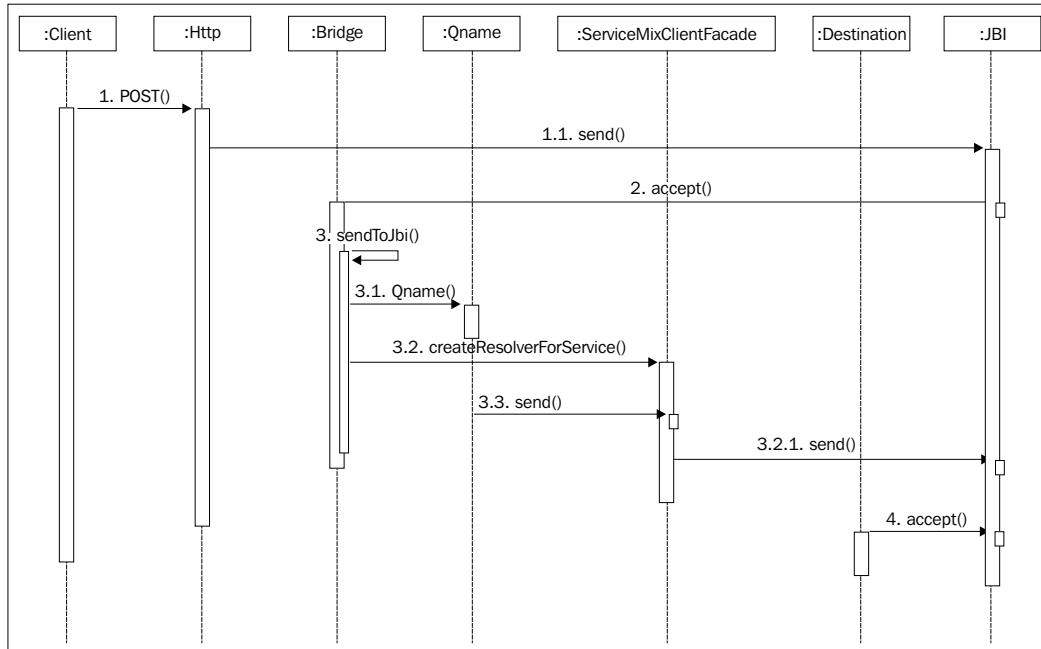
```
ServiceMixClient client = new ServiceMixClientFacade(
                                this.context);
```
 - b. It then creates a `QName` to refer the destination service.

```
QName service = new QName("http://binildas.com/esb/
                                jsrpojo", "destinationHelloService");
```
 - c. Using the above `ServiceMixClient` and the service reference, it then creates an `EndpointResolver`.

```
EndpointResolver resolver =
                                client.createResolverForService(service);
```
 - d. Now, we can send the message to the **JBI** bus using the `ServiceMixClient`.

```
client.send(resolver, null, null, messageToDespatch);
```
5. Now, `destinationHelloService` is the service pointing to JSR POJO Destination. So, the message from the JBI bus will be accepted by JSR POJO Destination.

The following figure shows the above sequences of events:



Sample Code Listing

There is not much difference in the code than what we have seen in the previous sample, but the `BridgeHelloServicePojo` class will be listed as follows, to discuss a few points:

```

public class BridgeHelloServicePojo implements BridgeHelloServiceBI
{
    private ComponentContext context;
    public void setContext(ComponentContext context)
    {
        this.context = context;
    }
    public String broker(String phrase)
    {
        try
        {
            send(phrase);
        }
        catch(JBIException jbiException)
        {
        }
    }
}
  
```



```
        jbiException.printStackTrace();
        return jbiException.getMessage();
    }
    return "Success";
}
private void send(String message) throws javax.jbi.JBIException
{
    System.out.println("BridgeHelloServicePojo.1. message : " +
        message);
    ServiceMixClient client = new ServiceMixClientFacade(
        this.context);
    QName service = new QName(namespaceURI, localPart);
    EndpointResolver resolver = client.createResolverForService(
        service);
    String messageToDespatch = getMessage(message);
    client.send(resolver, null, null, messageToDespatch);
}
private String getMessage(String message)
{
    StringBuffer stringBuffer = new StringBuffer();
    stringBuffer.append("<hello>").append("<helloRequest>")
        .append("<message xmlns=\"http://soap\">" + message
            + "</message>")
        .append("</helloRequest>").append("</hello>");
    return stringBuffer.toString();
}
}
```

Perhaps, you might get stuck at the message building section of the code above. This will lead to the following XML code:

```
<hello>
  <helloRequest>
    <message xmlns="http://soap">" + message + "</message>
  </helloRequest>
</hello>
```

This is slightly different from the usual SOAP enveloped messages, which we sent to the JBI bus. The difference is that we don't have a SOAP envelope, but just the message. This is because simple binding the POJO using `servicemix-jsr181` requires the consumer to send just a normalized (XML) message, and not a SOAP message.

Build, Deploy, and Run the Sample

The codebase for this sample is located at `ch09\Jsr181PojoAccessBus`. To build the entire codebase and deploy the sample, change directory to `ch09\Jsr181PojoAccessBus`, which contains a top-level `build.xml` file. Execute `ant` as follows:

```
cd ch09\ Jsr181PojoAccessBus
ant
```

Now, bring up the ServiceMix container by executing the `servicemix.xml` file contained in the same folder, as shown in the following commands:

```
cd ch09\ Jsr181PojoAccessBus
%SERVICEMIX_HOME%\bin\servicemix servicemix.xml
```

The `Client.html` file provided again in the same folder can be used to send messages to test the deployed service.

For completeness of the sample, we also have code to access the service using the Axis client in the folder `ch09\Jsr181PojoAccessBus\04_AxisClient`.

To build the entire Axis client codebase, assuming that ServiceMix is up and running, change directory to `ch09\Jsr181PojoAccessBus\04_AxisClient`, which contains a top-level `build.xml` file. Execute `ant` as follows:

```
cd ch09\ Jsr181PojoAccessBus
ant
```

This will generate the required Axis client-side stubs and compile the client classes. Now to run the client, execute the following command:

```
ant run
```

Summary

This chapter discussed POJO classes, and how we can hook them onto the JBI bus. Hence, the services provided by these classes can even be accessed remotely, just like you access a normal web service. JBI is all about SOI and thus provides a framework for integrating components as services to the bus. This is what we have seen in this chapter, when we bound a POJO to the NMR and accessed it through normal web service channels.

In the next chapter, we will look into yet another common scenario found in production and deployment of web services namely Web Services Gateways.

10

Bind Web Services in ESB— Web Services Gateway

Since SOI is all about integrating multiple SOA-based systems, web services play a critical role in the integration space. This chapter is all about the importance of web services in integration. We will use the samples to illustrate how to bind web services with the ServiceMix ESB to facilitate integration.

We will cover the following in this chapter:

- Web services and binding
- Introduction to HTTP
- ServiceMix's `servicemix-http` component
- The consumer and provider roles for the ServiceMix JBI components
- `servicemix-http` in the consumer and provider roles
- Web service binding (Gateway) sample

Web Services

Web services separate out the service contract from the service interface. This feature is one of the many characteristic required for an SOA-based architecture. Thus, even though it is not mandatory that we use the web service to implement an SOA-based architecture, yet it is clearly a great enabler for SOA.

Web services are hardware, platform, and technology neutral. The producers and/or consumers can be swapped without notifying the other party, yet the information can flow seamlessly. An ESB can play a vital role to provide this separation.

Binding Web Services

A web service's contract is specified by its WSDL and it gives the endpoint details to access the service. When we bind the web service again to an ESB, the result will be a different endpoint, which we can advertise to the consumer. When we do so, it is very critical that we don't lose any information from the original web service contract.

Why Another Indirection?

There can be multiple reasons for why we require another level of indirection between the consumer and the provider of a web service, by binding at an ESB.

Systems exist today to support business operations as defined by the business processes. If a system doesn't support a business process of an enterprise, that system is of little use. Business processes are never static. If they remain static then there is no growth or innovation, and it is doomed to fail. Hence, systems or services should facilitate agile business processes. The good architecture and design practices will help to build "services to last" but that doesn't mean our business processes should be stable. Instead, business processes will evolve by leveraging the existing services. Thus, we need a process workbench to assemble and orchestrate services with which we can "Mix and Match" the services. ESB is one of the architectural topologies where we can do the mix and match of services. To do this, we first bind the existing (and long lasting) services to the ESB. Then leverage the ESB services, such as aggregation and translation, to mix and match them and advertise new processes for businesses to use.

Moreover, there are cross service concerns such as versioning, management, and monitoring, which we need to take care to implement the SOA at higher levels of maturity. The ESB is again one way to do these aspects of service orientation.

HTTP

HTTP is the World Wide Web (www) protocol for information exchange. HTTP is based on character-oriented streams and is firewall-friendly. Hence, we can also exchange XML streams (which are XML encoded character streams) over HTTP. In a web service we exchange XML in the SOAP (Simple Object Access Protocol) format over HTTP. Hence, the HTTP headers exchanged will be slightly different than a normal web page interaction. A sample web service request header is shown as follows:

```
GET /AxisEndToEnd/services/HelloWebService?WSDL HTTP/1.1
User-Agent: Java/1.6.0-rc
Host: localhost:8080
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: keep-alive
```

```
POST /AxisEndToEnd/services/HelloWebService HTTP/1.0
Content-Type: text/xml; charset=utf-8
Accept: application/soap+xml, application/dime, multipart/related,
text/*
User-Agent: Axis/1.4
Host: localhost:8080
Cache-Control: no-cache
Pragma: no-cache
SOAPAction: ""
Content-Length: 507
```

The first line contains a method, a URI and an HTTP version, each separated by one or more blank spaces. The succeeding lines contain more information regarding the web service exchanged.

ESB-based integration heavily leverages the HTTP protocol due to its open nature, maturity, and acceptability. We will now look at the support provided by the ServiceMix in using HTTP.

ServiceMix's servicemix-http

Binding external web services at the ESB layer can be done in multiple ways but the best way is to leverage JBI components such as the `servicemix-http` component within ServiceMix. We will look in detail at how to bind the web services onto the JBI bus.

servicemix-http in Detail

`servicemix-http` is used for HTTP or SOAP binding of services and components into the ServiceMix NMR. For this ServiceMix uses an embedded HTTP server based on the Jetty.

In Chapter 3, you have already seen the following two ServiceMix components:

- `org.apache.servicemix.components.http.HttpInvoker`
- `org.apache.servicemix.components.http.HttpConnector`

As of today, these components are deprecated and the functionality is replaced by the `servicemix-http` standard JBI component. A few of the features of the `servicemix-http` are as follows:

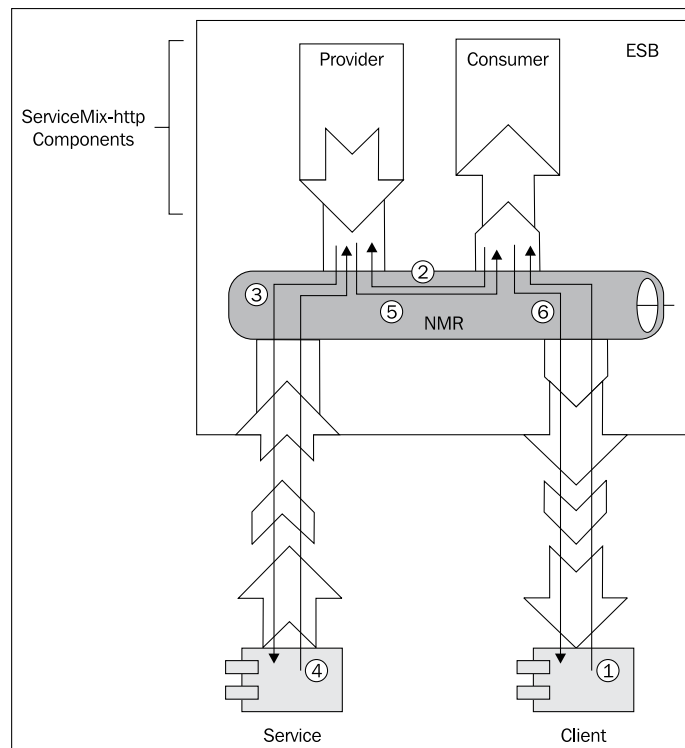
- Supports SOAP 1.1 and 1.2
- Supports MIME with attachments
- Supports SSL

- Supports WS-Addressing and WS-Security
- Supports WSDL-based and XBean-based deployments
- Support for all MEPs as consumers or providers

Since `servicemix-http` can function both as a consumer and a provider, it can effectively replace the previous `HttpInvoker` and `HttpConnector` component.

Consumer and Provider Roles

When we speak of the **Consumer** and **Provider** roles for the ServiceMix components, the difference is very subtle at first sight, but very important from a programmer perspective. The following figure shows the Consumer and Provider roles in the ServiceMix ESB:



The above figure shows two instances of `servicemix-http` deployed in the ServiceMix ESB, one in a provider role and the other in the consumer role. As it is evident, these roles are with respect to the NMR of the ESB. In other words, a

consumer role implies that the component is a consumer to the NMR whereas a provider role implies the NMR is the consumer to the component. Based on these roles, the NMR will take responsibility of any format or protocol conversions for the interacting components.

Let us also introduce two more parties here to make the role of a consumer and a provider clear—a client and a service. In a traditional programming paradigm, the client interacts directly with the server (or service) to avail the functionality. In the ESB model, both the client and the service interact with each other only through the ESB. Hence, the client and the service need peers with their respective roles assigned, which in turn will interact with each other. Thus, the ESB consumer and provider roles can be regarded as the peer roles for the client and the service respectively.

Any client request will be delegated to the consumer peer who in turn interacts with the NMR. This is because the client is unaware of the ESB and the NMR protocol or format. However, the `servicemix-http` consumer knows how to interact with the NMR. Hence any request from the client will be translated by the `servicemix-http` consumer and delivered to the NMR. On the service side also, the NMR needs to invoke the service. But the server service is neutral of any specific vendor's NMR and doesn't understand the NMR language as such. A peer provider role will help here. The provider receives the request from the NMR, translates it into the actual format or protocol of the server service and invokes the service. Any response will also follow the reverse sequence.

servicemix-http XBean Configuration

The `servicemix-http` components supports the XBean-based deployment. Since the `servicemix-http` component can be configured in both the consumer and provider roles, we have two sets of configuration parameters for the component. Let us look into the main configuration parameters:

- **servicemix-http as consumer:** A sample `servicemix-http` consumer component configuration is shown as follows:

```
<http:endpoint service="test:MyConsumerService"
    endpoint="HelloWebService"
    role="consumer"
    targetService="test:IHelloWebService"
    locationURI="http://localhost:8081/services/
                HelloWebService"
    soap="true"
    defaultMep="http://www.w3.org/2004/08/wsdl/in-out"
    wsdlResource="http://localhost:8080/AxisEndToEnd/
                services/HelloWebService?WSDL" />
```


The following table gives the explanation for the main configuration parameters:

Attribute Name	Type	Description	Mandatory or Not
service	QName	Service name of the proxy endpoint	Mandatory
endpoint	String	Endpoint name of the proxy endpoint	Mandatory
interfaceName	QName	Interface name of the proxy endpoint	Not Mandatory
targetService	QName	Service name of the target endpoint	Not Mandatory. Default is the value of the service attribute
targetEndpoint	String	Endpoint name of the target endpoint	Not Mandatory. Default is the value of the endpoint attribute
role	String	Whether a consumer or a provider	Mandatory. Value should be consumer
locationURI	URI	Http URL where this proxy endpoint will be exposed so that the ESB clients can access the proxy service.	Mandatory
defaultMEP	URI	The MEP URI by which clients interact with the consumer component	Not Mandatory
soap	boolean	If it is true, the component will parse the SOAP envelope and pass the contents to the NMR	Not Mandatory. Default value is false.
wSDLResource	Spring Resource	If it is set, the WSDL will be retrieved from this configured Spring resource.	Not Mandatory

Thus, the `locationURI` attribute in the `servicemix-http` consumer refers to the Http URL where this proxy endpoint is exposed, so that the ESB clients can access the proxy service. Later we will look at how to generate static client stubs out of this proxy URI.

- **servicemix-http as provider:** While configuring the provider, there are a few aspects to be taken care of with respect to the WSDL. If we have the sample WSDL as shown as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://AxisEndToEnd.axis.
                                apache.binildas.com"
```

```

xmlns:impl="http://AxisEndToEnd.axis.apache.
                                binildas.com">
<!-- other descriptions -->
<wsdl:service name="IHelloWebService">
  <wsdl:port binding="impl:HelloWebServiceSoapBinding"
    name="HelloWebService">
    <wsdlsoap:address location="http://localhost:8080/
      AxisEndToEnd/services/HelloWebService"/>
  </wsdl:port>
</wsdl:service>
</wsdl:definitions>

```

Now, while configuring the provider component you need to ensure that the service (IHelloWebService) and the endpoint (HelloWebService) match the service name and port elements of the WSDL that you use to correctly return the WSDL for the endpoint. Moreover, the service name will use the targetNamespace for the WSDL (http://AxisEndToEnd.axis.apache.binildas.com).

A sample servicemix-http provider component configuration is shown as follows:

```

<http:endpoint service="test:IHelloWebService"
  endpoint="HelloWebService"
  role="provider"
  locationURI="http://localhost:8080/AxisEndToEnd/
    services/HelloWebService"

  soap="true"
  soapAction=" "
  wsdlResource="http://localhost:8080/AxisEndToEnd/
    services/HelloWebService?WSDL" />

```

The following table gives the explanation for the main configuration parameters:

Attribute Name	Type	Description	Mandatory or Not
service	QName	Service name of the exposed endpoint	Mandatory
endpoint	String	Endpoint name of the exposed endpoint	Mandatory
interfaceName	QName	Interface name of the exposed endpoint	Not Mandatory
role	String	Whether a consumer or a provider	Mandatory. Value should be provider
locationURI	URI	Http URL of the target service.	Mandatory

Attribute Name	Type	Description	Mandatory or Not
soap	boolean	If it is true, the component will parse the SOAP envelope and pass the contents to the NMR	Not Mandatory. Default value is false.
soapAction	String	The SOAPAction header to be send over HTTP when invoking the web service	Not Mandatory. Default value is "".
wSDLResource	Spring Resource	If it is set, the WSDL will be retrieved from this configured Spring resource.	Not Mandatory

servicemix-http Lightweight Configuration

In addition to the XBean-based configuration, `servicemix-http` can also be deployed based on the lightweight mode to use in an embedded ServiceMix. The configuration would be as follows:

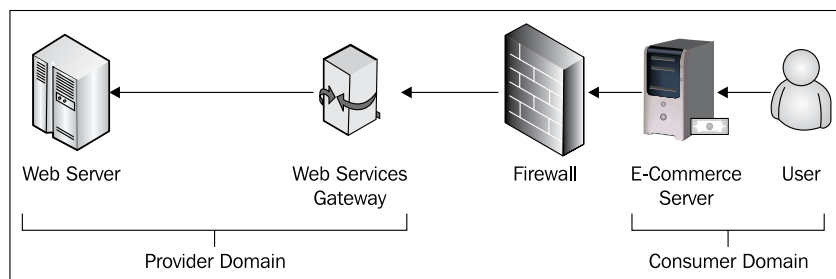
```
<sm:activationSpec>
  <sm:component>
    <http:component>
      <http:endpoints>
        <http:endpoint service="test:IHelloWebService"
          endpoint="HelloWebService"
          role="provider"
          locationURI="http://localhost:8080/
            AxisEndToEnd/services/HelloWebService"
          soap="true"
          soapAction=""
          wSDLResource="http://localhost:8080/
            AxisEndToEnd/services/
              HelloWebService?WSDL" />
        <http:endpoint service="test:MyConsumerService"
          endpoint="HelloWebService"
          role="consumer"
          targetService="test:IHelloWebService"
          locationURI="http://localhost:8081/
            services/HelloWebService"
          soap="true"
          defaultMep="http://www.w3.org/2004/08/
            wsdl/in-out"
          wSDLResource="http://localhost:8080/
            AxisEndToEnd/services/
              HelloWebService?WSDL" />
      </http:endpoints>
    </http:component>
  </sm:component>
</sm:activationSpec>
```

Web Service Binding Sample

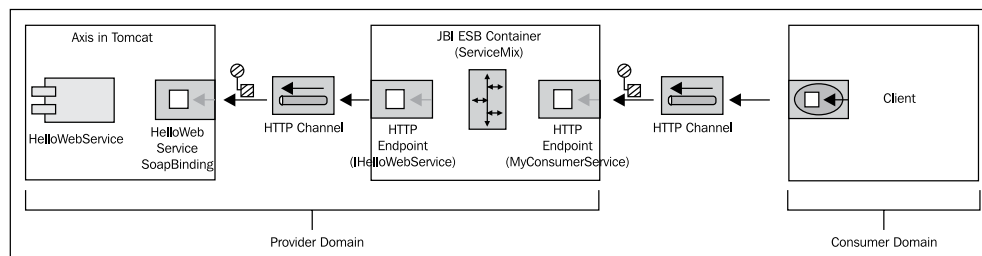
We will now look at a complete sample of how to bind a web service to the ServiceMix. While doing so, we will also see how to use the Apache Axis client-side tools to generate stubs based on the binding at ServiceMix. Normally we point to the actual WSDL URL to generate client stubs, but in this example we will point the tools to the ServiceMix binding. Then the ServiceMix binding will act completely as the web service gateway visible to the external clients, thus shielding the actual web service in the background.

Sample Use Case

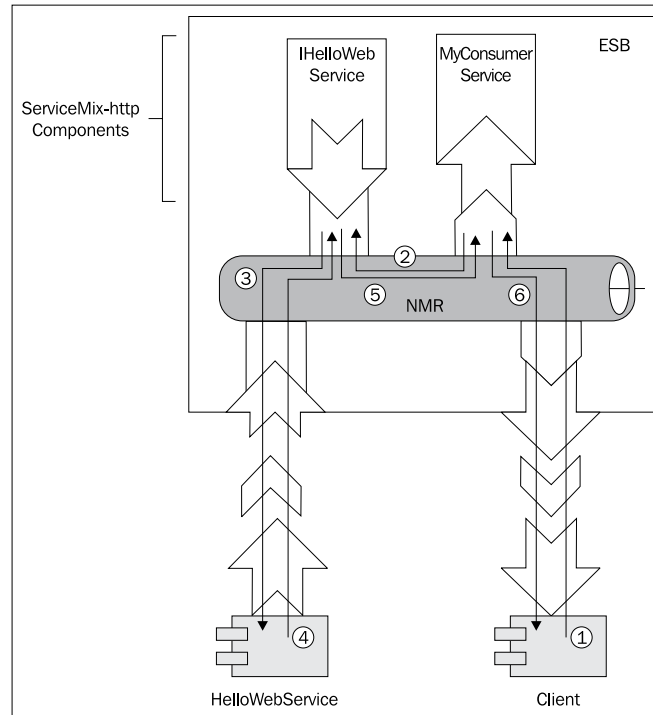
By using a web services gateway, you can use the intermediation to build and deploy the web services routing application. But keep in mind that the routing is just one of the various technical functionalities that you can implement at the gateway. For our sample use case, we have an external web service, deployed and hosted in a node remote to the ESB. In the ESB, we will set up a **Web Services Gateway**, which can proxy the remote web service. The entire setup is shown in the following figure:



Along with the previous discussion, we need the `servicemix-http` in the consumer and provider roles. `MyConsumerService` is a `servicemix-http` component in the consumer role and `IHelloWebService` is a `servicemix-http` component in the provider role. Both of them are shown in the following figure:



Let us now take a closer look at the gateway configured in the ESB. Here, we configure `servicemix-http` in both the consumer and provider roles and hook it to the NMR. Any client requests are intercepted by the consumer and the consumer then sends the request on behalf of the client to the NMR. From there the request will be routed to the destination web service through the provider. The message flow is marked in sequence in the following figure:



Deploy the Web Service

As a first step, if you haven't done it before, edit `examples.PROPERTIES` (provided along with the code download for this chapter), and change the paths there to match your development environment. The code download for this chapter also includes a `README.txt` file, which gives detailed steps to build and run the samples.

We have a simple web service in the codebase present in the folder `ch10\ServiceMixHttpBinding\01_ws`. To deploy the web service, first change directory to the `ch10\ServiceMixHttpBinding` folder and execute the `ant` command as follows:

```
cd ch10\ServiceMixHttpBinding
ant
```

In fact, the `build.xml` file will call the build in the subprojects to build the web service as well as the ServiceMix subproject.

The web service is built completely and the war file can be found in the folder `ch10\ServiceMixHttpBinding\01_ws\dist\AxisEndToEnd.war`. To deploy the web service, drop this war file into your favorite web server's webapps folder and restart the web server, if necessary.

Now to make sure that your web service deployment works fine, we have provided a web service test client. To invoke the test client, execute the following commands:

```
cd ch10\ServiceMixHttpBinding\01_ws
ant run
```

We can also check the web service deployment by accessing the WSDL from the URL:

```
http://localhost:8080/AxisEndToEnd/services/HelloWebService?WSDL
```

Let us list out the WSDL here, since we want to compare it with the WSDL accessed from the ServiceMix binding later to cross check the similarities. This is provided in `ch10\ServiceMixHttpBinding\HelloWebService-axis.wsdl`

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://AxisEndToEnd.axis.
    apache.binildas.com"
    xmlns:apachesoap="http://xml.apache.org/xml-soap"
    xmlns:impl="http://AxisEndToEnd.axis.apache.
        binildas.com"
    xmlns:intf="http://AxisEndToEnd.axis.apache.
        binildas.com"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/
        soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <wsdl:types>
    <schema elementFormDefault="qualified"
      targetNamespace="http://AxisEndToEnd.axis.apache.
        binildas.com"
      xmlns="http://www.w3.org/2001/XMLSchema">
      <element name="hello">
        <complexType>
          <sequence>
            <element name="in0" type="xsd:string"/>
          </sequence>
        </complexType>
      </element>
      <element name="helloResponse">
```

```
        <complexType>
            <sequence>
                <element name="helloReturn" type="xsd:string"/>
            </sequence>
        </complexType>
    </element>
</schema>
</wsdl:types>
<wsdl:message name="helloRequest">
    <wsdl:part element="impl:hello" name="parameters"/>
</wsdl:message>
<wsdl:message name="helloResponse">
    <wsdl:part element="impl:helloResponse" name="parameters"/>
</wsdl:message>
<wsdl:portType name="IHelloWeb">
    <wsdl:operation name="hello">
        <wsdl:input message="impl:helloRequest"
            name="helloRequest"/>
        <wsdl:output message="impl:helloResponse"
            name="helloResponse"/>
    </wsdl:operation>
</wsdl:portType>
<wsdl:binding name="HelloWebServiceSoapBinding"
    type="impl:IHelloWeb">
    <wsdlsoap:binding style="document" transport="http://schemas.
        xmlsoap.org/soap/http"/>
    <wsdl:operation name="hello">
        <wsdlsoap:operation soapAction=""/>
        <wsdl:input name="helloRequest">
            <wsdlsoap:body use="literal"/>
        </wsdl:input>
        <wsdl:output name="helloResponse">
            <wsdlsoap:body use="literal"/>
        </wsdl:output>
    </wsdl:operation>
</wsdl:binding>
<wsdl:service name="IHelloWebService">
    <wsdl:port binding="impl:HelloWebServiceSoapBinding"
        name="HelloWebService">
        <wsdlsoap:address location="http://localhost:8080/
            AxisEndToEnd/services/HelloWebService"/>
    </wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

XBean-based servicemix-http Binding

For XBean-based deployment of servicemix-http, our xbean.xml matches the following:

```
<beans xmlns:http="http://servicemix.apache.org/http/1.0"
      xmlns:test="http://AxisEndToEnd.axis.apache.binildas.com">
  <classpath>
    <location>./</location>
  </classpath>
  <http:endpoint service="test:IHelloWebService"
    endpoint="HelloWebService"
    role="provider"
    locationURI="http://localhost:8080/AxisEndToEnd/
                services/HelloWebService"

    soap="true"
    soapAction=" "
    wsdlResource="http://localhost:8080/AxisEndToEnd/
                services/HelloWebService?WSDL" />
  <http:endpoint service="test:MyConsumerService"
    endpoint="HelloWebService"
    role="consumer"
    targetService="test:IHelloWebService"
    locationURI="http://localhost:8081/services/
                HelloWebService"

    soap="true"
    defaultMep="http://www.w3.org/2004/08/wsdl/in-out"
    wsdlResource="http://localhost:8080/AxisEndToEnd/
                services/HelloWebService?WSDL" />
</beans>
```

The previous execution of ant has already built and packaged the service assembly for the sample.

Deploying and Running the Sample

To deploy the ServiceMix sample, we have the following servicemix.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:sm="http://servicemix.apache.org/config/1.0"
      xmlns:binil="http://www.binildas.com/voipservice">
  <bean id="propertyConfigurer"
    class="org.springframework.beans.factory.config.
            PropertyPlaceholderConfigurer">
    <property name="location"
      value="classpath:servicemix.properties" />
  </bean>
  <import resource="classpath:jmx.xml" />
  <import resource="classpath:jndi.xml" />
</beans>
```



```
<import resource="classpath:security.xml" />
<import resource="classpath:tx.xml" />
<sm:container id="jbi"
    MBeanServer="#jmxServer"
    useMBeanServer="true"
    createMBeanServer="true"
    rootDir="./wdir"
    installationDirPath="./install"
    deploymentDirPath="./deploy"
    flowName="seda">
    <sm:activationSpecs>
    </sm:activationSpecs>
</sm:container>
</beans>
```

To bring up the ServiceMix, change directory to `ch10\ServiceMixHttpBinding` and execute the ServiceMix script as follows.

```
cd ch10\ServiceMixHttpBinding
%SERVICEMIX_HOME%\bin\servicemix servicemix.xml
```

We can now test our ServiceMix deployment by using the following test client:

```
ch10\ServiceMixHttpBinding\Client.html
```

Access WSDL and Generate Axis Stubs to Access the Web Service Remotely

Now for the really cool stuff. As we discussed earlier, we have set up the ServiceMix as a separate web service gateway in front of the actual web service deployment. Now we have to check whether we can access the WSDL from the ServiceMix. For this, we can point our browser using the standard WSDL query string, like:

```
http://localhost:8081/services/HelloWebService/?wsdl
```

or

```
http://localhost:8081/services/HelloWebService/main.wsdl
```

Note that, the above URL points to the `locationURI` attribute configured for the consumer component, which is `http://localhost:8081/services/HelloWebService`. The WSDL placed in location `ch10\ServiceMixHttpBinding\HelloWebService-esb.wsdl`, matches the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:apachesoap="http://xml.apache.org/xml-soap"
    xmlns:impl="http://AxisEndToEnd.axis.apache.binildas.com"
    xmlns:intf="http://AxisEndToEnd.axis.apache.binildas.com"
```

```

xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://AxisEndToEnd.axis.apache.binildas.com">
<wsdl:types>
  <schema xmlns="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified"
    targetNamespace="http://AxisEndToEnd.axis.
      apache.binildas.com">

    <element name="hello">
      <complexType>
        <sequence>
          <element name="in0" type="xsd:string"/>
        </sequence>
      </complexType>
    </element>
    <element name="helloResponse">
      <complexType>
        <sequence>
          <element name="helloReturn" type="xsd:string"/>
        </sequence>
      </complexType>
    </element>
  </schema>
</wsdl:types>
<wsdl:message name="helloRequest">
  <wsdl:part element="impl:hello" name="parameters">
  </wsdl:part>
</wsdl:message>
<wsdl:message name="helloResponse">
  <wsdl:part element="impl:helloResponse" name="parameters">
  </wsdl:part>
</wsdl:message>
<wsdl:portType name="IHelloWeb">
  <wsdl:operation name="hello">
    <wsdl:input message="impl:helloRequest" name="helloRequest">
    </wsdl:input>
    <wsdl:output message="impl:helloResponse"
      name="helloResponse">
    </wsdl:output>
  </wsdl:operation>
</wsdl:portType>
<wsdl:binding name="HelloWebServiceBinding" type="impl:IHelloWeb">
  <wsdlsoap:binding style="document"
    transport="http://schemas.xmlsoap.org/
      soap/http"/>

  <wsdl:operation name="hello">
    <wsdlsoap:operation soapAction=""/>
    <wsdl:input name="helloRequest">
      <wsdlsoap:body use="literal"/>
    </wsdl:input>

```

```
        <wsdl:output name="helloResponse">
          <wsdlsoap:body use="literal"/>
        </wsdl:output>
      </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="MyConsumerService">
      <wsdl:port binding="impl:HelloWebServiceBinding"
        name="HelloWebService">
        <wsdlsoap:address location="http://localhost:8081/
          services/HelloWebService/" />
      </wsdl:port>
    </wsdl:service>
  </wsdl:definitions>
```

If we compare the two WSDL, the major difference is in the service description section. Here, ServiceMix forms the service and port name taking values from service and endpoint attributes of the consumer service – MyConsumerService and HelloWebService respectively.

If we are able to retrieve the WSDL, the next step is to use the Apache Axis tools to auto-generate the client-side stubs and binding classes, using which we can write simple Java client code to access the service through HTTP channel. The Axis client classes are placed in the directory `ch10\ServiceMixHttpBinding\03_AxisClient`.

To do that, we have to use the `wsdl2java` ant task. Let us first declare the task definition and execute that task to generate the stub classes.

```
<taskdef name="wsdl2java"
  classname="org.apache.axis.tools.ant.wsdl.Wsdl2javaAntTask"
  loaderref="axis" >
  <classpath refid="classpath"/>
</taskdef>
<target name="wsdl2java">
  <java classname="org.apache.axis.wsdl.WSDL2Java"
    fork="true"
    failonerror="true">
    <arg value="-o"/>
    <arg value="{src}"/>
    <arg value="-x"/>
    <arg value="http://io.java"/>
    <arg value="http://localhost:8081/services/HelloWebService/
      main.wsdl"/>
    <classpath>
      <path refid="classpath"/>
      <pathelement location="{build}"/>
    </classpath>
  </java>
</target>
```

The task will extract the WSDL from the specified location and generate the following client-side artifacts:

```
com\binildas\apache\axis\AxisEndToEnd\HelloWebServiceBindingStub.
java
com\binildas\apache\axis\AxisEndToEnd\IHelloWeb.java
com\binildas\apache\axis\AxisEndToEnd\MyConsumerService.java
com\binildas\apache\axis\AxisEndToEnd\MyConsumerServiceLocator.
java
```

The Client Java class can be written against these generated files as follows:

```
public class Client
{
    private static String wsdlUrl = "http://localhost:8081/services/
                                   HelloWebService/main.wsdl";
    private static String namespaceURI = "http://AxisEndToEnd.
                                   axis.apache.binildas.com";
    private static String localPart = "MyConsumerService";
    protected void executeClient(String[] args)throws Exception
    {
        MyConsumerService myConsumerService = null;
        IHelloWeb iHelloWeb = null;
        if(args.length == 3)
        {
            myConsumerService = new MyConsumerServiceLocator(args[0],
                                                             new QName(args[1], args[2]));
        }
        else
        {
            myConsumerService = new MyConsumerServiceLocator(wsdlUrl,
                                                             new QName(namespaceURI, localPart));
        }
        iHelloWeb = myConsumerService.getHelloWebService();
    }
    public static void main(String[] args)throws Exception
    {
        Client client = new Client();
        client.executeClient(args);
    }
}
```

To build the entire Axis client codebase, assuming that the ServiceMix is up and running, change directory to `ch10\ServiceMixHttpBinding\03_AxisClient`, which contains a `build.xml` file. Execute `ant` as shown as follows:

```
cd ch10\ServiceMixHttpBinding\03_AxisClient
ant
```

This will generate the required Axis client-side stubs and compile the client classes. Now to run the client, execute the following command:

```
ant run
```

Summary

We started this chapter by introducing the `servicemix-http` JBI component. Then we looked at the samples of binding web services to ESB using the `servicemix-http` binding component. By doing so, we have, in fact, implemented a complete functional web services gateway at the ESB.

A lot of times, we utilize this pattern to expose useful web services hosted deep inside your corporate networks protected by multiple levels of firewall. When we do so, the web services gateway is the access point for any external client. It should mock the actual web service not only in providing the functionality but also in exposing the web services contract (WSDL). Now, do you want to improve the QOS attributes of your web service?

The next chapter will take you through a similar exercise by demonstrating how to access your HTTP-based web services through an MOM channel like JMS.

11

Access Web Services Using the JMS Channel

Web services are great enablers for the SOA architectures which are neutral of the underlying platform and technology. It can also penetrate through the corporate firewalls, thus acting as a remote control switch. However, at times, we may want to guarantee a few QOS aspects of this service invocation. The reliability of the HTTP transport channel may not be sufficient for scenarios such as this. In this chapter, we will look at how Java JMS, which is a platform-dependent messaging technology, can increase the QOS features of the web services.

So we will look at the following in this chapter:

- What is JMS?
- Reliability and web services.
- SOAP versus JMS.
- JMS supporting components in ServiceMix.
- A protocol bridge to convert HTTP to JMS.
- A sample demonstrating the binding of web services to a JMS channel.

JMS

JMS defines the standard for a reliable enterprise messaging, also referred to as MOM. Enterprise messaging provides a reliable and flexible mechanism for the loosely coupled (asynchronous) exchange of critical business data and events throughout an enterprise. The JMS API adds to this a common API and a provider framework that enables the development of portable, message-based applications in the Java programming language.

The JMS API enhances J2EE in the following ways:

- Message-driven beans based on the JMS enable the asynchronous consumption of the JMS messages.
- JMS message exchange can participate in the Java Transaction API (JTA) transactions.
- The JCA interfaces allow JMS implementations from different vendors to be externally plugged into a J2EE environment.

Since MOM defines the backbone for many ESB implementations, JMS plays a critical role in Java-based ESB.

Web Service and JMS

Reliability is of the prime concern in critical applications, especially in interactions involving financial transactions. Consider the scenario of fund transfer where we debit one account and credit another account with the same amount of money. We cannot allow either of these transactions to fail or to debit twice in an account as a result of message duplications. Unreliable transport channels like HTTP and non-reliability in message delivery will impede the correctness of the mission critical transactions. This is where the alternatives to SOAP over HTTP, which is being explored.

Specifications for Web Service Reliable Messaging

The web services world defines two specifications to introduce reliability, namely:

- WS-Reliability.
- WS-Reliable Messaging.

Both these specifications use SOAP headers to exchange message grouping and correlation information between a consumer and a provider so that the reliability layer can guarantee the following quality concerns:

- Guaranteed delivery.
- Once and only once delivery.
- Message ordering.

Web Services Reliability (WS-Reliability) is a SOAP-based protocol with the purpose of exchanging SOAP messages with guaranteed delivery, no duplicates, and guaranteed message ordering. WS-Reliability is defined as SOAP header extensions

and is, in fact, independent of the underlying protocol used. At the same time, this specification contains a binding to HTTP. The reliable message protocol is abstracted from the communication between a sending Reliable Messaging Processor (RMP) and a receiving RMP. Hence the SOAP intermediaries do not play any active role in the reliability mechanisms. WS-Reliability was published in January 2003 by Hitachi, Oracle, Sonic, and Sun Microsystems, and then submitted to the OASIS Web Services Reliable Messaging Technical Committee.

The Web Services Reliable Messaging (WSRM) protocol is based on the WS-Reliable messaging specification and was published in March 2003 by software majors including BEA, IBM, Microsoft, and Tibco. WSRM also works similar to web services reliability. Instead of a group, the WSRM has the notion of a sequence to ensure message reliability concerns.

Depending upon the criticality of the business applications, varying orders of reliability might be required and WSRM-based products will support this. However, both the WSRM specifications are silent on many of the aspects. A few of these are listed as follows:

- Undeliverable messages – No definition of dead letter queue.
- Message priority.
- Message persistence.

The above aspects are out of the scope of the WSRM specifications, even though the implementations of reliable messaging needs to address all or any of them depending upon the order of reliability required. Traditional JMS-based MOM provides ways to address the above concerns, which we can leverage even in normal web services. It is in this context that we need to look into combining the MOM infrastructure with SOAP (Web Services), to get the best of both the worlds.

SOAP over HTTP versus SOAP over JMS

SOAP is transport independent and can be bound to any transport. The usual transport binding for SOAP is HTTP and SOAP over HTTP is what we usually look at as interoperable. HTTP belongs to the application layer of both the Open System Interface (OSI) model (layer 7) and the Internet Protocol (IP) suite (layer 4 or 5).

The following WSDL snippet shows what for a HTTP bound SOAP message looks like:

```
<wsdl:definitions>
  <wsdl:binding name="HelloWebServiceSoapBinding"
    type="impl:IHelloWeb">
```



```
<wsdlsoap:binding style="document"
                  transport="http://schemas.xmlsoap.org
                              /soap/http"/>
</wsdl:binding>
<wsdl:service name="IHelloWebService">
  <wsdl:port binding="impl:HelloWebServiceSoapBinding"
            name="HelloWebService">
    <wsdlsoap:address location="http://localhost:8080/
                          AxisEndToEnd/services/HelloWebService"/>
  </wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

Due to the many QOS features of JMS, SOAP over JMS offers more reliable and scalable messaging support than SOAP over HTTP. By building on top of the JMS, SOAP over JMS supports two messaging styles — one-way request style and two-way request and response style. One-way request messaging allows a web service client to unblock itself when the request message reaches a JMS queue or topic. Two-way request and response messaging blocks a web service client until the request reaches the server and a response message is received back at the client-side. A sample WSDL snippet is shown as follows:

```
<wsdl:definitions>
  <wsdl:binding name="HelloWebServiceSoapBinding"
                type="impl:IHelloWeb">
    <wsdlsoap:binding style="document"
                      transport="http://schemas.xmlsoap.org/
                                soap/jms"/>
  </wsdl:binding>
  <wsdl:service name="IHelloWebService">
    <wsdl:port binding="impl:HelloWebServiceSoapBinding"
              name="HelloWebService">
      <wsdlsoap:address location="jms:/queue?
                              destination=jms/queue&
                              connectionFactory=weblogic.jndi.
                              WLInitialContextFactory"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

But the downside is that SOAP over JMS is not yet standardized and hence may not be interoperable across platforms. This is because JMS is not an "over-the-wire" protocol. Instead, JMS is a Java API which requires a client to use a JMS provider library (a jar file) provided by the vendor of the JMS service provider. This is analogous to requiring a JDBC driver similar to `classes12.jar` from Oracle for connecting to an Oracle server from a java application. The actual "over-the-wire" protocol to be used under

the covers within the JMS provider library is not defined and is left open for vendors to have their own implementations. The interoperability is not easy even if we consider the java world alone. That is, if we have an IBM Websphere JMS provider and a client program from within the BEA Weblogic application server need to access the JMS server. Even though both the platforms are java-based, the integration might not be straightforward due to the JMS implementation clashes. Moreover, JMS being an API doesn't ask vendors to implement any additional services and doesn't standardize the data structure exchanged. However, JMS still is a viable mechanism to MOM that enables web services.

JMS in ServiceMix

ServiceMix provides us with the `servicemix-jms` component which makes it easy to bind the endpoints to the JMS channel, both in the consumer and provider roles. Hence, before we look into the details on how to bind the web services to the JMS transport, we will look at configuring the `servicemix-jms` component.

ServiceMix-jms

The `servicemix-jms` components allow you to send and receive JMS messages. The `servicemix-jms` components assume that the normalized message they are given is ready for marshalling into or out of JMS. Hence, they don't, by default, try to implement a SOAP stack or perform any complex message transformation other than to map normalized messages to JMS or vice versa. However, it is possible to specify the SOAP enveloped payload as messages so that `servicemix-jms` can perform wrapping and unwrapping of payload from within the SOAP envelope.

A few of the features of `servicemix-jms` are as follows:

- Standard JBI-compliant binding component.
- Supports lightweight and XBean-based deployments.
- Supports SOAP 1.1 and 1.2 support.
- Supports MIME with attachments.
- Supports WS-Addressing.
- Support for all MEPs in the consumer or the provider role.

Consumer and Provider Roles

`servicemix-jms` can be configured both as a consumer and a provider of services. Similar to `servicemix-http`, these roles are with respect to the NMR of the ESB. In other words, a consumer role implies that the component is a consumer to the NMR whereas a provider role implies that the NMR is the consumer to the component. Based on these roles, the NMR will take responsibility of any format or protocol conversions for the interacting components. You can refer to the *Consumer and Provider Roles* section in Chapter 10 to understand more on the contract and responsibility of these roles.

`servicemix-jms` XBean Configuration

The `servicemix-jms` component supports the XBean-based deployment. Since the `servicemix-jms` component can be configured in both the consumer and provider roles, we have two sets of configuration parameters for the component. Let us look into the main configuration parameters listed as follows:

- **`servicemix-jms` as consumer:** A sample `servicemix-jms` consumer component configuration is shown as follows:

```
<jms:endpoint service="test:MyConsumerService"
  endpoint="myConsumer"
  role="consumer"
  soap="true"
  targetService="test:pipeline"
  defaultMep="http://www.w3.org/2004/08/wsdl/in-only"
  destinationStyle="queue"
  jmsProviderDestinationName="queue/A"
  connectionFactory="#connectionFactory" />
```

The following table lists out the main attributes used to configure `servicemix-jms` component in the consumer role:

Attribute Name	Type	Description	Mandatory or Not
service	QName	Service name of proxy endpoint.	Mandatory.
endpoint	String	Endpoint name of proxy endpoint.	Mandatory.
role	String	Whether a consumer or a provider.	Mandatory. Value should be consumer.
soap	boolean	If it is true, the component will parse the SOAP envelope and pass the contents to the NMR.	Not Mandatory. Default value is false.

Attribute Name	Type	Description	Mandatory or Not
targetService	QName	Service name of the target endpoint.	Not Mandatory. Default is the value of the service attribute.
defaultMEP	URI	The MEP URI by which clients interact with the consumer component.	Not Mandatory.
destinationStyle	String	Indicates the destination type used with the jmsProviderDestinationName.	Not Mandatory (unless jmsProviderDestinationName is used).
jmsProviderDestinationName	String	The target JMS destination (Queue or Topic) will be created by the JMS provider.	Not Mandatory (either of destination, jndiDestinationName or jmsProviderDestinationName is mandatory).
connectionFactory	javax.jms. Connection Factory	The connectionFactory is to be used. Instead a JNDI configuration can be provided using jndiConnectionFactoryName.	Not Mandatory.
useMsgIdInResponse	boolean	True value indicates that the JMS correlation id will be set to the id of the JMS request message – if it is false, an artificial correlation id will be used instead.	Not Mandatory (in which case the default behaviour is to use the message exchange id as the correlation id).

In the sample configuration listed earlier, if a client has to send a message to the NMR it has to place the message in the queue queue/A. Since the MEP is in-only, the client will not receive any response. Hence, if a client has to send a normal request-response to the NMR, then we need to use additional MEP conversion bridges (Note: the `targetService` in the sample configuration which is an MEP conversion pipeline, which will be demonstrated in an example in this chapter). Even that is not sufficient since we need a mechanism to co-relate request and response messages. This can be done by setting the JMS correlation id (even though we don't demonstrate this in our sample).

- **servicemix-jms as provider:** A sample `servicemix-jms` provider component configuration is shown as follows:

```
<jms:endpoint service="test:MyProviderService"
  endpoint="myProvider"
  role="provider"
  soap="true"
```

```
destinationStyle="queue"
jmsProviderDestinationName="queue/B"
connectionFactory="#connectionFactory" />
```

The following table lists out the main attributes used to configure the `servicemix-jms` component in the provider role:

Attribute Name	Type	Description	Mandatory or Not
service	QName	Service name of the exposed endpoint.	Mandatory.
endpoint	String	Endpoint name of the exposed endpoint.	Mandatory.
role	String	Whether a consumer or a provider.	Mandatory. Value should be provider.
soap	boolean	If it is true, the component will parse the SOAP envelope and pass the contents to the NMR.	Not Mandatory. Default value is false.
destinationStyle	String	Indicates the destination type used with the <code>jmsProviderDestinationName</code> .	Not Mandatory (unless <code>jmsProviderDestinationName</code> is used).
jmsProviderDestinationName	String	The target JMS destination (Queue or Topic) will be created by the JMS provider.	Not Mandatory (either of <code>destination</code> , <code>jndiDestinationName</code> , or <code>jmsProviderDestinationName</code> is mandatory).
connectionFactory	javax.jms. Connection Factory	The <code>connectionFactory</code> is to be used. Instead a JNDI configuration can be provided using <code>jndiConnectionFactoryName</code> .	Not Mandatory.

servicemix-jms Lightweight Configuration

In addition to the XBean-based configuration listed earlier, `servicemix-jms` can also be deployed based on the lightweight mode for use in an embedded ServiceMix. The configuration would be as follows:

```
<sm:activationSpec>
  <sm:component>
    <jms:component>
      <jms:endpoints>
        <jms:endpoint service="test:MyConsumerService">
```

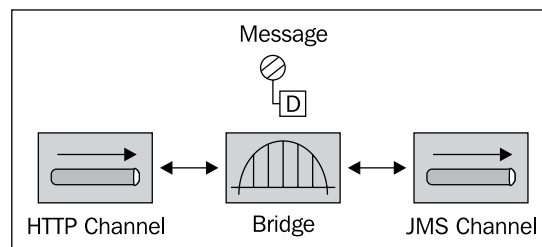
```

        endpoint="myConsumer"
        role="consumer"
        soap="true"
        targetService="test:pipeline"
        defaultMep="http://www.w3.org/2004/08/
                    wsdl/in-only"
        destinationStyle="queue"
        jmsProviderDestinationName="queue/A"
        connectionFactory="#connectionFactory" />
<jms:endpoint service="test:MyProviderService"
        endpoint="myProvider"
        role="provider"
        soap="true"
        destinationStyle="queue"
        jmsProviderDestinationName="queue/B"
        connectionFactory="#connectionFactory" />
</jms:endpoints>
</jms:component>
</sm:component>
</sm:activationSpec>

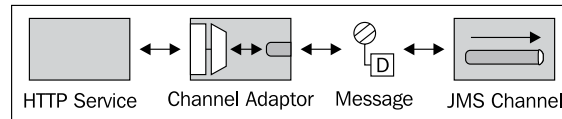
```

Protocol Bridge

A protocol bridge is an integration pattern to connect between two different protocols. For example, HTTP and JMS are two different transport protocols. If there was a way to bridge these two, we could leverage the best of both worlds (open standard nature of HTTP and reliable transport feature of JMS). Typically, it is not straightforward to connect two different protocols and exchange information. We need to connect individual, corresponding channels speaking different protocols in the messaging system. This is demonstrated in the following figure:



A protocol bridge can be viewed as a combination of channel adapters. An adapter can function as a client to the messaging system. With the client role, the adapter can invoke application functions through the application supplied interfaces. Thus a HTTP channel adapter can be used to access the HTTP service and publish messages on a JMS channel. Similarly, the same adapter can also receive messages from the JMS channel and invoke the functionality over the HTTP service.



Along those lines, we can now think of a web service adapter to translate between the HTTP-based web service and the JMS-based messaging system. By doing so, the SOAP formatted messages can be routed through the traditional HTTP channel through firewalls over the Internet, and also through the messaging channel over the intranet for access by interdepartmental systems.

For example, you can think of a scenario where an enterprise provides a "Product Query" service or an "Inventory Update" service. Such services can be accessed by interdepartmental or LOB systems such as the "Order Entry" system or "Shipping" system over JMS. If the enterprise wants to expose these services over the firewall for B2B interaction with other enterprises, they can do so in the SOAP over HTTP style of interaction. For both these two types of interaction, we don't need to define two separate services. Instead, using an adapter, the same service can be made accessible through multiple channels. The web services gateway provided by the IBM Websphere Application server is a first class example of a web service adapter.

Web Service in the JMS Channel Binding Sample

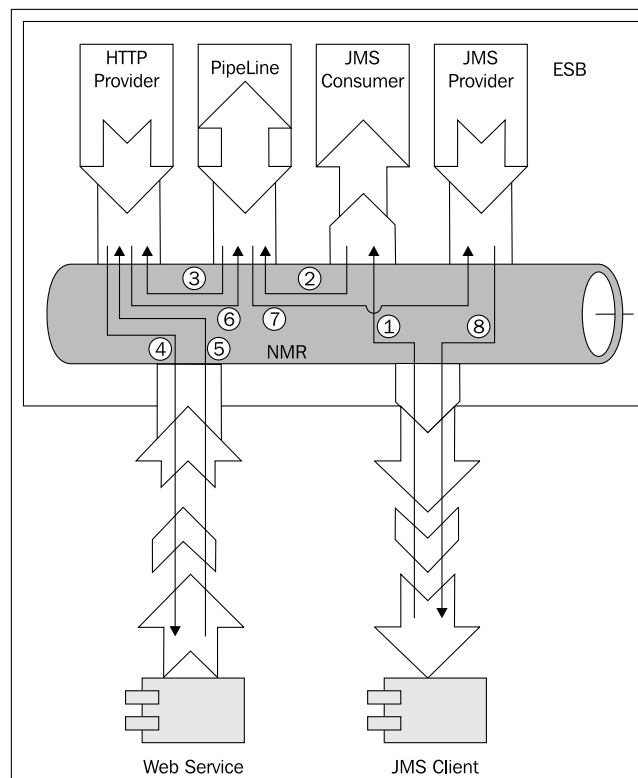
We will now look at a complete sample of how to bind a web service using the JMS channel to ServiceMix. While doing so, we will also see how to use the Apache Axis client-side APIs to send a request to and receive a response from the web service, through the JMS channel rather than the normal HTTP channel. We may not configure all of the QOS features for the JMS provider here such as transaction, message persistence, or guaranteed delivery. Most of them are outside the standard J2EE configurations and have to be done at the JMS provider-level based on the vendor-specific mechanisms. Since this book is about JBI and ServiceMix and not JMS, we will concentrate only on the binding part. Once we are successful in binding the web service to JMS, then enabling other QOS features is similar to what we do in the normal JMS configurations.

ServiceMix Component Architecture for the JMS Web Service

We will first look at the technical architecture for the whole component setup to see how we can route messages through various ServiceMix components. The major parties or roles taking part in the exchange are as follows:

- External client (sending request and accepting response).
- JMS consumer (request queue for client).
- Pipeline bridge.
- HTTP provider.
- External web service.
- JMS provider (response queue for client).

As we know, all the components are bound appropriately to the NMR and all the message exchanges take place through the NMR. The following figure shows the component architecture:



As shown in the above figure, when the client sends a message, the message-flow through the NMR through various JBI components are marked by numbers in sequence. You may note the dynamics of the **Pipeline** component in the previous figure. Here, the Pipeline will send the input message in an In-Out MEP to the HTTP Provider destination and then in turn forward the response in an In-Only MEP to the JMS Provider.

An aspect to notice in the above architecture is that the whole interaction is in a request-response style from the consumer (client) perspective. However, at the transport-level we implement this as a combination of a In-Only request and its corresponding response.

Let us now set up the individual components shown in the technical architecture figure.

Deploy the Web Service

As a first step, if you haven't done it before, edit `examples.PROPERTIES` (provided along with the code download for this chapter), and change the paths there to match your development environment. The code download for this chapter also includes a `README.txt` file, which gives the detailed steps to build and run the samples.

We have a simple web service in the codebase present in the folder `ch11\WebServiceInJmsChannel\01_ws`. To deploy the web service, first change the directory to `ch11\WebServiceInJmsChannel\01_ws` and execute `ant` as shown as follows:

```
cd ch11\WebServiceInJmsChannel\01_ws
ant
```

The web service will be completely built and the war file can be found in the folder:

```
ch11\WebServiceInJmsChannel\01_ws\dist\AxisEndToEnd.war.
```

To deploy the web service, just drop this war file into your favorite web server's `webapps` folder and restart the web server, if necessary.

Now to make sure that your web service deployment works fine, we have provided two test clients. To invoke the test client run the following commands:

```
cd ch11\ServiceMixHttpBinding\01_ws
ant run

and/or

ant test
```

We can also check the web service deployment by accessing the WSDL from the following URL:

```
http://localhost:8080/AxisEndToEnd/services/HelloWebService?WSDL
```

The top-level folder, that is `ch11\WebServiceInJmsChannel`, will have a single `build.xml` file which will build all the sub projects in a single go. To build the entire sample, change directory to this folder and execute `ant` as follows:

```
cd ch11\WebServiceInJmsChannel
ant
```

XBean-based servicemix-jms Binding

For XBean-based deployment of `servicemix-jms`, our `xbean.xml` file looks like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:jms="http://servicemix.apache.org/jms/1.0"
       xmlns:test="http://AxisEndToEnd.axis.apache.binildas.com">
  <classpath>
    <location>./</location>
  </classpath>
  <bean id="connectionFactory"
        class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL" value="tcp://localhost:61616" />
  </bean>
  <jms:endpoint service="test:MyConsumerService"
                endpoint="myConsumer"
                role="consumer"
                soap="true"
                targetService="test:pipeline"
                defaultMep="http://www.w3.org/2004/08/wsdl/in-only"
                destinationStyle="queue"
                jmsProviderDestinationName="queue/A"
                connectionFactory="#connectionFactory" />
  <jms:endpoint service="test:MyProviderService"
                endpoint="myProvider"
                role="provider"
                soap="true"
                destinationStyle="queue"
                jmsProviderDestinationName="queue/B"
                connectionFactory="#connectionFactory" />
</beans>
```

Perhaps, you might have noticed the fact that for the above consumer role the `targetService` is not the actual web service, but a pipeline which is explained next.

XBean-based servicemix-eip Pipeline Bridge

The pipeline component is an integration bridge between an In-Only (or Robust-In-Only) MEP and an In-Out MEP. By receiving an In-Only MEP by the pipeline, we will send the input message in an In-Out MEP to the transformer destination and then in turn forward the response in an In-Only MEP to the target destination. The pipeline is a standard EIP component and hence is available readily as a standard JBI EIP component in ServiceMix.

The pipeline EIP component can be configured using XBean and deployed. The configuration `xbean.xml` file is as shown as follows:

```
<beans xmlns:http="http://servicemix.apache.org/http/1.0"
      xmlns:eip="http://servicemix.apache.org/eip/1.0"
      xmlns:test="http://AxisEndToEnd.axis.apache.binildas.com">
  <classpath>
    <location>./</location>
  </classpath>
  <eip:pipeline service="test:pipeline" endpoint="pipeline">
    <eip:transformer>
      <eip:exchange-target service="test:IHelloWebService" />
    </eip:transformer>
    <eip:target>
      <eip:exchange-target service="test:MyProviderService" />
    </eip:target>
  </eip:pipeline>
</beans>
```

XBean-based servicemix-http Provider Destination

We now require an In-Out MEP-based target transformer as the message destination. Since our target service is a web service, it makes sense to use a `servicemix-http` component in the provider role to point to the external web service. The `xbean.xml` configuration is as follows:

```
<beans xmlns:http="http://servicemix.apache.org/http/1.0"
      xmlns:test="http://AxisEndToEnd.axis.apache.binildas.com">
  <classpath>
    <location>./</location>
```

```

</classpath>
<http:endpoint service="test:IHelloWebService"
  endpoint="HelloWebService"
  role="provider"
  locationURI="http://localhost:8080/AxisEndToEnd/
               ervices/HelloWebService"
  soap="true"
  soapAction=""
  wsdlResource="http://localhost:8080/AxisEndToEnd/
               services/HelloWebService?WSDL" />

</beans>

```

Deploying the Sample and Starting ServiceMix

To deploy the ServiceMix sample, we have the following servicemix.xml:

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:sm="http://servicemix.apache.org/config/1.0"
  xmlns:test="http://AxisEndToEnd.axis.apache.binildas.com" >
  <import resource="classpath:jmx.xml" />
  <import resource="classpath:jndi.xml" />
  <import resource="classpath:security.xml" />
  <import resource="classpath:tx.xml" />
  <import resource="classpath:activemq.xml" />
  <bean id="propertyConfigurer"
    class="org.springframework.beans.factory.config.
           PropertyPlaceholderConfigurer">
    <property name="location"
      value="classpath:servicemix.properties" />
  </bean>
  <sm:container id="jbi"
    MBeanServer="#jmxServer"
    useMBeanServer="true"
    createMBeanServer="true"
    rootDir="./wdir"
    installationDirPath="./install"
    deploymentDirPath="./deploy"
    flowName="seda">
    <sm:activationSpecs>
    </sm:activationSpecs>
  </sm:container>
</beans>

```

To bring up ServiceMix, change directory to `ch11\WebServiceInJmsChannel` and bring up the ServiceMix container.

```

cd ch11\WebServiceInJmsChannel
%SERVICEMIX_HOME%\bin\servicemix servicemix.xml

```

Test Web Service Using JMS Channel

To test the web service deployed sending messages through JMS channel, we will have two approaches as follows:

- **Test using JMS client – document style:** This approach is simple and straightforward. We can create a simple JMS client which can place a SOAP request document in the input queue. The same client will also listen to an output queue so that whenever (within a time limit) a response message is received back in this queue then the client can consume it.

```
public class JMSClient
{
    public static String MESSAGE_1 = "<?xml version=\"1.0\"
                                     encoding=\"UTF-8\"?> SOAP Message...";
    private static final long WAIT_TIME = 5 * 1000L;
    public static void main(String[] args) throws Exception
    {
        ActiveMQConnectionFactory factory = new
            ActiveMQConnectionFactory("tcp://localhost:61616");
        ActiveMQQueue pubTopic = new ActiveMQQueue("queue/A");
        ActiveMQQueue subTopic = new ActiveMQQueue("queue/B");
        Connection connection = factory.createConnection();
        Session session = connection.createSession(false,
            Session.AUTO_ACKNOWLEDGE);
        MessageProducer producer = session.createProducer(pubTopic);
        MessageConsumer consumer = session.createConsumer(subTopic);
        connection.start();
        producer.send(session.createTextMessage(MESSAGE_1));
        TextMessage textMessage = (TextMessage)
            consumer.receive(1000 * 10);
        if(textMessage == null)
        {
            System.out.println("Response timed out.");
        }
        else
        {
            System.out.println("Response was: " +
                textMessage.getText());
        }
        System.out.println("Closing.");
        connection.close();
    }
}
```

The JMSClient is placed in the folder `ch11\WebServiceInJmsChannel\03_BindJms\src`. The client class was already compiled when we built the full project. To run the test client, change directory to `ch11\WebServiceInJmsChannel\03_BindJms` and execute `ant` as follows:

```
cd ch11\WebServiceInJmsChannel\03_BindJms
ant run
```

- **Test using Axis client – RPC style:** In the previous example, we followed a pure document-oriented approach to invoke the web service and get back the response. Due to the fact that the transport channel is JMS which is detached and asynchronous, we may not be able to change the style fully from the document-oriented approach. However, we can make the invocation close to an RPC style by using the Apache Axis service and calling objects.

We need to have multiple artifacts to invoke the web service in RPC style, and they are listed here in detail:

- **JMSTestClientRPCWebService.java:** This is the main client class used to invoke the web service. This class is placed in the `ch11\WebServiceInJmsChannel\05_AxisClient\src\com\binildas\apache\axis\AxisEndToEnd` folder.

```
public class JMSTestClientRPCWebService
{
    public static void main(String args[]) throws Exception
    {
        org.apache.axis.client.Service axisServiceObj =
            new org.apache.axis.client.Service();
        org.apache.axis.client.Call axisCall =
            (org.apache.axis.client.Call)axisServiceObj.
                createCall();
        axisCall.setOperationName("hello");
        axisCall.addParameter("in0", org.apache.axis.
            encoding.XMLType.XSD_STRING, javax.xml.rpc.
                ParameterMode.IN );
        axisCall.setReturnType(org.apache.axis.encoding.
            XMLType.XSD_STRING);
        org.apache.axis.client.Transport transport = new
            JMSTransportForAxis();
        axisCall.setTransport(transport);
        axisCall.setProperty("REQUEST_QUEUE", "queue/A");
        axisCall.setProperty("RESPONSE_QUEUE", "queue/B");
        String res = (String) axisCall.invoke(new Object[]
            {"Binil"});
        System.out.println("res: " + res);
    }
}
```

As we can see in the code listing, we first create an Axis `Call` instance and set the operation name (which is the method name of the remote web service) and the input parameters to invoke the operation. We also need to set the return type details so that the `Call` instance can unmarshal any return type received from the transport channel to the suitable Java type.

The next important step is setting the transport class for the `Call` object. Here, we set an instance of `JMSTransportForAxis`, which provides the gateway to the transport channel.

- **JMSTransportForAxis.java:** This class extends the Apache Axis Transport class:

```
public class JMSTransportForAxis extends org.apache.axis.  
                                client.Transport  
{  
    public JMSTransportForAxis()  
    {  
        transportName = "JMSTransportForAxis";  
    }  
}
```

- **client-config.wsdd:** The next important piece is the `client-config.wsdd`, which should be there in the classpath while we invoke the `JMSTestClientRPCWebService` class. This configuration is placed in `ch11\WebServiceInJmsChannel\05_AxisClient\config`.

```
<?xml version="1.0" encoding="UTF-8"?>  
<deployment name="defaultClientConfig"  
    xmlns="http://xml.apache.org/axis/wsdd/"  
    xmlns:java="http://xml.apache.org/axis/wsdd/  
                providers/java">  
    <transport name="http"  
        pivot="java:org.apache.axis.transport.http.  
                HTTPSender"/>  
    <transport name="local"  
        pivot="java:org.apache.axis.transport.local.  
                LocalSender"/>  
    <transport name="java"  
        pivot="java:org.apache.axis.transport.java.  
                JavaSender"/>  
    <handler name="JMSSender"  
        type="java:org.apache.axis.transport.jms.  
                JMSSender" />  
    <transport name="JMSTransport" pivot="JMSSender"/>  
    <handler name="CustomJMSSender"  
        type="java:com.binildas.apache.axis.  
                AxisEndToEnd.JMSSender" />
```

```

    <transport name="JMSTransportForAxis" pivot=
        "CustomJMSSender"/>
</deployment>

```

The lines to be noted from the above configuration are reproduced as follows:

```

<handler name="CustomJMSSender"
    type="java:com.binildas.apache.axis.
        AxisEndToEnd.JMSSender" />
<transport name="JMSTransportForAxis" pivot=
    "CustomJMSSender"/>

```

If we observe the JMSTransportForAxis class, we can see that the value of the super class (Transport) field (transportName) is set as "JMSTransport-ForAxis". In the client-config.wsdd, we then pivot CustomJMSSender against JMSTransportForAxis value. Now, we again map the CustomJMSSender to a custom transport sender class java:com.binildas.apache.axis.AxisEndToEnd.JMSSender

- JMSSender.java: This is the class where the actual transport plumbing happens and this call is not much different from the JMSSender class we already saw in the previous test in the sense that we will have an input and an output queue to place a request and to read any response.

```

public class JMSSender extends org.apache.axis.handlers.
    BasicHandler
{
    public void invoke(org.apache.axis.MessageContext
        msgContext) throws org.apache.axis.AxisFault
    {
        try
        {
            ActiveMQConnectionFactory factory = new
                ActiveMQConnectionFactory("tcp://
                    localhost:61616");
            Object requestDestination = msgContext.
                getProperty("REQUEST_QUEUE");
            Object responseDestination = msgContext.
                getProperty("RESPONSE_QUEUE");
            ActiveMQQueue pubTopic = new ActiveMQQueue
                ((String) requestDestination);
            ActiveMQQueue subTopic = new ActiveMQQueue
                ((String) responseDestination);
            Connection connection = factory.
                createConnection();
            Session session = connection.createSession(false,
                Session.AUTO_ACKNOWLEDGE);

```



```
        MessageProducer producer = session.createProducer
            (pubTopic);
        MessageConsumer consumer = session.createConsumer
            (subTopic);

        connection.start();
        String reqSOAPMsgString = msgContext
            getRequestMessage().getSOAPPartAsString();
        producer.send(session.createTextMessage
            (reqSOAPMsgString));
        TextMessage m = (TextMessage) consumer.receive
            (1000*10);

        String respMessageStr = null;
        if( m == null )
        {
            System.out.println("Response timed out.");
        }
        else
        {
            respMessageStr = m.getText();
            System.out.println("Response was : " +
                respMessageStr);
        }
        System.out.println("Closing.");
        connection.close();
        org.apache.axis.Message respSoapMessage = new
            org.apache.axis.Message(respMessageStr);
        msgContext.setResponseMessage(respSoapMessage);
    }
    catch (Exception e)
    {
        throw new org.apache.axis.AxisFault
            ("failedSend", e);
    }
}
```

The previous project build has already built all the sub projects, including the one containing all the above client classes. So, to run this client, we need to change folder to `ch11\WebServiceInJmsChannel\05_AxisClient` and execute ant as follows:

```
cd ch11\WebServiceInJmsChannel\05_AxisClient
ant run
```

Summary

MOM including JMS is a great enabler for reliable communication between components, especially when you do that in a loosely coupled (asynchronous) manner. JMS provides the required APIs and provider-level SPIs for Java components to interact through MOM. Combining the power of messaging over a reliable channel along with the interoperability of web services provides us a greater flexibility with confidence in messaging characteristics. Web services over JMS are positioned in this space and it is nothing new since we have been doing that for many enterprise class transactions. The new thing here is the endless possibilities provided by the ESB architecture when combined with tested and proven EAI patterns. This is demonstrated in this chapter with samples. And keep reading—you are going to see more practical usages of the ESB architecture such as web service versioning in the coming chapters.

12

Java XML Binding using XStream

While the Java programming language provides us a means to write portable code, XML can be used to define portable data. We use XML extensively to format data in SOA-based architectures. Moreover, today all new generation platforms, frameworks, and even legacy platforms such as COBOL and Mainframes exhibit support for XML formatted data.

ServiceMix is all about SOI and hence it is also concerned with portable data. Naturally, the format of data inside the NMR is XML. Another aspect is that ServiceMix is a Java-based JBI framework. Hence, the developers need to write code in Java, whether they are SEs or BCs. It is in this context that the relationship between Java and XML in the ServiceMix context needs attention.

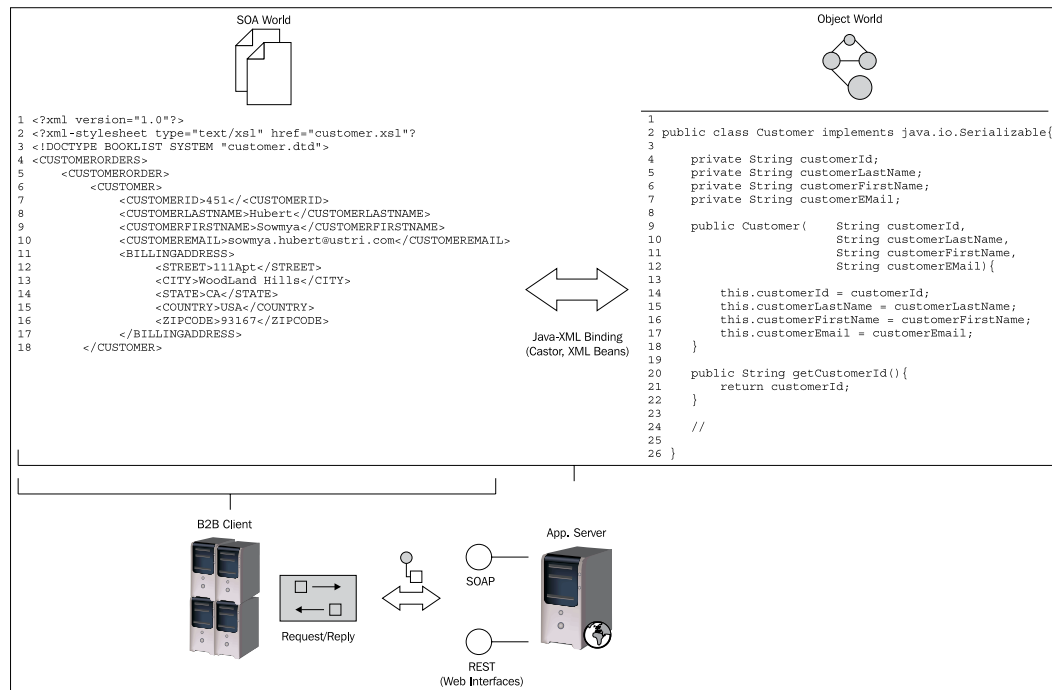
This chapter will provide a brief introduction to Java XML binding and to the concepts and technologies that it employs.

So we will cover the following in this chapter:

- Java XML binding in general
- Java XML binding frameworks including XStream
- XStream integration with the ServiceMix ESB
- Working code sample showing XStream in action in ServiceMix

Java XML Binding

Java XML binding deals with transforming the XML instances to the Java instances and vice versa. Even though we can do this by writing Java code from scratch against the XML APIs, today we have multiple tools and frameworks which will do the same.



The above figure shows a typical scenario we might come across in B2B interactions. Leave the advanced validations or CRUD operations one can do in the XML documents alone, we are interested in the marshalling and unmarshalling functionality of the JAXB. XML is the de-facto wire format in SOA and SOI. If we need to process the XML data from within our Java components, we have to do some form of XML binding. The frameworks such as Castor and XMLBeans do exactly this. JAXB is the Java reference implementation for Java XML binding. Today, we have XStream which will do the same functionality quickly. Let us see how these frameworks are relevant in the JBI discussion.

JAXB

Java API for XML Binding (JAXB) provides a convenient way to process XML content using Java objects by binding its XML schema to Java representation. JAXB provides an API and the required tool sets that automate the mapping between the XML documents and the Java instances. Thus we can list out the main features as:

- Unmarshal the XML instance into a Java instance.
- Access, update, and validate the Java representation against schema constraint.
- Marshal the Java instance of the XML content into the XML document instance.

XStream

XStream is a simple, open source library to serialize Java objects to the XML and back again. It is lightweight in the sense that it doesn't require much configuration or mapping files. XStream has a good integration with the ServiceMix. In fact, ServiceMix provides an XStream backed API itself named JavaSource.

The main features of XStream are listed as follows:

- **Ease for use:** A high-level facade-class called XStream is supplied which simplifies the common use cases.
- **No mappings required:** Most objects can be serialized by registering their class.
- **Performance:** XStream is suitable for large object graphs or systems with high message throughput.
- **Works on normal objects:** Serializes internal fields, including private and final. Supports non-public and inner classes. The classes are not required to have a default constructor.
- **Full object graph support:** Duplicate references encountered in the object-model will be maintained. Supports circular references.
- **Integrates with other XML APIs:** By implementing suitable interfaces, XStream supports serialization directly to or from any tree structure including XML.

Let us also look at a sample binding using XStream. Consider the following two classes namely the Customer and the Contact:

```
public class Customer
{
    private String firstName;
    private String lastName;
    private Contact phone;
}
public class Contact
{
    private int code;
    private String number;
}
```

Now, if you have to use XStream to convert the above entities to XML, then first instantiate the XStream façade and create aliases for your custom class names to XML element names as shown in the following code:

```
XStream xStream = new XStream();
xStream.alias("customer", Customer.class);
xStream.alias("contact", Contact.class);
```

Now it is a matter of creating the entity tree, populating its fields, and then calling the toXML method in XStream. This is reproduced in the following code:

```
Customer binil = new Customer("Binil", "Das");
binil.setPhone(new Contact(91, "471-2700888"));
String xml = xStream.toXML(binil);
```

The XML output will look like this:

```
<customer>
  <firstname>Binil</firstname>
  <lastname>Das</lastname>
  <phone>
    <code>91</code>
    <number>471-2700888</number>
  </phone>
</customer>
```

To reconstruct the object tree back from XML is easy, and is shown as follows:

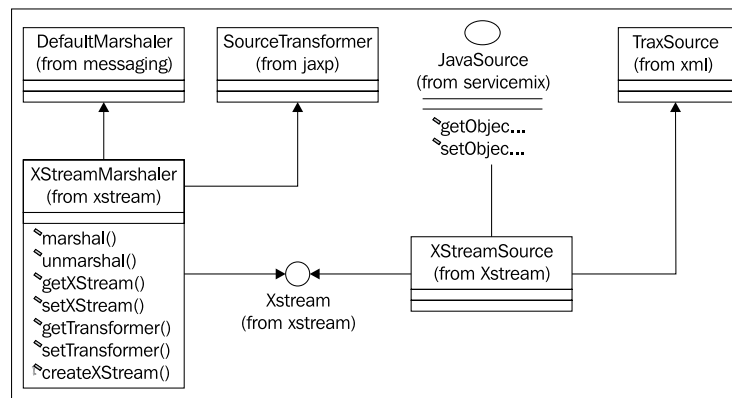
```
Customer binilBack = (Customer) xstream.fromXML(xml);
```

ServiceMix and XStream

ServiceMix has good integration with XStream as a mechanism for XML to Java binding and vice versa. This is made possible in ServiceMix by exposing a few APIs, the main ones are listed as follows:

- `org.apache.servicemix.jbi.messaging.DefaultMarshaler`
- `org.apache.servicemix.components.util.xstream.XStreamMarshaler`
- `org.apache.servicemix.components.util.xstream.XStreamSource`
- `org.apache.servicemix.JavaSource`

The relationship between these classes with `com.thoughtworks.xstream.XStream` is shown in the following figure:



Now, if we need to integrate with XStream in our custom transformation components, we can do so by creating an instance of XStreamSource.

```

XStream xStream = new XStream();
xStream.alias("ServiceParamTO", ServiceParamTO.class);
xStream.alias("CustomerTO", CustomerTO.class);
// Register other classes...
JavaSource JavaSource = new XStreamSource(payload, xStream);
  
```

Now, we just set this instance of XStreamSource as the in message to the `inOut` method and send the message exchange.

```

normalizedMessageIn.setContent(javaSource);
inOut.setInMessage(normalizedMessageIn);
sendSync(inOut);
  
```


As you know, the NMR of ServiceMix always deals with the normalized message format, which is the XML format. The required plumbing will be done by the XStream in the background.

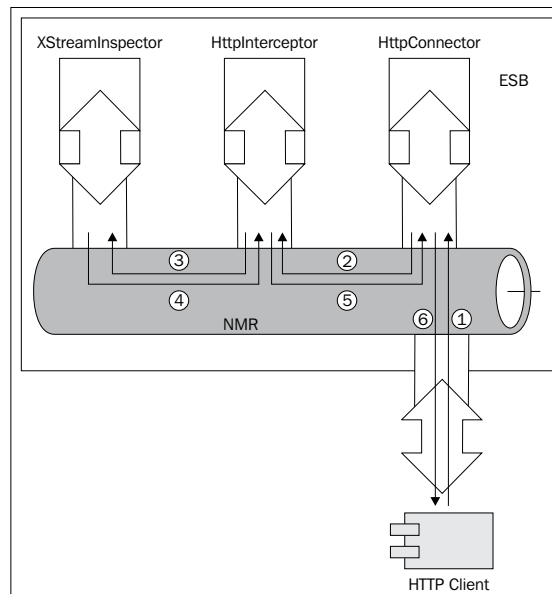
XStream in a Normalized Message Router Sample

In Chapter 7, you have learnt how to code and build custom components to be deployed in the ServiceMix container. There we used an `HttpInterceptor` class to intercept the contents of the JBI exchange and print them out on the console. We will enhance the code from that sample to demonstrate how we can integrate XStream with other JBI classes in ServiceMix.

Sample Use Case

In the SOA scenarios, we use the XML formatted messages to transport payload across nodes. Whether SOAP formatted or not, XML provides a flexible mechanism to transport data, which can be validated if required using the XML schemas. Let us also build our sample around this. Hence, we will assume that we have an external client sending an XML payload to the ESB. Let us see how we can interact with this data inside the bus.

This is diagrammatically represented in the following figure:

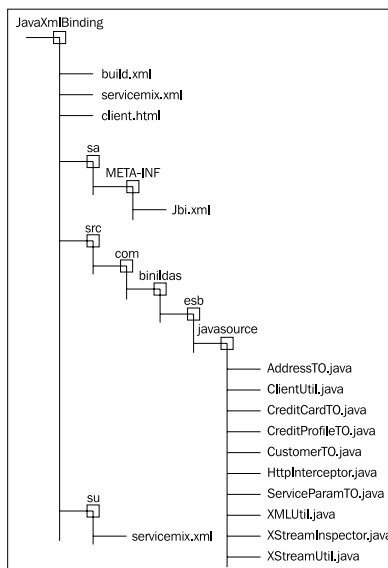


The sample use case will consist of the following components:

- **HTTP Client:** The HTTP Client is a client external to the ESB which interact with the HTTP Connector inside the ESB to send the request message and get the response back.
- **HTTP Connector:** The HTTP Connector is a ServiceMix `HttpConnector` component configured to listen to a specific port. Any messages arriving at the HTTP Connector will be directed to the next component in the flow, namely the HTTP Inspector.
- **HTTP Inspector:** The HTTP Inspector is a ServiceMix `Transform` Component. This component, as the name implies, will first intercept the message contents. The XML formatted message sent by the HTTP Client will be intercepted here, and then printed out to the console. We then unmarshal the XML instance into a Java instance using the XStream. Next comes the key sequence in this sample – we set the Java instance into the InOut of the message exchange in the next chain of the interaction, to be sent to the destination service, XStream Inspector.
- **XStream Inspector:** In XStream Inspector, we try to retrieve back the message content and print it to the console. Then XStream Inspector sends the same XML content back to the HTTP Inspector.

In the reverse stream flow, at HTTP Inspector we again print out the content received from the XStream Inspector to the console, and then send back the same XML to the HTTP Client.

The following figure lists how the various code artifacts for the sample are organized:



Code HTTPClient

HTTPClient is a normal HTTP Client and is coded in the file `ch12\JavaXmlBinding\Client.html`. This component is capable of sending the following XML request to the URL: `http://localhost:8912`.

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
              xmlns:tns="http://servicemix.apache.org/samples/wsdl-
                           first/types">
  <env:Body>
    <ServiceParamTO>
      <customerTO>
        <firstName>Ann</firstName>
        <lastName>Binil</lastName>
        <addressTO>
          <houseNumber>222</houseNumber>
          <street>Lake View</street>
          <city>Cochin</city>
        </addressTO>
      </customerTO>
      <creditCardTO>
        <cardNumber>8888-9999-1111-2222</cardNumber>
        <validTill>01-APR-2007</validTill>
        <cardType>Master Card</cardType>
      </creditCardTO>
    </ServiceParamTO>
  </env:Body>
</env:Envelope>
```

Unmarshalling to Transfer Objects

We have a set of Java TO classes so that the XStream can unmarshal the above XML document to the TO instances and back to the XML whenever required. This is shown in the following list:

- `ch12\JavaXmlBinding\src\com\binildas\esb\javasource\ServiceParamTO.java`

```
public class ServiceParamTO implements Serializable
{
    private CustomerTO customerTO;
    private CreditCardTO creditCardTO;
}
```

- ch12\JavaXmlBinding\src\com\binildas\esb\javasource\CustomerTO.
 java

```

public class CustomerTO implements Serializable
{
    private String firstName;
    private String lastName;
    private AddressTO addressTO;
}

```
- ch12\JavaXmlBinding\src\com\binildas\esb\javasource\AddressTO.
 java

```

public class AddressTO implements Serializable{
    private String houseNumber;
    private String street;
    private String city;
}

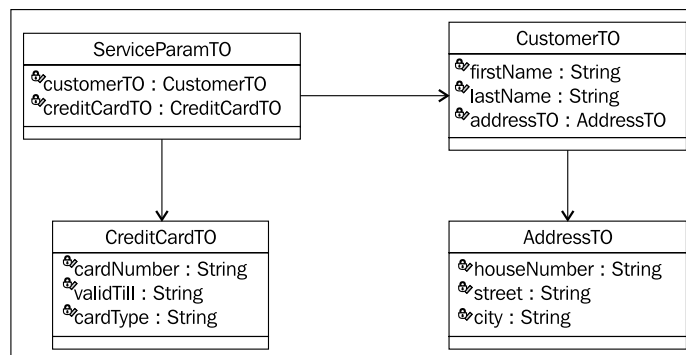
```
- ch12\JavaXmlBinding\src\com\binildas\esb\javasource\CreditCardTO.java

```

public class CreditCardTO implements Serializable{
    private String cardNumber;
    private String validTill;
    private String cardType;
}

```

These classes are related as shown in the following UML class diagram:



HttpInterceptor Component

The `HttpInterceptor` is a `ServiceMix Transform` component. This component will first intercept the message contents. The XML formatted message sent by the HTTP client will be intercepted here, and then printed out to the console. We then unmarshal the XML instance into a Java instance using `XStream`. The Java instance is then sent through the NMR to the next component in the flow, which is the `XStreamInspector`. Let us look into the code shown as follows:

```
public class HttpInterceptor extends TransformComponentSupport
{
    protected boolean transform(MessageExchange exchange,
        NormalizedMessage in, NormalizedMessage out) throws
        MessagingException
    {
        System.out.println("HttpInterceptor(" + name + ").transform01.
            exchange.getService() = " + exchange.getService());
        XStream xStream = null;
        String contentString = null;
        ServiceParamTO payLoad = null;
        JavaSource javaSource = null;
        QName service = null;
        InOut inOut = null;
        NormalizedMessage normalizedMessageIn = null;
        NormalizedMessage normalizedMessageOut = null;
        NormalizedMessage copyReturnMessage = null;
        Source contentReturn = null;
        InputStream inputStream = null;
        byte[] bytes = null;
        int available = 0;
        String contentReturnString = null;
        NormalizedMessage copyMessage = exchange.createMessage();
        getMessageTransformer().transform(exchange, in, copyMessage);
        Source content = copyMessage.getContent();
        if (content instanceof DOMSource) {
            contentString = XMLUtil.retreiveSoapContent(((DOMSource)
                content).getNode());
            System.out.println("HttpInterceptor(" + name +
                ").transform02.contentString = " + contentString);
            payLoad = (ServiceParamTO)
                XStreamUtil.xmlToObject(contentString);
            xStream = new XStream();
            xStream.alias("ServiceParamTO", ServiceParamTO.class);
            xStream.alias("CustomerTO", CustomerTO.class);
            xStream.alias("CreditCardTO", CreditCardTO.class);
            xStream.alias("AddressTO", AddressTO.class);
            javaSource = new XStreamSource(payLoad, xStream);
            service = new QName(namespaceURI, localPart);
            inOut = createInOutExchange(service, null, null);
        }
    }
}
```

```

        normalizedMessageIn = inOut.createMessage();
        normalizedMessageIn.setContent(javaSource);
        inOut.setInMessage(normalizedMessageIn);
        sendSync(inOut);
        normalizedMessageOut = inOut.getOutMessage();
        copyReturnMessage = exchange.createMessage();
        getMessageTransformer().transform(exchange,
            normalizedMessageOut, copyReturnMessage);
        contentReturn = copyReturnMessage.getContent();
        if (contentReturn instanceof StringSource)
        {
            try
            {
                inputStream = ((StringSource) contentReturn).
                    getInputStream();
                available = inputStream.available();
                bytes = new byte[available];
                inputStream.read(bytes);
            }
            catch(IOException ioException)
            {
                throw new MessagingException(ioException);
            }
            contentReturnString = new String(bytes);
            System.out.println("HttpInterceptor(" + name + ").
                transform03.contentReturnString = " +
                    contentReturnString);

            out.setContent(contentReturn);
            System.out.println("HttpInterceptor(" + name + ").
                transform04.contentReturnString = " +
                    contentReturnString);
        }
        System.out.println("HttpInterceptor(" + name + ").
            transform05. End");
    }
    return true;
}
}

```

In the code, we create an instance of `XStream` and register all TO classes to `XStream`. Then using this `XStream` instance and the XML payload, we create an instance of `XStreamSource`. The `XStreamSource` instance is a `JavaSource` type, which again is a `javax.xml.transform.Source` type. This makes it easy for us to set this `JavaSource` into the `InOut` of the message exchange in the next chain of the interaction, to be sent to the destination service which is `XStreamInspector`.

XStreamInspector Component

Compared to `HttpInterceptor`, the `XStreamInspector` class is simple. It retrieves back the message content and prints it to the console. The `XStreamInspector` class then sends the same XML content back to `HttpInspector`.

```
public class XStreamInspector extends TransformComponentSupport
{
    protected boolean transform(MessageExchange exchange,
        NormalizedMessage in, NormalizedMessage out) throws
        MessagingException
    {
        System.out.println("XStreamInspector(" + name + ").transform01.
            exchange.getService() = " + exchange.getService());
        NormalizedMessage copyMessage = exchange.createMessage();
        getMessageTransformer().transform(exchange, in, copyMessage);
        Source content = copyMessage.getContent();
        String contentString = null;
        if (content instanceof DOMSource)
        {
            contentString = XMLUtil.node2XML(((DOMSource) content).
                getNode());
            System.out.println("XStreamInspector(" + name + ").
                transform02.contentString = " + contentString);
        }
        out.setContent(new StringSource(contentString));
        System.out.println("XStreamInspector(" + name + ").transform03.
            End");
        return true;
    }
}
```

Configure Interceptor and Inspector Components

We can configure the `HttpInterceptor` and `XStreamInspector` components in a single SU. Hence, we will do the configuration of these components as an SU in the `servicemix.xml` file kept at `ch12\JavaXmlBinding\sus\servicemix.xml`.

The contents of this file is shown in the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:sm="http://servicemix.apache.org/config/1.0"
       xmlns:demo="http://www.binildas.com/esb/LightWeightOrPojo">
  <classpath>
    <location>./</location>
  </classpath>
  <sm:serviceunit id="jbi">
    <sm:activationSpecs>
      <sm:activationSpec componentName="interceptor"
                        endpoint="interceptor"
                        service="demo:interceptor">
        <sm:component>
          <bean class="com.binildas.esb.
                javasource.HttpInterceptor">
            <property name="name">
              <value>Interceptor-1</value>
            </property>
            <property name="namespaceURI">
              <value>
                http://www.binildas.com/esb/LightWeightOrPojo
              </value>
            </property>
            <property name="localPart">
              <value>inspector</value>
            </property>
          </bean>
        </sm:component>
      </sm:activationSpec>
      <sm:activationSpec componentName="inspector"
                        endpoint="inspector"
                        service="demo:inspector">
        <sm:component>
          <bean class="com.binildas.esb.javasource.
                XStreamInspector">
            <property name="name">
              <value>Inspector-1</value>
            </property>
          </bean>
        </sm:component>
      </sm:activationSpec>
    </sm:activationSpecs>
  </sm:serviceunit>
</beans>
```


Package Interceptor and Inspector Components

We will create an SU and then package it into an SA. We have already seen the SU configuration, let us now look into the SA configuration at:

ch12\JavaXmlBinding\sa\META-INF\jbi.xml

The jbi.xml file is reproduced in the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<jbi xmlns="http://java.sun.com/xml/ns/jbi" version="1.0">
  <service-assembly>
    <identification>
      <name>InterceptorAssembly</name>
      <description>Interceptor Service Assembly</description>
    </identification>
    <service-unit>
      <identification>
        <name>Interceptor</name>
        <description>Interceptor Service Unit</description>
      </identification>
      <target>
        <artifacts-zip>Interceptor-su.zip</artifacts-zip>
        <component-name>servicemix-lwcontainer</component-name>
      </target>
    </service-unit>
  </service-assembly>
</jbi>
```

Deploy Interceptor and Inspector Components

The main point to be noted in the SA jbi.xml is the target element of the service-unit. Here we specify that the SU artifact (that is Interceptor-su.zip) is to be deployed into the servicemix-lwcontainer target container.

To do this, we have servicemix.xml file in the topmost folder to start the ServiceMix container. This is shown as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:sm="http://servicemix.apache.org/config/1.0"
  xmlns:demo="http://www.binildas.com/esb/LightWeightOrPojo">
  <bean id="propertyConfigurer"
    class="org.springframework.beans.factory.config.
      PropertyPlaceholderConfigurer">
```

```

        <property name="location"
            value="classpath:servicemix.properties" />
    </bean>
    <import resource="classpath:jmx.xml" />
    <import resource="classpath:jndi.xml" />
    <import resource="classpath:security.xml" />
    <import resource="classpath:tx.xml" />
    <sm:container id="jbi"
        rootDir="./wdir"
        installationDirPath="./install"
        deploymentDirPath="./deploy"
        flowName="seda"
        monitorInstallationDirectory="true"
        createMBeanServer="true"
        useMBeanServer="true">
        <sm:activationSpecs>
            <sm:activationSpec componentName="httpReceiver"
                service="bt:httpBinding"
                endpoint="httpReceiver"
                destinationService="demo:interceptor">
                <sm:component>
                    <bean class="org.apache.servicemix.components.http.
                        HttpConnector">
                        <property name="host" value="127.0.0.1"/>
                        <property name="port" value="8912"/>
                    </bean>
                </sm:component>
            </sm:activationSpec>
        </sm:activationSpecs>
    </sm:container>
</beans>

```

Here you can see that we configure an `HttpConnector` to listen at port 8912. It is to this target that the HTTP Client sends the XML request.

Build and Run the Sample

As a first step, if you haven't done it before, edit `examples.PROPERTIES` (provided along with the code download for this chapter), and change the paths there to match your development environment. The code download for this chapter also includes a `README.txt` file, which gives detailed steps to build and run the samples.

To build and run the sample, first change directory to `ch12\JavaXmlBinding` folder and execute `ant` as shown here:

```
cd ch12\JavaXmlBinding
ant
```

We can bring up ServiceMix by running the following commands:

```
cd ch12\JavaXmlBinding
%SERVICEMIX_HOME%/bin/servicemix servicemix.xml
```

When we start ServiceMix, the JBI container is configured using the above `servicemix.xml` file.

To run the demo, there is a `Client.html` file provided in the top-level folder.

Summary

You have already deployed POJO components into ServiceMix and exposed them as services. An external client can invoke the POJO services by sending SOAP requests and receiving back the SOAP responses. At times, you may also need to deal with non-SOAP formatted, but plain XML messages. We also need to stream such messages too through firewalls to the bus and get them processed.

This chapter showed you how we can do this using XStream. Some legacy integration scenario might warrant this approach. You might also have noted the fact that we can replace the XStream used in this sample with any other Java XML binding framework such as Castor or XMLBeans, but XStream's advantage is the built-in integration XStream has with the ServiceMix JBI.

The ServiceMix JBI bus provides a framework for many lightweight integration libraries like XStream. It also realizes many design patterns used in software engineering like the well known Proxy pattern which we will explore in the next chapter.

13

JBIProxy

One of the most useful classes introduced by JDK 1.3 is the `Proxy` class in the `java.lang.reflect` package. It allows us to create classes implementing a particular type (interface) on the fly. ServiceMix provides a similar API in JBI so that we can proxy a particular service in the JBI bus. This helps us to implement many functionalities such as:

- Intercepting and re-routing the services.
- Wrapping and unwrapping messages targeted to a service.
- Using request-message, formatted for a particular service type, for a different service type.

In this chapter, we will first revisit JDK `Proxy` classes. This will set a background for further reading wherein we will explain JBI Proxy in detail with examples for multiple scenarios. Then the developer will be able to make use of Proxy pattern within the JBI-based ESB, to suit their technical requirements.

We will cover the following in this chapter:

- Proxy design pattern in general
- Proxy support in Java SDK with examples
- ServiceMix JBI Proxy
- A few samples of defining and exposing proxies to services in the JBI bus
- A practical use of JBI Proxy – to proxy external web services in the JBI bus

Proxy—A Primer

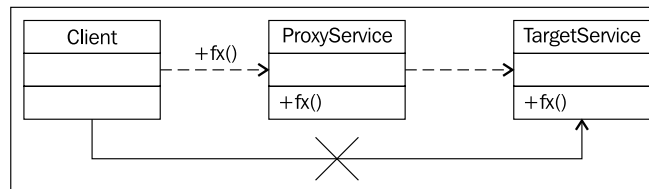
Wikipedia defines Proxy as:

Proxy may refer to something which acts on behalf of something else.

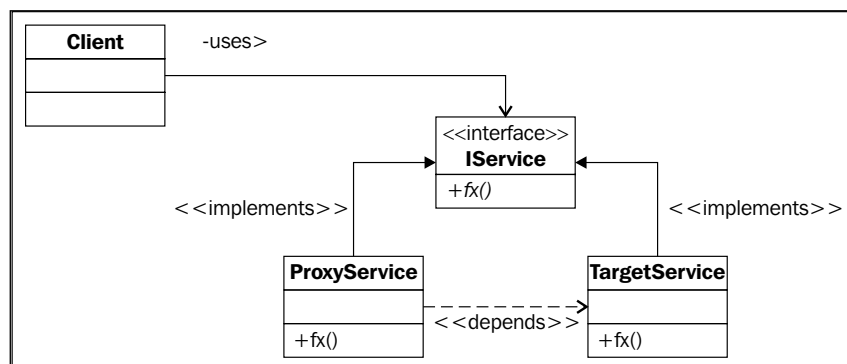
In the software a proxy is a substitute for a target instance and is a general pattern which appears in many other patterns in different variants.

Proxy Design Pattern

A proxy is a surrogate class for the target object. If a method call has to be invoked in the target object, it happens indirectly through the proxy object. The feature which makes proxy ideal for many situations is that the client or the caller is not aware that it is dealing with the proxy object. The proxy class is shown in the following figure:



In the above figure, when a client invokes a method target towards the Target service, the proxy intercepts the call in between. The proxy also expose a similar interface to the target, hence the client is unaware of the dealing with the proxy. Thus the proxy method is invoked. The proxy then delegates the call to the actual target since it cannot provide the actual functionality. When doing so, the proxy can provide call management towards the actual method. The entire dynamics is shown in the following figure:



A proxy is usually implemented by using a common, shared interface or super class. Both the proxy and the target share this common interface. Then, the proxy delegates the calls to the target class.

JDK Proxy Class

JDK provides both the class `Proxy` and the interface `InvocationHandler` in the `java.lang.reflect` package, since version 1.3. Using JDK Proxy classes, you can create your own classes implementing multiple interfaces of your choice, at run time.

`Proxy` is the super class for any dynamic proxy instances you create at run time. Moreover, the `Proxy` class also accommodates a host of static methods which will help you to create your proxy instances. `getProxyClass` and `newProxyInstance` are two such utility methods.

The Proxy API is listed in the following in brevity:

```
package java.lang.reflect;
public class Proxy implements java.io.Serializable
{
    protected InvocationHandler h;
    protected Proxy(InvocationHandler h);
    public static InvocationHandler getInvocationHandler(Object proxy)
        throws IllegalArgumentException;
    public static Class<?> getProxyClass(ClassLoader loader,
        Class<?>... interfaces)
        throws IllegalArgumentException;
    public static boolean isProxyClass(Class<?> cl);
    public static Object newProxyInstance(ClassLoader loader,
        Class<?>[] interfaces, InvocationHandler h)
        throws IllegalArgumentException
}
```

In the above code, you can invoke the `Proxy.getProxyClass` with a class loader and an array of interfaces for which you need to proxy, to get a `Class` instance for the proxy. Proxy objects have one constructor, to which you pass an `InvocationHandler` object associated with that proxy. When you invoke a method on the proxy instance, the method invocation is encoded and dispatched to the invoke method of its invocation handler. Let us also look at the `InvocationHandler` API reproduced as follows:

```
package java.lang.reflect;
public interface InvocationHandler
{
    Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable;
}
```

We need to implement this interface and provide code for the `invoke` method. Once you get a `Class` instance for the proxy by invoking the `Proxy.getProxyClass` with a class loader and an array of interfaces for which you need to proxy to. Now, you can get a `Constructor` object for this proxy from the `Class` instance. On the constructor you can use `newInstance` (passing in an invocation handler instance) to create the proxy instance. The created instance should be implementing all the interfaces that were passed to `getProxyClass`. The steps are shown in the following code:

```
InvocationHandler handler = new SomeInvocationHandler(...);
Class proxyClazz = Proxy.getProxyClass(Blah.class.getClassLoader(),
                                       new Class[] {Blah.class});
Blah blah = (Blah) proxyClazz.getConstructor(new Class[] {
    InvocationHandler.class }).newInstance(new Object[] {
    handler});
```

There is also a shortcut to get a proxy object. You can invoke `Proxy.newProxyInstance`, which takes a class loader, an array of interface classes, and an invocation handler instance.

```
InvocationHandler handler = new SomeInvocationHandler(...);
Blah blah = (Blah) Proxy.newProxyInstance(Blah.class.
    getClassLoader(), new Class[] {Blah.class},
    handler);
```

Now you can invoke methods on the proxy object during which these method invocations are turned into calls on to the invocation handler's `invoke` method is shown here:

```
blah.interfaceMethod();
```

Sample JDK Proxy Class

We will now write some simple code to demonstrate how you can write your own proxies at run time, for your interface classes.

As a first step, if you haven't done it before, edit `examples.PROPERTIES` (provided along with the code download for this chapter), and change the paths there to match your development environment. The code download for this chapter also includes a `README.txt` file, which gives detailed steps to build and run the samples.

We will now look at the source code that can be found in the folder `ch13\JdkProxy\src`.

The files are explained here:

ch13\JdkProxy\src\SimpleIntf.java

```
public interface SimpleIntf
{
    public void print();
}
```

SimpleIntf is a simple interface with a single method print. print does not accept any parameters and also does not return any value. Our aim is that when we invoke methods on the proxy object for SimpleIntf, the method invocation should be turned into calls to an invocation handler's invoke method. Let us now define an invocation handler in the following code:

ch13\JdkProxy\src\SimpleInvocationHandler.java

```
import java.lang.reflect.InvocationHandler;
import java.io.Serializable;
import java.lang.reflect.Method;
public class SimpleInvocationHandler implements InvocationHandler,
    Serializable
{
    public SimpleInvocationHandler() {}
    public Object invoke(final Object obj, Method method,
        Object[] args) throws Throwable
    {
        if (method.getName().equals("print") && (args == null
            || args.length == 0))
        {
            System.out.println("SimpleInvocationHandler.invoked");
        }
        else
        {
            throw new IllegalArgumentException("Interface method does
                not support param(s) : " + args);
        }
        return null;
    }
}
```

Since SimpleIntf.print() does not accept any parameters and also does not return any value, in the invoke method of SimpleInvocationHandler, we double check the intention behind the actual invoker. In other words, we check that no parameters are passed and we return null only.

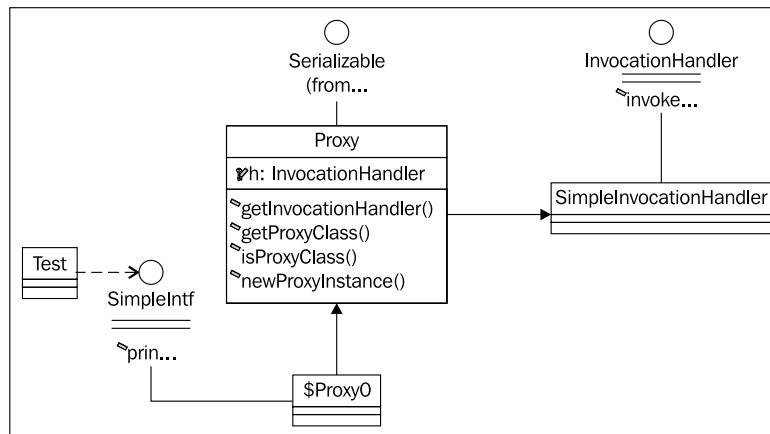
Now, we have all the necessary classes to implement a proxy for SimpleIntf interface. Let us now execute it by writing a Test class.

ch13\JdkProxy\src\Test.java

```
import java.lang.reflect.Proxy;
import java.lang.reflect.InvocationHandler;
public class Test
{
    public static void main(String[] args)
    {
        InvocationHandler handler = new SimpleInvocationHandler();
        SimpleIntf simpleIntf = (SimpleIntf)Proxy.newProxyInstance
            (SimpleIntf.class.getClassLoader(),new Class[] { SimpleIntf.
                                                         class }, handler);

        simpleIntf.print();
    }
}
```

The wiring of the above described interfaces and classes are better represented in the UML class diagram in the following figure:

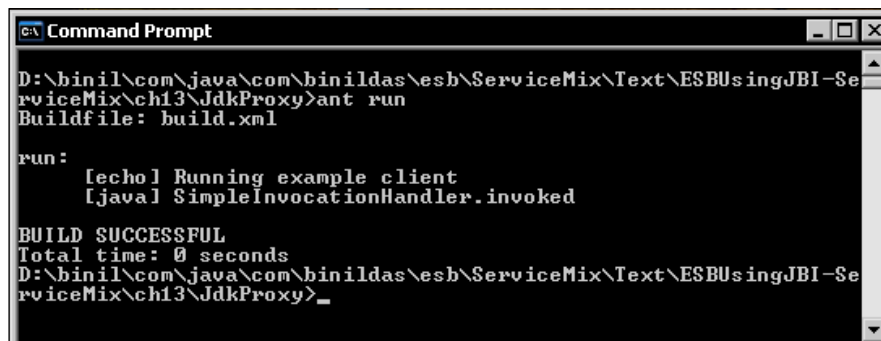


The above figure shows the relationship between various classes and interfaces in the sample. \$Proxy0 class represents the actual proxy class generated on the fly and as you can deduce it from the class diagram. \$Proxy0 is a type of our interface (SimpleIntf).

To build the sample, first change directory to `ch13\JdkProxy` and execute `ant` as shown here:

```
cd ch13\JdkProxy
ant
```

The command `ant run` will execute the `Test` class which will print out the following in the console:



```
Command Prompt
D:\binil\com\java\com\binildas\esb\ServiceMix\Text\ESBUsingJBI-SeviceMix\ch13\JdkProxy>ant run
Buildfile: build.xml

run:
[jecho] Running example client
[jjava] SimpleInvocationHandler.invoke

BUILD SUCCESSFUL
Total time: 0 seconds
D:\binil\com\java\com\binildas\esb\ServiceMix\Text\ESBUsingJBI-SeviceMix\ch13\JdkProxy>
```

ServiceMix JBI Proxy

Java proxies for the JBI endpoints can be created in ServiceMix using JSR181 components. For this, the requirement is that the JBI endpoints should expose a WSDL.

A `jsr181:endpoint` takes a value for the `serviceInterface` attribute. The JBI container will be able to generate the WSDL out of this `serviceInterface`. Thus, if we have a `jsr181:endpoint` exposing service to the JBI bus, it is possible to provide a proxy for that service too.

The basic configuration for defining a JBI proxy is shown as follows:

```
<jsr181:proxy id="proxyBean"
  container="#jbi"
  interfaceName="test:HelloPortType"
  type="test.Hello" />
```

Once a proxy is defined, the same can then be referenced from your client bean or from one of your components. The proxied JBI endpoint can then be invoked just like a normal POJO.

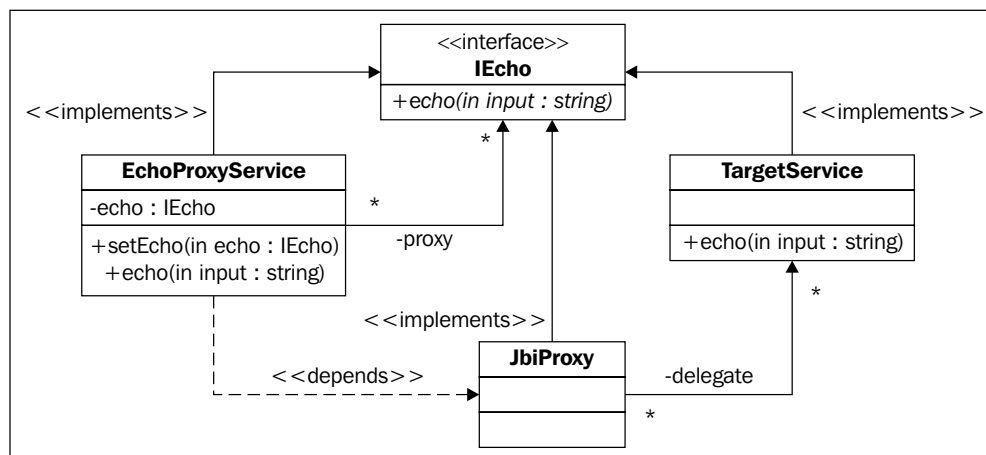
If you want to define a JBI proxy within a SU, you can follow the configuration given as follows:

```
<jsr181:endpoint annotations="none"
  service="test:echoService"
  serviceInterface="test.Echo">
  <jsr181:pojo>
    <bean class="test.EchoProxy">
      <property name="echo">
        <jsr181:proxy service="test:EchoService"
          context="#context"
          type="test.IService" />
      </property>
    </bean>
  </jsr181:pojo>
</jsr181:endpoint>
```

Let us now look into a few examples to make the concept clearer.

JBI Proxy Sample Implementing Compatible Interface

First, we will create a JBI proxy implementing an interface compatible with the target service. Then, in place of the target service we will use the proxy instance, so that any calls intended for the target service will be first routed to the proxy. The proxy in turn will delegate the call to the target service. The structural relationship between various classes participating in the interaction is shown in the following figure:



Here, `EchoProxyService` is the class which we later expose in the JBI bus as the service. This class implements the `IEcho` interface. In order to demonstrate the proxy, `EchoProxyService` doesn't implement the service as such, instead depends on the `JbiProxy` derived out of another class `TargetService`. The `TargetService` contains the actual service code. As you can see, both the `EchoProxyService` and the `TargetService` implement the same interface.

Proxy Code Listing

The codebase for the sample is located in the folder `ch13\JbiProxy\01_CompatibleInterface\01_JsrProxy\src`.

This folder contains an interface `IEcho` and two other classes implementing the `IEcho` interface namely `EchoProxyService` and `TargetService`. These classes are explained here:

- **IEcho.java:** The `IEcho` interface declares a single method `echo` which takes a `String` parameter and returns a `String`.

```
public interface IEcho
{
    public String echo(String input);
}
```

- **EchoProxyService.java:** `EchoProxyService` is a convenient class which will act as mechanism for routing requests to the JBI proxy. Moreover, `EchoProxyService` implements the above interface `IEcho`.

```
public class EchoProxyService implements IEcho
{
    private IEcho echo;
    public void setEcho(IEcho echo)
    {
        this.echo = echo;
    }
    public String echo(String input)
    {
        System.out.println("EchoProxyService.echo. this = " + this);
        return echo.echo(input);
    }
}
```

- **TargetService.java:** TargetService also implements the interface IEcho. TargetService is supposed to be our target service, and we will be generating a JBI proxy for the TargetService.

```
public class TargetService implements IEcho
{
    public String echo(String input)
    {
        System.out.println("TargetService.echo : String. this = " +
                           this);
        return input;
    }
}
```

XBean-based JBI Proxy Binding

Using XBean, we will now configure the JBI proxy to be deployed onto the standard servicemix-jsr181 JBI component. The xbean.xml is as shown as follows:

```
<beans xmlns:sm="http://servicemix.apache.org/config/1.0"
        xmlns:jsr181="http://servicemix.apache.org/jsr181/1.0"
        xmlns:http="http://servicemix.apache.org/http/1.0"
        xmlns:test="http://test">
    <classpath>
        <location>./</location>
    </classpath>
    <jsr181:endpoint annotations="none"
                    service="test:echoService"
                    serviceInterface="test.IEcho">
        <jsr181:pojo>
            <bean class="test.EchoProxyService">
                <property name="echo">
                    <jsr181:proxy service="test:TargetService"
                                context="#context"
                                type="test.IEcho" />
                </property>
            </bean>
        </jsr181:pojo>
    </jsr181:endpoint>
    <jsr181:endpoint annotations="none"
                    service="test:TargetService"
                    serviceInterface="test.IEcho">
        <jsr181:pojo>
            <bean class="test.TargetService" />
        </jsr181:pojo>
    </jsr181:endpoint>
</beans>
```

Here we first wire both `EchoProxyService` and `TargetService` as JSR181-compliant services onto the JBI bus. Next we define a JBI proxy for the `TargetService`. If we closely observe the proxy configuration, we can see that we are insisting that the proxy to implement the type `test.IEcho`. That makes sense and is not a surprise since the target service class, `test.TargetService` is also of type `test.IEcho`.

Deployment Configuration

For deployment, we will package the relevant artifacts for the JSR proxy binding into a standard SA. We will also have an HTTP bound SA so that we can use a simple HTTP client to test the setup. As the configurations are exactly the same as what we used in the previous example, they are not repeated here.

Deploying and Running the Sample

To build the entire codebase and deploy the sample, change directory to `ch13\JbiProxy\01_CompatibleInterface` which contains a top-level `build.xml` file. Execute ant as shown here:

```
cd ch13\JbiProxy\01_CompatibleInterface
ant
```

Now, bring up the ServiceMix container by executing the `servicemix.xml` file contained in the same folder.

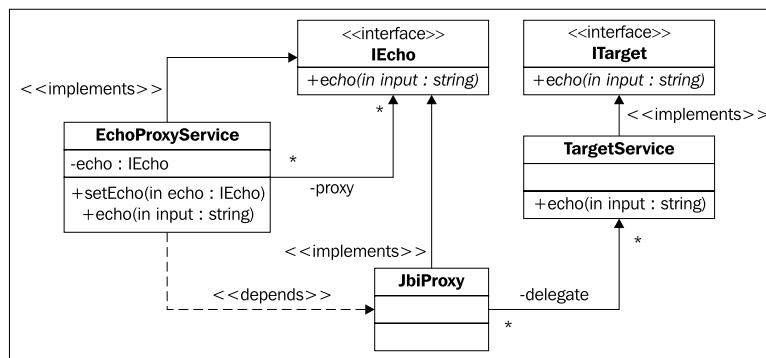
```
cd ch13\JbiProxy\01_CompatibleInterface
%SERVICEMIX_HOME%/bin/servicemix servicemix.xml
```

The `Client.html` provided again in the same folder can be used to send messages to test the deployed service. Now clicking Send on the client will route the request message to the ServiceMix ESB. At the ServiceMix console, you can see that `EchoProxyService.echo` is invoked first, which will then delegate the call to `TargetService.echo`. This is shown in the following screenshot:

```
ServiceMix
y: HttpBindAssembly
INFO - ServiceUnitLifeCycle - Initializing service unit: HttpBind
t: HttpBind
INFO - ServiceUnitLifeCycle - Starting service unit: HttpBind
INFO - jetty - jetty-6.0.1
INFO - jetty - Started SelectChannelConnector @ localhost:8081
INFO - AutoDeploymentService - Directory: deploy: Finished installation of archive: httpbind-sa.zip
INFO - AutoDeploymentService - Directory: deploy: Archive changed: processing jsrproxy-sa.zip ...
INFO - ServiceAssemblyLifeCycle - Starting service assembly: JsrProxyAssembly
INFO - ServiceUnitLifeCycle - Initializing service unit: JsrProxy
INFO - ServiceUnitLifeCycle - Starting service unit: JsrProxy
INFO - AutoDeploymentService - Directory: deploy: Finished installation of archive: jsrproxy-sa.zip
EchoProxyService.echo: this = test.EchoProxyService@352d87
TargetService.echo: String: this = test.TargetService@1923ca5
```

JBI Proxy Sample implementing In-Compatible interface

In the second sample on JBI Proxy, we will make a simple but significant change in the interfaces implemented. We will create a JBI proxy implementing an interface incompatible to the target service. Then, in place of the target service we will use the proxy instance. Then any calls intended to the target service will be first routed to the proxy and the proxy in turn will delegate the call to the target service. The structural relationship between various classes participating in the interaction is shown in the figure:



In the above figure, you might have noticed that even though we use two completely different types (IEcho and ITarget) as the interfaces, the methods declared in these two interfaces are the same in every respect. This is a hack we want to introduce intentionally. In other words our aim here is to invoke the method in TargetService. But, we want to do it through the proxy only. We want the proxy to be created implementing a different interface, IEcho. Hence, IEcho is different from the ITarget. This means if we go by the normal Java type compatibility rules, the proxy which we created here is "technically incompatible" with the target service. But a proxy is a proxy and hence it can proxy calls even to a different type. However, As we want to invoke the same method in the proxy too, we have purposefully kept the method name same in both the proxy interface and the target service interface.

Proxy Code Listing

The codebase for the sample is located in the folder `ch13\JbiProxy\02_IncompatibleInterface\01_JsrProxy\src`.

This folder contains the interface `IEcho` and the class `EchoProxyService` implementing the `IEcho` interface. Now, we have one more interface `ITarget` and another class `TargetService` implementing the `ITarget` interface.

These classes are explained here:

- **IEcho.java:** In this sample also, the `IEcho` interface declares a single method `echo` which takes a `String` parameter and returns a `String` too.

```
public interface IEcho
{
    public String echo(String input);
}
```

- **EchoProxyService.java:** Here also the `EchoProxyService` is a convenient class which will act as mechanism for routing requests to the JBI Proxy. Moreover, `EchoProxyService` implements the above interface `IEcho`.

```
public class EchoProxyService implements IEcho
{
    private IEcho echo;
    public void setEcho(IEcho echo)
    {
        this.echo = echo;
    }
    public String echo(String input)
    {
        System.out.println("EchoProxyService.echo. this = " + this);
        return echo.echo(input);
    }
}
```

- **ITarget.java:** Here, we introduce a new interface `ITarget`, which is incompatible to the interface `IEcho`. But purposefully we have retained the method `echo`.

```
public interface ITarget
{
    String echo(String input);
}
```

- **TargetService.java:** As per our discussion earlier, `TargetService` implements the new interface `ITarget`, not `IEcho`.

```
public class TargetService implements ITarget
{
    public String echo(String input)
    {
        System.out.println("TargetService.echo : String. this = " +
                           this);
        return input;
    }
}
```


XBean-based JBI Proxy Binding

Using XBean, we will now configure the JBI proxy to be deployed onto the standard servicemix-jsr181 JBI component. The xbean.xml is as shown as follows:

```
<beans xmlns:sm="http://servicemix.apache.org/config/1.0"
       xmlns:jsr181="http://servicemix.apache.org/jsr181/1.0"
       xmlns:http="http://servicemix.apache.org/http/1.0"
       xmlns:test="http://test">
  <classpath>
    <location>./</location>
  </classpath>
  <jsr181:endpoint annotations="none"
                  service="test:echoService"
                  serviceInterface="test.IEcho">
    <jsr181:pojo>
      <bean class="test.EchoProxyService">
        <property name="echo">
          <jsr181:proxy service="test:TargetService"
                      context="#context" type="test.IEcho" />
        </property>
      </bean>
    </jsr181:pojo>
  </jsr181:endpoint>
  <jsr181:endpoint annotations="none"
                  service="test:TargetService"
                  serviceInterface="test.ITarget">
    <jsr181:pojo>
      <bean class="test.TargetService" />
    </jsr181:pojo>
  </jsr181:endpoint>
</beans>
```

Observe the proxy configuration again. We can see that this time we are insisting that the proxy implements the type `test.IEcho`, even though the interface type for the target service is `test.ITarget`. This is what a proxy is all about. In clear terms, the JBI here will generate a proxy for `test.TargetService`. So the proxy's exposed interface is, by default, compliant to `test.ITarget`. However, we want the proxy to be compliant to `test.IEcho` also, which is the interface the client initially targeted to.

Deployment Configuration

For deployment, we will package again all the relevant artifacts for the JSR proxy binding into a standard SA. We will also have an HTTP bound SA so that we can use a simple HTTP client to test the setup. As the configurations are exactly the same as what we used in previous example, they are not repeated here.

Deploying and Running the Sample

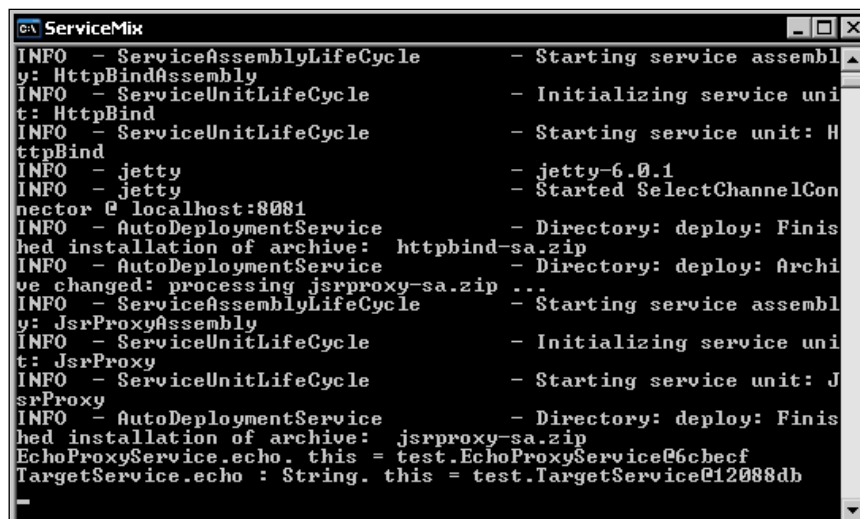
To build the entire codebase and deploy the sample, change directory to `ch13\JbiProxy\02_IncompatibleInterface` which contains a top-level `build.xml` file. Execute ant as shown here:

```
cd ch13\JbiProxy\02_IncompatibleInterface
ant
```

Now, bring up the ServiceMix container by executing the `servicemix.xml` file contained in the same folder.

```
cd ch13\JbiProxy\02_IncompatibleInterface
%SERVICEMIX_HOME%/bin/servicemix servicemix.xml
```

The `Client.html` file provided in the same folder can be used to send messages to test the deployed service. Now clicking Send on the client will route the request message to the ServiceMix ESB. At the ServiceMix console you can see that `EchoProxyService.echo` is invoked first which will then delegate the call to `TargetService.echo`. This is shown in the following screenshot:



```
ServiceMix
INFO - ServiceAssemblyLifeCycle - Starting service assembly
y: HttpBindAssembly
INFO - ServiceUnitLifeCycle - Initializing service unit
t: HttpBind
INFO - ServiceUnitLifeCycle - Starting service unit: H
ttpBind
INFO - jetty - jetty-6.0.1
INFO - jetty - Started SelectChannelCon
nector @ localhost:8081
INFO - AutoDeploymentService - Directory: deploy: Finis
hed installation of archive: httpbind-sa.zip
INFO - AutoDeploymentService - Directory: deploy: Archi
ve changed: processing jsrproxy-sa.zip ...
INFO - ServiceAssemblyLifeCycle - Starting service assembl
y: JsrProxyAssembly
INFO - ServiceUnitLifeCycle - Initializing service unit
t: JsrProxy
INFO - ServiceUnitLifeCycle - Starting service unit: J
srProxy
INFO - AutoDeploymentService - Directory: deploy: Finis
hed installation of archive: jsrproxy-sa.zip
EchoProxyService.echo. this = test.EchoProxyService@6cbeef
TargetService.echo : String. this = test.TargetService@12088db
```

Invoke External Web Service from the ServiceMix Sample

You have now seen how to set up a JBI proxy and how to invoke a proxy just like a POJO bound to JBI. Now you can extend the same principles if you need to call out from a JSR181 SU to a HTTP provider in order to interact with an external web service. You can use XFire to create stub classes based on your WSDL exposed by your external web service. Now you can inject the stub into your JSR181 SU. The stub will be used by the proxy to generate the exchange with the HTTP provider (which should be referenced as the "service").

You have already seen in Chapter 10, how to bind a web service external to the JBI onto the JBI bus. Then any JBI component can exchange messages with the remote web service. One aspect which you need to note is that we have been exchanging messages in a document-oriented fashion. However using JBI proxy now, it is possible to invoke web services in the RPC style from within the JBI bus. For this we leverage the stub classes generated out from the web service WSDL using Axis.

Web Service Code Listing

We are interested in proxy setup to access a remote web service, hence we will not discuss the details of the web service deployment in this section. Instead, we will just browse through the important web service interfaces and the associated WSDL and then move on to binding the proxy.

The web service implements the `IHellWeb` remote interface which in turn extends the `IHell` business interface. They are listed here as follows:

- **IHell.java:** `IHell` is a simple BI, having a single business method `hello`.

```
public interface IHell
{
    String hello(String param);
}
```

- **IHellWeb.java:** In order to deploy a web service, we need an interface complying with the Java RMI semantics, and `IHellWeb` will serve this purpose.

```
public interface IHellWeb extends IHell, java.rmi.Remote {}
```

- **HelloWebService.wsdl:**

The main sections in the web service WSDL is shown as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://AxisEndToEnd.
axis.apache.binildas.com" ...>

  <wsdl:types ... />
  <wsdl:message ... />
  <wsdl:portType name="IHelloWeb">
  </wsdl:portType>
    <wsdl:binding name="HelloWebServiceSoapBinding"
      type="impl:IHelloWeb">
    </wsdl:binding>
  <wsdl:service name="IHelloWebService">
    <wsdl:port binding="impl:HelloWebServiceSoapBinding"
      name="HelloWebService">
      <wsdlsoap:address
        location="http://localhost:8080/AxisEndToEnd/services/
        HelloWebService"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

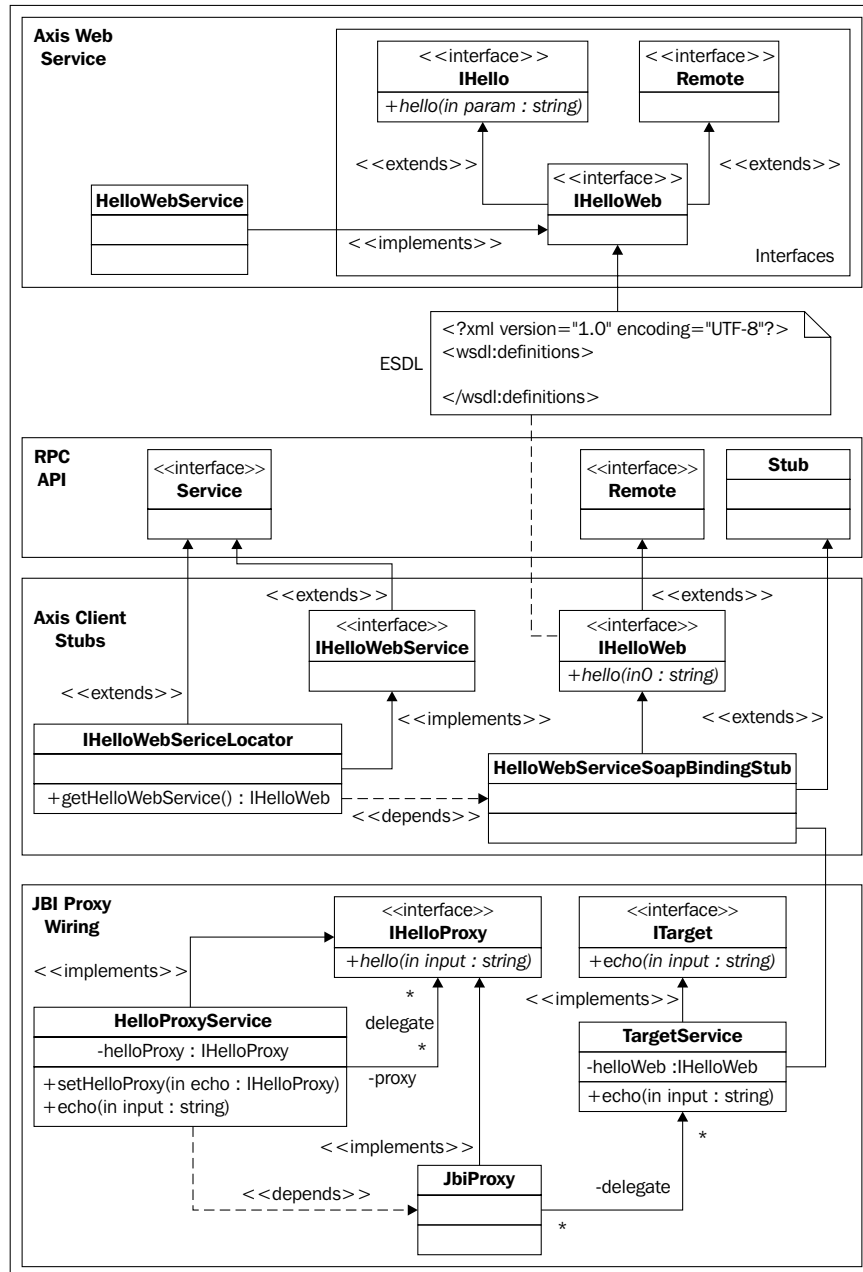
This is enough about the web service and we will move on to the next step.

Axis Generated Client Stubs

We use `org.apache.axis.wsdl.WSDL2Java` class in the `wsdl2java` task to generate client-side binding classes and stubs. The main classes are available in the folder `ch13\JbiProxy\03_AccessExternalWebService\01_ws\gensrc` and they are as follows:

- `HelloWebService.java`
- `HelloWebServiceSoapBindingStub.java`
- `IHelloWeb.java`
- `IHelloWebService.java`
- `IHelloWebServiceLocator.java`

All the above artifacts are Axis generated client-side stubs, hence we will not look into the details of them here. Instead, let us look into the structural relationship between the various developer created and Axis generated artifacts shown in the following figure:



Referring to the above diagram, let us understand the relevant artifacts. Similar to the samples previously listed in this chapter, here our aim is to generate a JBI proxy for an externally bound web service. We are doing this using the following classes:

- **ITarget.java:** This interface is synonymous to the BI `IHello`, having a single business method `hello`. We want to auto-route request-response through the JBI proxy. In order to facilitate this we have retained the method signature in the interfaces the same.

```
public interface ITarget
{
    String hello(String input);
}
```

- **TargetService.java:** In `TargetService`, we auto-wire the web service stub. So, the `helloWeb` instance field in `TargetService` will hold a reference to the stub to the web service. When the `hello` method is invoked in `TargetService`, the call is delegated to the stub which will invoke the remote web service.

```
public class TargetService implements ITarget
{
    private com.binildas.apache.axis.AxisEndToEnd.
                                   IHelloWeb helloWeb;

    public TargetService() {}
    public TargetService(com.binildas.apache.axis.
                        AxisEndToEnd.IHelloWeb helloWeb)
    {
        this.helloWeb = helloWeb;
    }
    public String hello(String input)
    {
        System.out.println("TargetService.echo : String. this = " +
                           this);

        try
        {
            return helloWeb.hello(input);
        }
        catch(Exception exception)
        {
            exception.printStackTrace();
            return exception.getMessage();
        }
    }
}
```

- **IHelloProxy.java:** We now need to wire the JBI proxy to the web services stub. `IHelloProxy` is an interface defined for this purpose and hence is having the same single business method, `hello`.

```
public interface IHelloProxy
{
    public String hello(String input);
}
```

- **IHelloProxyService.java:** `HelloProxyService` is a wrapper or adapter for the JBI proxy. In other words, the `helloProxy` instance field in `HelloProxyService` will refer to the JBI proxy.

```
public class HelloProxyService implements IHelloProxy
{
    private IHelloProxy helloProxy;
    public void setHelloProxy(IHelloProxy helloProxy)
    {
        this.helloProxy = helloProxy;
    }
    public String hello(String input)
    {
        System.out.println("HelloProxyService.hello. this = " + this);
        return helloProxy.hello(input);
    }
}
```

The bean wiring discussed in this section is done using Spring and is shown in the next section.

XBean-based JBI Proxy Binding

Using XBean, we will now configure the JBI proxy to be deployed onto the standard `servicemix-jsr181` JBI component.

The `xbean.xml` is as shown in the following code:

```
<beans xmlns:sm="http://servicemix.apache.org/config/1.0"
        xmlns:jsr181="http://servicemix.apache.org/jsr181/1.0"
        xmlns:http="http://servicemix.apache.org/http/1.0"
        xmlns:test="http://test">
    <classpath>
        <location>./</location>
    </classpath>
    <jsr181:endpoint annotations="none"
                    service="test:echoService"
```

```

        serviceInterface="test.IHelloProxy">
<jsr181:pojo>
    <bean class="test.HelloProxyService">
        <property name="helloProxy">
            <jsr181:proxy service="test:TargetService"
                context="#context"
                type="test.IHelloProxy" />
        </property>
    </bean>
</jsr181:pojo>
</jsr181:endpoint>
<jsr181:endpoint annotations="none"
    service="test:TargetService"
    serviceInterface="test.ITarget">
    <jsr181:pojo>
        <bean class="test.TargetService" >
            <constructor-arg type="com.binildas.apache.axis.
                AxisEndToEnd.IHelloWeb">
                <ref bean="stub"/>
            </constructor-arg>
        </bean>
    </jsr181:pojo>
</jsr181:endpoint>
<bean id="stub"
    class="com.binildas.apache.axis.AxisEndToEnd.
        HelloWebServiceSoapBindingStub">
    <constructor-arg type="java.net.URL" index="0">
        <ref bean="url"/>
    </constructor-arg>
    <constructor-arg type="javax.xml.rpc.Service" index="1">
        <ref bean="serviceLocator"/>
    </constructor-arg>
</bean>
<bean id="url" class="java.net.URL">
    <constructor-arg>
        <value>http://localhost:8080/AxisEndToEnd/
            services/HelloWebService</value>
    </constructor-arg>
</bean>
<bean id="serviceLocator"
    class="com.binildas.apache.axis.AxisEndToEnd.
        IHelloWebServiceLocator">
</bean>
</beans>

```


We now have enough configurations to invoke the external web service. Is there anything fishy in the configuration above? You can go through that once again and let me wait till the end of our discussion to explain what I am hiding from you at this point.

Deployment Configuration

For deployment, we will package again all the relevant artifacts for the JSR proxy binding into a standard SA. We will also have an HTTP bound SA so that we can use a simple HTTP client to test the setup. As the configurations are exactly the same as what we used in the previous example, they are not repeated here. But we need to mention one thing here. The JBI proxy uses web service stub classes to invoke the external service and hence depends on the Axis libraries. We resolve this dependency by compiling the stub classes and including them also in the SA. We also copy all relevant Axis API jars to the ServiceMix optional library path. The ant target for that is given here:

```
<target name="copy-dependency" depends="init">
  <javac srcdir="../01_ws/gensrc" destdir="${build.dir}">
    <classpath refid="javac.classpath" />
    <include name="**/*ServiceLocator.java"/>
  </javac>
  <copy todir="${servicemix.home}/lib/optional" overwrite="true">
    <fileset dir="${axis.home}/lib" includes="*.jar" />
  </copy>
</target>
```

Deploying and Running the Sample

To build the entire codebase and deploy the sample, change directory to `ch13\JbiProxy\03_AccessExternalWebService` which contains a top-level `build.xml` file. Execute ant as shown here:

```
cd ch13\JbiProxy\03_AccessExternalWebService
ant
```

This will build the web service, generate required web service client stubs, and also package all the necessary service assemblies. First, we need to deploy the web service. For that, transfer the web service war file placed in `ch13\JbiProxy\03_AccessExternalWebService\01_ws\dist` folder into the `webapps` folder of your favorite web container and restart the container. You can double check whether your web service deployment works by executing a client kept in the web service folder itself. For that, execute the ant `run` target as follows:

```
ch13\JbiProxy\03_AccessExternalWebService\01_ws
ant run
```

Now, bring up the ServiceMix container by executing the `servicemix.xml` file contained in the same folder.

```
ch13\JbiProxy\03_AccessExternalWebService
%SERVICEMIX_HOME%\bin\servicemix servicemix.xml
```

The `Client.html` provided again in the same folder can be used to send messages to test the deployed service.

Proxy and WSDL Generation

I have asked you to go through the JBI proxy XBean configuration once more in a previous section. Do you think we have actually made the web service a proxy? If we need to truly proxy the web service, then the XBean configuration should be something like the following code:

```
<jsr181:endpoint annotations="none"
    service="test:TargetService"
    serviceInterface="com.binildas.apache.axis.
        AxisEndToEnd.IHelloWeb">

    <jsr181:pojo>
        <bean class="com.binildas.apache.axis.AxisEndToEnd.
            HelloWebServiceSoapBindingStub" >
            <constructor-arg type="java.net.URL" index="0">
                <ref bean="url"/>
            </constructor-arg>
            <constructor-arg type="javax.xml.rpc.Service" index="1">
                <ref bean="serviceLocator"/>
            </constructor-arg>
        </bean>
    </jsr181:pojo>
</jsr181:endpoint>
```

The above configuration is right and that is what we need to proxy the web service. However the issue here is that the class `HelloWebServiceSoapBindingStub`, which if you look at the source code you can see, is dependent on many apache axis RPC API classes. For proper JBI proxying, the JBI container should be able to generate WSDL out of the exposed API but it may not make sense to generate WSDL out of these RPC-dependent classes. In fact the WSDL generator fails and throws an error. Hence instead what we have done in the previous example is that we have a proxy wrapper `TargetService` to which we inject the `HelloWebServiceSoapBindingStub` instance using the Spring injection mechanism. Then we delegate any calls from the `TargetService` instance to the stub instances, and that does the magic.

Summary

Proxies are strong features in the Java language package, and similar functionality can be availed from within your JBI ESB using JBI proxies. Interception and re-routing are some of the features we can implement using proxies. You have also seen how an external web service can be bound to the JBI bus and then exposed as proxies within the bus itself so that other components within the bus can route messages through these proxies. The building blocks demonstrated in this chapter can be used to solve your integration problems.

We will look into a more interesting concern in the services network – versioning of services, in the next chapter.

14

Web Service Versioning

Versioning service, especially of the web services, is a topic of heated discussions in many of the forums and sites. Even though there are many approaches to this topic, we cannot often find any concrete guideline or code showing the implementation. This is because the topic is not simple. The term "Versioning" means different things to different people, depending upon the context in which they are speaking. For some, versioning means a way to manage compatible change in the service implementation alone, without any major change in the service description. However, for those who define services for a large corporate enterprise, versioning is a mechanism or tool without which he cannot control the ever increasing complexity of an SOA ecosystem.

The effective use of an ESB infrastructure provides a means to solve the versioning problems and in this chapter we are going to look at the working code in action demonstrating how we can version the web services.

We will look into the following topics in particular:

- The what and why of service versioning
- Versioning in an SOA context – what is required
- Different strategies in versioning the web services
- Approaches in versioning the web services
- A service versioning sample – working code in ESB

Service Versioning—A Means to SOA

Versioning is important, whether we are dealing with binary programming paradigms such as the Global Assembly Cache (GAC) of .NET run time or we are dealing with SOA infrastructures such as the ESB. However when it comes to SOA, versioning will have a slightly different meaning which I will try to describe in this section.

Services are Autonomous

SOA is now the buzzword — people use it in every other context; everyone has their own beliefs and understanding. Whatever it is, autonomy becomes the prime concern in an SOA implementation. First, let us ask ourselves why we moved away from our old CORBA or our well-known Java RMI architectures for service implementation and consumption. Leaving aside all the varied definitions of SOA and the many advantages an SOA yields, autonomy is one of the best features SOA brings to the table of both the providers and consumers. Service providers can keep on changing their service implementations, either to add a new functionality or to extend or enhance the existing functionality. In doing so, service consumers should be unaffected; that is, they shouldn't even be aware that something has changed. Needless to say that the services description shouldn't change, not even the service URL should change.

Change is the Only Constant Thing

If everything remains static, it is an ideal world for an engineer, even though an artist would then curse the world. However, time has shown that the only constant thing in the world is change. Networks change, platforms and frameworks change, operating systems change in versions and supportability, and service implementations also change.

A change may be to enhance the QOS, perhaps to improve the response time by introducing a new algorithm in the code. Change can also add a new functionality to the existing service endpoints. Such changes are usually manageable by not revealing the change to the external world, especially to the service consumer. Sometimes, we may also need to introduce or cut short a parameter to the already existing service method. In such a case the existing consumers may have to do some changes at their end also, to make the stubs compatible to the server-side changes. In fact many times, the service consumers have to rebuild and recompile their client-side stubs and adjust their calling code to make them comply with the remote interface. For small deployments these can be done by developers with the best design and code practices and with the help of IDEs and tools. However, this will turn out to be a nightmare when the number of services keeps on increasing, which is typical of every growing enterprise.

All Purpose Interfaces

There is also the notion of a generic or whole purpose service interface. A sample is shown in the following code:

```
public String serviceMethod(String serviceXml);
```

The above interface talks about a generic service (even the method name is generic, "serviceMethod"), which takes a generic parameter and returns again a generic value. Passing the XML in a document style to a generic method interface like the one shown above is an example. Here, whenever a change is required we needn't reflect that in the interface-level, but every change is hidden in the string formatted request and response messages. At first sight this might seem to be a solution to the change problem, but experience has shown that this is in fact an anti-pattern in the SOA world. We lose all static type binding which means even the tools will not find out any mismatch or type validation errors. This will be revealed only after the service invocation as a reactive error scenario.

SOA Versioning—Don't Touch the Anti-Pattern

Let us consider a typical web service method such as the following:

```
public String transferFund(String fromAccount, String toAccount,  
                           double amount);
```

Here, the method will transfer the fund from one account to another. Let us assume that one fine day we want to change the service to accept one more extra parameter, like the one shown in the following code:

```
Public String transferFund(String fromAccount, String toAccount,  
                           double amount, String transactionPassword);
```

Now, here are a set of questions for the reader. We may or may not answer all these questions but at least agree that we identify the possible caveats to our traditional thinking! The following are the questions for the readers to think over.

1. Do we need to version services or operations?

The first question in SOA is whether `transferFund` is a service or an operation. We need to appreciate that a web service is described by a WSDL and a WSDL can contain multiple operations defined in the `portType`. If so, when `transferFund` needs to include one more parameter in the request, has the version of the operation changed or the version of the service changed?

2. Can we overload a service?

If we consider object-oriented programming, the same class can include both "versions" of the `transferFund` method—we call it method overloading. Fortunately or unfortunately, WSDL doesn't allow method overloading. Now we are left with few other strategies (or hacks?) to combat the change, which is described in the next point.

3. New version or new service?

To handle the new extra parameter, I can create the `transferFund02` method which will take an extra parameter or rename the service name to something different like `transferFundSecurely`. Whatever our strategy, the question here is have we created a new version of the service or an entirely new service?

4. Will our information model save us from schema changes?

Let us consider a new service:

```
public Address getAddress(CustomerInfo customerInfo);
```

The above service will return back an `Address` if you provide the `CustomerInfo` of the customer whose `Address` has to be retrieved. Now, let us look into `Address`.

```
public class Address{
    private String houseNumber;
    private String street;
    private String city;
}
```

Now, we change the `Address` to include an additional field to also take care of the zip code as shown here:

```
public class Address{
    private String houseNumber;
    private String street;
    private String city;
    private String zip;
}
```

Do you think this change has introduced any change to the `getAddress` service? Apparently not, since the service definition remains the same, but in fact I have altered my data type which in turn will make the existing consumer codebase incompatible with the new service.

So far so good, now the final question—for scenarios that we have seen above, should we create a new version of the existing service or should we define a new service itself by not disturbing the existing one?

Rather let us restate the above points in this manner—in SOA, services shouldn't mutate from version to version. This will safeguard any existing consumers. However, new services can be introduced which is not too complicated because new services will have their own separate endpoints. Moreover, irrespective of whether the consumers are new or existing, they will use the new endpoint to avail the new service. At the same time services can also be enhanced or upgraded. An enhancement can involve increasing the QOS features of the service or bringing in

new implementation technologies behind the scenes, keeping the service interface untouched. When you upgrade a service, you can now decide what to do with the "old version" of the same service. I purposely used the words "old version" this time because versioning of services makes sense in the context of enhancing or upgrading services only, not in the context of introducing new services.

Types can Inherit—Why not My Schemas

Again in OOP, you can define a base type (like *Animal*) and derive multiple types (like *Cat* and *Dog*) from that. The derived classes are always type compatible to the base type (You can always substitute an *Animal* with a *Cat* or a *Dog*).

W3C XML schema's `<extension>` provides us a means to extend XML schemas. How can this be of any use to our web service? Can we substitute an *Address* XML instance which contains a *zip code* element in place of where a service expects an *Address* without the *zip code*? Whether the service will still work or not depends on what extra hooks you attach to your web services infrastructure to bridge the schema mismatch. Hence, we acknowledge the fact that schemas are extensible and are also able to version. But that might not straightaway map to the concept of extending versioning principles to services.

If Not Versions, Then What

Having discussed enough about versioning around services, let us now look into what is needed in an SOA. We all agree that multiple flavor (variants) of service can exist and if all of them coexist, we need a good governance mechanism also to satisfy consumer requests as per the agreed upon SLA. This is where an ESB-based routing mechanism will add value. The word "version" is very common to the software developers and so, we prefer to retain that word in the discussions also – to point to multiple variants of the service. Let us now look into versioning of services in this context.

Strategy to Version Web Service

Given the problem of change, the next thing is to find out the exact mechanism to follow to version control the services. In fact, we have already seen one way, which is the generic string formatted messages approach, but as discussed, this may not be the best available method. We will now look into multiple options available for service versioning.

Which Level to Version

One important question to answer in service versioning is the level at which we have to version control. By level we mean whether at the whole service interface-level or at the individual service method-level. Perhaps this question is out of context because when we speak on SOA and its constituent services, we always means services which are coarse grained, which are never individual fine grained method invocations. In other words, we always speak about "transfer fund", "authorize credit", or "validate service" request in SOA, but not fine grained methods such as "update balance cell in account table". This follows that services can be versioned as a whole. For example, in a fund transfer service we can either version control for the service fund transfer or for the individual, composed services such as withdrawal and deposit. To version control as a whole, we need to version the fund transfer service, not the composed fine grained methods.

Version Control in a Schema

An XML schema is used in a web service description to define the message parameters exchanged between the consumers and providers. The XML schema also defines a namespace which differentiates one schema from another. Moreover, a schema also provides a mechanism for extensibility. Thus, a XML schema is a double edged sword. On one side, it gives the flexibility of extension and on the other side it gives a mechanism to constraint or separate out between extensions. Let us look into a sample schema and explain this.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.product.org">
  <xs:complexType name="employee">
    <xs:sequence>
      <xs:element name="firstName" type="xs:string"/>
      <xs:element name="lastName" type="xs:string"/>
      <xs:any namespace="##any"
                processContents="lax"
                minOccurs="0"
                maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

The above is an extensible XML schema. The `firstName` and `lastName` elements are bound whereas it provides an extension mechanism to add additional constructs after the `lastName` (for example, Binil Das Mr, Craig Maret PhD, and so on) while still remaining valid based on the overall schema definition. At the same time, the `targetNamespace` mechanism also helps us to separate out different schemas, and thus to differentiate between schema constrained XML data.

targetNamespace for WSDL

A WSDL document is the description of a web service and has a definitions element that contains the types, message, portType, binding, and service elements.

For the definitions element, targetNamespace is the namespace for information about the referred service. One WSDL document can import other WSDL documents, and setting targetNamespace to a unique value ensures that the namespaces do not clash. The default namespace of the WSDL document is xmlns, and it is set to `http://schemas.xmlsoap.org/wsdl/`. All the WSDL elements, such as definitions, types and messages reside in this namespace. xmlns:xsd and xmlns:soap are the standard namespace definitions used for specifying SOAP-specific information as well as data types. xmlns:tns stands for this namespace.

The WSDLs are generated out of the service interfaces. These interfaces may be in Java or .NET. When tools generate the WSDL, they will have their default strategy on what value to put for the targetNamespace attribute. Many a times, this can be overridden by the tools. Apache Axis WSDL2Java and Java2WSDL do this. Hence, targetNamespace is an attribute which we can control if required. The targetNamespace can be given values in multiple formats, depending upon which way we want to control. Examples are given here:

```
targetNamespace=http://www.binildas.com/types/products/v1/1
targetNamespace=http://www.binildas.com/2006/12/30/products
```

This also provides an excellent mechanism to version control services and we will demonstrate this in examples later in this chapter.

Version Parameter

Including a version parameter is another method. This special parameter is usually passed through the headers of the web service request. It is also possible to include this parameter in the message payload (or body content). In either form, in order to make this mechanism work we need to pass the value for the version parameter along with every request. A sample SOAP request with a version parameter in the header is shown in the following code:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  <soap:Header>
    <Version xmlns="http://product.services/binildas.com">
      2.2
    </Version>
  </soap:Header>
  <soap:Body>
    ....
  </soap:Body>
</soap:Envelope>
```

As discussed before, the same can be included in the body as shown in the following code:

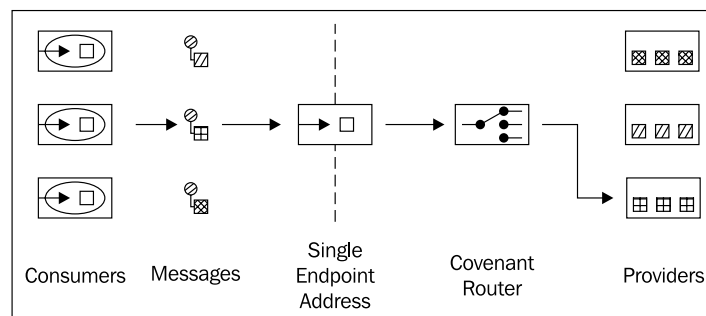
```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  <soap:Header>
</soap:Header>
  <soap:Body>
    <svc:hello xmlns:svc="http://product.services/binildas.com">
      <Version xmlns="http://product.services/binildas.com">
        2.2
      </Version>
      ....
    </svc:hello>
  </soap:Body>
</soap:Envelope>
```

Web Service Versioning Approaches

We have seen a few strategies of web service versioning. Let us now look at some approaches of implementing those versioning strategies.

Covenant

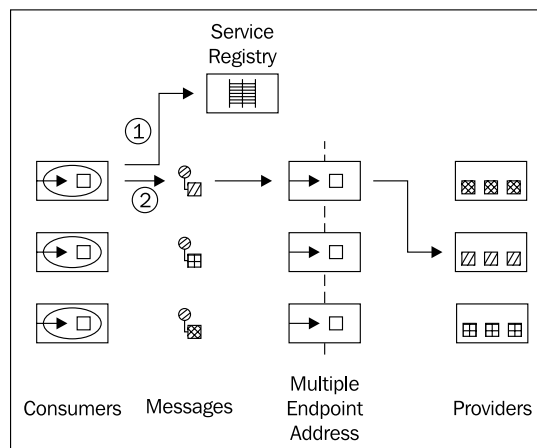
A covenant is an if-then-else way of versioning approach. This can be done using multiple strategies which we saw earlier. In any strategy, the covenant approach looks for some ifs, and depending upon the outcome of the condition a suitable then clause will be executed. It can also have an else or a default clause. Hence this approach is usually combined with a versioning flag and this flag can be either in the form of a version parameter or in the form of a version sensitive value in the targetNamespace. This is shown in the following figure:



A covenant is usually exposed in a **Single Endpoint Address**. Hence, all Consumers will send messages to the same address. Depending upon which version of the service contract the Consumer is using a suitable version flag will be passed either as a version parameter or through the `targetNamespace`. The covenant usually implements a content-based router. A content-based router is used to route each message to the correct recipient based on the message content. The routing can be based on a number of criteria such as the existence of a version parameter and the version info in `targetNamespace`. The advantage of the covenant approach is that Consumers are unaware that multiple service versions coexist at the Provider's end. Hence they needn't adjust their address to route to specific service versions. Instead, they will place their version flag in the message and send it to the same address. It is up to the covenant to redirect the message to the appropriate version of the service.

Multiple Endpoint Addresses

In the **Multiple Endpoint Address** approach, each version of the web service is allotted a separate endpoint address, and they are then bound to a look up mechanism like a registry. Now, the Consumer has to decide the endpoint address for service invocation by looking at the Service Registry, and cross matching the version of interest. Then the Consumer sends messages to the selected endpoint address of interest. This channel will route the Message straight to the exact version of the service. The schema is shown in the following figure:

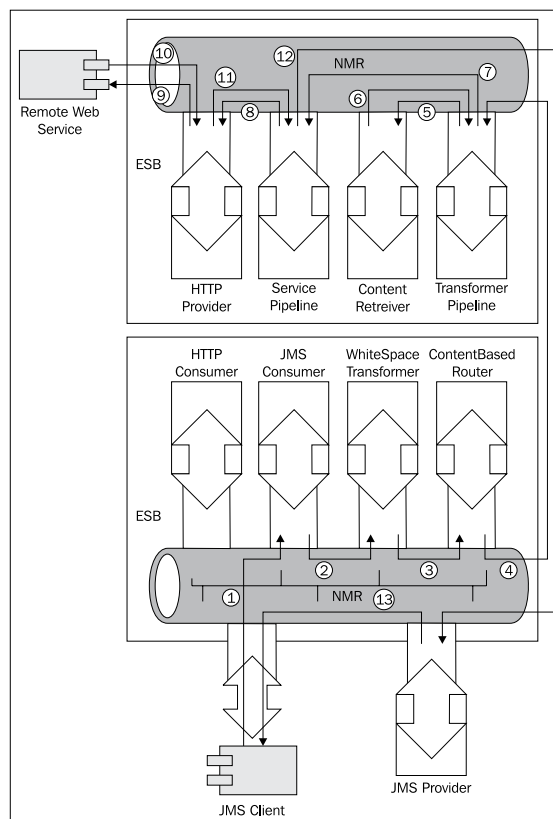


Web Service Versioning Sample using ESB

We discussed the theory of web service versioning, now it is time to put that in code. Let us do that with the help of a sample use case. One thing to be noted here is that to implement the sample we make use of EIPs building blocks which are described in detail in a chapter of their own.

Sample Use Case

The sample use case is about setting up the JBI components to effectively enable the versioning mechanism in the services. All these components are configured in the ESB. An external client interacts with the ESB thus testing the versioning mechanism. The ESB is bound to the different versions of services, which are defined external and remote to the ESB. The ESB will apply the versioning rules and route the requests to the respective version of the service. The sample use case is illustrated here in the following figure:



In fact this sample is not as complicated as the figure make it look. We will see the individual components first and understand the flow.

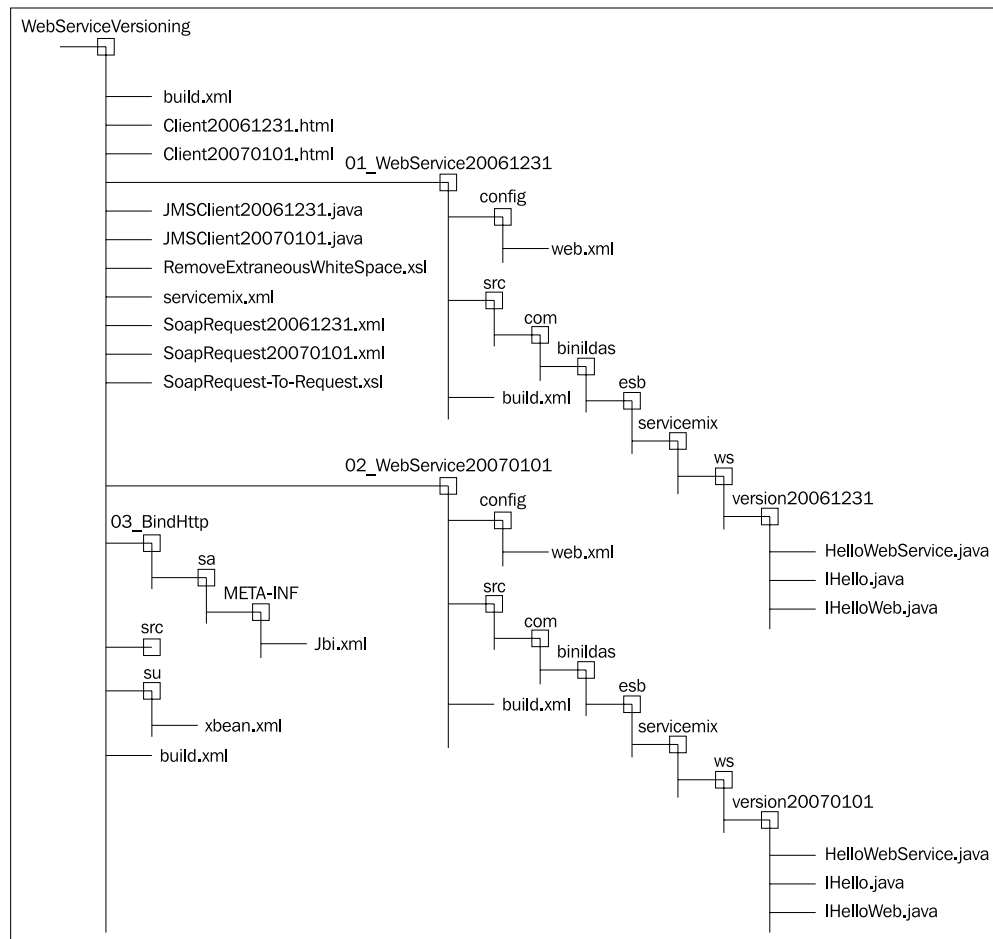
- **JMS client:** In this sample, we use the JMS channel to send web service requests. The JMS Client reads the SOAP request message from a file and sends the message to the JMS Consumer queue configured in the ESB. One point to be noted here is that we are going to use the `targetNamespace` mechanism of the WSDL to implement the service versioning. If we open the SOAP request, we can see how we have designed the SOAP request so as to include a version specific value for the namespace attribute. The value corresponds to the `targetNamespace` attribute of WSDL for the SOAP request.
- **JMS consumer:** The JMS Consumer is a `servicemix-jms` component which allows you to send JMS messages to the configured queue. A consumer role for the `servicemix-jms` component implies the component is a consumer to the NMR. Any messages coming to the queue will be transferred to the `targetService` attribute for the JMS Consumer.
- **WhiteSpace transformer:** A valid SOAP request can come in several forms, all in a single long line or separated out into multiple lines, formatted with indentations and spaces. To make the content ready for an XPath query match later in the flow, we use the extraneous WhiteSpace Transformer component which will trim out all extraneous white spaces from the SOAP request payload and normalize it into a single line.
- **Content-based router:** The ContentBased Router is used for all kinds of content-based routing. We earlier discussed that we have designed the SOAP request so as to include a version specific value in the `targetNamespace` attribute for the SOAP request. Hence now we will use a ContentBased Router which is a `servicemix-eip` component. This component will inspect the value for the namespace attribute corresponding to the `targetNamespace` attribute of the WSDL embedded in the SOAP request and see the matching rule configured at the component-level. Depending upon the match, the ContentBased Router will route the SOAP request message to any one of the set of Transformer Pipelines configured.
- **Transformer pipeline:** The pipeline is a standard EIP component and hence is available readily as a standard JBI EIP component in ServiceMix. The pipeline component is an integration bridge between an In-Only (or Robust-In-Only) MEP and an In-Out MEP. By receiving an In-Only MEP by the pipeline, it will send the input message in an In-Out MEP to the transformer destination and then in turn forward the response in an In-Only MEP to the target destination. As per that, our aim here is to send the message to an In-Out MEP component (a content retriever in our case) and then to route the out message from that component to the next chain in the flow (which again in our case is a second pipeline).

- **Content retriever:** The functionality of this component is to extract the payload part from the incoming SOAP request. Hence, we trim out the SOAP envelope and body tags, and extract only the contents within the body element to be sent to the next component. We said that we need to invoke a web service and you might wonder why we want to extract the payload alone rather than sending the whole SOAP request to the target web service. The reason why we are doing this will be evident when we review the HTTP Provider. Content Retriever will send the out part of the message back to the transformer pipeline. The Transformer Pipeline will route this message to the next component in the chain, which is the Service Pipeline.
- **Service pipeline:** Our aim here is to send the input message in an In-Out MEP to the next component which is the HTTP Provider. Then in turn forward the response in an In-Only MEP to the target destination, which is the output JMS queue.
- **HTTP provider:** A HTTP Provider role implies that the NMR is the consumer to the component. Hence, the HTTP Provider is linked with the Remote Web Service so that any request coming to the HTTP Provider can be routed to the web service. When the HTTP Provider sends a request to the web service, it needs to specify a SOAP action in the request header. To facilitate this we have two attributes namely, `soap` and `soapAction` at the HTTP Provider-level. A true value for `soap` attributes is supposed to wrap the message body in a SOAP envelope whereas the value for `soapAction` will be embedded as the SOAP action in the request header. For some reason, if we specify the `soapAction` attribute alone, `servicemix-http` does not forward the SOAP action header. Hence we are specifying `soap` and `soapAction` together. However, `soap` will wrap the request in the SOAP envelope. We don't want the body content wrapped in two levels of SOAP envelope. That is why in the Content Retriever component (we already have seen this short while ago) we extract only the contents within the body element to be send to the next component.
- **Remote web service:** This is a normal web service which can be deployed in any web container infrastructure.
- **JMS provider:** The SOAP response is placed back in the queue configured in the JMS Provider component from where the client program picks up the message.
- **HTTP consumer:** The HTTP Consumer components configured here are just to help you to send arbitrary messages to the sample set up for any ad-hoc testing.

We have now seen the major components configured for the sample application and how the message flows through them. We also have a few more settings so that we can actually demonstrate that we can control multiple versions of web services to be exposed to clients in the covenant-based settings. Let us look that now.

In fact, we have two SOAP requests, each having two different values for the `targetNamespace` attribute. We also have two JMS client programs so that the readers can easily test these two SOAP requests on the ESB. To make sure the requests are routed separately to different web services based on the value of `targetNamespace` attribute, we also have two web services hosted and bound to the ESB infrastructure.

Now we will move on to the details of the configuration and other settings for the demonstration. For that, also have a look at the files and folder structure organization we have so that we can host and manage two versions of web service in the demo setup.



The figure shows only the files required for the demo setup. The top-level build is capable of calling the build files in the child project folders and thus can build the entire demo in a single go. This will then create many intermediate and final folders and artifacts which are not shown here.

Configure Components in ESB

In the sample use case section you have seen the various components used for the demo, and you have also seen the flow of message. Let us follow the same order for components and see how they are configured in ServiceMix so that you can easily correlate to the flow we explained earlier.

- **JMS client:** There are two versions of the JMS clients. They are:

```
ch14\WebServiceVersioning\JMSClient20061231.java
ch14\WebServiceVersioning\JMSClient20070101.java
```

Both these client programs are similar except the fact that `JMSClient20061231.java` refers to `SoapRequest20061231.xml` whereas `JMSClient20070101.java` refers to `SoapRequest20070101.xml`. The code for `JMSClient20061231` is shown as follows:

```
public class JMSClient20061231
{
    private static final String REQUEST_FILE =
        "/SoapRequest20061231.xml";
    public static void main(String[] args) throws Exception
    {
        ActiveMQConnectionFactory factory =
            new ActiveMQConnectionFactory("tcp://localhost:61616");
        ActiveMQQueue pubTopic = new ActiveMQQueue("queue/A");
        ActiveMQQueue subTopic = new ActiveMQQueue("queue/B");
        Connection connection = factory.createConnection();
        Session session = connection.createSession(false,
            Session.AUTO_ACKNOWLEDGE);
        MessageProducer producer = session.createProducer(pubTopic);
        MessageConsumer consumer = session.createConsumer(subTopic);
        connection.start();
        InputStream inputStream = JMSClient20061231.class.
            getClass().getResourceAsStream(REQUEST_FILE);
        int available = inputStream.available();
        byte[] bytes = new byte[available];
        inputStream.read(bytes);
        inputStream.close();
    }
}
```

```

        String requestString = new String(bytes);
        producer.send(session.createTextMessage(requestString));
        TextMessage textMessage = (TextMessage)
            consumer.receive(1000 * 10);
        if(textMessage == null)
        {
            System.out.println("Response timed out.");
        }
        else
        {
            System.out.println("Response was: " +
                textMessage.getText());
        }
        connection.close();
    }
}

```

The program opens the file `SoapRequest20061231.xml`, reads the content, and then sends the content to the JMS queue configured as the web service gateway channel at the ESB end. Let us also look at the SOAP request format to understand how we place a value for the namespace of the XML element corresponding to the `targetNamespace` attribute of WSDL. For this, `SoapRequest20061231.xml` contents are shown as follows:

```

<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/
    soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-
        instance">

    <soapenv:Body>
        <ns1:hello soapenv:encodingStyle=
            "http://schemas.xmlsoap.org/soap/encoding/"
            xmlns:ns1="http://version20061231.ws.
                servicemix.esb.binildas.com">

            <in0 xsi:type="soapenc:string"
                xmlns:soapenc="http://schemas.xmlsoap.org/
                    soap/encoding/">

                Binil
            </in0>
        </ns1:hello>
    </soapenv:Body>
</soapenv:Envelope>

```

Let us pay attention to the `ns1` namespace. We have given a value of `"http://version20061231.ws.servicemix.esb.binildas.com"` for this namespace and we are going to use this tweak to version control the services. You may want to have a look at the WSDL for the remote web service also to correlate how the `ns1` namespace corresponds to the `targetNamespace` attribute of WSDL.

- **JMS consumer:** JMS consumer is a `servicemix-jms` component configured in the consumer role. We configure queue "A" as the input queue for all the test messages. We have also configured `test:extraneousWhiteSpaceTransformer` as the `targetService` for this JMS consumer so that the JMS consumer will route any messages coming to the queue "A" to the `extraneousWhiteSpaceTransformer` component. This is shown in the following code:

```
<sm:activationSpec>
  <sm:component>
    <jms:component>
      <jms:endpoints>
        <jms:endpoint service="test:MyConsumerService"
                      endpoint="myConsumer"
                      role="consumer"
                      soap="false"
                      targetService="test:
                        extraneousWhiteSpaceTransformer"
                      defaultMep="http://www.w3.org/2004/08/
                        wsdl/in-only"
                      destinationStyle="queue"
                      jmsProviderDestinationName="queue/A"
                      connectionFactory="#connectionFactory"
                      />
      </jms:endpoints>
    </jms:component>
  </sm:component>
</sm:activationSpec>
```

- **Whitespace transformer:** The whitespace transformer is a `ServiceMixXsltComponent`. It uses a stylesheet to remove all extraneous white spaces from the message. The trimmed message is then routed to the next component, which is the `test:router`.

```
<sm:activationSpec componentName="extraneousWhiteSpaceTransformer"
                  service="test:extraneousWhiteSpaceTransformer"
                  destinationService="test:router">
  <sm:component>
    <bean class="org.apache.servicemix.components.
              xslt.XsltComponent">
      <property name="xsltResource"
```

```

value="RemoveExtraneousWhiteSpace.xml"/>
    </bean>
</sm:component>
</sm:activationSpec>

```

The `XsltComponent` uses a stylesheet to strip out all extraneous white spaces from the message. The stylesheet is shown as follows:

```

<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/
    Transform" version="1.0">
<xsl:output method="xml" version="1.0" indent="no"/>
<xsl:strip-space elements="*" />
<xsl:preserve-space elements="xsl:text" />
<xsl:template match="/*|*|processing-instruction()">
    <xsl:copy>
        <xsl:apply-templates select="@*|node()
            [not(self::comment())]" />
    </xsl:copy>
</xsl:template>
</xsl:stylesheet>

```

- **Content-based router:** This is the core of the demonstration setup and this is where the decision as to which version of the service is to be invoked takes place. Let us look at this component in detail.

```

<eip:content-based-router service="test:router"
    endpoint="endpoint">
    <eip:rules>
        <eip:routing-rule>
            <eip:predicate>
                <eip:xpath-predicate xpath="number(substring-before(
                    substring-after(namespace-uri-for-prefix(
                        substring-before(name(/*/child::node()/
                            child::node()), ':'), */child::node()/
                                child::node()), 'http://version'),
                                    '.ws.servicemix.esb.binildas.com'))
                    < 20061231" />
            </eip:predicate>
            <eip:target>
                <eip:exchange-target service="test:
                    pipelineTransformBefore20061231" />
            </eip:target>
        </eip:routing-rule>
        <eip:routing-rule>
            <eip:predicate>

```

```
<eip:xpath-predicate xpath="number(substring-before(
    substring-after(namespace-uri-for-prefix(
        substring-before(name(/*/child::node()/
            child::node()), ':'), /*/child::node()/
            child::node()), 'http://version'),
        '.ws.servicemix.esb.binildas.com'))
    = 20061231" />

</eip:predicate>
<eip:target>
    <eip:exchange-target service="test:
        pipelineTransform20061231" />
</eip:target>
</eip:routing-rule>
<eip:routing-rule>
    <eip:predicate>
        <eip:xpath-predicate xpath="number(substring-before(
            substring-after(namespace-uri-for-prefix(
                substring-before(name(/*/child::node()/
                    child::node()), ':'), /*/child::node()/
                    child::node()), 'http://version'),
                        '.ws.servicemix.esb.binildas.com'))
                            &gt; 20061231" />

    </eip:predicate>
    <eip:target>
        <eip:exchange-target
            service="test:pipelineTransformAfter20061231" />
    </eip:target>
</eip:routing-rule>
<eip:routing-rule>
    <eip:target>
        <eip:exchange-target
            service="test:pipelineTransformAfter20061231" />
    </eip:target>
</eip:routing-rule>
</eip:rules>
</eip:content-based-router>
```

The above content-based router has three conditional targets and a default target. To evaluate the content, the router uses the XPath predicate where we can plug in our rules in the XPath format.

The rule which we plug in here is repeated again:

```
number(substring-before(substring-after(namespace-uri-for-prefix(
    substring-before(name(/*/child::node()/child::node()), ':'),
        /*/child::node()/child::node()), 'http://version'),
        '.ws.servicemix.esb.binildas.com'))
```

This rule has to be understood in relation to the SOAP request which we have already seen. Let us repeat the relevant portion again here:

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope ...>
  <soapenv:Body>
    <ns1:hello xmlns:ns1="http://version20061231.ws.
      servicemix.esb.binildas.com" ...>
      <in0 xsi:type="soapenc:string"
        xmlns:soapenc="http://schemas.xmlsoap.org/
          soap/encoding/">Binil</in0>
    </ns1:hello>
  </soapenv:Body>
</soapenv:Envelope>
```

The XPath predicate shown above is intended to cut the version part alone from the namespace attribute in the SOAP request. We have appended year, month and date in integer form together to form an easy representation of version. If required, we can also append the timestamp portion to the version attribute to make it more agile! In any case, we take the version value and compare it in the router. In the above router, we have following pseudocoded rule:

```
if(${version} < 20061231)
  target : test:pipelineTransformBefore20061231
else
  if(${version} = 20061231)
    target : test:pipelineTransform20061231
  else
    if(${version} > 20061231)
      target : test:pipelineTransformAfter20061231
    else
      target : test:pipelineTransformAfter20061231
```

Hence, depending upon which version the SOAP request is targeted to, the content-based router will route the request to the appropriate pipeline transform.

We have provided two SOAP request files for testing. They are:

```
SoapRequest20061231.xml
SoapRequest20070101.xml
```

As is evident from their names, `SoapRequest20061231.xml` has 20061231 embedded as the version in the body whereas `SoapRequest20070101.xml` has 20070101 as its version. The readers are free to change this version values to their own figures to test whether or not the router behaves appropriately. If required, the readers can also add more clauses to the rules in the content-based router and understand the behavior.

- **Transformer pipeline:** The pipeline is a standard EIP component and is an integration bridge between an In-Only (or Robust-In-Only) MEP and an In-Out MEP. By receiving an In-Only MEP by the pipeline, it will send the input message in an In-Out MEP to the transformer destination and then in turn forward the response in an In-Only MEP to the target destination. Let us look at our pipeline configuration shown in the following code:

```
<eip:pipeline service="test:pipelineTransformBefore20061231"
              endpoint="pipelineTransformBefore20061231">
  <eip:transformer>
    <eip:exchange-target service="test:soapContentRetreiver" />
  </eip:transformer>
  <eip:target>
    <eip:exchange-target
      service="test:pipelineServiceBefore20061231" />
    </eip:target>
  </eip:pipeline>
```

The pipeline receives the SOAP request in an In-Only MEP. This request is then routed to the `test:soapContentRetreiver` target in an In-Out MEP. The response from `soapContentRetreiver` is then routed to the `exchange-target` of the pipeline, which is another pipeline in the chain, `test:pipelineServiceBefore20061231`.

- **Content retriever:** The functionality of this component is to extract the payload part from the incoming SOAP request. So, we trim out the SOAP envelope and body tags, and extract only the contents within the body element to be sent to the next component. We use a `ServiceMix XsltComponent` as the content retriever. It uses a `stylesheet` to remove all extraneous white spaces from the message. The trimmed message is then routed to the consumer for this component which is the previous transformer pipeline.

```
<sm:activationSpec componentName="soapContentRetreiver"
                    service="test:soapContentRetreiver">
  <sm:component>
    <bean class="org.apache.servicemix.components.
              xslt.XsltComponent">
      <property name="xsltResource"
        value="SoapRequest-To-Request.xsl"/>
    </bean>
  </sm:component>
</sm:activationSpec>
```

This `XsltComponent` also uses a stylesheet to extract the payload from the SOAP message. The stylesheet is shown as follows:

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

<xsl:output method="xml"/>
  <xsl:template match="/">
    <xsl:copy-of select="soapenv:Envelope/soapenv:Body/*" />
  </xsl:template>
</xsl:stylesheet>
```

The output of this component is shown as follows:

```
<ns1:hello soapenv:encodingStyle="http://schemas.xmlsoap.org/
    soap/encoding/"
    xmlns:ns1="http://version20061231.ws.
        servicemix.esb.binildas.com">

  <in0 xsi:type="soapenc:string"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
    Binil
  </in0>
</ns1:hello>
```

- **Service pipeline:** This component is very similar to the transformer pipeline seen above. Let us look at the configuration and then understand it more:

```
<eip:pipeline service="test:pipelineServiceBefore20061231"
    endpoint="pipelineServiceBefore20061231">
  <eip:transformer>
    <eip:exchange-target service="version20061231:
        IHelloWebService" />
  </eip:transformer>
  <eip:target>
    <eip:exchange-target service="test:MyProviderService" />
  </eip:target>
</eip:pipeline>
```

This pipeline will send the SOAP payload alone (without the SOAP envelope) as an In-Out MEP to the `eip:exchange-target` which is `version20061231:IHelloWebService`. Any response is then routed to the `test:MyProviderService`, which is the output queue for the client.

- **HTTP provider:** The HTTP provider is a `servicemix-http` used for HTTP or SOAP binding of services and components into the ServiceMix NMR. A provider role implies that the NMR is the consumer to the component. Hence, the NMR sends out an In-Out to the HTTP provider and the HTTP provider in turn routes the message to the remote web service. The web service gets invoked and the response received is routed back to the consumer of the HTTP provider which is the service pipeline described above. The HTTP provider configuration is shown as follows:

```
<http:endpoint service="version20061231:IHelloWebService"
  endpoint="HelloWebService"
  role="provider"
  locationURI="http://localhost:8080/
    AxisEndToEnd20061231/services/
    HelloWebService20061231"
  soap="true"
  soapAction=""
  wsdlResource="http://localhost:8080/
    AxisEndToEnd20061231/services/
    HelloWebService20061231?WSDL" />
```

Here, the `locationURI` refers to the actual URL where the web service is hosted. As we have two versions of the web services hosted to test the version functionality, we also have another HTTP provider pointing to the other web service versions as shown in the following code:

```
<http:endpoint service="version20070101:IHelloWebService"
  endpoint="HelloWebService"
  role="provider"
  locationURI="http://localhost:8080/
    AxisEndToEnd20070101/services/
    HelloWebService20070101"
  soap="true"
  soapAction=""
  wsdlResource="http://localhost:8080/
    AxisEndToEnd20070101/services/
    HelloWebService20070101?WSDL" />
```

- **Remote web service:** This section needs some detailed explanation, as we need to look into mechanisms on hooking the version control into our development engineering process itself.

We have seen that there are options to set the `targetNamespace` attribute on the WSDL. It is better to hook this to the tool infrastructure associated with web service generation and most tools including the Apache Axis provides mechanisms to do this. Let us look into that now.

Axis provides the Java `org.apache.axis.wsdl.Java2WSDL` class to help us in generating WSDL from the Java interfaces used for creating the web service. The WSDL generation can be controlled using the parameters. Two parameters that are important to us are the following:

```
-n, --namespace <target namespace>
```

indicates the name of the target namespace of the WSDL.

```
-p, --PkgToNS <package> <namespace>
```

indicates the mapping of a package to a namespace.

If a package is encountered that does not have a namespace, the `Java2WSDL` emitter will generate a suitable namespace name. This option may be specified multiple times. By default, `Java2WSDL` will take the package name of the web service interface class.

Now, we need to version control the web service exposure as well as the whole web service generation process. Hence, we have arranged our web service artifacts (source files, and their package names too) in a folder structure which represents the version. By doing that we don't need to control further the `Java2WSDL` process using the above parameters, instead the "versioned" package name of the classes is used to define the `targetNamespace` attribute on the WSDL.

We have two versions, namely 20061231 and 20070101, referring to the two versions of the service, one defined on the 2007 New Year's eve and the other on the new year itself.

Let us look at this in the WSDLs generated for the two web services we have:

In `HelloWebService20061231.wsdl` we have the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://version20061231.ws.
    servicemix.esb.binildas.com" ...>
  <wsdl:types>
    <schema elementFormDefault="qualified"
      targetNamespace="http://version20061231.ws.
        servicemix.esb.binildas.com"
      xmlns="http://www.w3.org/2001/XMLSchema">
    </schema>
  </wsdl:types>
  <wsdl:message name="helloResponse">
    <!-- other code goes here -->
  </wsdl:message>
  <wsdl:portType name="IHelloWeb">
```

```
<!-- other code goes here -->
</wsdl:portType>
<wsdl:binding name="HelloWebService20061231SoapBinding"
              type="impl:IHelloWeb">
    <!-- other code goes here -->
</wsdl:binding>
<wsdl:service name="IHelloWebService">
    <wsdl:port binding="impl:HelloWebService20061231SoapBinding"
              name="HelloWebService20061231">
        <wsdlsoap:address location="http://localhost:8080/
                               AxisEndToEnd20061231/services/
                               HelloWebService20061231"/>
    </wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

In HelloWebService20070101.wsdl, we have the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://version20070101.ws.
                                servicemix.esb.binildas.com" ...>

    <wsdl:types>
        <schema elementFormDefault="qualified"
                  targetNamespace="http://version20070101.ws.
                                servicemix.esb.binildas.com"
                  xmlns="http://www.w3.org/2001/XMLSchema">
        </schema>
    </wsdl:types>
    <wsdl:message name="helloRequest">
        <!-- other code goes here -->
    </wsdl:message>
    <wsdl:portType name="IHelloWeb">
        <!-- other code goes here -->
    </wsdl:portType>
    <wsdl:binding name="HelloWebService20070101SoapBinding"
                  type="impl:IHelloWeb">
        <!-- other code goes here -->
    </wsdl:binding>
    <wsdl:service name="IHelloWebService">
        <wsdl:port binding="impl:HelloWebService20070101SoapBinding"
                  name="HelloWebService20070101">
            <wsdlsoap:address location="http://localhost:8080/
                                   AxisEndToEnd20070101/services/
                                   HelloWebService20070101"/>
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

If you observe these WSDL files closely you can see that the `wsdlsoap:address` points to different addresses for different versions of the service. If we expose the WSDL as such to consumers, then the consumers will be tempted to access the web service using the "Multiple endpoint address" approach. However, we want to demonstrate the covenant-based approach and for that we can edit the `wsdlsoap:address` element in the WSDLs of both the web services to point to the same endpoint address (the JMS consumer address as is in our case, or a HTTP consumer address). If so, our content-based router will do the rest.

- **JMS provider:** The JMS provider is similar to the JMS consumer already discussed and the configuration is listed as follows:

```
<jms:endpoint service="test:MyProviderService"
    endpoint="myProvider"
    role="provider"
    soap="false"
    destinationStyle="queue"
    jmsProviderDestinationName="queue/B"
    connectionFactory="#connectionFactory" />
```

Here, we refer the output queue as "B". This is the queue to where the Service pipeline targets the SOAP response from the remote web service. The JMS client can retrieve the response from this queue.

Deploy and Run the Sample

To build the entire sample, it is easier to change directory to the top-level folder and execute the `build.xml` file provided there. As a first step, if you haven't done it before, find `examples.properties` file provided along with the sample and change the paths accordingly to match your development environment. Then execute the following:

```
cd ch14\WebServiceVersioning
ant
```

This will build the following both versions of web service and place the deployable war file in the `dist` folder as shown as follows:

```
ch14\WebServiceVersioning\01_WebService20061231\dist\
AxisEndToEnd20061231.war

ch14\WebServiceVersioning\02_WebService20070101\dist\
AxisEndToEnd20070101.war
```

You can transfer these war files into the webapps folder of your favorite web container and restart the web server. Make sure your web services are deployed correctly by trying out the following URLs:

```
http://localhost:8080/AxisEndToEnd20061231/services/HelloWebService20061231?WSDL
```

```
http://localhost:8080/AxisEndToEnd20070101/services/HelloWebService20070101?WSDL
```

Now there are also test clients provided for you to test the two versions of the web service. To test that, do the following:

To test the version 20061231 of the service, execute the following commands:

```
cd ch14\WebServiceVersioning\01_WebService20061231
ant run
```

To test the version 20070101 of the service, execute the following commands:

```
cd ch14\WebServiceVersioning\02_WebService20070101
ant run
```

As the second step, you need to bring up your ServiceMix ESB. Before doing that you have to plug-in the Saxon XPath Factory class to the Java run time. This can be done in two steps:

- Place the Saxon jars in %SERVICEMIX_HOME%\lib\optional folder.
- In the %SERVICEMIX_HOME%\bin\servicemix.bat file, add the following entry:
 - set BOOT_OPTS=%BOOT_OPTS% -Djavax.xml.xpath.
XPathFactory:http://java.sun.com/jaxp/xpath/
dom="net.sf.saxon.xpath.XPathFactoryImpl"

Now bring up ServiceMix by executing the following commands:

```
cd ch14\WebServiceVersioning
%SERVICEMIX_HOME%\bin\servicemix servicemix.xml
```

Your web service and ESB are ready now to serve clients. Two clients are provided to test the web service. You can test them by executing the following commands:

```
cd ch14\WebServiceVersioning
ant run1

and

cd ch14\WebServiceVersioning
ant run2
```

run1 will test the web service requesting version 20061231 and run2 will test version 20070101. Both these clients will send messages to the covenant and the covenant will route the requests to different versions of the web service.

For your convenience we have also provided two HTTP clients which will directly send requests to the different versions of the web service endpoint addresses configured in the ESB. They are just for your convenience and they are not intended to test the web service versioning mechanism per se. These clients are:

```
cd ch14\WebServiceVersioning\Client20061231.html
```

and

```
cd ch14\WebServiceVersioning\Client20070101.html
```

Web Service Versioning Operational Perspective

At the operations perspective, we can use tools to edit the WSDLs to reflect this change and then place all the WSDLs in a repository for the consumers to find or even supply these WSDLs to the consumers. The consumers can then use their conventional tools to generate client stubs and send SOAP requests to access the service. The version aspect is not revealed to the clients, instead they are limited to the WSDLs.

As we are using an ESB as the middleware messaging infrastructure, another option is that the WSDLs can be retrieved from their original source (for example, <http://localhost:8080/AxisEndToEnd20061231/services/HelloWebService20061231?WSDL>) on demand basis from the client. Moreover, do an XSL transform to replace the different endpoint addresses with a single covenant address and supply them.

Summary

While the need to version a service is still to be debated, one aspect which we all need to accept is that services need to be maintained in multiple variants to satisfy multiple classes of consumers. Service request can be enhanced to include additional information which will help providers to apply rules to route requests to the appropriate class or variant of the service. While it is tricky to handle this kind of routing using a trivial handler or interceptor-level, an ESB provides you all the required design patterns and hooks to enforce content-based routing. Then neither the service provider nor the service consumer needs to be aware of these complexities. These design patterns and hooks are grouped under the broader heading of EIP and the next chapter is going to look at them in greater depth, again with working code samples.

15

Enterprise Integration Patterns in ESB

By now you will appreciate the fact that integration is not simple. Now, how can you make it manageable and/or repeatable and thus simplify integration? Haven't you come across the same problem in traditional software engineering before? How did you manage the problem then? Yes, I am referring to nothing else other than Patterns.

In this chapter, we are going to cover the following:

- EAI patterns in general
- EAI patterns in ServiceMix
- Working code demonstrating EAI patterns

Enterprise Integration Patterns

Enterprise integration has to be made simpler. We need to apply best practices tested and proven in the traditional software engineering paradigms like OOP to enterprise integration scenarios also. Reusability is a prime concern in all IT design related decisions and this reusability has to happen at multiple levels including architecture, design, implementation, and testing. While we architect or design for integration problems, one way of reusing the existing artifacts is to abstract out the common methods used for solutions and apply them again and again for recurring, similar problems – we call this EIP. Let us delve more into this.

What are EAI Patterns?

Even though we do enterprise integration daily knowingly or unknowingly, it takes some experience and a holistic approach to separate out the integration aspects from the routine application development aspects in a systems environment. When we understand this difference, we have taken the first step in recognizing EAI as a separate stream, in fact a specialized stream which requires specific skill sets to look at systems and services from an integration point of view.

We will start thinking in terms of connectors, brokers, or message routers which have a very specific role and responsibility in the integration domain. These are integration blocks which when combined together in different ways will give out new styles or patterns of message exchange. Hence, EAI patterns are nuggets of advice made out of aggregating basic MEP elements to solve frequently recurring integration problems. For every practical purpose these are similar to design patterns and we can look at EAI patterns as design patterns for solving integration problems.

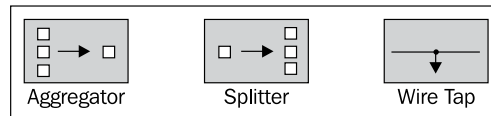
EAI Patterns Book and Site

In writing a chapter discussing EAI patterns, I have to invariably point to the great book on EAI patterns by Gregor Hohpe and Bobby Woolf. In fact, this book is a collection of about 65 patterns, all providing means to solve the day to day integration problems. Each of the patterns is described using the following subheadings:

- Name
- Icon
- Context
- Problem
- Forces
- Solution
- Sketch
- Results
- Next
- Sidebars
- Examples

The beauty of these patterns is that they solve not only common integration problems but also enable the integration architect to combine them together to create designs. Using these they can solve problems which they had never thought about.

Equally important is the influence of the diagrams and notations in the integration patterns. This helps architects and designers to share a common vocabulary in integration. The EAI book and the associated site provide notations which have a certain "sketch" quality. Then it is not required to read hundreds of pages of a manual like that of UML to understand and use. However, due to the objective-oriented notations, it is easy to convey the essence of the pattern to the reader at a quick glance. For example, let us look into a few notations given in the following:



Let us now look into simple definitions for the patterns corresponding to the notations shown above:

- **Aggregator:** An aggregator is used to collect and store individual message parts until a complete set of co-related message parts has been received. Once all the related parts have been received, they are aggregated together to form a single message.
- **Splitter:** A splitter can split out a composite message into a series of individual message parts.
- **Wire Tap:** A wire tap can consume messages from a single input channel and publish the unmodified message to two output channels.

Having gone through the simple explanation for the above notations, if you now look at the notations per se, you will appreciate how easy is it to correlate the functionality with its notation.

ServiceMix EAI Patterns

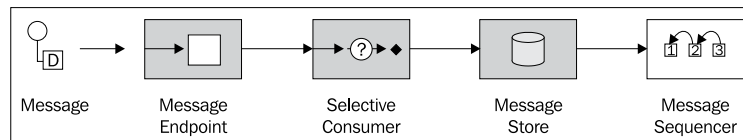
ServiceMix and JBI are all about integrating business services. Let us now see what ServiceMix has to offer in terms of the EAI patterns.

Why ServiceMix for EAI Patterns?

ServiceMix provides the `servicemix-eip` component as a standard JBI-compliant component. This provides implementations for many of the patterns discussed in the EAI patterns book. When I first read the EAI patterns book, I was wondering whether we have all these patterns together in some reusable form, other than just patterns and diagrams. But as we all agree, patterns are nuggets of guidelines which we can implement in many ways, using any technology or platform.

In fact, if we look at many MOM for integrations such as Websphere MQ, Microsoft Biztalk, TIBCO, Webmethods, and Microsoft MQ, they already provide many of these patterns at the framework-level. Perhaps, we didn't notice them as patterns in the first go, but surely as experience builds up we cannot miss a repeated way of addressing a problem of a particular nature.

Months after my first reading of the EAI patterns book I started evaluating a few of the SOI-based products, especially in the Java world including Mule and the ESB framework. The best part of these frameworks is that we have many of the EAI patterns in the form of code! Believe me, if I were to have used these integration frameworks a few years back, it would have saved me many man months! Today, I am wondering how much code our team has written previously for receiving, selecting, storing, and then sequencing, and aggregating multiple passenger name lists (PNL) messages for a major airline. Look at the flow given in the following:



Here, I used EAI pattern blocks to connect them together to create my own design for my PNL part merging problem. For this, we no longer write Java or .NET code to parse my entire PNL message part, select them, and store them and finally when all parts are reached, to resequence them to form the full PNL. Instead, as these individual problems are already addressed as EAI pattern components, I just need to select the required blocks, configure, and connect them together and route my message through that. Don't you think that is a smart way of doing things?

I can understand, you may not agree if you are reading about the EAI patterns for the first time. However, I can guarantee that once you finish reading this chapter and also a couple of more chapters in this book which solves some of the major headaches in the industry, you will be in a better position to appreciate the importance of EAI patterns in integration.

For example, building a protocol bridge to bring reliability to a web service channel or to implement a web service versioning mechanism, you no more think traditionally. However, I need to remind you that EAI, EAI patterns, and MOM are great enablers for SOA and SOI, but may not be the solution for every problem at hand. In other words, none of these patterns or frameworks is going to replace the judicious decision you are obliged to take as a designer or an architect, to pick the best technology or frameworks to solve your problems.

Hence, by any chance if you decide that MOM is the way to go and you want to use ServiceMix as your ESB middleware, you can use many EAI pattern blocks available here. The ServiceMix `servicemix-eip` component is a routing container onto which you can deploy your own EAI patterns or combination of patterns to solve complex routing or MEP problems.

For example, you may sometimes want to transform an In-Out MEP to a combination of In-Only MEP or you may want to route your messages destination. Parallely you also want to deliver a copy of the message to a trace component in between. Don't ever write code for these in ServiceMix, instead route your messages through a suitable EAI pattern block deployed onto `servicemix-eip`.

ServiceMix EAI Patterns Configuration

We can configure `servicemix-eip` either in a SU as a standard JBI component or using the `servicemix.xml` configuration file. This is shown in the following list:

- **Configure `servicemix-eip` as a standard JBI component:** `servicemix-eip` supports the standard XBean-based deployment. For this, the `xbean.xml` file will have the following entry:

```
<beans xmlns:eip="http://servicemix.apache.org/eip/1.0">
  <!-- configure your EAI Pattern components here -->
</beans>
```

- **Configure `servicemix-eip` using the `servicemix.xml` file:**

```
<beans xmlns:sm="http://servicemix.apache.org/config/1.0"
  xmlns:eip="http://servicemix.apache.org/eip/1.0"
  xmlns:test="http://test.eip.servicemix.esb.binildas.com">
  <sm:container ...>
    <sm:activationSpecs>
      <sm:activationSpec>
        <sm:component>
          <eip:component>
            <eip:endpoints>
              <!-- configure EAI Pattern components here -->
            </eip:endpoints>
          </eip:component>
        </sm:component>
      </sm:activationSpec>
    </sm:activationSpecs>
  </sm:container>
</beans>
```

EAI Patterns—Code and Run Samples in ESB

The main EAI pattern components supported by ServiceMix are listed as follows:

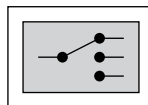
- Content-based router
- Content enricher
- XPath splitter
- Message filter
- Split aggregator
- Pipeline
- Wiretap
- Static recipient list
- Static routing slip

Let us now configure EAI pattern components in the ServiceMix and run the samples. All the samples are arranged in subfolders under `ch15\`. Make sure that you edit `examples.PROPERTIES` and change the paths there to match your development environment to build the samples.

Content-based Router

A content-based router consumes a message from one message channel. Based on a set of conditions on the headers or the body content of the message, it republishes the message on to a different message channel.

Notation



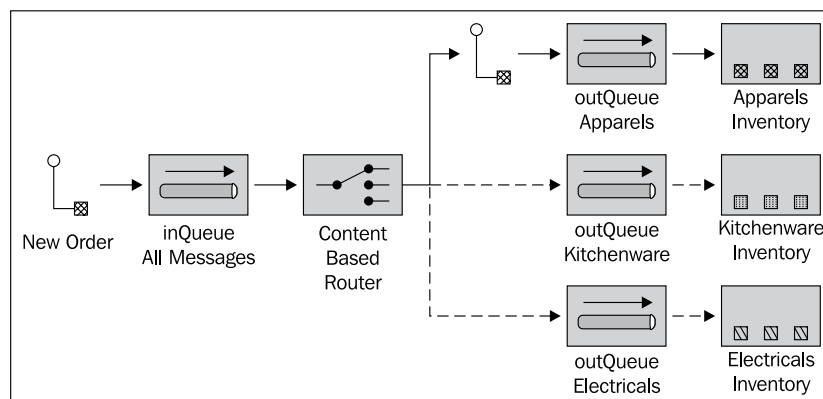
Explanation

The message router inspects the contents of the message and routes the message to multiple channels. While doing so, the router will not alter the message contents. While inspecting, the router can look into a field in the message body or the message header. Usually, a condition or a rule is attached to the router which will try a match with the field in the message. Hence this rule matching hook is to be designed as extensible and is the target of constant maintenance when we want to add more rules or when we want to plug-in more channels matching the rules.

Illustrative Design

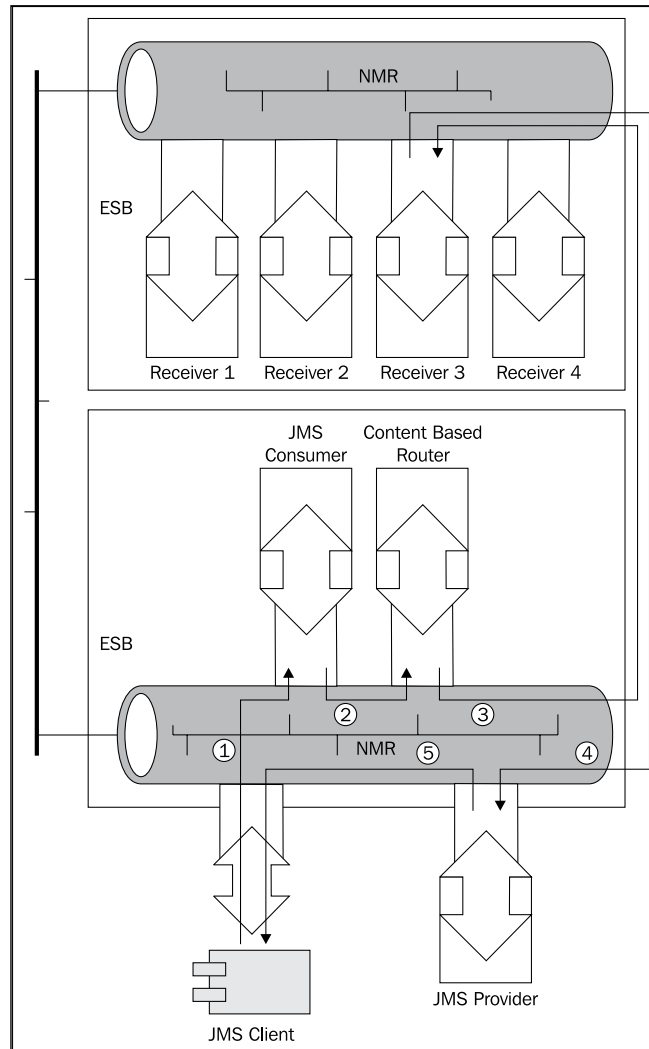
Let us consider the Acme Company providing the web interface to end customers to place orders for buying the gadgets online. The Acme ordering gateway can provide a single channel for all the incoming messages due to maintenance and security reasons. Now we need a mechanism to segregate out orders for different kinds of gadgets. Each kind of gadgets (apparels and electricals, for example) has its own inventory system. Hence we need to route different kinds of orders to their respective inventory systems.

We plug-in a content-based router and attach multiple outgoing channels to this router. The router will look at the message contents and identify the kind of order. Based on that, it will route the message to their respective outgoing channels from where the respective inventory systems can pick up the messages. This is shown in the following figure:



Sample Use Case

The sample use case has a set of components as shown in the following figure:



Let us now look into the individual components in detail in the following list:

- **JMS client:** This is a normal external JMS client, placing different XML messages onto the JMS consumer component configured within the ESB.
- **JMS consumer:** This is a `servicemix-jms` listening on queue "A". Any incoming messages to this queue will be routed to the next component in the flow chain, the content-based router.

- **Content-based router:** The content-based router is a `servicemix-eip` component. Based on the content of the XML message it receives, it will route the message, unaltered, to any one amongst the set of Receiver components configured.
- **Receiver:** The receiver component is a custom transform component. Any message it receives will be logged into the console and then echoed by writing back to the out message of the In-Out. The out message is placed in the out queue.
- **JMS provider:** This is a `servicemix-jms` listening on queue "B". The receiver component will place its out message onto this queue from where the JMS client can pick up (even though in this sample we don't pick up the messages).

Sample Code and Configuration

We configure the content-based router in the `servicemix.xml` file, along with other components described above. This is shown in the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xbean.org/schemas/spring/1.0"
       xmlns:sm="http://servicemix.apache.org/config/1.0"
       xmlns:test="http://cbr.eip.servicemix.apache.binildas.com"
       xmlns:eip="http://servicemix.apache.org/eip/1.0"
       xmlns:jms="http://servicemix.apache.org/jms/1.0">
  <classpath>
    <location>./location</location>
  </classpath>
  <import resource="classpath:jmx.xml" />
  <import resource="classpath:jndi.xml" />
  <import resource="classpath:security.xml" />
  <import resource="classpath:tx.xml" />
  <import resource="classpath:activemq.xml" />
  <bean id="propertyConfigurer" class="org.springframework.beans.
    factory.config.PropertyPlaceholderConfigurer">
    <property name="location"
      value="classpath:servicemix.properties" />
  </bean>
  <sm:container name="jbi"
    monitorInstallationDirectory="false"
    createMBeanServer="true"
    useMBeanServer="true">
    <sm:activationSpecs>
      <sm:activationSpec>
        <sm:component>
```



```
<jms:component>
  <jms:endpoints>
    <jms:endpoint service="test:MyConsumerService"
      endpoint="myConsumer"
      role="consumer"
      soap="false"
      targetService="test:router"
      defaultMep="http://www.w3.org/2004/
        08/wsdl/in-only"
      destinationStyle="queue"
      jmsProviderDestinationName="queue/A"
      connectionFactory="#connectionFactory"
      " />
    <jms:endpoint service="test:MyProviderService"
      endpoint="myProvider"
      role="provider"
      soap="false"
      destinationStyle="queue"
      jmsProviderDestinationName="queue/B"
      connectionFactory="#connectionFactory"
      " />
  </jms:endpoints>
</jms:component>
</sm:component>
</sm:activationSpec>
<sm:activationSpec componentName="receiver1"
  service="test:receiver1"
  destinationService="test:MyProviderService">
  <sm:component>
    <bean xmlns="http://xbean.org/schemas/spring/1.0"
      class="MyReceiver" >
      <property name="name">
        <value>1</value>
      </property>
    </bean>
  </sm:component>
</sm:activationSpec>
<sm:activationSpec componentName="receiver2"
  service="test:receiver2"
  destinationService="test:MyProviderService">
  <sm:component>
    <bean xmlns="http://xbean.org/schemas/spring/1.0"
      class="MyReceiver" >
      <property name="name">
        <value>2</value>
      </property>
    </bean>
```

```

    </sm:component>
</sm:activationSpec>
<sm:activationSpec componentName="receiver3"
    service="test:receiver3"
    destinationService="test:MyProviderService">
    <sm:component>
        <bean xmlns="http://xbean.org/schemas/spring/1.0"
            class="MyReceiver" >
            <property name="name">
                <value>3</value>
            </property>
        </bean>
    </sm:component>
</sm:activationSpec>
<sm:activationSpec componentName="receiver4"
    service="test:receiver4"
    destinationService="test:MyProviderService">
    <sm:component>
        <bean xmlns="http://xbean.org/schemas/spring/1.0"
            class="MyReceiver" >
            <property name="name">
                <value>4</value>
            </property>
        </bean>
    </sm:component>
</sm:activationSpec>
<sm:activationSpec id="servicemix-eip">
    <sm:component>
        <eip:component>
            <eip:endpoints>
                <eip:content-based-router service="test:router"
                    endpoint="endpoint">
                    <eip:rules>
                        <eip:routing-rule>
                            <eip:predicate>
                                <eip:xpath-predicate
                                    xpath="number(/hello/@id) < 2" />
                                </eip:predicate>
                                <eip:target>
                                    <eip:exchange-target service=
                                        "test:receiver1" />
                                </eip:target>
                            </eip:routing-rule>
                        <eip:routing-rule>
                            <eip:predicate>

```

```
        <eip:xpath-predicate xpath="number
            (/hello/@id) = 2" />
    </eip:predicate>
    <eip:target>
        <eip:exchange-target
            service="test:receiver2" />
    </eip:target>
</eip:routing-rule>
<eip:routing-rule>
    <eip:predicate>
        <eip:xpath-predicate
            xpath="number(/hello/@id) > 3" />
    </eip:predicate>
    <eip:target>
        <eip:exchange-target
            service="test:receiver4" />
    </eip:target>
</eip:routing-rule>
<eip:routing-rule>
    <eip:target>
        <eip:exchange-target
            service="test:receiver3" />
    </eip:target>
</eip:routing-rule>
</eip:rules>
</eip:content-based-router>
</eip:endpoints>
</eip:component>
</sm:component>
</sm:activationSpec>
</sm:activationSpecs>
</sm:container>
<bean id="connectionFactory"
    class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL" value="tcp://localhost:61616" />
</bean>
</beans>
```

The components configured in the above JBI configuration have already been explained in the previous section. We need to declare the eip namespace as `xmlns:eip=http://servicemix.apache.org/eip/1.0`. We will also use the `servicemix-jms` component for which we declare the JMS namespace as `xmlns:jms=http://servicemix.apache.org/jms/1.0`.

The `MyReceiver` class is shown as follows:

```
public class MyReceiver extends TransformComponentSupport
    implements MessageExchangeListener
{
    private String name;
    public void setName(String name){this.name = name;}
    public String getName(){return name;}
    protected boolean transform(MessageExchange exchange,
        NormalizedMessage in,NormalizedMessage out)
        throws MessagingException
    {
        NormalizedMessage copyMessage = exchange.createMessage();
        getMessageTransformer().transform(exchange, in, copyMessage);
        Source content = copyMessage.getContent();
        String contentString = null;
        if (content instanceof DOMSource)
        {
            contentString = XMLUtil.node2XML(((DOMSource)
                content).getNode());
        }
        if (content instanceof StreamSource)
        {
            contentString = XMLUtil.formatStreamSource((StreamSource)
                content);
        }
        System.out.println("MyReceiver.transform(" + name + ").
            contentString = " + contentString);
        out.setContent(new StringSource(contentString));
        return true;
    }
}
```

The `MyReceiver` class prints out the message and echoes the same message back.

Deploy and Run the Sample

To build the sample, change directory to `ch15\01_ContentBasedRouter` and type the following:

```
ant
```

This will compile all the files, including the JMS client program. Now to test the sample, first bring ServiceMix up by executing the following commands:

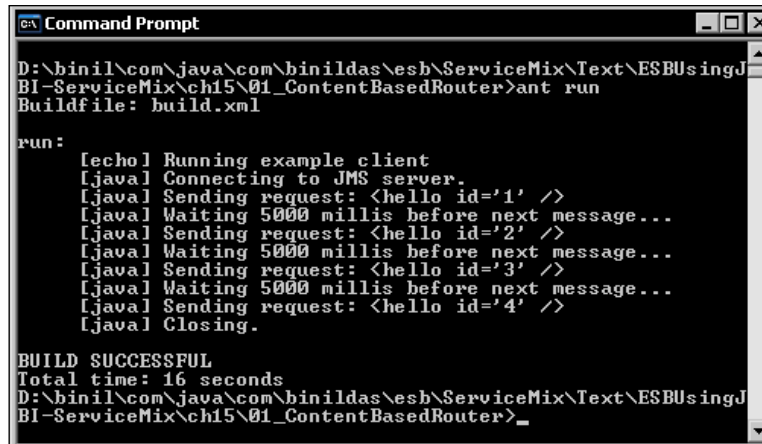
```
cd ch15\01_ContentBasedRouter
%SERVICEMIX_HOME%\bin\servicemix servicemix.xml
```

Now in a different command prompt execute `ant run`, as shown here:

```
cd ch15\01_ContentBasedRouter
```

```
ant run
```

The JMS client program console will print out the messages it sends to the ESB. This is shown in the following screenshot:

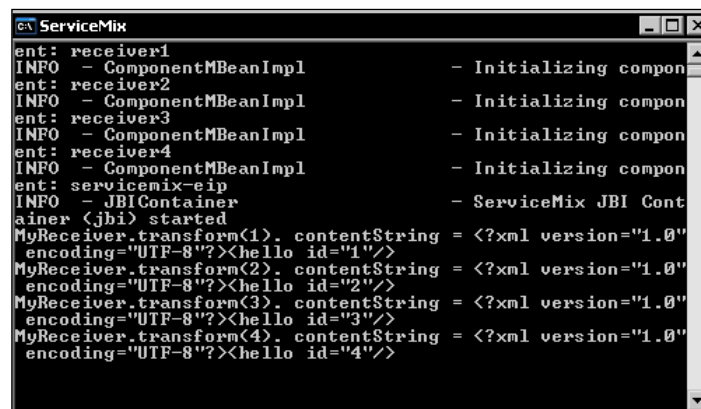


```
Command Prompt
D:\binil\com\java\com\binildas\esb\ServiceMix\Text\ESBUsingJBI-ServiceMix\ch15\01_ContentBasedRouter>ant run
Buildfile: build.xml

run:
[echo] Running example client
[javal] Connecting to JMS server.
[javal] Sending request: <hello id='1' />
[javal] Waiting 5000 millis before next message...
[javal] Sending request: <hello id='2' />
[javal] Waiting 5000 millis before next message...
[javal] Sending request: <hello id='3' />
[javal] Waiting 5000 millis before next message...
[javal] Sending request: <hello id='4' />
[javal] Closing.

BUILD SUCCESSFUL
Total time: 16 seconds
D:\binil\com\java\com\binildas\esb\ServiceMix\Text\ESBUsingJBI-ServiceMix\ch15\01_ContentBasedRouter>
```

Now, have a look at the content-based router. As shown in the content-based router configuration, we have the content-based router rules embedded as the XPath predicates. Based on the match in the message content, the router will route the message to their respective destinations (Receiver 1 or 2 or 3 or 4). The ESB-side console printout validates this as shown in the following screenshot:

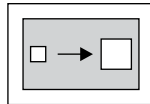


```
ServiceMix
ent: receiver1
INFO - ComponentMBeanImpl - Initializing compon
ent: receiver2
INFO - ComponentMBeanImpl - Initializing compon
ent: receiver3
INFO - ComponentMBeanImpl - Initializing compon
ent: receiver4
INFO - ComponentMBeanImpl - Initializing compon
ent: servicemix-eip
INFO - JBIContainer - ServiceMix JBI Cont
ainer <jbi> started
MyReceiver.transform(1). contentString = <?xml version="1.0"
encoding="UTF-8"?><hello id="1"/>
MyReceiver.transform(2). contentString = <?xml version="1.0"
encoding="UTF-8"?><hello id="2"/>
MyReceiver.transform(3). contentString = <?xml version="1.0"
encoding="UTF-8"?><hello id="3"/>
MyReceiver.transform(4). contentString = <?xml version="1.0"
encoding="UTF-8"?><hello id="4"/>
```

Content Enricher

The content enricher supplements the original message with more related information retrieved from the other sources. The original message will be altered to contain enriched information.

Notation

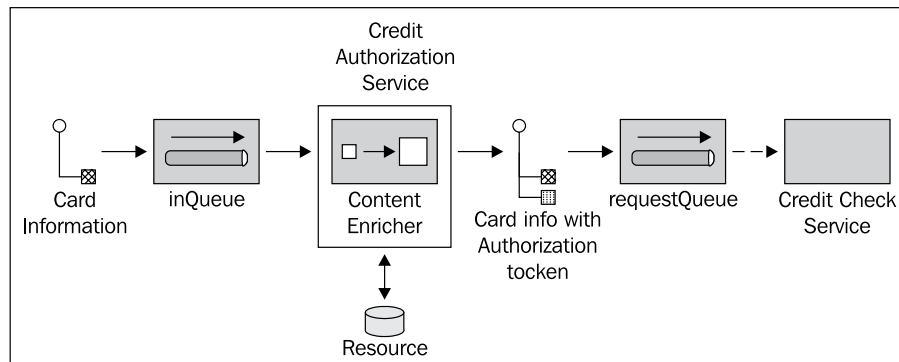


Explanation

Sometimes a message may not contain all the required information for the next processing step to act or the next processing required more additional information to be appended to the incoming message. In such scenarios, the content enricher can append the additional information to the original message and then send to the next link in the processing chain.

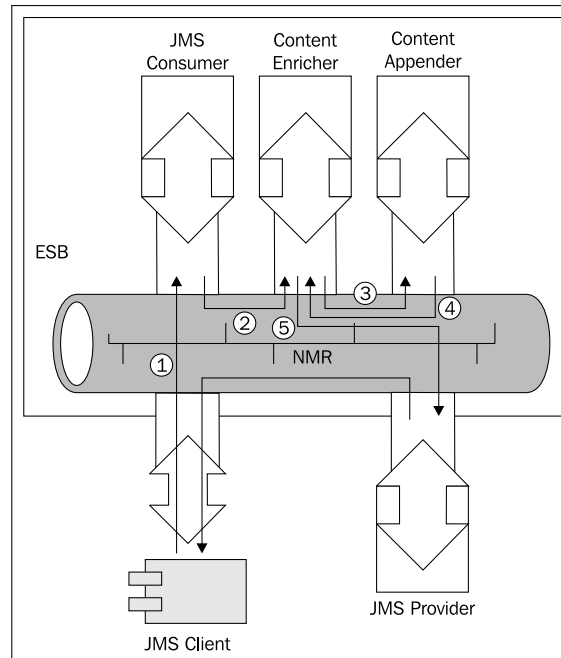
Illustrative Design

The Acme online customers at the time of checkout will enter credit card details. The Acme back end would need to do credit authorization. However, the data entered by the web page customers might be minimal and for the actual authorization we might need to add more details such as SSN, before the request is routed to Credit Check Service. A Content Enricher can get the customer identification data (customer key) from the incoming message and retrieve the extra details from a local resource store. The extra details can now be appended to the original message and then routed to the target service. The whole setup is shown in the following figure:



Sample Use Case

The following figure illustrates how various components can be assembled in the JBI bus for our sample use case:



The sample use case will have following components:

- **JMS client:** This is a normal external JMS client, placing XML messages onto the JMS Consumer component configured within the ESB.
- **JMS consumer:** This is a `servicemix-jms` listening on queue "A". Any incoming messages to this queue will be routed to the next component in the flow chain which is the Content Enricher.
- **Content enricher:** The Content Enricher router is a `servicemix-eip` component. It enriches the original message with additional information. The enriched message will be placed back in the JMS Provider queue.
- **Content appender:** The Content Appender is a custom transform component. It provides a `resultElement` with name `test:kerberosticket` so that the content enricher can enrich the original message with this additional `resultElement`.
- **JMS provider:** This is a `servicemix-jms` listening on queue "B". The Content Enricher component will place its out message into this queue from where the JMS Client can pick up.

Sample code and configuration

We configure the content enricher in the `servicemix.xml` file, along with other components described above. The content of this file is shown in the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xbean.org/schemas/spring/1.0"
       xmlns:sm="http://servicemix.apache.org/config/1.0"
       xmlns:test="http://cer.eip.servicemix.esb.binildas.com"
       xmlns:eip="http://servicemix.apache.org/eip/1.0"
       xmlns:jms="http://servicemix.apache.org/jms/1.0">
  <classpath>
    <location>./</location>
  </classpath>
  <import resource="classpath:jmx.xml" />
  <import resource="classpath:jndi.xml" />
  <import resource="classpath:security.xml" />
  <import resource="classpath:tx.xml" />
  <import resource="classpath:activemq.xml" />
  <bean id="propertyConfigurer" class="org.springframework.beans.
    factory.config.PropertyPlaceholderConfigurer">
    <property name="location"
      value="classpath:servicemix.properties" />
  </bean>
  <sm:container name="jbi"
    monitorInstallationDirectory="false"
    createMBeanServer="true"
    useMBeanServer="true">
    <sm:activationSpecs>
      <sm:activationSpec>
        <sm:component>
          <jms:component>
            <jms:endpoints>
              <jms:endpoint service="test:MyConsumerService"
                endpoint="myConsumer"
                role="consumer"
                soap="false"
                targetService="test:contentEnricher"
                defaultMep="http://www.w3.org/2004/
                  08/wsdl/in-only"
                destinationStyle="queue"
                jmsProviderDestinationName="queue/A"
                connectionFactory =
                  "#connectionFactory" />
              <jms:endpoint service="test:MyProviderService"
                endpoint="myProvider"
                role="provider"
                soap="false"
                destinationStyle="queue"
                jmsProviderDestinationName="queue/B"
```



```

        connectionFactory=
            "#connectionFactory" />
    </jms:endpoints>
</jms:component>
</sm:component>
</sm:activationSpec>
<sm:activationSpec id="servicemix-eip">
    <sm:component>
        <eip:component>
            <eip:endpoints>
                <eip:content-enricher
                    service="test:contentEnricher"
                    endpoint="endpoint"
                    enricherElementName="test:subject"
                    requestElementName="test:principal"
                    resultElementName="test:credential">
                    <eip:enricherTarget>
                        <eip:exchange-target
                            service="test:contentAppender" />
                        </eip:enricherTarget>
                        <eip:target>
                            <eip:exchange-target
                                service="test:MyProviderService" />
                            </eip:target>
                        </eip:content-enricher>
                    </eip:endpoints>
                </eip:component>
            </sm:component>
        </sm:activationSpec>
        <sm:activationSpec componentName="contentAppender"
            service="test:contentAppender">
            <sm:component>
                <bean xmlns="http://xbean.org/schemas/spring/1.0"
                    class="ContentAppender" >
                    <constructor-arg ref="jbi"/>
                </bean>
            </sm:component>
        </sm:activationSpec>
    </sm:activationSpecs>
</sm:container>
<bean id="connectionFactory"
    class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL" value="tcp://localhost:61616" />
</bean>
</beans>
```

When we configure the content enricher we can specify the `enricherElementName`, `requestElementName`, and the `resultElementName`. The output of the content enricher will be wrapped within these elements as seen in the sample output in the following screenshot. Let us also look at the `ContentAppender` Java class:

```
public class ContentAppender extends ComponentSupport
    implements MessageExchangeListener
{
    private JBIContainer container;
    public ContentAppender(JBIContainer container)
    {
        this.container = container;
    }
    public void onMessageExchange(MessageExchange exchange)
        throws MessagingException
    {
        if (exchange.getStatus() == ExchangeStatus.ACTIVE)
        {
            boolean txSync = exchange.isTransacted() && Boolean.TRUE.
                equals(exchange.getProperty(JbiConstants.SEND_SYNC));
            NormalizedMessage out = exchange.createMessage();
            out.setContent(new StringSource("<?xml version='1.0'
                encoding='UTF-8'?><test:kerberosticket xmlns:test=
                \"http://xslt.servicemix.apache.binildas.com\">
                123456789</test:kerberosticket>"));
            exchange.setMessage(out, "out");
            if (txSync)
            {
                sendSync(exchange);
            }
            else
            {
                send(exchange);
            }
        }
    }
}
```

Here, we write the information to enrich the original message to the out. In actual scenarios, we may want to retrieve some key from the incoming message, retrieve more data from a local resource using this key and generate the additional information.

Deploy and Run the Sample

To build the sample, change directory to `ch15\02_ContentEnricher` and type `ant`. This is shown as follows:

```
cd ch15\02_ContentEnricher
ant
```

This will compile all the files, including the JMS client program. Now to test the sample, first, bring the ServiceMix up by executing the following:

```
cd ch15\02_ContentEnricher
%SERVICEMIX_HOME%\bin\servicemix servicemix.xml
```

Now in a different command prompt execute `ant run`.

```
cd ch15\02_ContentEnricher
ant run
```

The JMS client program console will print out the messages it sends to the ESB. It also prints out the response message from the content enricher. The JMS client console is shown in the following screenshot:



```
Command Prompt
D:\binil\com\java\com\binildas\esb\ServiceMix\Text\ESBUsingJ
BI-ServiceMix\ch15\02_ContentEnricher>ant run
Buildfile: build.xml

run:
[echo] Running example client
[java] Connecting to JMS server.
[java] Sending request: <?xml version='1.0' encoding='U
TF-8'?><test:username xmlns:test='http://xslt.servicemix.apa
che.binildas.com'>Binildas</test:username>
[java] Response was: <?xml version='1.0' encoding='UTF-
8'?><test:subject xmlns:test='http://cer.eip.servicemix.esb.
binildas.com'><test:principal><test:username xmlns:test='htt
p://xslt.servicemix.apache.binildas.com'>Binildas</test:user
name></test:principal><test:credential><test:kerberosticket
xmlns:test='http://xslt.servicemix.apache.binildas.com'>1234
56789</test:kerberosticket></test:credential></test:subject>

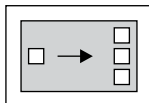
[java] Closing.

BUILD SUCCESSFUL
Total time: 1 second
D:\binil\com\java\com\binildas\esb\ServiceMix\Text\ESBUsingJ
BI-ServiceMix\ch15\02_ContentEnricher>
```

XPath Splitter

An XPath splitter is based on the original splitter EAI pattern. A splitter can identify repeating elements in a message and split the message and publish each element part as separate messages to a different channel. The splitter can also separate out non-repeating elements, in such case the published message will be a subset of the original message. XPath splitter uses XPath to find the repeating element pattern.

Notation

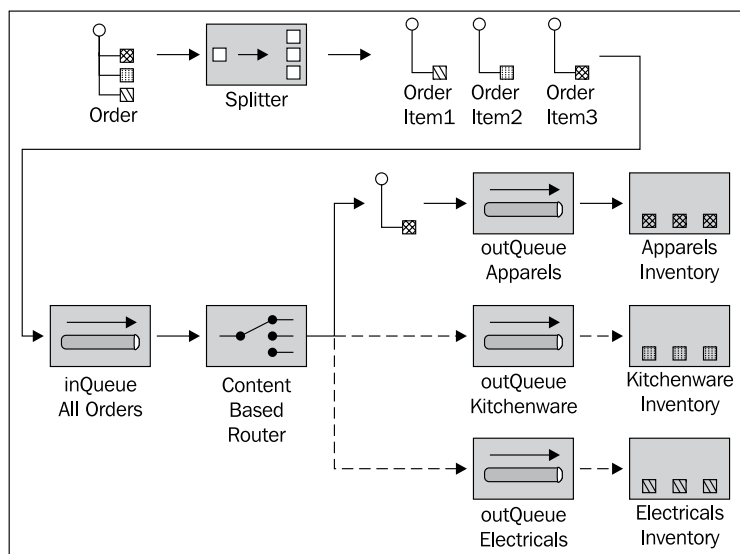


Explanation

Many composite documents like a full airline passenger name list or an order with many order items will contain repeating elements. Sometimes we may need to process each of these repeating elements separately. An XPath splitter can split the composite message into individual parts based on the repeat pattern and publishes the parts onto different destinations.

Illustrative Design

Using the Acme's online e-commerce pages, a customer can add multiple items to the shopping cart and at the end of the shopping trip he can check out. This will submit a single order in the back end, but the order can contain multiple order items. Now, for each order item, we need to do a separate inventory check. To solve this problem, we can route the order to an XPath splitter first. The splitter can split the order into individual order items and each order item can be pushed to the queue as a separate message. Now it is easy to plug-in a content-based router as we have discussed already. Then each new message encapsulating one order item can be routed to their respective inventory queues.

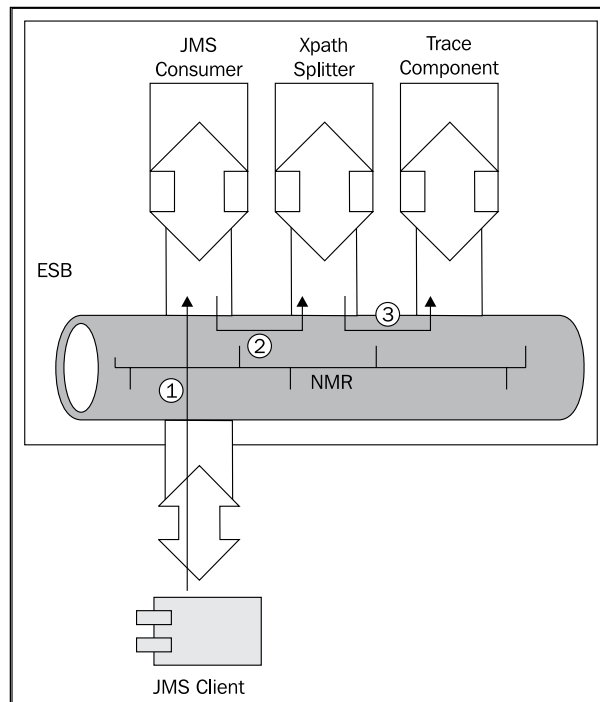


Sample Use Case

The sample use case will have the following components:

- **JMS client:** This is a normal external JMS client, placing the XML composite messages onto the JMS consumer component configured within the ESB.
- **JMS consumer:** This is a `servicemix-jms` listening on queue "A". Any incoming messages to this queue will be routed to the next component in the flow chain which is the XPath splitter.
- **XPath splitter:** The XPath splitter is a `servicemix-eip` component. On receiving the in messages, the splitter will try to match the XPath configured at the splitter-level with the message content. In finding a match, the splitter will split the original message into as many parts as there are repeating elements as per the XPath. Each of these individual elements is republished as a separate message to the exchange-target which is a trace component.
- **Trace component:** The trace component just spits out whatever message it receives into the console.

The following figure illustrates how the various components can be assembled in the JBI bus for our sample use case:



Sample Code and Configuration

We configure the XPath splitter in `servicemix.xml`, along with other components described above:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xbean.org/schemas/spring/1.0"
       xmlns:sm="http://servicemix.apache.org/config/1.0"
       xmlns:test="http://xslt.servicemix.apache.binildas.com"
       xmlns:eip="http://servicemix.apache.org/eip/1.0"
       xmlns:jms="http://servicemix.apache.org/jms/1.0">
  <classpath>
    <location>.</location>
  </classpath>
  <import resource="classpath:jmx.xml" />
  <import resource="classpath:jndi.xml" />
  <import resource="classpath:security.xml" />
  <import resource="classpath:tx.xml" />
  <import resource="classpath:activemq.xml" />
  <bean id="propertyConfigurer"
        class="org.springframework.beans.factory.config.
              PropertyPlaceholderConfigurer">
    <property name="location"
              value="classpath:servicemix.properties" />
  </bean>
  <sm:container name="jbi"
                monitorInstallationDirectory="false"
                createMBeanServer="true"
                useMBeanServer="true">
    <sm:activationSpecs>
      <sm:activationSpec>
        <sm:component>
          <jms:component>
            <jms:endpoints>
              <jms:endpoint service="test:MyConsumerService"
                            endpoint="myConsumer"
                            role="consumer"
                            soap="false"
                            targetService="test:xpathSplitter"
                            defaultMep="http://www.w3.org/2004/
                                      08/wsdl/in-only"
                            destinationStyle="queue"
                            jmsProviderDestinationName="queue/A"
                            connectionFactory=
                              "#connectionFactory" />
            </jms:endpoints>
          </jms:component>
        </sm:component>
      </sm:activationSpec>
    </sm:activationSpecs>
  </sm:container>
</beans>
```

```
</sm:activationSpec>
<sm:activationSpec id="servicemix-eip">
  <sm:component>
    <eip:component>
      <eip:endpoints>
        <eip:xpath-splitter service="test:xpathSplitter"
                           endpoint="xpathSplitterEndpoint"
                           xpath="/hello/*" >
          <eip:target>
            <eip:exchange-target service="my:trace" />
          </eip:target>
        </eip:xpath-splitter>
      </eip:endpoints>
    </eip:component>
  </sm:component>
</sm:activationSpec>
<sm:activationSpec componentName="trace" service="my:trace">
  <sm:component>
    <bean class="org.apache.servicemix.components.
           util.TraceComponent" />
  </sm:component>
</sm:activationSpec>
</sm:activationSpecs>
</sm:container>
<bean id="connectionFactory"
      class="org.apache.activemq.ActiveMQConnectionFactory">
  <property name="brokerURL" value="tcp://localhost:61616" />
</bean>
</beans>
```

The XPath we configured here is `/hello/*`. This will split each element inside the `hello` element into individual messages.

Deploy and Run the Sample

To build the sample, change directory to `ch15\03_XPathSplitter` and type `ant` as given here:

```
cd ch15\03_XPathSplitter
ant
```

This will compile all the files, including the JMS client program. Now to test the sample, first bring ServiceMix up by executing the following commands:

```
cd ch15\03_XPathSplitter
%SERVICEMIX_HOME%\bin\servicemix servicemix.xml
```

Now in a different command prompt execute `ant run` as shown here:

```
cd ch15\03_XPathSplitter
ant run
```

The JMS client program console will print out the messages it sends to the ESB as shown in the following screenshot:

```

C:\ Command Prompt
D:\binil\com\java\com\binildas\esb\ServiceMix\Text\ESBUsingJ
BI-ServiceMix\ch15\03_XPathSplitter>ant run
Buildfile: build.xml

run:
[echo] Running example client
[java] Connecting to JMS server.
[java] Sending request: <hello><one/><two/><three/></he
llo>
[java] Waiting 5000 millis before next message...
[java] Sending request: <hello/>
[java] Waiting 5000 millis before next message...
[java] Sending request: <hello><four/><five/></hello>
[java] Closing.

BUILD SUCCESSFUL
Total time: 11 seconds
D:\binil\com\java\com\binildas\esb\ServiceMix\Text\ESBUsingJ
BI-ServiceMix\ch15\03_XPathSplitter>_

```

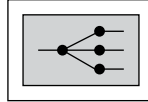
Now on the ESB-side, the first message will be split into three parts whereas the last (third) message will be split into two parts.

INFO - TraceComponent encoding="UTF-8"?><one/>	- Body is: <?xml version="1.0"
INFO - TraceComponent encoding="UTF-8"?><three/>	- Body is: <?xml version="1.0"
INFO - TraceComponent encoding="UTF-8"?><two/>	- Body is: <?xml version="1.0"
INFO - TraceComponent encoding="UTF-8"?><five/>	- Body is: <?xml version="1.0"
INFO - TraceComponent encoding="UTF-8"?><four/>	- Body is: <?xml version="1.0"

Static Recipient List

A recipient list can inspect an incoming message, and depending upon the number of recipients specified in the recipient list, it can forward the message to all channels associated with the recipients in the recipient list.

Notation

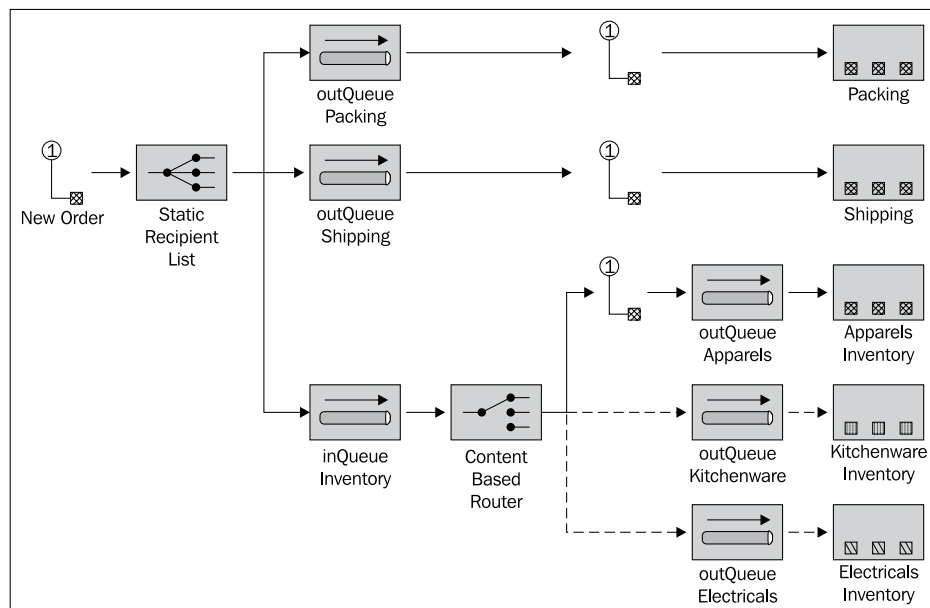


Explanation

The example for a scenario where we need to send the same message to multiple recipients is the email message. With every email message, the sender can specify multiple recipients in the **To**, **Cc**, or **Bcc** fields. Now, the email system can inspect these fields and if more than one recipient is specified in these lists, it will forward the same message to all the recipient addresses specified.

Illustrative Design

Whenever the Acme back-end system receives a confirmed order, the same has to be distributed to both packing systems and shipping systems, along with the inventory systems which we have already discussed. We can configure a static recipient list specifying a shipping queue, a packing queue, and a third queue for a content-based router for inventory systems as shown in the following figure. The advantage is that all these LOB systems will receive the same order so that they can initiate their associated business processes. This is illustrated in the following figure:

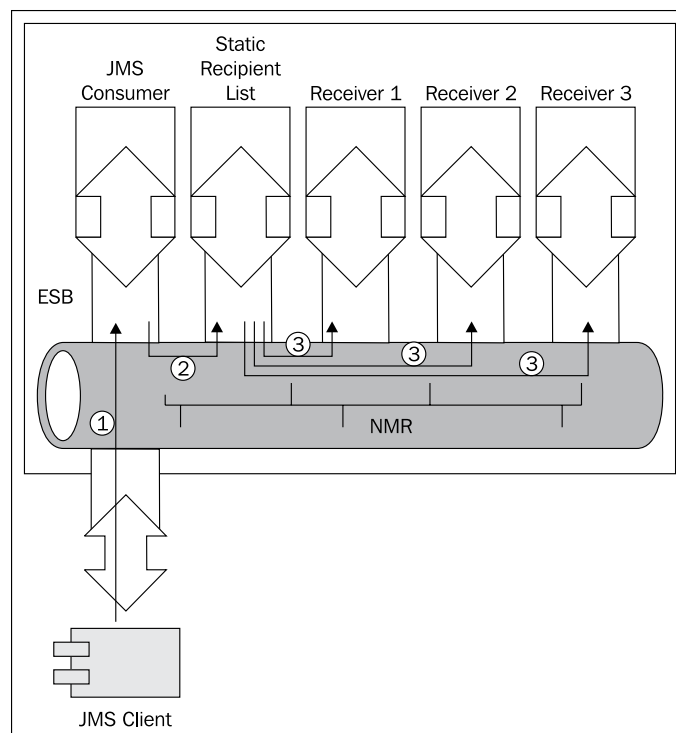


Sample Use Case

The sample use case will have following components:

- **JMS client:** This is a normal external JMS client, placing XML composite messages onto the JMS consumer component configured within the ESB.
- **JMS consumer:** This is a `servicemix-jms` listening on queue "A". Any incoming messages to this queue will be routed to the next component in the flow chain which is the Static Recipient List.
- **Static recipient list:** The Static Recipient List is a `servicemix-eip` component. On receiving in messages, the Static Recipient List will forward the message to all recipients, which are different instances of receiver components, configured in the Static Recipient List.
- **Receiver component:** The receiver component just spits out whatever message it receives into the console.

The following figure illustrates how the various components can be assembled in the JBI bus for our sample use case:



Sample Code and Configuration

We configure the Static Recipient List in `servicemix.xml`, along with the other components described above. The `servicemix.xml` file is shown in the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xbean.org/schemas/spring/1.0"
       xmlns:sm="http://servicemix.apache.org/config/1.0"
       xmlns:test="http://xslt.servicemix.apache.binildas.com"
       xmlns:eip="http://servicemix.apache.org/eip/1.0"
       xmlns:jms="http://servicemix.apache.org/jms/1.0">
  <classpath>
    <location>./</location>
  </classpath>
  <import resource="classpath:jmx.xml" />
  <import resource="classpath:jndi.xml" />
  <import resource="classpath:security.xml" />
  <import resource="classpath:tx.xml" />
  <import resource="classpath:activemq.xml" />
  <bean id="propertyConfigurer"
        class="org.springframework.beans.factory.config.
              PropertyPlaceholderConfigurer">
    <property name="location"
              value="classpath:servicemix.properties" />
  </bean>
  <sm:container name="jbi"
                monitorInstallationDirectory="false"
                createMBeanServer="true"
                useMBeanServer="true">
    <sm:activationSpecs>
      <sm:activationSpec>
        <sm:component>
          <jms:component>
            <jms:endpoints>
              <jms:endpoint service="test:MyConsumerService"
                            endpoint="myConsumer"
                            role="consumer"
                            soap="false"
                            targetService="test:recipients"
                            defaultMep="http://www.w3.org/2004/
                                      08/wsdl/in-only"
                            destinationStyle="queue"
                            jmsProviderDestinationName="queue/A"
                            connectionFactory =
                              "#connectionFactory" />
            </jms:endpoints>
          </jms:component>
        </sm:component>
      </sm:activationSpec>
    </sm:activationSpecs>
  </sm:container>
</beans>
```

```

    </sm:component>
  </sm:activationSpec>
  <sm:activationSpec id="servicemix-eip">
    <sm:component>
      <eip:component>
        <eip:endpoints>
          <eip:static-recipient-list
            service="test:recipients"
            endpoint="endpoint">
            <eip:recipients>
              <eip:exchange-target
                service="test:receiver1" />
              <eip:exchange-target
                service="test:receiver2" />
              <eip:exchange-target
                service="test:receiver3" />
            </eip:recipients>
          </eip:static-recipient-list>
        </eip:endpoints>
      </eip:component>
    </sm:component>
  </sm:activationSpec>
  <sm:activationSpec componentName="receiver1"
    service="test:receiver1">
    <sm:component>
      <bean xmlns="http://xbean.org/schemas/spring/1.0"
        class="MyReceiver" >
        <property name="name">
          <value>1</value>
        </property>
      </bean>
    </sm:component>
  </sm:activationSpec>
  <sm:activationSpec componentName="receiver2"
    service="test:receiver2">
    <sm:component>
      <bean xmlns="http://xbean.org/schemas/spring/1.0"
        class="MyReceiver" >
        <property name="name">
          <value>2</value>
        </property>
      </bean>
    </sm:component>
  </sm:activationSpec>

```

```
<sm:activationSpec componentName="receiver3"
    service="test:receiver3">
  <sm:component>
    <bean xmlns="http://xbean.org/schemas/spring/1.0"
      class="MyReceiver" >
      <property name="name">
        <value>3</value>
      </property>
    </bean>
  </sm:component>
</sm:activationSpec>
</sm:activationSpecs>
</sm:container>
<bean id="connectionFactory"
  class="org.apache.activemq.ActiveMQConnectionFactory">
  <property name="brokerURL" value="tcp://localhost:61616" />
</bean>
</beans>
```

We have configured three recipients namely `test:receiver1`, `test:receiver2`, and `test:receiver3` in the static recipient list above.

Deploy and Run the Sample

To build the sample, change directory to `ch15\04_StaticRecipientList` and type `ant` as shown here:

```
cd ch15\04_StaticRecipientList
ant
```

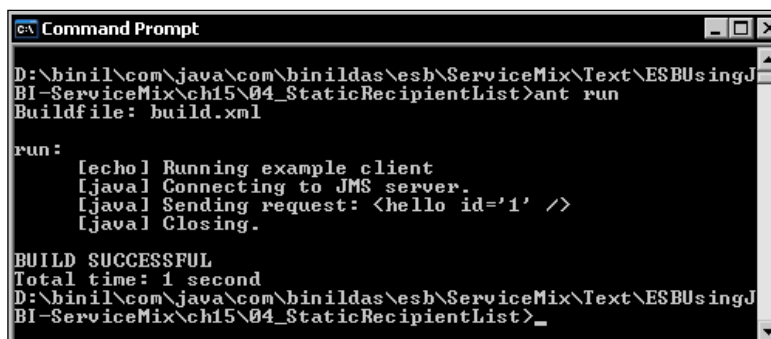
This will compile all the files, including the JMS client program. Now to test the sample, first, bring ServiceMix up by executing the following commands:

```
cd ch15\04_StaticRecipientList
%SERVICEMIX_HOME%\bin\servicemix servicemix.xml
```

Now in a different command prompt execute `ant run`.

```
cd ch15\04_StaticRecipientList
ant run
```

The JMS client program console will print out the messages it sends to the ESB as shown in the following screenshot:



```

C:\ Command Prompt

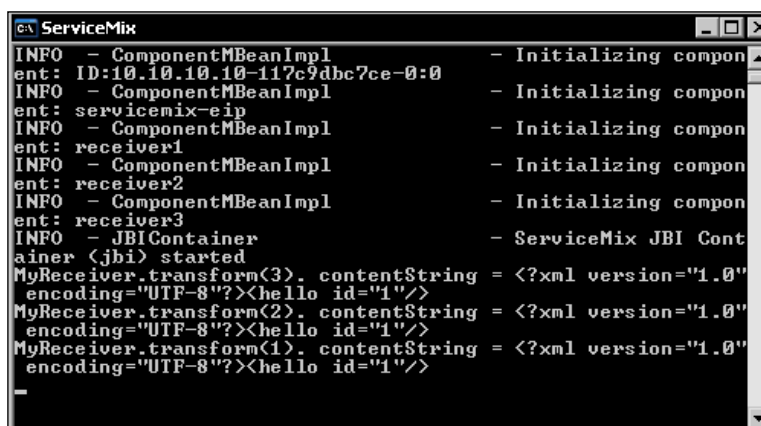
D:\bin\com\java\com\bin\ildas\esh\ServiceMix\Text\ESBUsingJ
BI-ServiceMix\ch15\04_StaticRecipientList>ant run
Buildfile: build.xml

run:
    [echo] Running example client
    [java] Connecting to JMS server.
    [java] Sending request: <hello id='1' />
    [java] Closing.

BUILD SUCCESSFUL
Total time: 1 second
D:\bin\com\java\com\bin\ildas\esh\ServiceMix\Text\ESBUsingJ
BI-ServiceMix\ch15\04_StaticRecipientList>_

```

At the same time, if you observe the ESB console, you can see the message is delivered to all the three recipients in the recipient list.



```

C:\ ServiceMix

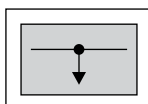
INFO - ComponentMBeanImpl - Initializing compon
ent: ID:10.10.10.10-117c9dbc7ce-0:0
INFO - ComponentMBeanImpl - Initializing compon
ent: servicemix-eip
INFO - ComponentMBeanImpl - Initializing compon
ent: receiver1
INFO - ComponentMBeanImpl - Initializing compon
ent: receiver2
INFO - ComponentMBeanImpl - Initializing compon
ent: receiver3
INFO - JBIContainer - ServiceMix JBI Cont
ainer (jbi) started
MyReceiver.transform(3). contentString = <?xml version="1.0"
encoding="UTF-8"?><hello id="1"/>
MyReceiver.transform(2). contentString = <?xml version="1.0"
encoding="UTF-8"?><hello id="1"/>
MyReceiver.transform(1). contentString = <?xml version="1.0"
encoding="UTF-8"?><hello id="1"/>
-

```

Wiretap

The wiretap is a kind of simple recipient list which, when inserted into the message channel, will publish each incoming message into the main channel as well as into a secondary channel.

Notation

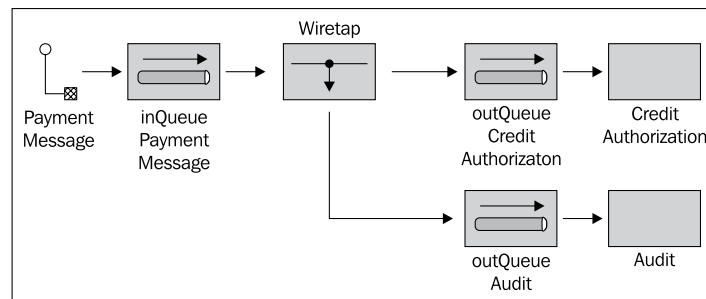


Explanation

The wiretap is a static recipient list, with only two recipients in the list. When inserted into the main channel, it publishes the input message into the main channel as well as the secondary channel configured in the wiretap. The wiretap will not modify the message contents in any manner.

Illustrative Design

Suppose you want to do an audit for every credit card payment that you make through your Acme online store. Your credit card information will reach the back-end system and there we can use a wiretap. Hence this message is also published into a secondary audit module which will write all required information to a non-erasable disk.



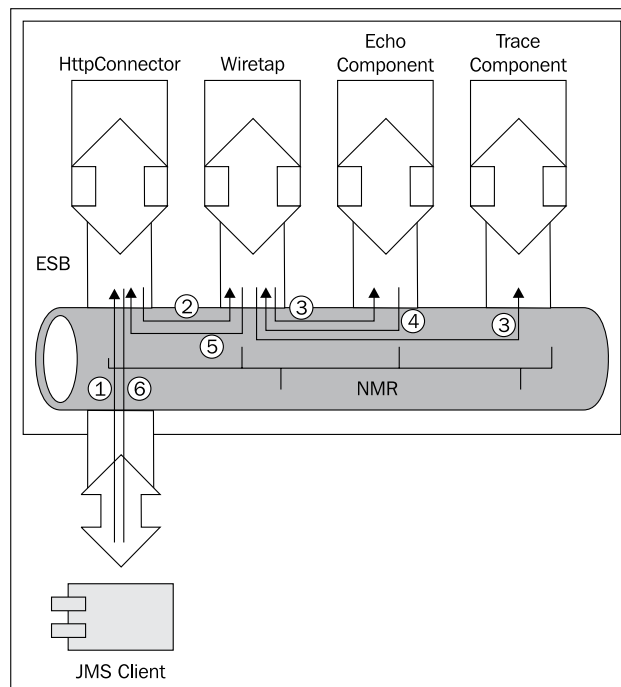
Sample Use Case

The sample use case will have the following components:

- **HTTP client:** This is an external HTTP client, placing the XML messages onto the HTTP connector component configured within the ESB.
- **HTTP connector:** This is a ServiceMix HTTP component listening on port 8912. Any incoming messages to this queue will be routed to the next component in the flow chain which is the wiretap.
- **Wiretap:** The wiretap is a `servicemix-eip` component. On receiving in messages, the wiretap will forward the message to the main target (an Echo component in our case) as well as to the listener (a Trace component here). Wiretap can handle all four standard MEPs, but can only send an In-Only MEP to the listener.
- **Echo component:** The echo component is the main target of the wiretap and will take part in an In-Out MEP in the flow. Hence the request is echoed back to the wiretap and then to the consumer component of the wiretap (HTTP connector).

- **Trace component:** The Trace component takes part in an In-Only MEP in the flow and will print the message in the console.

The following figure illustrates how the various components can be assembled in the JBI bus for our sample use case:



Sample Code and Configuration

We configure the wiretap in the `servicemix.xml` file, along with other components described above. This file is reproduced in the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xbean.org/schemas/spring/1.0"
       xmlns:sm="http://servicemix.apache.org/config/1.0"
       xmlns:test="http://xslt.servicemix.apache.org/binildas.com"
       xmlns:eip="http://servicemix.apache.org/eip/1.0"
       xmlns:jms="http://servicemix.apache.org/jms/1.0">
  <classpath>
    <location>./</location>
  </classpath>
  <import resource="classpath:jmx.xml" />
  <import resource="classpath:jndi.xml" />
  <import resource="classpath:security.xml" />
```



```
<import resource="classpath:tx.xml" />
<import resource="classpath:activemq.xml" />
<bean id="propertyConfigurer"
      class="org.springframework.beans.factory.
            config.PropertyPlaceholderConfigurer">
  <property name="location"
            value="classpath:servicemix.properties" />
</bean>
<sm:container name="jbi"
              monitorInstallationDirectory="false"
              createMBeanServer="true"
              useMBeanServer="true">
  <sm:activationSpecs>
    <sm:activationSpec componentName="httpReceiver"
                      service="test:httpBinding"
                      endpoint="httpReceiver"
                      destinationService="test:wireTap">
      <sm:component>
        <bean class="org.apache.servicemix.
                  components.http.HttpConnector">
          <property name="host" value="localhost"/>
          <property name="port" value="8912"/>
        </bean>
      </sm:component>
    </sm:activationSpec>
    <sm:activationSpec componentName="echo" service="test:echo">
      <sm:component>
        <bean class="org.apache.servicemix.
                  components.util.EchoComponent" />
      </sm:component>
    </sm:activationSpec>
    <sm:activationSpec componentName="trace" service="test:trace">
      <sm:component>
        <bean class="org.apache.servicemix.components.
                  util.TraceComponent" />
      </sm:component>
    </sm:activationSpec>
    <sm:activationSpec id="servicemix-eip">
      <sm:component>
        <eip:component>
          <eip:endpoints>
            <eip:wire-tap service="test:wireTap"
                          endpoint="wireTapEndpoint">
              <eip:target>
                <eip:exchange-target service="test:echo" />
              </eip:target>
            </eip:wire-tap>
          </eip:endpoints>
        </eip:component>
      </sm:component>
    </sm:activationSpec>
  </sm:activationSpecs>
</sm:container>
```

```

        </eip:target>
        <eip:inListener>
            <eip:exchange-target service="test:trace" />
        </eip:inListener>
    </eip:wire-tap>
</eip:endpoints>
</eip:component>
</sm:component>
</sm:activationSpec>
</sm:activationSpecs>
</sm:container>
</beans>

```

We have configured both `test:echo` and `test:trace` as the listeners for the message exchange in the wiretap. Hence the message will be forwarded to both these components.

Deploy and Run the Sample

To build the sample, change directory to `ch15\05_WireTap` and type `ant`:

```

cd ch15\05_WireTap
ant

```

This will compile all the files, including the HTTP client program. Now to test the sample, first bring ServiceMix up by executing:

```

cd ch15\05_WireTap
%SERVICEMIX_HOME%\bin\servicemix servicemix.xml

```

Now in a different command prompt execute `ant run`.

```

cd ch15\05_WireTap
ant run

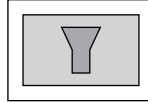
```

The HTTP client program will send the test XML message to the HTTP connector configured in the ESB. In the ESB console we can validate that the message is delivered to both the `test:echo` and `test:trace` services. The HTTP client also prints out the response received back from the ESB.

Message Filter

Message filter can eliminate the unwanted messages from a set of messages published.

Notation

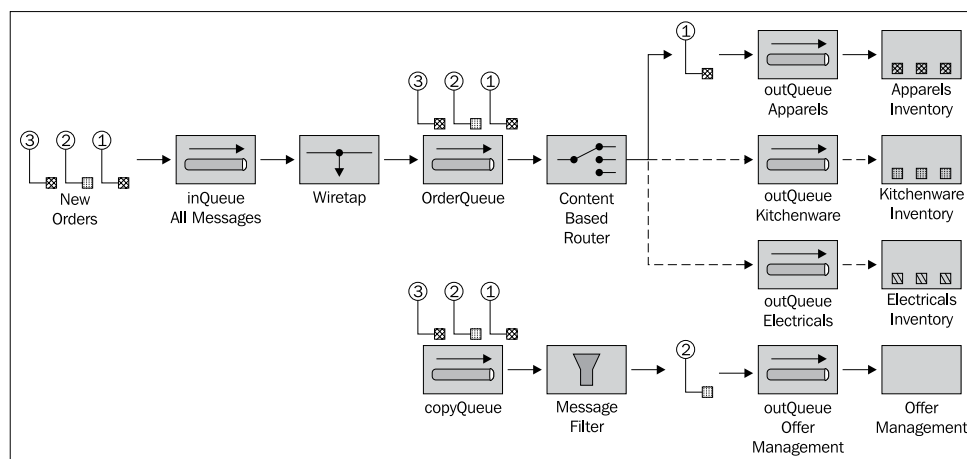


Explanation

Messages are of different kinds, and the same is true with events also. There are unwanted events and events which are to be tracked and reacted based on event type in particular fashions. Whether a message is of interest or not is again based on the message content. Hence the messages which match the filter rule of interest need to be routed further whereas the messages which didn't match the filter criterion are ignored. All the messages which match the criterion will be routed to the output channel of the message filter.

Illustrative Design

Acme has decided to offer discounts and gifts for all purchases exceeding a certain amount in a single checkout. To process this, we can first use a wiretap to send a copy of all orders to a message filter. The message filter will inspect the message for the total dollar value of checkout and if it exceeds the threshold, the message will be forwarded to the offer management module. Any order messages with total dollar value less than the threshold will be ignored (dropped) at the message filter, not forwarding them to the offer management module. Note that, we are directing copies of orders only to the message filter hence all the original messages will be routed to their respective inventory modules, irrespective of whether their copies are forwarded or dropped at the message filter. This is illustrated in the following figure:

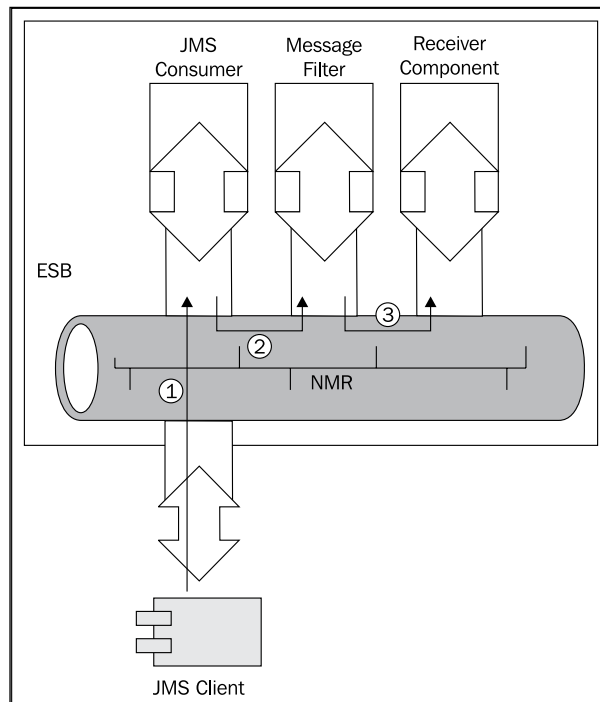


Sample Use Case

The sample use case will have following components:

- **JMS client:** This is a normal external JMS client, placing XML messages onto the JMS consumer component configured within the ESB.
- **JMS consumer:** This is a `servicemix-jms` listening on queue "A". Any incoming messages to this queue will be routed to the next component in the flow chain which is the Message Filter.
- **Message filter:** The Message Filter is a `servicemix-eip` component. On receiving the in messages, the filter will try to match the filter rule configured at the filter-level with the message content. In the sample, we use the XPath predicate to define the filter rule. In finding a match, the filter will forward the message to the exchange-target which is a receiver component. Any message which doesn't match will be dropped at the filter-level.
- **Receiver component:** The receiver component just spits out whatever message it receives into the console.

The following figure illustrates how the various components can be assembled in the JBI bus for our sample use case:



Sample Code and Configuration

We configure the Message Filter in the servicemix.xml file, along with other components described above. This is shown in the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xbean.org/schemas/spring/1.0"
       xmlns:sm="http://servicemix.apache.org/config/1.0"
       xmlns:test="http://xslt.servicemix.apache.binildas.com"
       xmlns:eip="http://servicemix.apache.org/eip/1.0"
       xmlns:jms="http://servicemix.apache.org/jms/1.0">
  <classpath>
    <location>.</location>
  </classpath>
  <import resource="classpath:jmx.xml" />
  <import resource="classpath:jndi.xml" />
  <import resource="classpath:security.xml" />
  <import resource="classpath:tx.xml" />
  <import resource="classpath:activemq.xml" />
  <bean id="propertyConfigurer"
        class="org.springframework.beans.factory.config.
              PropertyPlaceholderConfigurer">
    <property name="location"
              value="classpath:servicemix.properties" />
  </bean>
  <sm:container name="jbi"
                monitorInstallationDirectory="false"
                createMBeanServer="true"
                useMBeanServer="true">
    <sm:activationSpecs>
      <sm:activationSpec>
        <sm:component>
          <jms:component>
            <jms:endpoints>
              <jms:endpoint service="test:MyConsumerService"
                            endpoint="myConsumer"
                            role="consumer"
                            soap="false"
                            targetService="test:messageFilter"
                            defaultMep="http://www.w3.org/2004/
                                      08/wsdl/in-only"
                            destinationStyle="queue"
                            jmsProviderDestinationName="queue/A"
                            connectionFactory =
                              "#connectionFactory" />
            </jms:endpoints>
          </jms:component>
        </sm:component>
      </sm:activationSpec>
    </sm:activationSpecs>
  </sm:container>
</beans>
```

```

        </sm:component>
    </sm:activationSpec>
    <sm:activationSpec componentName="receiver"
        service="test:receiver">
        <sm:component>
            <bean xmlns="http://xbean.org/schemas/spring/1.0"
                class="MyReceiver" >
                <property name="name">
                    <value>1</value>
                </property>
            </bean>
        </sm:component>
    </sm:activationSpec>
    <sm:activationSpec id="servicemix-eip">
        <sm:component>
            <eip:component>
                <eip:endpoints>
                    <eip:message-filter service="test:messageFilter"
                        endpoint="messageFilterEndpoint">
                        <eip:target>
                            <eip:exchange-target
                                service="test:receiver" />
                        </eip:target>
                        <eip:filter>
                            <eip:xpath-predicate
                                xpath="/hello/@id = '1'"/>
                            </eip:filter>
                        </eip:message-filter>
                    </eip:endpoints>
                </eip:component>
            </sm:component>
        </sm:activationSpec>
    </sm:activationSpecs>
</sm:container>
<bean id="connectionFactory"
    class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL" value="tcp://localhost:61616" />
</bean>
</beans>

```

The filter rule we are specifying here is the XPath `/hello/@id = '1'`. The code for `MyReceiver.java` class is very similar to the `MyReceiver` used in the content-based router sample. The only difference is that it won't write any content back to the out. Hence we are not repeating the code here.

Deploy and Run the Sample

To build the sample, change directory to `ch15\06_MessageFilter` and type `ant` as shown here:

```
cd ch15\06_MessageFilter
ant
```

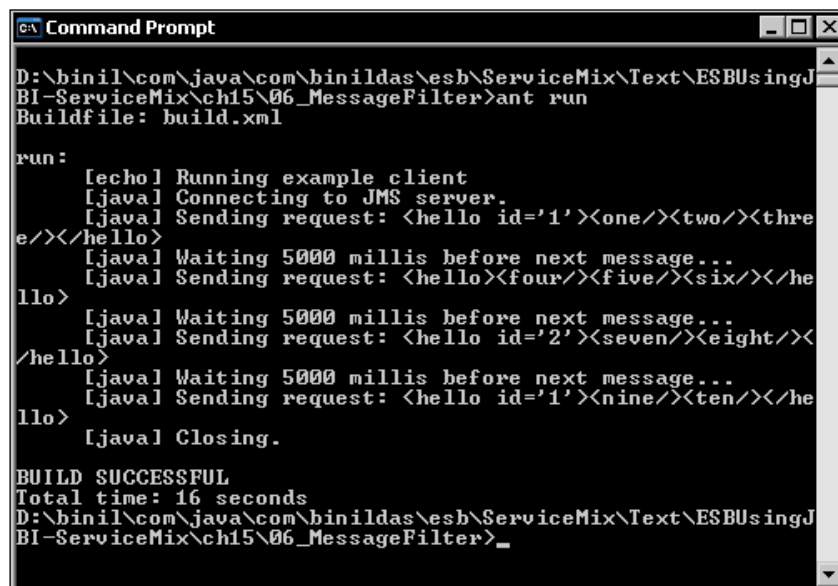
This will compile all the files, including the JMS client program. Now to test the sample, first, bring ServiceMix up by executing:

```
cd ch15\06_MessageFilter
%SERVICEMIX_HOME%\bin\servicemix servicemix.xml
```

Now in a different command prompt execute `ant run`.

```
cd ch15\06_MessageFilter
ant run
```

The JMS client program console will print out the messages it sends to the ESB as shown in the following figure:

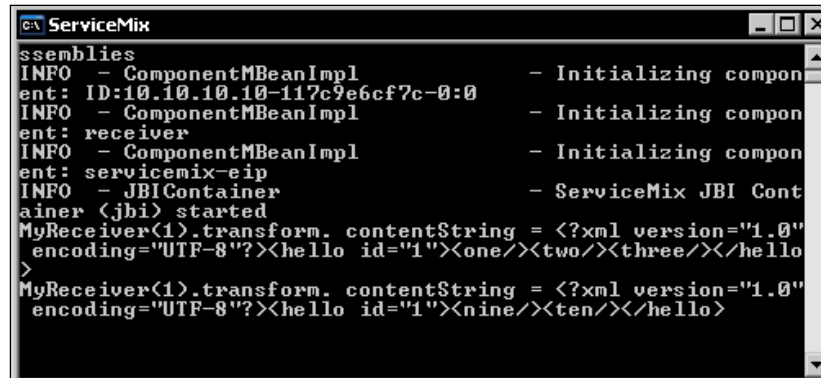


```
Command Prompt
D:\binil\com\java\com\binildas\esb\ServiceMix\Text\ESBUsingJBI-ServiceMix\ch15\06_MessageFilter>ant run
Buildfile: build.xml

run:
[echo] Running example client
[java] Connecting to JMS server.
[java] Sending request: <hello id='1'><one/><two/><three/></hello>
[java] Waiting 5000 millis before next message...
[java] Sending request: <hello><four/><five/><six/></hello>
[java] Waiting 5000 millis before next message...
[java] Sending request: <hello id='2'><seven/><eight/></hello>
[java] Waiting 5000 millis before next message...
[java] Sending request: <hello id='1'><nine/><ten/></hello>
[java] Closing.

BUILD SUCCESSFUL
Total time: 16 seconds
D:\binil\com\java\com\binildas\esb\ServiceMix\Text\ESBUsingJBI-ServiceMix\ch15\06_MessageFilter>
```

If we look at the ESB console, we can see that only two of the messages are filtered and forwarded to the receiver component and the other two messages are dropped.



```

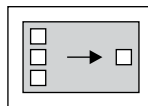
ServiceMix
ssemblies
INFO - ComponentMBeanImpl - Initializing compon
ent: ID:10.10.10.10-117c9e6cf7c-0:0
INFO - ComponentMBeanImpl - Initializing compon
ent: receiver
INFO - ComponentMBeanImpl - Initializing compon
ent: servicemix-eip
INFO - JBIContainer - ServiceMix JBI Cont
ainer <jbi> started
MyReceiver(1).transform. contentString = <?xml version="1.0"
encoding="UTF-8"?><hello id="1"><one/><two/><three/></hello
>
MyReceiver(1).transform. contentString = <?xml version="1.0"
encoding="UTF-8"?><hello id="1"><nine/><ten/></hello>

```

Split Aggregator

An aggregator is a kind of stateful filter which can store message parts which are correlated by some form of ID or field. When all the parts are ready it can aggregate all the parts and publish a single aggregate message.

Notation

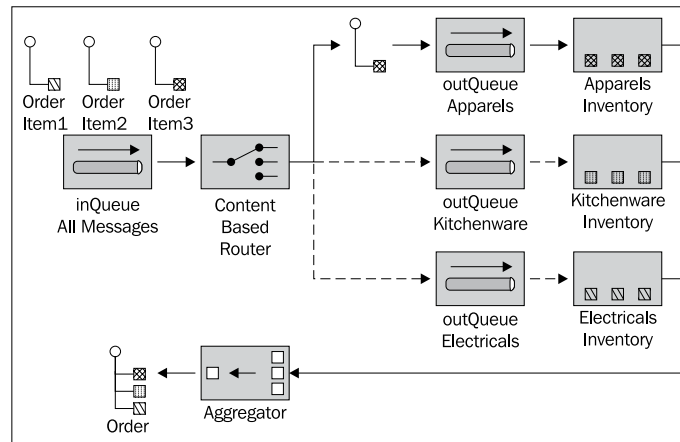


Explanation

In many cases message parts which are co-related arrive at different times and we cannot proceed further processing unless we receive the message in full. The scenario is common in the airline domain where the messaging channel splits and sends passenger reservation lists and passenger name lists in parts, due to size limitations imposed by EDI messaging gateways. The processing module has to wait until it receives all the parts of a co-related message. To help the correct ordering of messages back and to identify whether we have received all parts of the message, the aggregator depends on several properties such as count, index, and correlation ID. Each intermittent result is stored by the aggregator until the message is fully aggregated. Hence the aggregator is stateful in nature.

Illustrative Design

Every full order in the Acme back-end system has to be validated with an available stock before we can confirm the order. We already discussed that we split the order into order items and send individual messages corresponding to each item to different inventory modules. Now, before we can return back the order validation message to the customer we need to examine the validation status of each order item from the inventory modules. One possible configuration is to use an aggregator to which all the inventory modules can route the individual order item status messages. The aggregator, when it receives the status messages from all the related order items, can evaluate the overall status and send back the decision (confirmed order or error). This is shown in the following figure:

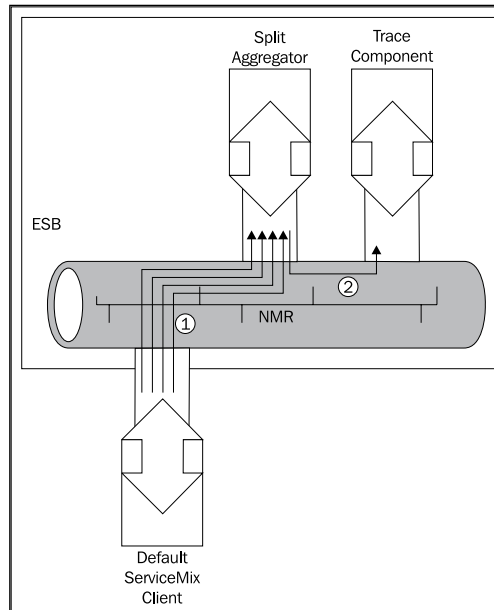


Sample Use Case

The sample use case will have the following components:

- **Default ServiceMix client:** The default ServiceMix client will send several In-Only messages to the split aggregator configured in the ESB. Each of these messages is correlated, and they also carry the `splitCount` and `splitIndex` along with their message properties.
- **Split aggregator:** The split aggregator is a `servicemix-eip` component. On receiving the in messages, the aggregator will look at the `correlation ID`, `splitCount`, and `splitIndex` and can rearrange and aggregate the messages irrespective of the order in which the messages arrive. When every part of a message has arrived, the fully aggregated message is forwarded to the next component in the flow chain which is the trace component.
- **Trace component:** The trace component prints out the aggregated messages received from the split aggregator to the ESB console.

The following figure illustrates how the various components can be assembled in the JBI bus for our sample use case:



Sample Code and Configuration

We configure the Split Aggregator in the `servicemix.xml` file, along with other components described above.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xbean.org/schemas/spring/1.0"
       xmlns:sm="http://servicemix.apache.org/config/1.0"
       xmlns:test="http://xslt.servicemix.apache.org/binildas.com"
       xmlns:eip="http://servicemix.apache.org/eip/1.0"
       xmlns:jms="http://servicemix.apache.org/jms/1.0">
  <classpath>
    <location>./</location>
  </classpath>
  <import resource="classpath:jmx.xml" />
  <import resource="classpath:jndi.xml" />
  <import resource="classpath:security.xml" />
  <import resource="classpath:tx.xml" />
  <import resource="classpath:activemq.xml" />
  <bean id="propertyConfigurer"
        class="org.springframework.beans.factory.config.
              PropertyPlaceholderConfigurer">
    <property name="location"
              value="classpath:servicemix.properties" />
  </bean>
</beans>
```

```
</bean>
<sm:container id="jbi"
    monitorInstallationDirectory="false"
    createMBeanServer="false"
    useMBeanServer="false">
    <sm:activationSpecs>
        <sm:activationSpec id="servicemix-eip">
            <sm:component>
                <eip:component>
                    <eip:endpoints>
                        <eip:split-aggregator service="test:aggregator"
                            endpoint="aggregatorEndpoint"
                            aggregateElementName=
                                "test:MessageEnvelope"
                            messageElementName=
                                "test:MessagePart">

                            <eip:target>
                                <eip:exchange-target service="test:trace" />
                            </eip:target>
                        </eip:split-aggregator>
                    </eip:endpoints>
                </eip:component>
            </sm:component>
        </sm:activationSpec>
        <sm:activationSpec componentName="trace" service="test:trace">
            <sm:component>
                <bean class="org.apache.servicemix.components.
                    util.TraceComponent" />
            </sm:component>
        </sm:activationSpec>
    </sm:activationSpecs>
</sm:container>
<bean id="client" class="org.apache.servicemix.client.
    DefaultServiceMixClient">
    <constructor-arg ref="jbi"/>
</bean>
</beans>
```

The split aggregator is configured with the elements `aggregateElementName` and `messageElementName`. These two elements will decide the message envelopes to be used when the split aggregator aggregates the message.

Deploy and Run the Sample

To build the sample, change directory to `ch15\07_SplitAggregator` and type `ant` as shown here:

```
cd ch15\07_SplitAggregator
ant
```

This will compile all the files. Now to test the sample, just execute the run target of build.xml as shown as follows:

```
cd ch15\07_SplitAggregator
ant run
```

Observe the ESB console; you can see the messages being sent out to the aggregator and finally the aggregator combines all related messages and forwards them to the trace component which will print out the message to the console. This is shown as follows:

```
Send msg : corrId<1178104210727> : splitterCount<3> : splitterIndex<1>
: msg<<hello id="1"><binil/><sowmya/></hello>>
Waiting 5000 millis before next message...
Send msg : corrId<1178104215726> : splitterCount<2> : splitterIndex<1>
: msg<<hello id="1"><ann/></hello>>
Waiting 5000 millis before next message...
Send msg : corrId<1178104210727> : splitterCount<3> : splitterIndex<0>
: msg<<hello id="0"><binil/><sowmya/></hello>>
Waiting 5000 millis before next message...
Send msg : corrId<1178104215726> : splitterCount<2> : splitterIndex<0>
: msg<<hello id="0"><ann/></hello>>
Waiting 5000 millis before next message...

INFO - TraceComponent - Exchange: InOnly[
  id: ID:10.10.10.10-1124c7badc9-2:0
  status: Active
  role: provider
  service: {http://xslt.servicemix.apache.binildas.com}trace
  endpoint: trace

  in: <?xml version="1.0" encoding="UTF-8"?><test:MessageEnvelope
xmlns:test="http://xslt.servicemix.apache.binildas.com"
count="2"><test:MessagePart index="0"><hello id="0"><ann/></hello></
test:MessagePart><test:MessagePart index="1"><hello id="1"><ann/></
hello></test:MessagePart></test:MessageEnvelope>
] received IN message: org.apache.servicemix.jbi.messaging.Normalized
MessageImpl@199197b{properties: {org.apache.servicemix.eip.splitter.
corrid=1178104215726}}
INFO - TraceComponent - Body is: <?xml version="1.0"
encoding="UTF-8"?><test:MessageEnvelope xmlns:test="http://xslt.
servicemix.apache.binildas.com" count="2"><test:MessagePart
index="0"><hello id="0"><ann/></hello></test:MessagePart><test:
MessagePart index="1"><hello id="1"><ann/></hello></test:
MessagePart></test:MessageEnvelope>

Send msg : corrId<1178104210727> : splitterCount<3> : splitterIndex<2>
: msg<<hello id="2"><binil/><sowmya/></hello>>
```

```
Waiting 5000 millis before next message...

INFO - TraceComponent - Exchange: InOnly[
  id: ID:10.10.10.10-1124c7badc9-2:1
  status: Active
  role: provider
  service: {http://xslt.servicemix.apache.binildas.com}trace
  endpoint: trace

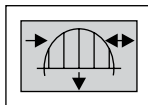
  in: <?xml version="1.0" encoding="UTF-8"?><test:MessageEnvelope
xmlns:test="http://xslt.servicemix.apache.binildas.com"
count="3"><test:MessagePart index="0"><hello id="0"><binil/
><sowmya/></hello></test:MessagePart><test:MessagePart
index="1"><hello id="1"><binil/><sowmya/></hello></test:
MessagePart><test:MessagePart index="2"><hello id="2"><binil/
><sowmya/></hello></test:MessagePart></test:MessageEnvelope>
] received IN message: org.apache.servicemix.jbi.messaging.Normalized
MessageImpl@195ff24{properties: {org.apache.servicemix.eip.splitter.
corrid=1178104210727}}

INFO - TraceComponent - Body is: <?xml
version="1.0" encoding="UTF-8"?><test:MessageEnvelope xmlns:
test="http://xslt.servicemix.apache.binildas.com" count="3"><test:
MessagePart index="0"><hello id="0"><binil/><sowmya/></hello></
test:MessagePart><test:MessagePart index="1"><hello id="1"><binil/
><sowmya/></hello></test:MessagePart><test:MessagePart
index="2"><hello id="2"><binil/><sowmya/></hello></test:MessagePart></
test:MessageEnvelope>
```

Pipeline

A Pipeline is a kind of bridge which can transform one form of MEP to another.

Notation



For a pipeline, there is no notation provided in the EAI patterns collection. Hence the bridge EAI pattern is extended and shown here. The motivation behind doing so is that it is possible to build a bridge between two message exchange patterns using a pipeline.

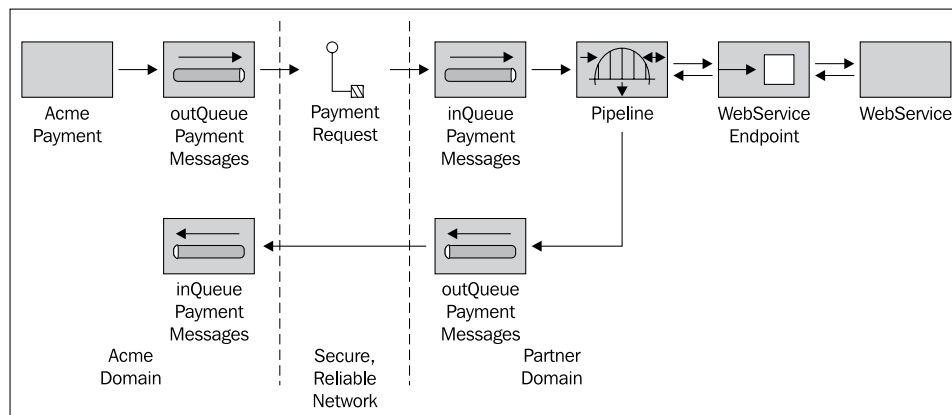
Explanation

A pipeline can be configured to transform an In-Only MEP to an In-Out. That is, when the pipeline receives an In-Only MEP, it sends the message in an In-Out MEP to the transformer component. Hence, the transformer will send a response as an out message for the In-Out. This out message is then forwarded to the pipeline target in another In-Only MEP.

Using the MEP transformation property of pipeline, it is possible to construct a protocol bridge. For example, we can bridge between a HTTP protocol (In-Out) and a JMS (In-Only) protocol. This is what we have done in Chapter 11 (*Access Web Services using JMS Channel*).

Illustrative Design

The Acme e-commerce system allows customers online to make payments and each such payment message is routed to the financial institution's system in a secure and reliable manner. The financial institution is Acme's payment partner and their systems are separated in a different network and domain. Sending requests through Internet is a feasible method, but the usual HTTP channel does not have the required reliability. Hence there is a dedicated network between Acme and the financial institution. Moreover, to make messaging reliable it is possible to send messages using a reliable channel through MOM. Using JMS, we can send messages through MOM. However, the payment service at the financial institution's end is a HTTP service with a request-reply style. Hence the In-Only MEP used in JMS doesn't fit well. A pipeline can solve this problem by providing a bridge between the two protocols—JMS and HTTP. This design is shown in the following figure:



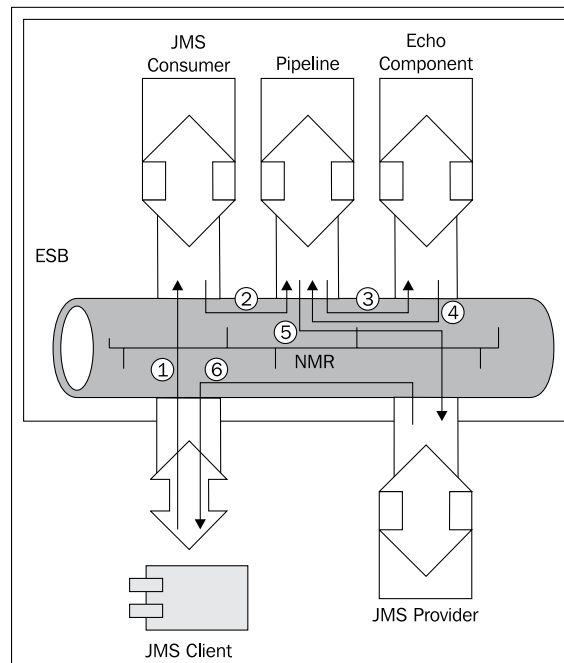
You may also refer the Chapter 11 (*Access Web Service using JMS Channel*) to see an implementation for this.

Sample Use Case

The sample use case will have the following components:

- **JMS client:** This is a normal external JMS client, placing XML messages onto the JMS consumer component configured within the ESB.
- **JMS consumer:** This is a `servicemix-jms` listening on queue "A". Any incoming messages to this queue will be routed to the next component in the flow chain which is the pipeline.
- **Pipeline:** The Pipeline is a `servicemix-eip` component. The pipeline receives an In-Only MEP from JMS consumer. The pipeline then sends the same message in an In-Out MEP to the echo component. The echoed response is send back to the pipeline which is then forwarded to the JMS consumer in another In-Only MEP.
- **Echo component:** The echo component echoes back any messages it receives. The message received from the pipeline is thus sent back to the pipeline.
- **JMS provider:** This is a `servicemix-jms` listening on queue "B". The pipeline will place the echoed back message into this queue in an In-Only MEP from where the JMS client can pick up.

The following figure illustrates how the various components can be assembled in the JBI bus for our sample use case:



Sample Code and Configuration

We configure the pipeline in the `servicemix.xml` file, along with other components described above. The content of this file is shown here:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xbean.org/schemas/spring/1.0"
       xmlns:sm="http://servicemix.apache.org/config/1.0"
       xmlns:test="http://xslt.servicemix.apache.binildas.com"
       xmlns:eip="http://servicemix.apache.org/eip/1.0"
       xmlns:jms="http://servicemix.apache.org/jms/1.0">
  <classpath>
    <location>.</location>
  </classpath>
  <import resource="classpath:jmx.xml" />
  <import resource="classpath:jndi.xml" />
  <import resource="classpath:security.xml" />
  <import resource="classpath:tx.xml" />
  <import resource="classpath:activemq.xml" />
  <bean id="propertyConfigurer"
        class="org.springframework.beans.factory.config.
              PropertyPlaceholderConfigurer">
    <property name="location"
              value="classpath:servicemix.properties" />
  </bean>
  <sm:container name="jbi"
                monitorInstallationDirectory="false"
                createMBeanServer="true"
                useMBeanServer="true">
    <sm:activationSpecs>
      <sm:activationSpec>
        <sm:component>
          <jms:component>
            <jms:endpoints>
              <jms:endpoint service="test:MyConsumerService"
                            endpoint="myConsumer"
                            role="consumer"
                            soap="false"
                            targetService="test:pipeline"
                            defaultMep="http://www.w3.org/2004/
                                      08/wsdl/in-only"
                            destinationStyle="queue"
                            jmsProviderDestinationName="queue/A"
                            connectionFactory=
                              "#connectionFactory" />
            
```



```
        <jms:endpoint service="test:MyProviderService"
                    endpoint="myProvider"
                    role="provider"
                    soap="false"
                    destinationStyle="queue"
                    jmsProviderDestinationName="queue/B"
                    connectionFactory=
                        "#connectionFactory" />
    </jms:endpoints>
</jms:component>
</sm:component>
</sm:activationSpec>
<sm:activationSpec componentName="echo" service="test:echo">
    <sm:component>
        <bean class="org.apache.servicemix.components.
                util.EchoComponent" />
    </sm:component>
</sm:activationSpec>
<sm:activationSpec id="servicemix-eip">
    <sm:component>
        <eip:component>
            <eip:endpoints>
                <eip:pipeline service="test:pipeline"
                            endpoint="pipelineEndpoint">
                    <eip:transformer>
                        <eip:exchange-target service="test:echo" />
                    </eip:transformer>
                    <eip:target>
                        <eip:exchange-target
                            service="test:MyProviderService" />
                    </eip:target>
                </eip:pipeline>
            </eip:endpoints>
        </eip:component>
    </sm:component>
</sm:activationSpec>
</sm:activationSpecs>
</sm:container>
<bean id="connectionFactory"
      class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL" value="tcp://localhost:61616" />
</bean>
</beans>
```

Deploy and Run the Sample

To build the sample, change directory to `ch15\08_Pipeline` and type `ant` as shown here:

```
cd ch15\08_Pipeline
ant
```

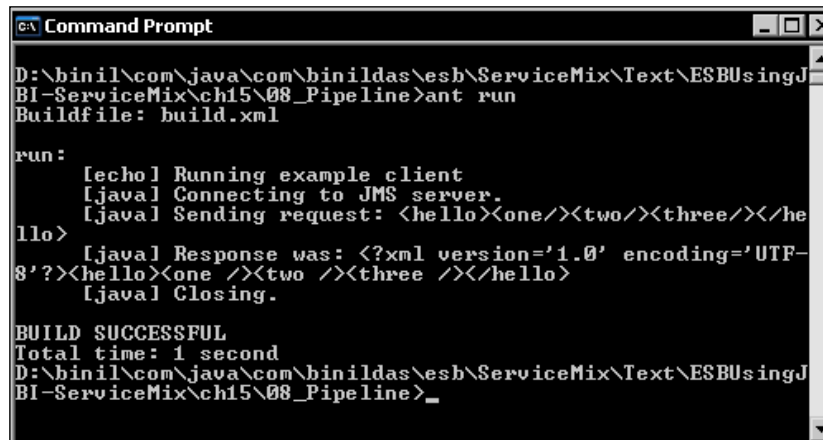
This will compile all the files, including the JMS client program. Now to test the sample, first bring ServiceMix up by executing:

```
cd ch15\08_Pipeline
%SERVICEMIX_HOME%\bin\servicemix servicemix.xml
```

Now in a different command prompt execute `ant run` as shown here:

```
cd ch15\08_Pipeline
ant run
```

The output for the above command is shown in the following screenshot:



```

D:\binil\com\java\com\binildas\esb\ServiceMix\Text\ESBUsingJ
BI-ServiceMix\ch15\08_Pipeline>ant run
Buildfile: build.xml

run:
    [echo] Running example client
    [java] Connecting to JMS server.
    [java] Sending request: <hello><one /><two /><three /></he
llo>
    [java] Response was: <?xml version='1.0' encoding='UTF-
8'?'><hello><one /><two /><three /></hello>
    [java] Closing.

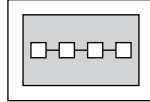
BUILD SUCCESSFUL
Total time: 1 second
D:\binil\com\java\com\binildas\esb\ServiceMix\Text\ESBUsingJ
BI-ServiceMix\ch15\08_Pipeline>_
  
```

As seen in the figure, the request and response message will pass through the pipeline which will bridge the two In-Only MEP exchanges (request and response) to an In-Out MEP exchange (Echo component).

Static Routing Slip

A static routing slip can route a message coming in an In-Out MEP through a series of configured target services.

Notation

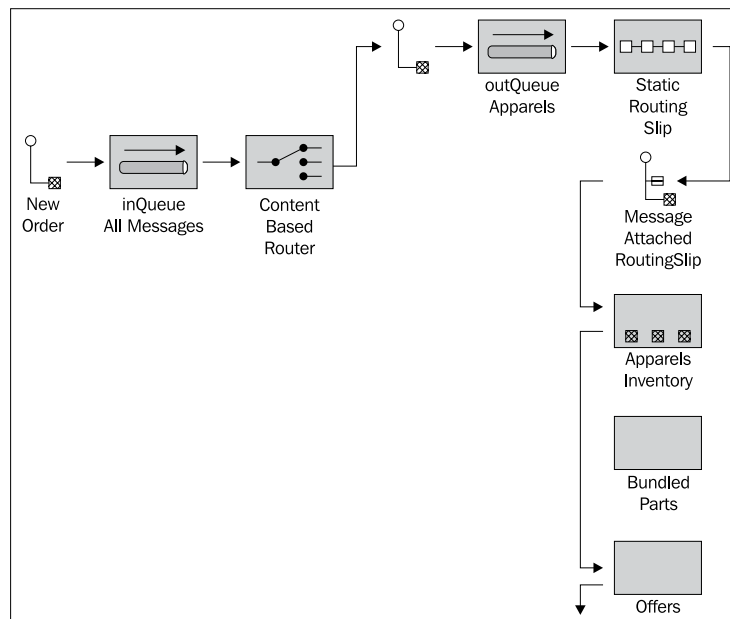


Explanation

You might be aware of the servlet filters, which is a kind of pipe and filter configuration to do processing at the presentation tier. Sometimes we may need to do similar processing, through a series of process blocks. The exact processing blocks through which the message has to be routed can be dynamic based on the type of message. In a static routing slip, these processing blocks or target services are fixed. The ServiceMix routing slip uses In-Out MEPs and errors or faults sent by the targets are reported back to the consumer. In case of errors the routing process is interrupted.

Illustrative Design

When the Acme customer drops items into the shopping cart, we may also need to extend any offers or discounts associated with the selected item. This has to be done after the initial inventory check. We can attach a routing slip to each message so that the messages are routed in series through the inventory module and the offer module. This is explained in the following figure:

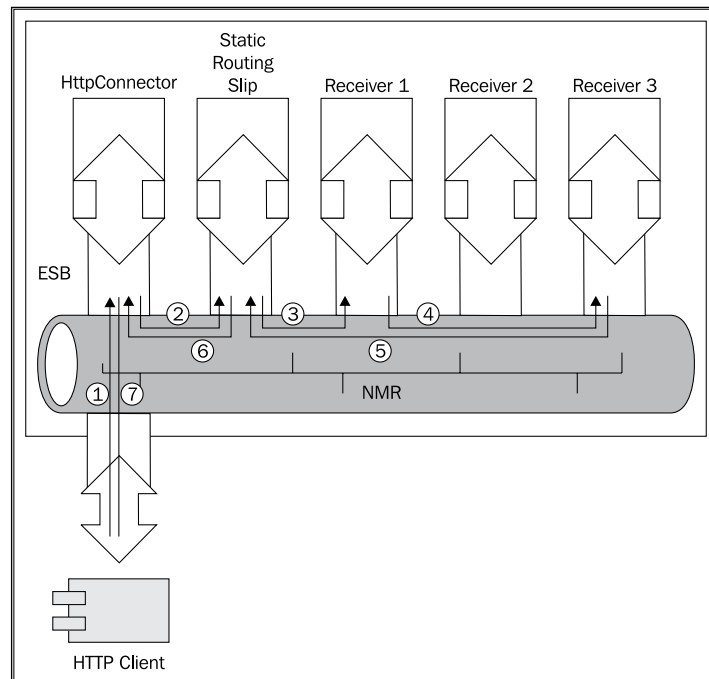


Sample Use Case

The sample use case will have following components:

- **HTTP client:** This is an external HTTP client, placing the XML messages onto the HTTP connector component configured within the ESB.
- **HTTP connector:** This is a ServiceMix HTTP component listening on port 8912. Any incoming messages to this queue will be routed to the next component in the flow chain which is the Static Routing Slip.
- **Static routing slip:** The Static Routing Slip is a `servicemix-eip` component. On receiving in messages, the Static Routing Slip will attach routing slips which are pre-configured to the message. Now the ESB can route the message in series through the services specified through the slips.
- **Receiver component:** Multiple instances of the receiver component are configured in the ESB. The routing slip attaches slips corresponding to only a few of the receivers with every incoming message. Hence the messages are forwarded through these receiver instances only, others are neglected.

On completing the flow, the HTTP client can retrieve back the message from the HTTP connector as shown in the following figure:



Sample Code and Configuration

We configure the pipeline in the `servicemix.xml` file, along with other components described above.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xbean.org/schemas/spring/1.0"
       xmlns:sm="http://servicemix.apache.org/config/1.0"
       xmlns:test="http://xslt.servicemix.apache.org/binildas.com"
       xmlns:eip="http://servicemix.apache.org/eip/1.0"
       xmlns:jms="http://servicemix.apache.org/jms/1.0">
  <classpath>
    <location>.</location>
  </classpath>
  <import resource="classpath:jmx.xml" />
  <import resource="classpath:jndi.xml" />
  <import resource="classpath:security.xml" />
  <import resource="classpath:tx.xml" />
  <import resource="classpath:activemq.xml" />
  <bean id="propertyConfigurer"
        class="org.springframework.beans.factory.config.
              PropertyPlaceholderConfigurer">
    <property name="location"
              value="classpath:servicemix.properties" />
  </bean>
  <sm:container name="jbi"
                monitorInstallationDirectory="false"
                createMBeanServer="true"
                useMBeanServer="true">
    <sm:activationSpecs>
      <sm:activationSpec componentName="httpReceiver"
                          service="test:httpBinding"
                          endpoint="httpReceiver"
                          destinationService="test:routingSlip">
        <sm:component>
          <bean class="org.apache.servicemix.components.
                    http.HttpConnector">
            <property name="host" value="localhost"/>
            <property name="port" value="8912"/>
          </bean>
        </sm:component>
      </sm:activationSpec>
      <sm:activationSpec componentName="receiver1"
                          service="test:receiver1">
        <sm:component>
          <bean xmlns="http://xbean.org/schemas/spring/1.0"
                class="MyReceiver" />
        </sm:component>
      </sm:activationSpec>
    </sm:activationSpecs>
  </sm:container>
</beans>
```

```

        <property name="name">
            <value>1</value>
        </property>
    </bean>
</sm:component>
</sm:activationSpec>
<sm:activationSpec componentName="receiver2"
    service="test:receiver2">
    <sm:component>
        <bean xmlns="http://xbean.org/schemas/spring/1.0"
            class="MyReceiver" >
            <property name="name">
                <value>2</value>
            </property>
        </bean>
    </sm:component>
</sm:activationSpec>
<sm:activationSpec componentName="receiver3"
    service="test:receiver3">
    <sm:component>
        <bean xmlns="http://xbean.org/schemas/spring/1.0"
            class="MyReceiver" >
            <property name="name">
                <value>3</value>
            </property>
        </bean>
    </sm:component>
</sm:activationSpec>
<sm:activationSpec id="servicemix-eip">
    <sm:component>
        <eip:component>
            <eip:endpoints>
                <eip:static-routing-slip service="test:routingSlip"
                    endpoint="routingSlipEndpoint">
                    <eip:targets>
                        <eip:exchange-target
                            service="test:receiver1" />
                        <eip:exchange-target
                            service="test:receiver3" />
                    </eip:targets>
                </eip:static-routing-slip>
            </eip:endpoints>
        </eip:component>
    </sm:component>
</sm:activationSpec>
</sm:activationSpecs>
</sm:container>
</beans>

```

Deploy and Run the Sample

To build the sample, change directory to `ch15\09_StaticRoutingSlip` and type `ant` as shown here:

```
cd ch15\09_StaticRoutingSlip
ant
```

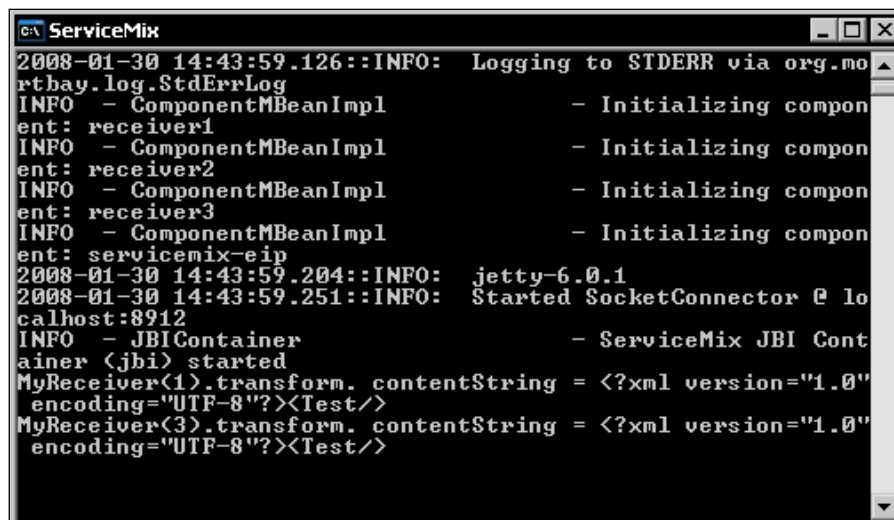
This will compile all the files, including the JMS client program. Now to test the sample, first bring ServiceMix up by executing:

```
cd ch15\09_StaticRoutingSlip
%SERVICEMIX_HOME%\bin\servicemix servicemix.xml
```

Now in a different command prompt execute `ant run`.

```
cd ch15\09_StaticRoutingSlip
ant run
```

The output for the above command is shown in the following screenshot:

A screenshot of a Windows command prompt window titled "ServiceMix". The window displays the following log output:

```
2008-01-30 14:43:59.126::INFO: Logging to STDERR via org.mortbay.log.StderrLog
INFO - ComponentMBeanImpl - Initializing component: receiver1
INFO - ComponentMBeanImpl - Initializing component: receiver2
INFO - ComponentMBeanImpl - Initializing component: receiver3
INFO - ComponentMBeanImpl - Initializing component: servicemix-eip
2008-01-30 14:43:59.204::INFO: jetty-6.0.1
2008-01-30 14:43:59.251::INFO: Started SocketConnector @ localhost:8912
INFO - JBIContainer - ServiceMix JBI Container (jbi) started
MyReceiver(1).transform. contentString = <?xml version="1.0" encoding="UTF-8"?><Test/>
MyReceiver(3).transform. contentString = <?xml version="1.0" encoding="UTF-8"?><Test/>
```

If you observe the ESB console you can see that the ESB routes the message in series through the services specified through the slips (`test:receiver1` and `test:receiver3`).

Summary

Integration requires a level of thinking different from the traditional software engineering where you have to think in terms of software "plugs" and "sockets". I have to admit that my university courses in Machine Design helped me a lot to think of the software integration problem in terms of "shafts" and coupling" – in fact, their counterparts in EAI. I hope you too admit that after reading this chapter. So, the next time when you want to integrate software components think in terms of EAI and patterns. Moreover, try to "select and assemble integration building blocks" rather than hand coding and hardwiring endpoints. This will help you to integrate in a loosely-coupled manner which will facilitate easy service collaboration and orchestration. The next chapter will show you a sample of aggregating multiple services on the ESB using the EAI patterns and guidelines which we learned in this chapter.

16

Sample Service Aggregation

In the previous chapters we covered many JBI components in ServiceMix and also saw a few useful use cases to help solve real life problems using the ESB patterns. Another useful application of ESB is to provide a "Services Workbench" wherein we can use various integration patterns available to aggregate services to carry out business processes. We will look into such a sample use case in this chapter.

We will look at the following topics in the business integration sample:

- Solution architecture
- JBI-based ESB component architecture
- Understanding the message exchange
- Deploy and run the sample

Provision Service Order—Business Integration Sample

To demonstrate the service aggregation, we will choose a typical business integration scenario happening as an outcome of a customer attempting to make a web order entry. Let us take the scenario of a Communication Service Provider (CSP) providing a web access channel for its customers to order **Voice over IP (VOIP) service**.

We will consider a single process of order generation, which is a core process in an order management system (OMS). The order generation process accepts and issues orders. This process can be divided into steps such as order entry and validation. If we consider a single step such as the validation, it can be decomposed into multiple validation activities to be performed by more than one third-party service providers. This makes sense in today's service-oriented environment, where services are provided by multiple vendors and a single aggregate service is composed of multiple line-item services offered by multiple vendors.

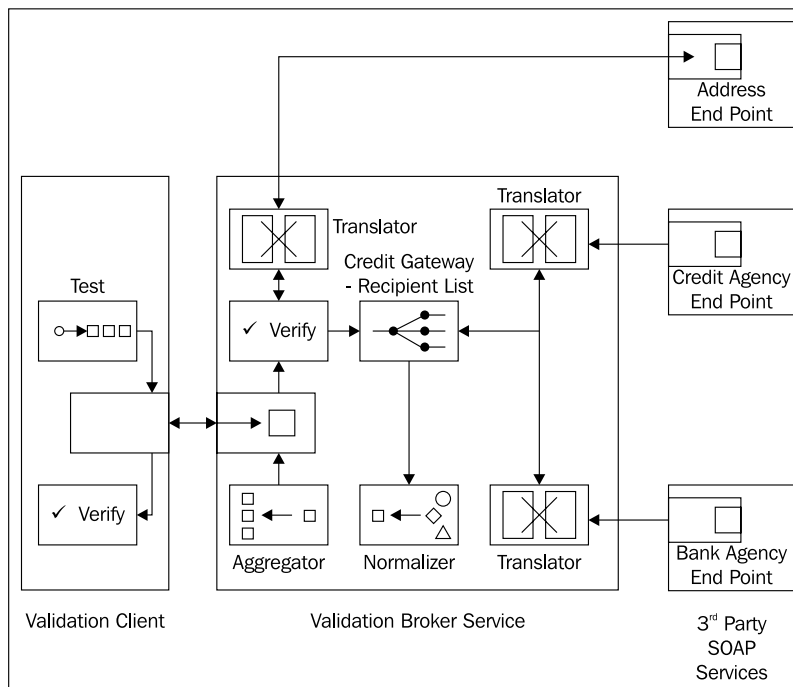
In our example scenario, let us consider three validation services to be done:

- Address validation
- Credit card validation
- Bank history validation

The above fine-grained services are offered by different third-party vendors. Hence, as per industry standards, the best way to access those services is using SOAP over HTTP. The Address Service validates the address entered by the user and also checks to see whether the VOIP service can be provided in the requested area. Credit Gateway is a composite service, and the successful operation of this service depends upon the responses from two other third-party vendor services called "Credit Agency" and "Bank Agency". The Credit Agency checks for the credit worthiness of the customer, and the Bank Agency checks for the customer's banking transaction history.

Solution Architecture

Let us first look at what JBI-based technical components are required for framing our solution architecture. This is shown in the following figure:

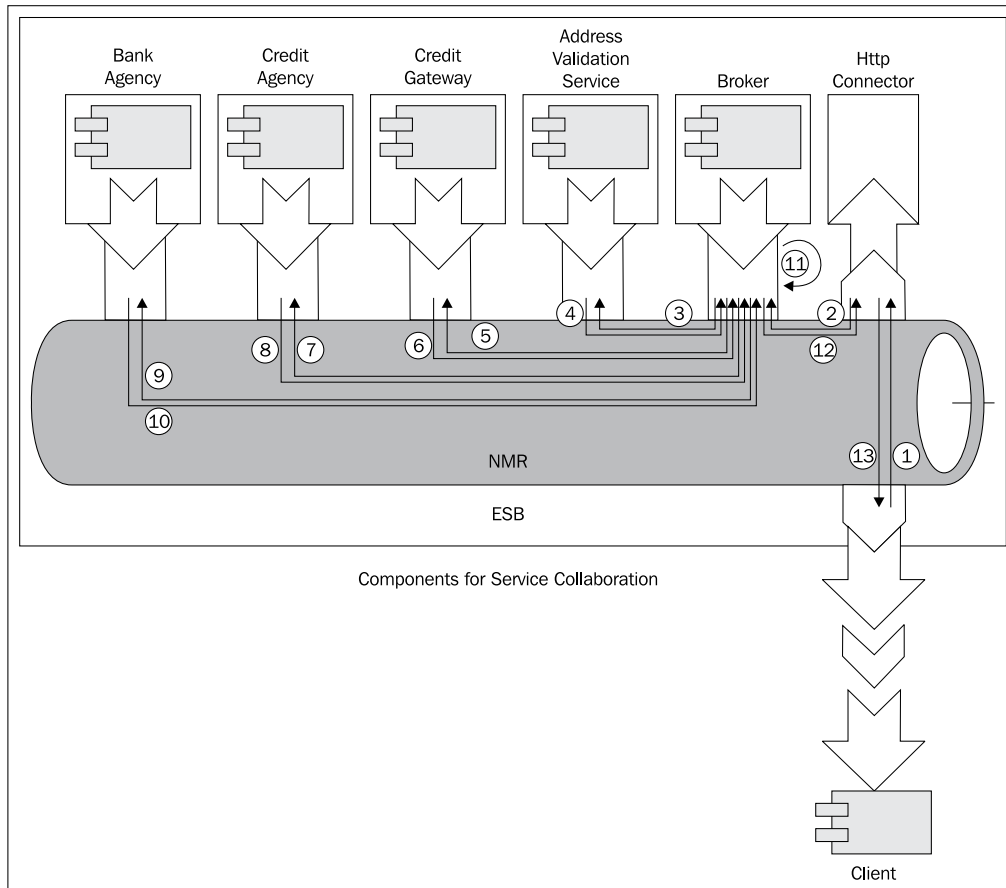


The above figure shows the solution architecture for the business scenario described earlier. In order to frame the ESB solution architecture for the sample scenario, we will first list the individual components required. Individual ESB components are chosen based on the transport requirements to facilitate the information flow. These components are explained in the following list:

- **Endpoint:** An endpoint connects services across systems, applications, and enterprises together. An endpoint exposes standard interfaces and hides all transport-specific aspects, thus providing an abstract plug-in point. The endpoints are required for the sample business scenario to access the address, credit, and bank service.
- **Translator:** The translators convert between message formats and are synonymous with the adapter pattern listed in the *Gang of Four* Design Patterns book. In the sample architecture, the validation broker service needs to talk to the external services through the SOAP protocol. Hence, the Java objects need to be translated to XML format, and vice versa.
- **Normalizer:** The credit and bank services deal with messages that have the same meaning but different formats. This is because different external systems have their own message formats. This means the messages are in different formats, but are semantically equal. A normalizer routes semantically equal messages to different message translators.
- **Recipient list:** In scenarios where we need to route message to multiple endpoints, we will use a recipient list. The recipients can be specified dynamically also. In our scenario, we need to send the same message to both the credit and bank services. Here, once the endpoints are defined, the recipient list will forward the message to all channels associated with the recipients in the list.
- **Aggregator:** As the Credit Gateway depends upon the combined outcome of two other services (Credit Agency and Bank Agency), we need to use a stateful filter to collect and store individual messages until a complete set of related messages has been received. An aggregator does this job by combining the results of individual, but related messages so that they can be processed as a whole.

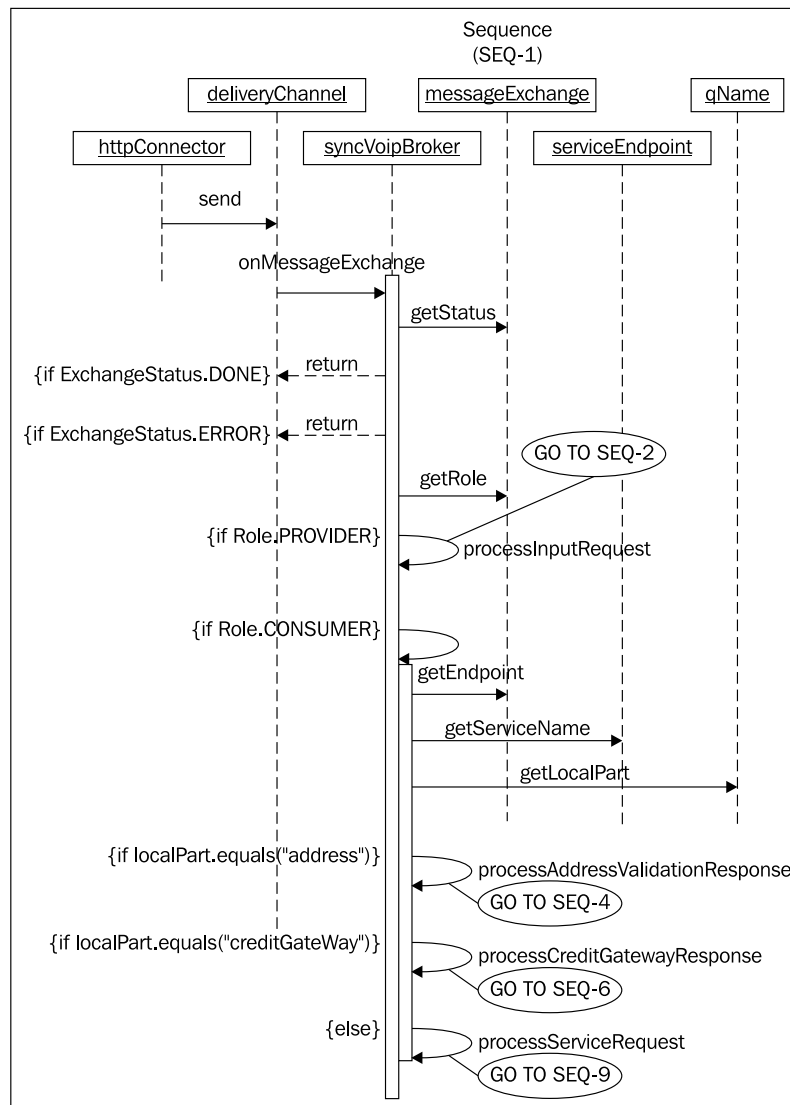
JBI-based ESB Component Architecture

The following figure shows the various JBI components we assemble to design the integration solution for the order validation business process. The functionality of these components is already discussed. We are showing the message flow in this diagram. We will detail out the message flows in the next section with the help of a UML diagram.



Understanding the Message Exchange

As it's said that a diagram speaks a thousand words, so does the UML diagram that narrates complex message flows in a simple manner. Hence, let us understand the message exchanges with the help of sequence diagrams. The **Sequence 1** is shown in the following figure:



The Sequence 1 shows the main "Control flow", orchestrated by the broker. The control code is provided within the `onMessageExchange` method in the broker and is reproduced in the following code:

```
public class SyncVoipBroker extends ComponentSupport implements
                                MessageExchangeListener
{
    private Map aggregations = Collections.synchronizedMap(
                                                new HashMap());

    private JBICContainer container;
    private String name;
    private String addressNamespaceURI;
    private String addressLocalPart;
    private String creditGatewayNamespaceURI;
    private String creditGatewayLocalPart;
    private Object payLoad;
    private boolean creditVetoed;
    private volatile int stack = 0;
    public void onMessageExchange(MessageExchange exchange)
                                throws MessagingException
    {
        System.out.println("SyncVoipBroker.onMessageExchange.
                                ExchangeId          :X: " +
                                exchange.getExchangeId());
        ServiceEndpoint serviceEndpoint = null;
        if (exchange.getStatus() == ExchangeStatus.DONE)
        {
            System.out.println("SyncVoipBroker.onMessageExchange.
                                ExchangeStatus.DONE");
            dispose();
            return;
        }
        if (exchange.getStatus() == ExchangeStatus.ERROR)
        {
            System.out.println("SyncVoipBroker.onMessageExchange.
                                ExchangeStatus.ERROR");
            return;
        }
        if (exchange.getRole() == Role.PROVIDER)
        {
            System.out.println("SyncVoipBroker.onMessageExchange.
                                Role.PROVIDER");
            processInputRequest(exchange);
        }
        else
        {
            System.out.println("SyncVoipBroker.onMessageExchange.
                                Role.CONSUMER");
            serviceEndpoint = exchange.getEndpoint();
        }
    }
}
```

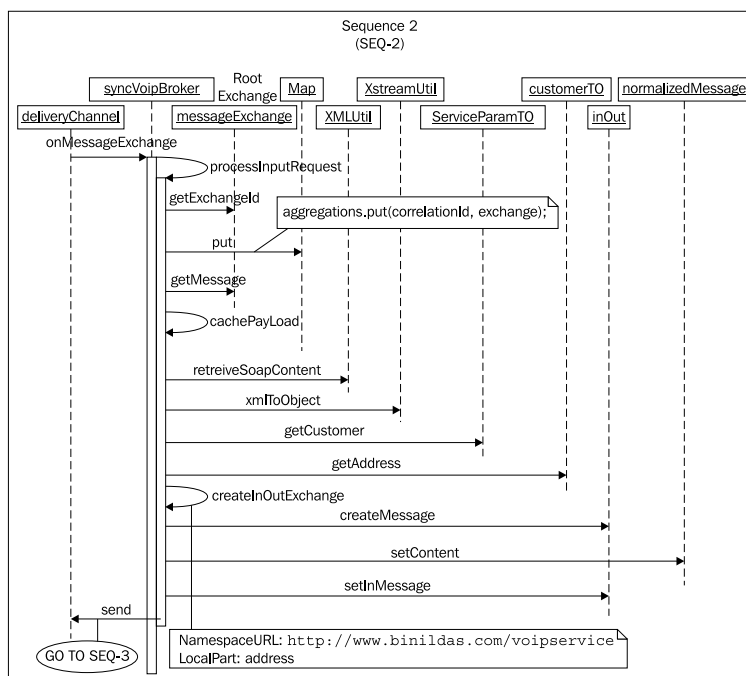
```

        if (serviceEndpoint.getServiceName().getLocalPart().
            equals(addressLocalPart))
        {
            processAddressValidationResponse(exchange);
        }
        else if (serviceEndpoint.getServiceName().getLocalPart().
            equals(creditGatewayLocalPart))
        {
            processCreditGatewayResponse(exchange);
        }
        else
        {
            processServiceRequest(exchange);
        }
    }
}
}

```

If the `ExchangeStatus` is "DONE" or "ERROR" for the initial `MessageExchange`, we simply log and return. If not, then the broker component can take part in the message exchange in two roles namely the provider or the consumer.

For the consumer role, there are multiple paths which will be explained later. But for the provider role, the broker transfers control to **Sequence 2** listed in the following figure:



In the normal flow when a client sends a message to the ESB, the `onMessageExchange` method in the broker will be invoked in the provider role. We label this `messageExchange` as the Root Exchange (to differentiate this exchange from other nested exchanges, which will be described shortly) and keep a pointer of that in a Map for future reference. The XML payload which we retrieve from the `normalizedMessage` is converted to Java objects using the XStream Java XML binding utilities and stored for future reference. We then create a new **InOut** exchange with address as the LocalPart and send it to the `deliveryChannel`. The following code snippet details out these sequence of events:

```
public class SyncVoipBroker extends ComponentSupport implements
                                MessageExchangeListener
{
    private void processInputRequest(MessageExchange exchange)
                                throws MessagingException
    {
        NormalizedMessage copyMessage = exchange.createMessage();
        NormalizedMessage inNormalizedMessage =
            exchange.getMessage("in");
        getMessageTransformer().transform(exchange,
            inNormalizedMessage, copyMessage);
        Source content = copyMessage.getContent();
        String contentString = null;
        if (content instanceof DOMSource)
        {
            contentString = XMLUtil.retrieveSoapContent(
                ((DOMSource) content).getNode());
            payLoad = XStreamUtil.xmlToObject(contentString);//Cache it
        }
        CustomerTO customerTO = ((ServiceParamTO)
            payLoad).getCustomer();
        AddressTO addressTO = customerTO.getAddress();
        String xmlAddress = XStreamUtil.objectToXml(addressTO);
        Source addressSource = new StreamSource(new
            ByteArrayInputStream(xmlAddress.
                getBytes()));

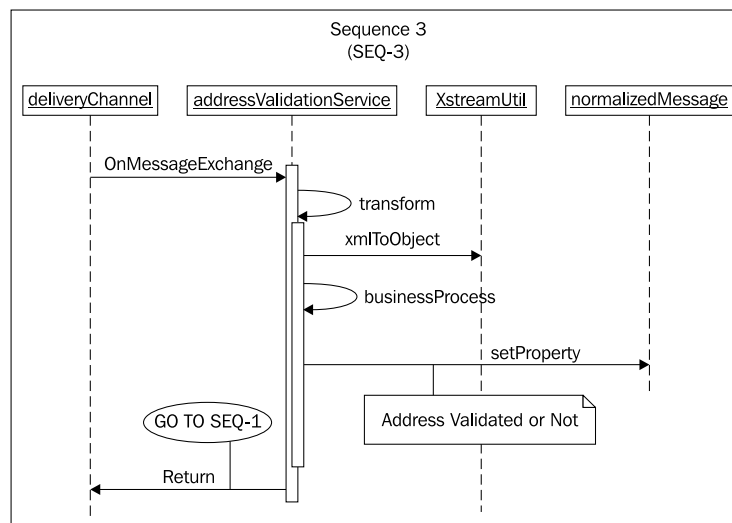
        String correlationId = null;
        if (exchange.getStatus() == ExchangeStatus.ACTIVE)
        {
            correlationId = exchange.getExchangeId();
            aggregations.put(correlationId, exchange);
            InOut inout = createInOutExchange(new QName(
                addressNamespaceURI, addressLocalPart),
                null, null);
            inout.setProperty(Constants.CORRELATION_ID_KEY,
```

```

                                correlationId);
    NormalizedMessage msg = inout.createMessage();
    msg.setContent(addressSource);
    inout.setInMessage(msg);
    send(inout);
}
System.out.println("SyncVoipBroker.processInputRequest.
                    correlationId          : " + correlationId);
}
}

```

Note the newly created `messageExchange` which will again invoke the `onMessageExchange` method of the address (which in turn calls the `transform` method of `Address`—Sequence 3). When the `Address` service returns, the `onMessageExchange` method in the broker will be invoked again, but this time in the consumer role—consumer to the address. This difference is subtle but important because in an In-Out exchange, it is the consumer who has to end a message exchange with a "done" or "error" status. The `Address` service is simple and is shown in Sequence 3. The following figure shows the **Sequence 3** UML diagram:



In `Address` service, we validate the address and the result of the validation is set as a property in the "out" message of the `MessageExchange`. This is shown in the following code:

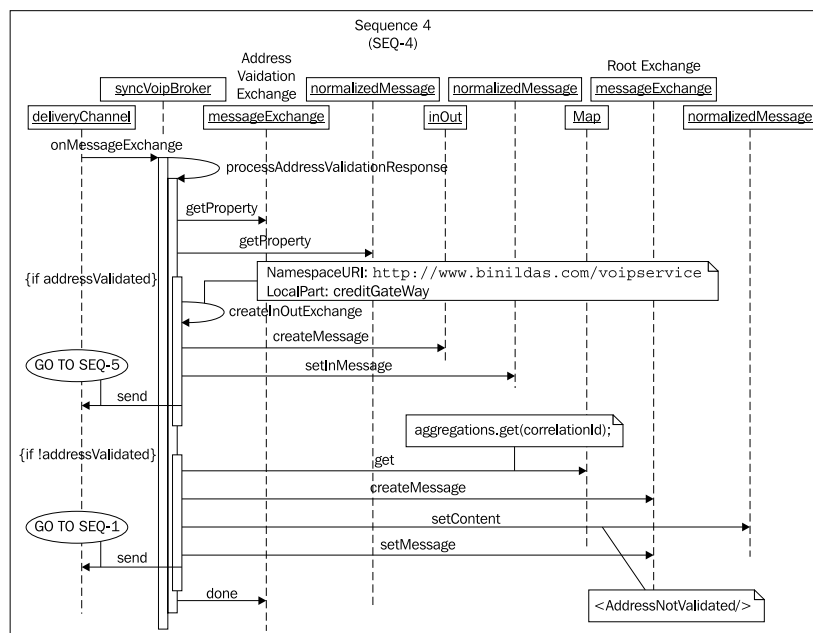
```

public class AddressValidationService extends
    TransformComponentSupport
{

```

```
protected boolean transform(MessageExchange exchange,
                           NormalizedMessage in, NormalizedMessage out)
                           throws MessagingException
{
    String correlationId = (String)exchange.getProperty(
        Constants.CORRELATION_ID_KEY);
    System.out.println("AddressValidationService.transform.
        correlationId          : " + correlationId);
    Source content = in.getContent();
    String contentString = null;
    AddressTO addressTO = null;
    if (content instanceof StreamSource)
    {
        contentString = XMLUtil.formatStreamSource(
            (StreamSource) content);
        addressTO = (AddressTO)XStreamUtil.
            xmlToObject (contentString);
    }
    else
    {
        log.debug("AddressValidationService-content.getClass() : " +
            content.getClass() + " ; content : " + content);
    }
    out.setProperty(Constants.ADDRESS_VALIDATED_KEY, Boolean.TRUE);
    return true;
}
```

When the Address service returns, the `onMessageExchange` method in the broker will be invoked (Sequence 1). Now as the broker component is in the consumer role and the LocalPart is "address", the control flows to the `processAddressValidationResponse` which is given in **Sequence 4**.



In `processAddressValidationResponse` we check whether the address is validated. If not, we set the XML message "`<AddressNotValidated/>`" at the `normalizedMessage` to the `Root Exchange` so that the message exchange flow will be completed and control flows back to the `deliveryChannel` first and then to the client. If address is validated, we create a new `InOut` exchange with `creditGateWay` as the `LocalPart` and send it to the `deliveryChannel`.

In both the scenarios above, note that we set the (Address Validation) `messageExchange` status to "done". The code listing in the following shows processing the address validation response:

```

public class SyncVoipBroker extends ComponentSupport
    implements MessageExchangeListener
{
    private void processAddressValidationResponse(MessageExchange
        exchange) throws MessagingException
    {
        String correlationId = (String) getProperty(exchange,
            Constants.CORRELATION_ID_KEY);
        System.out.println("SyncVoipBroker.
            processAddressValidationResponse.correlationId :
                " + correlationId);
        boolean isAddressValidated = ((Boolean)
            getOutProperty(exchange,
                Constants.ADDRESS_VALIDATED_KEY)).booleanValue();
        MessageExchange rootExchange = null;
    }
}

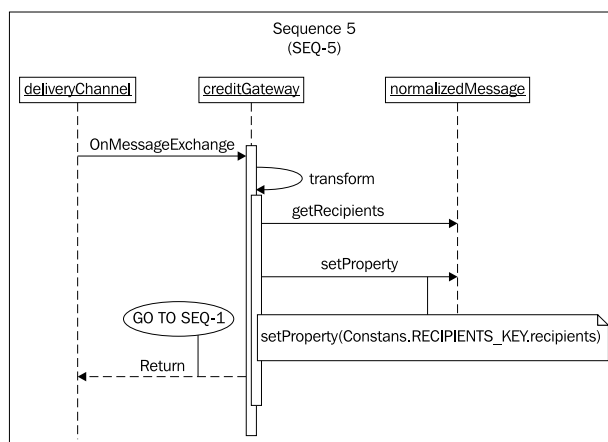
```

```

    if (isAddressValidated)
    {
        InOut inout = createInOutExchange(new QName(
            creditGatewayNamespaceURI,
            creditGatewayLocalPart), null, null);
        inout.setProperty(Constants.CORRELATION_ID_KEY,
            correlationId);
        NormalizedMessage msg = inout.createMessage();
        inout.setInMessage(msg);
        send(inout);
    }
    else
    {
        rootExchange = (MessageExchange)aggregations.
            get(correlationId);
        NormalizedMessage response = rootExchange.createMessage();
        response.setContent(new StringSource("
            <AddressNotValidated/>"));
        rootExchange.setMessage(response, "out");
        send(rootExchange);
        aggregations.remove(correlationId);
    }
    done(exchange);
}
}

```

Assuming the "Address Validated" scenario as the normal flow, the newly created `MessageExchange` will now again invoke the `onMessageExchange` method of the Credit Gateway (which in turn calls the `transform` method of `creditGateway` – Sequence 5). When the Credit Gateway service returns, the `onMessageExchange` method in the broker will be invoked again, this time also in the consumer role – consumer to the Credit Gateway service. The Credit Gateway service is simple. The **Sequence 5** is shown in the following figure:



In the Credit Gateway service we create a recipient list and set it as a property in the "out" message of the MessageExchange. This is shown in the following code:

```
public class CreditGateway extends TransformComponentSupport
{
    private Map endPoints;
    public void setEndPoints(Map endPoints)
    {
        this.endPoints = endPoints;
    }
    protected boolean transform(MessageExchange exchange,
                               NormalizedMessage in, NormalizedMessage out)
        throws MessagingException
    {
        String correlationId = (String)exchange.getProperty(
            Constants.CORRELATION_ID_KEY);
        System.out.println("CreditGateway.transform.correlationId
                           : " + correlationId);
        out.setProperty(Constants.RECIPIENTS_KEY, getRecipients());
        return true;
    }
    private QName[] getRecipients()
    {
        QName[] recipients = new QName[endPoints.size()];
        String theNamespaceURI = null;
        String theLocalPart = null;
        int times = 0;
        for(Iterator iterator = endPoints.keySet().iterator();
            iterator.hasNext();)
        {
            theLocalPart = (String) iterator.next();
            theNamespaceURI = (String) endPoints.get(theLocalPart);
            recipients[times++] = new QName(theNamespaceURI,
                                             theLocalPart);
        }
        return recipients;
    }
}
```

Here we generate a static recipient list. The recipient list is configured in the servicemix.xml file for the Credit Gateway component as shown here:

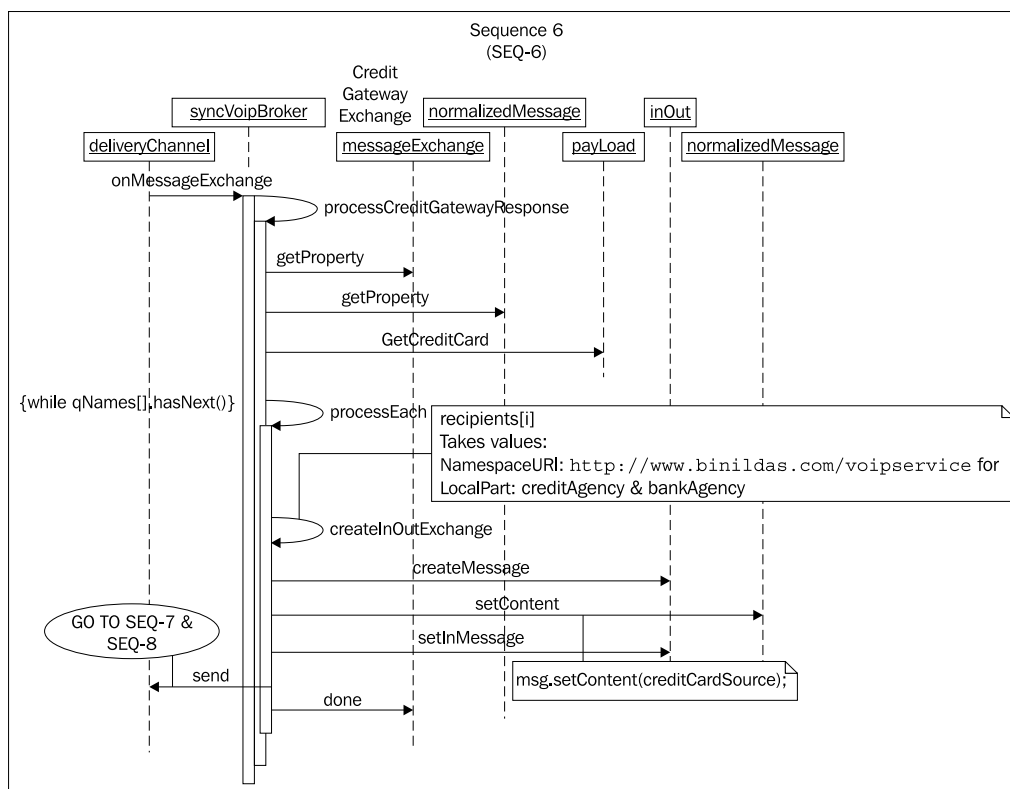
```
<sm:activationSpec componentName="creditGateWay"
                    endpoint="creditGateWay"
                    service="binil:creditGateWay">
  <sm:component>
    <bean class="com.binildas.esb.servicemix.serviceassembly.
              voipservice.CreditGateway">
      <property name="name">
        <value>Credit</value>
```

```

    </property>
    <property name="endPoints">
      <map>
        <entry key="creditAgency">
          <value>http://www.binildas.com/voipservice</value>
        </entry>
        <entry key="bankAgency">
          <value>http://www.binildas.com/voipservice</value>
        </entry>
      </map>
    </property>
  </bean>
</sm:component>
</sm:activationSpec>

```

When the Credit Gateway service returns, the `onMessageExchange` method in the broker will be invoked (Sequence 1). Now as the broker component is again in the consumer role and the LocalPart is "creditGateWay", the control flows to the `processCreditGatewayResponse` which is given in **Sequence 6**. This is shown in the following figure:



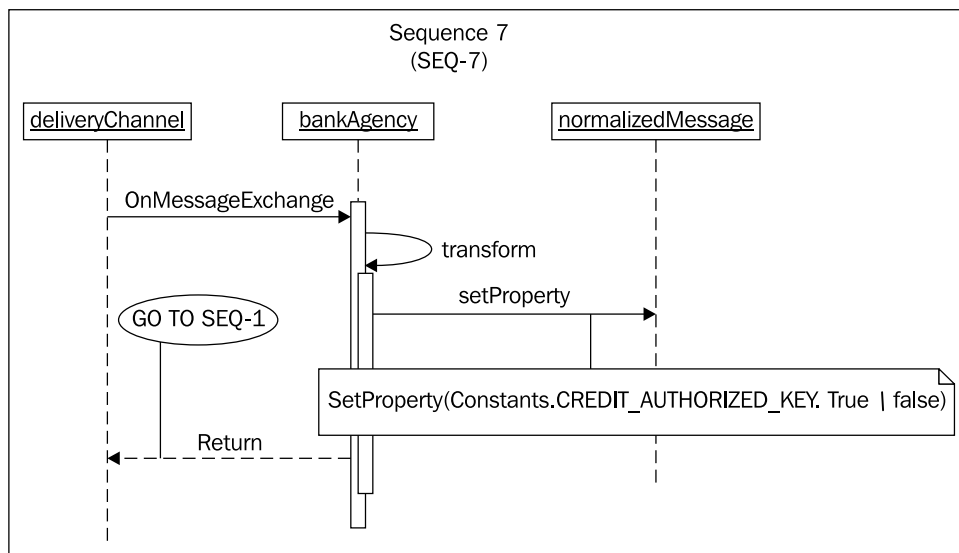
Our next aim is to route the message to all the targets defined in the recipient list. We create as many new InOut exchanges as there are entries in the recipient list and send them to the delivery channel. The following code snippet details out these sequence of events:

```
public class SyncVoipBroker extends ComponentSupport
    implements MessageExchangeListener
{
    private void processCreditGatewayResponse(MessageExchange
        exchange) throws MessagingException
    {
        String correlationId = (String) getProperty(exchange,
            Constants.CORRELATION_ID_KEY);
        System.out.println("SyncVoipBroker.
            processCreditGatewayResponse.
                correlationId      : " +
                    correlationId);
        QName[] recipients = (QName[]) getOutProperty(exchange,
            Constants.RECIPIENTS_KEY);
        CreditCardTO creditCardTO = ((ServiceParamTO)
            payload).getCreditCard();
        String xmlCreditCard = XStreamUtil.objectToXml(creditCardTO);
        Source creditCardSource = null;
        InOut inout = null;
        NormalizedMessage msg = null;
        for(int qNames = 0; qNames < recipients.length; qNames++)
        {
            inout = createInOutExchange(recipients[qNames], null, null);
            inout.setProperty(Constants.CORRELATION_ID_KEY,
                correlationId);

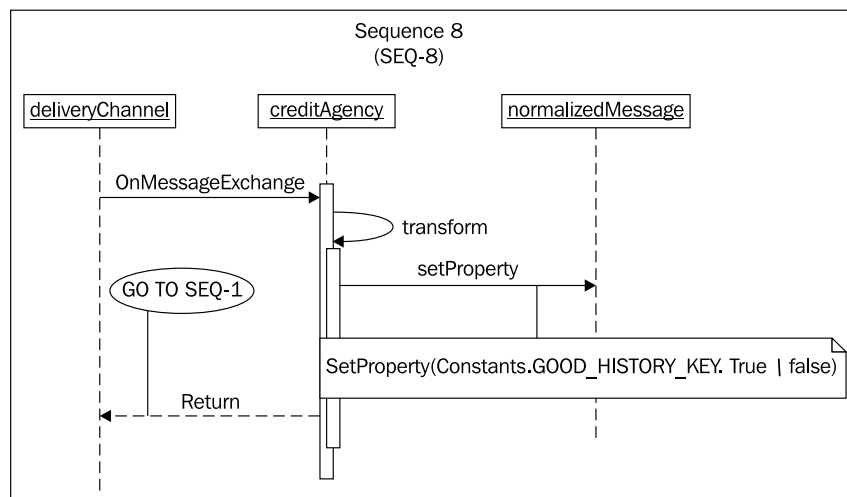
            msg = inout.createMessage();
            creditCardSource = new StreamSource(new
                ByteArrayInputStream(xmlCreditCard.
                    getBytes()));
            msg.setContent(creditCardSource);
            inout.setInMessage(msg);
            stack++;
            send(inout);
        }
        done(exchange);
    }
}
```


Here we send the credit card details to the recipients (Bank Agency & Credit Agency).

The newly created message exchanges will now invoke the `onMessageExchange` method of the Bank Agency & Credit Gateway (which in turn calls the `transform` method of Bank Agency & Credit Agency respectively) which is again simple and is shown in Sequence 7 and Sequence 8. The code is not reproduced here, since they are very trivial. The **Sequence 7** is shown in the following figure:

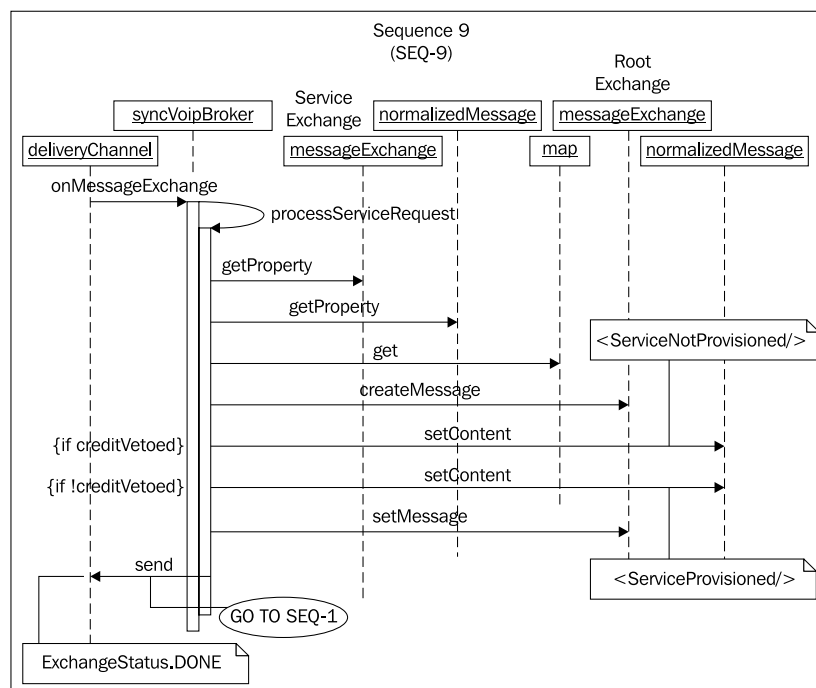


The **Sequence 8** is shown in the following figure:



When the Bank Agency or the Credit Agency services returns, the `onMessageExchange` method in the broker will be invoked again in the consumer role – consumer to the Bank Agency and Credit Agency. In both the scenarios, the control will be the `transform` method.

The **Sequence 9** is shown in the following figure:



Let us now look into the code of `processServiceRequest` in detail:

```

public class SyncVoipBroker extends ComponentSupport
    implements MessageExchangeListener
{
    private void processServiceRequest(MessageExchange exchange)
        throws MessagingException
    {
        String correlationId = (String) getProperty(exchange,
            Constants.CORRELATION_ID_KEY);
        System.out.println("SyncVoipBroker.processServiceRequest.
            correlationId          : " + correlationId);
        Boolean creditauthorized = (Boolean) getOutProperty(exchange,
            Constants.CREDIT_AUTHORIZED_KEY);
        Boolean goodhistory = (Boolean) getOutProperty(exchange,
            Constants.GOOD_HISTORY_KEY);
        MessageExchange rootExchange = (MessageExchange) aggregations.
    }
}

```

```
get(correlationId);

done(exchange);
stack--;
if(((creditauthorized != null) &&
    !(creditauthorized.equals(Boolean.TRUE)))
    ||
    ((goodhistory != null) &&
    !(goodhistory.equals(Boolean.TRUE))))
{
    creditVetoed = true;
    System.out.println("SyncVoipBroker.processServiceRequest -
                        creditVetoed : " + creditVetoed);
}
if(0 == stack)
{
    NormalizedMessage response = rootExchange.createMessage();
    if(creditVetoed)
    {
        response.setContent(new StringSource(
            "<ServiceNotProvisioned/>"));
    }
    else
    {
        response.setContent(new StringSource(
            "<ServiceProvisioned/>"));
    }
    rootExchange.setMessage(response, "out");
    send(rootExchange);
    aggregations.remove(correlationId);
}
}
```

Here, based on the outcome of the previous service responses we can decide whether the service can be provisioned or not and accordingly we can create an XML message. Now comes the difference – we create a normalized message out of the Root Exchange (we are not going to create a new message exchange, since we are almost done with the process), set the XML message as its content and call `send` on the `rootExchange`.

The Root Exchange will again be routed through the delivery channel back to the `onMessageExchange` of the Broker, but this time the message exchange status is already set as "DONE". So as shown in Sequence 1, the broker now returns back to the delivery channel and the delivery channel in turn sends back any response to the client.

This completes the entire process.

Deploying and Running the Sample

As a first step, if you haven't done it before, edit `examples.PROPERTIES` (provided along with the code download for this chapter) and change the paths there to match your development environment. Now to build the entire codebase and deploy the sample in a single go, change directory to `ch16\voipservice` which contains a top-level `build.xml` file. Execute `ant` there.

```
cd ch16\voipservice
ant
```

Now, bring up the ServiceMix container by executing the `broker.xml` file contained in the same folder.

```
cd ch16\voipservice
%SERVICEMIX_HOME%/bin/servicemix broker.xml
```

The `Client.html` file provided again in the same folder can be used to send messages to test the deployed service.

The message exchanges we described in the previous section can be understood better by looking at the ServiceMix console logging shown in the following screenshot:



```
ServiceMix
e unit: CreditGateway
INFO - ServiceUnitLifeCycle - Starting service un
it: CreditGateway
INFO - ComponentMBeanImpl - Initializing compon
ent: creditGateWay
INFO - ComponentMBeanImpl - Starting component:
creditGateWay
INFO - AutoDeploymentService - Directory: deploy:
Finished installation of archive: creditgateway-sa.zip
SyncVoipBroker.onMessageExchange. ExchangeId
:X: ID:10.10.10.10-117cb8e36f1-2:0
SyncVoipBroker.onMessageExchange. Role.PROVIDER
SyncVoipBroker.processInputRequest. correlationId
: ID:10.10.10.10-117cb8e36f1-2:0
AddressValidationService.transform. correlationId
: ID:10.10.10.10-117cb8e36f1-2:0
SyncVoipBroker.onMessageExchange. ExchangeId
:X: ID:10.10.10.10-117cb8e36f1-6:0
SyncVoipBroker.onMessageExchange. Role.CONSUMER
SyncVoipBroker.processAddressValidationResponse. correlation
Id : ID:10.10.10.10-117cb8e36f1-2:0
CreditGateway.transform. correlationId
: ID:10.10.10.10-117cb8e36f1-2:0
SyncVoipBroker.onMessageExchange. ExchangeId
:X: ID:10.10.10.10-117cb8e36f1-6:1
SyncVoipBroker.onMessageExchange. Role.CONSUMER
SyncVoipBroker.processCreditGatewayResponse. correlationId
: ID:10.10.10.10-117cb8e36f1-2:0
CreditAgency.transform. correlationId
: ID:10.10.10.10-117cb8e36f1-2:0
BankAgency.transform. correlationId
: ID:10.10.10.10-117cb8e36f1-2:0
SyncVoipBroker.onMessageExchange. ExchangeId
:X: ID:10.10.10.10-117cb8e36f1-6:3
SyncVoipBroker.onMessageExchange. Role.CONSUMER
SyncVoipBroker.processServiceRequest. correlationId
: ID:10.10.10.10-117cb8e36f1-2:0
SyncVoipBroker.onMessageExchange. ExchangeId
:X: ID:10.10.10.10-117cb8e36f1-6:2
SyncVoipBroker.onMessageExchange. Role.CONSUMER
SyncVoipBroker.processServiceRequest. correlationId
: ID:10.10.10.10-117cb8e36f1-2:0
SyncVoipBroker.onMessageExchange. ExchangeId
:X: ID:10.10.10.10-117cb8e36f1-2:0
SyncVoipBroker.onMessageExchange. ExchangeStatus.DONE
```

Summary

MOM has been serving as a great enabler for creating loosely coupled applications for many years. Now MOM provides us the EAI pattern to integrate not only applications but also services and components in a seamless manner.

Distributed components are the norm in today's enterprise computing, aggregating, and orchestrating the message flows across such multiple components provide us with a new flexibility in defining business process – by separating out individual services into multiple components and thus reducing the overall complexity of the process.

We use the "divide and rule" principle to manage the complexity by splitting out the functionality into multiple components and services. However, then these distributed components have to integrate together to provide aggregate or composite services which we can do at the ESB-level. You have seen such a sample in this chapter.

In the next chapter we will look at the JBI bus in a slightly different perspective – in a non-functional or a QOS perspective.

17

Transactions, Security, Clustering, and JMX

No book on programming with any framework will be complete without a mention of the various non functional and QOS features supported by the framework. In this chapter we will visit a few selected QOS features which have an impact on the programming and deployment aspects using the ServiceMix ESB, which are listed as follows:

- Transactions
- Security
- Clustering
- JMX

In fact, if we have to address the above features exhaustively then we may need many pages (or a single book by itself). However, as this book is intended to cover many other aspects, we will limit our discussion to a single chapter. At the same time we will see that the reader will not only have an overview of the above features within ServiceMix, but also is equipped with the tools, code samples, and design aspects so as to enable him for further reading and development.

In this chapter we will look into the following:

- Cross cutting concerns in ServiceMix
- Samples demonstrating transactions, security, clustering, and JMX

Cross Cutting Concerns—Support Inside ServiceMix

This chapter is slightly different from others because almost all the other chapters were dealing with a particular way of binding services to ServiceMix. In this chapter we are going to address programming and/or deployment concerns which can be applied across any or all of the binding mechanisms covered in the other chapters.

Just as it is possible in normal programming such as web component development or server-side business component development, we can also attach various QOS features to the component deployment model in ServiceMix. The peculiar thing with these QOS features is that they are not hardwired through code along with the BCs or SEs, but applied over the components in a declarative manner. The advantage is that the characteristics of these QOS features can be easily changed by altering the particular configuration.

Let us now look into the selected QOS features and their support within ServiceMix.

Transactions

Transactions guarantee atomicity of the operations (message flows) between the components. We know that various components can be plugged into the JBI bus which can take part in the message exchange. Transactions can be associated with these message exchange flows. The scope and synchronicity of these transactions mainly depends upon which "send" primitive we use to exchange messages between components.

The ServiceMix JBI provides an option to set transactions at the container-level. The following listing shows how to enlist JBI exchanges in transactions.

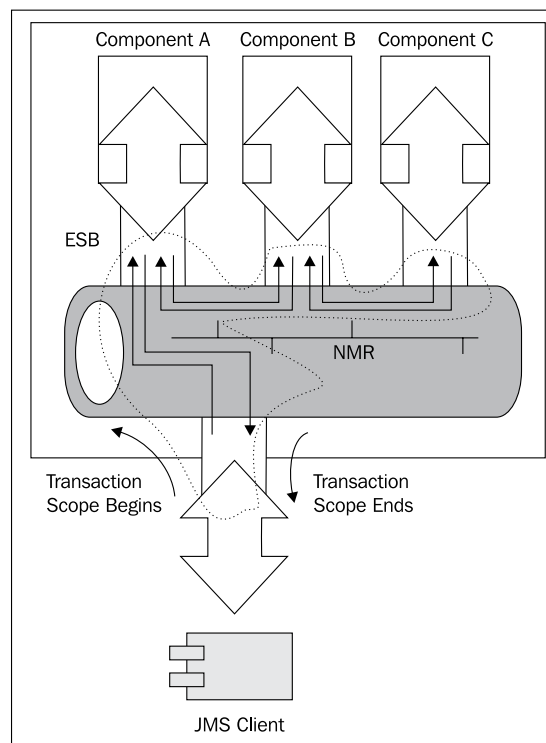
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:sm="http://servicemix.apache.org/config/1.0">
  <sm:container id="jbi"
    embedded="true"
    depends-on="jndi,broker"
    autoEnlistInTransaction="true"
    transactionManager="#transactionManager">
  </sm:container>
  <jencks:transactionManager id="transactionManager" />
</beans>
```

We can set the `autoEnlistInTransaction` attribute for the ServiceMix JBI container to true so that every time a JBI exchange is sent, it will be enlisted in the current transaction.

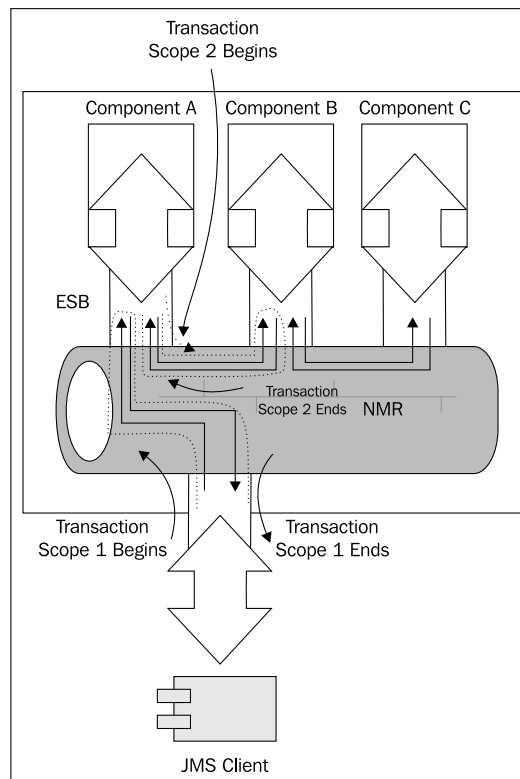
We can now use `send` or `sendSync` to send the message exchanges and depending upon which one is used, the semantics of transaction propagation also is different.

If `sendSync` is used to send an exchange, the implied semantic is that the transaction flows with the exchange and that the service provider has to answer synchronously and also enlist any required resources inside the transaction. In Chapter 3 we discussed about the various flow (NMR Flows) options available within ServiceMix of which the JCA flow needs special attention. The `servicemix-jms` component, if used in a JCA flow, can start transactions whereas transactions within a synchronous flow are not created by default.

The following figure shows the propagation of transaction context when `sendSync` is used:



If we use `send`, the sending of the message will be enlisted in the current transaction, but the message exchange processing will be deferred and then handled in its own thread disjoint from the previous transaction. We have to use the JCA flow to use the transactions with the asynchronous message exchanges. A sample flow is shown in the following figure:



Programming the two types of synchronicity of transaction in the message exchange are shown in the following list:

- **Programming synchronous transactional exchange:** Synonymous to the JDBC API, we need to first get a handle to the underlying `TransactionManager` and then mark the transaction to begin. Then we send the message exchange using `sendSync` followed by a commit or a rollback. The sequences of steps are shown here:

```
TransactionManager txManager = (TransactionManager)
    jbiContext.getTransactionManager();
```

```
tm.begin();
InOnly messageExchange = createInOnly();
jbiContext.getDeliveryChannel().sendSync(messageExchange);
tm.commit();
```

- **Programming asynchronous transactional exchange:** Creating asynchronous transactional exchange is similar to the synchronous method, but we use `send` instead of `sendSync`.

```
TransactionManager txManager = (TransactionManager)
    jbiContext.getTransactionManager();
tm.begin();
InOnly messageExchange = createInOnly();
jbiContext.getDeliveryChannel().send(messageExchange);
tm.commit();
```

The later sections in this chapter will have working samples which will make clear the points explained here.

Security

ServiceMix's HTTP component namely `servicemix-http` provides facility to configure security. Let us look at a few of them listed here:

- **HTTP basic authentication:** HTTP 1.1 specifications defines basic authentication. As per that, when a client tries to access a resource marked as protected in the server, the server prompts for a username and password combination. This is followed by the browser prompting with a username and password entry form where the user can enter the relevant information and submit. If the username and password entered by the user is authenticated by the server, access will be granted to the requested resource. Otherwise, depending upon the server's policy, the prompt will be repeated for a few times (usually three). The main drawback with HTTP basic authentication is that the passwords are sent across the network base64 encoded, which is a kind of plain text format. As the passwords are not encrypted, they are vulnerable to hacks. Hence as an additional precaution, we can either do encryption or use some other stronger mechanism.

For the `servicemix-http` component, we can configure basic authentication by configuring the endpoint as shown in the following code:

```
<http:endpoint service="test:httpConsumer"
    endpoint="httpConsumer"
    targetService="test:echo"
    role="consumer"
    locationURI="http://localhost:8198/Service/"
    authMethod="basic"
    defaultMep="http://www.w3.org/2004/08/wsdl/in-out">
</http:endpoint>
```

In this case, before the `servicemix-http` component routes the request to the `targetService`, a basic authentication challenge is initiated by the server and the request will be routed to the `targetService` if and only if the username and the password entered by the user matches with what is configured in the container.

- **SSL:** ServiceMix can be configured to use secure transport using SSL.

The HTTP consumer role for the `servicemix-http` component can be configured to use SSL as follows:

```
<http:endpoint service="test:YourConsumerService"
               endpoint="yourConsumer"
               role="consumer"
               locationURI="https://localhost:8193/Service/"
               defaultMep="http://www.w3.org/2004/08/wsdl/in-out">
  <http:ssl>
    <http:sslParameters keyStore="classpath:com/binildas/esb/
                           servicemix/security/server.keystore"
                       keyStorePassword="keystorepassword"/>
  </http:ssl>
</http:endpoint>
```

In the consumer role, `test:YourConsumerService` will be set as the destination for the above consumer after the message is received through the SSL transport.

Similarly, the HTTP provider role for the `servicemix-http` component can be configured to use SSL as follows:

```
<http:endpoint service="test:YourProviderService"
               endpoint="yourProvider"
               role="provider"
               locationURI="https://localhost:8193/Service/"
               defaultMep="http://www.w3.org/2004/08/wsdl/in-out">
  <http:ssl>
    <http:sslParameters keyStore="classpath:com/binildas/esb/
                           servicemix/security/server.keystore"
                       keyStorePassword="keystorepassword"
                       trustStore="classpath:com/binildas/esb/
                           servicemix/security/client.keystore"
                       trustStorePassword="truststorepassword"/>
  </http:ssl>
</http:endpoint>
```

We will have a working sample demonstrating HTTP basic security later in this chapter.

Clustering

Architecting and designing an application by looking at the Non Functional Requirements (NFR) along with the usual functional requirements is important to ensure fail safe operation under normal as well as abnormal usage conditions. There can be specific hours in a day or specific months during a year when your application will receive more hits than normal. The applications need to be designed taking this into account.

The scalability and availability of any application depends on the multiple layers in the application deployment stack. The following list gives a few of the main aspects:

- Application algorithms and design patterns.
- Application frameworks for functionalities such as caching, session replication, and persistence.
- Operating system support providing heap limit, green threads, and native threads.
- Hardware infrastructure, providing 32 or 64 bit word capability.

The above list is never exhaustive, but just the main and evident layers.

The availability of any application depends on the ability of the deployment environment to recover from a failure with a minimum amount of downtime without any data corruption. Software and hardware are prone to failure, but that is no reason to show back "Application not available" messages to the end user through the browser. Such a scenario will create unsatisfied customers who might migrate to a competitor's service. We all agree that retaining an existing customer is as (or more) important than gaining a new customer.


One way to address software and hardware failure is to leverage the industry-leading clustering solutions to deliver best-in-class high availability, manageability, and performance for applications in enterprise and application service grid environments. If we don't want to go for such third-party clustering services, we can also see whether our own chosen application infrastructure (such as ServiceMix) will provide its own clustering features. So, let us understand the ServiceMix cluster in more detail.

A ServiceMix cluster is a logical grouping of multiple ServiceMix instances running simultaneously and working together thus providing increased scalability and reliability. From a client or a consumer perspective, the cluster is transparent. This means, a ServiceMix cluster appears to clients to be a single ServiceMix container instance. The ServiceMix instances that participate in a cluster can run on the same machine, or be located on different machines. You can also increase a ServiceMix cluster's capacity by adding additional ServiceMix instances to the cluster on an existing machine, or you can also add new machines to the cluster to host new ServiceMix server instances.


Individual components or application components need to be installed to the ServiceMix server instances taking part in the cluster. It is recommended to follow a uniform installation schema where we will deploy components homogenously into all the ServiceMix server instances in the cluster. However, this is not mandatory as is shown in the clustering sample we provide later in the chapter. The exact topology and deployment schema for a ServiceMix cluster has to be decided based on what level of QOS features we are targeting from each of the deployed components.

There are multiple benefits which we can leverage from a ServiceMix clustering topology, a few of them are listed as follows:

- **Scalability:** A software system is said to be scalable if its performance does not degrade significantly as the load on the system increases. The scalability of a ServiceMix cluster can be increased dynamically to meet the demand. You can add the ServiceMix instances to a cluster without interruption of the deployed service – the applications and services already deployed continue to run without impact to the existing consumers.
- **High availability:** Availability is defined as the fraction of time the software system is up and available to its consumers.



For example, a system with 99.99% availability over a period of 1 year
would be unavailable for:
 $(1 - 0.9999) \times 1 \text{ Year} \times 365 \text{ Days} \times 24 \text{ Hours per Day} \times 60 \text{ Minutes per Hour}$
=
52.56 Minutes.



For many systems such as the web-based e-commerce systems a 99.99% availability would be sufficient but for many other systems like those used for life saving or defence purposes a higher-level of availability would be required.

In a ServiceMix cluster, SEs and BCs can still continue to run in a different server instance when one of the server instances fails. As you can deploy the application components to multiple server instances in the cluster, if a server instance on which a component is running fails, another server instance on which that component is deployed can continue the processing thus increasing the overall system availability.

- **Load balancing:** Load balancing is the even distribution of jobs and processes across the computing and networking resources in your ServiceMix cluster. For load balancing to occur, there must be multiple copies of a ServiceMix component that can serve a particular consumer. Information about the location and operational status of all the ServiceMix components must be available centrally and across.

To set up a ServiceMix cluster, a JMS flow is used. The JMS flow collaborates the communication between more than one ServiceMix JBI container instance. A message queue is used for each JBI endpoint, so that multiple instances of the same named component deployed in different instances of the ServiceMix in the cluster have requests load balanced across them.

In a ServiceMix cluster, deployment happens in the same way as we do in a normal ServiceMix JBI container (both for POJO and archive Component deployment) but all ServiceMix container instances in the cluster are notified of a deployment. The underlying JMS flow will handle automatic routing, load balancing, and failover of MessageExchange(s) between the different ServiceMix containers instances in the cluster.

In the cluster mode all ServiceMix instances participating in the cluster must have a unique name in the whole cluster. Let us look into a sample configuration to understand this well. In our hypothetical sample cluster, assume that we will have three ServiceMix containers instances. As one of these ServiceMix instances also manages a JMS connection broker, we will arbitrarily name that instance with the name "admin".

```
<beans xmlns:sm="http://servicemix.apache.org/config/1.0">
  <bean id="propertyConfigurer"
        class="org.springframework.beans.factory.config.
                PropertyPlaceholderConfigurer">

  <bean id="jndi"
        class="org.apache.xbean.spring.jndi.
                SpringInitialContextFactory"
        factory-method="makeInitialContext" singleton="true" />
  <sm:container id="jbi"
        name="admin"
        flowName="jms?jmsURL=tcp://localhost:61616"
        useMBeanServer="true"
        createMBeanServer="true"
        rmiPort="1111">
    <sm:activationSpecs>
      <!-- other code -->
    </sm:activationSpecs>
  </sm:container>
  <bean id="broker"
        class="org.apache.activemq.xbean.BrokerFactoryBean">
    <property name="config" value="classpath:activemq.xml"/>
  </bean>
  <bean id="jmsFactory"
        class="org.apache.activemq.pool.PooledConnectionFactory">
    <property name="connectionFactory">
      <ref bean="connectionFactory"/>
    </property>
  </bean>
```

```
<bean id="connectionFactory"
      class="org.apache.activemq.ActiveMQConnectionFactory">
  <property name="brokerURL" value="tcp://localhost:61616" />
</bean>
</beans>
```

The important thing to note here is that, we first set the `flowName` to be of JMS type and then set the `jmsURL` to point to the location where the JMS broker is listening. Now, we can add any number of instances for the ServiceMix to the cluster, simply by pointing them to the same JMS broker. For our sample purpose we will have two instances for ServiceMix named "managed1" and "managed2".

The flow settings for managed1 are shown in the following code:

```
<beans xmlns:sm="http://servicemix.apache.org/config/1.0" >
  <bean id="jndi"
        class="org.apache.xbean.spring.jndi.
          SpringInitialContextFactory"
        factory-method="makeInitialContext" singleton="true" />
  <sm:container id="jbi"
                name="managed1"
                flowName="jms?jmsURL=tcp://localhost:61616"
                useMBeanServer="true"
                createMBeanServer="true">
    <sm:activationSpecs>
      <!-- other code -->
    </sm:activationSpecs>
  </sm:container>
</beans>
```

We will have a similar configuration for managed2. This is listed in the following code:

```
<beans xmlns:sm="http://servicemix.apache.org/config/1.0" >
  <bean id="jndi"
        class="org.apache.xbean.spring.jndi.
          SpringInitialContextFactory"
        factory-method="makeInitialContext" singleton="true" />
  <sm:container id="jbi"
                name="managed2"
                flowName="jms?jmsURL=tcp://localhost:61616"
                useMBeanServer="true"
                createMBeanServer="true">
    <sm:activationSpecs>
      <!-- other code -->
    </sm:activationSpecs>
  </sm:container>
</beans>
```

We will look into a clustering sample with code later in this chapter.

JMX

JMX (Java Management Extensions) technology provides the required APIs and tools for building distributed, web-based, modular, and dynamic solutions for managing and monitoring devices, applications, and service-driven networks. Starting with the J2SE platform 5.0, JMX technology is included in the Java SE platform.

The ServiceMix JBIContainer will expose internal services and components through the JMX. The JBIContainer can be passed as a JMXBeanServer. Alternatively, it can be configured to create one if it doesn't exist.

The following code will create a remote JMXConnector to the JBIContainer:

```
String jndiPath = "jmxrmi";
JMXServiceURL url = new JMXServiceURL
("service:jmx:rmi:///jndi/rmi://127.0.0.1:1099/" + jndiPath) ;
JMXConnector connector = JMXConnectorFactory.connect(url);
```

Configuring JMX in ServiceMix is done in the `jmx.xml` file again found in the `conf` directory. The following code shows how to do this:

```
<sm:jmxConnector objectName="connector:name=rmi"
    serviceUrl="${jmx.url}"
    threaded="true"
    daemon="true"
    depends-on="rmiRegistry, jndi"
    environment="#jmxConnectorEnvironment" />
```

The `jmx.url` is `service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi`.

These are the default settings for ServiceMix version 3.x. For version 2.x, `jmx.url` changes to `service:jmx:rmi:///jndi/rmi://localhost:1099/defaultJBIJMX`.

Once ServiceMix is up and running, then you can use any JMX compatible console tools to connect to the ServiceMix container. We will demonstrate this later in this chapter.

Sample Demonstrating Transaction

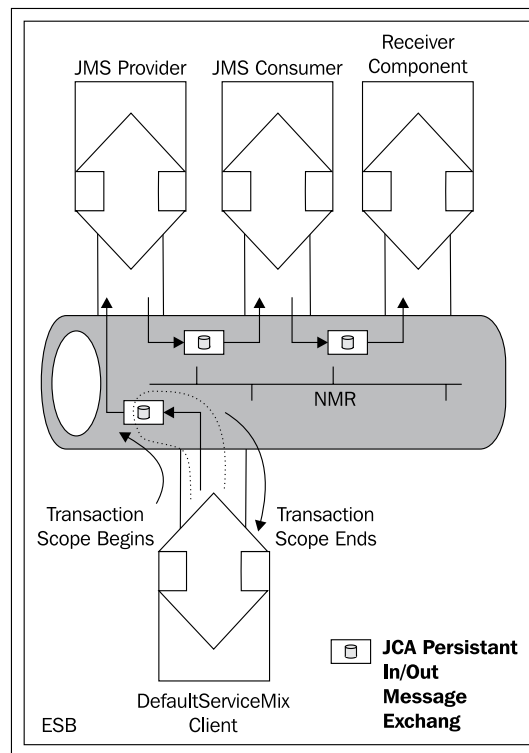
In this section we will demonstrate configuring transactions for a message exchange in an asynchronous pattern.

Sample Use Case

In the sample scenario to demonstrate transactions, we will configure `DefaultServiceMixClient` within the JBI container as the client or consumer for the JBI bus. We will first start a transaction and then ask `DefaultServiceMixClient` to send an `InOnly` message exchange to the JBI bus. We will use "send" here so that the act of sending the message will be enlisted in the current transaction, but the processing of the message exchange will be deferred and handled in a separate thread.

We will have a `servicemix-jms` component configured in the provider role to which the `DefaultServiceMixClient` can target message exchange. We then have a `servicemix-jms` in the consumer role to which the JMS provider pipelines messages. For the JMS consumer, we have configured a `Receiver` component using the `targetService` attribute. Hence any messages in the chain will be ultimately routed to the `Receiver` component.

The component setup is shown in the following figure:



In the sample, we use "send" for sending the message. Hence sending will be enlisted in the current transaction which includes reception of the message by the bus. Any further processing, including forwarding the message to the next component will be deferred and handled in separate threads.

Configure Transaction in ServiceMix

All the components specified in the selected use case are configured in the servicemixjms.xml file as shown here:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:sm="http://servicemix.apache.org/config/1.0"
       xmlns:jms="http://servicemix.apache.org/jms/1.0"
       xmlns:amq="http://activemq.org/config/1.0"
       xmlns:amqra="http://activemq.org/ra/1.0"
       xmlns:jencks="http://jencks.org/2.0"
       xmlns:test="http://binildas.com/esb/servicemix/tx/
                               jms/inonlyasync">

  <sm:container id="jbi"
               embedded="true"
               depends-on="jndi,broker"
               autoEnlistInTransaction="true"
               transactionManager="#transactionManager">

    <sm:flows>
      <sm:sedaFlow />
      <sm:jcaFlow connectionManager="#connectionManager"
                  jmsURL="tcp://localhost:61616?
                  jms.asyncDispatch=true&
                  jms.useAsyncSend=true" />
    </sm:flows>
    <sm:activationSpecs>
    <sm:activationSpec>
      <sm:component>
        <jms:component>
          <jms:endpoints>
            <jms:endpoint service="test:MyConsumerService"
                          endpoint="consumerEP"
                          targetService="test:
                          MyReceiverService"
                          role="consumer"
                          defaultMep="http://www.w3.org/2004/
                                  08/wsdl/in-only"
                          processorName="jca"
                          connectionFactory="#connectionFactory"
                          resourceAdapter="#resourceAdapter">
```

```
        bootstrapContext="#bootstrapContext"
        synchronous="false">
    <jms:activationSpec>
        <amqra:activationSpec
            destination="queue/A"
            destinationType="javax.jms.Queue" />
    </jms:activationSpec>
</jms:endpoint>
<jms:endpoint service="test:MyProviderService"
    endpoint="providerEP"
    role="provider"
    processorName="jca"
    connectionFactory="
        #connectionFactory"
    destinationStyle="queue"
    jmsProviderDestinationName=
        "queue/A" />

</jms:endpoints>
</jms:component>
</sm:component>
</sm:activationSpec>
<sm:activationSpec id="receiver"
    service="test:MyReceiverService"
    endpoint="receiverEP">
    <sm:component>
        <bean class="org.apache.servicemix.tck.
            ReceiverComponent" />
    </sm:component>
</sm:activationSpec>
</sm:activationSpecs>
</sm:container>
<bean id="client"
    class="org.apache.servicemix.client.
        DefaultServiceMixClient">
    <constructor-arg ref="jbi"/>
</bean>
<bean id="jndi"
    class="org.apache.xbean.spring.jndi.
        SpringInitialContextFactory"
    factory-method="makeInitialContext"
    singleton="true">
<property name="entries">
    <map>
        <entry key="jms/ConnectionFactory"
            value-ref="connectionFactory" />
    </map>
```

```

        </property>
    </bean>
    <amqra:managedConnectionFactory id="activemqMCF"
        resourceAdapter="#resourceAdapter" />
    <amqra:resourceAdapter id="resourceAdapter"
        serverUrl="tcp://localhost:61616?
           .jms.asyncDispatch=true&
           .jms.useAsyncSend=true"/>
    <jencks:connectionFactory id="connectionFactory"
        managedConnectionFactory="#activemqMCF"
        connectionManager="#connectionManager"/>
    <amq:broker id="broker" persistent="false">
        <amq:transportConnectors>
            <amq:transportConnector uri="tcp://localhost:61616" />
        </amq:transportConnectors>
    </amq:broker>
    <jencks:transactionManager id="transactionManager" />
    <jencks:workManager id="workManager"
        transactionManager="#transactionManager" />
    <jencks:bootstrapContext id="bootstrapContext"
        workManager="#workManager"
        transactionManager="#transactionManager"/>
    <jencks:connectionTracker id="connectionTracker"
        geronimoTransactionManager=
            "#transactionManager" />
    <jencks:poolingSupport id="poolingSupport"
        allConnectionsEqual="false" />
    <jencks:connectionManager id="connectionManager"
        containerManagedSecurity="false"
        transaction="xa"
        transactionManager="#transactionManager"
        poolingSupport="#poolingSupport"
        connectionTracker="#connectionTracker"
        />
</beans>

```

In order to leverage transactions with asynchronous message exchanges, the JCA flow must be used. This is what we do by including `jcaFlow` inside the `flows` element. While we configure the JBI container, if we set the `autoEnlistInTransaction` flag to `true`, each time a JBI exchange is sent, it will be enlisted in the current transaction.

Deploy and Run the Sample

Before running any samples in this chapter, if you haven't done it before edit `examples.PROPERTIES` (provided along with the code download for this chapter) and change the paths there to match your development environment.

Now to build the entire sample, it is easier to change directory to the top-level folder and execute the `build.xml` file provided there:

```
cd ch17\01_Transactions\InOnlyAsync
ant
```

This will build the entire codebase for the transaction demonstration. Now we need to open another command prompt and start ServiceMix in the embedded mode, as follows:

```
cd ch17\01_Transactions\InOnlyAsync
ant run
```

In case you want to run the test as a JUnit test case, execute the following code:

```
ant test
```

At the end of the run, we will attempt to close the JBI container by destroying the context. As we are starting the ServiceMix in embedded mode it will generate exceptions in the console. These errors can be ignored. Instead you can concentrate on the application logging which is reproduced here:

```
[junit] JmsInOnlyAsyncTest.testJmsInOnlySync - Start...
[junit] JmsInOnlyAsyncTest.testJmsInOnlySync - Transaction Committed.
[junit] INFO - MessageList - Waiting for message to
arrive
[junit] INFO - MessageList - End of wait for 1001
millis
[junit] JmsInOnlyAsyncTest.testJmsInOnlySync - End.
[junit] JmsInOnlyAsyncTest.tearDown...
[junit] Closing down the Spring ApplicationContext
```

We can see that the **MessageList** waits for messages which it receives at the end of **1001** milliseconds. The following screenshot shows the ESB console:

```

C:\> Command Prompt
D:\bin\com\java\com\binildas\esh\ServiceMix\Text\ESBUsingJ
BI-ServiceMix\ch17\01_Transactions\InOnlyAsync>ant test
Buildfile: build.xml

test:
[junit] Testsuite: com.binildas.esb.servicemix.tx.jms.in
onlyasync.JmsInOnlyAsyncTest
[junit] Tests run: 1, Failures: 0, Errors: 0, Time elaps
ed: 11.034 sec
[junit]
[junit] ----- Standard Output -----
[junit] JmsInOnlyAsyncTest.setUp...
[junit] JmsInOnlyAsyncTest.initAppContext...
[junit] INFO - JBIContainer - Service
Mix 3.1.1-incubating JBI Container <ServiceMix> is starting
[junit] INFO - JBIContainer - For hel
p or more informations please see: http://incubator.apache.o
rg/servicemix/
[junit] WARN - ManagementContext - Failed
to start rmi registry: internal error: ObjID already in use
[junit] WARN - ManagementContext - Failed
to start jmx connector: Cannot bind to URL [rmi://localhost:
1099/jmxrmi]: javax.naming.NameAlreadyBoundException: jmxrmi
[Root exception is java.rmi.AlreadyBoundException: jmxrmi]
[junit] INFO - ComponentMBeanImpl - Initial
izing component: #SubscriptionManager#
[junit] INFO - DeploymentService - Restori
ng service assemblies
[junit] INFO - ComponentMBeanImpl - Initial
izing component: ID:10.10.10.10-117cfb8c8a6-0:0
[junit] INFO - ComponentMBeanImpl - Initial
izing component: receiver
[junit] INFO - JBIContainer - Service
Mix JBI Container <ServiceMix> started
[junit] INFO - ComponentMBeanImpl - Initial
izing component: ID:10.10.10.10-117cfb8c8a6-0:1
[junit] INFO - ComponentMBeanImpl - Startin
g component: ID:10.10.10.10-117cfb8c8a6-0:1
[junit] JmsInOnlyAsyncTest.testJmsInOnlySync - Start...
[junit] JmsInOnlyAsyncTest.testJmsInOnlySync - Transacti
on Committed.
[junit] INFO - MessageList - Waiting
for message to arrive
[junit] INFO - MessageList - End of
wait for 1001 millis
[junit] JmsInOnlyAsyncTest.testJmsInOnlySync - End.
[junit] JmsInOnlyAsyncTest.tearDown...
[junit] Closing down the Spring ApplicationContext
[junit] INFO - JBIContainer - Deactiv
ating component #SubscriptionManager#
[junit] INFO - JBIContainer - Service
Mix JBI Container <ServiceMix> stopped
[junit] WARN - ActiveMQManagedConnection - Connect
ion failed: javax.jms.JMSEException: java.io.EOFException

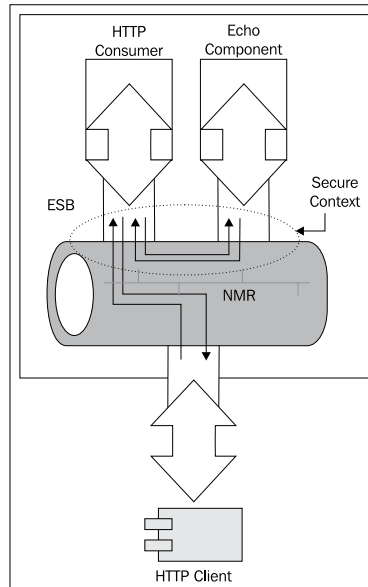
```

Sample demonstrating Security

The security in ServiceMix can be configured at multiple levels and at multiple layers. In this section we will demonstrate a simple security configuration for the servicemix-http component. We will configure HTTP basic authentication and as per that, when a client tries to access a resource marked as protected in the server, the server prompts for a username and password combination. This is followed by the browser prompting with a username and password entry form where the user can enter the relevant information and submit.

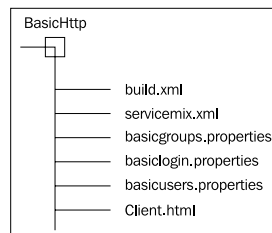
Sample Use Case

The setup of components for the sample use case is shown in the following figure:



The security sample use case will have `servicemix-http` configured in the consumer role, listening for HTTP transport for a particular port. We then configure an `Echo` service component as the `targetService` for the above HTTP consumer. When doing so, we also tell that the HTTP basic authentication has to be applied. An external HTTP client provided can be used to target messages to the HTTP consumer. When the external client sends requests, the server prompts for a username and password combination. This is followed by the browser prompting with a username and password entry form.

Configure Basic Authorization in servicemix-http



We will use a set of properties file and configuration files as shown above, for setting up the security sample.

The servicemix.xml file will host the main security configurations as shown here:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:sm="http://servicemix.apache.org/config/1.0"
       xmlns:http="http://servicemix.apache.org/http/1.0"
       xmlns:soap="http://servicemix.apache.org/soap/1.0"
       xmlns:test="http://binildas.com/esb/servicemix/security">
  <import resource="classpath:activemq.xml" />
  <import resource="classpath:jmx.xml" />
  <import resource="classpath:jndi.xml" />
  <import resource="classpath:tx.xml" />
  <import resource="classpath:security.xml" />
  <classpath>
    <location>./</location>
  </classpath>
  <bean id="propertyConfigurer"
        class="org.springframework.beans.factory.config.
              PropertyPlaceholderConfigurer">
    <property name="location"
              value="classpath:servicemix.properties" />
  </bean>
  <sm:systemProperties>
    <property name="properties">
      <map>
        <entry key="java.security.auth.login.config">
          <bean class="org.springframework.util.ResourceUtils"
                factory-method="getFile">
            <constructor-arg value="classpath:basiclogin.
                                properties"/>
          </bean>
        </entry>
      </map>
    </property>
  </sm:systemProperties>
  <sm:container id="jbi" rootDir="./wdir">
    <sm:broker>
      <sm:securedBroker>
        <sm:authorizationMap>
          <sm:authorizationMap>
            <sm:authorizationEntries>
              <sm:authorizationEntry service="*:*"
                                    roles="superuser" />
              <sm:authorizationEntry service="test:echo"
```



```
roles="secureuser" />
    </sm:authorizationEntries>
  </sm:authorizationMap>
</sm:authorizationMap>
</sm:securedBroker>
</sm:broker>
<sm:activationSpecs>
  <sm:activationSpec id="http">
    <sm:component>
      <http:component>
        <http:endpoints>
          <http:endpoint service="test:httpConsumer"
            endpoint="httpConsumer"
            targetService="test:echo"
            role="consumer"
            locationURI="http://localhost:
              8192/Service/"
            authMethod="basic"
            defaultMep="http://www.w3.org/
              2004/08/wsd/in-out">
            </http:endpoint>
          </http:endpoints>
        </http:component>
      </sm:component>
    </sm:activationSpec>
    <sm:activationSpec componentName="echo" service="test:echo">
      <sm:component>
        <bean class="org.apache.servicemix.components.
          util.EchoComponent" />
      </sm:component>
    </sm:activationSpec>
  </sm:activationSpecs>
</sm:container>
</beans>
```

Here, for the `http:endpoint` element you define the HTTP basic authentication by setting `authMethod="basic"`.

We can also plug-in authorization in the configuration. For this, we first define a secured broker (`sm:broker`) which can match the basic HTTP authenticated user against an Access Control List (ACL). To plug-in the ACL, we first set the system properties with key `"java.security.auth.login.config"` and value pointing to `basiclogin.properties`. `basiclogin.properties` is shown in the following code:

```

servicemix-domain
{
    org.apache.servicemix.jbi.security.login.PropertiesLoginModule
                                   required

    debug=true
    org.apache.servicemix.security.properties.user="basicusers.
                                                properties"
    org.apache.servicemix.security.properties.group="basicgroups.
                                                properties";
};

```

The `basicgroups.properties` defines various groups and maps which all users belong to which all groups. The `basicgroups.properties` is shown here:

```

superuser=manager
secureuser=binil

```

Now, the user credentials are stored in the `basicusers.properties` and are shown here:

```

system=manager
binil=binil
user1=user1

```

The secured broker puts the authorization rules as follows:

- As a default policy, let only the "superuser" role be allowed to access JBI endpoints.


```
<sm:authorizationEntry service="*:*" roles="superuser" />
```
- Let users with role "secureuser" be authorized to access `test:echo` service.


```
<sm:authorizationEntry service="test:echo" roles="secureuser" />
```

Deploy and Run the Sample

To build the entire sample, change directory to the `BasicHttp` folder and execute the `build.xml` file provided as follows:

```

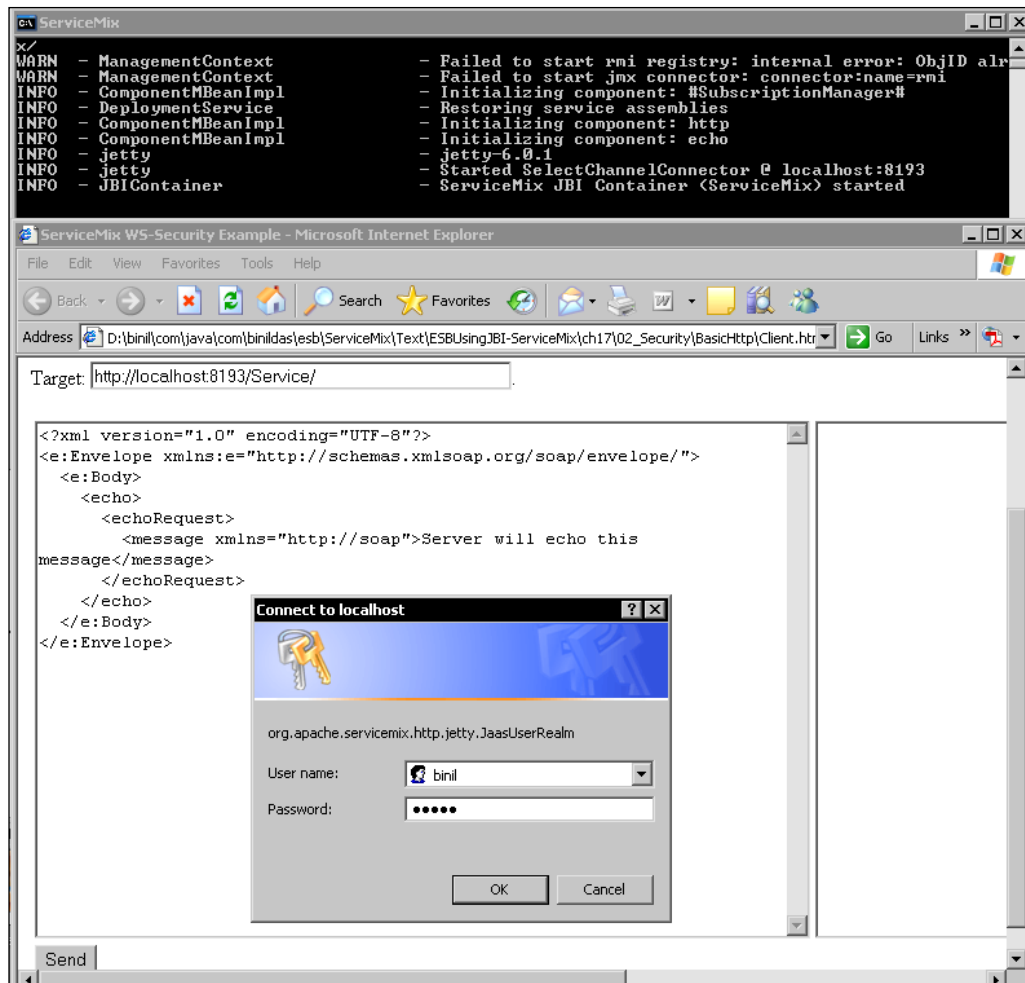
cd ch17\02_Security\BasicHttp
ant

```

Now bring the ServiceMix server up by executing the following command:

```
%SERVICEMIX_HOME%\bin\servicemix servicemix.xml
```

Once ServiceMix is up, execute `Client.html` file provided again in the same folder, and click the **send** button. The browser will prompt for the username and password entry form where you can enter the credentials `binil` and `binil` belonging to the `secureuser` group to access the service.



If you try to access the service with a different credential (for which the ACL permission is not granted), the service will not be accessible for you.

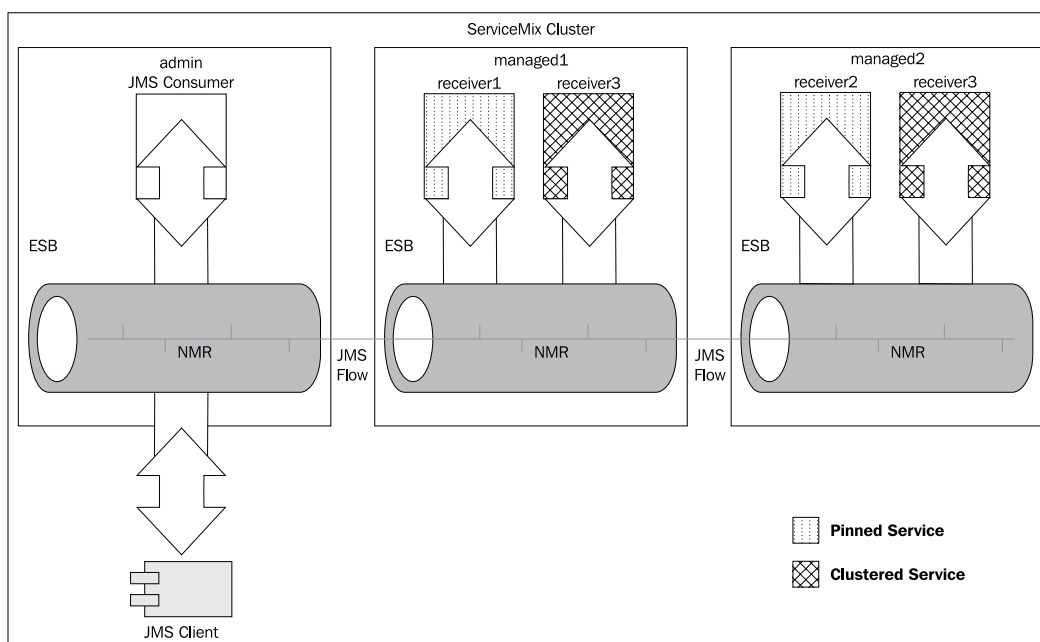
Sample Demonstrating Clustering

This section will demonstrate the clustering features provided by ServiceMix which we discussed earlier with the help of the running samples.

Sample Use Case

The sample scenario we use for clustering will consist of three ServiceMix instances configured in a cluster. For the sake of simplicity, all three instances will currently run in a single physical node, hence we are using the "localhost" as the server IP everywhere. However, it is possible to distribute these instances into different physical nodes in which case we may have to use the IP address of these nodes to form the cluster.

The following figure shows the cluster set up. Here, we have three different ServiceMix instances. Since in cluster mode, all the ServiceMix instances must have a unique name in the whole cluster. We will name the instances with different names namely `admin`, `managed1`, and `managed2`. One of these ServiceMix instances also manages a JMS connection broker, and hence we have arbitrarily named that instance with the name "admin".

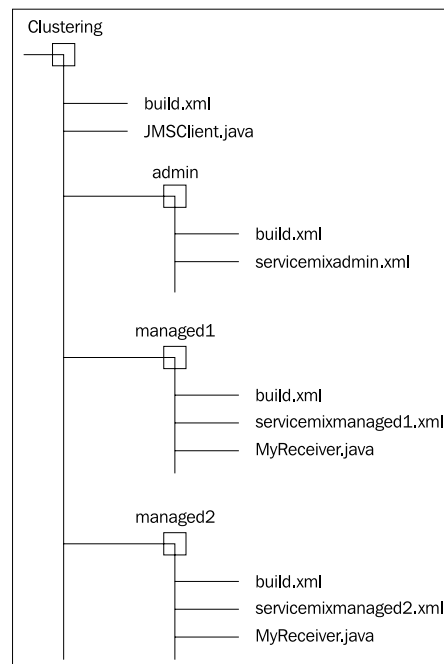


The JBI services can be deployed into any of the ServiceMix instances in this cluster. To deploy a service into the cluster, we will deploy those services into all (or many) instances in the cluster. When we do so, all the containers in the cluster are notified of the deployment. Now the cluster can administer routing or load balancing policies when it receives client requests by automatically routing `MessageExchange(s)` between the members of the cluster. Such services which we deploy across clusters are called **clustered services**. We can also deploy services into a single instance of the JBI container in the cluster in which case we call the service a **pinned service**.

For a clustered service, even if one of the ServiceMix instances hosting the service is down the service is still up since there are other cluster instances to serve the request. However for a pinned service we have to make sure that the server instance to which the service is pinned should be up, otherwise the service will cease to serve.

For the sample (refer to the previous figure) we will make use of both clustered and pinned services. As shown in figure, receiver3 is a clustered service since it is deployed in both the instances of the cluster. receiver1 and receiver2 are pinned services since they are deployed to managed1 and managed2 instances alone, respectively. The external JMS client will target the messages to all the above three services. We can see that for messages targeted to receiver1 and receiver2, they are always routed to managed1 and managed2 server instances respectively. However, for messages targeted to receiver3 the cluster will load balance the requests to any of the servers in the cluster. If by any chance we bring down any of the managed servers, then the pinned services in that instance will no longer be available, but all the subsequent requests for the clustered service will then onwards be routed to the other instance(s) of the server in the cluster alone.

The various artifacts used for the clustering demonstration are arranged within different subfolders as shown in the directory structure in the following figure:



Configure ServiceMix Cluster

For each of the ServiceMix server instances participating in the cluster, we have different server configuration files, and let us look at them one by one:

The admin server configuration is included in `ch17\03_Clustering\admin\servicemixadmin.xml`, which is shown in the following code.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:sm="http://servicemix.apache.org/config/1.0"
       xmlns:amq="http://activemq.org/config/1.0"
       xmlns:eip="http://servicemix.apache.org/eip/1.0"
       xmlns:jms="http://servicemix.apache.org/jms/1.0"
       xmlns:foo="http://servicemix.org/demo/">
  <bean id="propertyConfigurer"
        class="org.springframework.beans.factory.config.
              PropertyPlaceholderConfigurer">
    <property name="location"
              value="classpath:servicemix.properties" />
  </bean>
  <import resource="classpath:jmx.xml" />
  <import resource="classpath:jndi.xml" />
  <import resource="classpath:security.xml" />
  <import resource="classpath:tx.xml" />
  <import resource="classpath:activemq.xml" />
  <bean id="jndi"
        class="org.apache.xbean.spring.jndi.
              SpringInitialContextFactory"
        factory-method="makeInitialContext" singleton="true" />
  <sm:container id="jbi"
                name="admin"
                flowName="jms?jmsURL=tcp://localhost:61616"
                useMBeanServer="true"
                createMBeanServer="true"
                rmiPort="1111">
    <sm:activationSpecs>
      <sm:activationSpec>
        <sm:component>
          <jms:component>
            <jms:endpoints>
              <jms:endpoint service="test:MyConsumerService"
                            endpoint="myConsumer"
                            role="consumer"
                            soap="false"
                            targetService="foo:recipients"
                            defaultMep="http://www.w3.org/2004/
                                      08/wsdl/in-only"
                            destinationStyle="queue"
                            jmsProviderDestinationName="queue/A">
```

```

        connectionFactory=
            "#connectionFactory" />
    </jms:endpoints>
</jms:component>
</sm:component>
</sm:activationSpec>
<sm:activationSpec id="servicemix-eip">
    <sm:component>
        <eip:component>
            <eip:endpoints>
                <eip:static-recipient-list service="foo:recipients"
                    endpoint="endpoint">
                    <eip:recipients>
                        <eip:exchange-target
                            service="foo:receiver1" />
                        <eip:exchange-target
                            service="foo:receiver2" />
                        <eip:exchange-target
                            service="foo:receiver3" />
                    </eip:recipients>
                </eip:static-recipient-list>
            </eip:endpoints>
        </eip:component>
    </sm:component>
</sm:activationSpec>
</sm:activationSpecs>
</sm:container>
<bean id="broker"
    class="org.apache.activemq.xbean.BrokerFactoryBean">
    <property name="config" value="classpath:activemq.xml"/>
</bean>
<bean id="jmsFactory"
    class="org.apache.activemq.pool.PooledConnectionFactory">
    <property name="connectionFactory">
        <ref bean="connectionFactory"/>
    </property>
</bean>
<bean id="connectionFactory"
    class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL" value="tcp://localhost:61616" />
</bean>
</beans>
```

Here we have a JMS consumer through which the external JMS client can send messages to the NMR. Then we have configured JMS broker to be listening at localhost:61616. Moreover, we have also set the flow to be of type JMS by setting flowName="jms?jmsURL=tcp://localhost:61616".

The first managed server configuration is included in `ch17\03_Clustering\managed1\ servicemixmanaged1.xml`, which is listed in the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:sm="http://servicemix.apache.org/config/1.0"
  xmlns:amq="http://activemq.org/config/1.0"
  xmlns:eip="http://servicemix.apache.org/eip/1.0"
  xmlns:foo="http://servicemix.org/demo/">
  <classpath>
    <location>./</location>
    <location>./build</location>
  </classpath>
  <bean id="jndi"
    class="org.apache.xbean.spring.jndi.
      SpringInitialContextFactory"
    factory-method="makeInitialContext" singleton="true" />
  <sm:container id="jbi"
    name="managed1"
    flowName="jms?jmsURL=tcp://localhost:61616"
    useMBeanServer="true"
    createMBeanServer="true">
    <sm:activationSpecs>
      <sm:activationSpec componentName="receiver1"
        service="foo:receiver1">
        <sm:component>
          <bean xmlns="http://xbean.org/schemas/spring/1.0"
            class="managed1.MyReceiver" >
            <property name="name">
              <value>1</value>
            </property>
          </bean>
        </sm:component>
      </sm:activationSpec>
      <sm:activationSpec componentName="receiver3"
        service="foo:receiver3">
        <sm:component>
          <bean xmlns="http://xbean.org/schemas/spring/1.0"
            class="managed1.MyReceiver" >
            <property name="name">
              <value>3</value>
            </property>
          </bean>
        </sm:component>
      </sm:activationSpec>
    </sm:activationSpecs>
  </sm:container>
</beans>
```


Here we set name="managed1" and also set flowName="jms?jmsURL=tcp://localhost:61616". Then this managed server also will form a part of the same cluster joined by the admin server. Then we deploy receiver1 and receiver3 services to this server.

The second managed server configuration is included in ch17\03_Clustering\managed2\ servicemixmanaged2.xml, which is listed as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:sm="http://servicemix.apache.org/config/1.0"
       xmlns:amq="http://activemq.org/config/1.0"
       xmlns:eip="http://servicemix.apache.org/eip/1.0"
       xmlns:foo="http://servicemix.org/demo/">
  <classpath>
    <location>./</location>
    <location>./build</location>
  </classpath>
  <bean id="jndi"
        class="org.apache.xbean.spring.jndi.
          SpringInitialContextFactory"
        factory-method="makeInitialContext" singleton="true" />
  <sm:container id="jbi"
                name="managed2"
                flowName="jms?jmsURL=tcp://localhost:61616"
                useMBeanServer="true"
                createMBeanServer="true"
                rmiPort="1111">
    <sm:activationSpecs>
      <sm:activationSpec componentName="receiver3"
                        service="foo:receiver3">
        <sm:component>
          <bean xmlns="http://xbean.org/schemas/spring/1.0"
                class="managed2.MyReceiver" >
            <property name="name">
              <value>3</value>
            </property>
          </bean>
        </sm:component>
      </sm:activationSpec>
      <sm:activationSpec componentName="receiver2"
                        service="foo:receiver2">
        <sm:component>
          <bean xmlns="http://xbean.org/schemas/spring/1.0"
                class="managed2.MyReceiver" >
            <property name="name">
              <value>2</value>
            </property>
          </bean>
        </sm:component>
      </sm:activationSpec>
    </sm:activationSpecs>
  </sm:container>
</beans>
```

```

        </sm:component>
    </sm:activationSpec>
</sm:activationSpecs>
</sm:container>
</beans>

```

Here the main difference is that we set `name="managed2"`. Moreover we set `flowName="jms?jmsURL=tcp://localhost:61616"` so that this managed server too will form a part of the same cluster joined by the admin and previous managed server. Then we deploy `receiver2` and `receiver3` services to this server.

So in a nutshell, `receiver1` and `receiver2` are pinned services whereas `receiver3` is a clustered service.

Deploy and run the sample

To build the entire sample, it is easier to change directory to the top-level folder and execute the `build.xml` file provided there:

```

cd ch17\03_Clustering
ant

```

This will build the entire codebase for the clustering demonstration. Now we need to take three different command prompts and bring up all the server instances in the cluster, in the same order as shown as follows:

```

cd ch17\03_Clustering\admin
%SERVICEMIX_HOME%\bin\servicemix servicemixadmin.xml
cd ch17\03_Clustering\managed1
%SERVICEMIX_HOME%\bin\servicemix servicemixmanaged1.xml
cd ch17\03_Clustering\managed2
%SERVICEMIX_HOME%\bin\servicemix servicemixmanaged2.xml

```

The cluster should be up by now. Now to test the cluster setup, in a different command prompt execute the following:

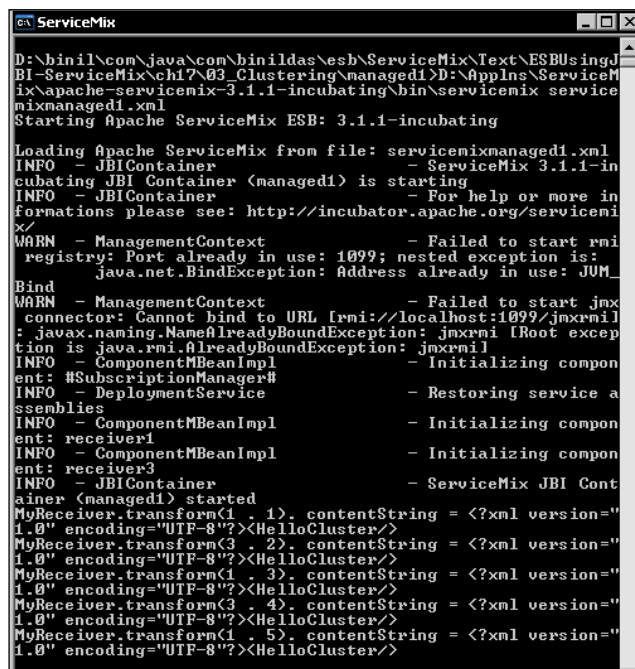
```

cd ch17\03_Clustering
ant run

```

Keep watching on the server-side console printouts, especially the console of the `managed1` and `managed2` servers. Execute `ant run` a couple of times and understand how the cluster load balances requests targeted to the different services.

The managed1 server console is shown in the following screenshot:

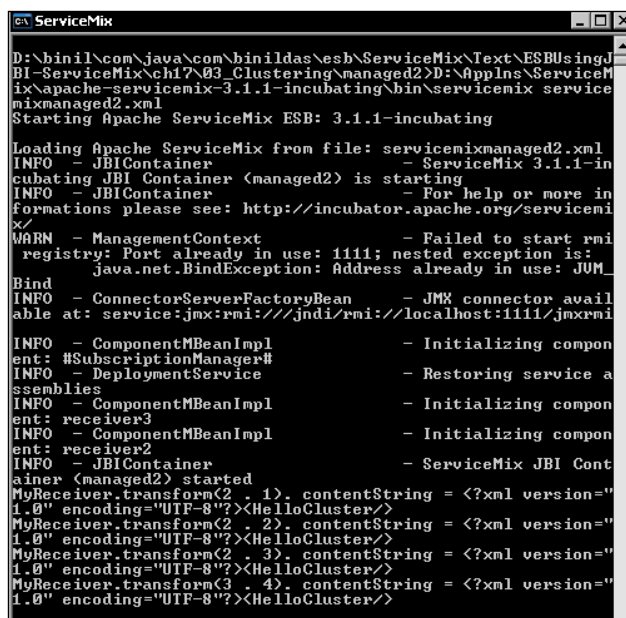


```

D:\bin\com\java\com\bin\ildas\esh\ServiceMix\Text\ESBUsingJ
BI-ServiceMix\chl7\03_Clustering\managed1>D:\Applns\ServiceM
ix\apache-service-mix-3.1.1-incubating\bin\service-mix service
mixmanaged1.xml
Starting Apache ServiceMix ESB: 3.1.1-incubating
Loading Apache ServiceMix from file: servicemixmanaged1.xml
INFO - JBIContainer - ServiceMix 3.1.1-in
cubating JBI Container <managed1> is starting
INFO - JBIContainer - For help or more in
formations please see: http://incubator.apache.org/service-m
ix/
WARN - ManagementContext - Failed to start rmi
registry: Port already in use: 1099; nested exception is:
java.net.BindException: Address already in use: JUM_
Bind
WARN - ManagementContext - Failed to start jmx
connector: Cannot bind to URL [rmi://localhost:1099/jmxrmi]
: javax.naming.NameAlreadyBoundException: jmxrmi [Root excep
tion is java.rmi.AlreadyBoundException: jmxrmi]
INFO - ComponentMBeanImpl - Initializing compon
ent: #SubscriptionManager#
INFO - DeploymentService - Restoring service a
sssemblies
INFO - ComponentMBeanImpl - Initializing compon
ent: receiver1
INFO - ComponentMBeanImpl - Initializing compon
ent: receiver2
INFO - JBIContainer - ServiceMix JBI Cont
ainer <managed1> started
MyReceiver.transform(1 . 1). contentString = <?xml version="
1.0" encoding="UTF-8"?><HelloCluster/>
MyReceiver.transform(3 . 2). contentString = <?xml version="
1.0" encoding="UTF-8"?><HelloCluster/>
MyReceiver.transform(1 . 3). contentString = <?xml version="
1.0" encoding="UTF-8"?><HelloCluster/>
MyReceiver.transform(3 . 4). contentString = <?xml version="
1.0" encoding="UTF-8"?><HelloCluster/>
MyReceiver.transform(1 . 5). contentString = <?xml version="
1.0" encoding="UTF-8"?><HelloCluster/>

```

The managed2 server console is shown in the following screenshot:



```

D:\bin\com\java\com\bin\ildas\esh\ServiceMix\Text\ESBUsingJ
BI-ServiceMix\chl7\03_Clustering\managed2>D:\Applns\ServiceM
ix\apache-service-mix-3.1.1-incubating\bin\service-mix service
mixmanaged2.xml
Starting Apache ServiceMix ESB: 3.1.1-incubating
Loading Apache ServiceMix from file: servicemixmanaged2.xml
INFO - JBIContainer - ServiceMix 3.1.1-in
cubating JBI Container <managed2> is starting
INFO - JBIContainer - For help or more in
formations please see: http://incubator.apache.org/service-m
ix/
WARN - ManagementContext - Failed to start rmi
registry: Port already in use: 1111; nested exception is:
java.net.BindException: Address already in use: JUM_
Bind
INFO - ConnectorServerFactoryBean - JMX connector avail
able at: service:jmx:rmi:///jndi/rmi://localhost:1111/jmxrmi
INFO - ComponentMBeanImpl - Initializing compon
ent: #SubscriptionManager#
INFO - DeploymentService - Restoring service a
sssemblies
INFO - ComponentMBeanImpl - Initializing compon
ent: receiver3
INFO - ComponentMBeanImpl - Initializing compon
ent: receiver2
INFO - JBIContainer - ServiceMix JBI Cont
ainer <managed2> started
MyReceiver.transform(2 . 1). contentString = <?xml version="
1.0" encoding="UTF-8"?><HelloCluster/>
MyReceiver.transform(2 . 2). contentString = <?xml version="
1.0" encoding="UTF-8"?><HelloCluster/>
MyReceiver.transform(2 . 3). contentString = <?xml version="
1.0" encoding="UTF-8"?><HelloCluster/>
MyReceiver.transform(3 . 4). contentString = <?xml version="
1.0" encoding="UTF-8"?><HelloCluster/>

```

We can see that the request routed to `receiver3` (which is a clustered service) is load balanced whereas the requests routed to `receiver1` and `receiver2` (which are pinned services) are always served by their respective pinned servers.

You may also want to kill one of the managed servers and try the effect of that on new incoming requests. You can later bring this dead server back to join the cluster without disturbing the cluster setup.

Sample demonstrating JMX

To demonstrate JMX in ServiceMix, we will use the same sample we used for Chapter 9 (*Pojo Binding Using Jsr181*). The sample `ch09\Jsr181BindPojo` is repeated in this chapter and is kept in folder `ch17\04_JMX`.

Enable JMX in ServiceMix Application

ServiceMix uses the following parameter for enabling JMX:

The default `namingPort`: 1099.

The default container name: `jmxrmi`.

The JMX Service URL: `service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi`.

These are the default settings for ServiceMix version 3.x. For version 2.x, JMX Service URL alone changes to: `service:jmx:rmi:///jndi/rmi://localhost:1099/defaultJBIJMX`.

The above values are configured through `%SERVICEMIX_HOME%\conf\jmx.xml`. To start simple, edit the `jmx.xml` file to disable the security feature. This can be done by first searching for "Comment the following lines to disable JAAS authentication for jmx" and then commenting the succeeding lines.

Now, bring ServiceMix up. This can be done by trying out any of the samples in the previous chapters or you can use the JMX sample provided with this chapter. To do that, change directory to `ch17\04_JMX` which contains a top-level `build.xml` file. Execute `ant` there.

```
cd ch17\04_JMX
ant
```

Now, bring up the ServiceMix container by executing the `servicemix.xml` file contained in the same folder.

```
%SERVICEMIX_HOME%\bin\servicemix servicemix.xml
```

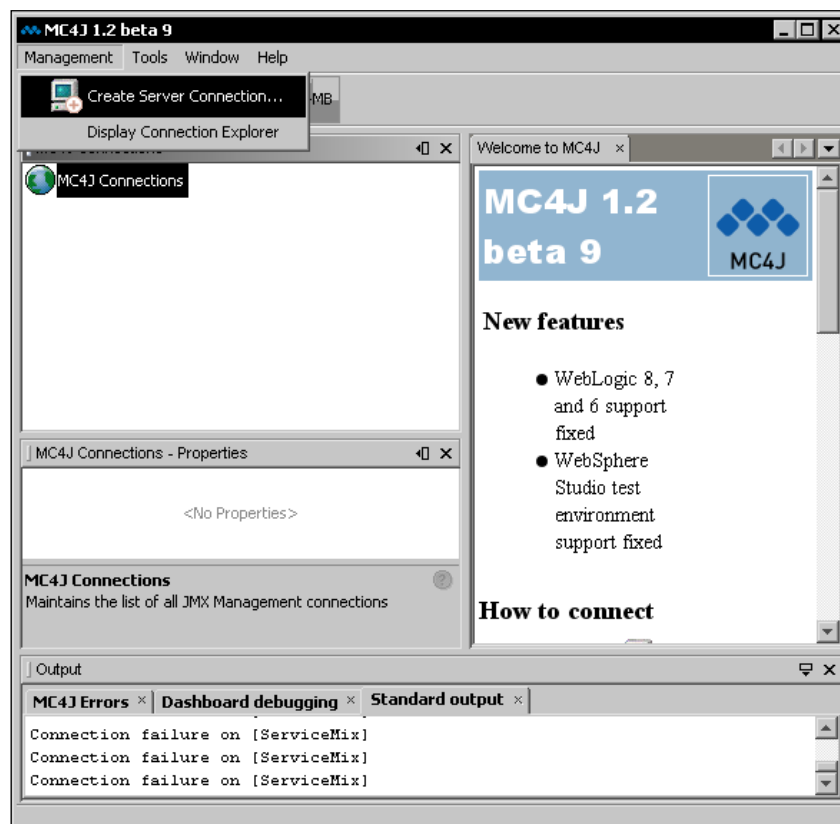
Initialize JMX Console—MC4J

You can use any of your favorite JMX tools to control the ServiceMix application. The Java-2 Platform, Standard Edition (J2SE) 5.0 release onwards includes a JMX monitoring tool, JConsole. To bring JConsole up, execute the following commands:

```
cd %JAVA_HOME%\bin
jconsole
```

For our demonstration we will use MC4J provided by SourceForge. MC4J is the management software for J2EE application servers and other Java applications. It utilizes the JMX specification to connect to and introspect to the information within the supported servers and applications. It provides the ability to browse the existing managed beans (MBeans), update configurations, monitor operations, and execute tasks.

Click on the "MC4J Console 1.2b9.exe" file found in the top-level directory of the MC4J installation to bring up the MC4J window. This MC4J window is shown in the following screenshot:



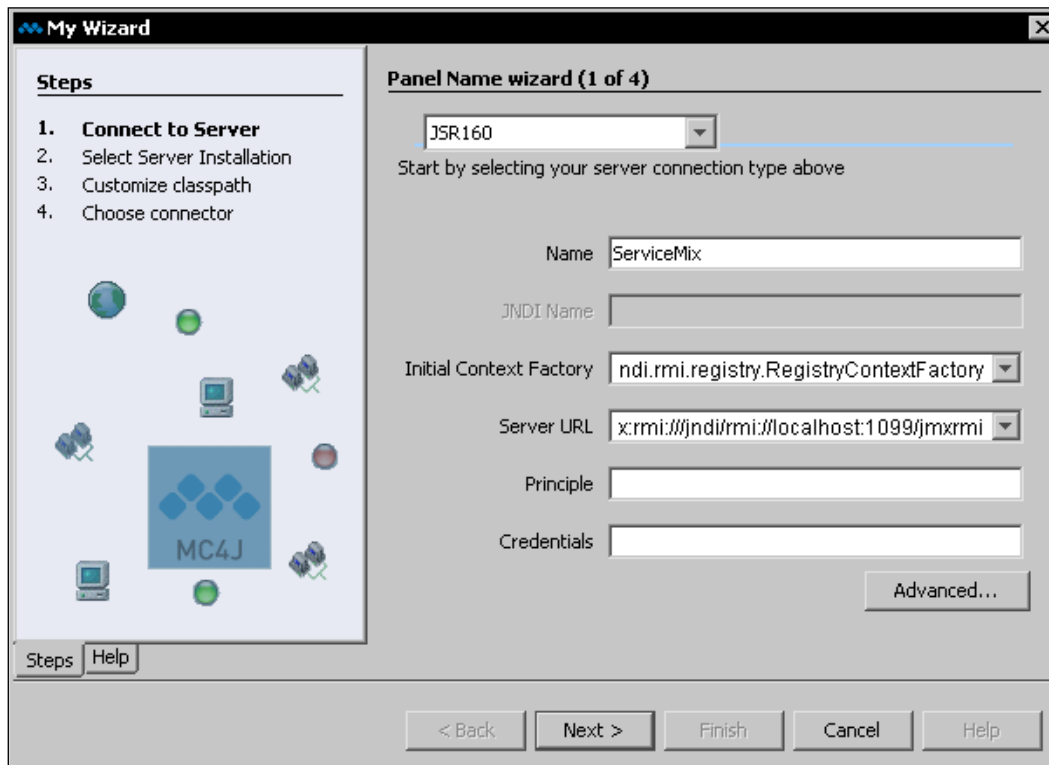
Select **Management | Create Server Connection...** from the menu. This will start **My Wizard**. A connection to ServiceMix can be created using the wizard.

In the wizard, enter the following into the text boxes and pull-down menus:

Select your server connection type as **JSR160**.

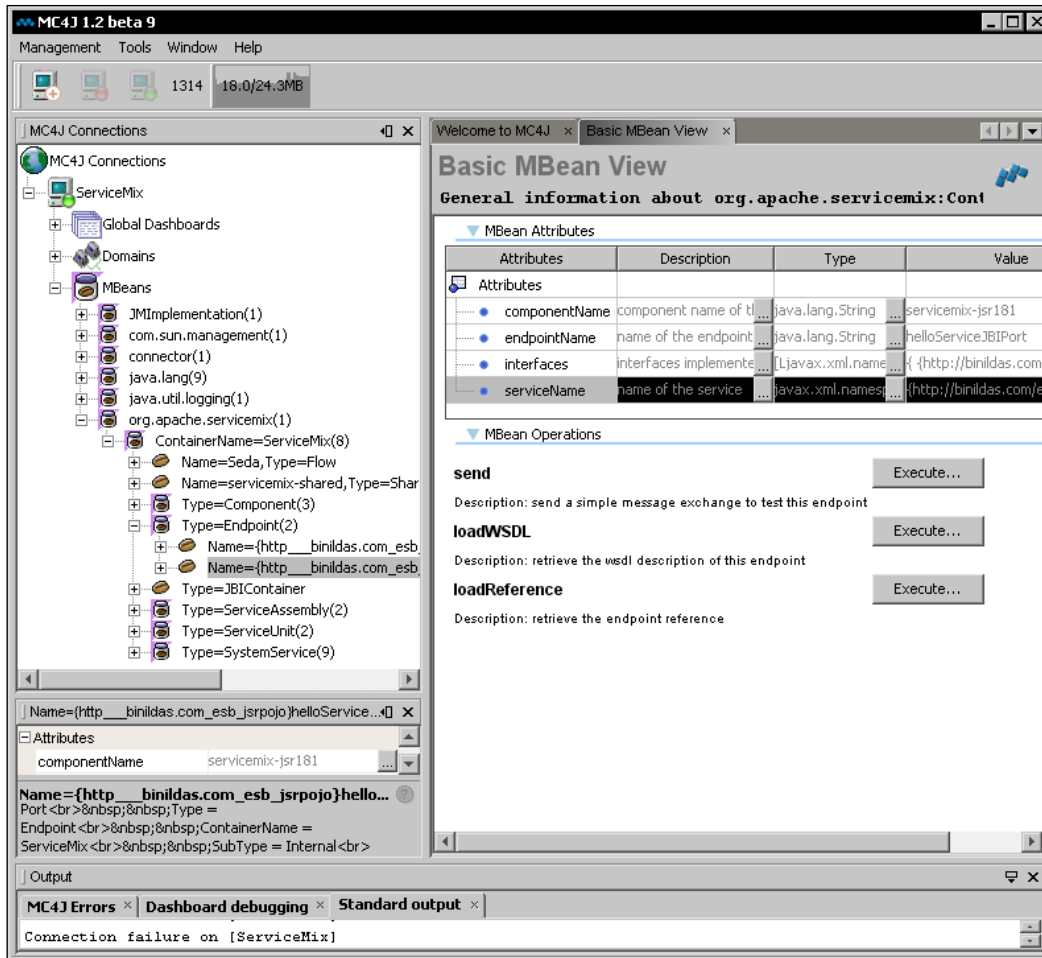
The **Name** textbox is filled with any name, for example, **ServiceMix**.

Select **Server URL** as **service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi**.



Accept the defaults for the rest of the fields in the wizard and click the **Next>** button. Now click **Finish** in the next window. This will make a connection to ServiceMix!

Click on **org.apache.servicemix(1)**. The components of the POJO binding example will be shown. Right-click on a component and select **Available dashboards...** | **Basic MBean View** to see the JMX information available as shown in the following screenshot:



Retrieve WSDL through JMX

At times when you try out your samples in ServiceMix you may feel that something is not working properly or some information retrieved is not as per your expectation. In such scenarios it makes sense to make use of a JMX console and look into the component configurations. Let us do a similar activity now.

In the POJO binding sample you have already wired the service interface and service implementation as follows:

```
<jsr181:endpoint annotations="none"
    service="test:helloService"
    serviceInterface="samples.HelloServiceBI">

  <jsr181:pojo>
    <bean class="samples.HelloServicePojo">
      </bean>
    </jsr181:pojo>
  </jsr181:endpoint>
```

Let us now load the WSDL generated by the JBI bus by clicking on the **loadWSDL** button. You may have to uncheck the **View as HTML** checkbox to make the WSDL visible, which is shown in the following figure:



Summary

I hope you have enjoyed this last chapter. The samples demonstrated here, especially the usage of the JMX tool to look into the artifacts inside the bus will be highly useful when you run your own components inside the ServiceMix JBI bus.

If you have gone through all the previous chapters also in this text, by now you should have "hands on" knowledge of what an ESB is and what JBI has to do in defining ESB-based architectures. This will give you an edge over your peers in understanding ESB and in preparing yourselves to leverage ESB to solve your integration problems.

In my entire career I have interacted with a lot of people who talk and write a lot on ESB in white papers, but they never gave me any code to play around with. It is in this context I decided to spend a few chapters on ESB with code. I hope you enjoyed reading just as I enjoyed writing it.

Index

A

Apache SOAP

- about 85
- artifacts 86
- binding services 86
- checkMustUnderstands attribute, service element 87
- encoding styles 85
- message oriented 86
- provider element 87
- RPC oriented 85
- service element 86
- type attribute, provider element 87

Apache SOAP binding 83, 84

Artix 64

autonomy 262

B

binding

- about 83
- Apache SOAP binding 84
- endpoints 84

bindings, XFire

- Aegis 100
- Castor 100
- JAXB 100
- Message 100
- XMLBeans 100

C

Celtix

- about 63
- objectives 63, 64

ChainBuilder 64

CIM 22

clustering, QOS features

- about 373
- application deployment stack, layers 373
- sample 388
- ServiceMix clustering, features 374

clustering, sample

- deploying 395
- running 395, 397
- sample use case 389, 390
- ServiceMix cluster, configuring 391, 392, 393, 394, 395

Common Object Request Broker Protocol.

See CORBA

Communication Service Provider (CSP) 347

component helper classes, ServiceMix

- MessageExchangeListener 137
- TransformComponentSupport 137, 139

components, EAI patterns

- configuring, in ServiceMix 294
- content-based router 294
- content enricher 303
- message filter 323
- pipeline 334
- split aggregator 329
- static recipient list 313
- static routing slip 339
- wiretap 319
- XPath splitter 308

components, ServiceMix

- httpConnector 79
- httpGetData 79
- httpInvoker 79
- httpReceiver 79
- servicemix-jsr181 162
- trace, configuring 79

- quartz 79
- component versus services**
 - about 147, 148
 - EJB components with services, coexistence 148
 - technical indiscrimination 148
- consumer configuration parameters, servicemix-http**
 - defaultMEP 186
 - endpoint 186
 - interfaceName 186
 - locationURI 186
 - role 186
 - service 186
 - soap 186
 - targetEndpoint 186
 - targetService 186
 - wSDLResource 186
- consumer configuration parameters, service-mix-jms**
 - connectionFactory 205
 - defaultMEP 205
 - destinationStyle 205
 - endpoint 204
 - jmsProviderDestinationName 205
 - role 204
 - service 204
 - soap 204
 - targetService 205
 - useMsgIdInResponse 205
- content-based router, EAI patterns components**
 - about 269, 294
 - configuring 297
 - content-based router component, sample use case 297
 - definition 295
 - illustrative design, Acme company 295
 - JMS client component, sample use case 296
 - JMS consumer component, sample use case 296
 - JMS provider component, sample use case 297
 - receiver component, sample use case 297
 - sample, deploying 301
 - sample, running 301
 - sample code 297-301
 - sample use case 296
 - symbol 294
- content enricher, EAI patterns components**
 - about 303
 - configuring 305
 - content appender component, sample use case 304
 - content enricher component, sample use case 304
 - definition 303
 - illustrative design, Acme company 303
 - JMS client component, sample use case 304
 - JMS consumer component, sample use case 304
 - JMS provider component, sample use case 304
 - sample, deploying 307
 - sample, running 307
 - sample code 305, 306, 307
 - sample use case 304
 - symbol 303
- CORBA 12**
- covenant approach**
 - about 268, 269
 - advantages 269
- E**
- EAI 12**
- EAI pattern blocks 292**
- EAI patterns**
 - about 290
 - aggregator, notations 291
 - book 290
 - components 294
 - notations 291
 - ServiceMix, need for 291
 - site 291
 - splitter, notations 291
 - wire tap, notations 291
- EIP**
 - about 289
 - EAI patterns 290
- EJB-SOAP sample**
 - Apache SOAP binding, code listing 90, 91
 - client, running 93, 94
 - code listing 89

- EJB, binding to SOAP 92, 93
- EJB, deploying 91, 92
- running 91
- ServiceMix, relating to 96
- session EJB, code listing 89
- scenario 88
- EJB-XFire sample**
 - classes 115-119
 - client, running 120
 - code listing 115
 - EJB, deploying 119, 120
 - running 119, 120
 - sample scenario 114
- EJB resources**
 - reconciling 160
- EJB sample, binding**
 - Axis client-side stubs, generating 158-160
 - Axis client code base, building 160
 - EJB, binding to ServiceMix 150-154
 - EJB binding, deploying in ServiceMix 155
 - EJB service, defining 149
 - EJB service, deploying 150
 - WSDL, accessing 156, 157
- Enterprise Application Integration.** *See* **EAI**
- enterprise integration**
 - about 8
 - issues 8
- Enterprise Integration Patterns.** *See* **EIP**
- enterprise message bus 15, 17**
- Enterprise Service Bus.** *See* **ESB**
- ESB**
 - about 16
 - abstraction, beyond interface 24
 - business concerns 20
 - channels, for interoperability 20, 21
 - data redundancy 22
 - EIP 289
 - features 17
 - functionalities 17
 - industry adoption 19
 - issues 20
 - linked servers 28
 - linked services 28
 - new systems, addition 22
 - service 23, 24
 - service aggregation 25
 - service consolidation 26
 - service enablement 26
 - service reuse 23
 - services fabric 30
 - service sharing 27, 28
 - service virtualization 29, 30
 - system management 23
 - system monitoring 23
 - versus message bus 17
 - versus message bus, differences 18
 - versus message bus, similarities 17
 - volatile interfaces 22
- ESB integration**
 - about 16
 - features 16
- external web service invoking sample, JBI proxy**
 - Axis generated client-side stubs 253
 - deploying 258
 - deployment, configuring 258
 - HelloWebService.wsdl 253
 - IFHello business interface 252
 - IFHello.java 252
 - IFHelloProxy.java 256
 - IFHelloProxyService.java 256
 - IFHelloWeb.java 252
 - ITarget.java 255
 - JBI proxy binding, XBean-based 256, 258
 - running 258
 - TargetService.java 255
 - web service code listing 252
 - WSDL, generating 259
- G**
- Global Assembly Cache (GAC) 261**
- H**
- HTTP**
 - about 182
 - headers 182
 - web service request header, sample 182, 183
- I**
- installing, ServiceMix**
 - classpath issues, resolving 67, 68

- hardware requirements 65
- in Unix 67
- in Windows 66
- OS requirements 65
- run-time environment 65
- ServiceMix, configuring 67
- ServiceMix, starting 67
- ServiceMix, stopping 67
- integration 11, 12**
- integration architectures**
 - about 12
 - enterprise message bus integration 15
 - enterprise message bus integration, diagrammatic representation 15
 - enterprise service bus integration, diagrammatic representation 16
 - ESB integration 16
 - hub-and-spoke architecture 13
 - hub-and-spoke architecture, diagrammatic representation 14
 - hub-and-spoke architecture, drawbacks 14
 - hub-and-spoke architecture, features 13
 - message broker 13
 - point-to-point integration 13
 - point-to-point integration, diagrammatic representation 13
- issues, enterprise integration**
 - autonomous system 9
 - data duplication 8, 9
 - intranet versus internet 10
 - multiple systems 8
 - trading partners system 10
- J**
- J2EE**
 - components 37
 - JB1 36
- J2EE Connector Architecture. *See* JCA**
- Java-2 Platform Standard Edition (J2SE) 398**
- Java 2 Enterprise Edition. *See* J2EE**
- Java API for XML Binding. *See* JAXB**
- Java XML binding**
 - about 222
 - Castor, frameworks 222
 - frameworks 222
 - JAXB 223
 - sample binding, XStream used 224
 - XMLBeans, frameworks 222
 - XStream 223
- JAXB**
 - about 223
 - features 223
- JB1**
 - about 38
 - in J2EE 36
 - JCA, competing with 38
 - JSR 208 39
 - message exchange patterns 47
 - nomenclature 40
 - provider-consumer contract 42
- JB1-POJO binding sample**
 - about 164
 - Axis client-side stubs, generating 172
 - Axis client codebase, building 173
 - deploying 167, 168
 - POJO class 164, 166
 - POJO code listing 166
 - running 169
 - use case 164
 - WSDL, accessing 169, 171
 - XBean-based POJO binding 166, 167
- JB1 bus, accessing**
 - components, for implementing sample use case 175
 - sample 173
 - sample, building 179
 - sample, deploying 179
 - sample, running 179
 - sample code listing 177, 178
 - sample use case 175
- JB1 compliant container**
 - Artix 64
 - Celtix 63
 - ChainBuilder 64
 - Mule 63
 - PEtALS 64
- JB1 components**
 - binding components 41
 - delivery channel 42
 - JB1 container 41
 - JB1 environment 40
 - normalized message 41
 - normalized message router 41

- pluggable components 42
 - service consumers 42
 - service engine 41
 - service providers 42
 - JBIs components, ServiceMix**
 - developing 135
 - HttpInterceptor component, building 144
 - HttpInterceptor component, coding 140, 141
 - HttpInterceptor component, configuring 142
 - HttpInterceptor component, deploying 143
 - HttpInterceptor component, packaging 142
 - HttpInterceptor component, running 144
 - JBIs container**
 - about 41
 - ServiceMix 57
 - JBIs Proxy sample**
 - compatible interface, implementing 244
 - in-compatible interface, implementing 248
 - JBIs Proxy sample, compatible interface**
 - deploying 247
 - deployment, configuring 247
 - EchoProxyService.java 245
 - IEcho.java 245
 - IEcho interface 245
 - JBIs proxy binding, XBean used 246
 - proxy code listing 245
 - running 247
 - TargetService.java 246
 - JBIs Proxy sample, in-compatible interface**
 - deploying 251
 - deployment, configuring 251
 - EchoProxyService.java 249
 - IEcho.java 249
 - ITarget.java 249
 - JBIs proxy binding, XBean used 250
 - proxy code listing 248
 - running 251
 - TargetService.java 249
 - JCA 37**
 - JDK Proxy class**
 - about 239
 - getProxyClass, utility methods 239
 - InvocationHandler 239
 - InvocationHandler, implementing 240
 - newProxyInstance, utility methods 239
 - sample 240-242
 - sample, building 243
 - utility methods 239
 - JMS**
 - about 199
 - ServiceMix 203
 - JMS API**
 - about 199
 - J2EE, enhancing ways 200
 - JMX, QOS features**
 - about 377
 - configuring, in ServiceMix 377
 - JConsole tool 398
 - sample 397
 - JMX, sample**
 - enabling, in ServiceMix 397
 - JMX console, initializing 398, 399
 - WSDL, retrieving 400, 401
 - JSR181 101, 162**
 - JSR 208**
 - about 39
 - abstract business process 39
 - business protocol 39
- ## L
- lightweight JBIs components, ServiceMix**
 - about 69
 - cache 69
 - component helper classes 69
 - drools 69
 - email 69
 - file 69
 - FTP 69
 - Groovy 69
 - HTTP 69
 - Jabber 69
 - JAX WS 69
 - JCA 69
 - JMS 69
 - Quartz 70
 - reflection 70
 - RSS 70
 - SAAJ 70
 - scripting 70
 - validation 70
 - VFS 70

WSIF 70
XFire 70
XSLT 70
XSQL 70

Line of Business. *See* **LOB**
LOB 7

M

MC4J 398

MEP 205

message exchange patterns, JBI

- In-Only MEP 48
- In-Only MEP, normal scenario 48
- In-Optional-Out MEP 52
- In-Optional-Out MEP, consumer fault scenario 54
- In-Optional-Out MEP, one way scenario 52
- In-Optional-Out MEP, provider fault scenario 53
- In-Optional-Out MEP, two way scenario 52, 53
- In-Out MEP 50
- In-Out MEP, fault scenario 51
- In-Out MEP, normal scenario 50
- Robust In-Only MEP 48
- Robust In-Only MEP, fault scenario 49, 50
- Robust In-Only MEP, normal scenario 48
- service invocations 47

message filter, EAI patterns components

- about 323
- configuring 326
- definition 324
- illustrative design, Acme company 324
- JMS client component, sample use case 325
- JMS consumer component, sample use case 325
- message filter component, sample use case 325
- receiver component, sample use case 325
- sample, deploying 328
- sample, running 328
- sample code 326, 327
- symbol 324

Message Oriented Middleware. *See* **MOM**
MOM. *See* **JMS**

MOM

- about 19
- Microsoft Biztalk 292
- Microsoft MQ 292
- TIBCO 292
- Webmethods 292
- WebSphere MQ 292

Mule 63

multiple endpoint address approach 269

N

NMR flow types, ServiceMix

- JCA flow 62
- JMS flow 61
- SEDA flow 60
- ST flow 59

O

Order Management System (OMS) 347

P

packaging and deployment sample, ServiceMix

- component development, phase 128
- component packaging, phase 129-131
- phases 127
- running 132, 134

patterns 291

PEtALS 64

pipeline, EAI patterns components

- about 334
- configuring 337
- definition 335
- echo component, sample use case 336
- illustrative design, Acme company 335
- JMS client component, sample use case 336
- JMS consumer component, sample use case 336
- JMS provider component, sample use case 336
- pipeline component, sample use case 336
- sample, deploying 339
- sample, running 339
- sample code 337, 338

- sample use case 336
- symbol 334

POJO

- about 161
- advantages 162
- overview 161

POJO class 164

protocol bridge 207, 208

provider-consumer contract, JBI

- detached message exchange 44
- message exchange 47
- provider-consumer, responsibilities 46
- provider-consumer role 45
- service invocation 47
- WSDL representation of service 43

provider configuration parameters,

servicemix-http

- endpoint 187
- interfaceName 187
- locationURI 187
- role 187
- service 187
- soap 188
- soapAction 188
- wsdlResource 188

provider configuration parameters,

servicemix-jms

- connectionFactory 206
- destinationStyle 206
- endpoint 206
- jmsProviderDestinationName 206
- role 206
- service 206
- soap 206

proxy

- about 238
- design pattern 238

Q

QOS 11

QOS features

- clustering 373
- JMX 377
- security 371
- transactions 368

Quality of Service. *See* QOS

S

security, QOS features

- about 371
- HTTP basic authentication 371
- HTTP basic authentication, configuring 371
- HTTP basic authentication, drawback 371
- HTTP consumer role configuring, SSL used 372
- HTTP provider role configuring, SSL used 372, 373
- sample 383
- SSL 372

security, sample

- deploying 387
- HTTP basic security, configuring 385, 386
- running 388
- sample use case 384

service aggregation

- business integration sample 347

service aggregation, business integration

sample

- Address validation service 348
- aggregator, ESB components 349
- Bank history validation service 348
- Credit card validation service 348
- deploying 365
- endpoint, ESB components 349
- ESB components 349
- JBI-based ESB component architecture 350
- JBI-based ESB component architecture, UML diagram 350
- message exchange 351
- message exchange, onMessageExchange method 352
- message exchange, sequence 1 352, 353
- message exchange, sequence 2 354, 355
- message exchange, sequence 3 355, 356
- message exchange, sequence 4 357, 358
- message exchange, sequence 5 359
- message exchange, sequence 6 360, 362
- message exchange, sequence 7 362
- message exchange, sequence 8 363
- message exchange, sequence 9 363, 364
- message exchange, sequence diagrams 351
- message exchange, transform method 358
- normalizer, ESB components 349

- recipient list, ESB components 349
- running 365
- solution architecture 348
- translator, ESB components 349
- validation services 348
- Service Level Agreement.** *See* **SLA**
- SLA 23**
- ServiceMix**
 - about 58, 123
 - architecture 58
 - client code, running 78
 - component helper classes 136
 - components 68
 - custom JBI components, need for 135, 136
 - deployment 126
 - external HTTP service, binding 70
 - features 58
 - HTTP service, configuring 74, 75
 - installing 65
 - JBI components, developing 135
 - JMS 203
 - lightweight JBI components 69
 - NMR flows 59
 - packaging 124
 - packaging and deployment sample 127
 - QOS features, support 368
 - running 76, 77
 - sample 70
 - servicemix-eip component 293
 - servicemix-http 183
 - servicemix-jms 203
 - servicemix-jsr181 162
 - servlet-based HTTP service 71-73
 - Spring XML configuration 79
 - standard JBI components 68
- ServiceMix, deployment**
 - about 126
 - deployment modes 126
 - lightweight, deployment modes 127
 - standard and JBI compliant, deployment modes 126
- ServiceMix, packaging**
 - about 124
 - installing 124, 125
 - jbi.xml installation descriptor 124
 - Service Assembly, jbi.xml deployment descriptor 125
 - Service Assembly packaging 125
 - Service Unit, jbi.xml deployment descriptor 126
 - Service Unit packaging 126
- servicemix-http, ServiceMix**
 - about 183
 - configuration parameters 185
 - features 183, 184
 - lightweight configuration 188
 - servicemix-http as consumer 185
 - servicemix-http as provider 186, 187
 - XBean configuration 185
- servicemix-jms, ServiceMix**
 - about 203
 - configuration parameters 204
 - configuring 204
 - features 203
 - lightweight configuration 206
 - servicemix-jms as consumer 204
 - servicemix-jms as provider 205
 - XBean configuration 204
- servicemix-jsr181, ServiceMix**
 - about 162
 - deploying, XBean-based deployment used 163, 164
 - endpoints, configuring 164
 - features 163
 - JBI channel, linking to XFire transport 163
 - JBI channel, linking to XFire transport 162
 - POJO as services, exposing 162
- ServiceMix cluster**
 - about 373
 - configuring 375, 376
 - deploying 375
 - features 374
 - setting up 375
- ServiceMix EAI patterns**
 - configuring 293
 - servicemix-eip, configuring as standard JBI component 293
 - servicemix-eip configuring, servicemix.xml file used 293
- ServiceMix JBI Proxy 243**
- Service Oriented Integration.** *See* **SOI**
- SOA**
 - about 32
 - drawbacks 32

- features 32
 - need for 32
 - SOI 36
 - technologies 32
 - web services 33
 - SOA, versioning**
 - about 261
 - autonomous services 262
 - change, need for 262
 - fund transfer example 263, 264
 - interfaces 262, 263
 - jargon version 265
 - schemas, need for 265
 - web service method 263
 - SOAP 201**
 - SOI 36**
 - split aggregator, EAI patterns components**
 - about 329
 - configuring 331
 - Default ServiceMix client component, sample use case 330
 - definition 329
 - illustrative design, Acme company 330
 - sample, deploying 332
 - sample, running 333
 - sample code 331, 332
 - sample use case 330
 - split aggregator component, sample use case 330
 - symbol 329
 - trace component, sample use case 330
 - standard JBI components, ServiceMix**
 - about 68
 - servicemix-bean 68
 - servicemix-bpe 68
 - servicemix-camel 68
 - servicemix-drools 68
 - servicemix-eip 68
 - servicemix-file 68
 - servicemix-ftp 68
 - servicemix-http 68
 - servicemix-jms 68
 - servicemix-jsr181 68
 - servicemix-lwcontainer 68
 - servicemix-quartz 69
 - servicemix-saxon 69
 - servicemix-script 69
 - servicemix-wsn2005 69
 - servicemix-xmpp 69
 - static recipient list, EAI patterns components**
 - about 313
 - configuring 316
 - definition 314
 - illustrative design, Acme company 314
 - JMS client component, sample use case 315
 - JMS consumer component, sample use case 315
 - receiver component, sample use case 315
 - sample, deploying 318
 - sample, running 318
 - sample code 316, 317, 318
 - sample use case 315
 - static recipient list component, sample use case 315
 - symbol 313
 - static routing slip, EAI patterns components**
 - about 339
 - configuring 342
 - definition 340
 - HTTP client component, sample use case 341
 - HTTP connector component, sample use case 341
 - illustrative design, Acme company 340
 - receiver component, sample use case 341
 - sample, deploying 344
 - sample, running 344
 - sample code 342, 343
 - sample use case 341
 - static routing slip component, sample use case 341
 - symbol 340
 - STP 22**
 - Straight Through Processing. *See* STP**
 - system integration**
 - EAI 12
- ## T
- transactions, QOS features**
 - about 368
 - asynchronous transactional exchange, programming 371

- JBIs exchanges 368
- message exchange sending, send method used 370
- message exchange sending, sendSync method used 369
- sample 377
- synchronous transactional exchange, programming 370

transactions, sample

- configuring, in Servicemix 379, 381
- deploying 382
- running 382
- sample use case 378

V

versioning 261

Voice over IP (VOIP) 347

W

web service

- about 181
- binding 182
- building, XFireConfigurableServlet used 101
- building, XFire Spring Jsr181 handler used 109
- building, XFire Spring XFireExporter used 106
- consumer and provider, indirection 182
- SOAP over HTTP versus SOAP over JMS 201, 202
- testing, Axis client used 215
- testing, document style 214
- testing, JMS channel used 214
- testing, JMS client used 214, 215
- testing, RPC style 215

web service, classes

- client-config.wsdd, RPC style 216
- JMSSender.java 217
- JMSTestClientRPCWebService.java, RPC style 215
- JMSTransportForAxis.java, RPC style 216

web service-JMS channel binding sample

- servicemix-eip pipeline bridge, XBean-based 212

- servicemix-http provider destination, XBean-based 212
- servicemix-jms, XBean-based binding 211
- ServiceMix component architecture 209
- ServiceMix sample, deploying 213
- web service, deploying 210

web service binding sample

- Axis-based stubs, generating 196, 197
- Axis client codebase, building 198
- deploying 193
- running 193
- use case 190
- web service, deploying 190,-192
- WSDL, accessing 194, 195
- XBean-based deployment 193

Web Service Reliable Messaging

- specifications 200
- specifications, aspects 201
- Web Services Reliability specification 200
- Web Services Reliable Messaging specification 201

web service, SOA

- SOAP 34
- SOAP request 35
- SOAP response 35
- WSDL 33
- WSDL, sections 34

web service versioning

- approaches 268
- levels 266
- operational perspective 287
- strategy 265
- targetNamespace, for WSDL 267
- version parameter 267
- XML schema used 266

web service versioning, approaches

- covenant 268, 269
- multiple endpoint address 269

web service versioning-ESB sample

- about 270
- content-based router, configuring in ServiceMix 277, 279
- content-based router, JBI components 271
- content retriever, configuring in ServiceMix 280
- content retriever, JBI components 272

- deploying 285, 287
- HTTP consumer, JBI components 272
- HTTP provider, configuring in ServiceMix 282
- HTTP provider, JBI components 272
- JMS client, configuring in ServiceMix 274, 276
- JMS client, JBI components 271
- JMS consumer, configuring in ServiceMix 276
- JMS consumer, JBI components 271
- JMS provider, configuring in ServiceMix 285
- JMS provider, JBI components 272
- pipeline component, configuring in ServiceMix 280
- pipeline component, JBI components 271
- remote web service, configuring in ServiceMix 282, 284
- remote web service, JBI components 272
- running 285, 287
- sample use case 270
- service pipeline, configuring in ServiceMix 281
- service pipeline, JBI components 272
- whitespace transformer, configuring in ServiceMix 276
- whitespace transformer, JBI components 271
- wiretap, EAI patterns components**
 - about 319
 - configuring 321
 - definition 320
 - echo component, sample use case 320
 - HTTP client component, sample use case 320
 - HTTP connector component, sample use case 320
 - illustrative design, Acme company 320
 - sample, deploying 323
 - sample, running 323
 - sample code 321, 323
 - sample use case 320
 - symbol 319
 - trace component, sample use case 321
 - wiretap component, sample use case 320
- WS-Reliability 200**
- WS-Reliable Messaging 201**
- WSDL document 267**
- X**
- XFire**
 - about 99
 - binding 100
 - transport mechanism 100
- XFireConfigurableServlet**
 - classes 102-104
 - client, running 105
 - code listing 102
 - sample, running 104
 - sample scenario 101
- XFire Spring Jsr181 handler**
 - classes 111, 112
 - client, running 113
 - code listing 110
 - sample, running 113
 - sample scenario 109
- XFire Spring XFireExporter**
 - classes 108, 109
 - client, running 109
 - code listing 107
 - sample, running 109
 - sample scenario 106
- XML 222**
- XML schema 266**
- XML streams 182**
- XPath splitter, EAI patterns components**
 - about 308
 - configuring 311
 - definition 309
 - illustrative design, Acme company 309
 - JMS client component, sample use case 310
 - JMS consumer component, sample use case 310
 - sample, deploying 312
 - sample, running 312, 313
 - sample code 311, 312
 - sample use case 310
 - symbol 309
 - trace component, sample use case 310
 - XPath splitter component, sample use case 310

XStream

- about 223
- features 223
- integrating, with ServiceMix 225, 226

XStream in NMR sample

- building 235
- components 227
- HTTPClient 228
- HTTP Connector, components 227
- HTTPInterceptor, components 227, 230, 231
- HTTPInterceptor component, configuring 232, 233
- HTTPInterceptor component, deploying 234

- HTTPInterceptor component, packaging 234

Java Transfer Objects classes 228, 229

running 235

sample use case 226

XML documents, unmarshaling to Transfer Objects 228

XStreamInspector, components 227, 232

XStreamInspector component, configuring 232, 233

XStreamInspector component, deploying 234

XStreamInspector component, packaging 234