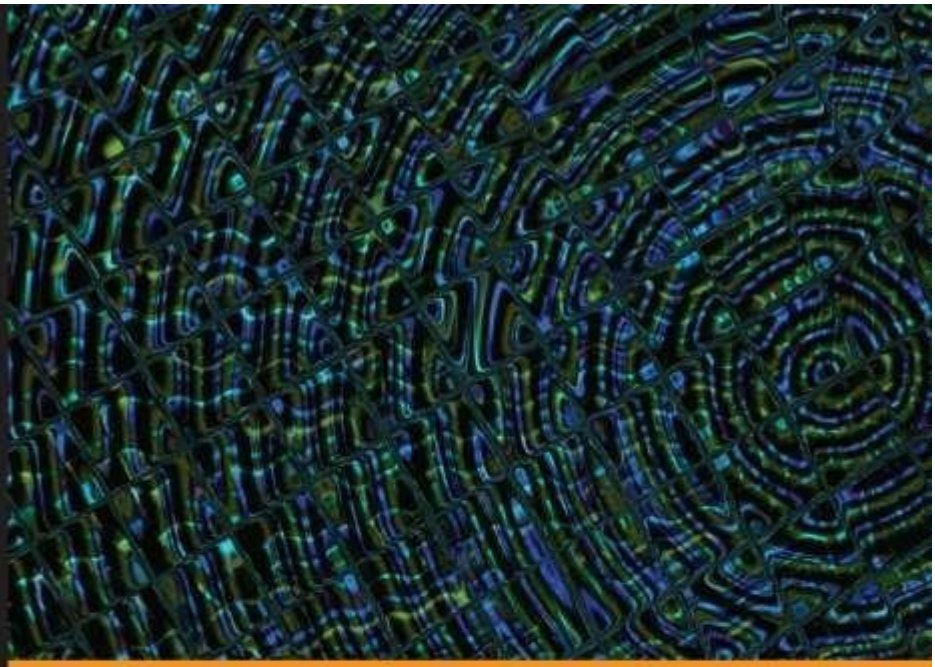


Community Experience Distilled

# Enterprise Integration with WSO2 ESB

Over 15 recipes to calibrate seamless modularity to SOA and address commonly-faced enterprise integration challenges with a zero-code approach

Prabath Siriwardena

[PACKT] open source*

# Table of Contents

# Enterprise Integration with WSO2 ESB

# Enterprise Integration with WSO2 ESB



First published: October 2013

Production Reference: 1171013

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK..

ISBN 978-1-78328-019-3

www.packtpub.com

Cover Image by Sheetal Aute (<sheetala@packtpub.com>)

# Credits

**Author**

Prabath Siriwardena

**Reviewers**

Kasun Indrasiri

Rajika Kumarasiri

Kishanthan Thangarajah

**Acquisition Editor**

Vinay Argekar

**Commissioning Editor**

Subho Gupta

Priyanka Shah

**Technical Editors**

Tanvi Bhatt

Shiny Poojary

**Project Coordinator**

Romal Karani

**Proofreader**

Linda Morris

**Indexer**

Mariammal Chettiyar

Rekha Niar

**Production Coordinator**

Melwyn D'sa

**Cover Work**

Melwyn D'sa

# About the Author

**Prabath Siriwardena** is the Director of Security Architecture at WSO2 Inc., a company that produces a wide variety of open source software from data to screen. Before that he chaired the Management Committee—Integration Technologies at WSO2. He completed his degree and then a Masters from the University of Moratuwa, Sri Lanka. He is a member of OASIS Identity Metasystem Interoperability (IMI) TC, OASIS eXtensible Access Control Markup Language (XACML) TC, and OASIS Security Services (SAML) TC. He is also a member of Apache Axis PMC. He has spoken at numerous international conferences including OSCON, ApacheCon, WSO2Con, EIC, and IDentity Next. He has more than 9 years industry experience and has worked with many Fortune 100 companies.

# Acknowledgement

WSO2 ESB is one of the leading ESBs out there in terms of features, scalability, and performance. It's being battle-tested at eBay and many other Fortune 100 companies. At the time of this writing, eBay is handling more than 4 billion transactions per day through the WSO2 ESB. In this book, I will cover some of the key features of WSO2 ESB mostly used in enterprise integration. Each feature is covered at the introductory level, to help anyone who is not that familiar with the WSO2 ESB, to catch up and proceed.

I would first like to thank, Vinay Argekar, an Acquisition Editor at Packt Publishing who came up with the idea of writing a book on the most popular WSO2 product—undoubtedly the ESB. Then, I would like to thank Sneha Modi, Priyanka Shah, Subho Gupta, Romal Karani, Tanvi Bhatt, and all the others from Packt Publishing who helped me throughout to make this book a reality from the initial idea. Thank you very much for all your continuous support.

If not for Dr. Sanjiva Weerawarana and Paul Fremantle we wouldn't have had a WSO2 ESB today to talk about. They founded WSO2 in 2005 with a mission to build a new era for SOA, and WSO2 ESB is a key ingredient. Today WSO2 provides a fully open source platform with more than 16 products. I am truly grateful to both Dr. Sanjiva and Paul for everything they have done for this field and for the community. And also, for mentoring and moulding me with care and patience.

Also, I would like to thank Samisa Abeysinghe, who is our VP, Training at WSO2 for being a great mentor to me throughout all these years.

My beloved wife, Pavithra. She wanted me to write this book even more than I wanted to. If I say she is the driving force behind this book, I am not exaggerating. She simply went beyond just feeding me with all the encouragement, but also helped immensely in reviewing the book and developing samples. She was the first reader, as always. Thank you very much Pavithra.

Kasun Indrasiri, who is the Product Manager and the Architect of WSO2 ESB, added so much value to the book with his technical expertise by reviewing the book. Thank you very much Kasun. I would also like to thank Rajika, Kishanthan and Charitha for reviewing the book for technical accuracy. Your inputs are highly appreciated.

Miyuru Daminda, Dushan Abeyruwan, Isuru Udana, and all the members of the WSO2 ESB team helped me a lot, clarifying all my doubts related to the product internals. Thank you very much, I appreciate your help a lot.

Last, but not least, my parents and my sister are the driving force behind me all the time since my birth. If not for them I wouldn't be who I am today. I am so very grateful to them for leading my way to write my first book.

Although this sounds like a one-man effort—it's a team. I would like to thank everyone who supported me in different ways.

# About the Reviewers

**Kasun Indrasiri** is a software architect at WSO2 Inc., who is mainly focusing on WSO2 Enterprise Service Bus. He is an elected member of the Apache Software Foundation and a Project Management Committee member and committer for the Apache Synapse open source ESB project.

His expertise lies in Enterprise Integration, designing Highly Efficient Message Processing Systems, Complex Event Processing, Distributed Systems, and Information Retrieval Systems.

He has provided technology consulting on customer engagements, helping to successfully design and implement solutions for integrating Web Services, SAP Integration, FIX, and various other integration solutions.

He is proficient in Java and C/C++ development and he was a contributor in Apache Axis2/C and Apache Rampart/C projects. He is also a speaker at WSO2Con 2011, WSO2Con 2013 London, and WSO2Con 2013 San Francisco.

He holds a BSc Engineering degree in computer science and engineering from the University of Moratuwa.

**Rajika Kumarasiri** is a graduate student from the University of Toledo. He has the expertise and experience in data structures and algorithms, designing and implementing high performance adapters, connectors and transports for enterprise integration products, and products developments. He is also a committer and a PMC member at Apache Software Foundation.

You can get to know more about him on http://www.linkedin.com/pub/rajika-kumarasiri/54/146/352

**Kishanthan Thangarajah** is a Project Management Committee (PMC) member and a committer of the Apache Software Foundation. He is currently contributing to the Apache Axis2 project. He is also a Senior Software Engineer at WSO2 Inc. where he mainly works with the WSO2 Carbon Platform Team. His main interest is in Distributed Computing. As a research project during his undergraduate studies, he worked on **Mahasen: A distributed Storage Resource Broker for managing a large amount of distributed data**. He holds a B.Sc. Honors degree in Computer Science & Engineering from the University of Moratuwa, Sri Lanka.

He also loves playing tennis and captained the university team in 2010 and was the key member in winning the championship for five consecutive years (2007-2011). He also represented Sri Lanka for tennis at the World

University Games in 2009 (Belgrade, Serbia) and 2011 (Shenzhen, China). He can be reached via his blog at: http://kishanthan.wordpress.com/

# www.PacktPub.com

## Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at <service@packtpub.com> for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.


http://PacktLib.PacktPub.com

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

## Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

## Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

# Preface

**The Enterprise Service Bus** (**ESB**) today serves as a key component in most of the enterprise grade deployments. In most cases the ESB removes point-to-point dependencies in your system to build a highly scalable and loosely coupled solution. But that does not necessarily mean ESB means **SOA**. ESB is a key ingredient to build an SOA infrastructure, but it's not a must. Even with an ESB if not followed industry best practices and patterns you will end up with a mess.

Enterprise Integration with WSO2 ESB guides you through common patterns used in the industry to address enterprise integration challenges with WSO2 ESB.

# What this book covers

Chapter 1, *Getting Started* focuses on giving a brief introduction to the Enterprise Service Bus in general, when to use it and when not to. We will also see how to deploy WSO2 ESB in a production environment with optimal settings.

Chapter 2, *Enterprise Integration Patterns* explains how to implement a mostly used set of **Enterprise Integration Patterns** (**EIP**) with WSO2 ESB. It covers Content-Based Router, Dynamic Router, Splitter, Scatter & Gather, Publish & Subscribe, Service Chaining, Content Enricher, and Detour with examples. This is only a limited set of EIPs, which provides a base, while WSO2 ESB is capable of implementing almost all the EIPs highlighted in *Gregor Hohpe's Enterprise Integration Patterns* book.

Chapter 3, *Integration with Third-party Message Brokers* explains how to integrate WSO2 ESB with third-party message brokers. First, we will start with Apache ActiveMQ and show how WSO2 ESB can consume messages from an ActiveMQ Queue. Then, we will see how WSO2 ESB acts as a publisher. Later, we will explain how we can integrate Apache Qpid as a message broker to the WSO2 ESB message store.

Chapter 4, *Business Messaging and Transformations* explains, by example, a selected set of capabilities of WSO2 ESB in business messaging and transformations. FIX, HL7, and SAP are covered in the first three sections. WSO2 ESB ships with a rich set of connectors to a wide variety of business applications. Later in this chapter, we explain how to use Twitter connector and then how to transform REST/JSON messages into SOAP and vice versa.

Chapter 5, *Task Scheduling* explains how to setup scheduled tasks through WSO2 ESB.

Appendix A, *WSO2 ESB Terminology* introduces commonly used terminology related to WSO2 ESB.

# What you need for this book

This book assumes the reader is quite familiar with SOA and related concepts. To run the samples provided, you need to have the following software installed.

- JDK 1.6_26 +
- WSO2 ESB 4.8.0 +
- cURL

# Who this book is for

Anyone quite familiar with SOA, SOAP, and REST could follow this book without having any prior understanding about WSO2 ESB. Also prior knowledge in Java is not required.

# 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.

Code words in text are shown as follows: "In Windows you can simply run `ESB_HOME/bin/wso2server.bat`."

A block of code is set as follows:

```
http.socket.timeout=120000
worker_pool_size_core=400
worker_pool_size_max=500
io_buffer_size=16384
```

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

```
$ java -Dhl7_payload=request.hl7 -cp
xercesImpl-
2.8.1.wso2v2.jar:hapi_1.2.0.wso2v1.jar:com.packt.wso2.esb.client.hl7.jar:
logging-
1.1.3.jar com.packt.wso2.esb.client.hl7.HL7Client
```

**New terms** and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Select **Axis2 Transport HL7** and then click on **Install** and follow the wizard"

> ## Note
>
> Warnings or important notes appear in a box 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 send an e-mail to <feedback@packtpub.com>, and mention the book title through the subject of your message.

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

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

## Errata

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

## Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at <copyright@packtpub.com> with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

## Questions

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

# Chapter 1. Getting Started

The first part of this chapter will focus on giving a brief introduction to the **Enterprise Service Bus** (**ESB**) in general, when to use it and when not to. Next, we will see how to deploy WSO2 ESB in a production environment with optimal settings.

The Enterprise Service Bus (ESB) today serves as a key component in most of the enterprise grade deployments.

> *An ESB is a middleware solution that enables interoperability among heterogeneous environments using a service-oriented model. An ESB models an application endpoint as a service. The ESB may host the service agent locally, or the service may execute remotely. In both cases, the ESB provides an abstraction layer that virtualizes the service and separates it from infrastructure concerns. The ESB makes the service accessible to other applications via one or more middleware protocols. As a general rule, one of the protocols that an ESB supports is Simple Object Access Protocol (SOAP), but it doesn't require all services to communicate via SOAP. The ESB mediates interactions between service endpoints and enables dissimilar systems to interoperate.*

That's how *Gartner* defines the *Enterprise Service Bus*, as explained in depth at http://www.gartner.com/id=1405237.

In most cases the ESB removes point-to-point dependencies in your system to build a highly-scalable and loosely coupled solution. But that does not necessarily mean that ESB means SOA. ESB is a key ingredient to build an SOA infrastructure, but it's not a must. Even with an ESB, if not followed in industry's best practices and patterns, you will end up with a mess. Take ESB just as a tool to build an SOA in your enterprise and do not try to design an SOA around an ESB. Let your needs ask for an ESB, but do not start from it.

WSO2 ESB is an open source Enterprise Service Bus released under the business friendly Apache 2.0 license. It has support for most of the **Enterprise Integration Patterns** (**EIP**) highlighted by *Gregor Hohpe* and *Bobby Woolf* in their famous book, which laid the cornerstone for many of the ESBs out there today. You can read more about EIPs from http://www.eaipatterns.com. The complete catalogue of enterprise integration patterns supported by WSO2 ESB can be found at http://docs.wso2.org/wiki/display/IntegrationPatterns/Enterprise+Integration+Patterns+with+WSO2+ESB.

In most of the enterprise grade production deployments, ESB acts as a Service Gateway. The Service Gateway is a centralized access point where heterogeneous service providers meet heterogeneous service clients.

We would expect the following functionalities from a Service Gateway but not limited. In the sections to follow, we'll find out how capable WSO2 ESB is to cater for these requirements:

- Protocol switching.
- Message transformations.
- Centralized policy enforcement for authentication, authorization, and throttling.
- Centralized auditing and monitoring.
- Message screening and schema validation.
- Message routing.
- Expose legacy systems through standard interfaces.
- Expose business functionalities through service orchestration.
- Handle versioning.
- Act as a reliable message store for persistence and rate limit matching.

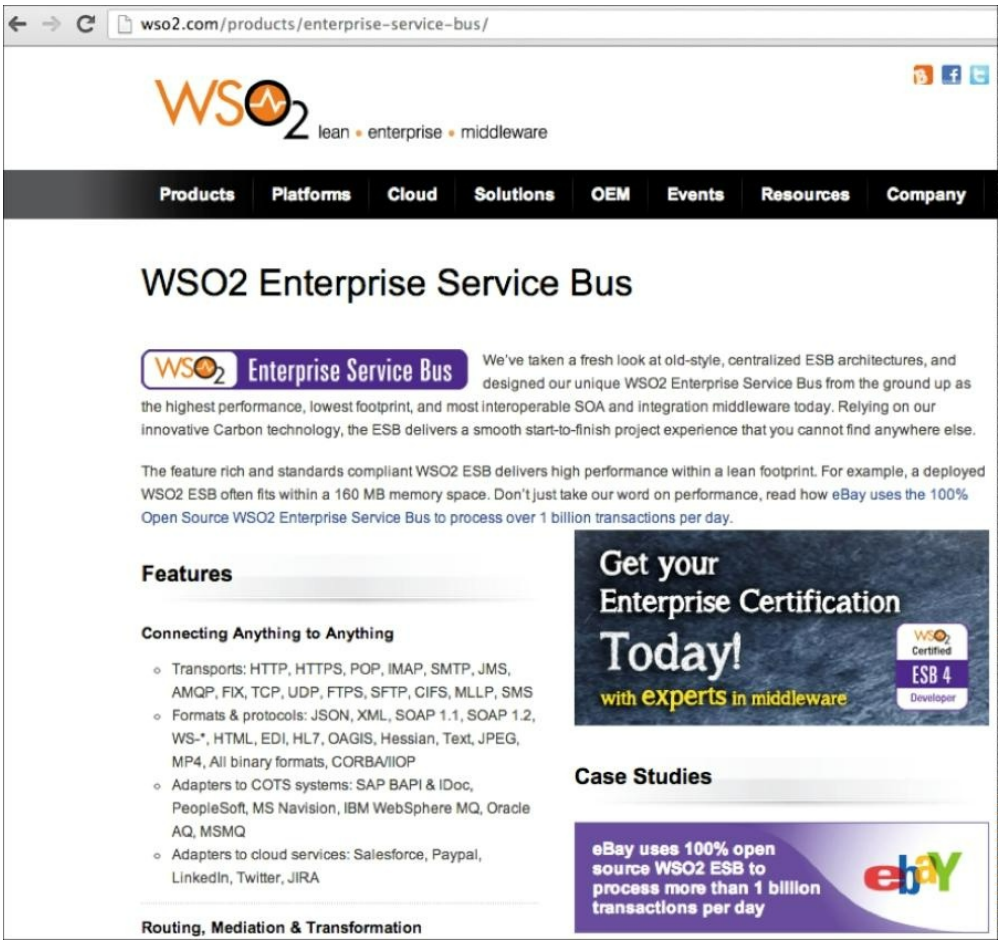
# Setting up WSO2 ESB for production use (Simple)

This recipe will guide you through the process to setup WSO2 ESB from scratch in your production deployment and fine-tune it for optimal performance.

## Getting ready

Setting up WSO2 ESB is not a huge/tedious task. In simple steps this is what you need to do:

1. Download the latest version of WSO2 ESB from http://wso2.com/products/enterprise-service-bus/. As of this writing the latest is 4.8.0 and all the samples presented throughout the book are tested with that. In case version 4.8.0 is not yet available at the above provided link, you can download the milestone 4 build from https://svn.wso2.org/repos/wso2/people/prabath/esb/packtbook/wso2esb-4.8.0.zip.




> **Note**
>
> You can run WSO2 ESB on Microsoft Windows, Linux (Ubuntu, Fedora, Gentoo, SUSE, and Debian, and so on), Oracle Solaris and Mac OS X with a JDK 1.6.x runtime. The recommended JDK is Oracle JDK 1.6_26+.

2. Set `JAVA_HOME` environment variable. You need to point `JAVA_HOME` to the root directory where you have installed JDK in your local machine.
3. Start WSO2 ESB. In Windows you can simply run `ESB_HOME/bin/wso2server.bat` and under Linux you can run `ESB_HOME/bin/wso2server.sh`. `ESB_HOME` points here to the local directory that you unzipped, the WSO2 ESB distribution.

> **Note**

Step-by-step instructions to set up WSO2 ESB on Windows can be found at http://docs.wso2.org/wiki/display/ESB480/Installing+on+Windows.

Step-by-step instructions to set up WSO2 ESB on Linux can be found at http://docs.wso2.org/wiki/display/ESB480/Installing+on+Linux.

## How to do it...

Once we have the initial setup completed we can start tightening the security of the WSO2 ESB. The following steps list some of the recommendations for a secured deployment. If you are not worried about production deployment guidelines yet, then you can skip the rest and move to the next section.

1. Change the key stores. You can find two key stores (`.jks`) that ship with WSO2 ESB at `ESB_HOME/repository/resources/security/`. Out of those two, `wso2carbon.jks` is the primary key store and `client-truststore.jks` is the trust store, where you will have the public certificates of trusted **Certificate Authorities** (**CAs**). Once you change the key stores, you need to update the corresponding entries in the following configuration files:

    - `ESB_HOME/repository/conf/tomcat/catalina-server.xml`
    - `ESB_HOME/repository/conf/carbon.xml`
    - `ESB_HOME/repository/conf/axis2/axis2.xml`

2. Use Secure Vault to encrypt all the passwords in the following configuration files:

    - `ESB_HOME/repository/conf/user-mgt.xml`
    - `ESB_HOME/ repository /conf/carbon.xml`
    - `ESB_HOME/ repository /conf/axis2/axis2.xml`
    - `ESB_HOME/repository/conf/datasources/master-datasources.xml`
    - `ESB_HOME/repository/conf/tomcat/catalina-server.xml`

3. Change the default ports. By default, WSO2 ESB runs on HTTPS port `9443` and HTTP port `9763`. Also, WSO2 ESB exposes services over `8243` and `8280`. To change the ports you can update the value of `<Offset>` at `ESB_HOME/repository/conf/carbon.xml`
4. If you are deploying WSO2 ESB in a **Demilitarized Zone** (**DMZ**) make sure you remove all the UI components from it. You can remove all the UI features by logging in to the WSO2 ESB and then go to **Configure** | **Features**. If you still prefer to configure ESB through a

UI, then you can deploy another ESB instance inside the LAN and connect the UI of that ESB to the ESB runtime deployed inside DMZ.

Now, we can start fine tuning WSO2 ESB for the highest performance. It performs at its best with Linux, so the underlying parameters related to that are as follows:

1. Kernel level parameters can be fine tuned for optimal performance via the `/etc/sysctl.conf` file in Linux. Open `/etc/sysctl.conf` with your favorite editor and add the following parameters to it:

    - `net.ipv4.tcp_fin_timeout = 30`

      Specifies the time in seconds TCP takes to wait for the final FIN, before the socket is closed.

    - `fs.file-max = 2097152`

      Specifies the maximum number of concurrently open file descriptors in the system.

    - `net.ipv4.tcp_tw_reuse = 1`

      By default, the value of this parameter is `0`. When set to `1`, the `TIME-WAIT` sockets for new connections can be reused across the system.

    - `net.ipv4.tcp_tw_recycle = 1`

      Once `net.ipv4.tcp_tw_reuse` is set to `1`, to enable fast recycling of `TIME-WAIT` socket status, we need to set the above to `1`.

    - `net.core.rmem_default = 524288`

      Specifies the default operating system receive buffer size for all types of connections.

    - `net.core.wmem_default = 524288`

      Specifies the default operating system send buffer size for all types of connections.

    - `net.core.rmem_max = 67108864`

      Specifies the maximum operating system receive buffer size for all types of connections.

    - `net.core.wmem_max = 67108864`

Specifies the maximum operating system send buffer size for all types of connections.

- `net.ipv4.tcp_rmem = 4096 87380 16777216`

  The first value specifies the minimum receive buffer for each TCP connection. The second value is the default receive buffer allocated for each TCP socket. The third specifies the maximum receive buffer that can be allocated for a TCP socket.

- `net.ipv4.tcp_wmem = 4096 65536 16777216`

  The first value specifies the minimum send buffer for each TCP connection. The second value is the default send buffer allocated for each TCP socket. The third specifies the maximum send buffer that can be allocated for a TCP socket.

- `net.ipv4.ip_local_port_range = 1024 65535`

  Once specified, it will increase the IP port limit of the system.

2. Under Linux, to change the maximum number of open files allowed for system users, go to `/etc/security/limits.conf` and configure the following parameters:

   ```
   soft nofile 4096
   hard nofile 65535
   ```

3. The maximum heap size and the maximum perm gen size allocated for WSO2 ESB can be changed by modifying the `ESB_HOME/bin/wso2server.sh` file in Linux and the `ESB_HOME/bin/wso2server.bat` file in Windows.

   The default setting for WSO2 ESB 4.8.0 is: `-Xms256m -Xmx512m -XX:MaxPermSize=256m`

   - `-Xms`: Minimum heap size
   - `-Xmx`: Maximum heap size
   - `-XX:MaxPermSize`: Maximum perm gen size.

4. The streaming `XPath` parser optimizes the way `XPath` expressions are being parsed. To enable `XPath` in WSO2 ESB, configure the following two parameters in the `ESB_HOME/repository/conf/synapse.properties` file.

   ```
   synapse.streaming.xpath.enabled=true
   synapse.temp_data.chunk.size=3072
   ```

> **Note**
>
> To learn more about the streaming XPath parser for WSO2 ESB, go to http://wso2.com/library/articles/2013/01/streaming-xpath-parser-wso2-esb.

5. The access logs can be used to log requests and responses passing through WSO2 ESB. For optimal performance, we can disable access logs by setting the following values in the `ESB_HOME/repository/conf/log4j.properties` file:

```
log4j.logger.org.apache.synapse.transport.nhttp.access=WARN
log4j.logger.org.apache.synapse.transport.passthru.access=
WARN
```

6. WSO2 ESB uses **Pass Thru Transport** (**PTT**) as the default transport. If we want to use the HTTP-NIO transport, comment out PTT and un-comment the HTTP-NIO transport in the `ESB_HOME/repository/conf/axis2/axis2.xml` file. Also, a preconfigured file for HTTP-NIO transport ships with the product, available at `ESB_HOME/repository/conf/axis2/axis2_nhttp.xml`. You can simply rename it to `axis2.xml`.
7. To optimize the performance of PTT, set the following parameters in the `ESB_HOME/repository/conf/passthru-http.properties` file:

```
http.socket.timeout=120000
worker_pool_size_core=400
worker_pool_size_max=500
io_buffer_size=16384
```

8. The HTTP-NIO transport performance can be fine tuned by creating an `nhhttp.properties` file under the `ESB_HOME/repository/conf` directory, and adding the following configuration parameters:

```
http.socket.timeout=120000
http.socket.buffer-size=8192
http.tcp.nodelay=1
http.connection.stalecheck=0
# HTTP Sender thread pool parameters
snd_t_core=200
snd_t_max=250
snd_alive_sec=5
snd_qlen=-1
snd_io_threads=16
# HTTP Listener thread pool parameters
lst_t_core=200
lst_t_max=250
lst_alive_sec=5
lst_qlen=-1
```

```
lst_io_threads=16
```

## Tip

### Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

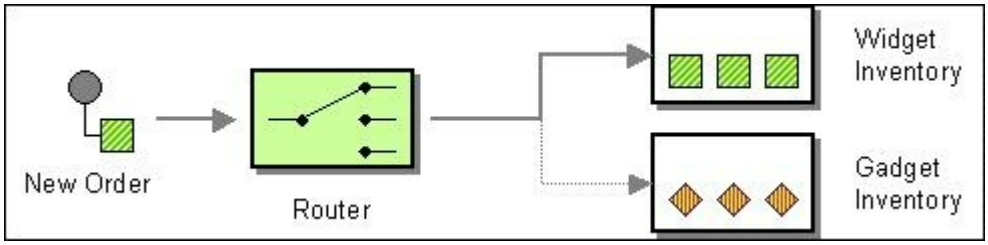
# Chapter 2. Enterprise Integration Patterns

This chapter explains how to implement a widely used set of **Enterprise Integration Patterns** (**EIP**) with WSO2 ESB. It covers Content-Based Router, Dynamic Router, Splitter, Scatter and Gather, Publish and Subscribe, Service Chaining, Content Enricher, and Detour with examples. This is only a limited set of EIPs, which provides a base, while WSO2 ESB is capable of implementing almost all the EIPs highlighted in the *Enterprise Integration Patterns* book by *Gregor Hohpe*.

## Content-Based Router (Simple)

The Content-Based Router Enterprise Integration Pattern explains how to handle a scenario where a single logical function is being implemented across multiple different systems.

The following image extracted from the *Enterprise Integration Patterns* book by *Gregor Hohpe* illustrates an order request being routed to different inventory management systems based on the content of the message:




### Getting ready

Let's say we have a `CreditCardPaymentService`, which would accept payment-processing requests, but would delegate further processing to different other services based on the type of the credit card, VISA, AMEX, or Master.

Note the following facts.

- The schema of the requests coming into the `CreditCardPaymentService` may not be identical.
- The schema of the response going out from the `CreditCardPaymentService` may not be identical.
- The schema of the requests coming into the `CreditCardPaymentService` are identical and in a common format, but the `VISAProcessingService` expects the requests to be in a

different format. Similarly, AMEX and Master card expect requests to be in different formats.

- The schema of the responses coming out from the `CreditCardPaymentService` are identical and in a common format, but the responses from the `VISAProcessingService` are in a different format. Similarly, the responses from AMEX and Master card are in different formats.

## How to do it...

1. Setup `VISAProcessingService` and `AMEXProcessingService`. We need to have these business services running, so WSO2 ESB can route messages to them.
2. Get `VISAProcessingService.aar` and `AMEXProcessingService.aar` from `SAMPLE-1` and copy those to `wso2esb-4.8.0/samples/axis2Server/repository/services/`. If there is no `services` directory create one.

> **Note**
>
> WSO2 ESB comes with a simple Axis2Server where you can run sample services to test integration scenarios with WSO2 ESB. Here we are using it to deploy our two business services. In a production deployment these two services could be running on IBM WebSphere, Oracle WebLogic, or WSO2 Application Server.

3. Start simple Axis2Server from `wso2esb-4.8.0/samples/axis2Server`.

   Linux command: `sh axis2server.sh`

   Windows command: `axis2server.bat`

> **Note**
>
> By default the server will get started on HTTP 9000 and HTTPS 9002.

4. Validate whether the two business services are up and running. (`http://localhost:9000/services/VISAProcessingService?wsdl` and `http://localhost:9000/services/AMEXProcessingService?wsdl`)
5. Start WSO2 ESB and login with username as `admin` and password also as `admin`. ESB will start on `https://localhost:9443`.

Linux command: `sh bin/wso2server.sh`

Windows command: `bin/wso2server.bat`

> **Note**
>
> There are three ways you can configure WSO2 ESB. One is through the Graphical Editor. Another way is through the WSO2 Developer Studio and the other is through the Source View directly by the synapse configuration language. This book will follow the latter, but you can always edit through the Graphical Editor.

6. Get `synapse-cbr.xml` from `SAMPLE-1`, copy the content of it, go to **Main** | **Manage** | **Service Bus** | **Source View**, and paste it there, overriding the existing and click on **Update**.
7. The previous step will create a proxy service called `CreditCardPaymentService` in ESB, which will route messages to the two business services running in simple Axis2Server.
8. Go to **Main** | **Manage** | **Services** | **List**. You should be able to see the `CreditCardPaymentService` proxy service listed there.

> **Note**
>
> To edit/update a proxy service created through source, via WSO2 ESB graphical editor, go to **Main** | **Manage** | **Services** | **List** and click on the service to edit, click on the **Edit** link under **Specific Configuration**.

9. Now let's test our setup. We'll be using cURL throughout the book.

```
$ curl -d @request-visa.xml -H "Content-Type:
application/soap
+xml action=doPayment"
http://localhost:8280/services/CreditCardPaymentService

$ curl -d @request-amex.xml -H "Content-Type:
application/soap
+xml action=doPayment"
http://localhost:8280/services/CreditCardPaymentService
```

You can get `request-visa.xml` and `request-amex.xml` from `SAMPLE-1`.

# How it works...

Let's have a deep look at the synapse configuration. The following explains the key configuration elements.

- If you look at the WSDLs of `VISAProcessingService` and `AMEXProcessingService` you will notice that they have two different contracts. In other words, the schema of the requests and responses are different from each other.
- The proxy service, `CreditCardPaymentService` has a different WSDL from all the business services. This is how we introduce a common message format or a common contract to the consumers.
- The following mediators are used to implement the Content-Based Router pattern.
    - **Switch Mediator**: The <`switch`> mediator is used to extract `cardType` from the incoming request message and based on the card type we can define the endpoints for the business services.

```
<switch source="xpath">
  <case regex="string">
    mediator+
  </case>+
  <default>
    mediator+
  </default>?
</switch>
```

    - **Send Mediator**: The <`send`> mediator is used within the `<switch>` mediator to send messages to different end points.

```
<send>
  (endpointref | endpoint)+
</send>
```

    - **Payload Factory Mediator**: The `<payloadFactory>` mediator is used to craft new requests to the business services based on the incoming requests to the proxy service. The same mediator is used to craft the common response from the responses received via business services.

```
<payloadFactory>
  <format>"xmlstring"</format>
  <args>
    <arg (value="literal" |
expression="xpath")/>*
  </args>
</payloadFactory>
```

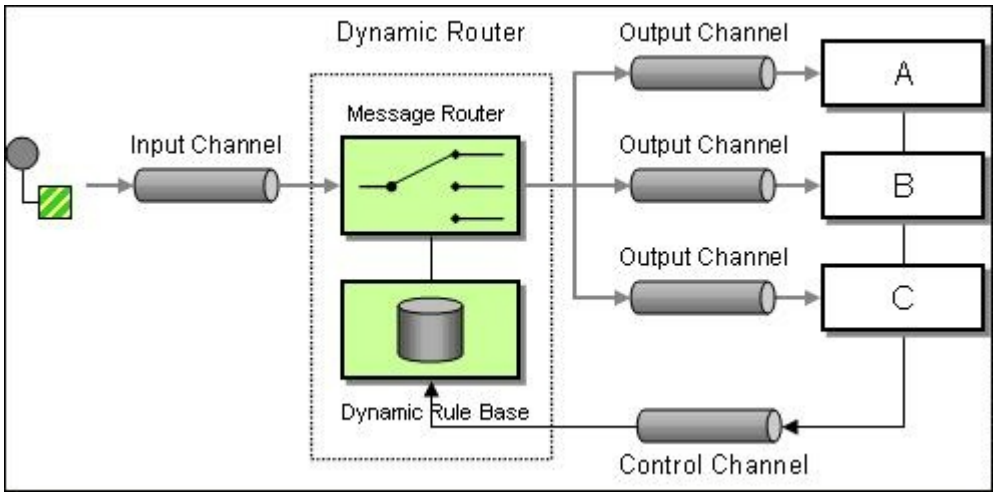
- **Filter Mediator**: The Filter mediator is used in `outSequence` to craft the response, based on the `cardType`. This is very similar to the <`switch`> mediator we introduced before.

```
<filter (source="xpath" regex="string")
| xpath="xpath">
  <then [sequence="string"]>
    mediator+
  </then>
  <else [sequence="string"]>
    mediator+
  </else>
</filter>
```

# Dynamic Router (Simple)

The Content-Based Router EIP is the mostly used form of the more generic Message Router pattern. With the Content-Based Router, the system should know the capabilities of the backend business services beforehand. That is how we route payments with card type VISA to `VISAProcessingService` and with card type AMEX to `AMEXProcessingService`. The downside of this approach is whenever we add or remove a business service, we need to change the configuration, or reconfigure the Content-Based Router.

The following image extracted from *Enterprise Integration Patterns* book by *Gregor Hohpe*, illustrates a request from the input channel being routed to different output channels based on the rules picked from the dynamic rule base:




## Getting ready

The Dynamic Router EIP explains how to avoid dependency of the router on all possible destinations/business services while maintaining its efficiency. The Dynamic router can be self-configured based on special configuration messages from participating destinations. With the Dynamic Router EIP, each business service has to announce their capabilities and the Dynamic Router will maintain a list of them.

You might have noticed in our Content-Based Router example, the endpoints for `VISAProcessingService` and `AMEXProcessingService` are hardcoded inside the `<send/>` mediator. Also, if you look at the `<switch/>` mediator, you will also notice we have hardcoded the card type too. `VISAProcessingService` and `AMEXProcessingService` are the recipients or the possible destinations. The Content-Based Router is coupled to both the

capabilities and the destinations. Let's see how to avoid this with the Dynamic Router.

The first thing we need to do is to decouple capabilities and destinations from the ESB. In other words, we need to externalize the capabilities/destinations mapping. We can externalize this mapping into a database or to a file. In the following example, we are going to use the filesystem to load endpoints.

## How to do it...

1. Create two files inside `ESB_HOME/repository` with the names, `VISA` and `AMEX`.
2. The name of each file is equal to the possible card types.
3. Edit the `VISA` file and add the following entry to it:
   `<value>http://localhost:9000/services/VISAProcessingService</value>`
4. Edit the `AMEX` file and add the following entry to it:
   `<value>http://localhost:9000/services/AMEXProcessingService</value>`
5. You might have noticed the pattern now. Each file has the name of a card type and in it, has the endpoint of the business service, which can handle that card type.
6. Get `VISAProcessingService.aar` and `AMEXProcessingService.aar` from `SAMPLE-2` and copy those to `wso2esb-4.8.0/samples/axis2Server/repository/services/`. If there is no `services` directory create one.
7. Start simple Axis2Server from `wso2esb-4.8.0/samples/axis2Server`.
8. Validate whether the two business services are up and running.
9. Start WSO2 ESB and login with the username as `admin` and the password also as `admin`. The ESB will start on `https://localhost:9443`.
10. Now let's load the synapse configuration to implement this Dynamic Router pattern.
11. Get `synapse-dynamic.xml` from `SAMPLE-2`, copy the content of it, go to **Main** | **Manage** | **Service Bus** | **Source View** and paste it there, overriding the existing and click on **Update**.
12. Now let's test our setup. We'll be using cURL.

```
curl -d @request-visa.xml -H "Content-Type:
application/soap
+xml action=doPayment"
http://localhost:8280/services/CreditCardPaymentService

curl -d @request-amex.xml -H "Content-Type:
application/soap
+xml action=doPayment"
http://localhost:8280/services/CreditCardPaymentService
```

You can get `request-visa.xml` and `request-amex.xml` from `SAMPLE-2`.

13. If you want to add a new capability to the system, only thing you need to do is to drop a file, following the convention, into the `ESB_HOME/repository` directory and add a sequence with the name of the `cardType` to do the message transformation.

## How it works...

The following mediators are used to implement the Dynamic Router pattern. The mediators already being highlighted in the Content-Based Router example are not discussed here.

- **Property Mediator**: The Property Mediator is used here to load the content of the corresponding file from the file system and to set its value, which is the endpoint, to a property, so it can be referred from the `<send/>` mediator. Also, note that we are using `org.wso2.carbon.mediation.registry.ESBRegistry` as the registry provider to load the files from the filesystem.

```
<registry
provider="org.wso2.carbon.mediation.registry.ESBRegistry">
  <parameter
name="localRegistry">/</parameter>
  <parameter
name="cachableDuration">15000</parameter>
</registry>
```

This is referred from the <`property>` mediator in our example, as shown here:

```
<property name="EPR"
expression="get-
property('registry',filePath)"/>
```

The file will be loaded by the `org.wso2.carbon.mediation.registry.ESBRegistry` and the value will be cached for the interval specified in `cachableDuration` parameter in milliseconds. Registry  is one of the scopes that the Property mediator can operate on.

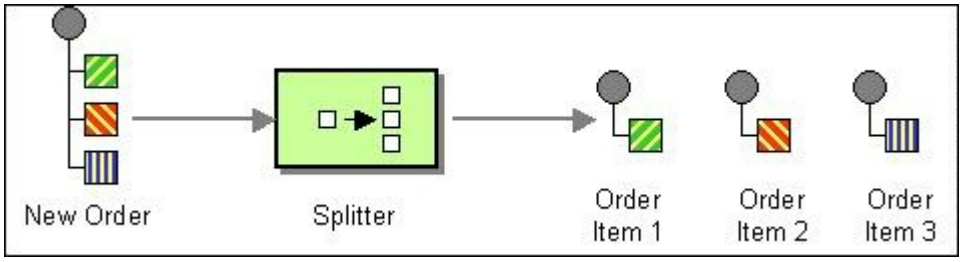
- **Sequence Mediator**: The Sequence Mediator is used to invoke a named sequence. Here we define different named sequences per card type. Each named sequence will know how to do the message transformation.

```
<sequence key="name"/>
```

# Splitter (Simple)

The Splitter EIP explains how to handle a scenario where the incoming request brings multiple elements in it and each element needs to be handled in a different manner.

The following image extracted from **Gregor Hohpe's** *Enterprise Integration Patterns* book illustrates a new order request from the input channel being split into multiple order items in the output channel:




## Getting ready

Let's take an `InventoryManagementService` as an example. Each store at the end of the day does bulk updates talking to the `InventoryManagementService` running at the head office. Each update request will include a set of item records. Items are handled by different business services based on the item code. `InventoryManagementService` running in the ESB will split the message by item and based on the item code, will route that item to the corresponding business service for further processing.

## How to do it...

1. Setup `ItemFooProcessingService` and `ItemBarProcessingService`. We need to have these business services up and running, so WSO2 ESB can route messages to them.
2. Get `ItemFooProcessingService.aar` and `ItemBarProcessingService.aar` from `SAMPLE-3` and copy those to `wso2esb-4.8.0/samples/axis2Server/repository/services/`. If there is no `services` directory, create one.
3. Start simple Axis2Server from `wso2esb-4.8.0/samples/axis2Server`.
4. Validate whether the two business services are up and running.
5. Start WSO2 ESB and login with `admin/admin`. The ESB will start on `https://localhost:9443`.
6. Get `synapse.xml` from `SAMPLE-3`, copy the content of it, go to **Main** | **Manage** | **Service Bus** | **Source View**, and paste it there, overriding the existing and click on **Update**.

7. The previous step will create a proxy service called `InventoryManagementService` in the ESB, which will route messages to the other two business services running in simple Axis2Server.
8. Go to **Main** | **Manage** | **Services** | **List**. You should be able to see `InventoryManagementService` listed there.
9. Now let's test our setup. We'll be using cURL.

```
$ curl -d @request.xml -H "Content-Type:
application/soap
+xml action=process"
http://localhost:8280/services/InventoryManagementService
```

You can get `request.xml` from `SAMPLE-3`.

# How it works...

Let's have a deep look at the synapse configuration. The following explains the key configuration elements:

- `ItemFooProcessingService` and `ItemBarProcessingService` business services do not return back to the ESB. They have in-only/void operations. To convey this to the ESB, we use the following two properties. Even though it's in-only for the business services, it's out-only for the ESB.

```
<property name="FORCE_SC_ACCEPTED"
value="true"scope="axis2"/>
<property name="OUT_ONLY" value="true"/>
```

- The following mediators are used to implement the Splitter EIP:
  - **Clone Mediator**: As its name implies the `<clone>` mediator can be used to create multiple identical copies of a given message and to process those messages in different ways.

```
<clone [id="id"] [continueParent=(true |
false)]
       [sequential=(true | false)]>
 <target [to="uri"]
  [soapAction="qname"]
  [sequence="sequence_ref"]
  [endpoint="endpoint_ref"]>
  <sequence>
   (mediator)+
  </sequence>?
  <endpoint>
   endpoint
  </endpoint>?
 </target>+
```

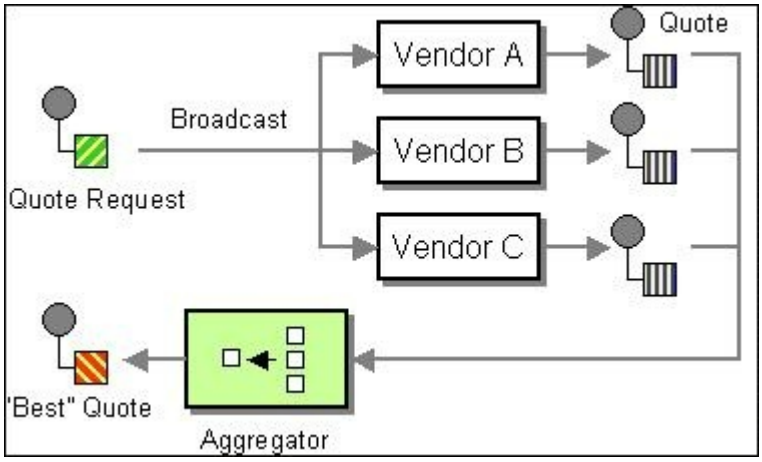
```
        </clone>
```

- **Payload Factory Mediator**: The `<payloadFactory>` mediator is used here within the <`clone`> mediator. Once the <`clone`> mediator creates a copy of the incoming message, the <`payloadFactory`> mediator is used to create different outgoing messages per each business service.
- **Drop Mediator**: The `<drop>` mediator is used to stop further processing of the current message. Here we use the `<drop/>` mediator inside `outSequence,` as we do not expect to return anything back to the client.

# Scatter and Gather (Simple)

The Scatter and Gather EIP explains how to handle a scenario where the incoming request has to be handled by multiple recipients and each recipient will reply back to form an aggregated response.

The following image extracted from the *Enterprise Integration Patterns* book by *Gregor Hohpe* illustrates a quote response from different vendors been gathered and aggregated to form the best quote:



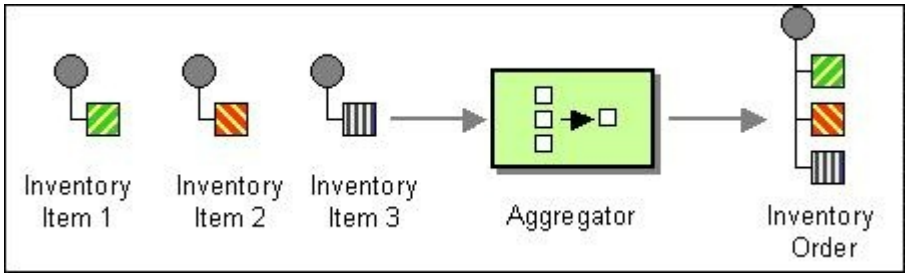

## Getting ready

Let's take a `CabReservationService` as an example. The `CabReservationService` will internally call three other business services to find the best taxi fare for a given destination.

We can implement the Scatter and Gather EIP to scatter the incoming message and send it across to these three business services and to aggregate the results from them to form the aggregated response, which will include the best taxi fare.

The Scatter and Gather EIP uses the Aggregator EIP internally to gather the responses to form the aggregated result. The Aggregator EIP talks about combining the results of individual, but related messages, so the result can be processed as a whole.

The following image extracted from the *Enterprise Integration Patterns* book by *Gregor Hohpe* illustrates different inventory items being aggregated to form the inventory order:




# How to do it...

1. Set up `YellowCabService`, `NewYorkCityCabService`, and `KangarooCabService`. We need to have these business services up and running, so WSO2 ESB can route messages to them.
2. Get `YellowCabService.aar`, `NewYorkCityCabService.aar`, and `KangarooCabService.aar` from `SAMPLE-4` and copy those to `wso2esb-4.8.0/samples/axis2Server/repository/services/`. If there is no `services` directory, create one.
3. Start simple Axis2Server from `wso2esb-4.8.0/samples/axis2Server.sh`.
4. Validate whether the three business services are up and running by accessing their WSDLs.
5. Start WSO2 ESB and login with username as `admin` and password also as `admin`. The ESB will start on `https://localhost:9443`.
6. Get `synapse.xml` from `SAMPLE-4`, copy the content of it, go to **Main** | **Manage** | **Service Bus** | **Source View**, and paste it there, overriding the existing and click on **Update**.
7. The above will create a proxy service called `CabReservationService` in the ESB, which will route messages to the three business services running in simple Axis2Server.
8. Go to **Main** | **Manage** | **Services** | **List**. You should be able to see `TripBookingService` listed there.
9. Now let's test our setup with cURL.

```
$ curl -d @request.xml -H "Content-Type:
application/soap
+xml action=findBestCabDeal"
http://localhost:8280/services/CabReservationService
```

You can get `request.xml` from `SAMPLE-4`.

# How it works...

Let's have a deep look at the synapse configuration. The following explains key configuration elements:

- The following mediators are used to implement the Scatter and Gather pattern with WSO2 ESB.
    - **Clone Mediator**: The `<clone/>` mediator is used here to clone the incoming message and send in parallel to three other different cab services to find the best deal.
    - **Aggregate Mediator**: The `<aggregate/>` mediator is used within `outSequence` to gather all the responses and to find out the best deal. Once the `completeCondition` is satisfied the `onComplete` block will get executed.

```
<aggregate [id="id"]>
  <correlateOn expression="xpath"/>?
  <completeCondition
  [timeout="time-in-seconds"]>
  <messageCount
  min="int-min" max="int-max"/>?
  </completeCondition>?
  <onComplete expression=»xpath»
  [sequence=»sequence-ref»]>
  (mediator +)?
  </onComplete>
</aggregate>
```

    - **Enrich Mediator**: The `<enrich/>` mediator can process a message based on a given source configuration and then perform the specified action on the message by using the target configuration. In our case, this is used to build the aggregated response.

```
<enrich>
  <source [clone=true|false]

 [type=custom|envelope|body|property|
inline]
  xpath="" property="" />
  <target [action=replace|child|sibiling]

 [type=custom|envelope|body|property|
inline]
  xpath="" property="" />
</enrich>
```

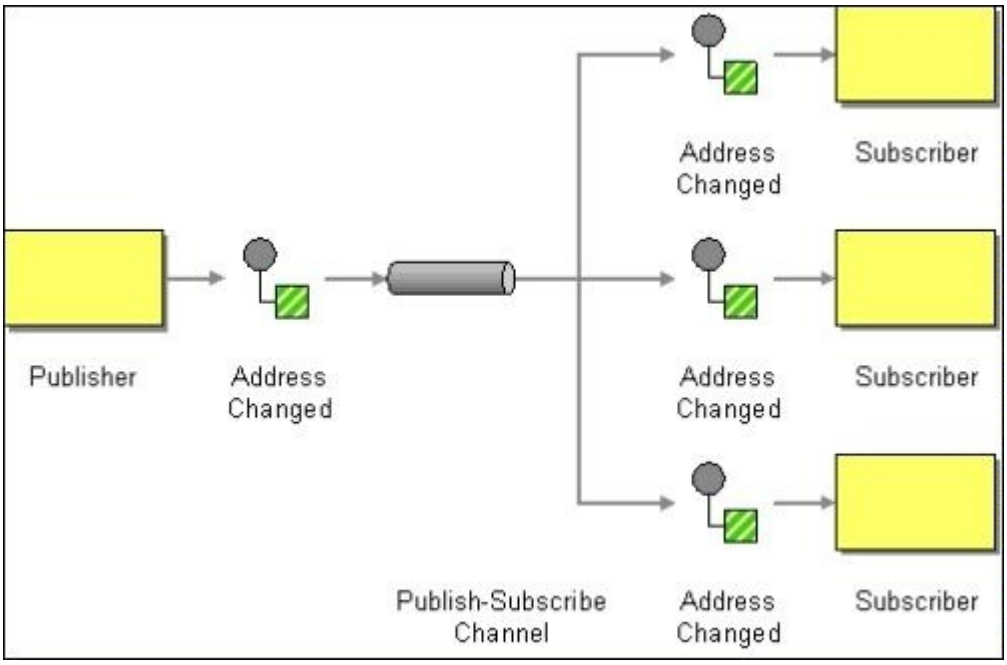
    - **Payload Factory Mediator**: Here we share the same WSDL for all the business services. But, the proxy service has a different WSDL. The `<payloadFactory>` mediator is used to craft the response from the proxy service to the client.

# Publish and Subscribe (Simple)

The Publish & Subscribe EIP explains how to handle a scenario where one needs to publish events to all the interested parties without maintaining any hard coupling between them.

## Getting ready

The following image extracted from the *Enterprise Integration Patterns* book by *Gregor Hohpe* illustrates an address changed message being sent to a set of subscribers who have subscribed to receive address changed messages:




If you are a frequent air traveller, you must be definitely worried about getting updates about your flight status. There are many channels that you can subscribe to. It can be over SMS, E-mail or even you can have an Apple or Android app on your mobile phone. Also, you can see electronic flight status boards hanging at the airport itself.

All these flight status updates are generated from a single source and broadcast to all the interested parties. If you are interested in getting updates, you can subscribe to the channel you want. Also you should be able to set filters only to get the updates you want, maybe by the airport or airline.

## How to do it...

Let's see how to implement the Publish & Subscribe pattern using WSO2 ESB.

1. Setup `AAFlightStatusAlertService`, `DeltaFlightStatusAlertService`, and `EmiratesFlightAlertService`. These are the business services that subscribe to the main `FlightStatusService` with a filter by the airline. So the `AAFlightStatusAlertService` will only get flight status updates for American Airlines and `DeltaFlightStatusAlertService` for Delta flights.
2. Get `AAFlightStatusAlertService.aar`, `DeltaFlightStatusAlertService.aar`, and `EmiratesFlightAlertService.aar` from `SAMPLE-5` and copy those to `wso2esb-4.8.0/samples/axis2Server/repository/services/.` If there is no `services` directory, create one.
3. Start simple Axis2Server from `wso2esb-4.8.0/samples/axis2Server`.
4. Validate whether the three business services are up and running by accessing their WSDLs.
5. Start WSO2 ESB and login with the username as `admin` and the password also as `admin`. The ESB will start on `https://localhost:9443`.
6. Go to **Main** | **Manage** | **Topics** | **Add**. Type `flightStatus` as the name of the topic and click on **Add Topic**.
7. Go to **Main** | **Manage** | **Topics** | **Browse**. You will see the `flightStatus` topic that we just created, listed there. Click on it and then on **Add Subtopic**.
8. Create a subtopic with the name `aa` and click on **Add Topic**.
9. Repeat steps 7 and 8 to create two more subtopics with the names `emirates` and `delta`.
10. Go to **Main** | **Manage** | **Topics** | **Browse**. You will see the `aa` subtopic under the `flightStatus` topic that we just created, listed there. Click on it and then on **Subscribe**.
11. Pick **Topic and Children** for the **Subscription Mode**.
12. Type `http://localhost:9000/services/AAFlightStatusAlertService` as the **Event Sink Url**.
13. Keep the **Expiration Time** blank and click on **Subscribe** to complete the first subscription.
14. Repeat steps 10 to 13 for `DeltaFlightStatusAlertService` and `EmiratesFlightAlertService`. Change steps 10 and 12 appropriately to point to the corresponding subtopic and the corresponding service.
15. Get `synapse.xml` from `SAMPLE-5`, copy the content of it, go to **Main** | **Manage** | **Service Bus** | **Source View** and paste it there, overriding the existing and click on **Update**.

16. The previous step will create a proxy service called `FlightStatusService` in the ESB, which will route messages to the three business services running in simple Axis2Server.
17. Go to **Main** | **Manage** | **Services** | **List**. You should be able to see `FlightStatusService` listed there.
18. Now let's test our setup with cURL.

```
$ curl -d @request-aa.xml -H "Content-Type:
application/soap
+xml action=updateStatus"
http://localhost:8280/services/FlightStatusService

$ curl -d @request-ek.xml -H "Content-Type:
application/soap
+xml action=updateStatus"
http://localhost:8280/services/FlightStatusService

$ curl -d @request-dl.xml -H "Content-Type:
application/soap
+xml action=updateStatus"
http://localhost:8280/services/FlightStatusService
```

You can get `request-aa.xml`, `request-ek.xml`, and `request-dl.xml` from `SAMPLE-5`.

## How it works...

The following mediators are used to implement the Publish and Subscribe pattern with WSO2 ESB.

- **Event Mediator**: The `<event/>` mediator is used within `inSequence` of the proxy service to publish events to corresponding topics. All the events are pumped into `FlightStatusService` and via the `<event/>` mediator `FlightStatusService` publishes events to the subscribed parties.

```
<event
xmlns="http://ws.apache.org/ns/synapse"
topic="" [expression=""] />
```

- **Switch Mediator**: The `<switch/>` mediator is used to identify the incoming messages and based on that the `<event/>` mediator will pick the topic to publish the event.

# Service Chaining (Intermediate)

Service Chaining is one of the common and popular use case for an ESB to support. To cater for a request from the client, the ESB may have to call a chain of business services. This is different from the Scatter and Gather pattern we covered before. The Scatter and Gather EIP explains how to handle a scenario where the incoming request has to be handled by multiple recipients and each recipient will reply back to form an aggregated response. Service Chaining does more intelligent decision making and simply does which is very much similar to service orchestration.

## Getting ready

Let's take `TravelManagementService` as an example. To cater a travel arrangement request from the client, `TravelManagementService` has to call `AirLineReservationService` with the provided dates for travelling. If that succeeds it will call `RentACarService` and finally it will call the `HotelReservationService` to reserve a hotel.

## How to do it...

1. Setup `AirLineReservationService`, `RentACarService`, and `HotelReservationService`. We need to have these business services up and running, so WSO2 ESB can route messages to them.
2. Get `AirLineReservationService.aar`, `RentACarService.aar`, and `HotelReservationService.aar` from `SAMPLE-6` and copy those to `wso2esb-4.8.0/samples/axis2Server/repository/services/`. If there is no `services` directory create one.
3. Start simple Axis2Server from `wso2esb-4.8.0/samples/axis2Server`.
4. Validate whether the three business services are up and running by accessing their WSDLs.
5. Start WSO2 ESB and login with username as `admin` and password also as `admin`. The ESB will start on `https://localhost:9443`.
6. Get `synapse.xml` from `SAMPLE-6`, copy the content of it, go to **Main** | **Manage** | **Service Bus** | **Source View**, and paste it there, overriding the existing and click on **Update**.
7. The above will create a proxy service called `TravelManagementService` in the ESB, which will route messages to the three business services running in simple Axis2Server.
8. Go to **Main** | **Manage** | **Services** | **List**. You should be able to see `TravelManagementService` listed there.
9. Now let's test our set up with cURL.

```
$ curl -d @request.xml -H "Content-Type:
application/soap
```

```
+xml action=arrangeMyTravel"
http://localhost:8280/services/TravelManagementService
```

You can get `request.xml` from `SAMPLE-6`.

# How it works...

Let's have a deep look at the synapse configuration. The following explains key configuration elements:

- When the request hits `inSequence` we have to first call the `AirLineReservationService` and need to process its response. If the response is true, only we will proceed to call `RentACarService`. The same applies when we call `HotelReservationService`.

```
<send receive="rentACarSeq">
  <endpoint>
    <address
uri="…/AirLineReservationService"/>
  </endpoint>
</send>
```

- The `receive` attribute in the `<send/>` mediator will make sure the response from calling endpoint goes to the `<sequence/>` defined with that particular name. In this case, the response here will hit the `rentACarSeq` sequence. If we do not define this attribute then the response will directly go to the `outSequence`.
- The `rentACarSeq` sequence has to process the response from the `AirLineReservationService`. If it's false, then it will call the `processResponse` sequence to send a message back to the client, without further processing. If the response is true, then `rentACarSeq` will proceed to call `RentACarService`.

```
<sequence name="rentACarSeq">
  <log level="full"/>
  <switch xmlns:xsd="..."
    xmlns:ns="..."

 source="//ns:reserveAirLineResponse/ns:return">
    <case regex="false">
      <sequence key="processResponse"/>
    </case>
    <default>
      <send receive="reserveHotelSeq">
        <endpoint>
          <address uri="…/RentACarService"/>
        </endpoint>
      </send>
    </default>
  </switch>
```

```
</sequence>
```

- The `processResponse` sequence sends the response back to the client. Here we are using the `<payloadFactory/>` mediator to craft the response. Also when you want to send a response back to the client using the `<send/>` mediator we need to set the `RESPONSE` property to true prior to that.

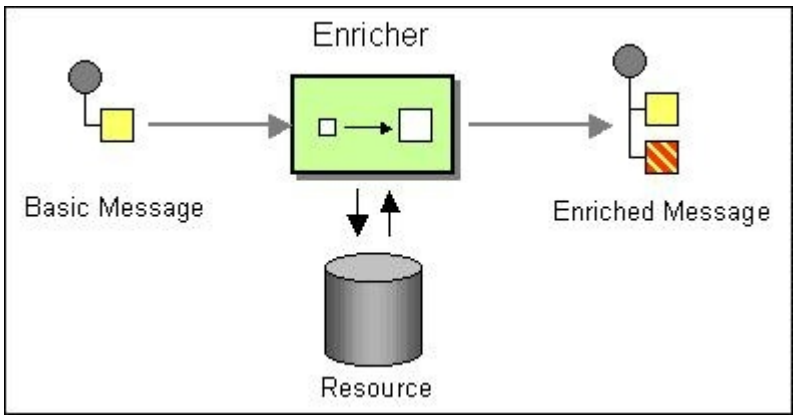
```
<sequence name="processResponse">
  <payloadFactory>
    <format>
      <res:arrangeMyTravel xmlns:res="...">
      <res:return>$1</res:return>
      </res:arrangeMyTravel>
    </format>
    <args>
      <arg evaluator="xml"

 expression="$ctx:businessServiceResponse"/>
    </args>
  </payloadFactory>
  <property name="RESPONSE" value="true"
scope="default"
  type="STRING"/>
  <send/>
</sequence>
```

- A better approach to send back the response to the client was introduced from WSO2 ESB 4.8.0 onwards. There you need to use the `<respond>` mediator instead of the above.

# Content Enricher (Intermediate)

The Content Enricher EIP explains how to handle a scenario where a middleman has to inject some missing information into the message, which is required by the backend business service for processing. The end user or the client may not have this information beforehand with him to add to the request. The Content Enricher will intercept the message, retrieve the relevant data from a data source and inject it into the message.




## Getting ready

Let's take `CreditCardApplicationService` as an example. Assume credit cards are issued only for trusted bank customers. The trust level for a given customer is calculated based on his/her activities done with the bank in the past. It is not visible to the customer and only known to the bank. Once `CreditCardApplicationService` gets the request from the customer, it has to call the `CreditCardApplicationProcessingService` to process the request. This service expects the trust level of the customer in the request, in addition to the other customer information.

## How to do it...

1. Set up `CreditCardApplicationProcessingService`. We need to have this SOAP based business service up and running, so WSO2 ESB can route messages to it.
2. Get `CreditCardApplicationProcessingService.aar` from `SAMPLE-7` and copy those to `wso2esb-4.8.0/samples/axis2Server/repository/services/`. If there is no `services` directory create one.
3. Start simple Axis2Server from `wso2esb-4.8.0/samples/axis2Server`.
4. Validate whether the business service is up and running by verifying the presence of its WSDL.

5. Start MySQL server; run `sample7.sql` from `SAMPLE-7`. This will create a database with the name `CustomerDB` and a table with some sample data to test our scenario.
6. Copy the MySQL driver file `mysql-connector-java-5.1.26-bin.jar` from `SAMPLE-7` to `wso2esb-4.8.0/repository/components/lib`.
7. Start WSO2 ESB and login with username as `admin` and password also as `admin`. The ESB will start on `https://localhost:9443`.
8. Get `synapse.xml` from `SAMPLE-7`, copy the content of it, go to **Main** | **Manage** | **Service Bus** | **Source View**, and paste it there, overriding the existing and click on **Update**.
9. The above will create a proxy service called `CreditCardApplicationService` in the ESB, which will talk to the `CustomerDB` to get the corresponding trust level of the end user and invoke the business service running in simple Axis2Server for further processing.
10. Go to **Main** | **Manage** | **Services** | **List**. You should be able to see `CreditCardApplicationService` listed there.
11. Now let's test our setup with cURL.

```
$ curl -d @request.xml -H "Content-Type:
application/soap
+xml action=apply"
http://localhost:8280/services/CreditCardApplicationService
```

You can get `request.xml` from `SAMPLE-7`.

# How it works...

Let's have a deep look at the synapse configuration. The following explains key configuration elements.

- When the request hits `inSequence` of `CreditCardApplicationService` we have to first get the `trustLevel` corresponding to the user (from the request) from the database and inject that into the request going to the `CreditCardApplicationProcessingService`. To access the database we use `<dblookup/>` mediator. Inside the `<dblookup/>` element we execute a query against the `CustomerDB` and store the results in the `<result/>` element.

```
<dblookup>
  <connection>
    <pool>
      <password>mysql</password>
      <user>root</user>

 <url>jdbc:mysql://localhost:3306/CustomerDB</url>
```

```
      <driver>com.mysql.jdbc.Driver</driver>
    </pool>
  </connection>
  <statement>
    <sql>SELECT trust_level FROM
CustomerDB.customer wherename = ?</sql>
    <parameter
xmlns:crd="http://crd.services.esb.wso2.packt.com"
    expression="//crd:apply/crd:userName"
    type="VARCHAR"/>
    <result name="trust_level"
column="trust_level"/>
  </statement>
</dblookup>
```

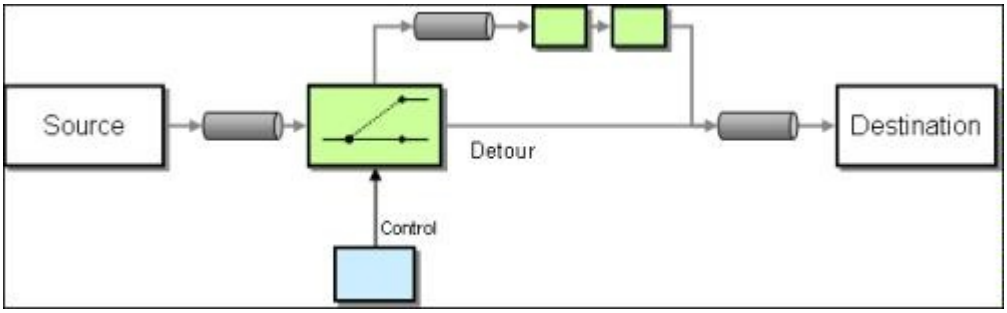
- Here you can see the `result` value retrieved from the database lookup through the `<dblookup/>` mediator is stored as a property. This property is read inside the `<payloadFactory/>` mediator to build the message to send to the `CreditCardApplicationProcessingService`.

```
<payloadFactory media-type="xml">
  <format>
    <crd:apply xmlns:crd="..">
    <crd:userName>$1</crd:userName>
    <crd:trustLevel>$2</crd:trustLevel>
    </crd:apply>
  </format>
  <args>
    <arg xmlns:crd=".."  evaluator="xml"
      expression="//crd:apply/crd:userName"/>
    <arg evaluator="xml"

 expression="get-property('trust_level')"/>
  </args>
</payloadFactory>
```

# Detour (Intermediate)

The Detour EIP explains how to treat a message in different ways based on a certain state of the message. In a normal scenario it will flow through the default route. Under some circumstances it will take a different route. This optional route can be picked on certain criteria. It can be statically configured or dynamically picked based on the message content.




## Getting ready

Let's take an `OrderProcessingService` as an example. Orders can be placed by anyone outside the company by invoking the `OrderProcessingService`. Or else customers can come to the company and hand over an order request in a paper to the staff. In the second case, staff will use a data entry client to add the order details to the system, which will also eventually hit the `OrderProcessingService`. Anyone from outside using the publicly exposed service can initiate a **Denial of Service** (**DoS**) attack against the system by sending large invalid XML payloads. So we need to have an extra check before we process an order. This is not required for the orders generated from the data entry client from the internal network as it's within our trust domain. Based on the IP address of the request we can differentiate a request generated inside from one coming through the public Internet.

## How to do it...

1. Set up `BusinessOrderProcessingService`. We need to have this business service up and running, so WSO2 ESB can route messages to it.
2. Get `BusinessOrderProcessingService.aar` from `SAMPLE-8` and copy it to `wso2esb-4.8.0/samples/axis2Server/repository/services/`. If there is no `services` directory create one.
3. Start simple Axis2Server from `wso2esb-4.8.0/samples/axis2Server`.
4. Validate whether the business service is up and running by accessing its WSDL.

5. Start WSO2 ESB and login with username as `admin` and password also as `admin`. The ESB will start on `https://localhost: 9443`.
6. Get `synapse.xml` from `SAMPLE-8`, copy the content of it, go to **Main** | **Manage** | **Service Bus** | **Source View**, and paste it there, overriding the existing and click on **Update**.
7. The above will create a proxy service called `OrderProcessingService` in the ESB, which will route messages to the business service running in simple Axis2Server.
8. Go to **Main** | **Manage** | **Services** | **List**. You should be able to see `OrderProcessingService` listed there.
9. Now let's test our setup with cURL.

```
$ curl -d @request.xml -H "Content-Type:
application/soap
+xml action=placeOrder"
http://localhost:8280/services/OrderProcessingService
```

You can get `request.xml` from `SAMPLE-8`.

# How it works...

Let's have a deep look at the synapse configuration. The following explains key configuration elements:

- When the request hits `inSequence` of `OrderProcessingService` we have to first find the IP address of the client and then decide whether we should validate the message or not. If the message is initiated from a private IP address, then we will not validate it.

```
<property name="IP_ADDRESS"
expression="get-
property('axis2','REMOTE_ADDR')"/>
<filter source="get-property('IP_ADDRESS')"
regex="..">
  <then/>
  <else />
</filter>
```

The regular expression we use here to identify a private IP address is `(^127\.0\.0\.1)|(^10\.)|(^172\.1[6-9]\.)|(^172\.2[0-9]\.)|(^172\.3[0-1]\.)|(^192\.168\.)`.

- If the message is coming from outside to our private domain, then we will use the `<validate/>` mediator to validate the incoming messages against a schema. If the validation fails, then we set a property (which will be read later in the message flow) to indicate this is an invalid order.

```
<validate>
```

```
  <schema key="validate_order"/>
  <on-fail>
    <property name="INVALID_ORDER"
value="true"
      scope="default" type="STRING"/>
  </on-fail>
</validate>
```

- The `validate_order` schema key is defined as a `localEntry`. We are validating all the incoming messages from external parties against this XML schema.

```
<localEntry key="validate_order">
  <xs:schema xmlns:xs=".."
    xmlns=".."
    xmlns:mcd=".."
    elementFormDefault="qualified"
    attributeFormDefault="unqualified"
    targetNamespace="..">
  </xs:schema>
</localEntry>
```

- If the order is invalid we send back a SOAP fault to the client. A SOAP fault is generated using the `<makefault/>` mediator.

```
<makefault version="soap11">
  <code xmlns:tns=".." value="tns:Receiver"/>
  <reason value="Invalid Order"/>
</makefault>
```

- When we want to send a response back to the client using the `<send/>`mediator we need to set the `RESPONSE` property to true prior to that.

```
<property name="RESPONSE" value="true"
scope="default"
type="STRING"/>
<send/>
```

- A better approach to send back the response to the client was introduced from WSO2 ESB 4.8.0 onwards. There you need to use the `<respond>` mediator instead of the ones mentioned earlier.

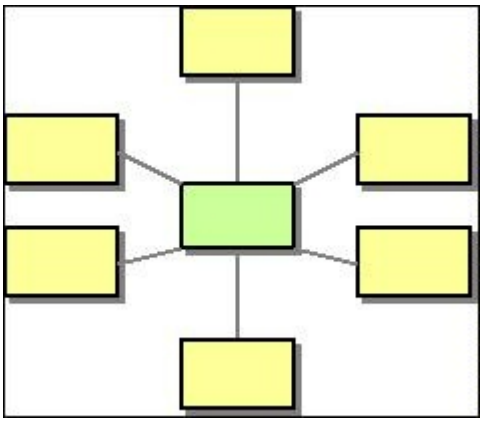
# Chapter 3. Integration with Third-Party Message Brokers

This chapter explains how to integrate WSO2 ESB with third-party message brokers. First we will start with Apache ActiveMQ and show how WSO2 ESB can consume messages from an ActiveMQ queue. Then we will see how WSO2 ESB acts as a publisher. Later we will explain how we can integrate Apache Qpid as a message broker to the WSO2 ESB message store.

## WSO2 ESB (subscriber) with Apache ActiveMQ Message Broker (Intermediate)

The Message Broker Enterprise Integration Pattern explains how to decouple the destination of a message from the sender and maintain central control over the flow of messages. With this approach the sender does not need to know exactly who the recipient is, so it scales up nicely as the number of recipients for a given message can grow without the sender being aware of it. This pattern is also known as a **hub-and-spoke** architectural style.

The following image, extracted from the *Enterprise Integration Patterns* book by *Gregor Hohpe*, illustrates the hub-and-spoke architectural style:



### Getting ready

Let's take the same example that we took under the *Publish and Subscribe* recipe, in Chapter 2, *Enterprise Integration Patterns*. There the updates are initially published to the `FlightStatusService` proxy service deployed in

WSO2 ESB and within the `FlightStatusService` proxy, it uses the `<event/>` mediator to publish events to the `DeltaFlightStatusAlertService` and `EmiratesFlightAlertService`.

There, the `FlightStatusService` proxy is exposed over HTTP and the client directly sends messages to the proxy. In this example, we are going to change that behavior. Instead of sending messages directly to the proxy, the client will send messages to a JMS Queue in ActiveMQ Message Broker and `FlightStatusService` will pick messages from the JMS Queue and send those to its subscribers.

# How to do it...

Let's see how to integrate Apache ActiveMQ with WSO2 ESB to implement the Message Broker EIP:

1. We need to set up Apache ActiveMQ in our environment. You can download the latest stable version of Apache ActiveMQ from http://activemq.apache.org/activemq-580-release.html. At the time of writing, the latest available version is 5.8, which is used in this book.

   > **Note**
   >
   > You can refer to http://activemq.apache.org/getting-started.html to get ActiveMQ installed and get started.

   Under Linux, to start ActiveMQ, type the following from `apache-activemq-5.8.0/bin`:

   ```
   $ ./activemq start
   ```

   To find the status of ActiveMQ, run the following command:

   ```
   $ ./activemq status
   ```

2. Copy the following jars from `apache-activemq-5.8.0/lib/` to `wso2esb-4.8.0/repository/components/lib/`:

   - `activemq-client-5.8.0.jar`
   - `geronimo-j2ee-management_1.1_spec-1.0.1.jar`
   - `geronimo-jms_1.1_spec-1.1.1.jar`

3. Go to `wso2esb-4.8.0/repository/conf/axis2/axis2.xml` and uncomment on the following section. Make sure that you uncomment

the section related to ActiveMQ. There are two other JMSListeners defined and commented out for Apache Qpid and WSO2 MB.

```
<transportReceiver name="jms"
class="org.apache.axis2.transport.jms.JMSListener">
  ---------
</transportReceiver>
```

This assumes ActiveMQ is running on localhost with port number `61616`. If that is not the case you need to change the value of the `java.naming.provider.url` parameter in all three occurrences.

4. Start WSO2 ESB and you will see the following log entries at the start-up:

```
INFO - JMSConnectionFactory JMS
ConnectionFactory: myTopicConnectionFactory
initialized
INFO - JMSConnectionFactory JMS
ConnectionFactory: myQueueConnectionFactory
initialized
INFO - JMSConnectionFactory JMS
ConnectionFactory: default initialized
INFO - JMSListener JMS Transport
Receiver/Listener initialized...
```

5. Get `AAFlightStatusAlertService.aar`, `DeltaFlightStatusAlertService.aar`, and `EmiratesFlightAlertService.aar` from `[SAMPLE-9]`, and copy those to `wso2esb-4.8.0/samples/axis2Server/repository/services/`. If there is no services directory, create one.
6. Start simple Axis2Server from `wso2esb-4.8.0/samples/axis2Server`.
7. Validate whether the three business services are up and running by accessing their WSDLs.
8. Start WSO2 ESB and log in with username as `admin` and password also as `admin`. The ESB will start on `https://localhost:9443`.
9. Go to **Main** | **Manage** | **Topics** | **Add**. Type `flightStatus` as the name of the topic and click on **Add Topic**.
10. Go to **Main** | **Manage** | **Topics** | **Browse**. You will see the `flightStatus` topic that we just created, listed there. Click on it and then on **Add Subtopic**.
11. Create a subtopic with the name `aa` and click on **Add Topic**.
12. Repeat steps 10 and 11 to create two more subtopics with the names `emirates` and `delta`
13. Go to **Main** | **Manage** | **Topics** | **Browse**. You will see the `aa` subtopic listed under the `flightStatus` topic that we just created. Click on it and then on **Subscribe**.
14. Pick **Topic and Children** for the **Subscription Mode**.

15. Type `http://localhost:9000/services/AAFlightStatusAlertService` as the **Event Sink Url**.
16. Keep **Expiration Time** blank and click on **Subscribe** to complete the first subscription.
17. Repeat steps 13 to 16 for `DeltaFlightStatusAlertService` and `EmiratesFlightAlertService`. Change steps 13 and 15 appropriately to point to the corresponding subtopic and the corresponding service.
18. Get `synapse.xml` from `[SAMPLE-9]`, copy the content of it, go to **Main** | **Manage** | **Service Bus** | **Source View**, and paste it there, overriding the existing and update.
19. The preceding procedure will create a proxy service called `FlightStatusService` in the ESB, which will route messages to the three business services running in simple Axis2Server.
20. Go to **Main** | **Manage** | **Services** | **List**. You should be able to see `FlightStatusService` listed there.
21. Now let's test our setup with the `FlighStatusJMSClient`:

```
$ java -Djms_payload=request-aa.xml -cp
activemq-all-
5.8.0.jar:com.packt.wso2.esb.client.jms.jar
com.packt.wso2.esb.client.jms.FlighStatusJMSClient

$ java -Djms_payload=request-ek.xml -cp
activemq-all-
5.8.0.jar:com.packt.wso2.esb.client.jms.jar
com.packt.wso2.esb.client.jms.FlighStatusJMSClient

$ java -Djms_payload=request-dl.xml -cp
activemq-all-
5.8.0.jar:com.packt.wso2.esb.client.jms.jar
com.packt.wso2.esb.client.jms.FlighStatusJMSClient
```

You can get `request-aa.xml`, `request-ek.xml`, `request-dl.xml`, `activemq-all-5.8.0.jar`, and `com.packt.wso2.esb.client.jms.jar` from `[SAMPLE-9]`.

# How it works...

Let's have a look at the Synapse configuration:

- In the proxy service definition we have enabled `jms` transport.

```
<proxy name="FlightStatusService"
transports="jms" startOnLoad="true"
trace="disable">
```

By default, the ESB will subscribe to a queue with the name of the proxy service, once the preceding snippet is set.

- If we want to override the above behavior, or to use a queue different from the proxy service name, then we need to set the following service parameter. Add the following just after the `<proxy>` definition:

```
<parameter
name="transport.jms.Destination">wso2packtqueue</parameter>
```

- Now, let's run the client with the system property `jms_dest`:

```
$ java
-
Djms_dest=dynamicQueues/wso2packtqueue  -
Djms_payload=request-
ek.xml -cp activemq-all-
5.8.0.jar:com.packt.wso2.esb.client.jms.jar
com.packt.wso2.esb.client.jms.FligHStatusJMSClient
```

# WSO2 ESB (publisher) with Apache ActiveMQ Message Broker (Intermediate)

In the previous section, WSO2 ESB was acting as a subscriber to the ActiveMQ Message Broker. WSO2 ESB was picking messages from a queue defined in ActiveMQ and sending them across to its own subscribed web services over SOAP/HTTP. Here we will see how WSO2 ESB can publish to a queue defined in ActiveMQ. WSO2 ESB will receive messages over SOAP/HTTP and then it will publish them to a queue in ActiveMQ.

## Getting ready

In the previous recipe, *WSO2 ESB (subscriber) with Apache ActiveMQ Message Broker*, we had three SOAP-based business services where WSO2 ESB will route messages to. Here, instead of the three SOAP-based services, WSO2 ESB will send messages to three different message queues.

## How to do it...

1. Make sure ActiveMQ is up and running as explained in the previous recipe, WSO2 ESB (subscriber)with the Apache ActiveMQ Message Broker.
2. Shut down WSO2 ESB, if it's already running.
3. Go to `wso2esb-4.8.0/repository/conf/axis2/axis2.xml` and uncomment the following section:

```
<transportSender name="jms"
class="org.apache.axis2.transport.jms.JMSSender"/>
```

4. Copy the following jars from `apache-activemq-5.8.0/lib/` to `wso2esb-4.8.0/repository/components/lib/`:

    - `activemq-client-5.8.0.jar`
    - `geronimo-j2ee-management_1.1_spec-1.0.1.jar`
    - `geronimo-jms_1.1_spec-1.1.1.jar`

5. Start WSO2 ESB and log in with username as `admin` and password also as `admin`. The ESB will start at `https://localhost:9443`.
6. Get `synapse.xml` from `[SAMPLE-10]`, copy the content of it, go to **Main** | **Manage** | **Service Bus** | **Source View**, and paste it there, overriding the existing and update.

7. The preceding step will create a proxy service called `FlightStatusService` in the ESB, which will route messages to the three message queues in ActiveMQ.
8. Go to **Main** | **Manage** | **Services** | **List**. You should be able to see `FlightStatusService` listed there.
9. Now let's test our setup with cURL.

```
$ curl -d @request-aa.xml -H "Content-Type:
application/soap
+xml action=updateStatus"
http://localhost:8280/services/FlightStatusService

$ curl -d @request-ek.xml -H "Content-Type:
application/soap
+xml action=updateStatus"
http://localhost:8280/services/FlightStatusService

$ curl -d @request-dl.xml -H "Content-Type:
application/soap
+xml action=updateStatus"
http://localhost:8280/services/FlightStatusService
```

   You can get `request-aa.xml`, `request-ek.xml`, and `request-dl.xml` from `[SAMPLE-10]`.

10. Now we can log in to the ActiveMQ administrator console and see the messages in queues.
11. Go to `http://localhost:8161/admin` and log in with the default ActiveMQ username and password. Then click on **Queues**. You will find three queues being created, `emirates`, `delta`, and `aa`. You can click on the queue name to see the messages published on to those.

# How it works...

Let's have a look at the Synapse configuration:

- In the proxy configuration we have enabled HTTP and HTTPS, so it can accept SOAP messages over HTTP/HTTPS.

```
<proxy name="FlightStatusService"
transports="http https" startOnLoad="true"
trace="disable">
```

- Inside the `<send/>` mediator, we use the uri pointing to the corresponding Queue in ActiveMQ.

```
<send>
  <endpoint>
```

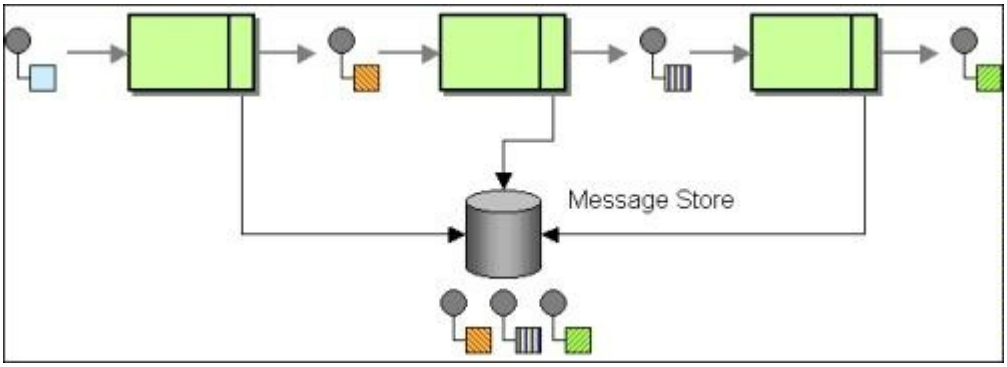
```
  <address
uri="jms:/emirates?
transport.jms.DestinationType=queue&amp;transport.jms.ContentTypePro
  </endpoint>
</send>
```

The preceding snippet publishes the message to the queue `emirates` and we have two similar other entries which will publish messages to `delta` and `aa` queues.

# Message Store with Apache Qpid Message Broker (Intermediate)

The Message Store Enterprise Integration Pattern explains how to capture information about each message in a central location. Also, the Message Store can be used to match the rate limits expected by backend services. There can be clients pumping messages at 500 concurrency but the actual backend service may only be able to handle 300. Then the persistent Message Store at the ESB can be used to persist incoming messages and forward those to the backend service at a compatible rate.

The following image, extracted from the *Enterprise Integration Patterns* book by *Gregor Hohpe*, illustrates the Message Store EIP:




In this section, we are going to explain how to integrate the Apache Qpid Message Broker with WSO2 ESB as the Message Store.

## Getting ready

Let's take the same example that we took under the Publish & Subscribe EIP. There the updates are initially published to the `FlightStatusService` proxy service deployed in WSO2 ESB and within the `FlightStatusService` proxy it uses the `<event/>` mediator to publish events to `AAFlightStatusAlertService`, `DeltaFlightStatusAlertService`, and `EmiratesFlightAlertService`.

There, the `FlightStatusService` proxy is exposed over HTTP and the client directly sends messages to the proxy. In this example, we are going to change that behavior. Instead of the ESB directly sending messages to the backend service, it will write all the incoming messages to a Message Store. Then we will have a Message Processor, which will read messages from the Message Store and forward those to the corresponding business service.

# How to do it...

Let's see how to integrate Apache Qpid with WSO2 ESB to implement the Message Store EIP:

1. We need to set up Apache Qpid in our environment. You can download the latest stable version of Apache Qpid from http://qpid.apache.org/components/java-broker/index.html. At the time of writing this book, the latest available version is 0.20 (`qpid-java-broker-0.20.tar.gz`), which is used in this book.

   > **Note**
   >
   > You can refer to http://qpid.apache.org/getting_started.html to get Apache Qpid installed and get started. The way it manages virtual hosts is different from Qpid 0.20 to 0.22. The instructions presented in this chapter will work with 0.20. In case you are using 0.22 or later, then you need to log in to the Qpid management console by navigating to `http://localhost:8080`. Log in with the default credentials (the username as `admin` and the password also as `admin`) and then click on **Add Virtual Host**. Use `localhost` as the **Name**, click on the **Configuration File** option, give the path to `virtualhosts.xml` from `[SAMPLE-11]` and click on **Save**.

   Under Linux, to start Qpid – type the following from `qpid-broker-0.20/bin`:

   ```
   $ ./qpid-server
   ```

2. Download the `qpid-client` jars (`qpid-java-client-0.20.tar.gz`) from http://qpid.apache.org/components/qpid-jms/index.html and copy all the jars under the `lib` directory to `wso2esb-4.8.0/repository/components/lib/`.
3. Go to `wso2esb-4.8.0/repository/conf/axis2/axis2.xml` and uncomment the following section. Make sure that you uncomment the section related to Apache Qpid. There are two other JMSListeners defined and commented out for Apache ActiveMQ and WSO2 MB.

   ```
   <transportReceiver name="jms"
   class="org.apache.axis2.transport.jms.JMSListener">
   ---------
   </transportReceiver>
   ```

   And also:

```
<transportSender name="jms"
class="org.apache.axis2.transport.jms.JMSSender"/>
```

4. Go to `wso2esb-4.8.0/repository/conf/jndi.properties` and add the following properties to it and override the existing content:

```
connectionfactory.TopicConnectionFactory=amqp://admin:admin@clientID
brokerlist='tcp://localhost:5672'
connectionfactory.QueueConnectionFactory=amqp://admin:admin@clientID
brokerlist='tcp://localhost:5672'
queue.wso2packt=example.wso2packt
topic.wso2packt=example.wso2packt
```

> **Note**
>
> You can read more about the pattern of AMQP connection URL from https://cwiki.apache.org/confluence/display/qpid/Connection+URL+Format.

5. Start WSO2 ESB and you will see the following log entries at the start-up:

```
INFO - JMSConnectionFactory JMS
ConnectionFactory: myTopicConnectionFactory
initialized
INFO - JMSConnectionFactory JMS
ConnectionFactory: myQueueConnectionFactory
initialized
INFO - JMSConnectionFactory JMS
ConnectionFactory: default initialized
INFO - JMSListener JMS Transport
Receiver/Listener initialized...
```

6. Get `AAFlightStatusAlertService.aar`, `DeltaFlightStatusAlertService.aar`, and `EmiratesFlightAlertService.aar` from `[SAMPLE-11]` and copy those to `wso2esb-4.8.0/samples/axis2Server/repository/services/`. If there is no services directory create one.
7. Start simple Axis2Server from `wso2esb-4.8.0/samples/axis2Server`.
8. Validate whether the three business services are up and running by accessing their WSDLs.
9. Start WSO2 ESB and log in with username as `admin` and password also as `admin`. The ESB will start on `https://localhost:9443`.
10. Get `synapse.xml` from `[SAMPLE-11]`, copy the content of it, go to **Main** | **Manage** | **Service Bus** | **Source View**, and paste it there, overriding the existing and update.
11. The preceding procedure will create a proxy service called `FlightStatusService` in the ESB, which will route messages to the

three business services running in simple Axis2Server by picking the messages from the Message Store.

12. Go to **Main** | **Manage** | **Services** | **List**. You should be able to see `FlightStatusService` listed there.
13. Now let's test our setup with the cURL:

```
$ curl -d @request-aa.xml -H "Content-Type:
application/soap
+xml action=updateStatus"
http://localhost:8280/services/FlightStatusService

$ curl -d @request-ek.xml -H "Content-Type:
application/soap
+xml action=updateStatus"
http://localhost:8280/services/FlightStatusService

$ curl -d @request-dl.xml -H "Content-Type:
application/soap
+xml action=updateStatus"
http://localhost:8280/services/FlightStatusService
```

You can get `request-aa.xml`, `request-ek.xml`, and `request-dl.xml` from `[SAMPLE-11]`.

## How it works...

Let's have a look at the Synapse configuration:

- We need to define our message store in the following manner:

```
<messageStore
class="org.apache.synapse.message.store.impl.jms.JmsStore"
name="wso2packt">
  <parameter
name="java.naming.factory.initial">org.apache.qpid.jndi.PropertiesFi
  <parameter
name="store.jms.cache.connection">false</parameter>
  <parameter
name="java.naming.provider.url">repository/conf/jndi.properties</par
  <parameter
name="store.jms.JMSSpecVersion">1.1</parameter>
  <parameter
name="store.jms.destination">wso2packt;
{create:always,
node:{durable:True}}</parameter>
</messageStore>
```

Connection details related to Apache Qpid are read from the `repository/conf/jndi.properties` file. This is the file that we configured in step 4 previously. The value of the parameter

`store.jms.destination` should match the queue name we defined in `repository/conf/jndi.properties`.

- Here we define the Message Processor which talks to the message store and fetches messages from it for further processing.

```
<messageProcessor
class="org.apache.synapse.message.processors.
impl.forwarder.ScheduledMessageForwardingProcessor"
  name="wso2packprocessor"
  messageStore="wso2packt">
  <parameter name="interval">1000</parameter>
</messageProcessor>
```

  Here we set the value of `messageStore` to `wso2packt`, which is the name of the message store we defined previously. The Message Processor will query the messages from the message store in every 1000 milliseconds as per the preceding configuration.

- The following element sets the endpoint to be used by the Message Processor.

```
<property name="target.endpoint"
value="EmiratesEp"/>
```

  We also need to define the actual endpoint corresponding to the preceding code snippet, in the Synapse configuration.

```
<endpoint name="EmiratesEp">
  <address
uri="http://localhost:9000/services/EmiratesFlightAlertService"/>
</endpoint>
```

- Finally, we need to have the `<store/>` mediator added to `inSequence`. So, the incoming messages will be stored in the message store.

```
<store messageStore="wso2packt"/>
```

  Here we set the value of the `messageStore` to `wso2packt`, which is the name of the message store we defined previously.

# Chapter 4. Business Messaging and Transformations

This chapter explains by example, a selected set of capabilities of WSO2 ESB in business messaging and transformations. FIX, HL7, and SAP are covered in the first three sections. WSO2 ESB ships with a rich set of connectors to a wide variety of business applications. Later in this chapter we explain how to use the Twitter connector and then how to transform REST/JSON messages into SOAP and vice-versa.

## Transforming messages from FIX to SOAP (Advanced)

WSO2 ESB is capable of doing numerous types of protocol switching and message transformations. Under this section, we will see how to transform a **Financial Information eXchange** (**FIX**) message generated from a FIX source to a SOAP message over HTTP.

### Getting ready

Let's use the **Banzai** FIX application that comes with **QuickFIX/J**. Banzai is a Java swing application that will generate FIX messages, which will be accepted by the `BanzaiProxy` that we create in WSO2 ESB. `BanzaiProxy` will transform the FIX message to a SOAP message and send it across to the `SimpleStockQuoteService` running in simple Axis2Server.

### How to do it...

1. Download QuickFIX/J from http://www.quickfixj.org/downloads/. At the time of this writing, the latest version is 1.5.3 which is used in this example.
2. Copy `banzai.cfg` to `quickfixj/etc`. You can get `banzai.cfg` from the `SAMPLE-12` folder provided in the code bundle.
3. Get `SimpleStockQuoteService.aar` from the `SAMPLE-12` folder and copy it to `wso2esb-4.8.0/samples/axis2Server/repository/services/`. If there is no `services` directory create one.
4. Start simple Axis2Server from `wso2esb-4.8.0/samples/axis2Server`.
5. Validate whether the `SimpleStockQuoteService` is up and running..`http://localhost:9000/services/SimpleStockQuoteService?wsdl`

6. Open `wso2esb-4.8.0/repository/conf/axis2/axis2.xml` and uncomment the following two sections:

```
<transportReceiver name="fix"
class="org.apache.synapse.transport.fix.FIXTransportListener"/>

<transportSender name="fix"
class="org.apache.synapse.transport.fix.FIXTransportSender"/>
```

7. Copy `fix-synapse.cfg` to `wso2esb-4.8.0/repository/`. You can get `fix-synapse.cfg` from `SAMPLE-12`.
8. Start WSO2 ESB and log in with username as `admin` and password also as `admin`. The ESB will start on `https://localhost:9443`.
9. Get `synapse.xml` from `SAMPLE-12`, copy the content of it, go to **Main** | **Manage** | **Service Bus** | **Source View**, and paste it there, overriding the existing and click on **Update**.
10. The above step will create a proxy service called `BanzaiProxy` in the ESB, which will route messages to the `SimpleStockQuoteService` running in simple Axis2Server.
11. Go to **Main** | **Manage** | **Services** | **List**. You should be able to see `BanzaiProxy` listed there.
12. Start Banzai QuickFIX/J application from `quickfixj/bin`.

```
$ sh banzai.sh ../etc/banzai.cfg
```

13. The previous command will start a Java swing application. Enter appropriate values and click on **submit**. Use IBM, MSFT as the symbol and any integer value as the quantity. Observe the simple Axis2Server console to see the logs being printed.

# How it works...

Let's have a look at the Synapse configuration.

- The `BanzaiProxy` acts as a FIX acceptor and we need to have the following two service parameters.

```
<parameter
name="transport.fix.AcceptorMessageStore">file</parameter>

<parameter
name="transport.fix.AcceptorConfigURL">file:repository/fix-
synapse.cfg</parameter>
```

  - `transport.fix.AcceptorMessageStore`: This is the type of message store to be used with the acceptor. Allowed values for this parameter are file, JDBC, memory, and sleepycat. If not specified, a memory-based message store will be used by default. Additional parameters required to configure each of the

message stores should be specified in the acceptor configuration file.

- `transport.fix.AcceptorConfigURL`: If a service needs to listen to incoming FIX messages from a remote initiator, then WSO2 ESB needs to create an acceptor. This parameter should contain the URL of the file, which contains the FIX configuration for the acceptor.

- `SocketAcceptPort` in `repository/fix-synapse.cfg` is set to 9876. This is the same port number we setup in `quickfixj/etc/banzai.cfg` as the `SocketConnectPort`.
- The WSO2 ESB accepts a FIX payload and converts it into a SOAP message. In the `<payloadFactory/>` mediator we have XPaths defined against the transformed SOAP message ignoring the namespaces.

# Transforming messages from HL7 to SOAP (Advanced)

**Health Level Seven International** (**HL7**) is a data standard, widely used in the health care industry. You can find more about HL7 standard from http://www.hl7.org/implement/standards/index.cfm. Having a common standard will let different health-care systems speak to each other and integrate quite seamlessly. In this section we'll see how to integrate a system that only speaks SOAP with an HL7 system.

## Getting ready

Let's say we have a health-care system called **Medicare-SFO**, which only speaks SOAP. Now Medicare-SFO wants to accept messages from Medicare-NYC, which only knows standard HL7. To facilitate this we will have WSO2 ESB deployed in Medicare-SFO, which will transform HL7 messages coming from Medicare-NYC to SOAP messages that are understood by Medicare-SFO.

The HL7 feature does not ship with WSO2 ESB by default, so we need to start installing it first.

## How to do it...

1. Start WSO2 ESB and log in with username as `admin` and password also as `admin`. The ESB will start on `https://localhost:9443`.
2. Go to **Configure** | **Feature** | **Available Features** | **Add Repository**.
3. Enter **Name** as `wso2packt`, and the **Location (URL)** as http://dist.wso2.org/p2/carbon/releases/turing. Click on **Add.** In case WSO2 ESB 4.8.0 is not yet available, you can use https://svn.wso2.org/repos/wso2/people/prabath/esb/packtbook/p2-repo/ as the **Location (URL)**.
4. Go to **Configure** | **Feature** | **Available Features.** Enter `HL7` in the **Filter by feature name** section, uncheck **Group features by category** and click on **Find Features**.
5. Select **Axis2 Transport HL7** and then click **Install** and follow the wizard.
6. Shutdown the ESB.
7. Open `wso2esb-4.8.0/repository/conf/axis2/axis2.xml` and uncomment the following four sections.

```
<messageFormatter
contentType="application/edi-
hl7"
class="org.wso2.carbon.business.messaging.hl7.message.HL7MessageForm
```

```
<messageBuilder
contentType="application/edi-
hl7"
class="org.wso2.carbon.business.messaging.hl7.message.HL7MessageBuil

<transportReceiver name="hl7"
class="org.wso2.carbon.business.messaging.hl7.transport.HL7Transport

<transportSender name="hl7"
class="org.wso2.carbon.business.messaging.hl7.transport.HL7Transport
```

8. Start WSO2 ESB and log in with the username as `admin` and the password also as `admin`.
9. Get `synapse.xml` from the `SAMPLE-13` folder, copy the content of it and go to **Main** | **Manage** | **Service Bus** | **Source View** and paste it there, overriding the existing and click on **Update**.
10. The previous step will create a proxy service called `MedicareSFOHL7Proxy` in the ESB, which will transform HL7 messages to SOAP and route them to the `MedicareSFOService` running in simple Axis2Server.
11. Get `MedicareSFOService.aar` from `SAMPLE-13` and copy it to `wso2esb-4.8.0/samples/axis2Server/repository/services/`. If there is no `services` directory create one.
12. Start simple Axis2Server from `wso2esb-4.8.0/samples/axis2Server`.
13. Validate whether the `MedicareSFOService` service is up and running (`http://localhost:9000/services/MedicareSFOService?wsdl`).
14. Now let's test our setup with the `HL7` Client

```
$ java -Dhl7_payload=request.hl7 -cp
xercesImpl-
2.8.1.wso2v2.jar:hapi_1.2.0.wso2v1.jar:com.packt.wso2.esb.client.hl7
logging-
1.1.3.jar
com.packt.wso2.esb.client.hl7.HL7Client
```

15. You can get `request.hl7`,`xercesImpl-2.8.1.wso2v2.jar`, `hapi_1.2.0.wso2v1.jar`,`com.packt.wso2.esb.client.hl7.jar`, and `commons-logging-1.1.3.jar` from `SAMPLE-13`.

# How it works...

Let's have a look at the Synapse configuration:

- In the proxy configuration we have enabled HL7 transport, so it can accept HL7 messages.

```
<proxy name="MedicareSFOHL7Proxy"
transports="hl7" startOnLoad="true"
trace="disable">
```

- Add the service parameter `transport.hl7.Port` to the proxy. `HL7TransportListener` will start on this port.

```
<parameter
name="transport.hl7.Port">9293</parameter>
```

- When you save the proxy configuration with the previous parameter you will see the following printed on the ESB console.

```
INFO - HL7TransportListener Started HL7
endpoint on port: 9293
```

- If you pick a different port/hostname, then, while running the client you need to specify them in the two system properties `hl7_port` and `hl7_host`.

```
$ java -Dhl7_port=9293 -Dhl7_host=localhost
-
Dhl7_payload=request.hl7 -cp
xercesImpl-
2.8.1.wso2v2.jar:hapi_1.2.0.wso2v1.jar:com.packt.wso2.esb.client.hl7
logging-
1.1.3.jar
com.packt.wso2.esb.client.hl7.HL7Client
```

- Since we need to send the message out to the business service in SOAP, we need to add the `format` attribute to the corresponding endpoint.

```
<endpoint>
  <address
uri="http://localhost:9000/services/MedicareSFOService"format="soap1
</endpoint>
```

# Enterprise Integration with SAP (Advanced)

**Systems, Applications, and Products** (**SAP**) for data processing is industry leading enterprise software to manage business operations and customer relations.

WSO2 ESB provides the integration layer so an existing SAP R/3 system can be integrated with external services/systems. Further more, it will take care of advanced routing, mediation and work flows between the interconnected systems. WSO2 ESB can also be used to bring quality of services, such as reliability and security to a SAP based system.

Here, we will see how to setup WSO2 ESB in a SAP environment and to install the SAP JCo middleware library, SAP **Intermediate Document** (**IDoc**) and **Business Application Programming Interface** (**BAPI**) adapters.

## Getting ready

Let's take the 7-Eleven convenient store as an example. Say they have a SAP system running internally integrating a POS system, a CRM system and an ERP system. All these systems talk SAP, so integrating them with each other is not an issue. Let's say 7-Eleven acquired Walgreens and now they want to integrate 7-Eleven's internal system with Walgreens' CRM. But, Walgreens' CRM system is not SAP compliant and we have an issue with smooth integration. Let's see how to solve this with WSO2 ESB.

## How to do it...

1. Download `sapidoc3.jar` and `sapjco3.jar` from the SAP support portal https://websmp109.sap-ag.de/connectors and copy those to `wso2esb-4.8.0/repository/components/lib`. You need to have SAP login credentials to access this site.
2. Download the native SAP JCo library and copy it to the system path. For 32-bit Linux you need to copy `libsapjco3.so` to `JAVA_HOME/jre/lib/i386/server`. For 64-bit Linux you need to copy the same library to `JAVA_HOME/jre/lib/amd64`. In Windows you need to have `sapjco3.dll` copied to `WINDOWS_HOME/system32`.
3. Copy the following SAP endpoint property files to `wso2esb-4.8.0/repository/conf/sap`. We have to have two property files at the server-end and the client-end to communicate with an external SAP endpoint with IDoc or BAPI.

- `*.dest`: This is where we keep SAP endpoint parameters when WSO2 ESB has to act as a client to an external SAP endpoint.
- `*.server`: This is where we keep SAP endpoint parameters when WSO2 ESB has to act as a server to an external SAP endpoint.

For this example, we accept SAP messages and then send them across to a SOAP endpoint. So, we only need to have the `.sever` file. Let's name it as `SEVENLELEVEN.server`.

4. Open `wso2esb-4.8.0/repository/conf/axis2/axis2.xml` and uncomment the following two sections to enable the **BAPI/RFC** adapter.

```
<transportSender name="bapi"
class="org.wso2.carbon.transports.sap.SAPTransportSender"/>

<transportReceiver name="bapi"
class="org.wso2.carbon.transports.sap.SAPTransportListener"/>
```

5. Open wso2esb-4.8.0/repository/conf/axis2/axis2.xml and uncomment on the following two sections to enable IDoc adapter.

```
<transportSender name="idoc"
class="org.wso2.carbon.transports.sap.SAPTransportSender"/>

<transportReceiver name="idoc"
class="org.wso2.carbon.transports.sap.SAPTransportListener"/>
```

6. Get `WalgreensService.aar` from the `SAMPLE-14` folder and copy it to `wso2esb-4.8.0/samples/axis2Server/repository/services/`. If there is no `services` directory create one.
7. Start simple Axis2Server from `wso2esb-4.8.0/samples/axis2Server`.
8. Validate whether the `WalgreensService` is up and running, `http://localhost:9000/services/WalgreensService?wsdl`
9. Start WSO2 ESB and log in with username as `admin` and password also as `admin`.
10. Get `synapse.xml` from `SAMPLE-14`, copy the content of it, and go to **Main** | **Manage** | **Service Bus** | **Source View** and paste it there, overriding the existing and **Update**.
11. The above will create a proxy service called `WalgreenSAPProxy` in ESB, which will transform SAP messages to SOAP and route them to the `WalgreensService`, which is running in simple Axis2Server.

# How it works...

Let's have a look at the Synapse configuration.

- In the proxy configuration we have enabled the `idoc` transport so that it can accept SAP IDoc messages.

```
<proxy name="WalgreenSAPProxy"
transports="idoc" startOnLoad="true"
trace="disable">
```

- Add the service parameter `transport.sap.serverName` to the proxy. The value of this parameter should be the name of the file `SEVENLELEVEN.server` at `wso2esb-4.8.0/repository/conf/sap`.

```
<parameter
name="transport.sap.serverName">SEVENLELEVEN</parameter>
```

- Since we need to send the message out to the business service in SOAP, we need to add the format attribute to the corresponding endpoint.

```
<endpoint>
  <address
uri="http://localhost:9000/services/WalgreensService"format="soap12"
</endpoint>
```

- In the `SEVENLELEVEN.server` configuration file you need to define the following properties appropriately:

| **Property** | **Description** |
|---|---|
| `Gwhost` | Host name of the SAP gateway |
| `Gwserv` | Service name of the SAP gateway |
| `Progid` | Program ID of the server, which is optional. The default value is the destination. |
| `Trace` | The default is 0. To enable tracing you need to set the value of this to 1. |
| `Params` | Parameters required by the RFC library |

| Property | Description |
| --- | --- |
| snc_myname | SNC name of the RFC server program |
| snc_qop | Quality of protection. Permitted values are 1,2,3,8,9 and the default is 3. |
| snc_lib | Path and file name of the gssapi library |
| profile_name | Profile file name to be used in the startup. |
| Unicode | Specifies whether to connect in Unicode mode or not |
| max_startup_delay | Maximum server startup delay time in seconds |

# Using Twitter Connector (Intermediate)

Twitter Connector is part of the mediation libraries available with WSO2 ESB. It ships many other Cloud Connectors along with the Twitter Connector from 4.8.0 release onwards. Salesforce, LinkedIn, Paypal, Jira, AWS are a few to name.

## Getting ready

Twitter Connector can be used to Tweet messages looking at the incoming message.

Let's take the same example that we took under the Publish & Subscribe EIP. There the updates are initially published to the `FlightStatusService` proxy service deployed in WSO2 ESB and within the `FlightStatusService` proxy it uses the `<event/>` mediator to publish events to the `DeltaFlightStatusAlertService` and `EmiratesFlightAlertService`.

We'll modify that example here. In addition to events being published to the backend business services, the ESB will also send a Tweet if a flight is delayed.

## How to do it...

1. First we need to create a Twitter App. Go to https://dev.twitter.com/apps and click on **Create New App**.
2. Once the app is created go to https://dev.twitter.com/apps and click on the link to the app that you just created.
3. Go to **Settings** and check **Read and Write** and click on **Update this twitter application's settings** at the bottom of the page. Allow some time getting the new changes updated.
4. Go to **Details** and click on **Create my access token**.
5. Refresh **Details** and copy the values of the following attributes.

    - Consumer key
    - Consumer secret
    - Access token
    - Access token secret

    That's all we need to do on Twitter.

6. Now we need to setup `AAFlightStatusAlertService`, `DeltaFlightStatusAlertService`, and `EmiratesFlightAlertService`. These are the business services that

subscribe to the main `FlightStatusService` with a filter by the airline. So the `AAFlightStatusAlertService` will only get flight status updates for American Airlines and `DeltaFlightStatusAlertService` for Delta flights.

7. Get `AAFlightStatusAlertService.aar`, `DeltaFlightStatusAlertService.aar`. and `EmiratesFlightAlertService.aar` from `SAMPLE-15` and copy those to `wso2esb-4.8.0/samples/axis2Server/repository/services/`. If there is no `services` directory create one.
8. Start simple Axis2Server from `wso2esb-4.8.0/samples/axis2Server`.
9. Validate whether the three business services are up and running by accessing their WSDLs.
10. Start WSO2 ESB and login with the username as `admin` and the password also as `admin`. The ESB will start on `https://localhost:9443`.
11. Go to **Main** | **Manage** | **Topics** | **Add**. Type `flightStatus` as the name of the topic and click on **Add Topic**.
12. Go to **Main** | **Manage** | **Topics** | **Browse**. You will see the `flightStatus` topic that we just created is listed there. Click on it and then on **Add Subtopic**.
13. Create a subtopic with the name `aa` and click on **Add Topic**.
14. Repeat steps 12 and 13 to create two more subtopics with the names `emirates` and `delta`.
15. Go to **Main** | **Manage** | **Topics** | **Browse**. You will see the **aa** subtopic under the **flightStatus** topic that we just created listed there. Click on it and then on **Subscribe**.
16. Select **Topic and Children** for the **Subscription Mode**.
17. Type `http://localhost:9000/services/AAFlightStatusAlertService` as the **Event Sink URL**.
18. Keep the **Expiration Time** blank and click on **Subscribe** to complete the first subscription.
19. Repeat steps 15 to 18 for `DeltaFlightStatusAlertService` and `EmiratesFlightAlertService`. Change steps 15 and 17 appropriately to point to the corresponding subtopic and the corresponding service.
20. Go to **Main** | **Manage** | **Connectors** | **Add** | **Choose File** and select the `twitter-connector.zip` library from `SAMPLE-15` and click on **Upload**
21. Allow some time to update the system and go to **Main** | **Manage** | **Connectors** | **List**. There you will see the `twitter_1.1` library we just added being listed. Click on the **Enable** link against it.
22. Get `synapse.xml` from `SAMPLE-15`, copy the content of it, and go to **Main** | **Manage** | **Service Bus** | **Source View** and paste it there, overriding the existing and click on **Update**.

The above will create a proxy service called `FlightStatusService` in the ESB, which will route messages to the three business services running in simple Axis2Server, as well as Tweet whenever a flight is being delayed.

23. Go to **Main** | **Manage** | **Services** | **List**. You should be able to see **FlightStatusService** listed there.
24. Go to **Main** | **Manage** | **Service Bus** | **Source View** and find the following block in the synapse configuration and update it with the values from step 5.

```
<twitter_1.1.config>
  <consumerSecret>XXXXX</consumerSecret>
  <accessTokenSecret>XXXXX</accessTokenSecret>
  <accessToken>XXXXX</accessToken>
  <consumerKey>XXXXX</consumerKey>
</twitter_1.1.config>
<twitter_1.1.updateStatus>

 <status>{get-
property('airLineDelayed')}</status>
</twitter_1.1.updateStatus>
```

25. Now let's test our setup with cURL.

```
$ curl -d @request-aa.xml -H "Content-Type:
application/soap
+xml action=updateStatus"
http://localhost:8280/services/FlightStatusService

$ curl -d @request-ek.xml -H "Content-Type:
application/soap
+xml action=updateStatus"
http://localhost:8280/services/FlightStatusService

$ curl -d @request-dl.xml -H "Content-Type:
application/soap
+xml action=updateStatus"
http://localhost:8280/services/FlightStatusService
```

You can get `request-aa.xml`, `request-ek.xml`, and `request-dl.xml` from `SAMPLE-15`.

# How it works...

Let's have a look at the Synapse configuration.

- The Twitter mediation library we uploaded in step 20 introduces `twitter_1.1.updateStatus`, which takes care of sending messages

to Twitter. The element `twitter_1.1.config` contains all the configuration parameters required to connect to and authenticate to Twitter.

> **Note**
>
> Twitter uses OAuth 1.0 for authentication and the above parameters are related to OAuth 1.0 authentication.

- Before using the Twitter mediation library we need to import it. When you enable the connector in step 21, the following will be added automatically to the synapse configuration.

```
<import name="twitter_1.1"
package="org.wso2.carbon.connectors"
status="enabled"/>
```

- You need to have `org.wso2.carbon.mediation.registry.WSO2Registry` as the registry provider when we work with the Twitter connector.

```
<registry
provider="org.wso2.carbon.mediation.registry.WSO2Registry">
  <parameter
name="cachableDuration">15000</parameter>
</registry>
```

For the rest of the configuration you can refer to the *Publish & Subscribe recipe, in* Chapter 2, *Enterprise Integration Patterns*.

# Transforming messages from REST/JSON to SOAP and SOAP to REST/JSON (Simple)

Message transformation is one of the key features expected from an ESB in enterprise integration. Here we will see how to transform a REST/JSON message into a SOAP message, and in the return path a SOAP message into a JSON message.

## Getting ready

Let's use the same example we used in explaining the Content-Based Router pattern. There we had the proxy service `CreditCardPaymentService`, which accepts payment-processing requests over HTTP, and delegates further processing to different other SOAP based services depending on the type of the credit card such as VISA, AMEX, or Master.

Here we will introduce a REST API from the WSO2 ESB, which will accept requests in JSON, and then will delegate further processing to backend SOAP based services.

## How to do it...

1. Set up `VISAProcessingService` and `AMEXProcessingService`. We need to have these SOAP based business services up and running, so WSO2 ESB can route messages to them.
2. Get `VISAProcessingService.aar` and `AMEXProcessingService.aar` from `SAMPLE-16` and copy those to `wso2esb-4.8.0/samples/axis2Server/repository/services/`. If there is no `services` directory create one.
3. Start simple Axis2Server from `wso2esb-4.8.0/samples/axis2Server`.
4. Validate whether the two business services are up and running.
5. Start WSO2 ESB and log in with admin/admin. The ESB will start on `https://localhost:9443`.
6. Get `synapse.xml` from `SAMPLE-16`, copy the content of it, and go to **Main** | **Manage** | **Service Bus** | **Source View** and paste it there, overriding the existing and click on **Update**.

   The previous step will create an API called `CreditCardPayment` in the ESB, which will route messages to two other business services running in simple Axis2Server.

7. Go to **Main** | **Manage** | **Service Bus** | **APIs**. You should be able to see `CreditCardPayment` listed there.
8. Now let's test our setup with cURL.

```
$ curl -v -d @request-visa.json  -H
"Content-
type: application/json"
http://localhost:8280/card

$ curl -v -d @request-amex.json  -H
"Content-
type: application/json"
http://localhost:8280/card
```

You can get `request-visa.json` and `request-amex.json` from `SAMPLE-16`.

# How it works...

Let's have a look at the Synapse configuration.

- The following is the API definition.

```
<api name="CreditCardPayment" context="/card">
  <resource methods="POST">
  </resource>
</api>
```

The above block will get hit for any HTTP POST request coming to the context `/card`.

- Inside the `<resource/>` element you can have any number of mediators.
- The configuration inside `<resource/>` element is very much similar to what is in the Content-Based Router pattern example.
- The WSO2 ESB accepts a JSON payload and converts that into a SOAP message. In the `<payloadFactory/>` mediator we are having XPaths defined against the transformed SOAP messages ignoring the namespaces. Since we need to send the message out to the business service in SOAP, we are adding the `format` attribute to the corresponding endpoint.

```
<endpoint>
  <address
uri=http://localhost:9000/services/AMEXProcessingService
format="soap12"/>
</endpoint>
```

- In the `outSequence` we once again use the `<payloadFactory/>` mediator to build an xml payload in the format we need, using the returned SOAP payload from the business service. To convert this XML payload to JSON we need to add the following property just before the `<send/>` mediator.

```
<property name="messageType"
value="application/json" scope="axis2"/>
```

# Chapter 5. Task Scheduling

This chapter explains how to set up scheduled tasks through WSO2 ESB.

# Setting up scheduled tasks (Simple)

WSO2 ESB is capable of scheduling and executing tasks periodically. A task can be scheduled to run *n* number of times in a given *t* time duration. Also you can schedule a task to run just once after WSO2 ESB starts. If you want to get more control in task scheduling then you can use cron-style. With cron-style, you can schedule a task to run everyday at 6 PM or on 25th at 5 PM every month, likewise.

Scheduled tasks can be used within WSO2 ESB for either the internal management purposes or to cater for business needs.

The scheduled task implementation that comes with WSO2 ESB is built on top of Quartz.

## Getting ready

Let's say we have a business service called `ReportGenerationService`. This service needs to run on 20th of every month and generate a report on employees' attendance. To generate the report, this service needs the following employee statistics for the last month.

- Number of employees on full-day leave
- Number of employees on half-day leave
- Number of employees on no-pay leave

Let's see how to do this with WSO2 ESB.

## How to do it...

1. Set up `ReportGenerationService`. We need to have this SOAP-based business service up and running, so WSO2 ESB can route messages to it when required to generate reports.
2. Get `ReportGenerationService.aar` from the `SAMPLE-17` folder and copy it to `wso2esb-4.8.0/samples/axis2Server/repository/services/`. If there is no `services` directory, create one.
3. Start simple Axis2Server from `wso2esb-4.8.0/samples/axis2Server`.

4. Validate whether the business service is up and running by verifying the presence of its WSDL.
5. Start MySQL server; run `sample17.sql` from `SAMPLE-17`. This will create a database with the name `EmployeeHRDB` and a table with some sample data to test our scenario.
6. Copy the MySQL driver file `mysql-connector-java-5.1.26-bin.jar` from `SAMPLE-17` to `wso2esb-4.8.0/repository/components/lib`.
7. Start WSO2 ESB and log in with the username as admin and password also as `admin`. The ESB will start on `https://localhost:9443`.
8. Get `synapse.xml` from `SAMPLE-17`, copy the content of it, go to **Main** | **Manage** | **Service Bus** | **Source View** and paste it there, overriding the existing and click on **Update**.

   The previous step will create a proxy service called `EmployeeHRService` in the ESB, which will talk to the `EmployeeHRDB` to get related employee data and invoke the business service running in simple Axis2Server to generate reports.

9. Go to **Main** | **Manage** | **Services** | **List**. You should be able to see `EmployeeHRService` listed there.
10. In addition to the `EmployeeHRService` proxy service in step 8, our `synapse.xml`, will also create the configuration for the scheduled task. This task is configured to invoke `EmployeeHRService` on 20th every month at 9.00 AM.
11. Go to **Main** | **Manage** | **Service Bus** | **Scheduled Tasks**. You should be able to see `GenerateReportsTask` listed there. If you click on the **Edit** button against that you will see the corresponding configuration.
12. To test the setup we need to wait until 9.00 A.M. 20th. Or else, we can adjust the system clock of the machine where WSO2 ESB is running.

# How it works...

Let's have a look at the Synapse configuration.

- The following is the scheduled task definition.

```
<task name="GenerateReportTask"
class="org.apache.synapse.startup.tasks.MessageInjector"
group="synapse.simple.quartz">
  <trigger cron="0 0 9 20 * ?"/>
  <property xmlns:task=".." name="message">
    <hr:generateReports xmlns:hr=".."/>
  </property>
  <property xmlns:task=".."
  name="injectTo"
  value="proxy"/>
  <property xmlns:task=".."
  name="format"
```

```
        value="soap12"/>
        <property xmlns:task=".."
        name="proxyName"
        value="EmployeeHRService"/>
        <property xmlns:task=".. "
        name="soapAction"
        value="urn:generateReports"/>
      </task>
```

This task will get executed on 20th every month at 9.00 A.M. and invoke the `generateReports` method of the `EmployeeHRService` proxy service. Also note the value we have for the property `proxyName`. This is set to the name of the proxy service, which we want to invoke when the task is triggered. The value of the property `injectTo` should be set to `proxy`.

> **Note**
>
> http://quartz-scheduler.org/documentation/quartz-2.2.x/tutorials/crontrigger explains cron syntax in detail.

A task can be scheduled to run *n* number of times in a given *t* time duration or you can schedule a task to run just once after WSO2 ESB starts.

> **Note**
>
> To run a task only once after the WSO2 ESB gets started we need to have the following configuration:
>
> ```
> <trigger once="true"/>
> ```
>
> To run a task every 30 seconds for 10 times we can use this configuration:
>
> ```
> <trigger interval="30000" count="10"/>
> ```

- Inside the `<dblookup/>` element we execute a query against `EmployeeHRDB` and store the results in multiple `<result />` elements.

```
      <dblookup xmlns="">
        <connection>
          <pool>
            <driver>com.mysql.jdbc.Driver</driver>

       <url>jdbc:mysql://localhost:3306/EmployeeHRDB</url>
```

```
      <user>root</user>
      <password>mysql</password>
    </pool>
  </connection>
  <statement>
    <sql>SELECT * FROM EmployeeHRDB.emp_leave
where id = (Select MAX(id) from
EmployeeHRDB.emp_leave)</sql>
    <result name="full_day_leave"
column="full_day_leave"/>
    <result name="half_day_leave"
column="half_day_leave"/>
    <result name="no_pay_leave"
column="no_pay_leave"/>
  </statement>
</dblookup>
```

To invoke the backend service `ReportGenerationService`, first we have to build the message from the data we retrieved from the database. Here we can use the `<payloadFactory/>` mediator.

```
<payloadFactory media-type="xml">
  <format>
    <hr:generateReport xmlns:hr=".. ">
    <hr:fullDayLeave>$1</hr:fullDayLeave>
    <hr:halfDayLave>$2</hr:halfDayLave>
    <hr:noPayLeave>$3</hr:noPayLeave>
    </hr:generateReport>
  </format>
  <args>
    <arg evaluator="xml"

 expression="get-property('full_day_leave')"/>
    <arg evaluator="xml"

 expression="get-property('half_day_leave')"/>
    <arg evaluator="xml"

 expression="get-property('no_pay_leave')"/>
  </args>
</payloadFactory>
```

Here you can see the multiple `result` values retrieved from the database lookup through the `<dblookup/>` mediator which are stored as properties. These properties are read inside the `<payloadFactory/>` mediator to build the message.

# Appendix A. WSO2 ESB Terminology

This chapter explains briefly the commonly used terminology associated with WSO2 ESB.

## Apache Synapse

Apache Synapse is the mediation engine where WSO2 ESB is built on top of. You can read more about Apache Synapse from http://synapse.apache.org/.

# Mediator

Mediator is the smallest functional unit in WSO2 ESB. A mediator is granular enough to perform a given specific task. WSO2 ESB comes with a rich collection of mediators addressing most of the common integration problems. For example, the **Log** mediator can be used to log any incoming/outgoing messages. The **DBLookup** mediator can be used to retrieve information from a database. The **Header** mediator can be used to add or remove SOAP headers.

# Class mediator

Although WSO2 ESB comes with a rich collection of mediators, it does not limit the user to those. If you want to extend the functionality of WSO2 ESB you can simply do it by writing your own mediator. Using a **Class** mediator is one of the easiest and the mostly used way of extending the ESB's functionality. Class mediators must be written in Java.

# Sequence

A sequence is a logical grouping of a set of mediators. In a way it organizes mediators to form a **Pipes and Filters** pattern. You can read more about pipes and filters pattern from http://www.eaipatterns.com/PipesAndFilters.html.

## Named sequence

A sequence can be either in-line or named. Named sequences can be saved with a name and can be recalled at different places by name. In other words, a named sequence can be reused in different places. Also note that a given sequence can call another sequence during its execution via the **Sequence** mediator.

## Main sequence

The main sequence is a pre-defined named sequence. Any message that is not directed to a proxy service or an API will hit the main sequence. WSO2 ESB comes with a default main sequence, which you can override.

# Proxy Service

A proxy service provides a well-defined SOAP endpoint to the outside. In most of the cases, a proxy service, as its name implies, proxies a real, concrete business service. A proxy service may or may not have a one to one mapping to a business service. It can simply provide a level abstraction over one concrete service or many other business services. In WSO2 ESB, a proxy service is built with an In-Sequence, an Out-Sequence, and a fault sequence.

## In-sequence

A request message coming in to a given proxy service will hit the In-Sequence defined for that proxy service.

## Out-sequence

A response message coming from a concrete or a business service will go through the Out-Sequence defined for the corresponding proxy service.

## Fault-sequence

You can also associate a Fault-Sequence with a proxy service and it will get executed when an exception happens in a proxy operation. This sequence won't get executed for the exceptions thrown from the backend business services. Those will still go through the Out-Sequence.

# End-point

An end-point is a logical abstraction over an external destination where WSO2 ESB has to deliver the message. The end point defined in WSO2 ESB can also take care of quality of service aspects like security and reliability corresponding to the external destination.

## Load-balancing End-point

A load-balancing endpoint is an abstraction over a set of endpoints that you want to distribute the incoming load. By default, WSO2 ESB supports the round-robin load-balancing algorithm, but it does not prevent you from having your own. Having support for load-balancing endpoints, you can also use WSO2 ESB as a load balancer. In fact, WSO2 Elastic Load Balance (ELB) is built on top of WSO2 ESB. You can read more about WSO2 ELB from http://wso2.com/products/elastic-load-balancer/.

## Fail-over End-point

A fail-over endpoint is an abstraction over a set of endpoints where you can define the fail-over behavior. If the primary endpoint fails then the ESB will start sending messages to the next available one. The default fail over behavior is a dynamic fail-over and it will fall back to the primary endpoint as soon as it is available. Whenever the ESB discovers a given endpoint is down, it will mark it as inactive.

# Templates

As we discussed before, a named sequence will enable the reusability of a given configuration. You can have a set of named sequences and use those in different proxy services without redefining. But in this case it will be exactly the same configuration. The templates remove this restriction. With the templates you can reuse a defined configuration and also parameterize it. This was introduced from WSO2 ESB 4.0.0.

# API Element

WSO2 ESB 4.0.3 introduced this as the cleanest way of exposing a REST API. This also laid the foundation to WSO2 API Manager where the API Gateway component of it is built on top of WSO2 ESB. You can read more about WSO2 API Manager from http://wso2.com/products/api-manager/.

# API Handler

The functionality in REST APIs in WSO2 ESB can be extended through custom handlers. One or more handlers can be engaged to a given API to intercept the message flow and add various QoS functionality into the APIs.

# Properties

Properties provide a way of accessing different attributes of the message, passing through the WSO2 ESB. Also you can use properties to change the behavior of the message flow as well as to set different attributes on the message passing through. You can read more about WSO2 ESB properties from here http://docs.wso2.org/wiki/display/ESB460/Properties+Reference.

# Transports

WSO2 ESB supports a variety of transports, which are heavily used in the industry. Those include HTTP, HTTPS, POP, IMAP, SMTP, JMS, AMQP, FIX, TCP, UDP, FTP, FTPS, SFTP, CIFS, MLLP, and SMS. Also you are not just restricted to use those. If you have a need you can write your own transport and configure that in WSO2 ESB.

# Apache Axis2

Apache Axis2 is the SOAP engine behind WSO2 ESB. You can read more about Apache Axis2 from http://axis.apache.org/axis2/java/core/

# Modules/Handlers

Modules/Handlers are the way of extending the functionality of the Apache Axis2 SOAP engine. For example, WS-Security specification is implemented via the Apache Rampart Axis2 handler. Similarly, the WS-ReliableMessaging specification is implemented via the Apache Sandesha Axis2 module. If you need to extend the behavior of the SOAP engine you can simply write your handler and plug that into WSO2 ESB.

# Index

## A



## B



## C

## D



## E

## M



## N

## O



## P



## R



## S

    - integrating, with Apache ActiveMQ Message Broker (intermediate) / WSO2 ESB (publisher) with Apache ActiveMQ Message Broker (Intermediate), How to do it..., How it works...

- WSO2 ESB (subscriber)
    - integrating, with Message Broker (intermediate) / WSO2 ESB (subscriber) with Apache ActiveMQ Message Broker (Intermediate), Getting ready, How to do it..., How it works...

- WSO2 ESB, terminology
    - Apache Synapse / Apache Synapse
    - mediator / Mediator
    - class mediator / Class mediator
    - sequence / Sequence
    - proxy service / Proxy Service
    - end-point / End-point
    - templates / Templates
    - API Element / API Element
    - API Handler / API Handler
    - properties / Properties
    - transports / Transports
    - Apache Axis2 / Apache Axis2
    - Modules/Handlers / Modules/Handlers