Beginning WSO2 ESB

A comprehensive beginner to expert guide for learning WSO2 ESB 5.0

First Edition

Kasun Indrasiri

Apress®

Beginning WSO2 ESB

First Edition

Kasun Indrasiri

Apress®

Beginning WSO2 ESB

Kasun Indrasiri
San Jose, California, USA

ISBN-13 (pbk): 978-1-4842-2342-0 ISBN-13 (electronic): 978-1-4842-2343-7
DOI 10.1007/978-1-4842-2343-7

Library of Congress Control Number: 2016961319

Managing Director: Welmoed Spahr
Lead Editor: Pramila Balan
Technical Reviewer: Isuru Udana
Editorial Board: Steve Anglin, Pramila Balan, Laura Berendson, Aaron Black, Louise Corrigan, Jonathan Gennick, Robert Hutchinson, Celestin Suresh John, Nikhil Karkal, James Markham, Susan McDermott, Matthew Moodie, Natalie Pao, Gwenan Spearing
Coordinating Editor: Prachi Mehta
Copy Editor: Kezia Endsley
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text are available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/. Readers can also access source code at SpringerLink in the Supplementary Material section for each chapter.

Printed on acid-free paper

Contents at a Glance

Contents

About the Author

Kasun Indrasiri is the Director of Integration Technologies at WSO2, a company that produces open source middleware solutions in enterprise integration, API management, security, and IoT domains. He currently provides the Architectural Leadership for the WSO2 integration platform.

Kasun has worked as a Software Architect and a Product Lead of WSO2 ESB with over seven years of experience with WSO2 ESB. He is an elected member of the Apache Software Foundation and a Project Management Committee member and a committer for the Apache Synapse open source ESB project. Kasun has provided Architectural and Technology Consulting for numerous customers in the United States, Europe, and Australia.

He researched high-performance message processing systems and contributed to an ACM publication called "Improved Server Architecture for Highly Efficient Message Mediation." Kasun holds an M.Sc. degree in Software Architecture and a B.Sc. Engineering degree in Computer Science and Engineering from the University of Moratuwa, Sri Lanka.

About the Technical Reviewer

Isuru Udana is a Technical Lead at WSO2 who mainly focuses on enterprise integration. He has more than five years of experience with the WSO2 enterprise service bus product as a Core Developer. Isuru is one of the Product Leads of the WSO2 ESB, and he provides Technical Leadership to the project. He is a committer and a PMC member for the Apache Synapse open source ESB project. Isuru graduated from the Department of Computer Science and Engineering, University of Moratuwa, Sri Lanka. As his final year project, he worked on Siddhi, a high performance complex event processing engine, which now ships with WSO2 CEP server. Isuru is an open source enthusiastic who has participated in the "Google Summer of Code" program as a student as well as a mentor in the last five years.

Acknowledgments

I would like to thank Apress for giving me the opportunity to write this book and, in particular, I must thank Pramila Balan, Acquisitions Editor and Prachi Mehta, Coordinating Editor, who have been constantly guiding and helping me from the very beginning of the writing process. Also I would like to thank Isuru Udana, the Technical Reviewer of this book. His expertise, knowledge, and feedback were quite useful to improve the technical content of this book.

I must thank Dr. Sanjiva Weerawarana who is the Founder, CEO, and Chief Architect of WSO2, for all the guidance he provided throughout all these years at WSO2. Also, I'm grateful to Prabath Siriwardana, who gave me the initial idea for writing a book, and for encouraging me with all his experiences on writing a book.

I'm grateful to my beloved wife Imesha, my parents, and my sister, who are the main driving forces behind all my success.

Last but not least, thank you to everyone who supported me in many different ways.

CHAPTER 1

Introduction to WSO2 ESB

Nowadays successful enterprises rely heavily on the underlying software applications they use. To fulfill diverse business needs, the enterprises have to pick and choose different software application and services, which are built with disparate technologies, use varying standards, and are built by different vendors. When building IT solutions for business scenarios, the enterprises have to make these disparate software applications and services work together to produce unified business functionality.

The task of plumbing different software applications, services, and systems, and forming new software solutions out of that is known as *enterprise integration*. The software application that is designed to enable that task is known as the Enterprise Service Bus (ESB). An ESB enables diverse applications, services, and systems to talk to each other, interact, and transact. It acts as the main messaging backbone in any Service Oriented Architecture (SOA); it's lightweight, built on top of open standards such as web services standards, and supports commonly used patterns in enterprise integration known as Enterprise Integration Patterns (EIP—for more information, see `www.eaipatterns.com`).

What is an ESB?

Let's suppose an organization is running a financial business and has web services, which expose underlying business functionalities such as providing to its customers information on stock quotes (the price of a stock as quoted on a stock exchange) for a given company. They want to expand the business by enabling mobile users to use the online store by making these business functions accessible on mobile devices.

But the mobile devices are inherently based on message formats such as JSON while the backend web service only supports the SOAP message format. The financial organization has to integrate these two systems, which are using disparate message formats, to work together to achieve its business objectives.

To solve this enterprise integration problem, someone could possibly modify either the mobile device application or the backend service to convert one message format to the message format that's understood by the other party. But this approach has several drawbacks. By modifying either the backend web service or the mobile application per the requirements of the other party, the two systems are tightly coupled to the same message format. If the backend service or the mobile application changes its message format, the company is forced to change the code of the other system. Also, if we have to

K. Indrasiri, *Beginning WSO2 ESB*, DOI 10.1007/978-1-4842-2343-7_1

further extend the business use case to include another backend service, then we need to wire all three systems with point-to-point links so that each system is connected to every other system. A change in one of these systems could break the entire business use case.

Therefore, you need a better way to integrate these systems with no modifications at the backend or the client, as well as use a configuration-based approach to integrate these systems without writing any code.

The ESB can be used as the intermediary integration layer between two or more disparate systems and services. Therefore, as illustrated in Figure 1-1, the ESB can be placed between the JSON-based mobile application and the SOAP-based web service. Without writing any code, you can configure the ESB to do the message format conversion. If the business use case needs to be extended further to include another service, you can integrate that service to the ESB, and rather than having point-to-point links, all three systems can be connected through the unified ESB integration layer.

Figure 1-1. *An ESB can be configured to convert messages between the formats recognized by the mobile app (JSON) and the web service (SOAP) it wants to talk to. This is a simple example of enterprise integration.*

Now you have a clear understanding of a concrete enterprise integration use case. Let's explore the enterprise integration space further and see how the ESB is used as the integration backbone.

Modern enterprises need to integrate all the heterogeneous systems (systems using disparate protocols, message formats, and so on) to form various business solutions. The integration between on-premise systems such as web services, file repositories (FTP), proprietary systems such as Enterprise Resource Planning systems (for example, SAP), legacy systems, and data residing in databases and cloud-based solutions such as Software as a Service (SaaS), is the key responsibility of an ESB.

The absence of an integration platform leads the enterprise to require links from a given system to all other systems in the enterprise IT solution. This is known as *point-to-point integration* or *spaghetti integration*.

As depicted in Figure 1-2, the point-to-point integration approach has inherent complexity because the number of systems that participate in the integration scenario increases. If you have to modify or remove one of the systems then that affects the interaction between most of the other systems in your enterprise integration scenario. Therefore, the point-to-point integration approach is extremely difficult to maintain, troubleshoot, and scale.

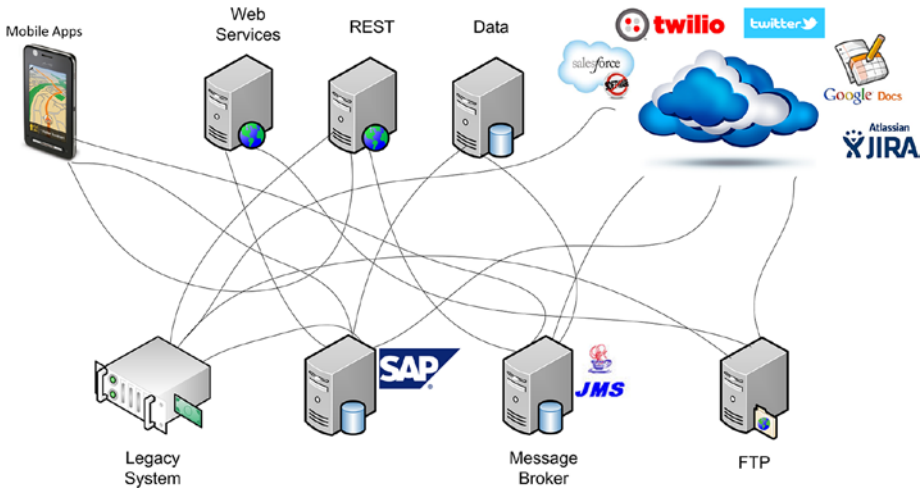

Figure 1-2. *When an enterprise does not use ESB, each system must know how to talk directly to every other system it needs to interact with. This is known as point-to-point integration.*

As depicted in Figure 1-3, ESB can be used as the bus that all the other systems can connect to. An ESB-based approach connects disparate systems using the ESB messaging backbone, and it connects on-premise as well as cloud services. As illustrated in Figure 1-3, ESB eliminates point-to-point integration and integrates all the disparate systems using the bus architecture.

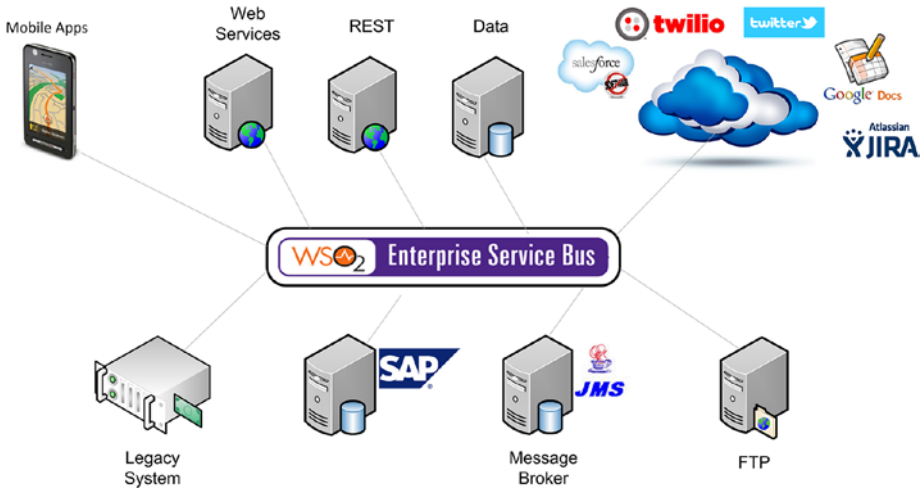

Figure 1-3. *When an enterprise uses ESB, only the ESB needs to know how to talk to each application. The applications themselves do not need to be modified. This is far more efficient than point-to-point integration.*

Based on the previously described integration use case, you can come up with a generic description for ESB. *ESB is an architecture pattern that enables the disparate systems and services to interact through a common communication bus, using lightweight and standard messaging protocols.*

In the next section, you'll discover the core functionalities that are common to any ESB.

Core Functionalities of an ESB

In general, ESB has to offer a wide range of integration capabilities from simple message routing to integrated proprietary systems using complicated integration adaptors. These are the generic functionalities that are common to most ESB products:

- *Message mediation:* Manipulate the message content, direction, destination, and protocols with message flow configurations.
- *Service virtualization:* Wrap existing systems or services with new service interfaces.
- *Protocol conversion:* Bridge different protocols. For example, JMS to HTTP.
- *Support for Enterprise Integration Patterns (EIP):* EIP is the de facto standard for Enterprise Integration (`http://www.eaipatterns.com/`).
- *Quality of service:* Apply security, throttling, and caching.
- *Connecting to legacy and proprietary systems:* Business adapters, including SAP, FIX, and HL7.
- *Connectors to cloud services and APIs:* Salesforce, Twitter, PayPal, and many more.
- *Configuration driven:* Most functionalities are driven by configuration but not code.
- *Extensibility*: There are extension points that can be used to integrate with any custom protocol or proprietary system.

For the most part in the ESB architecture, the ESB is considered a lightweight, stateless integration bus. The architecture itself is mostly based on SOA, but that doesn't mean that you can't integrate non-SOA systems, such as proprietary systems, by using ESB.

The ESB landscape is vast, where there are numerous ESB solutions ranging from open source to proprietary integration solutions. In the following section, you'll discover the key differentiators of WSO2 ESB.

Why WSO2 ESB?

In the ESB vendor space, most of the vendors have rebranded the monolithic and heavyweight enterprise integration solutions as an ESB. But WSO2 ESB is designed and developed from the ground up as the highest performance, lowest footprint, and most interoperable integration middleware. While WSO2 ESB has to improve its graphical

tooling support for designing message flows and graphical data mapping, it offers a broad range of integration capabilities and high-performance message routing support by using an enhanced and optimized message mediation engine, which was inspired by Apache Synapse. In this section, you'll discover key differentiators between WSO2 ESB and other ESB vendors.

Interoperability and EIP Support: Connecting Anything to Anything

WSO2 ESB offers a broad range of integration capabilities from simple message routing to smooth integration of complex proprietary systems. The de facto enterprise integration standards for Enterprise Integration Patterns (EIP) are fully supported in WSO2 ESB. It not only comes with 100% coverage of EIPs, but also with use cases and samples for implementing each and every EIP.

While supporting all the key ESB integration features discussed in the last section, WSO2 ESB offers various integration adapters to proprietary and legacy systems such as SAP. Also, it empowers the on-premise and cloud-based integration scenarios (hybrid integration) with numerous connectors that allow you to smoothly integrate to popular cloud services such as Salesforce, PayPal, and Twitter (see the Connector Store at https://store.wso2.com/store/).

WSO2 ESB offers all these integration capabilities that you can use by configuring the ESB without a single line of code, and in case of any custom requirement, such as supporting proprietary message formats, you can use the numerous extension points to plug in your custom code.

Performance and Stability: The Fastest Open Source ESB

The performance and latency of any ESB solution is a vital factor when it comes to handling large volumes of messages. Based on the regular performance comparisons done by WSO2 on the message routing performance of popular open source ESBs, WSO2 outperforms all the ESB vendors. (The latest ESB performance comparison is available at http://wso2.com/library/articles/2014/02/esb-performance-round-7.5/

The ESB performance comparison given in Figure 1-4 is based on the most recent performance test comparison against WSO2 ESB and other popular ESB vendors. For almost all the integration scenarios, WSO2 ESB outperforms all other ESB competitors.

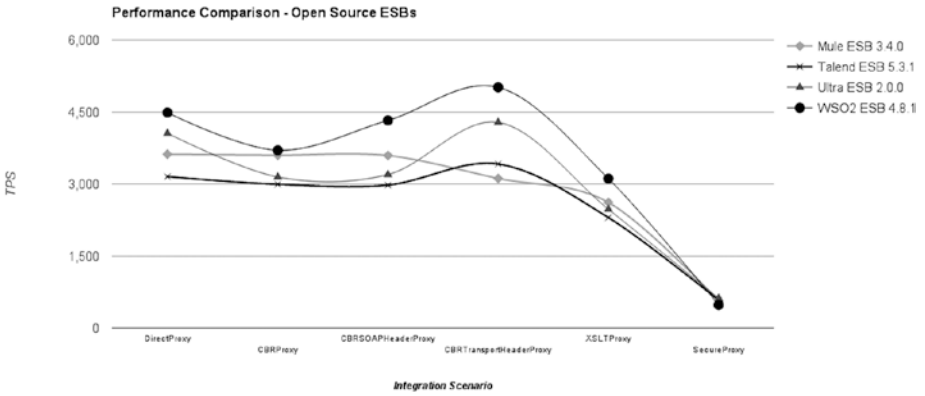

Figure 1-4. *ESB performances comparison between open source ESB vendors for commonly used message routing scenarios*

Stability is also another aspect that goes hand-in-hand with performance. Thousands of production deployments of WSO2 ESB show its stability and its maturity as an ESB solution. eBay uses WSO2 ESB for handling more than several billions of transactions per day in its live traffic (an eBay case study is available at `http://wso2.com/casestudies/ebay-uses-100-open-source-wso2-esb-to-process-more-than-1-billion-transactions-per-day/`).

The Platform Advantage: Part of the WSO2 Middleware Platform

WSO2 ESB is part of the comprehensive WSO2 middleware platform. When you're building real-world enterprise integration solutions, you require the integration capabilities offered by WSO2 ESB, as well as other middleware capabilities such as API management, identity management, data services, analytics, complex even processing, and so on, which are beyond the scope of an ESB. Few ESB vendors who aren't based on a platform concept have tried to have all these features in a monolithic ESB product, but have failed because such solutions cannot address the modern enterprise IT requirements.

A WSO2 middleware platform is built from the ground up with the holistic vision of facilitating all enterprise middleware requirements. The high-level objective of a WSO2 middleware platform is to enable a connected business.

- *All the WSO2 products are built from the ground up and on top of a common foundation:* WSO2 Carbon, a modular, reconfigurable, elastic, OSGi-based architecture, whereas most of the other middleware platforms are primarily built with acquisitions of heterogeneous middleware solutions.
- *Lean and optimized for maximum performance:* Every product in the WSO2 platform is lightweight and designed for achieving the highest performance. For instance, WSO2 ESB is the fastest open source ESB.

- *Largest middleware platform built on a single code base*: All the WSO2 products share the same code base built around a single kernel—WSO2 Carbon. Unlike middleware platforms built with the combination of heterogeneous middleware solutions, WSO2 offers frictionless cross-product integration.
- *100% free and open source under Apache License 2.0 with comprehensive commercial support:* WSO2 has no notion of commercial versus community editions. What you freely download from the `http://wso2.com` web site is the same version used for all the production deployments. The same architects and developers who contributed to the WSO2 platform drive the commercial support for the WSO2 products.
- *Cloud native:* Every WSO2 product inherently supports on-premise, cloud, or hybrid deployments.

At this point you've learned about the key differentiators of WSO2 ESB. In the next section, you'll learn the fundamental concept that's required to start integrating with WSO2 ESB.

How does WSO2 ESB Work?

In this section, you'll learn about the core functional components of WSO2 ESB and the complete end-to-end message flow. Let's design the same financial organization's integration scenario with WSO2 ESB and use that to understand the message flow of WSO2 ESB.

As illustrated in Figure 1-5, the main integration challenges that you have here are to integrate a backend service, which is a SOAP-based web service, with a mobile application that uses JSON. Therefore, the ESB primarily takes care of message format conversion (Message Translator EIP) and exposes a new JSON interface on behalf of the backend web service. The JSON request from the mobile app to the ESB is shown in Listing 1-1.

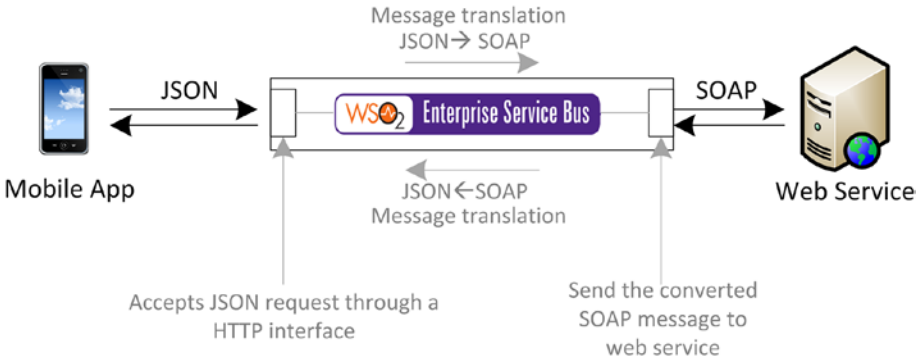

Figure 1-5. *Using WSO2 ESB to integrate a SOAP-based web service and a JSON-based mobile client*

Listing 1-1. JSON Request from Mobile App to ESB

```
{
  "getFinancialQuote": { "company": "WSO2" }
}
```

The backend web service accepts a SOAP message that's shown in Listing 1-2. Therefore, the message processing logic of the ESB needs to convert the request to the SOAP format and convert the SOAP response to back to JSON.

Listing 1-2. Request that Needs to be Sent to the Backend Web Service

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ser="http://services.samples">
  <soapenv:Header/>
  <soapenv:Body>
    <ser:getSimpleQuote>
      <ser:symbol>WSO2</ser:symbol>
    </ser:getSimpleQuote>
  </soapenv:Body>
</soapenv:Envelope>
```

The key steps related to the implementation of this integration scenario can be listed as follows:

- Build a virtual HTTP interface/service that can accept the JSON request from the mobile client and respond with the JSON response.
- In the request-handling path of the virtual HTTP interface/service, it should convert the incoming JSON message to a SOAP message and then send it to the SOAP-based web service backend.
- In the response path, the virtual HTTP interface/service needs to convert the incoming SOAP response to a JSON message and send it back to the client.

Now you'll discover how these steps can be implemented with WSO2 ESB by using its core functional components.

Functional Components

The high-level message flow of WSO2 ESB comprises three main logical components:

- *Message entry points:* Receive client requests.
- *Message processing units:* Contain the mediation logics to process client requests, request that ESB sends to the server, the server response processing logic, and the response that ESB sends back to the client.
- *Message exit points:* Integration points to backend services.

As depicted in Figure 1-6, the mobile clients send a request to the WSO2 ESB via one of its message entry points, then the ESB processes the request messages in the request message processing units, and the message is sent to the backend web service via message exit points. Once the response is received from the backend service, the message goes again through the response message processing units and finally sends the processed response to the client.

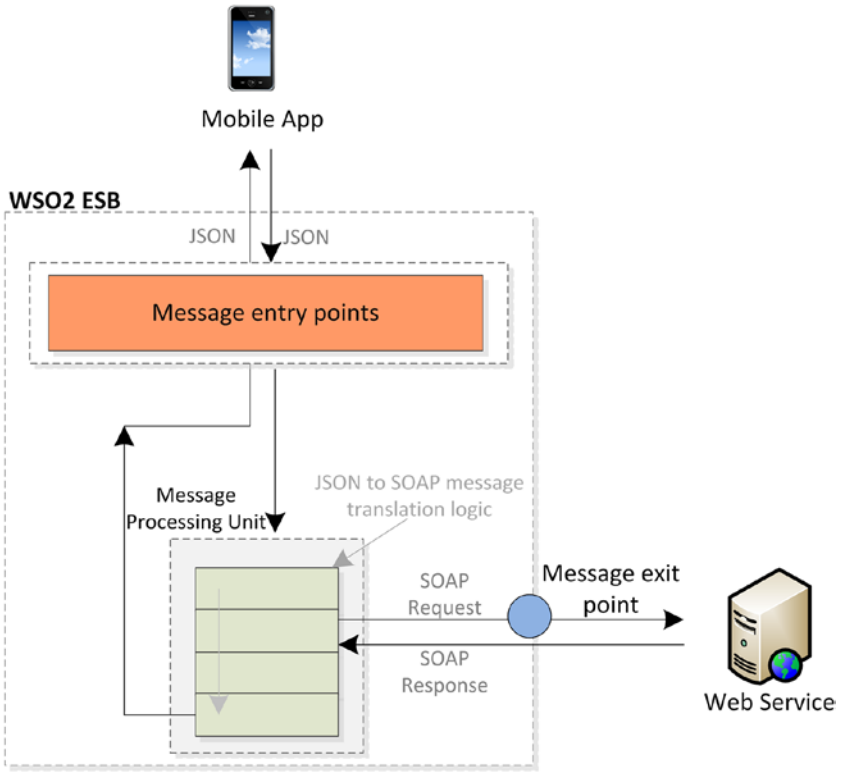

Figure 1-6. *High-level message flow of the integration between mobile app and backend web service*

The task of processing requests or responses in an ESB is known as *message mediation.* As shown in Figure 1-6, the message processing units are doing the required request and response message mediation work.

Message Entry Points: Proxy Service, APIs, and Inbound Endpoints

WSO2 ESB has three main message entry points:

- *Proxy service:* This is a web service interface exposed from the ESB.
- *REST APIs/HTTP:* An HTTP interface anchored at a specific URL context.
- *Inbound endpoints:* A message source with listening or polling capability.

The message entry points are the main components responsible for handling the message transferring from external systems to the ESB. The messages that come through any of these entry points are routed to the message-processing unit, which is responsible for the processing of the message context and message attributes. In the previous example, the messages sent from the mobile app via the HTTP protocol hit the message entry point API and route the message to the respective processing unit (sequence).

Message Processing Unit: Sequences and Mediators

The processing of the message takes place in components known as *sequences*. A given sequence can contain a sequence of components that can process a given message. These components are known as *mediators*. In our example, the logic to translate the message from JSON to SOAP and send out the message takes place in the message-processing unit.

Message Exit Points: Outbound Endpoints

The outbound endpoint (or endpoint) is the message exit point in the WSO2 ESB, which logically represents an external backend service endpoint. In our example, the service address of the backend web service is configured as an outbound endpoint and the message is routed to the outbound endpoint via a *call* or *send* mediator.

Figure 1-7 shows how the message flow is configured to implement the financial organization's integration scenario. The key points related to understanding the message flow in Figure 1-7 are as follows:

- *Expose an HTTP interface to the mobile client:* You should implement an HTTP interface that has to be exposed to the mobile client. That's where we need to configure the message entry points in the WSO2 ESB. Because we have to integrate a mobile client, we can go for the API/HTTP Service message entry point and configure that in the ESB.
- *Anchor an API/HTTP service:* An API/HTTP service is anchored at a URL context, which is designed by the ESB developer and is capable of receiving any request coming on HTTP protocol for that particular context.

- *Configure a mediation sequence:* When you create the API, you need to configure the sequence that will be used to process the request. The sequence contains a set of mediators to process the request.
- The first mediator is a payload factory mediator, which is used to convert the incoming JSON message format to the arbitrary SOAP message format and extract whatever values are needed from the original JSON request.
- You need to set a SOAP action as a header prior to sending the message out from the ESB, because it's required to send an action along with a SOAP 1.1 message. Therefore, a header mediator is used to set the SOAP action.
- Then you need a "call" mediator that can send the message out from the ESB.
- *Configure an outbound endpoint:* When you send out the message, you can configure the destination address of the message or the URL of the backend service using endpoint or outbound endpoint. In addition to specifying the address, you can add various conversion formats when configuring an endpoint, such as SOAP 1.1 and POX (Plain Old XML). Since you want to convert the message to SOAP format, you can use soap11 as the "format" attribute of the endpoint.
- *Configure the response mediation sequence:* Because you need to send a JSON response back to the client, the Property mediator is used as a flag to change the response message format to the JSON message format.
- *Set up a fault mediation sequence:* Any failure scenario can be handled using the fault sequence.

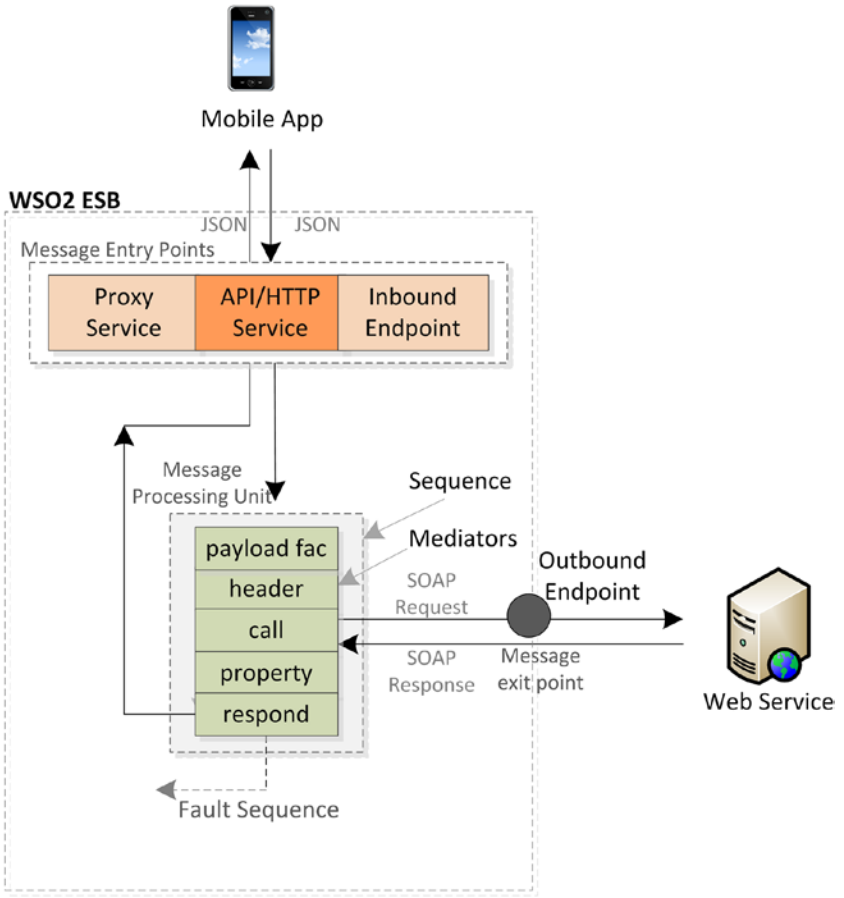

Figure 1-7. *WSO2 ESB message flow for integrating a SOAP-based backend service with a JSON based mobile client*

This completes the end-to-end message flow of our integration scenario. Also, you have an alternative approach to implement the same scenario with a two-sequence model.

As depicted in Figure 1-8, the API/HTTP service can use its built-in, in-sequence, and out-sequence to implement the same integration scenario that was illustrated in Figure 1-8. The only difference here is that the request message always goes to the in-sequence. Hence, the request message processing takes place there. Unlike in earlier approaches, a *send* mediator is used instead of *call* mediator. The main difference is that once you use the send mediator at a given sequence to send out the request, the message flow stops at that point. When you get the response from the backend service, the response flow starts from the out-sequence. Therefore, the out-sequence is the place that you can do the response processing.

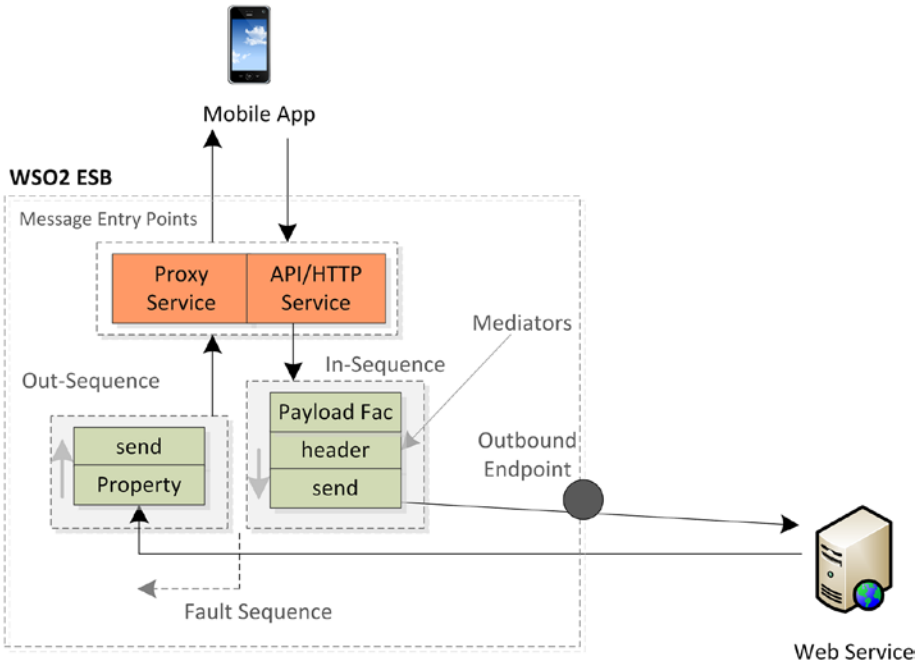

Figure 1-8. *WSO2 ESB message flow for integrating a SOAP-based backend service with a JSON based mobile client using "In" and "Out" sequences*

You can use both of the previously described approaches, but in cases when you're implementing complex service orchestration/chaining scenarios, the *call* mediator-based approach is less complicated.

WSO2 ESB Configuration Language

The entire message flow that you learned during the financial organization's use case is implemented using the XML-based configuration language of WSO2 ESB. Because the WSO2 ESB integration scenario development procedure is completely configuration driven, all the message entry points, message processing units, and message exit points are configured using an XML-based configuration language.

Graphical message-flow editors are available for WSO2 ESB. They're built on top of its configuration language, but throughout this book you'll find all the samples and use cases implemented in raw configuration language.

The following sample configuration represents the implementation of the financial organization's integration scenario that we discussed previously. Here I used an API/HTTP service as the message entry point to the WSO2 ESB and used the single sequence approach with *call* and *respond* mediators. With the following configuration, you can create a new API named `ShoppingInfo` that can be accessed through the HTTP protocol. The complete steps for implementing, deploying, and testing the integration scenarios are covered in detail in Chapter 2.

Listing 1-3. JSON Request from Mobile App to ESB

```
<api xmlns="http://ws.apache.org/ns/synapse"
name="ShoppingInfo"                              <!-- [1] -->
        context="/ShoppingInfo">
   <resource methods="POST">
      <inSequence>                                 <!-- [2] -->
         <payloadFactory media-type="xml">
            <format>
               <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/
               soap/envelope/" xmlns:ser="http://services.samples">
                  <soapenv:Header></soapenv:Header>
                  <soapenv:Body>
                     <ser:getSimpleQuote>
                        <ser:symbol>$1</ser:symbol>
                     </ser:getSimpleQuote>
                  </soapenv:Body>
               </soapenv:Envelope>
            </format>
            <args>
               <arg evaluator="json" expression="$.getFinancialQuote.
               company"></arg>
            </args>
         </payloadFactory>
         <header name="Action" value="urn:getSimpleQuote"></header>
         <call>
            <endpoint>                                    <!-- [3] -->
               <address uri="http://localhost:9000/services/
               SimpleStockQuoteService" format="soap11">
</address>
            </endpoint>
         </call>
         <property name="messageType" value="application/json"
         <!-- [4] -->scope="axis2" type="STRING">
 </property>
         <respond/>            <!-- [5] -->

      </inSequence>
   </resource>
</api>
```

1. Anchoring an HTTP service/API on the context / ShoppingInfo as a message entry point.
2. Request message processing (JSON to SOAP conversion and sending messages out).
3. Configuring backend web service as the message exit point.

4. Response message processing; change the response format to JSON.

5. Sending the response back to the client.

As shown in Listing 1-3, a top-level configuration element known as `api` is the message entry point configuration, followed by the `inSequence`, which is the message processing component. The `inSequence` has the mediators, which are the message processing units. To send out the message, you have a *call* mediator with an *endpoint*. The response processing happens in the mediators that are placed after the *call* mediator (property and respond).

You've discovered about all the building blocks required to design a real-world enterprise integration scenario with WSO2 ESB. You'll learn more about how you can implement and try out the same scenario in WSO2 ESB in Chapter 2.

How to Try the Use Cases in this Book

You can try most of the sample use cases that are discussed throughout this book.

- All the source code related to these use cases can be found at `https://github.com/kasun04/maestro/`.
- The configurations for all the use cases of a given chapter are located at: `https://github.com/kasun04/maestro/tree/master/src/main/resources`.
- All the instructions to run each example are specified under the README file of each chapter. For example if you want to run samples from Chapter 2, the instructions can be found at `https://github.com/kasun04/maestro/blob/master/src/main/resources/ch_02/uc_01/README.txt`

Also, note that the latest configuration is always kept and updated at the GitHub. So, in rare cases, there can be slight mismatches between the code snippet that you find in the book and the sample configuration you'll find in the GitHub repository.

Summary

In this chapter, you learned the fundamentals that are required to build integration solutions with WSO2 ESB. Let's summarize them as follows:

- An Enterprise Service Bus (ESB) enables diverse applications, services, and systems to talk to each other through a common communication bus, using lightweight and standard messaging protocols such as SOAP and JSON. It acts as the main messaging backbone in any Service Oriented Architecture (SOA).

- WSO2 ESB was built on top of a common foundation, called WSO2 Carbon. It's a modular, reconfigurable, elastic, OSGi-based architecture on which all WSO2 products are based. It's the fastest ESB implementation currently available, 100% free, open source under the Apache License 2.0, and supports on-premise, cloud, and hybrid deployments.
- An ESB has three main logical components. Message entry points receive client requests, message processing units contain the logic to process client requests and server responses, and message exit points provide a way to send client requests onto the server and receive their response.
- To configure a message flow through WSO2 ESB between a client and a server, you should configure a message entry point for the client to talk to the ESB, configure a mediation sequence that can process the client request into a message the server understands and sends it out to the server, configure the outbound endpoint for the message to the server, configure another mediation sequence to process and send on the server's response for the client, and set up a fault sequence to deal with failures.

CHAPTER 2

Getting Started with WSO2 ESB

In this chapter, you'll get started with building integration scenarios with WSO2 ESB. The first example provides the foundation for the rest of the chapters, by ensuring your ESB server is set up correctly. We'll also start with the most basic example of system integration by implementing the message transformation use case discussed in Chapter 1.

Designing a Simple Integration Scenario with WSO2 ESB

The best way to get started with WSO2 ESB is to build a simple but real-world integration scenario. The main objective of this use case is to transform messages between two common data formats: JSON and XML.

For our use case, assume that a financial company with the domain `example.com` exists, that hosts SOAP-based web services to expose its business functionalities as services. The `StockQuoteService` financial service is one of the key business functionalities offered from `example.com`, which gives you the stock quote details for a given organization. But the `example.com` financial organization wants to enable this business functionality to its mobile users (who use a JSON as the message format) without modifying the existing backend service and the mobile client.

Suppose that the JSON request format is as follows and the response accepted by the mobile client is the one-to-one transformation of the SOAP response from a backend service to JSON.

```
{
  "getFinancialQuote": { "company": "WSO2" }
}
```


K. Indrasiri, *Beginning WSO2 ESB*, DOI 10.1007/978-1-4842-2343-7_2

The key design steps of the *integration scenario* illustrated in Figure 2-1 can be identified as follows:

- Creating an HTTP interface at the ESB layer on behalf of the existing StockQuote web service.
- Transforming the incoming JSON request to the appropriate SOAP request that needs to be sent to the `SimpleStockQuote` service of the `example.org` financial service and then invoking the service.
- Handling the SOAP response message from the `SimpleStockQuote` service and transforming it back to JSON before sending back the request.

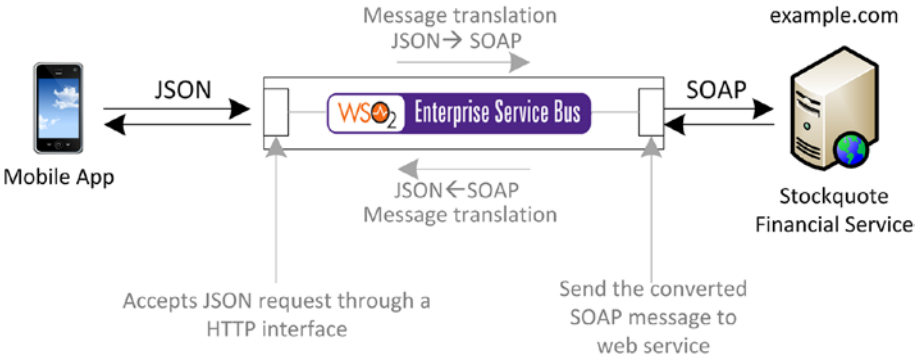

Figure 2-1. *Integrating SOAP-based SimpleStockQuote financial service and a mobile client with WSO2 ESB*

Now let's proceed to the realization of the previous design steps using WSO2 ESB.

Building the Integration Scenario

Since now you're familiar with the sample integration scenario that we are planning to build with WSO2 ESB, in this section I'll walk you through all the steps that are required to build that integration scenario using WSO2 ESB. For our example to be minimally viable, however, we need to do preliminary work to set up the `StockQuote` service.

As illustrated in Figure 2-2, the implementation of this integration scenario requires you to configure a message entry point in WSO2 ESB, configure message-processing components, and finally configure a response-sending logic. The key design steps discussed in the previous section can be mapped into the main implementation steps that you can follow in WSO2 ESB.

1. Creating an HTTP service/API in WSO2 ESB.
2. Creating the request that needs to be sent to the backend service.

3. Sending the request to the backend service.
4. Transforming and sending the response back to the client.
5. Testing and verifying the integration scenario.

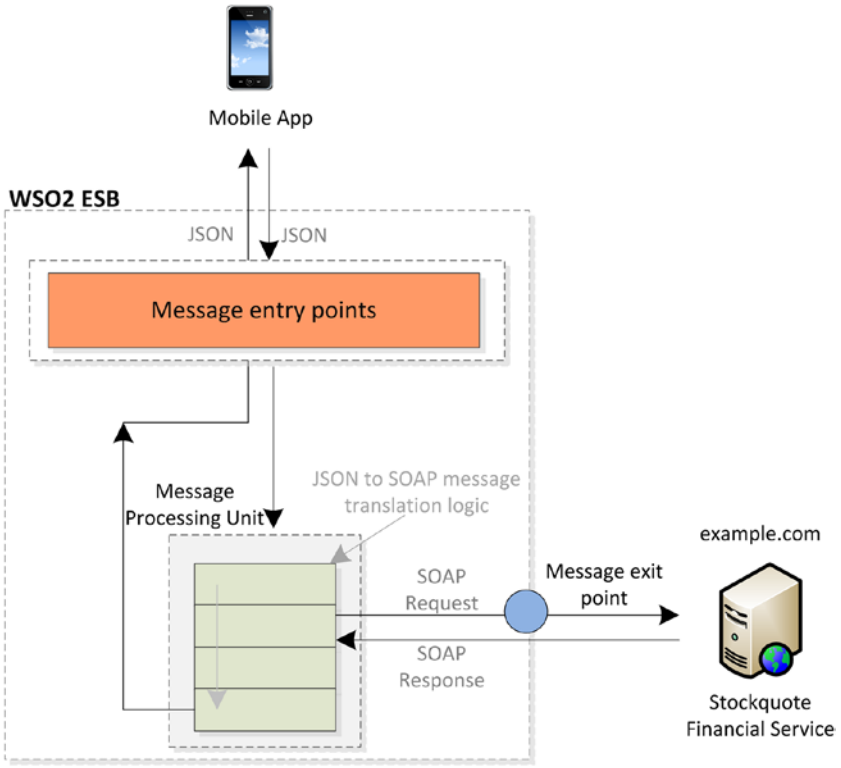

Figure 2-2. *Integrating a SOAP-based StockQuote financial service and a mobile client with WSO2 ESB*

Now let's see how each of the previous steps can be implemented using WSO2 ESB. (In this use case, you'll use the single sequence message flow configuration approach with call mediator and respond mediator.)

Creating a HTTP Service/API in WSO2 ESB

The first step of integrating the financial service and the mobile client is to expose an HTTP interface from the ESB layer, which will be the message entry point to WSO2 ESB. As the message entry point to ESB, you can select HTTP service/API and configure that in WSO2 ESB.

As you know, the development of the integration scenarios in WSO2 ESB is completely configuration driven (using an XML-based configuration language). Therefore, you can start configuring the HTTP service/API in WSO2 ESB by creating a `StockQuoteInfoProvider.xml` file in an arbitrary location in the file system, and later you can copy that file to the WSO2 ESB configuration file location. You can use the configuration provided in Listing 2-1 to configure an HTTP service/API in WSO2 ESB.

Listing 2-1. Configuring the StockQuoteInfoProvider HTTP Service/API as the Message Entry Point

```
<api xmlns="http://ws.apache.org/ns/synapse"
     name="StockQuoteInfoProvider"
     context="/StockQuoteInfo">
   <resource methods="POST">
      <inSequence>
      </inSequence>
   </resource>
</api>
```

The `StockQuoteInfoProvider` API exposes an HTTP interface at the ESB layer that's pinned on the context `/StockQuoteInfo`. You can invoke this HTTP interface from your mobile client by sending an HTTP request to `http://localhost:8280/StockQuoteInfo`. As you can see, the API is configured to respond only to the HTTP POST request coming into this interface. As you learned in Chapter 1, the requests that satisfy the aforementioned criteria are routed to the `inSequence`.

At this point, you have a skeleton of an HTTP interface and deployed in your WSO2 ESB. All the JSON requests that are sent from the mobile client will end up in the `inSequence`.

Creating the Request Sent to the Backend Service

The request message that comes into the HTTP service/API that we created in the previous section can be configured within the `inSequence`. What you get inside the `inSequence` is the JSON request that the mobile client sent. Your objective here is to extract the required values from that JSON message and create a new request that needs to be sent to the backend web service. The key steps are listed here:

1. Creating the SOAP message payload. Inside the `inSequence` of the `StockQuoteInfoProvider` API, you can add the payload factory mediator configuration shown in Listing 1-6. In the payload factory mediator, you can add any arbitrary payload that will replace the current payload of the in-flight message. It allows you to extract any part of the incoming in-flight message and use that when you create the new payload of the incoming message. In our use case, you can use the SOAP payload that's required to invoke the backend web service as the payload of the payload factory mediator. (To find the required message format of a given backend service, you have

to use the WSDL of the backend service and a tool such as SOAP UI to generate the request message format.) Transform the incoming JSON request to the appropriate SOAP request that needs to be sent to the `SimpleStockQuote` service of the `example.org` financial service and then invoke the service.

2. Extracting values from the JSON request. To extract the required values from the original JSON request, the arguments used are `<arg evaluator="json" expression="$.getFinancialQuote.company"/>` with an expression format that allows you to query through a JSON message known as `JSONPath`. This expression extracts the company name from the original request and preserver as the `argument-1` of the payload factory. Later, when you specify the message format for the payload factory, we use that argument as a variable in `<ser:symbol>$1</ser:symbol>`.

3. Configuring the SOAP action. As we send a SOAP request to the backend service, it's required to set the SOAP action as a header in the outgoing HTTP request. The configuration `<header name="Action" value="urn:getSimpleQuote"/>` is there for setting the required SOAP action prior to sending the request to the backend service.

Listing 2-2. Configuring the inSequence to Formulate the SOAP Request to the Backend Web Service

```
<api xmlns="http://ws.apache.org/ns/synapse"
     name="StockQuoteInfoProvider"
     context="/StockQuoteInfo">
    <resource methods="POST">
        <inSequence>
            <payloadFactory media-type="xml">
                <format>
                    <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.
                    org/soap/envelope/"
                                      xmlns:ser="http://services.samples">
                        <soapenv:Header/>
                        <soapenv:Body>
                            <ser:getSimpleQuote>
                                <ser:symbol>$1</ser:symbol>
                            </ser:getSimpleQuote>
                        </soapenv:Body>
                    </soapenv:Envelope>
                </format>
                <args>
                    <arg evaluator="json"
```

```
expression=" $.getFinancialQuote.company"/>
                </args>
            </payloadFactory>
            <header name="Action" value="urn:getSimpleQuote"/>
        </inSequence>
    </resource>
</api>
```

At this point you've successfully transformed the incoming JSON message to the SOAP message format that's required to invoke the backend web service.

Sending the Request to the Backend Service

Now you have to send the request that you created in the `inSequence` to the backend web service. Here are the main steps for sending a request message to the backend service. The related WSO2 ESB configuration is in Listing 1-7.

- To send the request message, you can use the *call* mediator, and to represent the backend web service while sending the request, you can use an endpoint.
- The backend service URI can be configured inside the endpoint configuration. As you invoke the web service backend, you can use the address endpoint and configure the `uri` attribute.
- Inside the call mediator you need to specify the endpoint, which represents the SOAP-based backend service. You can specify the endpoint address as the endpoint `uri`.
- The request message that came through the message entry point (HTTP service/API) isn't a SOAP message (rather it's a RESTful message) and therefore you need to indicate to the mediation engine that you want to do a SOAP 1.1 call to the backend service. This can be configured using the `format="soap11"` attribute of the endpoint configuration.

At the point you use the call mediator, the request is sent to the backend service and the current request message flow stops until we get the response.

Once the response is received, it resumes right after the call mediator with the new in-flight message, which is the response from the backend service, as shown in Listing 2-3.

Listing 2-3. Sending the Request to the Backend Web Service

```
<api xmlns="http://ws.apache.org/ns/synapse"
     name="StockQuoteInfoProvider"
     context="/StockQuoteInfo">
    <resource methods="POST">
        <inSequence>
            <payloadFactory media-type="xml">
                <format>
```

```
                    <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.
                    org/soap/envelope/"
                                      xmlns:ser="http://services.samples">
                        <soapenv:Header/>
                        <soapenv:Body>
                            <ser:getSimpleQuote>
                                <ser:symbol>$1</ser:symbol>
                            </ser:getSimpleQuote>
                        </soapenv:Body>
                    </soapenv:Envelope>
                </format>
                <args>
                    <arg evaluator="json"
expression="$.getFinancialQuote.company"/>
                </args>
            </payloadFactory>
            <header name="Action" value="urn:getSimpleQuote"/>
            <call>
                <endpoint>
                    <address uri="http://localhost:9000/services/
                    SimpleStockQuoteService"
                             format="soap11"/>
                </endpoint>
            </call>
        </inSequence>
    </resource>
</api>
```

Now you've configured the message flow to transform the incoming JSON to the required SOAP message format and sent it to the backend service.

Transforming and Sending the Response Back to the Client

When the backend service response message arrives at the WSO2 ESB, the response message flow starts from the point that we use the call mediator. Therefore, the response message flow configuration goes right after the call mediator. You can follow these steps to configure the response message flow:

- In our integration scenario, it's required to send a JSON response back to the mobile client. Now at the point where it resumes the message flow, you have a SOAP response from the `StockQuote` service as the in-flight message and you need to convert it back to JSON.

- For simplicity, we assumed that the required JSON message format is the one-to-one conversion of the SOAP message format that we get as the response. Therefore you can convert the message from SOAP to JSON by specifying a flag using the property mediator (the actual conversion happens when you send the message out from the ESB).
- Now you can send the response back to the mobile client. For that, you can use a respond mediator in the message flow. The *respond* mediator is capable of sending a given message back to the original client, who sends the request to the ESB.

This completes the entire end-to-end message flow configuration of the integration scenario; the complete configuration is shown in Listing 2-4.

Listing 2-4. Transforming and Sending Responses Back to the Client

```
<api xmlns="http://ws.apache.org/ns/synapse"
     name="StockQuoteInfoProvider"
     context="/StockQuoteInfo">
    <resource methods="POST">
        <inSequence>
            <payloadFactory media-type="xml">
                <format>
                    <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.
                    org/soap/envelope/"
                                      xmlns:ser="http://services.samples">
                        <soapenv:Header/>
                        <soapenv:Body>
                            <ser:getSimpleQuote>
                                <ser:symbol>$1</ser:symbol>
                            </ser:getSimpleQuote>
                        </soapenv:Body>
                    </soapenv:Envelope>
                </format>
                <args>
                    <arg evaluator="json"
expression="$.getFinancialQuote.company"/>
                </args>
            </payloadFactory>
            <header name="Action" value="urn:getQuote"/>
            <call>
                <endpoint>
                    <address uri=" http://localhost:6060/services/
                    StockQuoteService"
                             format="soap11"/>
                </endpoint>
            </call>
```

```
            <property name="messageType"
                      value="application/json"
                      scope="axis2"
                      type="STRING"/>
            <respond/>
        </inSequence>
    </resource>
</api>
```

The complete message flow diagram is illustrated in Figure 2-3. You can try to map the relevant components of that with the configuration element that we followed so far.

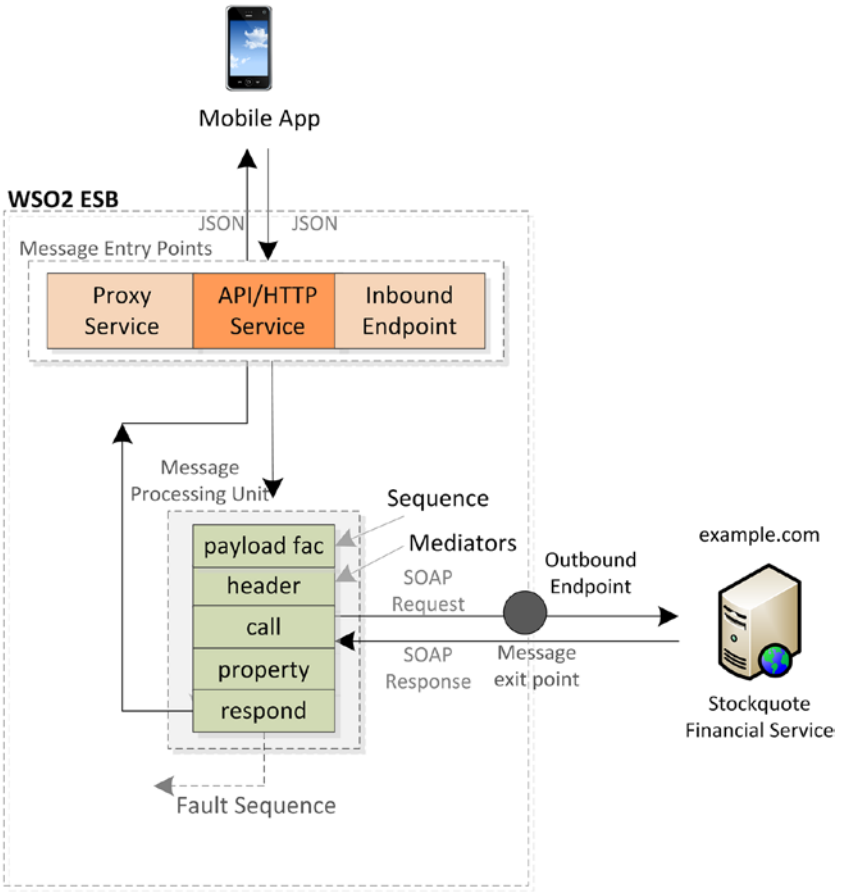

Figure 2-3. *Integrating the SOAP-based StockQuote financial service and a mobile client with WSO2 ESB*

Once you've configured the `StockQuoteInfoProvider.xml`, as shown in Listing 1-8, you can deploy that HTTP service/API in the WSO2 ESB.

Try it Out

Now it's the time to implement, deploy, and run this use case in your WSO2 ESB. For that, you need to follow these steps:

- All the instruction and sample configuration of WSO2 ESB use cases can be found at https://github.com/kasun04/maestro.
- Install and start WSO2 ESB with the instructions provided in https://github.com/kasun04/maestro/blob/master/src/main/resources/SETUP.txt.
- Start backend services as per the instructions provided in the `README.txt` of Chapter 2 (see https://github.com/kasun04/maestro/blob/master/src/main/resources/ch_02/README.txt).
- Copy and deploy the ESB configuration by running `deploy.sh deploy.bat`. (This copies the `StockQuoteInfoProvider.xml` file to the `ESB_HOME/repository/deployment/server/synapse-configs/default/api` directory and then ESB automatically deploys the new configuration.)
- The successful deployment of your HTTP service/API will be shown as an `INFO` log in the console output as follows:

```
INFO - APIDeployer API named 'StockQuoteInfoProvider' has been deployed from
file :<file location>
```

You just deployed your first integration scenario in WSO2 ESB. Now you're all set to test and verify the scenario.

So, you need to invoke the HTTP service that we created inside the WSO2 ESB through a client. The JSON message format that the client would send to ESB is shown in Listing 2-5.

Listing 2-5. The Message Format of the Request from the Mobile Client to ESB: getQuoteMobileClientRequest.json

```
{
  "getFinancialQuote": { "company": "WSO2" }
}
```

Now you need to send this JSON message to the HTTP interface of WSO2 ESB that we developed. You need to send a request with the previous JSON format and use the URI of the HTTP interface/API of WSO2 ESB as http://localhost:8280/StockQuoteInfo. The Content-Type should be `application/json`. You can use any tool such as "Advanced REST client" in Chrome, SOAP UI, POSTMAN-REST Client, and so on and so forth, to

send this request to ESB. Or you can use the `curl` command in the same directory that contains `getQuoteMobileClientRequest.json`.

```
curl -X POST -H 'Content-Type: application/json' -d @
getQuoteMobileClientRequest.json http://localhost:8280/StockQuoteInfo
```

This gives a JSON response that's similar to the following format.

```
{
  "getSimpleQuoteResponse": {
    "return": {
      "@type": "ax21:GetQuoteResponse",
      "change": -2.361885589227761,
      "earnings": 13.698995171522952,
      "high": 152.16537214923102,
      "last": 145.7558347576971,
      "lastTradeTimestamp": "Thu Mar 05 12:18:33 IST 2015",
      "low": 152.63811443571913,
      "marketCap": 56473296.64839228,
      "name": "WSO2 Company",
      "open": -144.60242264890434,
      "peRatio": 25.94446951878229,
      "percentageChange": 1.722243455583964,
      "prevClose": -137.14005308424368,
      "symbol": "WSO2",
      "volume": 7821
    }
  }
}
```

Also, you can verify that the scenario works successfully by observing the console output of the backend service; the following output can be seen in the console.

```
...SimpleStockQuoteService :: Generating quote for : WSO2
```

Congratulations! You've successfully implemented and tested your very first integration scenario with WSO2 ESB.

Summary

In this chapter, you learned how to install and run WSO2 ESB in your development environment. Also, you discovered how a real-world integration scenario could be designed, implemented, and tested with WSO2 ESB. In the course of this chapter, you learned how to apply the fundamentals that you learned in Chapter 1. All the concepts that you learned in this chapter will be discussed in detail in upcoming chapters of the book.

In the next chapter, we will dive deep into the fundamentals of WSO2 ESB, which can be used when implementing various integration scenarios.

CHAPTER 3

Fundamentals of WSO2 ESB

Thus far, you've seen what an ESB is, why using an ESB is a great idea for bringing together disparate systems, and an overview of how WSO2 ESB in particular works. In the general request and response messaging, you saw that the fundamental message flow inside the ESB comprises three basic building blocks, as shown in Figure 3-1.

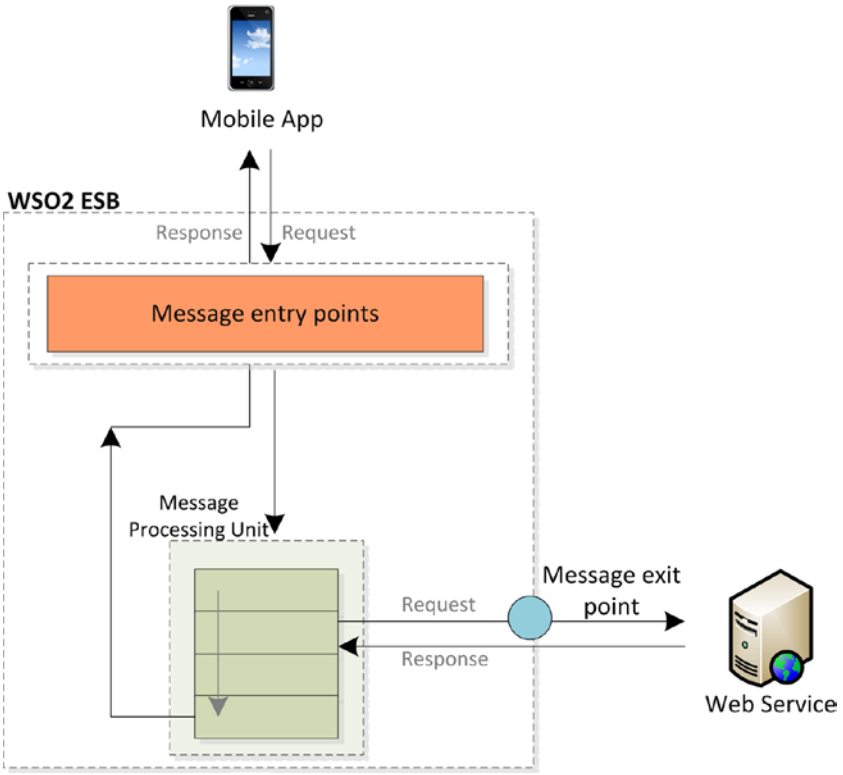

Figure 3-1. *WSO2 ESB has three basic building blocks: message entry points, a message processing unit, and message exit points*

K. Indrasiri, *Beginning WSO2 ESB*, DOI 10.1007/978-1-4842-2343-7_3

In this chapter, you'll discover basic message flow constructs in WSO2 ESB, when and how to use those constructs to build integration solutions. The goal of this chapter is to lay the foundation for the rest of the book. That said, let's start with a request/response messaging scenario where a message is sent to the ESB from an external client/message source, ESB routes the message to a backend service and sends the response back to the client.

Message Entry Points

A message entry point in WSO2 ESB is responsible for listening for inbound messages or polling a message source for the inbound messages, and then injecting such messages into the ESB's message processing units. Recall in Chapter 1 that the main message entry points of WSO2 ESB, illustrated in Figure 3-2, are proxy services, API/HTTP services, and inbound endpoints.

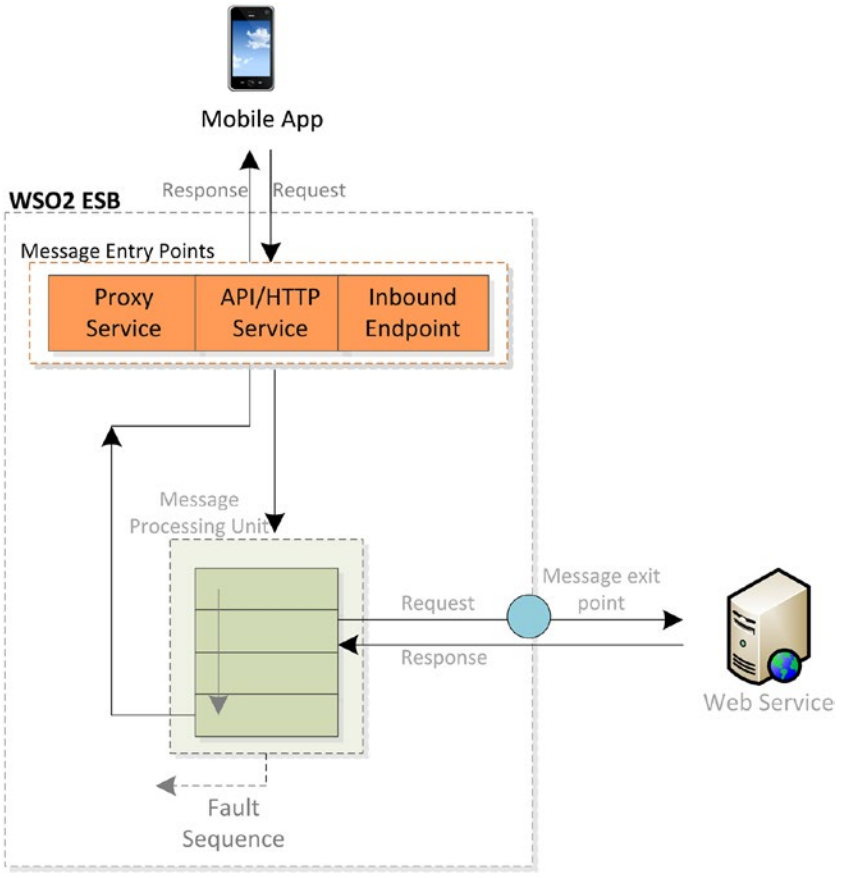

Figure 3-2. *Message entry points of WSO2 ESB*

When it comes to the development of integration scenarios, based on your requirements, you need to choose different message entry points and configure them in WSO2 ESB. You can select any of the following message entry points that match your integration criteria.

- *Proxy service:* When you want to expose a web service interface for an existing backend system (primarily SOAP over HTTP/S).
- *API/HTTP service:* Expose an HTTP interface that's fully compliant with REST (Representational State Transfer).
- *Inbound endpoint:* Fully dynamic inbound messaging source with polling or listening capability.

In this section you'll learn when and how to use these message entry points in WSO2 ESB.

Using Proxy Services

Let's suppose that a banking software solution has a legacy financial system that uses XML (or *Plain Old XML* – PoX) as the message format over HTTP. The banking software solution needs to expose this system as a web service, so that it supports web services standards such as SOAP, `WS-Addressing`, WSDL, and so on. You can use WSO2 ESB's *Proxy Services* message entry point to implement this integration scenario, as shown in Figure 3-3.

Figure 3-3. *Proxy service exposes a web service interface (with WSDL) by wrapping the legacy system. The SOAP to XML (or Plain Old XML) message conversion takes place in the message processing units (sequences and mediators).*

For the existing legacy system, you can create a proxy service in ESB that exposes a web services interface to its web services client. The message protocol translation takes place inside the message processing logic. In this scenario, the service exposed from the ESB is hosted at `http://[hostname]:8280/services/OrderProcessingWS`. You can also create your own web service definition (WSDL) if required and expose it along with the service (`http://[hostname]:8280/services/OrderProcessingWS?wsdl`).

Therefore, a given *proxy service* doesn't contain a real service implementation but exposes a virtual web service interface (with the support for SOAP, `WS-Addressing`, AND WSDL) on top of an existing non-web service-based system or service. Figure 3-4 illustrates the structure and the message flow of a proxy service invocation.

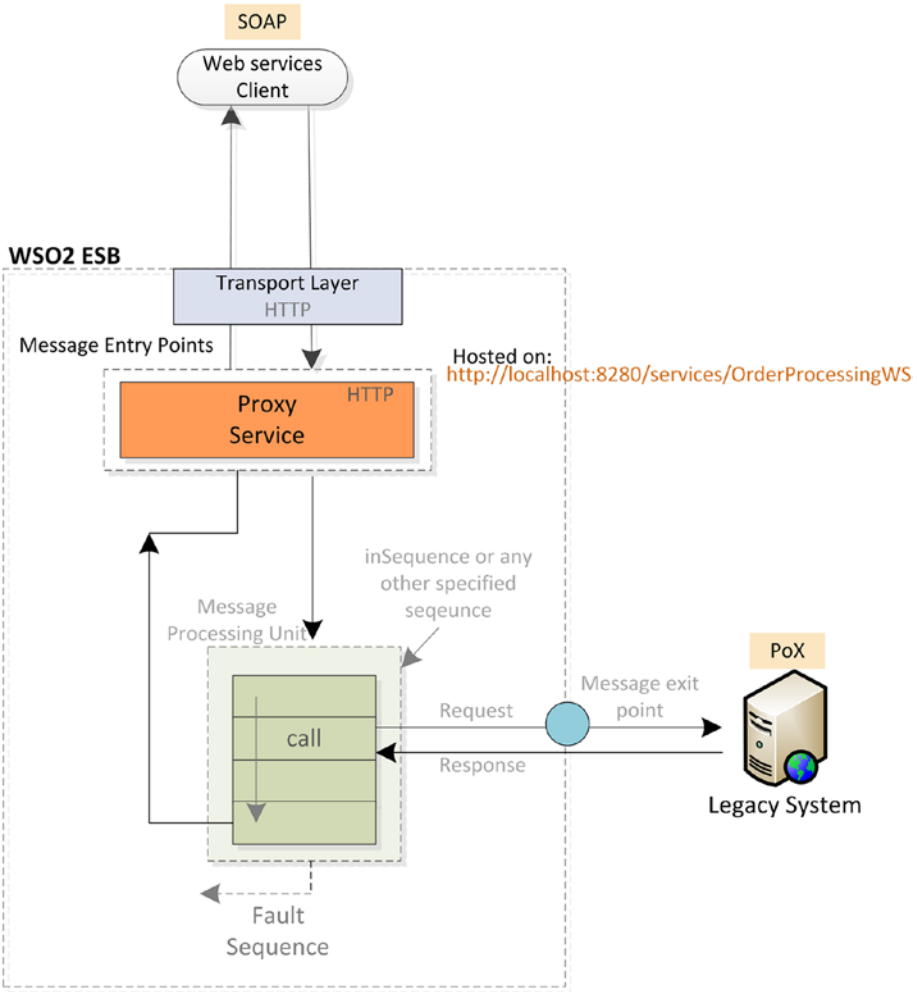

Figure 3-4. *Proxy service exposes a web service interface by wrapping the legacy system and using the HTTP transport layer. The SOAP to XML (or Plain Old XML) message conversion takes place in the message processing units (inSequence/mediators). In the case of an error, the specified fault sequence will get invoked and you can do the fault handling there.*

The message flow of the banking integration scenario can be summarized as follows:

- The proxy service is deployed and hosted on
- `http://[hostname]:8280/services/OrderProcessingWS`.
- Web service client sends a SOAP request to WSO2 ESB via HTTP.
- The proxy service can specify the protocol it uses to receive messages and in this case it's HTTP. For the HTTP protocol, WSO2 ESB has a global transport receiver (receives messages on that protocol) that's configured in `$ESB_HOME/repository/conf/axis2/axis2.xml`.
- Therefore, the request message comes through the transport layer and then dispatches to the respective proxy service.
- At the proxy level various web service specific validations take place, such as WSDL and `WS-Addressing` validations. Once these are successful, the message is dispatched to the in-sequence of that proxy service.
- You can configure the message processing logic inside the in-sequence of the proxy service, and as discussed in Chapter 1, you can either follow the single sequence approach or use the in-and-out sequence approach.

This completes the end-to-end message flow of a proxy service invocation scenario. Let's take a closer look at the implementation of such scenarios in WSO2 ESB and the respective configuration language.

Proxy service can be exposed in multiple protocols such as HTTP, JMS, VFS (file), and so on. But with the introduction of inbound endpoints, the proxy services are primarily used for exposing web service interfaces on top of the HTTP protocol. (Initially, proxy services are designed to be used to receive message from other protocols than HTTP, but with the introduction of inbound endpoints, proxy services are more or less used for HTTP. However for some protocols that are not still available as inbound endpoints, you may use proxy services with a specific transfer protocol.) For other types of inbound messaging scenarios, inbound endpoints are more suitable. The configuration of a given proxy service resides in `$ESB_HOME/repository/deployment/server/synapse-configs/default/proxy-services/{ProxyServiceName}.xml`.

In WSO2 ESB, to deploy a given proxy service, the proxy service configuration should be placed in

```
$ESB_HOME/repository/deployment/server/synapse-configs/default/proxy-services/OrderProcessingWS.xml.
```

Then ESB deploys a virtual web service on ESB that's anchored at `http://[hostname]:8280/services/OrderProcessingWS`. Once you copy your proxy service configuration, it's deployed in a running ESB instance (no restarting required). The configuration for the banking integration scenario is shown in Listing 3-1.

Listing 3-1. Proxy Service to Wrap a Legacy System from a Web Service Interface

```
<proxy xmlns="http://ws.apache.org/ns/synapse"
       name="OrderProcessingWS"                                    <!-- [1] -->
       transports="http"
       startOnLoad="true"
       trace="disable">
   <description/>
   <publishWSDL key="gov:/OrderProcessingWS.wsdl" />   <!-- [2] -->
   <target>
      <inSequence>                                              <!-- [3] -->
         <log level="full"/>                                    <!-- [4] -->
          <call>
              <endpoint
                <address
uri="http://localhost:9000/services/SimpleStockQuoteService"
                            format="pox"/>
              </endpoint>
          </call>
                < log level="full"/>                    <!-- [5] -->
                 <respond/>
      </inSequence>
   </target>
</proxy>
[1] Proxy service name exposed as a part of the URL of the proxy service.
Proxy service is exposed on HTTP transport so request can be sent only via
HTTP transport.
[2] You can specify the service contract (WSDL) that should be exposed to
the external client.
[3] Using in-sequence as the message-processing unit.
[4] Request message-processing logic.
[5] Response message-processing logic.
```

As shown in Listing 3-1, the proxy service is configured on `http://localhost:8280/services/OrderProcessingWS` and exposed on an HTTP protocol. The message from the web service client goes through the transport layer (which is responsible for handling messages that comes with different messaging protocols such as HTTP) and injects the message into the respective proxy service. The HTTP transport receiver (configured in `$ESB_HOME/repository/conf/axis2/axis2.xml`) accepts the messages and then dispatches it to the relevant proxy service based on the URL context (`/services/OrderProcessingWS`). The transport layer, comprised of a transport receiver and transport sender, receives and sends messages out of a proxy service. The HTTP transport configuration will be shared among all the proxy services and the transport receiver/sender configuration cannot be updated dynamically.

As discussed previously, the main use case for a proxy service is to expose a virtual web service interface over HTTP protocol; therefore, for enabling web service-related capabilities such as `WS-Security` and `WS-Addressing`, you can use built-in parameters such as `enableSecurity`, `enableAddressing`, and so on. You'll learn more about these capabilities in Chapter 5, which discusses web service-based integration.

Using APIs/HTTP Services

API or HTTP services allow you to create HTTP interfaces in WSO2 ESB and process messages based on REST (Representational State Transfer) style. Let's examine a sample use case to understand the concept of an API/HTTP service.

Pizzashop is a pizza outlet that allows the users to order pizza online. Initially they used a web service that allows the customers to browse and buy pizzas, but now the organization wants to expand its business functionalities for its mobile users. Therefore, they need to expose this functionality as REST interfaces over HTTP and use JSON as the message format. The key business requirements they have are listing the available pizza menu and allowing the customers to buy a pizza with a given pizza ID, as shown in Figure 3-5.

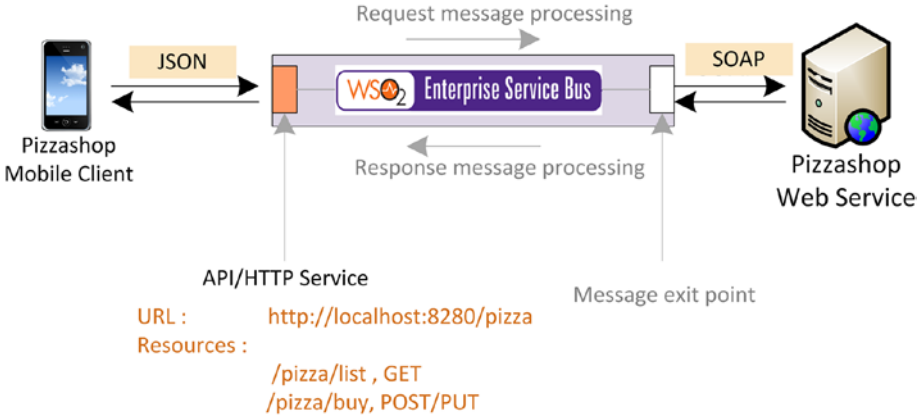

Figure 3-5. *Pizzashop exposes an API/HTTP service on top of the existing SOAP web service. The API/HTTP service is anchored at* `http://{hostname}:8280/pizza` *and two resources are defined for buying and listing business functionalities. The message translation takes place in the message-processing unit.*

As per the business requirements, we can list the main steps involved in designing integration scenario shown in Figure 3-5.

- The first step is to decide the URL context of the pizzashop API/HTTP service that you'll expose. For example, you can use the URL `http://hostname:8280/pizza`, where all the client requests are sent to the URL starting with URL context pizza. Therefore the API/HTTP service you create is anchored at a user-defined URL context.
- Then you can create two resources (`/list` and `/buy`) for the different functionalities of the pizzashop API. For that, you can use resources in an API definition and define an URL mapping or URI template for each resource.

- In the pizzashop example, two resources are defined with URL mapping—`/list` and `/buy/{pizzaId}`. This is to restrict the type of HTTP requests processed by a particular resource. This means the resource with the URL mapping `/list` can only process the HTTP request with `http://hostname:8280/pizza/list` and the other resource can only process the HTTP request with `http://hostname:8280/pizza/buy/<pizzaId>`.
- Also, you can restrict the type of HTTP request that a given resource can process by specifying the HTTP method/verb. For example, in the pizzashop API, the resource with `/list` can only process HTTP GET requests.
- Then each resource of pizza RESTAPI has an in-sequence and an out-sequence. This is exactly the same message processing logic that you learned in proxy services. The request message processing of a given resource can use the single sequence model with call-and-respond mediator, or you can use the in- and out-sequence approaches.

The message flow of the invocation of the API in the pizzashop integration scenario is depicted in Figure 3-6 and is similar to that of a proxy service. But the main difference is that APIs there can be multiple resources with their own message processing units. The message dispatching to the matching resource is done using the URL mapping parameter.

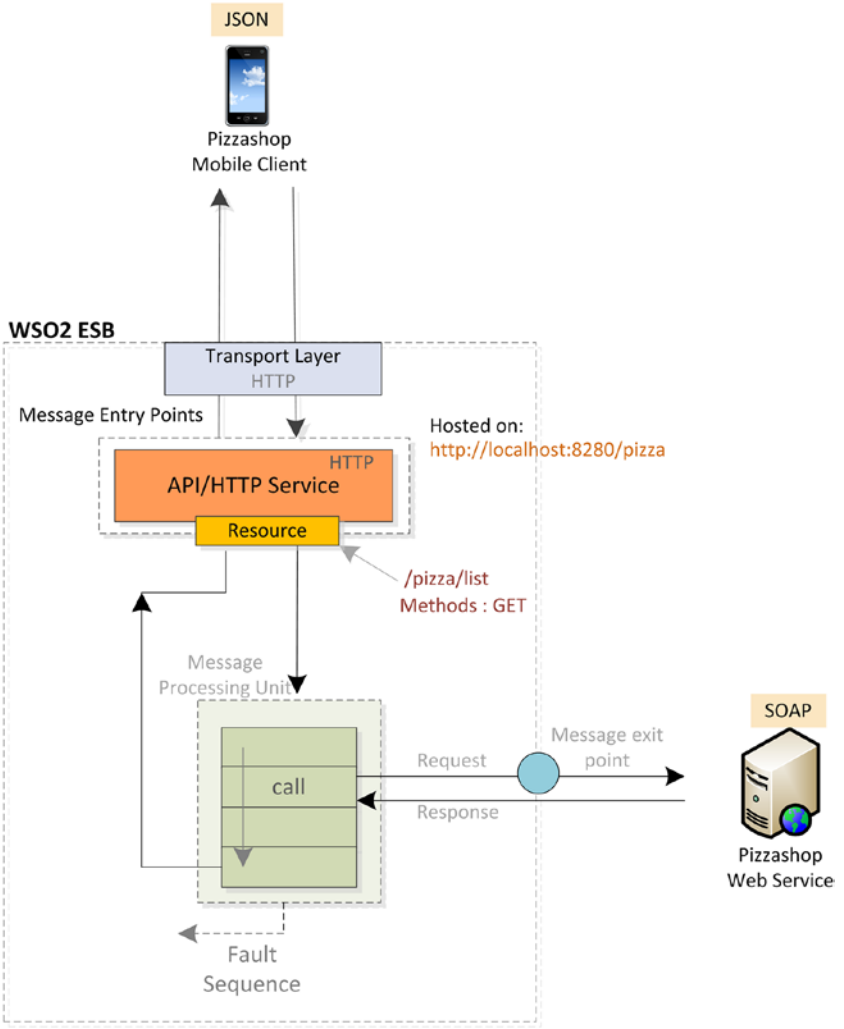

Figure 3-6. *Pizzashop exposes an API/HTTP service on top of the existing SOAP web service. The API/HTTP service is anchored at* `http://{hostname}:8280/pizza` *and two resources are defined for buying and listing business functionalities. The message translation takes place in the message-processing unit.*

The related configuration of the pizzashop scenario is shown in Listing 3-2.

Listing 3-2. Exposing a HTTP Service/API for the Pizzashop Scenario

```
<api name="PizzaRESTAPI" context="/pizza">
 <resource url-mapping="/list" methods="GET" inSequence="seq3" outSequence=
"seq4"/>
```

```
<resource uri-template="/buy/{pizzaId}" methods="PUT POST" inSequence="seq5"
outSequence="seq6"/>
</api>
```

The primary use of an API/HTTP service is to expose a REST interface from the WSO2 ESB. You can use an API/HTTP service to wrap an existing web service backend with a simple REST interface. As with proxy services, messages are dispatched to APIs via the transport layer, which is common to all proxy services and APIs.

API/HTTP services can be exposed in HTTP/S protocol only. The configuration of a given API resides in $ESB_HOME/repository/deployment/server/synapse-configs/default/api/{ApiName}.xml.

The main difference between the proxy service and API is that APIs are purely for HTTP and REST interfaces have various message formats such as JSON and XML. The proxy services are primarily for web service interfaces over HTTP with SOAP message formats and other web service related capabilities, such as `WS-Addressing`, `WS-Security`, and so on. You'll learn about API/HTTP services in detail in Chapter 5.

Using Inbound Endpoints

The message entry points (proxy services and APIs) that we discussed so far use underlying transport receivers as the inbound messaging channels. The transport layer configuration is static and globally applied for all the proxy services and APIs that are exposed from a given protocol. For instance, the HTTP port for a given proxy or API is 8280 and you can't configure the port dynamically for a specific proxy service or an API (static configuration is required in `axis2.xml` and only actives after an ESB restart).

But integration use cases exist where you need to configure inbound message channels dynamically. For instance, you may want to get a message from a JMS (Java Message Service) message queue and inject it into the ESB message flow by dynamically configuring the queue name and JMS connection parameters. Or you can configure an API to listen to an HTTP port that you can configure dynamically. Moreover, you may want to have integration scenarios deployed in completely isolated runtimes (this is known as multitenancy support in WSO2 ESB).

With the integration requirements listed previously, you have a need for a dedicated ESB construct to configure inbound channels dynamically. The inbound endpoints are designed to serve that purpose. Before moving into the details of inbound endpoints, let's consider a simple integration scenario where you need to configure WSO2 ESB to read messages from a JMS queue and later send them to a backend web service.

Let's suppose that you need to implement an order processing system in a retail store. The orders are placed in a JMS queue of a message broker by an external system/client. You need to get those orders periodically, perform message transformations, and send them to a web service or any other backend service. All the logic related to getting messages from the JMS queue must configured dynamically. The respective use case is illustrated in Figure 3-7.

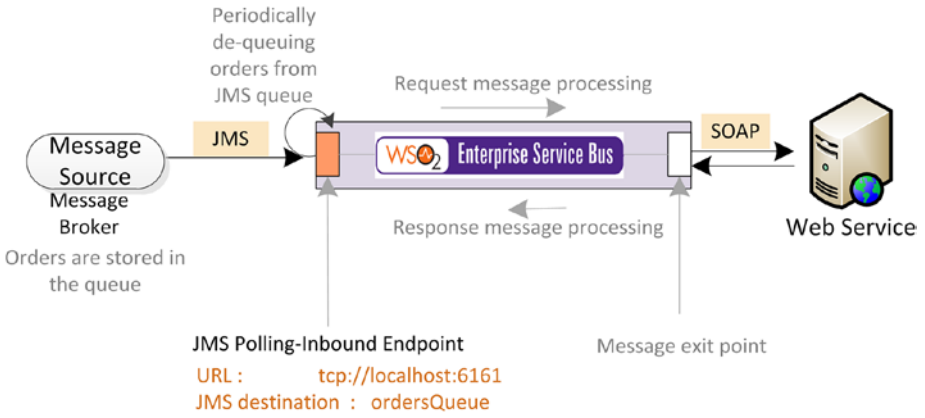

Figure 3-7. *The orders are placed in a JMS queue by the external systems, and ESB has to process them by de-queuing the message and sending it to the backend web service. The JMS inbound endpoint can poll for the messages in the queue and inject them into the message-processing unit (sequences and mediators).*

You can implement the previously described integration scenario with ESB with the use of JMS inbound endpoint.

- The first step is to configure an inbound endpoint with JMS protocol. (Because this is about integrating to an external message broker, it's mandatory to have all the required client libraries copied into `$ESB_HOME/repository/component/lib`.)
- As shown in Listing 3-3, you can specify the protocol as `jms` and add all the required parameters such as JMS destination, connection factory, connection URL, and so on.
- Once the message is fetched from the queue, you can inject the message into a sequence (`orderProSeq` sequence) and do all the generic message mediation (transform, enrich, and send to a backend service) in that sequence.

Listing 3-3 illustrates the definition of an inbound endpoint and the respective parameters that you need to configure for this JMS use case.

Listing 3-3. Using a JMS Inbound Endpoint to Process Messages in a JMS Queue

```
<inboundEndpoint
    xmlns="http://ws.apache.org/ns/synapse"
    name="OrderProcesseingInboundEP"            <!-- [1] -->
        sequence="orderProcSeq" onError="fault" <!-- [2] -->
    protocol="jms"                              <!-- [3] -->
         suspend="false">
      <parameters>                              <!-- [4] -->
```

```
        <parameter name="interval">1000</parameter>
        <parameter name="transport.jms.Destination">ordersQueue</parameter>
        <parameter name="transport.jms.CacheLevel">1</parameter>
        <parameter name="transport.jms.ConnectionFactoryJNDIName">
              QueueConnectionFactory</parameter>
        <parameter name="sequential">true</parameter>
        <parameter name="java.naming.factory.initial">
          org.apache.activemq.jndi.ActiveMQInitialContextFactory
        </parameter>
        <parameter name="java.naming.provider.url">tcp://localhost:61616
        </parameter>
        <parameter name="transport.jms.SessionAcknowledgement">AUTO_
        ACKNOWLEDGE</parameter>
        <parameter name="transport.jms.SessionTransacted">false</parameter>
        <parameter name="transport.jms.ConnectionFactoryType">queue
        </parameter>
      </parameters>
</inboundEndpoint>
[1] Name of the inbound endpoint.
[2] Message processing sequence that the inbound endpoint injects the
message to.
[3] Protocol of the inbound endpoint.
[4] Parameters that are required to poll the message from the message source
through the specified protocol.
```

Now that you've seen inbound endpoints in action, let's move on to the discussion of the inbound endpoints fundamentals.

An inbound endpoint allows you to dynamically integrate external message sources with different protocols with ESB. By configuring inbound endpoint parameters, you can configure inbound channels dynamically. The inbound endpoints are deployed in the `$ESB_HOME/repository/deployment/server/synapse-configs/default/inbound-endpoints` directory. The inbound endpoints can be categorized into two types based on their behavior: polling and listening inbound endpoints.

Polling Inbound Endpoints

A polling inbound endpoint is capable of polling a given message source that's specified with the *protocol* (such as JMS or File) and with the polling *interval*. It can also inject the message into a specified sequence. The JMS inbound endpoint shown in Listing 3-3 is an example of a polling inbound endpoint. The polling job is executed by the WSO2 ESB's internal tasks implementation, and it supports various coordination mechanisms such as one executor at a given time and multiple executors. Figure 3-8 shows the message execution flow inside a polling inbound endpoint.

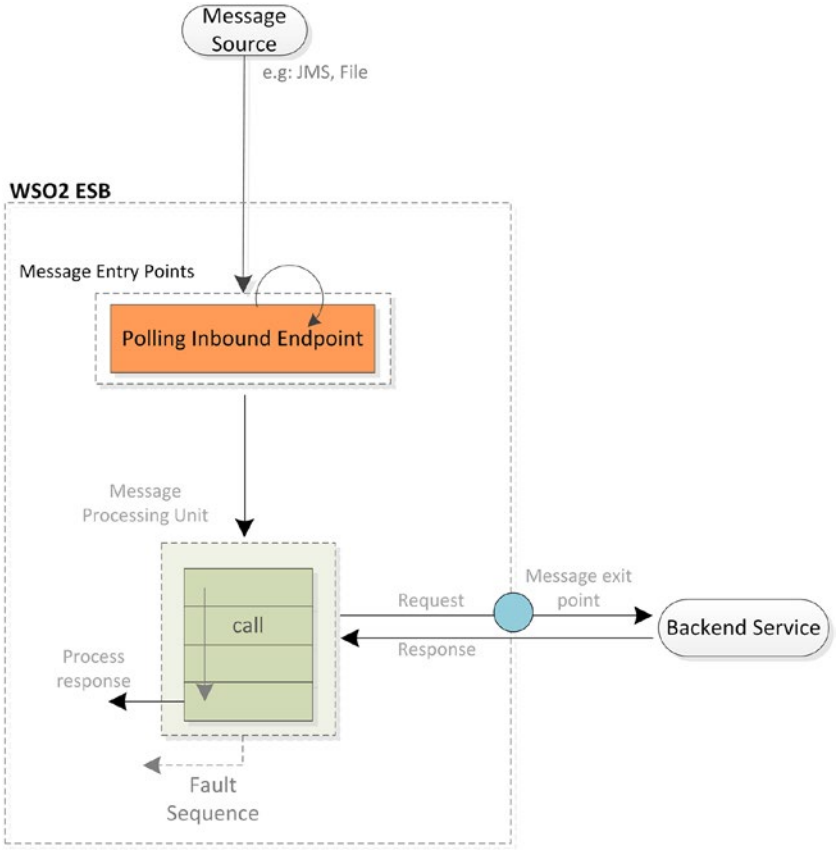

Figure 3-8. *The polling inbound endpoint periodically checks for new messages in the message source, and if there are messages available, they're injected into the message processing unit (sequence). The messaging pattern of the polling inbound endpoints is always asynchronous (no request/reply pattern).*

The messages that come through polling inbound endpoints are asynchronous; no response message is sent back to the original message source. The message-processing unit can process the request, invoke the backend service if required, and process the response too, but it cannot send it back. You'll learn more about polling inbound endpoints when we discuss JMS and file-based integration scenarios in the upcoming chapters.

Listening Inbound Endpoints

Listening inbound endpoints are capable of opening up a message listener interface so that the external clients can send requests to those interfaces. An HTTP inbound endpoint is an example of a listening inbound endpoint. An example use case is similar to the scenarios you've seen in proxy service and API. But with inbound endpoints, configuration can be dynamically configured with listening inbound endpoints.

As shown in Listing 3-4, an HTTP inbound can be configured so that it opens up a new port in the specified port and injects all the incoming messages to the specified sequence. The protocols such as HTTP, TCP, and HL7 are implemented as listening inbound endpoints.

Listing 3-4. Exposing a HTTP Interface Using a HTTP Inbound Endpoint

```
<inboundEndpoint name="HttpOrderProcessorInboundEP"          <!-- [1] -->


        protocol="http"                                      <!-- [2] -->
                suspend="false"
                sequence="orderProcSeq" onError="fault" >    <!-- [3] -->
    <p:parameters xmlns:p="http://ws.apache.org/ns/synapse"> <!-- [4] -->
        <p:parameter  name="inbound.http.port">6060</p:parameter>
    </p:parameters>
</inboundEndpoint>
[1] Name of the inbound endpoint.
[2] Protocol that it's listening on.
[3] Message processing and error handling sequences.
[4] Parameters required to start listening on the given protocol such as
HTTP port.
```

You can create an inbound polling endpoint and attach a sequence to it and delegate message-processing tasks to that sequence. In addition, with an HTTP inbound endpoint, you can configure it to dispatch messages to a named sequence, a proxy service, or an API. Figure 3-9 illustrates the message flow of an HTTP inbound endpoint.

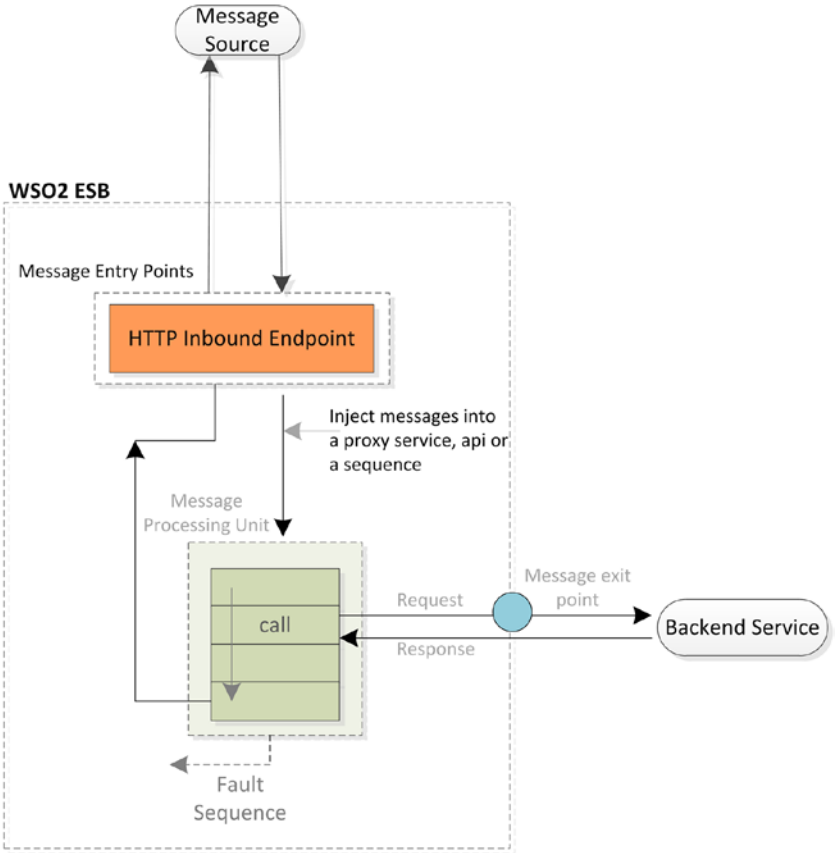

Figure 3-9. *An HTTP inbound endpoint listens for an incoming HTTP request on the specified port and injects the message to the specified sequence, proxy service, or API*

The HTTP inbound endpoint illustrated in Figure 3-9 can listen to the messages on a given HTTP port and inject them into proxy service, API, or sequences. Unlike polling inbound endpoints, listening inbound endpoints allow you to send the message back to the client.

Message Processing Unit: Sequence and Mediators

Once the message enters the WSO2 ESB through the message entry points, the message processing or message mediation occurs in sequences and mediators. A mediator is the most fundamental message-processing unit, but message entry points cannot directly inject a message into a mediator. Instead, the message processing needs to be done with a sequential arrangement of mediators, which is known as a *sequence,* shown in Figure 3-10.

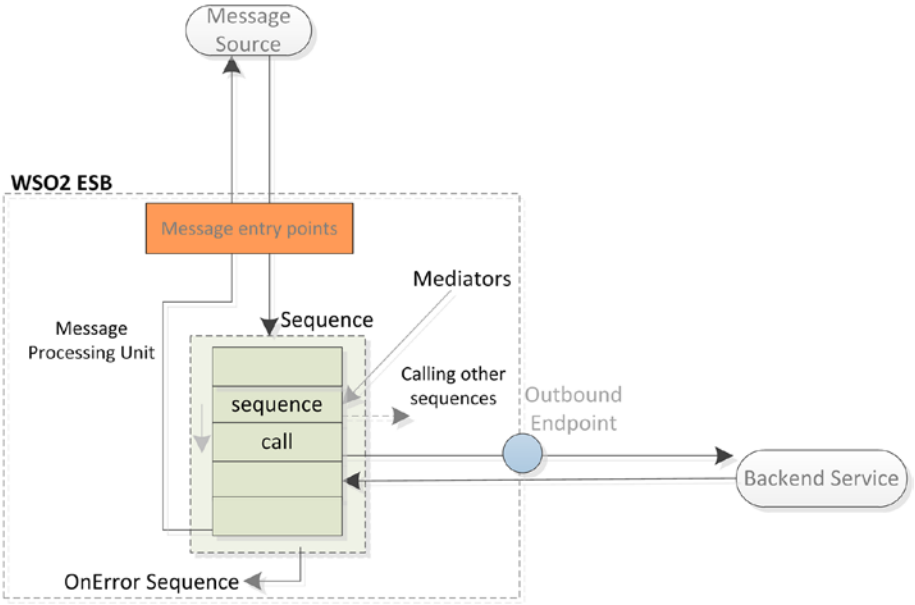

Figure 3-10. *Message processing with sequences and mediators*

You'll look specifically at message processing in Chapter 4, but for now let's talk about basics of sequences and mediators.

Sequences

A sequence can be directly invoked from any message entry point, and you can call a sequence from another sequence. You can also configure the entire message flow and message processing logic in a single sequence, but it's good practice to design integration scenarios such that sequences address different parts of a large integration scenario. That will decompose a complex message processing use case to simple message processing units and foster reuse and troubleshooting capabilities.

```
The sequences can be deployed in $ESB_HOME/repository/deployment/server/
synapse-configs/default/sequences directory.
```

As you learned in proxy services and APIs, they come with built-in special sequences known as in-sequence and out-sequence, but you can configure any existing sequence as the in- or out-sequences. Figure 3-11 illustrates how you can use the dual sequence (in- and out-sequence) model to achieve the same scenario in Figure 3-10. We used the send mediator instead of the call mediator.

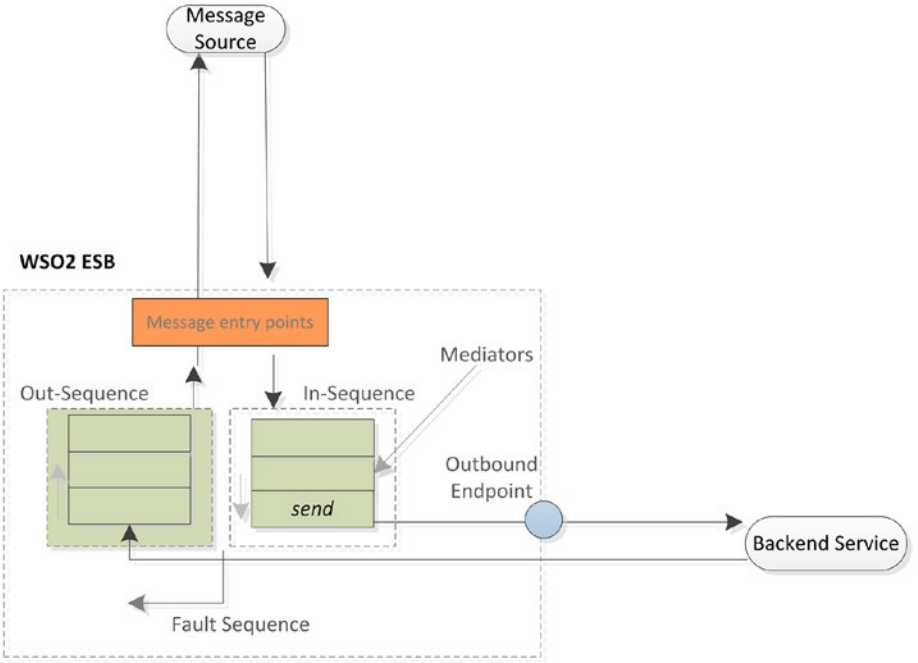

Figure 3-11. *Message processing with sequences and mediators*

In Listing 3-5, you can find a sequence with several mediators that process the incoming message and then call another sequence to validate the message (as shown in Figure 3-10). Therefore, the message validation logic is completely independent from the message processing sequence.

Listing 3-5. Message Processing with Sequences and Mediators

```
<sequence name="OrderProcSeq" onError="errorHandlingSeq"> <!-- [1] -->
    <log level="full"/>                                    <!-- [2] -->
    <property name="test" value="test value"/>             <!-- [3] -->
    <sequence key="messageValidationSeq"/>                 <!-- [4] -->
    <call>                                                 <!-- [5] -->
            <endpoint>
               <address uri="http://localhost:9000/services/
               SimpleStockQuoteService" format="soap11">
                </address>
            </endpoint>
         </call>
    <respond/>                                             <!-- [6] -->
</sequence>
[1] Declaring sequence along with the respective error handling sequeunce.
[2] Log mediator to logs the entire message console and log files.
```

```
[3] Property mediator sets the specified properties to the in-flight message
used in down-stream mediators.
[4] Sequence mediator can be used to call another sequence.
[5] Call mediator to send the request to an external service and receive the
response back to the same place.
[6] Respond mediator sends back the response it received to the original
client.
```

The message flow inside a sequence always follows the exact same sequential order of mediators that appear in the sequence. For example, it's guaranteed that the message validation sequence is executed prior to executing the *call* mediator in Listing 3-5. A given sequence can have an on-error sequence, and if some error occurred in processing messages in a given sequence, then the message is propagated to the specified error sequence so that you can do all the error handling logic in the on-error sequence.

Mediators

A mediator is the simplest message-processing unit in WSO2 ESB. A given mediator can be placed inside a sequence and is capable of processing/mediating the message flowing through it. You can configure a given mediator to process/mediate the message as per your requirements. The message processing or mediation is a broad area, where different types of message processing and mediation take place.

WSO2 ESB comes with numerous mediators that are designed for specific message processing and mediation requirements. And mediators are the key components supporting Enterprise Integration Pattern (EIP). For example, for filtering you can use filters or switch mediators; for transformation you can use xslt, payload factory, or header mediators; for sending messages you can use send, call, or respond mediators; and so on.

You'll learn about most of the mediators of WSO2 ESB in the rest of the book. When you design and develop the message processing logic, consider the following three main scenarios and developing the processing logic accordingly.

Content Unaware Mediation

Message processing logic doesn't require accessing the content of the message. The message processing logic has nothing to do with the content of the message. Therefore, you don't need to worry about the incoming message format. In most scenarios, such as message header based routing, this is all about getting the message from a client through an entry point and sending it to another backend service/system.

Content Aware Mediation with SOAP or JSON

Message processing logic requires accessing the content of the message and the incoming message format is either SOAP or JSON. When you need to process message based on its content, for SOAP and JSON you can use the message format specific processing logic. For example, you can use XPath for SOAP message filtering and JSONPath for JSON message filtering.

Content Aware Mediation with Canonical Format

Message processing logic requires accessing the content of the message when the incoming format is neither SOAP nor JSON. When the incoming message format is neither SOAP nor JSON, the processing logic needs to use a canonical message format (a common message format for all messages). For example, assume that you send a text message to WSO2 ESB and the processing must be done using the canonical message format, which is the SOAP equivalent of the original text message.

You'll discover how to develop message-processing logics for each of these scenarios in Chapter 4.

Message Exit Points: Outbound Endpoints

Now that you're familiar with message entry points as well as with message processing units, the outbound message flow from an ESB is designed on top of a functional component known as outbound endpoints or endpoints. When you want to implement an integration scenario, as shown in Figure 3-12, where ESB has to call a backend service, there should be a way to represent/configure the destination address of the backend service.

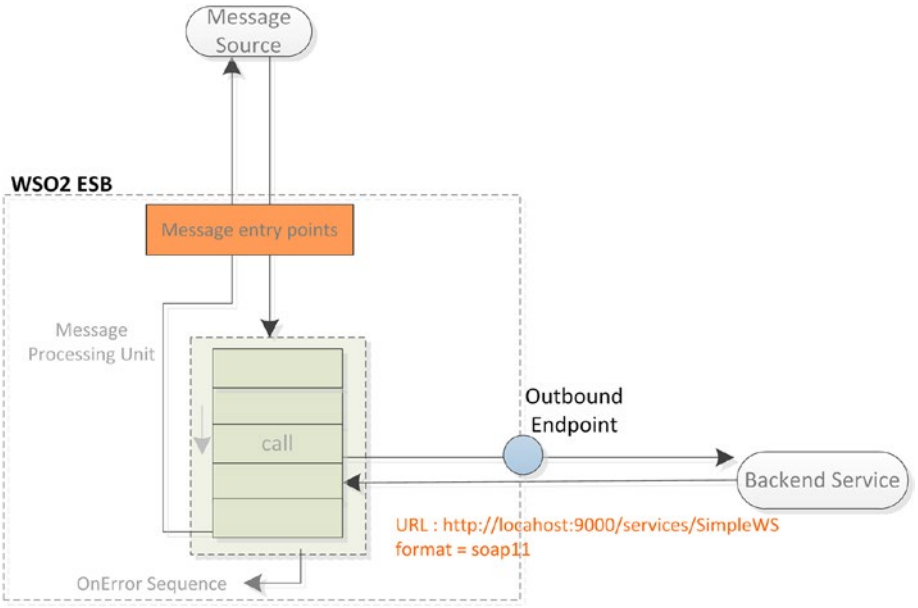

Figure 3-12. *Using outbound endpoints to represent external backend services and systems and configure outbound messaging behavior*

Listing 3-6 shows how an endpoint can be used along with sequences and mediators. In this example, the message coming into the `ShoppingInfo` API is sent (using call mediator) to the backend service, which is specified in the URI of the endpoint over the HTTP transport.

Listing 3-6. Using Outbound Endpoints

```
<api xmlns="http://ws.apache.org/ns/synapse" name="ShoppingInfo" context="/
ShoppingInfo">
   <resource methods="POST">
      <inSequence>
         <log/>
         <call>
            <endpoint>                          <!-- [1] -->
               <address
uri="http://localhost:9000/services/SimpleStockQuoteService"<!-- [2] -->
                        format="soap11">              <!-- [3] -->
</address>
            </endpoint>
         </call>
         <respond></respond>
      </inSequence>
   </resource>
</api>
[1] Declaring Endpoint.
[2] Address of the backend service.
[3] Change the outbound messaging format to SOAP 1.1 from the existing
message format.
```

An outbound endpoint/endpoint is a logical representation of an external destination for an outgoing message. An endpoint can logically represent one or more backend service destinations in ESB. An endpoint must be used inside a send or call mediator because they're the mediators that can be used for outbound messaging. In Listing 3-6, the endpoint definition is done inline with the sequence configuration, but it's also possible to define the endpoint as a separate entity and refer that using call or send mediators. In WSO2 ESB, you can create and deploy your endpoints under `$ESB_HOME/repository/deployment/server/synapse-configs/default/endpoints`.

You have already tried most of the use cases that uses endpoints. Hence I'm not going to discuss simple use cases related to how to use outbound endpoints. Let's proceed to the discussion on endpoint types. Several endpoints types are available in WSO2 ESB. Let's look at the commonly used endpoint types and their usages.

Endpoint Types

The commonly used endpoints types are *address endpoint, http endpoint, load balancing,* and *failover endpoints.* Choose the most appropriate endpoint type for your use case.

Address Endpoint

The address endpoint can represent any type of an endpoint reference (EPR) inside the WSO2 ESB. Therefore, the address endpoint can be used to represent any remote service provider such as a web service exposed over HTTP, a JMS queue, or even a file location. Listing 3-7 shows the available configuration options of address endpoints.

Listing 3-7. Address Endpoints

```
<address uri="http://localhost:7070/OrderService"
        format="soap11|soap12|pox" optimize="mtom|swa"
        encoding="charset encoding">

<enableSec policy="path_to_key">
<enableAddressing/>
</address>
```

In addition to the remote endpoint URL, several other endpoint attributes are available for changing the format and the behavior of outbound messaging.

`format` specifies the message for the outbound message for that endpoint. If you don't specify the message format, there won't be any modifications to the outbound message format. If you need to alter the message format of the inbound message to a different message format, you can specify the required outbound message format with these values for format attribute: soap11 - transforming message to SOAP 1.1, soap12 - transforming message to SOAP 1.2, or pox - Plain Old XML (POX) - transforming to plain old XML format.

QoS (Quality of service) aspects such as `WS-Security` and `WS-Addressing` may be enabled on messages sent to an endpoint using the `enableSec` and `enableAddressing` elements. Optionally, the `WS-Security` policies can be specified using the policy attribute `enableSec policy="key"`. This enables `WS-Security` as per the provided policy, for the outbound message. Optimize is used for optimization of the message, which transfers binary data (SwA - Optimized as a SwA (SOAP with Attachment) message and MTOM—Optimized as a MTOM).

You'll learn more about using endpoint outbound messaging techniques in Chapter 4.

HTTP Endpoint

HTTP endpoint is designed for invoke RESTful services and APIs from WSO2 ESB. HTTP endpoint is a logical representation of an actual resource that allows users to specify the resource URL and HTTP method, when invoking a RESTful service/API. In the example shown in Listing 3-8, the resource URL is provided as an URI template with variables that would get resolved during the runtime. For example, {`uri.var.servicename`} needs to be defined in the message flow prior to use the endpoint. Also, you can specify the HTTP method for the outbound request.

Listing 3-8. HTTP Endpoints

```
<endpoint xmlns="http://ws.apache.org/ns/synapse" name="HTTPEndpoint">
    <http uri-template=        "http://localhost:8080/restapi/{uri.var.
servicename}/menu?category={uri.var.category}&amp;type={uri.var.pizzaType}"
                method="GET">
    </http>
</endpoint>
```

The URI template is fully compliant with the RFC 6570 [http://tools.ietf.org/html/rfc6570] specification, and the variable names must start with `uri.var.*` or `query.param.*`. You'll find the real-world use cases related to RESTful integration by using HTTP endpoint in Chapter 5.

Load Balancing and Fail-Over Endpoint

In real-world integration use cases, it's essential to have multiple backend services to enable load distribution and high availability (or failover). Therefore, ESB needs to invoke the services, which are available as independent server instances but offer the same functionality.

The load balancing endpoint allows you to logically group a set of backend services that you need to distribute among the load. The load-balanced endpoint distributes the messages, arriving among a set of listed endpoints evaluating the load balancing policy and any other relevant parameters.

Listing 3-9. Load Balancing Endpoint

```
<send>
    <endpoint>
        <loadbalance>
            <endpoint>
                <address uri="http://localhost:9001/services/LBService1">
                    <enableAddressing/>
                    <suspendDurationOnFailure>60</suspendDurationOnFailure>
                </address>
            </endpoint>
            <endpoint>
                <address uri="http://localhost:9002/services/LBService1">
                    <enableAddressing/>
                    <suspendDurationOnFailure>60</suspendDurationOnFailure>
                </address>
            </endpoint>
            <endpoint>
                <address uri="http://localhost:9003/services/LBService1">
                    <enableAddressing/>
                    <suspendDurationOnFailure>60</suspendDurationOnFailure>
                </address>
            </endpoint>
```

```
        </loadbalance>
    </endpoint>
</send>
```

Listing 3-9 demonstrates a load balancing endpoint, which balances the load among three backend service instances (they are on the same host but different ports). The load-balancing algorithm can be configured, and by default it uses the round robbing algorithm (using the `policy` and `algorithm` attributes). Also, you can configure the next endpoint once the currently selected endpoint has failed by configuring the `failover` attribute (by default it's set to `true`).

Similarly, *failover endpoints* send messages to the listed endpoints with the following failover behavior. At the start, the first listed endpoint is selected as the primary and all other endpoints are treated as backups. Incoming messages are always sent only to the primary endpoint. If the primary endpoint fails, the next active endpoint is selected as the primary and the failed endpoint is marked as inactive.

Understanding Endpoint States and Endpoint Attributes

As the endpoints are the representation of the actual backend services, the endpoint has different states. At any given time, a given endpoint can be in the `ACTIVE`, `TIMEOUT`, `SUSPENDED`, or `OFF` state. The behavior of endpoints in each state can be summarized as follows.

- `ACTIVE`: Endpoint is running and handling requests.
- `TIMEOUT`: Endpoint encountered an error but still can send and receive messages. If it continues to encounter errors, it will be suspended.
- `SUSPENDED`: Endpoint encountered errors and cannot send or receive messages. Incoming messages to a suspended endpoint result in a fault.
- `OFF`: Endpoint is not active. To put an endpoint into the `OFF` state, or to move it from `OFF` to `ACTIVE`, you should use JMX.

The main reason to have a set of endpoint states is to avoid the overhead on sending requests to the services, which are not up and running or not properly functioning. The endpoint configuration allows you to configure the state transitions based on your requirements (types of the error that have occurred, duration, and so on). To understand the endpoint states and the relevant configuration parameters in detail, consider the endpoint configuration in Listing 3-10.

Listing 3-10. Understanding Endpoint States

```
<endpoint name="Sample_First" statistics="enable" >
        <address uri="http://localhost/myendpoint"      <!-- [1] -->
        statistics="enable" trace="disable">
```

```
        <timeout>                    <!-- [2] -->
        <duration>60000</duration>
        </timeout>
                <markForSuspension>            <!-- [3] -->
                        <errorCodes>101504, 101505</errorCodes>
                        <retriesBeforeSuspension>3</retriesBeforeSuspension>
                        <retryDelay>1</retryDelay>
                </markForSuspension>

                <suspendOnFailure>                <!-- [4] -->
                        <errorCodes>101500, 101501, 101506, 101507, 101508
                        </errorCodes>
                        <initialDuration>1000</initialDuration>
                        <progressionFactor>2</progressionFactor>
                        <maximumDuration>60000</maximumDuration>
                </suspendOnFailure>
        </address>
</endpoint>
[1] Configuring  the endpoint URI.
[2] Endpoint should wait for 60s for a response and if no response received,
it goes to the timeout state.
[3] Endpoint is moved into timeout state for the specified error codes, and
there will be three retries from the same endpoint before it moves into the
suspended state.
[4] Endpoint is suspended for the specified error codes, and you can
configure the suspension time.
```

In that particular endpoint configuration, you can find the backend service URL and several other configuration parameters. The endpoint behavior for this configuration can be explained as follows.

- *TIMEOUT*: If the remote endpoint does not respond within the specified time duration (in ms), it will be marked as TIMEOUT. If the remote endpoint does not respond within 60000 ms, this endpoint will move into the TIMEOUT state.
- markForSuspension: Here you can configure the endpoint behavior related to the TIMEOUT state. You can specify the error codes that would move the endpoint into the TIMEOUT state. When the ESB sends a request to the remote endpoint and gets one of these errors—101504 (Connection Timeout) or 101505 (Connection closed)—then the endpoint will be moved into the TIMEOUT state. (A complete list of endpoint error codes can be found at http://docs.wso2.com/enterprise-service-bus/Error+Handling#ErrorHandling-codes.)

- With `retriesBeforeSuspension`, you can configure the number of times the endpoint should be used to send requests when these errors occur. Therefore, after three requests to the same endpoint, fail due to these errors, the endpoint is moved into the `SUSPENDED` state. This duration between each retry is 1s and it's configured with `retryDelay`.
- *`suspendOnFailure`*: If an error occurred that's specified with error codes 101500, 101501, 101506, 101507, or 101508, then the endpoint is moved into the `SUSPENDED` state. For any other error code that's not specified in `suspendOnFailure` error codes, the endpoint remains in the `ACTIVE` state.
- When the endpoint is first suspended, the retry happens after 1 second. Because the progression factor is 2, the next suspension duration before retry is 2 seconds, then 4, then 8, and so on until it gets to 60 seconds, which is the maximum duration we have configured. At this point, all subsequent suspension periods will be 60 seconds until the endpoint succeeds and is back in the `ACTIVE` state, at which point the initial duration will be used on subsequent suspensions.

As you learned about a sample scenario in which you can configure the endpoint behavior, let's look closer at the endpoint states and transitions.

Based on our discussion with the scenario in Listing 3-10, we can come up with the endpoint transition diagram illustrated in Figure 3-13. The following is a generic explanation of each endpoint state and its transitions.

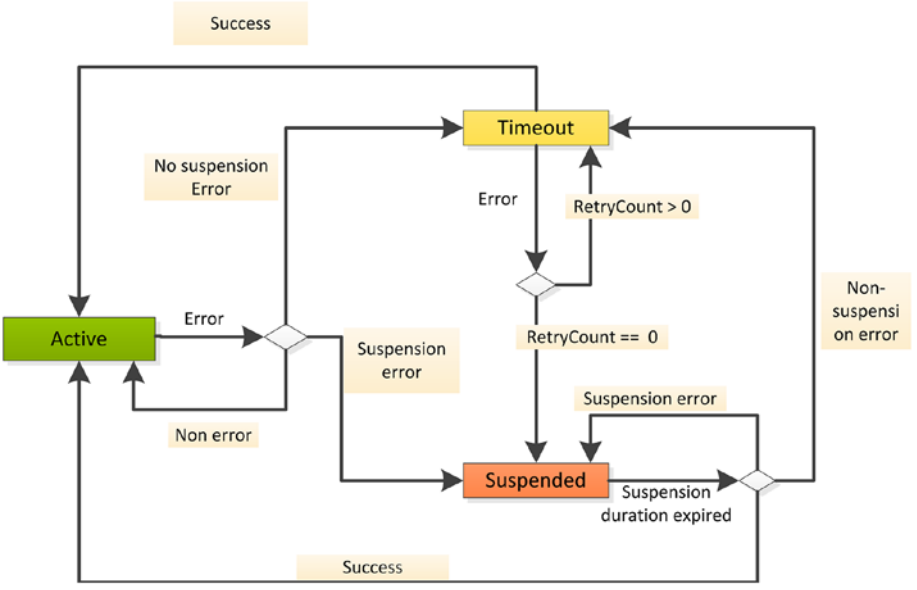

Figure 3-13. *Outbound endpoint state transitions*

ACTIVE State

When you create a new endpoint, it goes into the ACTIVE state. The endpoint is live and will route all the requests to the backend service. The endpoint will be in the ACTIVE state as long as no errors invoke the backend service.

The endpoint can be configured to stay in the ACTIVE state or to go to the TIMEOUT or SUSPENDED states based on the error codes you configure for those states. When an error occurs, the endpoint checks to see whether it's a TIMEOUT error first, and if not, it checks to see whether it's a SUSPENDED error. If the error isn't defined for either TIMEOUT or SUSPENDED, the error will be ignored and the endpoint will stay ACTIVE.

If no error codes are specified, the "HTTP Connection Closed" and "HTTP Connection Timeout" errors are considered TIMEOUT errors, and all other errors put the endpoint into the SUSPENDED state.

TIMEOUT State

When an endpoint is in the TIMEOUT state, it will attempt to receive messages until one message succeeds or the maximum retry setting is reached. If the maximum is reached, the endpoint is marked as SUSPENDED. If one message succeeds, the endpoint is marked as ACTIVE.

For example, let's assume the number of retries is set to three. When an error occurs and the endpoint is set to the TIMEOUT state, the ESB can try to send up to three more messages to the endpoint. If the next three messages sent to this endpoint result in an error, the endpoint is put in the SUSPENDED state. If one of the messages succeeds before the retry maximum is met, the endpoint will be marked as ACTIVE.

SUSPENDED State

A SUSPENDED endpoint cannot send or receive messages. When an endpoint is put into this state, the ESB waits until after an initial duration has elapsed (the default is 30 seconds) before attempting to send messages to this endpoint again. If the message succeeds, the endpoint is marked as ACTIVE. If the next message fails, the endpoint is marked as SUSPENDED or TIMEOUT depending on the error, and the ESB waits before retrying messages using the following formula:

```
Min(current suspension duration * progressionFactor, maximumDuration)
```

Default Configuration and Disabling Endpoint Suspension

You can configure the initial suspension duration, progression factor, and maximum duration as part of the suspendOnFailure settings. On each retry, the suspension duration increases up to the maximum duration.

When in the TIMEOUT state, markForSuspension or suspendOnFailure are not configured; endpoints will use the default timeout of 120s. When there is no response received for that duration, the endpoint is moved to the suspended state immediately.

And also, if there is any error occurred with the same endpoint configuration, the endpoint again goes to the SUSPENDED state.

If you want to disable endpoint suspension behavior to not suspend endpoints at all, then you can configure `suspendOnFailure errorCodes` with `-1` and `markForSuspension errorCodes` with `-1`.

```
<timeout>
     <duration>60000</duration>
     <responseAction>fault</responseAction>
 </timeout>
 <suspendOnFailure>
     <errorCodes>-1</errorCodes>
     <initialDuration>0</initialDuration>
     <progressionFactor>1.0</progressionFactor>
     <maximumDuration>0</maximumDuration>
 </suspendOnFailure>
 <markForSuspension>
     <errorCodes>-1</errorCodes>
 </markForSuspension>
```

You can use this endpoint configuration for the integration scenarios that you never want to suspend in a given endpoint.

Scheduled Tasks

In real-world integration use cases, you may need to have scheduled jobs and tasks running on your ESB runtime. For example, assume that you need to invoke a backend service with the time interval 1000ms 10 times. As you learned in previous sections, you can implement the service invocation logic inside a sequence (with all the required payloads). Now you need to invoke that sequence periodically. For that you can use a scheduled task in WSO2 ESB.

In WSO2 ESB, a task runs a piece of code triggered by a timer, allowing you to run scheduled jobs at specified intervals. In the task configuration, using the count and interval attributes, you can run the task a specified number of times and at a given interval. In addition, you can give a scheduled time as a cron-style entry. The tasks are deployed in WSO2 ESB under the `$ESB_HOME/repository/deployment/server/synapse-configs/default/tasks` directory.

As illustrated in Figure 3-14, a scheduled task can inject messages into a named sequence, proxy service, or an API.

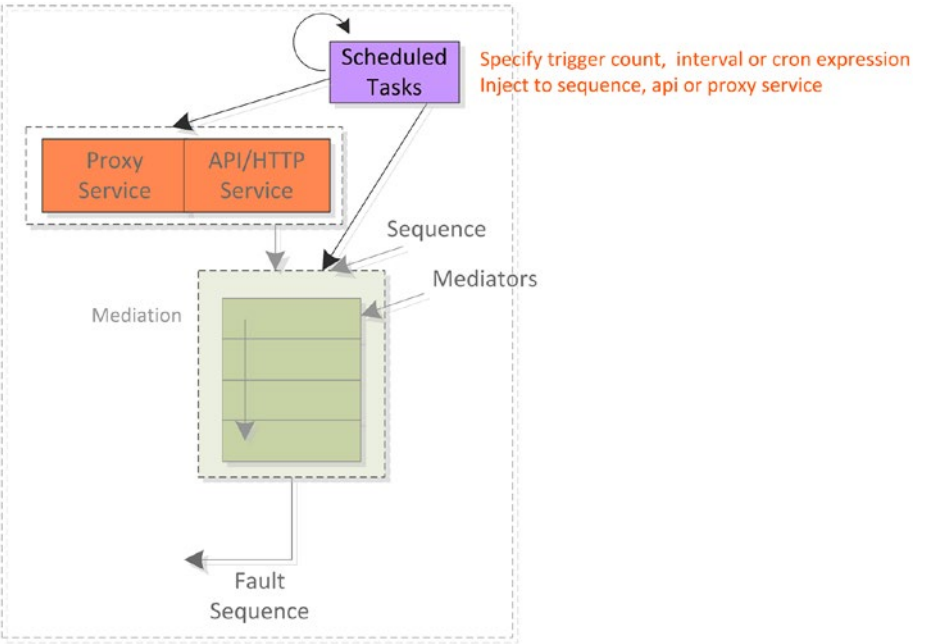

Figure 3-14. *Scheduled tasks can periodically inject messages into a sequence, proxy service, or API*

Let's take the same example and assume that you need to invoke a sequence named `SampleSequence` periodically with the interval 1000ms 10 times. Listing 3-11 shows scheduled tasks that can inject the message into a specified sequence (with the properties `injectTo` and `sequenceName`). If you need to set a message payload prior to injecting the message, then the message property can be set too.

Listing 3-11. Scheduled Tasks Inject Messages into a Named Sequence

```
<task name="SampleInjectToSequenceTask"
        class="org.apache.synapse.startup.tasks.MessageInjector" <!-- [1] -->
        group="synapse.simple.quartz">
     <trigger count="10" interval="1000"/>           <!-- [2] -->
       <property                                     <!-- [3] -->
              xmlns:task="http://www.wso2.org/products/wso2commons/tasks"
              name="message">
              <m0:getQuote xmlns:m0="http://services.samples">
          <m0:request>
             <m0:symbol>WSO2</m0:symbol>
          </m0:request>
        </m0:getQuote>
    </property>
```

```
        <property xmlns:task="http://www.wso2.org/products/wso2commons/
        tasks"
              name="injectTo"
              value="sequence"/>                    <!-- [4] -->

      <property xmlns:task="http://www.wso2.org/products/wso2commons/tasks"
              name="sequenceName"
              value="SampleSequence"/>
</task>
[1] Using the default message injector task can inject a given message to a
sequence, proxy service. or API.
[2] Triggering count and the interval.
[3] Payload of the message to be injected into the ESB.
[4] Specifying the  way that you want to inject the message into the ESB.
```

This example shows a common usage of the message injector scheduled tasks that come with WSO2 ESB. But if you need to have any other task implementation, that can be plugged into the ESB in a similar manner. From WSO2 ESB 4.9 onward, the distributed tasks coordination is also supported. You can execute tasks in a cluster of ESB nodes and at a given time only one node will execute the tasks. If one node fails, another cluster node will pick up the task execution.

Summary

In this chapter you learned that:

- The key components of WSO2 ESB are message entry points, message processing units, and message exit points.
- The three main types of message entry point are proxy service, API, and inbound endpoint.
- Proxy services are used for exposing web service interfaces over HTTP and offer all the web service-related functionalities such as `WS-Addressing`, WSDL, and `WS-Security`.
- API/HTTP services can be used to expose RESTful HTTP interfaces from ESB.
- Both proxy services and APIs leverage the underlying transports for a given protocol (such as HTTP). The transport configuration is static and shared among all the APIs and proxy services, which use a given protocol.
- Unlike API and proxy services, inbound endpoints have their own configuration to receive messages from different protocols. Therefore, inbound endpoints are used for dynamically configuring inbound message channels, such as JMS, file, and HTTP.

- Inbound endpoints are executed in polling modes (for example, JMS) or listening mode (for example, HTTP).
- Message processing units are configured as sequences of mediators. Each mediator has a specific function and is a reusable block of message processing code. For example, it can send a message to the server and send a response to the client.
- Endpoints are used to represent remote services or systems that ESB sends messages to. You can configure the outbound message format (such as SOAP 1.1), the duration that endpoint waits until the backend responds, and so on.
- Scheduled tasks are often used to inject messages into queues and start processing.

CHAPTER 4

Processing Messages with WSO2 ESB

Most ESB developers will spend a lot of time configuring message-processing logic to cater to their integration needs. In WSO2 ESB, there are several message processing techniques you can use and they are key to implementing any integration scenario. In the previous chapter, you learned the basics of message processing. This chapter carries on with the message processing and dives deep into the eight techniques you can use inside WSO2 ESB.

- *Message pass-through:* Request-response and one-way message passing through ESB without processing the message content
- *Message filtering and switching:* Implement filter/switch conditions in message flow based on message content or message attributes.
- *Message transformations:* Translate the message from one message format to the other.
- *Message enriching:* Enrich the message with a segment of external message content.
- *Message validation:* Validate a message format against a schema.
- *Service orchestration:* Invoke multiple backend services to server a single request.
- *Protocol and message format transformations:* Transform between different wire protocols and message formats.
- *Manipulating a message flow with properties:* Use properties to change the message flow and message attributes.

As you saw in Chapter 3, ESB message processing is done inside message processing units (sequences and mediators) and a given message can come from any message entry point as shown in Figure 4-1.

K. Indrasiri, *Beginning WSO2 ESB*, DOI 10.1007/978-1-4842-2343-7_4

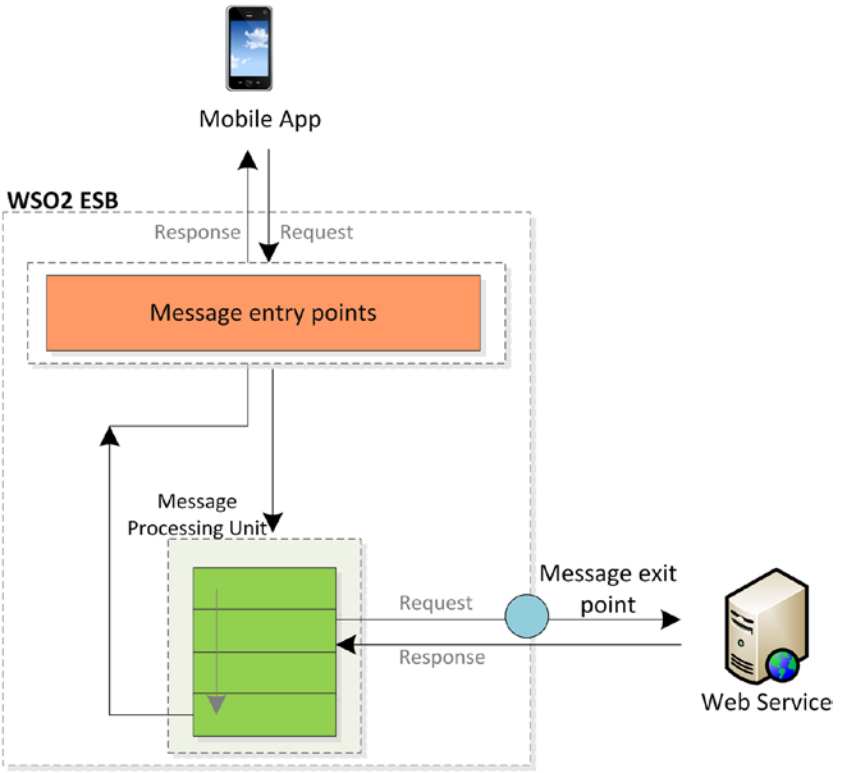

Figure 4-1. *WSO2 ESB has three main pieces: entry points, message processing units, and exit points. Each one is independent of the other two.*

The message processing logic is independent of the message entry point and so any message processing technique can be applied to a message regardless of the entry point that the message comes through.

It's also worth remembering from Chapter 3 that one of three scenarios will apply to any message processing logic you develop.

- The message processing logic doesn't need access to the content of the message.
- The message processing logic has nothing to do with the content of the message. Therefore, you don't need to worry about the incoming message format. In most scenarios, this is all about getting the message from a client through an entry point and sending it to another backend service/system.
- The message processing logic does need access to the content of the message and the incoming message format is either SOAP or JSON.

- When you need to process message based on its content, for SOAP and JSON you can use processing logic specific to that format. For example, you can use XPath for SOAP message filtering or JSONPath for JSON message filtering.
- The message processing logic does need access to the content of the message and the incoming format is neither SOAP nor JSON.
- When the incoming message format is not SOAP or JSON, your processing logic needs to assume there is a common message format for all messages (a canonical message format).

The main objective of this chapter is to provide you with a comprehensive overview of the eight message processing techniques as applied to these scenarios.

Pass-Through Messaging

Message pass-through is about sending messages through ESB without processing the message content or message attributes. Message pass-through can be implemented for request-response or one-way messaging scenarios. To understand the concept of message pass-through concepts, suppose a financial services company that wants to expose a set of business functionalities as a web service. This particular web service allows you to:

- Get a stock quote using `getQuote`: If you have a certain stock that you are interested in, you can get the price (stock quote) of that stock by providing its name.
- Place a stock order using `placeOrder`: You can place orders to buy options contracts and mutual funds.

The existing software solution of the financial service company has a web service that offers the `getQuote` functionality but for order placing what they have is a queuing mechanism where the orders need to be place in a JMS queue. You can use WSO2 ESB to build a web service layer around the existing software systems, so that ESB exposes a web service with two operations for `getQuote` and `placeOrder`.

By the nature of the business requirement of these operations, the `getQuote` operation is a two-way operation where you have to send some data and you expect a response data to be returned from the service (such as HTTP 200 OK with the response message in HTTP). Therefore, this is a *synchronous* request-response operation. The use case can be summarized as shown in Figure 4-2.

Figure 4-2. *In synchronous messaging when the web service client sends a request, it waits and expects a response message. WSO2 ESB supports synchronous messaging between the client and ESB, and also between ESB and the backend service.*

You can implement the synchronous messaging scenario in WSO2 ESB using any preferred message entry point and a message processing logic with the outbound endpoint. For financial service `getQuote` operation, you can use a mediation logic shown in Listing 4-1, in which we have used a proxy service as the message entry point. For simplicity, we assumed that the client is sending the same payload that is accepted by the backend service. Therefore no message transformation logic is required in the request message flow and you can simply send the request to the backend service via the call mediator. The response is sent back to the client without transforming it.

Listing 4-1. Implementing the getQuote Synchronous Messaging Scenario

```
<proxy xmlns="http://ws.apache.org/ns/synapse"
       name="StockQuoteProvider"
       transports="https http"
       startOnLoad="true"
       trace="disable">
   <description/>
   <target>
      <inSequence>                                    <!-- [1] -->
         <call>                            <!-- [2] -->
            <endpoint>
               <address          uri="http://127.0.0.1:9000/services/
                                 SimpleStockQuoteService"/>
            </endpoint>
         </call>
         <respond/>                                 <!-- [3] -->
      </inSequence>
   </target>
</proxy>

[1] Request is processed in the in-sequence of the proxy service.
[2] Request is directly sent to backend service without transforming it.
[3] Response is sent back to the client (without doing any changes).
```

On the other hand, `placeOrder` is a one-way operation, where you send the order details but do not expect a response from the service other than a status response from backend service (such as 202—Accepted in HTTP). Therefore `placeOrder` uses asynchronous messaging exchange pattern, which is illustrated in Figure 4-3.

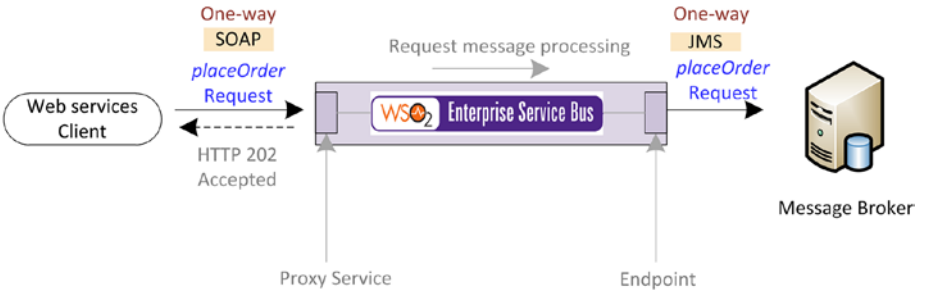

Figure 4-3. *The web service client sends the placeOrder request and doesn't wait for a response message. When the request is received by ESB, it sends a status message (HTTP 202 Accepted) to acknowledge the receipt of the message but not a response message. Then ESB queues the message and completes the asynchronous message flow.*

The implementation of the `placeOrder` asynchronous messaging scenario with WSO2 ESB is shown in the Listing 4-2. Here we have again used a proxy service and the `placeOrder` request is sent to the JMS queue via the call mediator by specifying the JMS endpoint address. But unlike with the synchronous messaging scenario, you don't need to send a response back to the client, but a status message acknowledging the receipt of the `placeOrder` request. For that, you can use a property (or a flag) `FORCE_SC_ACCEPTED` to indicate that the mediation flow should send the client a status message with HTTP 202 Accepted so that the client stops waiting for a response.

Now you have notified the client, and you can proceed with the `placeOrder` request. But prior to sending the `placeOrder` request, you need to specify another property/flag to indicate the message flow that you are doing a message exchange that initiates from ESB and goes out (but no response is accepted). This is done using the `OUT_ONLY` property, which indicates to ESB message flow that once the message is sent to the JMS queue, it should not wait for a response.

Listing 4-2. One-way messaging

```
<proxy xmlns="http://ws.apache.org/ns/synapse"
       name="OrderManagerProxy"
       transports="https http"
       startOnLoad="true"
       trace="disable">
   <description/>
   <target>
       <inSequence>                                          <!-- [1] -->
           <property name="FORCE_SC_ACCEPTED" value="true" scope="axis2"/>
                                                             <!-- [2] -->
```

```
            <property name="OUT_ONLY" value="true"/> <!-- [3] -->
            <call>
                <endpoint>   <!-- [4] -->
                    <address uri="jms:/StockQuotesQueue?transport.jms.Con
                    nectionFactoryJNDIName=QueueConnectionFactory&amp;ja
                    va.naming.factory.initial=org.wso2.andes.jndi.Properti
                    esFileInitialContextFactory&amp;java.naming.provider.
                    url=repository/conf/jndi.properties&amp;transport.jms.
                    DestinationType=queue"/>
                </endpoint>
            </call>
        </inSequence>
    </target>
</proxy>
[1] Request is processed in the in-sequence of the proxy service.
[2] this indicates the ESB that it should immediately send back a HTTP 202
Accepted status to the client, once it receives the message.
[3] Indicates the ESB that the outbound messaging is one way and no need to
wait for a response.
[4] Sending to the JMS endpoint with the use of the call mediator.
```

When you are building enterprise integration scenarios with WSO2 ESB, the initial task is to identify that whether you need a synchronous or asynchronous message exchange pattern. Therefore identifying the message exchange pattern and building the basic message flow will be the first step of any integration scenario. Once you have implemented the required message exchange pattern, you can proceed with other business requirements such filtering, transformations, and so on and so forth.

Message Filtering and Switching

In most of the enterprise integration solutions, based on the message attributes (headers, content) you need to apply certain message processing logics. For example, consider the same financial service scenario. As shown in Figure 4-4, suppose that the financial organization wants to serve `getQuote` requests with a specific quote name/symbol and reject all other requests.

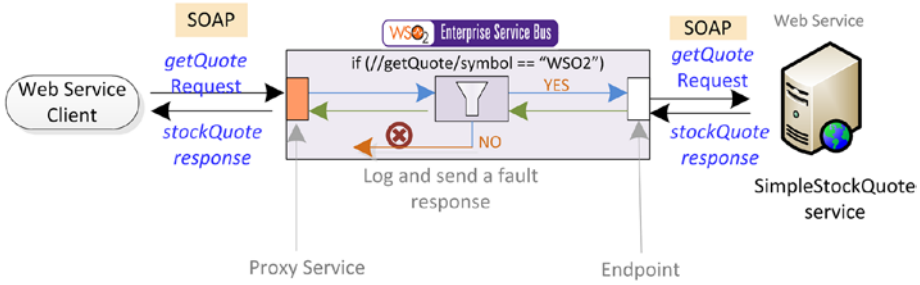

Figure 4-4. *Filter mediator can be used to filtering getQuote request with XPath and process the request only if the message contains WSO2 as the symbol. If it doesn't it logs a log message and sends a fault response back to the client.*

You can implement this by using a filter inside the request message flow of WSO2 ESB. As you can observe in Figure 4-4, the incoming request is in SOAP format. Therefore we need to write our logic so that we can filter a given SOAP message with the specified quote name. The request message format is shown in Listing 4-3.

Listing 4-3. Request Message Format

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ser="http://services.samples" xmlns:xsd="http://services.samples/xsd">
    <soapenv:Header/>
    <soapenv:Body>
        <ser:getQuote>
            <ser:request>
                <xsd:symbol>WSO2</xsd:symbol>
            </ser:request>
        </ser:getQuote>
    </soapenv:Body>
</soapenv:Envelope>
```

In WSO2 ESB, a message filter can be configured using the `filter` mediator and you can check for the specified message content using XPath (`http://www.w3.org/TR/xpath20/`) and a regular expression.

Listing 4-4. Implementing a Filtering Scenario with a Filter Mediator

```
<proxy xmlns="http://ws.apache.org/ns/synapse"
       name="MessageFilterProxy_4.3"
       transports="https http"
       startOnLoad="true"
       trace="disable">
    <description/>
    <target>
        <inSequence>
            <filter                              <!-- [1] -->
```

```
source="//ser:getQuote/ser:request/xsd:symbol" <!-- [2] -->
regex="WSO2*"                                  <!-- [3] -->
xmlns:ser="http://services.samples"            <!-- [4] -->
xmlns:xsd="http://services.samples/xsd">
                <then>                         <!-- [5] -->
                    <log level="custom">       <!-- [6] -->
                        <property
name="MessageFlowInfo"
value="Filter condition satisfied"/>
                    </log>
                    <call>                    <!-- [7] -->
                        <endpoint>
                            <address
        uri="http://127.0.0.1:9000/services/SimpleStockQuoteService"/>
                        </endpoint>
                    </call>
                    <respond/>                <!-- [8] -->
                </then>
                <else>                        <!-- [9] -->
                    <log level="custom">      <!-- [10] -->
                        <property name="MessageFlowInfo"
value="Filter condition NOT satisfied."/>
                    </log>
                    <makefault xmlns="http://ws.apache.org/ns/synapse"
                    <!-- [11] -->
                                  version="soap11">
                          <code
xmlns:soap11Env="http://schemas.xmlsoap.org/soap/envelope/"
value="soap11Env:Client" />
                        <reason value="Unsupported company name." />
                    </makefault>
                    <respond/>                  <!-- [12] -->
                </else>
            </filter>
        </inSequence>
    </target>
</proxy>
[1] Filter incoming message based on the content of the message.
[2] Select required element from the request message that should be used for
filtering the message.
[3] matches the selected value with a given regular expression
[4] Namespaces used in request message that are required for evaluating the
message.
[5] If the filtering condition is satisfied, then it goes to this message
processing logic.
[6] Print a custom log message.
[7] Sending message to the backend service.
[8] Response is received at this point and respond back to the client.
```

```
[9] If the condition specified in the filter mediator is not satisfied, then
this message logic will be executed.
[10] log a custom log message.
[11] creating a SOAP fault with fault mediator. (Either you can use Fault
mediator or Payload factory to create your own custom error response)
[12] Respond back the fault message back to the client.
```

The full configuration of the implementation of the financial service-filtering scenario is shown in Listing 4-4. In the filter mediator configuration, you can specify the `source` attribute, which is the expression to locate the value that matches the regular expression that you can define with the `regex` attribute. Therefore any request that has the symbol name WSO2 will be sent to the backend service and any other request will be rejected. The fault response will then be sent back to the client.

Message Filtering

As you have seen in the message filtering scenario, the *filter* mediator is used to filter messages based on various conditions such as content of the message and message headers or based on any other variable (or property) that is defined in the message flow. Filter mediator is based on the generic programming language concept of an if-then-else statement.

The generic syntax of a filter mediator is shown in Listing 4-5.

Listing 4-5. Filter Mediator Syntax

```
<filter
         (source="[XPath|json-eval(JSONPath)]"
                  regex="string") | xpath="[XPath|json-eval(JSONPath)]">
   mediator+
</filter>
```

Filter mediators can be configured either to use the `source` and `regex` combination (as you have seen in the financial service example) or use the `xpath` attribute to look for the presence of a given element. With both `source` and `xpath` expressions, you can use XPath or JSONPath expressions.

Message Switching

Unlike filter mediator, the switch mediator can have a number of possible execution paths. For example, supposed that the financial service wants to selectively serve the `getQuote` requests for companies WSO2 and XYZ and drop all the other requests. But unlike with the filter use case, assume that the request is coming from a mobile client and hence it is in JSON message format. Therefore, you need to use a switch mediator and you can select the value for the switch expression using JSONPath (`https://code.google.com/p/json-path/`) as the incoming message format is JSON. Suppose that incoming JSON message has the same format.

```
{
  "getFinancialQuote": { "company": "WSO2" }
}
```

The Listing 4-6, shows the complete configuration of this use case.

Listing 4-6. Using Switch Mediator for Message Switching

```
<api xmlns="http://ws.apache.org/ns/synapse"
     name="StockQuoteAPI"
     context="/StockQuoteAPI">
    <resource methods="POST">
        <inSequence>
            <switch source="json-eval($.getFinancialQuote.company)">
            <!-- [1] -->
                <case regex="WSO2">        <!-- [2] -->
                    <sequence key="StockQuoteReqSeq"/>
                    <log level="custom">
                        <property name="MessageFlowInfo"
value="== WSO2 Company =="/>
                    </log>
                    <respond/>
                </case>
                <case regex="XYZ">         <!-- [3] -->
                    <sequence key="StockQuoteReqSeq"/>
                    <log level="custom">
                        <property name="MessageFlowInfo"
value="== XYZ Company =="/>
                    </log>
                    <respond/>
                </case>
                <default>                  <!-- [4] -->
                    <log level="custom">
                        <property name="MessageFlowInfo"
value="Filter condition NOT satisfied."/>
                    </log>
                    <drop/>
                </default>
            </switch>

        </inSequence>
    </resource>
</api>

<sequence xmlns="http://ws.apache.org/ns/synapse"
          name="StockQuoteReqSeq">      <!-- [5] -->
    <payloadFactory media-type="xml">       <!-- [6] -->
        <format>
            <soapenv:Envelope
```

```
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                                xmlns:ser="http://services.samples">
                  <soapenv:Header/>
                  <soapenv:Body>
                      <ser:getSimpleQuote>
                          <ser:symbol>$1</ser:symbol>
                      </ser:getSimpleQuote>
                  </soapenv:Body>
              </soapenv:Envelope>
          </format>
          <args>
              <arg evaluator="json"
expression="$.getFinancialQuote.company"/>
          </args>
      </payloadFactory>
      <header name="Action" value="urn:getSimpleQuote"/>
      <call>
          <endpoint>
              <address
uri="http://localhost:9000/services/SimpleStockQuoteService"
                       format="soap11"/>
          </endpoint>
      </call>
      <property name="messageType"
                value="application/json"
                scope="axis2"
                type="STRING"/>
</sequence>
 [1] Message switch source JSONPath expression, which will be evaluated
against the incoming message and the resulting string will be used to match
with cases.
[2] matching case for WSO2 company name with the message mediation logic.
[3] matching case for XYZ company name with the message mediation logic.
[4] default message path for messages that doesn't match with above cases.
[5] the invocation of the web service backend service is delegated to a
sequence and sequence is called from case conditions of WSO2 and XYZ.
[6] PayloadFactory mediator is used to create the payload required to invoke
the backend service.  (We will further discuss about the PayloadFactory
mediator in the rest of this chapter)
```

As you have seen in Listing 4-6, the switch mediator can be used to implement a message switching logic for either SOAP or JSON messages. The generic syntax of the switch mediator is shown in Listing 4-7.

Listing 4-7. Switch Mediator Syntax

```
<switch source="[XPath|json-eval(JSON Path)]">
   <case regex="string">
     mediator+
   </case>+
   <default>
     mediator+
   </default>?
</switch>
```

You can evaluate the message using XPath or JSONPath and then do the case matching using regular expressions. Each case condition can have its own message processing logic (sequences) and you can configure a default message processing path if the message doesn't meet any of the specified `switch` case conditions.

■ **Note** Message filtering/switching can be implemented based on message attributes (such as message headers/transport headers) or message content. As explained at the beginning of the chapter, based on the message logic that you write, ESB will decide whether it needs to look at the message content or filter/switch the message.

Message Transformations

When you have two systems that are communicating about the same entity but with different message representations/formats, you need a message transformation layer in between the two systems. For example, assume that there is a financial organization that has a web service that provides the stock quote information for a given company name. The request `getQuote` message format that you need to invoke for the financial service is shown in Listing 4-8.

Listing 4-8. The getQuote Request Message Format

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ser="http://services.samples" xmlns:xsd="http://services.samples/xsd">
    <soapenv:Body>
        <ser:getQuote>
            <ser:request>
                <xsd:symbol>WSO2</xsd:symbol>
            </ser:request>
        </ser:getQuote>
    </soapenv:Body>
</soapenv:Envelope>
```

But the client who wants to consume that service is using a different message format and you won't be able to change, so that they use the same message format as the service. The client's `checkStockQuote` message format is shown in Listing 4-9.

Listing 4-9. The checkStockQuote Request Message Format

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
    <soapenv:Body>
        <checkStockQuote>
            <company>WSO2</company>
        </checkStockQuote>
    </soapenv:Body>
</soapenv:Envelope>
```

Therefore, in order for this particular web service client to consume the web service of the financial service, there needs to be a message format transformation from `checkStockQuote` to `getStockQuote` message formats. As depicted in Figure 4-5, WSO2 ESB can be used to serve this requirement and you can implement a message transformer inside the message processing logic.

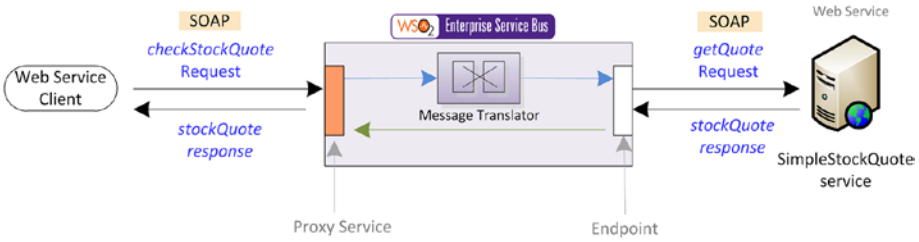

Figure 4-5. *Message transformer/translators in WSO2 ESB can be used to convert the incoming checkStockQuote request to getQuote request*

There are several ways to use message transformation/translation in WSO2 ESB. These are the most commonly used message transformation techniques and when you should select the specific message translation technique.

- *Using payload factory mediator:* One-to-one mapping between two disparate messages.
- *Using a combination of For-Each and the payload factory mediator:* Transformation requires to have an iteration of a simple message transformation logic.
- *Using XSLT mediator:* No one-to-one mapping and message transformation requires complex data mapping and computations.
- Using an enrich mediator to enrich a given message with some external message content.
- *Using a header mediator:* Adds and removes transport headers and SOAP message headers.

Let's try to dive deep into each of the message transforming techniques and find out how to pick the most appropriate one for your needs.

Using PayloadFactory Mediator

The message transformation scenario that we discussed in Figure 4-4 requires us to transform the message `checkStockQuote` to the `getQuote` request. If you observe the message formats in Listing 4-8 and 4-9, you can simply transform the `checkStockQuote` to `getQuote` by changing the XML element's name of the `checkQuote` request. Therefore it is a one-to-one message translation scenario and, as shown in Figure 4-6, for such scenarios you can use `PayloadFactory` mediator of WSO2 ESB.

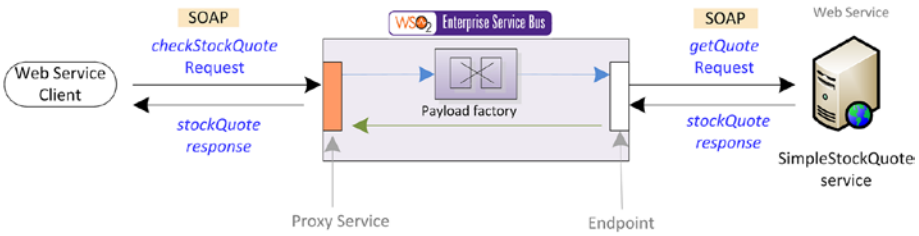

Figure 4-6. *One-to-one message transformations with payload to convert the incoming checkStockQuote request to a getQuote request*

In the `PayloadFactory` mediator, you can specify a message template of the `getQuote` message format inside a sequence and fill that template with the message attributes, such as company name for the `checkStockQuote` request. The related WSO2 ESB configuration is found in Listing 4-10.

Listing 4-10. One-to-One Message Transformation with the PayloadFactory Mediator

```
<proxy xmlns="http://ws.apache.org/ns/synapse"
       name="MessageTransformer_PF_Proxy_4.5"
       transports="https http"
       startOnLoad="true"
       trace="disable">
    <description/>
    <target>
        <inSequence>
            <payloadFactory media-type="xml">         <!-- [1] -->
                <format>                              <!-- [2] -->
                    <soapenv:Envelope
                           xmlns:soapenv="http://schemas.xmlsoap.org/soap/
                           envelope/"
                           xmlns:xsd="http://services.samples/xsd"
                           xmlns:ser="http://services.samples">
                        <soapenv:Body>
                            <ser:getQuote>
                                <ser:request>
                                    <xsd:symbol>$1</xsd:symbol> <!-- [3] -->
                                </ser:request>
```

```
                        </ser:getQuote>
                    </soapenv:Body>
                </soapenv:Envelope>
            </format>
            <args>                          <!-- [4] -->
                <arg
                        expression="//checkStockQuote/company"
                        evaluator="xml"/>          #E
            </args>
        </payloadFactory>
        <call>                          <!-- [5] -->
            <endpoint>
                <address
                        uri="http://127.0.0.1:9000/services/
                        SimpleStockQuoteService"/>
            </endpoint>
        </call>
        <respond/>                      <!-- [6] -->
    </inSequence>
</target>
</proxy>
[1] Specifying the media type of the result message template of the payload
factory as XML.
[2] You can provide the template/skeleton of the result message format.
[3] Specifying the arguments of the result message format.
[4] List of arguments of the values that should change dynamically.
[5] Populate the argument value dynamically from the incoming message of the
PayloadFactory mediator.
[6] Once the message is transformed, it is send to the backend service
[7] sending the response back to the client without doing any
transformation.
```

Therefore, the PayloadFactory mediator can be used to transform or replace the contents of a message. The general syntax of the PayloadFactory mediator is shown in Listing 4-11. The media-type of the message format can be either XML or JSON. Each argument in the mediator configuration can be a static value, or you can specify an XPath or JSONPath expression to get the value at runtime by evaluating the provided expression against the existing SOAP message. You can configure the format of the request or response and map it to the arguments provided.

Listing 4-11. PayloadFactory Mediator Syntax

```
<payloadFactory media-type="xml | json">
    <format ../>
    <args>
        <arg (value="string"|expression="{xpath}|{json_path}"
                        evaluator="xml |json")/>*
    </args>
</payloadFactory>
```

The PayloadFactory mediator is commonly used in many transformation scenarios where you just have one-to-one mapping between disparate messages.

Using PayloadFactory and For-Each Mediator

In some message transformation scenarios, a given message can have repetitive message elements and you need to transform each occurrence of that element. For example, assume that there is a web service that provides the weather forecast of a given location (specified by a ZIP/postal code). The response contains a detailed list of the weather forecast for seven days. Now you need to refine or transform this response so that you only send the summary of the weather forecast for each day. As shown in Figure 4-7, you can use the WSO2 ESB layer to implement this transformation use case.

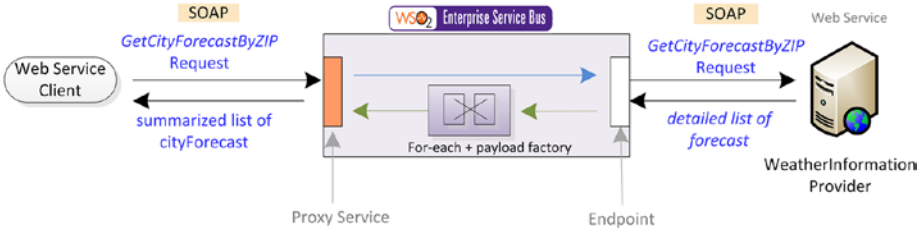

Figure 4-7. *The response from the WeatherDataProvider web service contains a repetitive list of detailed weather forecast. ESB needs to transform that response to a summarized list of weather forecasts. A combination of For-Each and the PayloadFactory mediator is used for that purpose.*

Listing 4-12 shows the sample response message format of the WeatherData provider web service for a GetCityForecastByZIP request.

Listing 4-12. Response from the WeatherData Web Service

```
<soap:Envelope
                  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <soap:Body>
      <GetCityForecastByZIPResponse xmlns="http://ws.cdyne.com/WeatherWS/">
         <GetCityForecastByZIPResult>
            <Success>true</Success>
            <ResponseText>City Found</ResponseText>
            <State>FL</State>
            <City>Lake Mary</City>
            <WeatherStationCity>Sanford</WeatherStationCity>
            <ForecastResult>
               <Forecast>
                  <Date>2014-08-18T00:00:00</Date>
```

```
                  <WeatherID>2</WeatherID>
                  <Description>Partly Cloudy</Desciption>
                  <Temperatures>
                     <MorningLow/>
                     <DaytimeHigh>96</DaytimeHigh>
                  </Temperatures>
                  <ProbabilityOfPrecipiation>
                     <Nighttime/>
                     <Daytime>30</Daytime>
                  </ProbabilityOfPrecipiation>
               </Forecast>
               <Forecast>
                  <Date>2015-06-18T00:00:00</Date>
                  <WeatherID>3</WeatherID>
                  <Description> Cloudy</Description>
                  <Temperatures>
                     <MorningLow/>
                     <DaytimeHigh>86</DaytimeHigh>
                  </Temperatures>
                  <ProbabilityOfPrecipiation>
                     <Nighttime/>
                     <Daytime>20</Daytime>
                  </ProbabilityOfPrecipiation>
               </Forecast>   <!-- [1] -->

               ...
            </ForecastResult>
         </GetCityForecastByZIPResult>
      </GetCityForecastByZIPResponse>
   </soap:Body>
</soap:Envelope>
[1] Forecast element repeats for multiple times to denote weekly forecast.
```

You need to transform this weather data response to the message format shown in Listing 4-13. There you can observe that the list of forecasts has to be transformed into `DailyForecast` and only a selected set of elements should be in the final response.

Listing 4-13. Response Message Format of the Summarized Weather Data

```
<soap:Envelope
                  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <soap:Body>
      <GetCityForecastByZIPResponse xmlns="http://ws.cdyne.com/WeatherWS/">
         <GetCityForecastByZIPResult>
            <Success>true</Success>
            <ResponseText>City Found</ResponseText>
```

```
            <State>FL</State>
            <City>Lake Mary</City>
            <WeatherStationCity>Sanford</WeatherStationCity>
            <ForecastResult>
               <DailyForecast>                              <!-- [1] -->
                  <Date>2014-08-18T00:00:00</Date>          <!-- [2] -->
                  <WeatherID>2</WeatherID>
                  <Desciption>Partly Cloudy</Desciption>
                           </DailyForecast>
               ...
            </ForecastResult>
         </GetCityForecastByZIPResult>
      </GetCityForecastByZIPResponse>
   </soap:Body>
</soap:Envelope>
[A] DailyForecast element repeats for multiple times.
[B] From each forecast element for a given day, only few elements are
selected and others ignored.
```

If you observe a repeating element of the backend service response (Listing 4-12) that has transformed into the summarized weather data response (Listing 4-13), the transformation between a given elements can be achieved with `PayloadFactory`. That logic needs to be repeated for the number of elements that you have in the original response from the backend service. For this purpose, you can use the `For-Each` mediator of WSO2 ESB.

The `For-Each` mediator is capable of executing a message flow repeatedly a specific number of times. You can define an XPath expression in the `For-Each` mediator configuration where it refers to the element that is repeating in the given message. Listing 4-14 shows the entire WSO2 ESB configuration for the implementation of the `WeatherData` integration scenario.

Listing 4-14. Transforming a Message with Repeated Elements with the For-Each and PayloadFactory Mediators

```
<proxy xmlns="http://ws.apache.org/ns/synapse"
       name="WeatherProxy"
       transports="https http"
       startOnLoad="true"
       trace="disable">
    <description/>
    <target>
        <inSequence>
            <call>                              <!-- [1] -->
                <endpoint>
                    <address uri="http://wsf.cdyne.com/WeatherWS/
                    Weather.asmx"/>
                </endpoint>
            </call>
```

```
            <log level="full"/>                <!-- [2] -->
            <foreach                           <!-- [3] -->
                                xmlns:wh="http://ws.cdyne.com/
                                WeatherWS/"
                expression="//wh:Forecast">    <!-- [4] -->
                <sequence>                          <!-- [5] -->
                    <payloadFactory media-type="xml">  <!-- [6] -->
                        <format>
                            <DailyForecast
                                                    xmlns="http://ws.cdyne.
                                                    com/WeatherWS/">
                                <Date>$1</Date>
                                <WeatherID>$2</WeatherID>
                                <Desciption>$3</Desciption>
                            </DailyForecast>
                        </format>
                        <args>
                            <arg evaluator="xml"
                                                    expression="//
                                                    wh:Date"/>
                            <arg evaluator="xml"
                                                    expression="//
                                                    wh:WeatherID"/>
                            <arg evaluator="xml"
                                                    expression="//
                                                    wh:Description"/>
                        </args>
                    </payloadFactory>
                </sequence>
            </foreach>
            <log level="full"/>                  <!-- [7] -->
            <respond/>                           <!-- [8] -->
        </inSequence>
    </target>
</proxy>
[1] Sending request to the backend service.
[2] Log the complete response received from the backend service
[3] ForEach mediator to iterate through the message for each element
specified by the for-each expression.
[4] XPath expression that refers to the repeating element of the  message.
[5] During each foreach iteration the specified sequence implementation is
invoked.
[6] PayloadFactory is doing the message transformation for each occurrence
of the repeating element.
[7] Logging the transformed response.
[8] Responding the backend service.
```

As you can observe in Listing 4-14, the primary purpose of the `For-Each` mediator is to iterate through a list of elements and the actual transformation is one for each portion of the message using a `PayloadFactory` mediator. The general syntax of the `For-Each` mediator is shown in Listing 4-15.

Listing 4-15. For-Each Mediator Syntax

```
<foreach                                        expression="{xpath}">
    <sequence>
                  ...
        </sequence>
</foreach>
```

Therefore, the `For-Each` mediator can be used for any integration use case where you need to use a `for-each` loop in the message flow.

Data Mapper Mediator

One of the most common requirements of ESB integration scenarios is to transform messages from one format to the other. This can be done using various technologies but rapid development of enterprise integration scenarios require a visual modeling tool that allows you to graphically map the different data formats. WSO2 ESB provides a data mapper (as a mediator) that you can use inside the ESB graphical editor. With the data mapper, you can provide the input message format and expected output message format and do the visual data mapping between the two message formats. The details of how to use the data mapper is covered in Chapter 10.

Using XSLT Mediator

In some of the integration scenarios, the message transformation logic is not one-to-one or cannot be implemented with simple `For-Each` and payload factory combinations. For example, consider a financial service provider that provides the stock quote information of a company for the last year or so. As illustrated in Figure 4-8, the `fullQuote` response contains all the data. But the client only needs the stock quote information from the last 10 days with a different response message format (`latestQuote`). You can use the XSLT mediator to implement this in WSO2 ESB. To transform the `getLatestQuote` request to `getFullQuoteRequest`, you have to use the `PayloadFacotry` mediator, as the transformation is a simple one-to-one transformation.

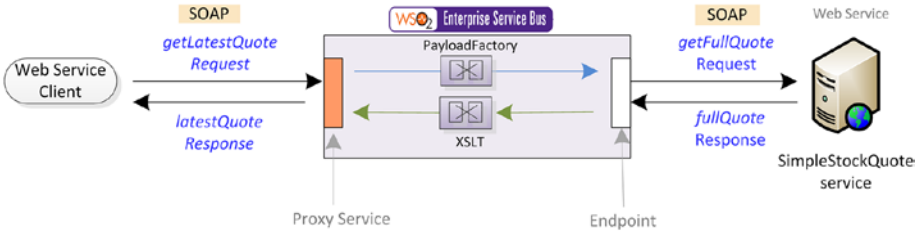

Figure 4-8. *WSO2 ESB uses an XSLT transformation to identify the most recent (last 10 days) stock quotes and transform the message so that it only contains the recent stock quote information*

The main task of this integration scenario is to implement the message transformation logic using XSLT. XSLT is a message transformation language created for XML and there are numerous functionalities available for doing complex message transformations. Discussing the details of XSLT is not in the scope of this book. Listing 4-16 shows an XSLT that transforms the `fullQuote` response as per the aforementioned requirements.

Listing 4-16. XSLT for Transforming the getFullQuote Message to the latestQuote Message

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:ns="http://services.samples"
                xmlns:ax21="http://services.samples/xsd"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                exclude-result-prefixes="ns ax21 xsi">
    <xsl:output method="xml" omit-xml-declaration="yes" indent="yes"/>

    <xsl:template match="/">                    <!-- [1] -->
        <xsl:apply-templates select="//ns:getFullQuoteResponse/ns:return"/>
    </xsl:template>

    <xsl:template match="ns:return">            <!-- [2] -->
        <latestQuote>
            <xsl:for-each select="ax21:tradeHistory">    <!-- [3] -->
                <xsl:choose>
                    <xsl:when test="ax21:day &gt; 354">  <!-- [4] -->
                        <tradeInfo>
                            <day><xsl:value-of select="ax21:day"/></day>#E
                            <quote>
                                <companyName><xsl:value-of
                                select="ax21:quote/ax21:name"/>
                                </companyName>            <!-- [5] -->
                                <earnings><xsl:value-of
                                select="ax21:quote/ax21:earnings"/></
                                earnings>
```

```
                        </quote>
                    </tradeInfo>
                </xsl:when>
                <xsl:otherwise>
                </xsl:otherwise>
            </xsl:choose>
        </xsl:for-each>
    </latestQuote >
</xsl:template>
</xsl:stylesheet>
[1] Template that matches the whole document/payload.
[2] Template that matches the ns:return element of the payload.
[3] Apply the transformation for each tradeHistory element
[4]Condition to select the latest stock quote values. (last 10 days)
[5] Select values from the original payload when we create the new payload.
```

You store this XSLT in ESB as a resource by adding it to the configuration as a local entry. Then, as shown in Listing 4-17, you can simply refer the XSLT in the response path of the message.

Listing 4-17. Complex Message Transformations with the XSLT Mediator

```
<proxy xmlns="http://ws.apache.org/ns/synapse"
       name="MessageTransformer_PF_Proxy_4.5"
       transports="https http"
       startOnLoad="true"
       trace="disable">
    <description/>
    <target>
        <inSequence>
            <call>
                <endpoint>
        <address uri="http://127.0.0.1:9000/services/
SimpleStockQuoteService"/>
                </endpoint>
            </call>
            <xslt key="latestQuoteSelector"/>          <!-- [1] -->
            <respond/>
        </inSequence>
    </target>
</proxy>
[1] XSLT mediator is used in the response path and it uses the above
latestQuoteSelector XSLT configuration which is stored as a local entry.
```

The general syntax of the XSLT mediator configuration is shown in Listing 4-18. In addition to the functionalities that you have seen, you can pass properties to the XSLT transformation so that those property values will be used in message transformation.

Listing 4-18. XSLT Mediator Syntax

```
<xslt key="string" [source="xpath"]>
     <property name="string" (value="literal" | expression="xpath")/>*
     <feature name="string" value="true| false" />*
     <resource location="string" key="string"/>*
</xslt>
```

With XSLT, you can implement almost any complex message transformation logic.

Using the Header Mediator

When transforming messages, sometimes you need to manipulate message headers (add, remove, or modify) without touching the message content. For example, in the financial service scenario, assume that the backend service requires consumers to send a custom HTTP header (X-STOCK-EX-API-APP-ID) to be present in every request to serve them successfully. But the web service client is not aware of that and, as illustrated in Figure 4-9, the ESB needs to take care of adding the missing HTTP header to the message.

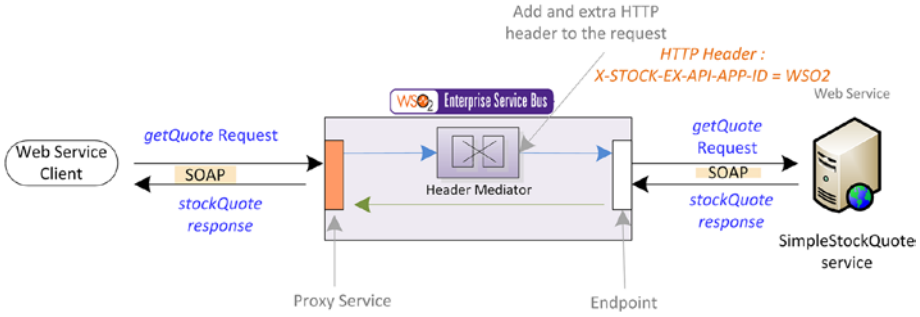

Figure 4-9. *Header mediator is used to add the missing custom HTTP headers into the message*

Listing 4-19 shows a simple example where we add a custom HTTP header using the header mediator. You can use the header mediator to specify the name of the header and its value. As we want to set this as a transport header, you need to specify the scope as transport. Then place it prior to sending the message out using the call mediator. The message that goes out to the specified backend service in the call mediator endpoint will contain the specified transport header.

Listing 4-19. Header Mediator Can Be Used to Add a HTTP Header into the Outgoing Message

```
<header name="X-STOCK-EX-API-APP-ID"
value="WSO2" scope="transport"/>          <!-- [1] -->
<call>
               <endpoint>
```

```
        <address uri="http://127.0.0.1:9000/services/
        SimpleStockQuoteService"/>
                </endpoint>
</call>
[1] By using header mediator you can specify the name of the header as
“X-STOCK-EX-API-APP-ID” and its value as “WSO2”.
```

The header mediator in WSO2 ESB can be used to add or remove message headers. You can add SOAP headers and transport header (HTTP) using the header mediator. Also if you want to remove any existing message headers, you can do so by setting the action to `remove`. The syntax of the header mediator is shown in Listing 4-19.

Listing 4-20. Header Mediator Syntax

```
<header name="string"
                (value="string|{property}" |
                expression="xpath") [scope=default|transport]
[action=set|remove]/>
```

The default scope allows you to set the headers as SOAP message headers while transport scope allows you to set transport headers using the header mediator.

Message Enriching

When implementing integration scenarios, there are instances where you need to enrich a message with some additional data. For example, assume that there is a financial service that allows you to place stock orders (`placeOrder` requests). The client is sending a `placeDefaultOrder` request with only the price and the symbol, but the quantity of the stock quote is missing, as it's the default order. The value required for the default value of the `placeOrder` request is stored in ESB configuration and, as shown in Figure 4-10, the ESB layer can add that missing quantity element with the default quantity for the order. Then the ESB can send the `placeOrder` request to the backend service. This scenario is similar to a message transformation scenario, but we only modify a selected portion of the message. Therefore, this is considered a special case of message transformation, which is known as message enriching.

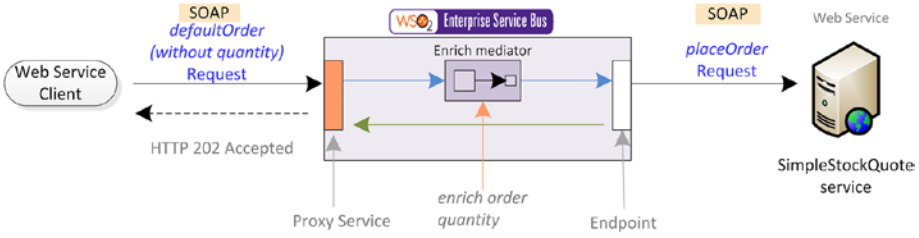

Figure 4-10. *Enrich mediator can be used to enrich the defaultOrder request with the missing order quantity and to create the placeOrder request. The value required to create the default order is stored in ESB and it is added to the message during the message enrichment.*

Listing 4-20 shows the complete configuration of the integration scenario that we discussed in Figure 4-9. You can use the enrich mediator to specify the element that you are going to use for enriching. This is specified under the `source` element in the enrich mediator configuration. As the source, you can specify any inline content or you can select it from a local or registry entry. The `target` element is used to specify the place that the enriching source element should be added.

Listing 4-20. Enriching defaultOrder with Missing Quantity Element Using the Enrich Mediator

```
<proxy xmlns="http://ws.apache.org/ns/synapse"
       name="MessageEnrichProxy_4.9"
       transports="https http"
       startOnLoad="true"
       trace="disable">
    <description/>
    <target>
        <inSequence>
            <log level="full"/>
            <enrich>
                <source type="inline">                      <!-- [1] -->
                         <xsd:quantity
                                          xmlns:xsd="http://services.
                                          samples/xsd">19
                                 </xsd:quantity>
                </source>
                <target action="child"                <!-- [2] -->
                                 xmlns:ser="http://services.samples"
                        xmlns:xsd="http://services.samples/xsd"
                          xpath="//ser:placeOrder/ser:order"/>  <!-- [3] -->
            </enrich>

<property name="FORCE_SC_ACCEPTED" value="true" scope="axis2"/>
           <property name="OUT_ONLY" value="true"/>
           <call>
               <endpoint>
                   <address uri="jms:/StockQuotesQueue?transport.jms.Con
                   nectionFactoryJNDIName=QueueConnectionFactory&amp;ja
                   va.naming.factory.initial=org.wso2.andes.jndi.Properti
                   esFileInitialContextFactory&amp;java.naming.provider.
                   url=repository/conf/jndi.properties&amp;transport.jms.
                   DestinationType=queue"/>
               </endpoint>
           </call>

            <log level="full"/>
            <call>
                <endpoint>
```

```
                    <address
                           uri="http://127.0.0.1:9000/services/
                           SimpleStockQuoteService"/>
                </endpoint>
            </call>
            <respond/>
        </inSequence>
    </target>
</proxy>
[1] Specify the element that you want to enrich into the message as the
source of enrich mediator.
[2] Configure target as the child of the target XPath
[3] Specify the XPath expression to the location that you want to enrich the
new element.
```

The general syntax of the enrich mediator is shown in Listing 4-21. The primary purpose of the enrich mediator is to enrich (add content or modify/replace the existing content) the inflight message with the content specified in the source. In addition to enriching content into the message, you can use the enrich mediator to extract content from the message and store it as in-memory message properties for use in the downstream message flow. Therefore, the enrich mediator can be used to preserve a given message as a property at some point in the message flow and later refer that message again (using envelope or body attributes).

Listing 4-21. Enrich Mediator Syntax

```
<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>
```

For integration scenarios where you only need to modify a given part or add a new content to the message, you can choose enrich mediator over any transformation mediators.

Message Validation

The validation of the messages against a given message format/schema is a common integration requirement. For example, assume that we have a financial service that provides stock quote information and the information that we got for a particular stock quote request needs to be verified against a predefined schema. As shown in Figure 4-11, in the response message path of the ESB, you can use a validate mediator to validate the message against a given schema. Since the backend service sends a SOAP message as the

stockQuote response message, you can specify an XML schema (http://www.w3.org/XML/Schema) to validate the incoming response message format.

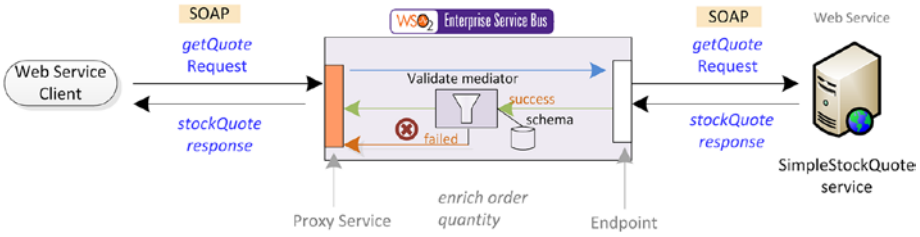

Figure 4-11. *Validating the getQuoteResponse message against the XML schema*

Listing 4-22 shows the complete configuration of the proxy service for the message validation integration scenario. When the message validation fails, there is an on-fail condition where you can specify invalid message handling logic.

Listing 4-22. Using the Validate Mediator to Schema Validation of the getQuoteResponse

```
<proxy name="StockQuoteProxy" startOnLoad="true">
      <target>
         <inSequence>
            <log level="full"/>
            <call>
               <endpoint key="GetQuote"/>
            </call>
            <log level="full"/>
            <validate>                                          <!-- [1] -->
               <schema key="response-validate"/>
               <on-fail>
                  <makefault version="soap11">
                     <code
                                 xmlns:tns="http://www.w3.org/2003/05/
                                 soap-envelope"
                                 value="tns:Receiver"/>
                     <reason
                                 value="Invalid Response!!!. Validation
                                 failed..."/>
                  </makefault>
               </on-fail>
            </validate>
            <respond/>
         </inSequence>
      </target>
   </proxy>
```

```
   <localEntry key="response-validate">            <!-- [2] -->
      <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
                 elementFormDefault="qualified"
                 attributeFormDefault="unqualified">
         <xs:element name="getQuoteResponse">
            <xs:complexType>
               <xs:sequence>
                  <xs:element name="return">
                     <xs:complexType>
                        <xs:sequence>
                           <xs:element name="name" type="xs:string"/>
                           <xs:element name="currentPrice"
                           type="xs:double"/>
                        </xs:sequence>
                     </xs:complexType>
                  </xs:element>
               </xs:sequence>
            </xs:complexType>
         </xs:element>
      </xs:schema>
   </localEntry>
   <endpoint name="GetQuote">
      <address uri="http://localhost:9500/simple/stock/quote/service"/>
   </endpoint>
[1] Call validate mediator to do schema validation of the response message.
[2] The local entry that contains the XML schema.
```

The general syntax of the validate mediator is shown in Listing 4-23. The primary use of validate mediator is to do schema validation of messages. The validate mediator validates the result of the evaluation of the source XPath expression against the schema specified. If the source attribute is not specified, the validation is performed against the first child of the SOAP body of the current message. If the validation fails, the on-fail sequence of mediators is executed. Properties could be used to turn some of the features of the validator on/off.

Listing 4-23. Validate Mediator Syntax

```
<validate [source="xpath"]>
                     <property name="validation-feature-id"
value="true|false"/>*
                     <schema key="string"/>+
                     <on-fail>
                        mediator+
                     </on-fail>
         </validate>
```

The usage of the validate mediator is limited to the validation of SOAP/XML messages.

Service Orchestration

When you are building an IT solution out of the existing systems/services, you need to use the business functionalities offered by each service/system. Therefore, at the ESB layer, you need to implement a message processing logic that can call and combine the results of multiple services/systems and expose that as an aggregated service. That is known as service orchestration. It's defined as the coordination and arrangement of multiple services/systems exposed as a single aggregate service.

There are multiple service orchestration patterns used in the enterprise integration space.

- *Service chaining pattern*: Expose a service that is calling multiple services one after the other and sending back the final response.
- *Split and aggregate pattern*: Expose a service that splits the incoming messages into multiple subrequests and sends them to a backend service in parallel. Then aggregate the responses for each request and send them back to the client as an aggregated response.
- *Clone and aggregate pattern*: Expose a service that clones the incoming messages into multiple messages and sends them to multiple backend services in parallel. Then aggregate the responses for each request and send them back to the client as an aggregated response.

Service Chaining

Service chaining is about exposing a service that can, behind the scenes, call multiple backend services sequentially and facilitate the business requirements of the client. For example, assume that there is an ATM locator service that allows you to find the nearby ATMs of the specified bank for your current geolocation (longitude and latitude). As illustrated in Figure 4-12, the input that you are providing to the service is longitude, latitude, and the name of the bank that you are searching ATMs for. But, there is no such banking service that you could provide these details out of box. Instead, there are three different services that you can use to implement this business use case.

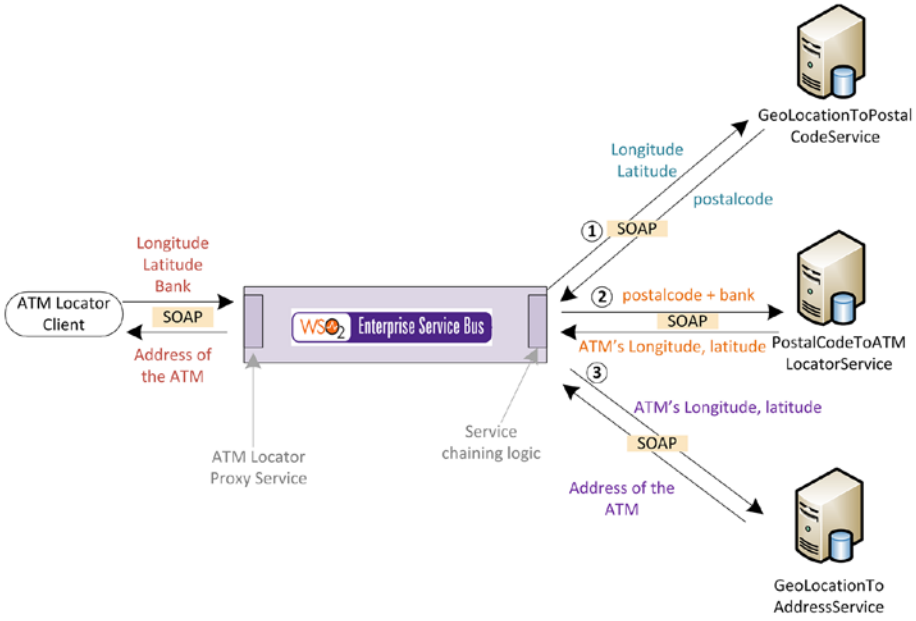

Figure 4-12. *WSO2 ESB exposes the ATM locator proxy service to the clients and that proxy service calls multiple backend services to find the address of a nearby ATM*

As illustrated in Figure 4-12, you need to:

- Invoke `GetLocationToPostalcode` service, with longitude and latitude to get the postal code of your geolocation.
- Once you have the postal code, you can invoke the `PostalcodeToATMLocator` service with postal code and bank name. As the response, you get the geolocation of the ATM that matches the given postal code and bank name (for simplicity, assume it's only returning one ATM).
- Then you can invoke the `GeoLocationToAddress` service with the longitude and latitude of the ATM to get its exact address.

Therefore you can discover that this is a service chaining scenario where you need to call services sequentially to implement the ATM locator service. When you are implementing this scenario with WSO2 ESB, you need to use some of the mediators that you learned in the previous sections of this chapter. For creating the request that needs to be sent to the backend services, we used the `PayloadFactory` mediator. `ATMLocatorService` is implemented as a proxy service and all the service chaining logic is carried out in the in-sequence of the proxy service. Call mediators and endpoints are used to send the requests to backend services. The complete configuration of the ATM locator service-chaining scenario is shown in the Listing 4-24.

Listing 4-24. Configuration of the ATMLocator Service-Chaining Scenario

```
<proxy name="ATMLocatorService"
       transports="https http"
       startOnLoad="true"
       trace="disable">
    <target>
        <inSequence>
            <property xmlns:ser="http://www.wso2.esb.sample"
                      name="Bank"
                      expression="//ser:getNearestATM/ser:bank"
                      scope="default"
                      type="STRING"/>                          <!-- [1] -->
            <property xmlns:ser="http://www.wso2.esb.sample"
                      name="Latitude"
                      expression="//ser:getNearestATM/ser:myLocation/
                      ser:lat"
                      scope="default"
                      type="STRING"/>
            <property xmlns:ser="http://www.wso2.esb.sample"
                      name="Longitude"
                      expression="//ser:getNearestATM/ser:myLocation/
                      ser:lon"
                      scope="default"
                      type="STRING"/>

            <payloadFactory media-type="xml" description="GeoLocation to
            PostalCode Service">                                <!-- [2] -->
                <format>
                    <ser:getPostalCode xmlns:ser=
                    "http://service.esb.wso2.com">
                        <ser:lat>$1</ser:lat>
                        <ser:lon>$2</ser:lon>
                    </ser:getPostalCode>
                </format>
                <args>
                    <arg evaluator="xml" expression="$ctx:Latitude"/>
                    <arg evaluator="xml" expression="$ctx:Longitude"/>
                </args>
            </payloadFactory>
            <header name="Action" value="urn:getPostalCode"/>  <!-- [3] -->
            <call>                                             <!-- [4] -->
                <endpoint key="GetPostalCode-EP"/>
            </call>
            <payloadFactory media-type="xml" description="PostalCode to
            BankATM Service">                  <!-- [5] -->
                <format>
                    <ser:getATMLocations xmlns:ser=
                    "http://service.esb.wso2.com">
```

```
                    <ser:areaCode>$1</ser:areaCode>
                    <ser:customerLocation>$2</ser:customerLocation>
                    <ser:Bank>$3</ser:Bank>
                </ser:getATMLocations>
            </format>
            <args>
                <arg xmlns:ser="http://service.esb.wso2.com"
                     evaluator="xml"
                     expression="//ser:code"/>
                <arg evaluator="xml"
                     expression="fn:concat($ctx:Latitude,',',
                     $ctx:Longitude)"/>
                <arg evaluator="xml" expression="$ctx:Bank"/>
            </args>
        </payloadFactory>
        <header name="Action" value="urn:getATMLocations"/>
        <call>                                  <!-- [6] -->
            <endpoint key="GetBankATM-EP"/>
        </call>
        <payloadFactory media-type="xml" description="PostalCode to
        BankATM Service">                    <!-- [7] -->
            <format>
                <ser:getAddressFromGeoLocation xmlns:ser="http://
                service.esb.wso2.com">
                    <ser:location>$1</ser:location>
                </ser:getAddressFromGeoLocation>
            </format>
            <args>
                <arg xmlns:ser="http://service.esb.wso2.com"
                     evaluator="xml"
                     expression="//ser:coordinates"/>
            </args>
        </payloadFactory>
        <header name="Action" value="urn:getAddressFromGeoLocation"/>
        <call>                    <!-- [8] -->
            <endpoint key="GetAddress-EP"/>
        </call>
        <respond/>                <!-- [9] -->
    </inSequence>
  </target>
</proxy>
[1] Extract the required values from the incoming message and save them into
properties.
[2] Create payload that need to be sent to GeoLocationToPostalCode service
[3] Set the required SOAPAction header using Header mediator.
[4] Sending request to first service.
```

```
[5] Extract the required parameters from the GeoLocationToPostalCode's
response and create the payload to be sent to PostalCodeToATMLocator
service.
[6] Sending request to second service.
[7] Create payload to be sent to third service.
[8] Sending request to third service.
[9]  Sending back the final response to the client.
```

As you can see in Listing 4-24, the primary constructs of building a service-chaining scenario are the message transformation mediators, call/response mediators, and other utility mediators such as property mediators (which will be discussed in detail in the next section). Using this single sequence model (unlike using send mediator with in-sequence and out-sequence), you can implement relatively complex service chaining scenarios with ease.

Split and Aggregate Pattern

There are service orchestration scenarios where you need to split the incoming message into multiple parts (for example, based on repeating elements) and send them to one or more backend services in parallel. Then you have to aggregate the response from all those service and process the aggregated response. Let's consider another variation of the financial service that we discussed before. Assume that there is a financial service that allows you to send a stock quote request for more than one company. That means you can send a list of `getQuote` elements inside a single request (say `getMultipleQuote` request) and the ESB layer has to split them into `getQuote` requests and send them in parallel to the specified backend service/s. As shown in Figure 4-13, for splitting the messages you can use the iterate mediator. Once the ESB sends the request to the backend service, it keeps on receiving responses. You can use the aggregate mediator to aggregate all those responses and send the aggregated response back to the client.

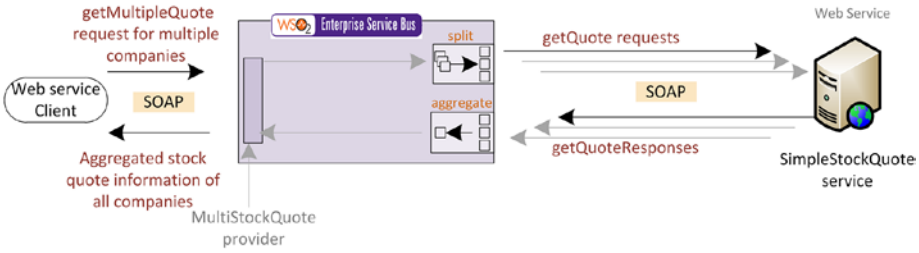

Figure 4-13. *WSO2 ESB receives a multipleQuoteRequest with repeating getQuote requests. ESB splits the multipleQuoteRequest into getQuote requests using the iterate mediator and sends them in parallel to the backend service. Once the responses for every request has arrived, the aggregated response (using aggregate mediator) is sent back to the client.*

As shown in Listing 4-25, `MultiStockquoteProvider` proxy accepts the `getMultipleQuote` request and the iterate mediator splits the messages into multiple messages and sends them in parallel to the `Stockquote` backend service. The aggregated mediator used after the iterate mediator waits until it get responses for all the requests that it sent out (with the default configuration, if the iterate mediator splits five messages, then aggregate will wait for five responses).

Listing 4-25. Message Splitting and Aggregation with Iterate and Aggregate Mediators

```
<proxy name="StockQuoteProxy" startOnLoad="true">
      <target>
         <inSequence>
            <iterate xmlns:m0="http://services.samples"
                     preservePayload="true"
                            expression="//m0:getQuote/m0:request"
                            <!-- [1] -->
                     attachPath="//m0:getQuote">            <!-- [2] -->
               <target>
                  <sequence>                                <!-- [3] -->
<call>
                              <endpoint>
                                 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
                              </endpoint>
                           </call>

                  </sequence>
               </target>
            </iterate>

<aggregate>                           <!-- [4] -->
  <completeCondition>
    <messageCount min="3" max="-1"/>
  </completeCondition>
  <onComplete
xmlns:ns="http://services.samples"
           expression="//ns:getQuoteResponse">    <!-- [5] -->
            <respond/>                            <!-- [6] -->
  </onComplete>
</aggregate>
         </inSequence>
</proxy>
[1] XPath expression to split the message in to multiple chunks.
[2] specify to which part of the original message that the spilt message
should be attached.
[3] This sequence is called for each iteration/split message
[4] Aggregate mediator waits till all the responses get aggregated.
```

```
[5] The messages that match the specified expression are only considered for
aggregation.
[6] Once the aggregation is completed, the aggregated response is sent back
to the client.
```

The general syntax of the iterate mediator is shown in Listing 4-26. The iterate mediator splits the message into a number of different messages derived from the parent message by finding matching elements for the XPath expression specified. New messages are created for every matching element and processed in parallel (default behavior) using either the specified sequence or endpoint. Created messages can also be processed sequentially by setting the optional sequential attribute to true. The original message can be continued or dropped depending on the Boolean value of the optional `continueParent` attribute.

Listing 4-26. Iterate Mediator Syntax

```
<iterate expression="xpath" (attachPath="xpath")?
                  [sequential=(true | false)]
                  [continueParent=(true | false)]
                  [id="string"]
                  [preservePayload=(true | false)] >
   <target>
     <sequence>
       (mediator)+
     </sequence>?
   </target>+
</iterate>
```

The `preservePayload` attribute specifies if the original message should be used as a template when creating the split messages and defaults to false, in which case the split messages would contain the split elements as the SOAP body. The optional ID attribute can be used to identify the iterator that created a particular split message when nested iterate mediators are used. This is particularly useful when aggregating responses of messages that are created using nested iterate mediators.

The general syntax of the aggregate mediator is shown in Listing 4-27. The aggregate mediator aggregates the response messages for messages that were split by the iterate mediator (or clone) and sent using the call (or send) mediator.

Listing 4-27. Aggregate Mediator Syntax

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

The aggregation expression specifies which elements should be aggregated. You can also define a `completeCondition` for determining the completion of aggregation. If no such condition is provided, the aggregation will be completed once the responses for all the split request (or cloned) are received.

The ID optional attribute can be used to aggregate only responses for split messages that are created by a specific clone/iterate mediator. Aggregate ID should be the same as the ID of the corresponding clone/iterate mediator that creates split messages. This is particularly useful when aggregating responses for messages that are created using nested clone/iterate mediators.

Clone and Aggregate Pattern

Unlike in the iterate-aggregate pattern, in the clone-aggregate pattern, the same request is cloned into multiple messages and sent to one or more endpoints. You can use an aggregate mediator to aggregate the responses for all the cloned requests that you sent. By default, the cloning of messages is executed in parallel but you can control the behavior using the `sequential=true|false` attribute. For example, consider a travel information provider (`find.airfare.com`) that provides its clients information about the airfare between two locations on a specified date. As shown in Figure 4-14, `find.airfare.com` needs to get the travel information from multiple services hosted by three different airlines. Therefore, when you get an `getAirfare` request, that request should be cloned and sent to multiple endpoints. Later you can aggregate the response received from these endpoints into one message.

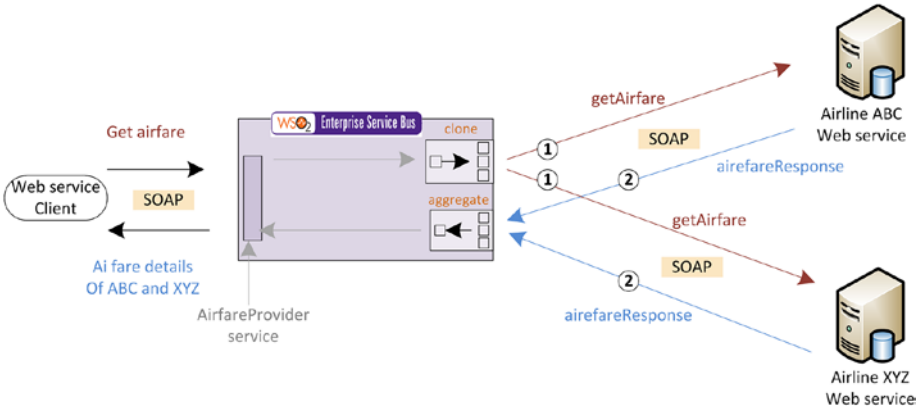

Figure 4-14. *WSO2 ESB clones the get airfare requests and sends them to different airlines. Once all the responses from airline web services are received, it aggregates all the responses and sends back the data to the web service client.*

In Listing 4-28, you can find the entire message flow configuration for this integration scenario. There are three clone mediators used for sending the cloned requests to the backend services. The clone targets are completely independent of each other and executed in parallel with the default configuration. Therefore, once you clone the message you can do any modification to the message inside the clone target.

Listing 4-28. Cloning Messages and Aggregating Responses

```
<proxy name="AirTicktingQuoteProxy" startOnLoad="true">
      <target>
         <inSequence>
            <log level="full"/>
            <clone>                                    <!-- [1] -->
               <target>                                <!-- [2] -->
                  <sequence>
                     <call>
                        <endpoint key="SriLankanAirLines"/>
                     </call>
                  </sequence>
               </target>
               <target>
                  <sequence>
                     <call>
                        <endpoint key="QatarAirways"/>
                     </call>
                  </sequence>
               </target>
               <target>
                  <sequence>
                     <call>
                        <endpoint key="EtihadAirways"/>
                     </call>
                  </sequence>
               </target>
            </clone>
            <aggregate>                          <!-- [3] -->
               <completeCondition>
                  <messageCount min="3" max="-1"/>
               </completeCondition>
               <onComplete xmlns:esb=http://esb.wso2.com expression=
               "//esb:getPriceResponse">
                  <log level="full"/>
                  <respond/>
               </onComplete>
            </aggregate>
         </inSequence>
      </target>
   </proxy>
   <endpoint name="QatarAirways">
      <address uri="http://localhost:9600/QatarAirways/ticketing/service"/>
   </endpoint>
   <endpoint name="EtihadAirways">
      <address uri="http://localhost:9700/EtihadAirways/ticketing/service"/>
   </endpoint>
```

```
   <endpoint name="SriLankanAirLines">
      <address uri="http://localhost:9500/SriLankanAirLines/ticketing/
service"/>
   </endpoint>
[1] Using clone mediator to clone the message flow into identical messages.
[2] Each clone target is executed in parallel and independent threads and
clone messages can be processing using all the generic mediation techniques.
[3] one the responses for all the cloned requests are received, the final
response is aggregated and sent back to the client.
```

The general syntax of the clone mediator is shown in Listing 4-29. The clone mediator can be used to clone a message into several messages. The original cloned message can be continued or dropped depending on the Boolean value of the optional `continueParent` attribute.

Listing 4-29. Clone Mediator Syntax

```
<clone          [continueParent=(true | false)]
[id="string"]
[sequential="true | false"]>
   <target [sequence="sequence_ref"]>
     <sequence>
       (mediator)+
     </sequence>?
   </target>+
</clone>
```

As with the iterate mediator, you can specify an ID for the clone mediator and correlate that with an aggregate mediator.

Changing the Message Protocol and Format

When building real-world IT solutions the integration between disparate technologies and protocols is vital. The messaging protocols such as HTTP, JMS, File, and so on are widely used in enterprise IT and if you have to integrate such protocols, you can use WSO2 ESB. Similarly, conversion between disparate message formats such as JSON and SOAP is also quite common in modern enterprises.

Protocol Conversions

Protocol conversion is extensively used in modern enterprise integration use cases. For example, consider a shipping order processing system in which the shipping orders are created in the form of files and they are added to a FTP server. Now you want to fetch these shipping orders, process them, and persist them in a JMS queue to be processed from another system. Figure 4-15 illustrates the implementation of the protocol conversion from File/FTP to JMS with the use of WSO2 ESB.

Figure 4-15. *Fetching the shipping orders stored in the FTP by using a file inbound endpoint, processing them, and then adding them to message queue via JMS protocol*

Listing 4-30 shows the WSO2 ESB configuration for the realization of the protocol conversion scenario between FTP and JMS.

Listing 4-30. Processing Orders Stored in Files and Persist Them in a JMS Broker

```
<inboundEndpoint xmlns=http://ws.apache.org/ns/synapse
name="ShippingOrderProcessor"                <!-- [1] -->
sequence="orderProcessingSeq"                <!-- [2] -->
onError="fault"
protocol="file" suspend="false">
   <parameters>
      <parameter name="interval">1000</parameter>
      <parameter name="sequential">true</parameter>
      <parameter name="coordination">true</parameter>
      <parameter name="transport.vfs.ActionAfterProcess">MOVE</parameter>
      <parameter name="transport.PollInterval">10</parameter>
      <parameter
name="transport.vfs.MoveAfterProcess">file:///home/user/test/out
 </parameter>
      <parameter name="transport.vfs.FileURI">file:///home/user/test/in
      </parameter>
      <parameter name="transport.vfs.MoveAfterFailure">file:///home/user/
      test/failed</parameter>
      <parameter name="transport.vfs.FileNamePattern">.*.txt</parameter>
      <parameter name="transport.vfs.ContentType">text/plain</parameter>
      <parameter name="transport.vfs.ActionAfterFailure">MOVE</parameter>
   </parameters>
</inboundEndpoint>

<sequence name="ShippingOrderProcessor">          <!-- [3] -->
<property name="OUT_ONLY" value="true"/>
     <call>                              <!-- [4] -->
         <endpoint>
```

```
            <address uri="jms:/StockQuotesQueue?transport.jms.Connecti
            onFactoryJNDIName=QueueConnectionFactory&amp;java.naming.
            factory.initial=org.wso2.andes.jndi.PropertiesFileInitialCont
            extFactory&amp;java.naming.provider.url=repository/conf/jndi.
            properties&amp;transport.jms.DestinationType=queue"/>
        </endpoint>
    </call>
</sequence>
[1] The file inbound endpoint polls the file system to get the available
files.
[2] The file content is processed and injected into the specified sequence.
[3] This sequence is invoked from the file inbound endpoint as there are new
files available in the file system
[4] Process the content of the file and add them to a JMS broker by sending
the messages to a JMS endpoint.
```

WSO2 ESB offers support for a wide range of protocols, including HTTP, JMS, FILE, TCP, SAP, and so on. The details of handling commonly used protocols will be discussed in the latter part of the book.

Message Format Conversions

Similar to handling disparate protocols, you need to integrate two or systems, which are using disparate, message formats. For example, assume that you need to integrate a banking system, which is using the POX (Plain Old XML) message format over HTTP protocol and a mobile client, which is using JSON over HTTP protocol. As shown in Figure 4-16, you can do one-to-one conversion from JSON to POX by specifying the endpoint format as POX.

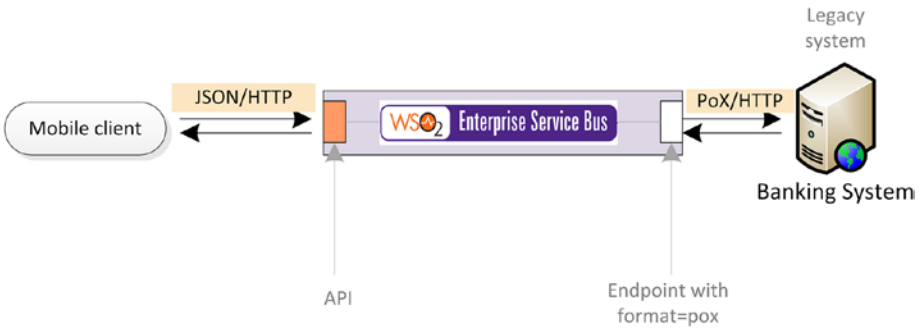

Figure 4-16. *The mobile client sends a JSON request and ESB takes care of transforming that request to the XML (PoX) message format and sending it to the legacy banking system*

Listing 4-31 shows the complete ESB configuration of the message format conversion from JSON to POX. Therefore, by specifying the endpoint format, you can do message format conversions of SOAP 1.1/1.2 and POX.

Listing 4-31. JSON to POX Message Conversion

```
<<api xmlns="http://ws.apache.org/ns/synapse"
     name="StockQuoteInfoProvider"
     context="/StockQuoteInfo">
    <resource methods="POST">
        <inSequence>
            <payloadFactory media-type="xml">        <!-- [1] -->
                <format>
<getSimpleQuote>
                            <symbol>$1</symbol>
                </ getSimpleQuote>

                </format>
                <args>
                    <arg evaluator="json" expression="$.getFinancialQuote.
                    company"/>
                </args>
            </payloadFactory>
                        <call>
                <endpoint>                    <!-- [2] -->
                    <address
uri="http://localhost:9000/services/SimpleStockQuoteService"
            format="pox"/>
                </endpoint>
            </call>
            <property name="messageType"
                      value="application/json"
                      scope="axis2"
                      type="STRING"/>
            <respond/>
        </inSequence>
    </resource>
</api>
[1] Using PayloadFactory to transform the message into the POX format
[2] specify at endpoint level that you need to send a POX message out from
the ESB.
```

But for some message format conversions, you need to set the flag `messageType` by specifying a property just before sending the message out. As shown in Listing 4-32, you can change the outgoing message format (i.e., the HTTP Content-Type) using the `messageType` property.

Listing 4-32. Using the messageType Flag to Convert the Message to a Different Message Format

```
<payloadFactory media-type="json">
      <format>{ "pizza": {    "name": "$1",    "price": $2,    "topping":
$3}}</format>
      <args>
              <arg evaluator="xml" expression="//pizza/name"/>
              <arg evaluator="xml" expression="//pizza/price"/>
              <arg evaluator="json" expression="pizza.toppings.topping"/>
          </args>
</payloadFactory>
<property name="messageType" value="application/json" scope="axis2"/>
<!-- [1] -->
<call> ...
[1] Since messageType flag is set. The outbound message will get converted
to a JSON.
```

So, far we discussed one-to-one message format conversions. But in many use cases, the message format conversion is not a simple one-to-one conversion (which we can achieve using `messageType` flag or the endpoint format attribute). For complex message format conversion use cases that involve SOAP/XML/JSON, you can use the message transformation techniques such as PayloadFactory, XSLT, and Enrich, along with `messageType` flag. For other message format conversions, you need to first convert all the message formats into the canonical form (by default, WSO2 ESB converts all message types other than JSON into canonical SOAP message format) and then do the transformation followed by the `messageType` flag for the required message format. You will further learn about the complex message format transformations in the ESB use case chapter.

Using Properties in the Message Flow

When implementing ESB message flow configurations, you may have to preserve some attribute of the message in a variable to be reused in down-stream message processing components. For example, in the financial service scenario, you might want to extract the company names and save those values to be used in the latter part of the message flow. For this purpose, you can use properties in WSO2 ESB message flow. There are two main usages of properties in WSO2 ESB—set/retrieve variables in the message flow and use of predefined properties to control the message flow.

Set/Retrieve Variables in the Message Flow

You can set properties in the message that is currently flowing through the sequence using the property mediator. You can also specify the scope when you define a property and scope is used to process messages at different layers when sending messages out from ESB.

- `default`: All the user-defined properties reside in this scope. The properties defined in this scope are available throughout the entire message flow.
- `transport`: This is primarily used to retrieve/alter transport headers.
- `axis2`: There are predefined properties to control the message flow (discussed in the next section).
- `operation`: The properties defined in this scope are shared between the threads of clone or iterate mediators.
- `registry`: The properties are defined and persisted in the ESB's registry.
- `system`: Set Java system properties.

Listing 4-33 shows setting properties in the message flow.

Listing 4-33. Using the Property Mediator in the Mediation Flow

```
<property xmlns:ser="http://www.wso2.esb.sample"
          name="Bank"
          expression="//ser:getNearestATM/ser:bank"
          scope="default"
          type="STRING"
          action="set"/>
<log level="custom">
    <property name="symbol" expression="$ctx:Bank"
</log>

<property name="symbol" value="FOO"/>
..
<log level="custom">
    <property name="symbol" expression="$ctx:Bank"
</log>
```

Any property that you define in the default scope can be retrieved using `$ctx:<property_name>` prefix, in any place that you want to use an expression (in any mediator such as Filter, Switch, Enrich, PayloadFactory, or log). You can retrieve properties from different scopes with the following shortcuts:

- `$ctx`: Default scope
- `$axis2`: axis2 scope
- `$trp`: Transport scope
- `$operation`: Operation scope
- `$registry`: Registry scope
- `$system`: System scope

The property mediator can be used with the XPath or JSONPath expressions. When retrieving properties, you can also use the XPath extension `get-property('<scope>' 'propertyName')`, but it is recommended to use the aforementioned shortcuts for optimum performance.

Use Predefined Properties to Control Message Flow

You can use the predefined properties in WSO2 ESB to control the message flow. For example, as you have seen in the implementation of the one-way messaging use case, you can use `<property name="OUT_ONLY" value="true"/>` to indicate to the message flow that you are doing one-way messaging. Likewise, predefined message properties from various message scopes are used for message flow controlling. You can refer them in the property catalog of WSO2 ESB at http://docs.wso2.com/enterprise-service-bus/Generic+Properties.

Summary

In this chapter you learned:

- The implementation techniques of request-response and one-way message scenarios with WSO2 ESB.
- Message filtering (based on message attributes) can be done using using the filter and switch mediators.
- For transforming one message to another, you can primarily use `PayloadFactory`, `For-Each`/Payload Factory, or XSLT mediators.
- The `PayloadFacotry` mediator is suited for one-to-one message transformation scenario where there is no complex message structure such as repeating elements.
- If you want to repeatedly transform the message with one-to-one mapping between elements, you can use the `For-Each` mediator along with `PayloadFactory`.
- The XSLT mediator is more suited for complex message transformations where you need to do complex mapping and computations in the transformation logic.
- To enrich content into a message, which is currently processed by the ESB, you can use the enrich mediator.
- The validated mediator can be used to validate a given message or a part of it against an XML schema.
- Service orchestration support in WSO2 ESB can be categorized into three main areas—service chaining, split and aggregate, and clone and aggregate.

- In service chaining, a given set of services is called one after the other. The response attribute from one service may be used in the next service invocation.
- The call mediator based single sequence model can be easily used to implement complex service chaining use cases.
- Split-aggregate messaging pattern can be implemented with iterate and aggregate mediators where messages are split into multiple parts. Each split message is executed in an independent thread, which are processed in parallel.
- The aggregate mediator can be used to aggregate the responses that you get once you iterate and send messages out.
- The clone-aggregate pattern is similar to the split-aggregate, but in this case the same message is cloned into multiple messages. You process them independently and in parallel.
- In the protocol conversion, messages from a given wire protocol are converted into messages of a different protocol. The inbound and outbound endpoint (with transport senders).
- Message format conversion is supported by the use of the `messageType` property with other message-transforming mediators.
- The properties in WSO2 ESB can be used to store/preserve variables and to use the predefined message properties to control the message flow.

CHAPTER 5

Integrating SOAP and RESTful Web Services

The success of the modern software solutions rely on how easily a given functionality can be exposed to the rest of the software applications and to which extent they can reuse the already implemented functionalities from other systems when building new software solutions. Web Services is the most widely used technology, and it allows the software application to interact over the world wide web (WWW). Web Services are widely implemented using two commonly used approaches, namely SOAP (Simple Object Access Protocol) and REST (Representational State Transfer).

Understanding SOAP and RESTful Web Services

SOAP is a standard-based web service access protocol that defines a standard communication protocol specification for XML-based message exchange. The other related specifications such as WSDL (Web Services Description Language) allow you to define a SOAP-based web service with the functionalities (operations) offered from the services and the required message formats. For example, consider a banking software solution that offers a SOAP-based web service to manage bank accounts, namely `AccountManagementSOAPService` (see Figure 5-1).

K. Indrasiri, *Beginning WSO2 ESB*, DOI 10.1007/978-1-4842-2343-7_5

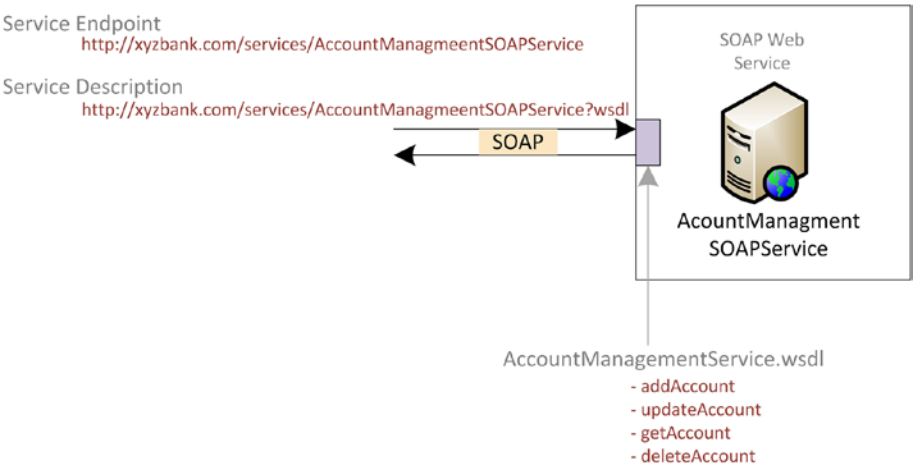

Figure 5-1. *The AccountManagementSOAPService exposes a SOAP web service over HTTP which allows the external parties to create, retrieve, modify, and delete accounts. Available operations, message formats, and service endpoint addresses are described in AccountmanagementService.wsdl.*

As depicted in Figure 5-1, this SOAP service offers four main functionalities—add account, update account, retrieve account, and delete account. These functionalities are implemented as operations of the `AccountManagementSOAPService` and they can be invoked with the respective SOAP message payload, which is sent to the web service endpoint URL. The sample message formats for those operations are listed in Listing 5-1. SOAP specification itself doesn't restrict the SOAP service to a single protocol such as HTTP, but in most cases, SOAP web services are exposed over HTTP. As per the SOAP 1.1 specification, with HTTP protocol, the `SOAPAction` HTTP request header field can be used to indicate the intent of the SOAP HTTP request.

Listing 5-1. Message Formats for Invoking Operations of AccountManagementSOAPService

```
<!-- Create Account -->
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:xyzbank="http://xyz.com">
    <soapenv:Body>
        <xyzbank:addAccount>
            <xyzbank:fullName>FF FDF</xyzbank:fullName>d
            <xyzbank:dob>22-10-1987</xyzbank:dob>
            <xyzbank:ssn>3223434</xyzbank:ssn>
            <xyzbank:initialAmount>USD 10</xyzbank:initialAmount>
        </xyzbank:addAccount>
    </soapenv:Body>
</soapenv:Envelope>
```

```
<!-- Read Account -->
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:xyzbank="http://xyz.com">
    <soapenv:Body>
        <xyzbank:getAccount>
            <xyzbank:accountNo>123456789</xyzbank:accountNo>
        </xyzbank:getAccount>
    </soapenv:Body>
</soapenv:Envelope>

<!-- Update Account -->
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:xyzbank="http://xyz.com">
    <soapenv:Body>
        <xyzbank:updateAccount>
            <xyzbank:accountNo></xyzbank:accountNo>
            <xyzbank:fullName>FF FDF</xyzbank:fullName>
            <xyzbank:dob>22-10-1987</xyzbank:dob>
            <xyzbank:ssn>3223434</xyzbank:ssn>
            <xyzbank:initialAmount>USD 10</xyzbank:initialAmount>
        </xyzbank:updateAccount>
    </soapenv:Body>
</soapenv:Envelope>

<!-- Delete Account -->
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:xyzbank="http://xyz.com">
    <soapenv:Body>
        <xyzbank:deleteAccount>
            <xyzbank:accountNo></xyzbank:accountNo>
        </xyzbank:deleteAccount>
    </soapenv:Body>
</soapenv:Envelope>
```

SOAP was initially widely adopted as the de facto standard for realizing web services, but has inherent drawbacks such as complexity of the protocol and related specifications. There are two main SOAP versions that are used in SOAP-based web services, namely SOAP 1.1 and 1.2. WSO2 ESB supports both versions and you can switch between each version seamlessly.

REST has emerged as a solution that is seeking to overcome the limitations of SOAP. REST describes a set of architectural principles by which data can be transmitted over the Internet (for instance, using HTTP). Unlike SOAP, REST does not define any constraints on the message format but focuses on design rules for the service. A client can access the resource using the unique URI and a representation of the resource is returned. While accessing RESTful resources with the HTTP protocol, the URL of the resource serves as the resource identifier and `GET`, `PUT`, `DELETE`, `POST`, and `HEAD` are the standard HTTP operations to be performed on that resource.

To understand the concept of the application of REST architecture, let's try to implement the same banking software solution's web service using the REST architecture. As depicted in Figure 5-2, you need to identify the resource and the respective operations that can take place on that resource. The `account` is the resource that can be identified by the URI `http://www.xyz.com/rest/accountmanagement/account`. The HTTP verbs `POST`, `GET`, `PUT`, and `DELETE` can be used to implement the account creation, retrieval, modification, and deletion, respectively.

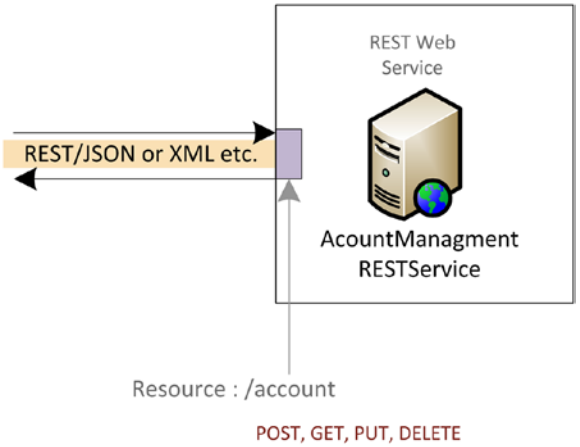

Figure 5-2. *AccountManagementRESTService exposes a web service that's based on REST architecture*

You can define all the four operations that are related to account management, as shown in Listing 5-2. For account creation, you can use an HTTP `POST` request along with the message payload that is required to create an account. The message format that can be used is not restricted in REST but often JSON or XML is used in REST web services. As the response, the service can return a reference such as an account ID, which can later be used to retrieve data using a `GET` request. A `PUT` request can be used to modify an existing account and the new content should be sent along with the request. An account can be deleted with a `DELETE` request, which is sent to the respective URI.

Listing 5-2. AccountManagementRESTService Operations

Create Account:

```
POST : http://www.xyz.com/rest/accountmanagement/account
Content-Type: application/json
{
  "Account": {
    "fullName": "FF FDF",
    "dob": "22-10-1987",
    "ssn": "3223434",
    "initialAmount": "USD 10"
}
```

```
Retrieve Account
GET : http://www.xyz.com/rest/accountmanagement/account/12345678

Modify Account
PUT : http://www.xyz.com/rest/accountmanagement/account/12345678
Content-Type: application/json
{
  "Account": {
    "id": "12345678"
    "fullName": "FF FDF",
    "dob": "22-10-1987",
    "ssn": "3223434",
    "initialAmount": "USD 10"
}

Delete Account
DELETE : http://www.xyz.com/rest/accountmanagement/account/12345678
```

In the context of an ESB, you must use ESB for integrating SOAP and/or REST web services. ESB has to talk to SOAP and REST services and expose SOAP or REST interfaces on top of the existing non-SOAP/non-REST based services/systems. Therefore, we can identify several use cases related to how you can use ESB in solving SOAP and REST web service integration problems. This chapter mainly focuses on addressing those use cases and how they can be implemented using WSO2 ESB.

Integrating SOAP Web Services

Integrating SOAP web services with WSO2 ESB is primarily about exposing SOAP web services at the ESB layer or invoking backend SOAP web services. In this section, you learn about several integration use cases related to SOAP based web services.

Exposing a SOAP Web Service Interface from WSO2 ESB

Exposing a SOAP web service interface at the ESB layer on behalf of existing backend services and systems is one of the most widely used use cases with an ESB.

Exposing a SOAP Web Service in Front of a Legacy Non-SOAP (PoX) Based Service

For example, assume that there is a baking software system that exposes a legacy account management web application, and it allows the external system to add, update, retrieve, and delete bank accounts by sending XML messages (POX) to it. But the bank wants to expose this legacy web application as a standard SOAP web service with a proper service description (WSDL).

As you learned in Chapter 4, the standard approach to expose a SOAP web service interface from WSO2 ESB is to use a proxy service. In addition, you need to create a service description for WSDL to be exposed to external parties. Based on the required operations and the required message format, we can come up with a WSDL (`AccountManagementService.wsdl`) (creating WSDLs is not covered in the scope of this book, as this is a generic SOAP web service related topic). Then, as depicted in Figure 5-3, we could create proxy service `AccountManagementService` and publish the created WSDL via the proxy service.

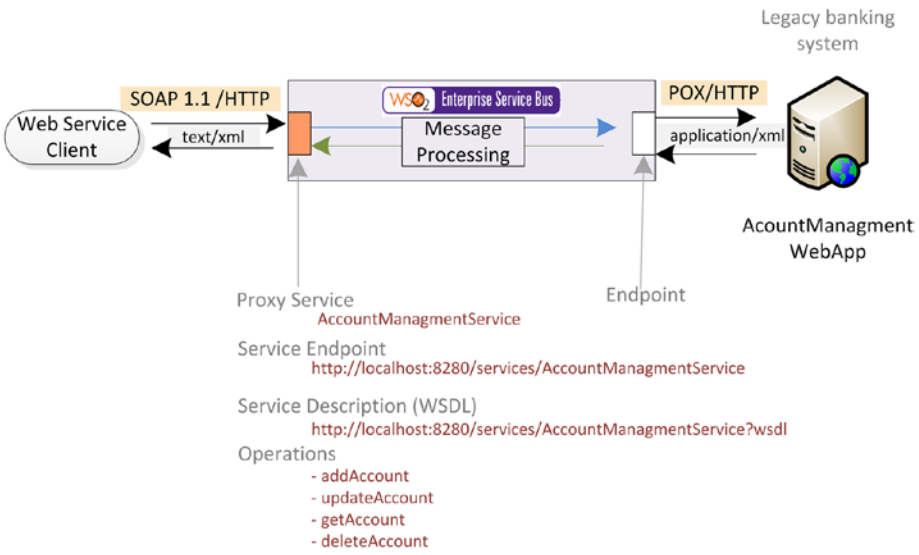

Figure 5-3. *Exposing a SOAP web service interface at the ESB layer using a proxy service on top of a legacy POX-based web application*

The web service client can determine the available operations, message format, and service endpoint by processing the WSDL of `AccountManagementService`. The request can be in SOAP 1.1 or 1.2 formats. Since the backend service accepts the PoX message format with `application/xml Content-Type`, you can set the outgoing message format of the endpoint to `format="pox"`. Here, we have assumed that there is a one-to-one mapping between the incoming SOAP message format and the outgoing XML (PoX) message format. Hence, we can simply set the outgoing message format of the endpoint as `pox`. However, in scenarios where there is no such mapping, we would need to use the message transformation techniques discussed in Chapter 4. Listing 5-3 shows the sample configuration of the `AccountManagementService`.

Listing 5-3. Proxy Service to Expose a SOAP Web Service Legacy XML (POX)-Based AccountManagementWebApp

```
<proxy xmlns="http://ws.apache.org/ns/synapse"
       name="AccountManagementService"                    <!-- [1] -->
       transports="http"                                  <!-- [2] -->
```

```
       startOnLoad="true"
       trace="disable">
   <description/>
   <publishWSDL key="gov:/AccountManagementService.wsdl" /> <!-- [3] -->
   <target>
      <inSequence>                                          <!-- [4] -->
         <log level="full"/>                                <!-- [5] -->
          <call>
              <endpoint
                <address
uri="http://localhost:6060/services/AccountManagementWebApp"
                           format="pox"/>
              </endpoint>
          </call>
                < log level="full"/>                <!-- [6] -->
                <respond/>
      </inSequence>
   </target>
</proxy>
[1] Proxy service name exposed as a part of the URL of the proxy service.
[2] Proxy service is exposed on HTTP transport so request can be sent only
via HTTP transport.
[3] You can specify the service contract (WSDL) that should be exposed to
the external client.
[4] Using in-sequence as the message-processing unit.
[5] Request message-processing logic.
[6] Response message-processing logic.
```

Exposing a SOAP Web Service with WS-Addressing

In addition to exposing a SOAP web service, there can be a requirement to enable Web Service Addressing (`WS-Addressing`) for a given SOAP web service interface. The main reason for using `WS-Addressing` is that SOAP does not provide a standard way to specify the destination of a message, where to return a response, or where to report an error. In the absence of `WS-Addressing`, that is pretty much handled at the transport level (for example using HTTP headers), but it's not sufficient in certain use cases.

For example, as shown in Listing 5-4, with `WS-Addressing` you can specify addressing information such as MessageID, To, Action, ReplyTo, FaultTo, and so on in the SOAP envelope. The request is delivered to the To URI. The Action URI indicates the action to be taken. (The Action URI should represent a service corresponding to a WSDL port type.) As the name implies, ReplyTo and FaultTo simply specify the reply and fault-handling locations.

Listing 5-4. Using WS-Addressing in AccountManagementService

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                      xmlns:xyzbank="http://xyz.com"
                      xmlns:wsa="http://www.w3.org/2004/12/addressing">
        <soapenv:Header>
            <wsa:MessageID>http://xyz.com/1234567</wsa:MessageID>
            <wsa:ReplyTo>
                <wsa:Address>http://xyz.com/WSClient</wsa:Address>
            </wsa:ReplyTo>
            <wsa:FaultTo>
                <wsa:Address>http://xyz.com/FaultHandler</wsa:Address>
  </wsa:FaultTo>
        <wsa:To>http://esb.prod.com:8280/services/AccoutManagementService
        </wsa:To>

                <wsa:Action>http://xyz.com/AddAccount</wsa:Action>
  </soapenv:Header>
    <soapenv:Body>
            <xyzbank:addAccount>
                <xyzbank:fullName>Foo Bar</xyzbank:fullName>
                <xyzbank:dob>22-10-1987</xyzbank:dob>
                <xyzbank:ssn>3223434</xyzbank:ssn>
                <xyzbank:initialAmount>USD 10</xyzbank:initialAmount>
            </xyzbank:addAccount>
    </soapenv:Body>
</soapenv:Envelope>
```

As depicted in Figure 5-4, if the `AccountManagementService` requires having the `WS-Addressing` support, then we can enable `WS-Addressing` at the ESB proxy service level.

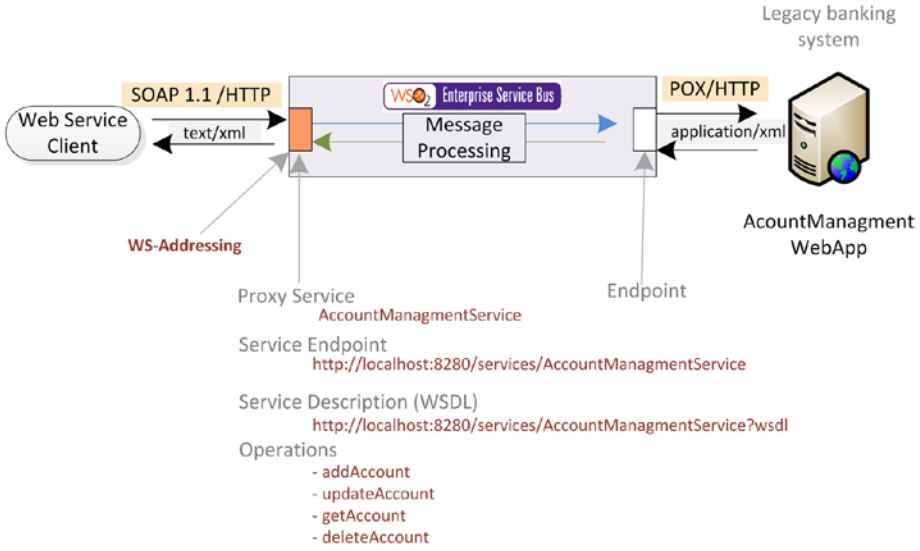

Figure 5-4. *Enabling WS-Addressing for a proxy service exposed from ESB*

You can simply enable `WS-Addressing` at the proxy service level by adding a service parameter to the proxy service, as shown in Listing 5-5.

Listing 5-5. Using WS-Addressing in AccountManagementService Proxy Service

```
<proxy xmlns="http://ws.apache.org/ns/synapse"
       name="AccountManagementService"
       transports="http"
       startOnLoad="true"
       trace="disable">
   <description/>
   <publishWSDL key="gov:/AccountManagementService.wsdl" />
   <target>
      <inSequence>
         <log level="full"/>
          <call>
              <endpoint
                <address
uri="http://localhost:6060/services/AccountManagementWebApp"
                              format="pox"/>
              </endpoint>
          </call>
                < log level="full"/>
                <respond/>
      </inSequence>
   </target>
```

```
   <parameter name="enforceWSAddressing">true</parameter> <!-- [1] -->
</proxy>
[1] Enabling WS-Addressing for the proxy service by specifying the
enforceWSAddressing service parameter
```

Therefore, all the requests coming into `AccountManagementService` will be served with the compliancy with `WS-Addressing`.

Exposing a SOAP 1.2 Web Service on Top of a SOAP 1.1 Web Service

In the context of SOAP web services, there are two flavors of SOAP message formats, namely SOAP 1.1 and SOAP 1.2. The integration between disparate types of SOAP services is often a common integration problem. For example, suppose that we have a financial service that provides `StockQuotes` for a given company name, and it's implemented as a SOAP 1.1 web service. But there is a client application that only supports SOAP 1.2 and it wants to consume the `StockQuote` service. For that scenario, you can use WSO2 ESB as depicted in Figure 5-5.

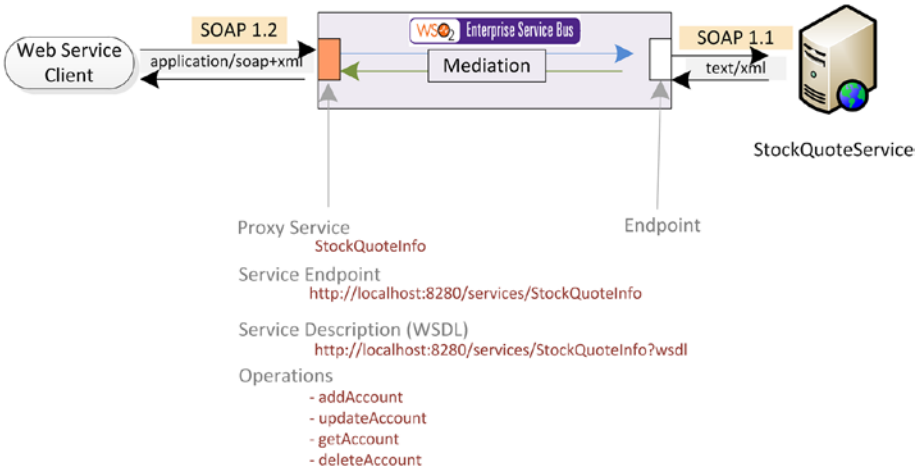

Figure 5-5. *Integrating a SOAP 1.2-compliant client and a SOAP 1.1 backend service*

In this scenario, the request message format sent in the SOAP 1.2 format and the request that is expected by `StockQuoteService` are also different (not just the SOAP version, but the message format of the SOAP body). Therefore, as shown in Listing 5-6, you need to do the message format transformation (from the `checkQuote` request to the `getQuote` request) as well as SOAP 1.2 to SOAP 1.1 conversion. You can convert the existing message format to either SOAP 1.1 or SOAP 1.2 by specifying the endpoint format as `format="soap11" | "soap12"`.

Listing 5-6. Proxy Service Configuration to Transform the Request Message and Convert the Message from SOAP 1.2 to SOAP 1.1 Format

```
<proxy xmlns="http://ws.apache.org/ns/synapse"
       name="StockQuoteProvider"
       transports="https,http"
       statistics="disable"
       trace="disable"
       startOnLoad="true">
   <target>
      <inSequence>
         <log level="custom">
            <property name="FLW" value="=== Service Invoked ==="/>
         </log>
         <payloadFactory media-type="xml">         <!-- [1] -->
            <format>
               <soap:Envelope
                              xmlns:soap="http://www.w3.org/2003/05/soap-
                              envelope"
                              xmlns:xsd="http://services.samples/xsd"
                              xmlns:ser="http://services.samples">
                  <soap:Header/>
                  <soap:Body>
                     <ser:getQuote>
                        <ser:request>
                           <xsd:symbol>$1</xsd:symbol>
                        </ser:request>
                     </ser:getQuote>
                  </soap:Body>
               </soap:Envelope>
            </format>
            <args>
               <arg xmlns:abc="http://abc.org"
                    evaluator="xml"
                    expression="//abc:checkQuote/abc:symbol"/>
            </args>
         </payloadFactory>
         <call>
            <endpoint>
               <address uri="http://127.0.0.1:9000/services/
               SimpleStockQuoteService"
                        format="soap11"/>        <!-- [2] -->
            </endpoint>
         </call>
         <property name="messageType"            <!-- [3] -->
                   value="application/soap+xml"
                   scope="axis2"
                   type="STRING"/>
```

```
        <log level="full"/>
        <respond/>
      </inSequence>
   </target>
   <description/>
</proxy>
[1] Transform the message from checkQuote request format to getQuote format.
[2] Convert the SOAP 1.2 to SOAP 1.1 message format.
[3] Ensure the response which is sent back to the client is in SOAP 1.2
format.
```

Once the outgoing message format is specified, the message format conversion (between SOAP versions) happens automatically.

Invoking SOAP Backend Web Services from ESB

Invoking a SOAP backend web service and exposing that service as a different interface is a common integration use case. For example, assume that the financial `StockQuoteService` is a SOAP web service and an external client who can only communicate with XML (POX) wants to consume that service. As shown in Figure 5-6, you can enable the message format attribute of the endpoint to `soap11` and expose the service through an HTTP API that takes care of the message transformation from SOAP to POX. Again, we have considered a scenario where there is no one-to-one mapping between the POX and SOAP request. Hence, you need to use message transformation mediators to convert the message into SOAP 1.1.

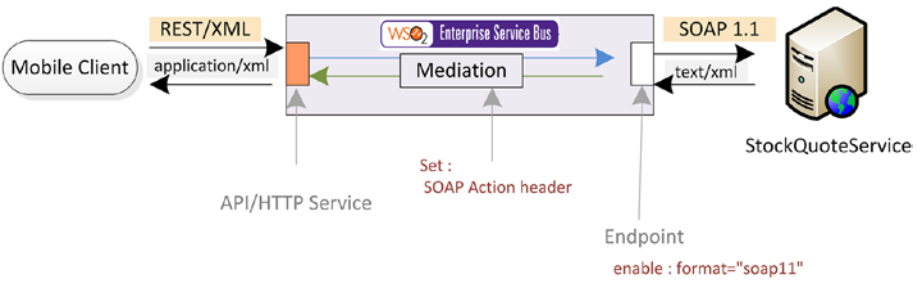

Figure 5-6. *Invoking a SOAP backend service from ESB and exposing that as a RESTful service*

As you are invoking a SOAP backend service, you need to send the required `SOAPAction` along with the request to the backend service (`SOAPAction` goes as an HTTP header). As you learned in Chapter 4, you can use the header mediator to set the `SOAPAction` header. Listing 5-7 illustrates the configuration of the API that implements this integration scenario.

Listing 5-7. API Configuration that Invokes a Backend SOAP Service and Exposes it as a POX interface/API

```
<api xmlns="http://ws.apache.org/ns/synapse"
                name="StockQuoteAPI" context="/stockquote">
   <resource methods="POST">
      <inSequence>
         <log level="custom">
            <property name="FLW" value="=== Service Invoked ==="/>
         </log>
         <payloadFactory media-type="xml">              <!-- [1] -->
            <format>
               <soap:Envelope xmlns:soap="http://www.w3.org/2003/05/
               soap-envelope" xmlns:xsd="http://services.samples/xsd"
               xmlns:ser="http://services.samples">
                  <soap:Header/>
                  <soap:Body>
                     <ser:getQuote>
                        <ser:request>
                           <xsd:symbol>$1</xsd:symbol>
                        </ser:request>
                     </ser:getQuote>
                  </soap:Body>
               </soap:Envelope>
            </format>
            <args>
               <arg evaluator="xml" expression="//checkQuote/symbol"/>
            </args>
         </payloadFactory>
         <log level="full"/>
                <!-- [2] -->
         <header name="Action" scope="default" value="urn:getSimpleQuote"/>
         <call>
            <endpoint>
               <address uri="http://127.0.0.1:9000/services/
               SimpleStockQuoteService" format="soap11"/>
            </endpoint>
         </call>
         <property name="messageType" value="application/xml" scope="axis2"
         type="STRING"/> <!-- [3] -->
         <log level="full"/>
         <respond/>
      </inSequence>
   </resource>
</api>
[1] Transform the message from checkQuote request format to getQuote format.
[2] Setting the SOAP action prior to invoking the SOAP backend service.
[3] Convert the response message format back to the message format expected
by the client.
```

Additionally, you may want to invoke a SOAP backend service that requires a `WS-Addressing` message header. To enable `WS-Addressing` for outgoing messages, you can use `enableAddressing` at the endpoint level, as shown in Listing 5-8.

Listing 5-8. Enabling WS-Addressing for Outgoing Messages

```
<endpoint>
   <address uri="http://127.0.0.1:9000/services/SimpleStockQuoteService">
      <enableAddressing/>
   </address>
</endpoint>
```

If you want to control any `WS-Addressing` headers prior to sending to the backend service, you can do so by using the header mediator in the mediation flow.

Integrating RESTful Web Services

RESTful web services (web services implemented based on the REST architecture) are increasingly getting popular and most of the modern web services implementations use the REST design paradigm. Therefore, for any ESB implementation, it is quite important to have the capability of exposing a RESTful web service interface as well as the capability to invoke the backend RESTful service. In the section, you get a comprehensive understanding about how you achieve those objectives using WSO2.

Exposing RESTful Services/APIs with WSO2 ESB

As you use proxy services to expose SOAP web service interfaces in WSO2 ESB, you can use APIs/HTTP services to expose a RESTful web service interface from WSO2 ESB. Let's take a sample scenario from what we discussed at the beginning of this chapter. In the banking scenario that we discussed, assume that the bank implemented its account management capabilities as a SOAP web service, named `AccountManagementService`, as shown in Figure 5-7.

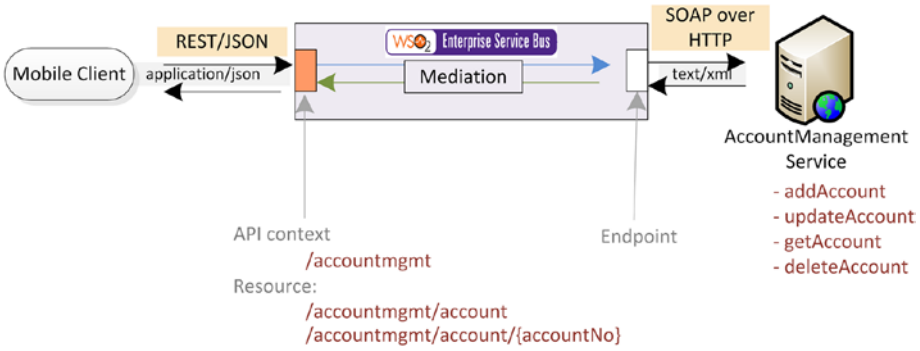

Figure 5-7. *Exposing a RESTful service using an API/HTTP service on top of an existing SOAP backend web service. API is anchored at <host>:<esb_port>/accountmgmt and the account is defined as a resource. All the SOAP web service operations are mapped to the HTTP request with respective verbs (POST, GET, PUT, DELETE) which are sent to the /account resource.*

`AccountMangementService` exposes all the supported functionalities—`addAccount`, `updateAccount`, `getAccount`, and `deleteAccount`, as SOAP web service operations defined in its WSDL. As we discussed in the beginning of the chapter, in the REST architecture, you can map those web service operations into resources and HTTP requests with different verbs. What follows are the main design aspects of the implementation of the use case in Figure 5-7.

- You can define a context for your API/HTTP service. Here we used `/accountmgmt`. That means that the client will be sending the request to `http://<esb_host>:<port>/accountmgmt*`. (Assume that it's `http://localhost:8280` for this example.)
- Then you can identify the account as a resource in the business use case. Hence, you can define that as a resource in your REST API with the resource URI `/account`.
- Then you can allow the client applications to send HTTP request (in this case, with the JSON message format), against the respective resource. The details of those requests are summarized in the next couple of points.
- Add a new account: HTTP `POST` request to `http://locahost:8280/accountmgmt/account` with the JSON payload required to create an account.
- Retrieve or delete an existing account: HTTP `GET` or `DELETE` request sent to `http://localhost:8280/accountmgmt/account/<accountNo>` with no message content.
- Update an existing account : HTTP `PUT` request sent to `http://localhost:8280/accountmgmt/account/<accountNo` with the JSON payload with updated account details.

To start, you can define the API configuration at the ESB level, as shown in Listing 5-9. Here we have defined a context and two resources that are mapped to `/account*` and `account/<accountNo>`. As discussed, the new account creation request doesn't have an account no. in its request. Hence they are handled in the resource with URI template `/account*`.

Listing 5-9. Creating the API/HTTP Service and its Resources

```
<api xmlns="http://ws.apache.org/ns/synapse"
name="AccountManagementAPI"                              <!-- [1] -->
context="/accountmanagement/account">                            <!-- [2] -->
    <resource methods="POST" uri-template="/"> <!-- [3] -->
        <inSequence>                                     <!-- [4] -->
          ...

        </inSequence>
    </resource>
    <resource methods="DELETE PUT GET"                   <!-- [5] -->
uri-template="/{accountNo}">                    <!-- [6] -->
        <inSequence>
          ...
        </inSequence>
    </resource>
</api>
[1] Name of the API/HTTP service. But this is not part of the API context.
[2] URI context of the API.  Any API at ESB is identified based on the
context.
[3] Resource definition of account to accept all account creation requests
(POST requests) comes with specified uri-template. Requests are filtered
based on the URI-Template.
[4] sequence for process/mediate messages that comes into the resource.
[5] resource to accept all GET, PUT and DELETE requests that comes with URI
format /account/<id>
[6] URI Template is used for filter and populate the matching variables
which later be used in mediation flow.
```

The URI template allows you to specify the incoming request URL format and, for any matched URL, the `URI-Template` variables are populated. Those values can be used in the mediation flow. You will further discover URI templates in the latter part of this section.

Now you can proceed with the mediation flow implementation of the account creation resource `/account*`. The configuration of the account creation mediation logic is shown in Listing 5-10.

Listing 5-10. Implementation of Account Creation Mediation Logic

```
<api xmlns="http://ws.apache.org/ns/synapse"
name="AccountManagementAPI"
context="/accountmanagement/account">
    <resource methods="POST" uri-template="/">
```

```
        <inSequence>
            <log level="custom">
                <property name="FLW" value="Adding account"/>
            </log>
            <payloadFactory media-type="xml">          <!-- [1] -->
                <format>
                    <acc:addAccountRequest
xmlns:acc="http://www.xyzbank.com/accounts">
                        <fullName>$1</fullName>
                        <dob>$2</dob>
                        <ssn>$3</ssn>
                        <accountBalance>$4</accountBalance>
                    </acc:addAccountRequest>
                </format>
                <args>
                    <arg evaluator="json"
                         expression="$.Account.fullName"/>
                    <arg evaluator="json"
                         expression="$.Account.dob"/>
                    <arg evaluator="json"
                         expression="$.Account.ssn"/>
                    <arg evaluator="json"
                         expression="$.Account.initialAmount"/>
                </args>
            </payloadFactory>

<header name="Action"                                    <!-- [2] -->
value="http://www.xyzbank.com/accounts/addAccount"/>
            <call>
                <endpoint>
                    <address
uri="http://localhost:7070/AccountManagementService"
                             format="soap11"/>          <!-- [3] -->
                </endpoint>
            </call>

            <property name="messageType"               <!-- [4] -->
                      value="application/json"
                      scope="axis2"
                      type="STRING"/>
            <respond/>

        </inSequence>
    </resource>
    <resource methods="DELETE PUT GET" uri-template="/{accountNo}">
        <inSequence>
          ...
        </inSequence>
```

```
    </resource>
</api>
[1] Create the SOAP request payload to be sent to AccountManagementService
by extracting values from incoming JSON message.
[2] Specify SOAP Action.
[3] Endpoints need to have format attribute set to soap11 (or soap12).
[4] Flag to convert the response message back to JSON.
```

In Listing 5-10, the mediation logic is quite straightforward and contains the mediators that we have already discussed in similar integration scenarios.

Let's proceed to the implementation of the other resource, which processes all the GET, PUT, and DELETE account requests. Listing 5-11 shows the complete configuration of the resources that handles the GET, PUT, and DELETE requests. In this configuration, the only additional thing is the use of HTTP_METHOD for filtering GET, PUT, and DELETE requests using a switch mediator and processing those requests in different case conditions. Also, unlike account creation, we need to change the HTTP method of the outgoing request. (In account creation, the incoming HTTP method is POST and the outgoing SOAP request is also an HTTP POST.) As SOAP HTTP requests are always HTTP POST, for all GET, PUT, and DELETE requests, we need to change the HTTP_METHOD (HTTP verb) to POST.

Listing 5-11. Implementation of Account Update, Retrieve, and Delete Mediation Logic

```
<api xmlns="http://ws.apache.org/ns/synapse"
name="AccountManagementAPI"
context="/accountmanagement/account">
    <resource methods="POST" uri-template="/">
        <inSequence>
          ...
        </inSequence>
    </resource>

    <resource methods="DELETE PUT GET" uri-template="/{accountNo}">
        <inSequence>
          <switch source="$axis2:HTTP_METHOD">
                <case regex="GET">
                    <log level="custom">
                        <property name="Message Flow"
value="--- Account Retrieve ---"/>
                    </log>

<payloadFactory media-type="xml">
                        <format>
                            <acc:getAccountRequest
  xmlns:acc="http://www.xyzbank.com/accounts">$1
</acc:getAccountRequest>
                        </format>
                        <args>
                            <arg expression="$ctx:uri.var.accountNo"/>
```

```
                    </args>
                </payloadFactory>
                <header name="Action"            <!-- [1] -->
value="http://www.xyzbank.com/accounts/getAccount"/>
            </case>
            <case regex="PUT">
                <log level="custom">
                    <property name="Message Flow"
value="--- Account Update ---"/>
                </log>

                <payloadFactory media-type="xml">
                    <format>
                        <acc:updateAccountRequest
xmlns:acc="http://www.xyzbank.com/accounts">
                            <fullName>$1</fullName>
                            <dob>$2</dob>
                            <ssn>$3</ssn>
                            <accountBalance>$4</accountBalance>
                            <accountNo>$5</accountNo>
                        </acc:updateAccountRequest>
                    </format>
                    <args>
                        <arg evaluator="json"
                             expression="$.Account.fullName"/>
                        <arg evaluator="json"
                             expression="$.Account.dob"/>
                        <arg evaluator="json"
                             expression="$.Account.ssn"/>
                        <arg evaluator="json"
                             expression="$.Account.initialAmount"/>
                        <arg expression="$ctx:uri.var.accountNo"/>
                    </args>
                </payloadFactory>
                <header name="Action"
value="http://www.xyzbank.com/accounts/updateAccount"/>
            </case>
            <case regex="DELETE">
                <log level="custom">
                    <property name="Message Flow"
value="--- Account Delete ---"/>
                </log>
<payloadFactory media-type="xml">
                    <format>
                        <acc:deleteAccountRequest
xmlns:acc="http://www.xyzbank.com/accounts">$1</acc:deleteAccountRequest>
                    </format>
                    <args>
```

```
                            <arg expression="$ctx:uri.var.accountNo"/>
                        </args>
                    </payloadFactory>
                    <header name="Action"
 value="http://www.xyzbank.com/accounts/getAccount"/>
                </case>
            </switch>
                <property name="HTTP_METHOD" value="POST" <!-- [2] -->
scope="axis2" type="STRING"/>

            <call>
                <endpoint>
                    <address
uri="http://localhost:7070/AccountManagementService"
                             format="soap11"/>
                </endpoint>
            </call>
            <property name="messageType"
                      value="application/json"
                      scope="axis2"
                      type="STRING"/>
            <respond/>
        </inSequence>
    </resource>
</api>
[1] Set required Action for each AccountManagementService operation.
[2] Change the all the HTTP_METHODs to POST as SOAP web services accepts
HTTP POST only
```

As you learned about a complete RESTful integration use case, let's summarize the concepts that are covered in this example.

Fundamentals of REST APIs

A REST API or HTTP service can be used to expose an HTTP interface, which adheres to REST architecture, at the ESB level. As shown in the sample REST API design in Figure 5-8, we can identify the key attributes of REST APIs in WSO2 ESB.

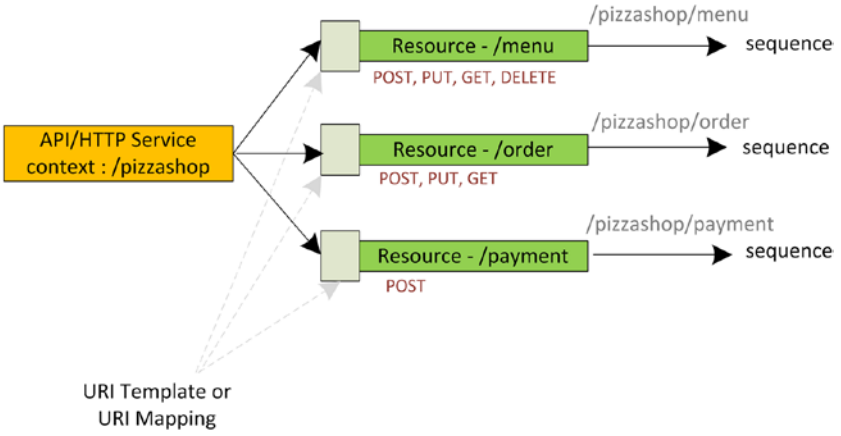

Figure 5-8. *REST API consists of a context, a set of resources defined for arbitrary numbers of HTTP verbs (GET, POST, etc.), and a URI template/mapping defined for each resource. Once a message comes into a resource, it is processed by a sequence.*

The REST API/HTTP service in WSO2 has the following attributes.

- A REST API is made of one or more resources (in this example, `/menu`, `/order`, and `/payments`).
- A resource is a logical component of an API, and can be accessed by making a particular type of HTTP calls.
- A resource can be associated with a user-defined URL pattern or a URI template. This way we can restrict the type of HTTP requests processed by a particular resource. In addition, the URI template can be used to filter and extract the values that match the template.
- A resource can be bound to a specific subset of HTTP verbs and header values. This option provides additional control over which requests are handled by a given resource. Once a request is dispatched into a resource, it will be mediated through the in-sequence of the resource.
- A resource can be associated with a *URL mapping* or a *URI template*.
- A *URL mapping* could be used to filter a given HTTP request that comes to a resource. Only the matching request will be processed by the resource. URL mapping can be any valid servlet mapping. Hence, as stated in the servlet specification, there are three types of URL mappings:
    - Path mappings (`/menu/*` and `/menu/pizza/*`)
    - Extension mappings (`*.jsp` and `*.do`)
    - Exact mappings (`/menu` and `/menu/pizza`)

- URI template represents a class of URIs using patterns and variables. As you discovered in the `AccountManagement` use case, the following URI template can be used to match the respective HTTP request.
- `/account/{accountNo} : Matching HTTP request / account/123456`
- Here, the variable `accountNo` is matched with `123456` and, in the mediation flow those values will be stored as properties. Hence, you can refer any matching variable using the following syntax.
- `$ctx:uri.var.<VariableName>`,for example, `$ctx:uri.var.accountNo`
- Similarly you can access any query parameter that comes in the request with `$ctx:query.param.<ParameterName>`. For example, if the incoming request contains `/pizzashop/api/menu/pizza?val=thin&type=crust`, you can retrieve the value of the `type` query parameter by using the `property $ctx:query.param.type`.

Now you have learned how to expose RESTful services from WSO2 ESB. In the next section, you will learn about how you can invoke RESTful services from ESB.

Invoking RESTful Services from WSO2 ESB

Invoking backend services, which are based on REST architecture, is another key area of the RESTful integration paradigm. WSO2 ESB provides a dedicated endpoint type for all such invocation of RESTful backend web services, namely HTTP endpoint (outbound endpoint). For example, let's consider a PizzaShop software solution that allows you to view available pizza menu, order pizza, and process payments. As depicted in Figure 5-9, the backend RESTful service has defined multiple resources for pizza, order, and purchasing.

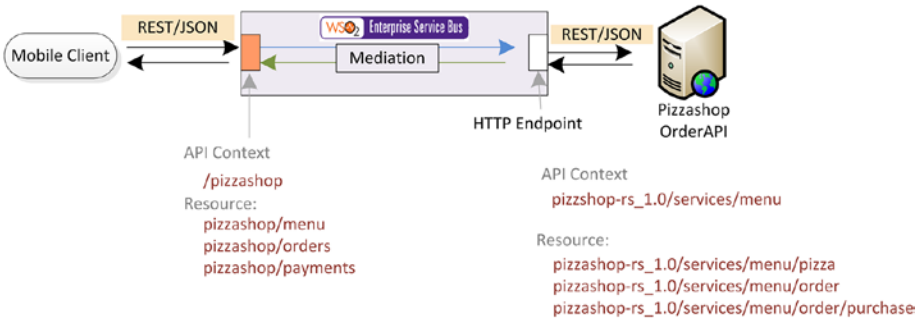

Figure 5-9. *Invoking a backend RESTful service using an HTTP endpoint*

Then as you learned in previous section, you can define a new context and a set of new resources at the ESB REST API level. Once you define the required API and resource, you need to invoke the RESTful service from ESB. For that, you can use HTTP endpoint. As an example, consider the pizza ordering use case. To create, retrieve, and delete an order, you need to send the following HTTP request to the backend service with the respective payloads.

```
Add order
POST http://localhost:9764/pizzashop-rs_1.0/services/menu/order
{"order": {
    "items": [
        {"id": 1, "qty": 2},
        {"id": 2, "qty": 1}
    ]
}}

Get/Delete order
GET/DELETE
http://localhost:9764/pizzashop-rs_1.0/services/menu/order/1379562336828

Update order
PUT http://localhost:9764/pizzashop-rs_1.0/services/menu/order/1379562336828
{order: {
    items: [
        {id: 1, qty: 5},
        {id: 3, qty: 1}
    ]
}}
```

Therefore, what is meant by invoking a PizzaShop RESTful service is that ESB needs to send the aforementioned HTTP requests to the backend. If you consider the `/pizzashop/order` resource implementation of the PizzaShop REST API, you can see HTTP endpoint is used to send this request to the backend service using a call mediator.

As shown in Listing 5-12, with HTTP endpoint, you can define the HTTP method name (`GET`, `POST`, `PUT`, etc.) and the outgoing resource URL as an URI template with variables. The variables need to be defined in the `{uri.var.varaibleName}` format. You can define any property value in your mediation flow with the prefix `uri.var.` and later use that in your HTTP endpoint URI template.

Listing 5-12. Implementation of Account Update, Retrieve, and Delete Mediation Logic

```
<endpoint>
  <http method="GET"
uri-template="http://localhost:9764/ps/order/{uri.var.orderId}">
     </http>
</endpoint>
```

Listing 5-13 shows a complete integration scenario of the /pizzashop/orders use case. Here we defined two resources: /api/order* and /api/order/{orderId}.

Listing 5-13. Implementing Pizzashop Orders API by Invoking a RESTful Backend Service Using HTTP Endpoint

```
<api xmlns="http://ws.apache.org/ns/synapse"
     name="PizzaShopRESTAPI"
     context="/pizzashop">

    <resource methods="POST" uri-template="/api/order*">
        <inSequence>
            <log>
                <property name="Message Flow"
value="Pizza Order API - IN"></property>
            </log>
            <call>
                <endpoint>
                    <http method="post"         <!-- [1] -->
                      uri-template="http://localhost:9764/pizzashop-rs_1.0/
                      services/menu/order"></http>
                </endpoint>
            </call>
            <respond/>
        </inSequence>
    </resource>

    <resource methods="GET DELETE PUT"
uri-template="/api/order/{orderId}">
        <inSequence>
            <log>
                <property name="Message Flow"
value="Pizza Order API - IN"></property>
            </log>
            <switch source="$axis2:HTTP_METHOD">
                <case regex="GET">
                    <log level="custom">
                        <property name="Message Flow"
value="--- Order GET ---"></property>
                    </log>
                    <call>
                        <endpoint>
                            <http method="GET"    <!-- [2] -->

uri-template="http://localhost:9764/pizzashop-rs_1.0/services/menu/order/
{uri.var.orderId}"></http>
                        </endpoint>
                    </call>
                </case>
```

```
                <case regex="PUT">
                    <log level="custom">
                        <property name="Message Flow"
value="--- Order PUT ---"></property>
                    </log>
                    <call>
                        <endpoint>
                            <http method="PUT"

uri-template="http://localhost:9764/pizzashop-rs_1.0/services/menu/order/
{uri.var.orderId}"></http>
                        </endpoint>
                    </call>
                </case>
                <case regex="DELETE">
                    <log level="custom">
                        <property name="Message Flow"
value="--- Order DELETE ---"></property>
                    </log>
                    <call>
                        <endpoint>
                            <http method="DELETE"

uri-template="http://localhost:9764/pizzashop-rs_1.0/services/menu/order/
{uri.var.orderId}"></http>
                        </endpoint>
                    </call>
                </case>
            </switch>
            <respond/>
        </inSequence>
    </resource>
</api>
[1] HTTP Endpoint is used to invoke the backend RESTful service. Here we
haven't used any variables in the URI template of HTTP endpoint.
[2] HTTP Endpoint is configured with a URI Template variable which is
populated during URI Template matching in resource.
```

Using HTTP Endpoint for RESTful Service Invocations

As you have seen in the examples in this chapter, HTTP endpoint is a logical representation of an actual resource that allows users to specify the noun and verb in RESTful style. Therefore, we can simply represent a resource with a URI template and the required HTTP verb can be selected too. The URI template is fully compliant with RFC 6570 (https://tools.ietf.org/html/rfc6570) and the variable names should start with

`uri.var.*` or `query.param.*`. A given REST API defined in WSO2 ESB can map the URI parameters in to `uri.var.*` or `query.param.*` and later can be reused when invoking the actual backend RESTful service.

Native JSON Support

JSON is becoming increasingly popular as a message interchange format with many RESTful integration scenarios. WSO2 ESB offers native support for JSON, which means there will be no message conversions (back and forth) between internal canonical forms, but you can write the mediation logic based on the incoming JSON payload with no conversion. Listing 5-14 shows several uses of JSONPath with the WSO2 ESB mediators.

Listing 5-14. Native JSON Support with JSONPath

```
<filter source="json-eval(pizza.name)" regex="Meat Sizzler"> <!-- [1] -->
    <then>
     <log level="custom">
         <property name="THEN_FLOW" value="Pizza Found"/>
     </log>
   </then>
   <else>
     <log level="custom">
         <property name="ELSE_FLOW" value="Not Found"/>
     </log>
   </else>
 </filter>
<payloadFactory media-type="json">
         <format>{"purchaseInformation": {"amount": "$1","cc": "$2"}}</
         format>
         <args>
                <arg evaluator="json" expression="$.payment.amount_lkr">
                </arg>   <!-- [2] -->
                <arg evaluator="json" expression="$.payment.card_no"></arg>
         </args>
</payloadFactory>
[1] Using json-eval to evaluate the JSON message and do filtering based on
its result.
[2] PayloadFactory can use JSONPath when you transform JSON to JSON.
```

Native JSON support is allowed in all the basic mediators, which includes property, filter, switch, and log mediator. The native JSON path (http://goessner.net/articles/JsonPath/) evaluation is done using `json-eval(json_path_expression)`. We demonstrated how to carry out content-based routing using a filter mediator against the incoming JSON without converting the incoming message to a canonical form.

Summary

In this chapter, you learned:

- The basic concepts of the SOAP and REST web services.
- How to design a SOAP web service for a real-world scenario (banking account management) and how to design the REST web service for the same use case.
- Exposing web service interfaces from WSO2 ESB using proxy services.
- Using WSDLs and `WS-Addressing` with proxy services.
- Invoking a backend SOAP web service from WSO2 ESB.
- Exposing RESTful HTTP interfaces from WSO2 ESB using WSO2 ESB REST APIs.
- Using REST API resources, URI templates/mapping, and HTTP verbs.
- Invoking a backend RESTful services by using HTTP endpoint.

CHAPTER 6

Enterprise Messaging with JMS, AMQP, MQTT, and Kafka

Modern software solutions are comprised of disparate software applications and they need to communicate with each other to realize a business use case. These software applications communicate with each other by exchanging messages over the network using enterprise-messaging systems. These application-to-application messaging systems are known as Message Oriented Middleware (MoM).

As shown in Figure 6-1, enterprise messaging systems, or MoMs, allow two or more software applications to exchange information in the form of messages. A message is self-contained and is comprised of business data along with the information required to route the message (message headers). In enterprise messaging or with MoM, messages are transferred between the applications over the network, asynchronously. Therefore, the sender doesn't have to wait until the messages are received by the receiver and the receiver can receive the messages when the sender is offline.

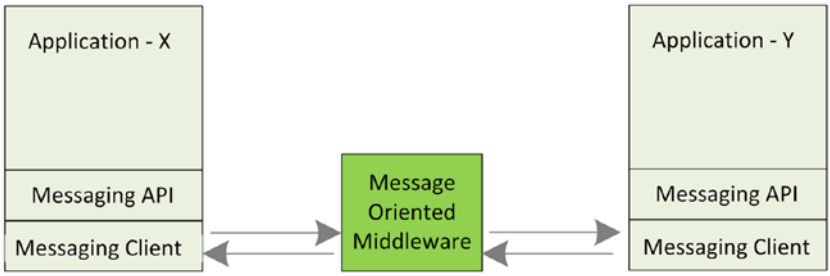

Figure 6-1. *Messages are transferred between software applications with the use of enterprise messaging systems or Message Oriented Middleware (MoM)*

K. Indrasiri, *Beginning WSO2 ESB*, DOI 10.1007/978-1-4842-2343-7_6

MoM vendors use different protocols and technologies, but the semantics for sending and receiving messages are the same. They provide an API to the developers for sending and receiving messages and the similarities between such API semantics motivates the enterprise to go for a vendor-agnostic Java API for sending and receiving messages in MoM, namely Java Message Service (JMS).

JMS provides an abstraction layer to send and receive messages from MoM. As shown in Figure 6-2, MoM exposes the message receiving and sending operations through a JMS API and it provides the required client libraries that contain the implementation of those APIs. The client that wants to send/receive message to MoM simply uses the JMS API so that his code is completely independent from the underlying MoM. The MoM products such as ActiveMQ, IBM WebSphere MQ, WSO2 MB, etc. implement JMS API to provide capabilities to the client to send and receive message from those MoM systems. JNDI is an implementation-independent API for directory and naming systems. A directory service provides JMS clients with access to the `ConnectionFactory` and `Destinations` (topics and queues) objects.

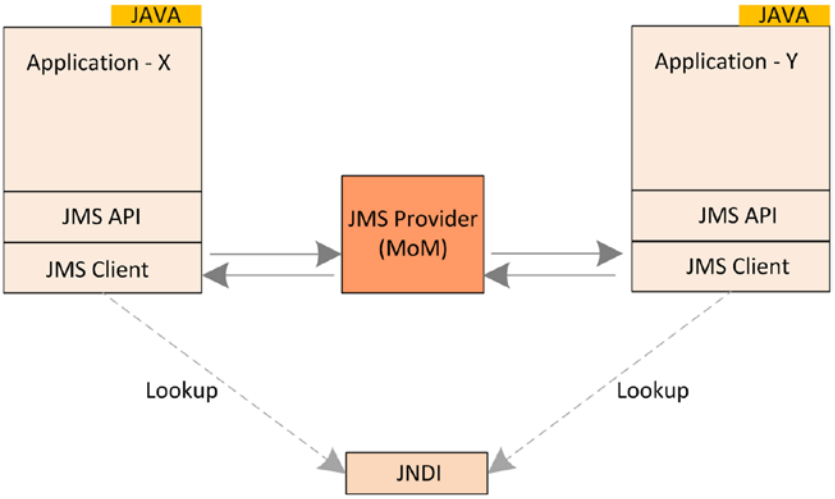

Figure 6-2. *JMS provides an API that provides abstraction for sending and receiving messages. That allows the application to use any JMS-compliant MoM to pass messages between Java applications. JMS uses the central Java naming service JNDI.*

In this chapter, you will learn about how WSO2 ESB can be used to integrate with enterprise messaging systems using JMS API. We will discuss the common integration patterns and use cases with JMS-based MoM and, during the latter part of the chapter, you will learn about other standards and technologies such as AMQP, MQTT, Kafka, etc. in the enterprise messaging space.

Integration with JMS-Based MoM

Integrating JMS-based enterprise messaging systems such as ActiveMQ, WSO2 MB etc., is a common integration required in the enterprise integration space. Often there are business use cases that need to consume messages from a JMS-based MoM or send messages to a JMS-based MoM. WSO2 ESB is capable of catering to both JMS consumer and JMS producer roles.

ESB as a JMS Consumer

You can use WSO2 ESB to consume messages from an enterprise messaging system or MoM using JMS API. Given that the MoM supports JMS API, WSO2 ESB can connect to JMS-based MoM (JMS provider) and consume messages from it. In the JMS specification, there are two types of messaging models presented—point-to-point queuing and publish-subscribe messaging models. The point-to-point is intended for one-to-one delivery of messages while publish-and-subscribe is intended for a one-to-many broadcast of messages. To understand how WSO2 ESB can be used to consume messages from a JMS-based MoM using these messaging models, let's consider a real-world use case.

As shown in Figure 6-3, suppose that a supermarket chain uses an enterprise messaging system based on JMS (Apache ActiveMQ) to store the price updates of each item in the supermarket. These price update details messages need to be sent to a SOAP-based web service periodically. The order of price updates must be preserved, as there can be multiple ordered messages for the same item. Therefore, price updates are stored in a JMS queue in ActiveMQ and the consumer has to consume them in the same order. To implement this use case, you can use JMS message consuming component in WSO2 ESB.

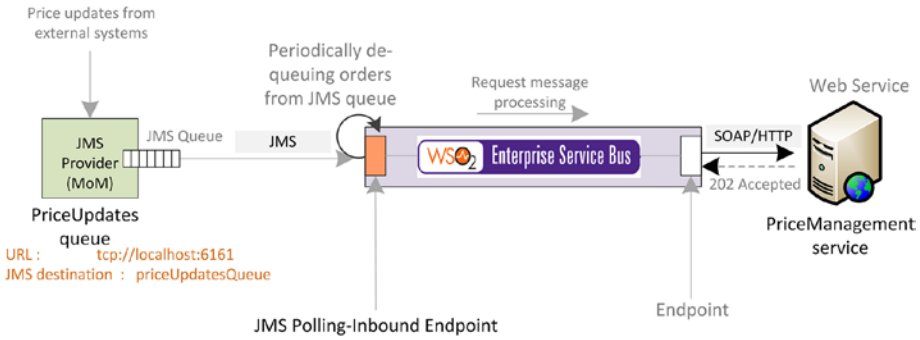

Figure 6-3. *JMS point-to-point messaging model in which ESB consumes messages in a JMS queue using a JMS inbound endpoint and sends the messages to a backend service*

As illustrated in Figure 6-3, you can use a JMS inbound endpoint in WSO2 ESB to consume message from a JMS provider and inject the message into a sequence where all the required mediation work is carried out. In this example, the messages that are de-queued from the ActiveMQ's `priceUpdatesQueue` and are delivered to the SOAP-based web service over HTTP protocol.

In the JMS inbound endpoint configuration, you can specify the details of the JMS provider and the other parameters that are required to consume the messages from the queue. Also, the JMS client libraries of the respective JMS provider need to be copied to `$ESB_HOME/repository/components/lib` prior to starting the server. Listing 6-1 shows the configuration of the JMS inbound endpoint and the message processing sequence. When the inbound endpoint is deployed on WSO2 ESB, it subscribes to the specified JMS queue on the JMS provider that's specified in the inbound endpoint parameters.

Listing 6-1. Using JMS Inbound Endpoint to Consume Messages from a Queue

```
<inboundEndpoint xmlns="http://ws.apache.org/ns/synapse"
name="JMSPriceUpdateConsumer"
sequence="priceUpdateProcessorSequence"        <!-- [1] -->
onError="priceUpdateErrorSeq"                  <!-- [2] -->
protocol="jms"                                 <!-- [3] -->
suspend="false">                               <!-- [4] -->
      <parameters>
         <parameter name="interval">1000</parameter>   <!-- [5] -->
         <parameter
name="transport.jms.Destination">priceUpdatesQueue
</parameter>                                           <!-- [6] -->
         <parameter name="transport.jms.CacheLevel">1</parameter>
         <parameter
name="transport.jms.ConnectionFactoryJNDIName">
QueueConnectionFactory
  </parameter>
         <parameter name="sequential">true</parameter>
         <parameter
name="java.naming.factory.initial">org.apache.activemq.jndi.
ActiveMQInitialContextFactory
</parameter>
         <parameter                                     <!-- [7] -->
name="java.naming.provider.url">tcp://localhost:6161
    </parameter>
         <parameter
name="transport.jms.SessionAcknowledgement">
AUTO_ACKNOWLEDGE</parameter>                           <!-- [8] -->
         <parameter
name="transport.jms.SessionTransacted">
false</parameter>                                      <!-- [9] -->
         <parameter
name="transport.jms.ConnectionFactoryType">queue</parameter>
      </parameters>                                    <!-- [10] -->
</inboundEndpoint>

<sequence name="priceUpdateProcessorSequence">
    <log level="custom">
      <property name="Message-Flow" value="PriceUpdate message received!"
    </log>
```

```
    <header name="Action" value="urn:getSimpleQuote"/>
    <call>
        <endpoint>
            <address                                    <!-- [11] -->
              uri="http://localhost:7070/services/PriceManagementService"
                    format="soap11"/>
        </endpoint>
    </call>

</sequence>
[1] Dequeued messages are injected this sequence.
[2] Error handling sequence.
[3] Specify protocol as 'jms' to configure a JMS Inbound Endpoint.
[4] suspend is set to false. Hence this is an active inbound endpoint.
[5] Polling interval.
[6] Name of the queue.
[7] JMS provider's url.
[8] Using auto acknowledge mode to ack the provider to dequeue the message
immediately after receiving.
[9] We are not using a transaction enabled JMS session.
[10] Since we are using point-to-point JMS messaging model, we have to
specify this as a 'queue'.
[11] Sending the message to a SOAP web service.
```

Since we used the point-to-point JMS messaging model, the messages are received in the same order that they are stored in the queue. The detail of the parameters that can be used with JMS inbound endpoint can be found at https://docs.wso2.com/display/ESB490/JMS+Inbound+Protocol.

In some JMS consumer scenarios, you may have to use the publish-subscribe JMS message model. For example, suppose that there is a weather data alert system that publishes weather alerts and the subscribers can receive those alerts. The weather alert publishing system can use a JMS provider to publish alerts and since there can be multiple subscribers, it needs to use the publish-subscribe JMS message model. As depicted in Figure 6-4, suppose that there are two web services that want to subscribe to weather alerts namely—`WeatherAlertReceiver-1` and `WeatherAlertReceiver-2`. But they both can receive messages on SOAP format over the HTTP protocol. Therefore, you can use WSO2 ESB to subscribe to the `weatherAlerts` JMS topic, receive weather alerts, and later send them to the weather alert receiver SOAP web services.

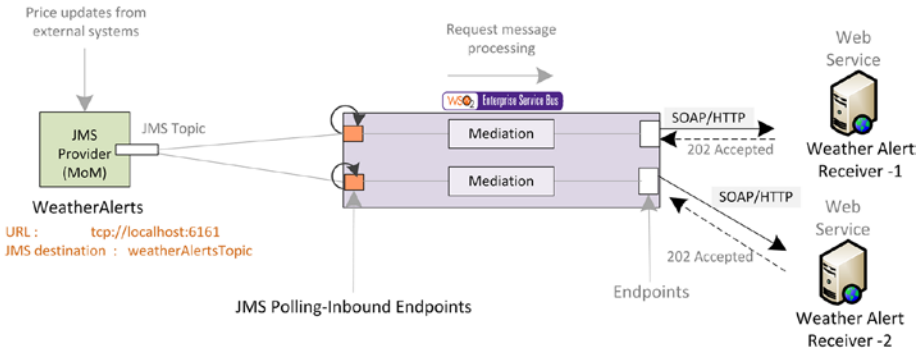

Figure 6-4. *JMS publisher-subscriber messaging model in which there are two JMS inbound endpoints subscribed to the same topic and the received weather alert messages are sent to two different services*

As shown in Figure 6-4, you can use two JMS inbound endpoints as two subscribers to the WeatherAlert JMS topic and when the weather alert system publishes message to the topic, both inbound endpoints will receive weather alert messages. In each JMS inbound endpoint, it can inject the message to a sequence, which sends the weather alert message to `WeatherAlertReceiver-1` and `WeatherAlertReceiver-2`.

Listing 6-2 shows the configuration of the two JMS inbound endpoints that are subscribed to the same topic to receive weather alerts. Any other external subscribers that are subscribed to the same topic will receive the same weather alert messages.

Listing 6-2. Using JMS Inbound Endpoints to Subscribe to a JMS Topic

```
<inboundEndpoint
        name="WeatherAlertsReceiver-1"
                sequence="weatherAlertProcessorSeq-1"
                onError="weatherAlertProcessorSeq-2"
                protocol="jms"
                suspend="false">
    <parameters>
       <parameter name="interval">1000</parameter>
       <parameter
                        name="transport.jms.Destination">weatherAlerts
                        </parameter>     <!-- [1] -->
       <parameter name="transport.jms.CacheLevel">5</parameter>
       <parameter
                name="transport.jms.ConnectionFactoryJNDIName">TopicConnect
                ionFactory
                </parameter>
       <parameter name="sequential">true</parameter>
       <parameter name="java.naming.factory.initial">
                org.apache.activemq.jndi.ActiveMQInitialContextFactory</
                parameter>
```

```
        <parameter
                name="java.naming.provider.url">
                        tcp://localhost:61616</parameter>
        <parameter
         name="transport.jms.SessionAcknowledgement">AUTO_ACKNOWLEDGE</
         parameter>
        <parameter name="transport.jms.SessionTransacted">false</parameter>
        <parameter
                        name="transport.jms.ConnectionFactoryType">topic</
                        parameter>  <!-- [2] -->
     </parameters>
  </inboundEndpoint>

   <inboundEndpoint
                name="WeatherAlertsReceiver-2"
                        sequence="weatherAlertProcessorSeq-2"
                onError="fault" protocol="jms" suspend="false">
      <parameters>
         <parameter name="interval">1000</parameter>
         <parameter
                        name="transport.jms.Destination">weatherAlerts</
                        parameter> <!-- [3] -->
         <parameter name="transport.jms.CacheLevel">5</parameter>
         <parameter
                        name="transport.jms.ConnectionFactoryJNDIName">
                        TopicConnectionFactory</parameter>
         <parameter name="sequential">true</parameter>
         <parameter
         name="java.naming.factory.initial">org.apache.activemq.jndi.
         ActiveMQInitialContextFactory</parameter>
         <parameter name="java.naming.provider.url">tcp://localhost:61616
         </parameter>
         <parameter name="transport.jms.SessionAcknowledgement">
         AUTO_ACKNOWLEDGE</parameter>
         <parameter name="transport.jms.SessionTransacted">false</parameter>
         <parameter name="transport.jms.ConnectionFactoryType">topic
         </parameter>
      </parameters>
   </inboundEndpoint>
<sequence name="weatherAlertProcessorSeq-1">                <!-- [4] -->
    <log level="custom">
      <property name="Message-Flow" value="Weather alert received -
      weatherAlertProcessorSeq-1"
    </log>
    <header name="Action" value="urn:postWeatherAlert"/>
    <call>
        <endpoint>
```

```
            <address
              uri="http://localhost:7070/services/WeatherAlertReceiver-1"
                    format="soap11"/>
        </endpoint>
    </call>
</sequence>

<sequence name="weatherAlertProcessorSeq-2">                <!-- [5] -->
    <log level="custom">
      <property name="Message-Flow" value="Weather alert received -
      weatherAlertProcessorSeq-2"
    </log>
    <header name="Action" value="urn:postWeatherAlert"/>
    <call>
        <endpoint>
            <address
              uri="http://localhost:7070/services/WeatherAlertReceiver-2"
                    format="soap11"/>
        </endpoint>
    </call>
</sequence>
[1] Inbound Endpoint – WeatherAlertsReceiver-1 is subscribed to
weatherAlerts topic.
[2] You can specify the connection factory type as a topic.
[3] Inbound Endpoint – WeatherAlertsReceiver-2 is also subscribed to the
same topic.
[4] sequence to process messages from inbound endpoint
WeatherAlertsReceiver-1
[5] sequence to process messages from inbound endpoint
WeatherAlertsReceiver-2
```

From each inbound endpoint, the weather alert message is sent to the respective sequences and each sequence is responsible for sending the message to the `WeatherAlertReceiver` services.

ESB as a JMS Producer

So far, you have learned about how WSO2 ESB can be used to consume messages in a JMS queue or topic (as JMS message consumer). Similarly, you may also have to produce messages to a JMS queue or topic from WSO2 ESB. In such scenarios, WSO2 ESB has to act as the JMS producer.

In this section, you will learn about how you can use WSO2 ESB to produce messages based on JMS API to an external JMS MoM. For example, suppose that a supermarket chain uses a JMS-based MoM system (WSO2 Message Broker) to store the price updates of each item that it sells. The price updates should be added to the JMS-based MoM with the SOAP message format, but the price update information comes through a HTTP/REST API with JSON message format. As depicted in Figure 6-5, you can use a REST API

in WSO2 ESB to expose a JSON-based HTTP interface to accept price updates and, in the mediation logic, you can convert the messages to the SOAP 1.1 format and store them in a JMS queue.

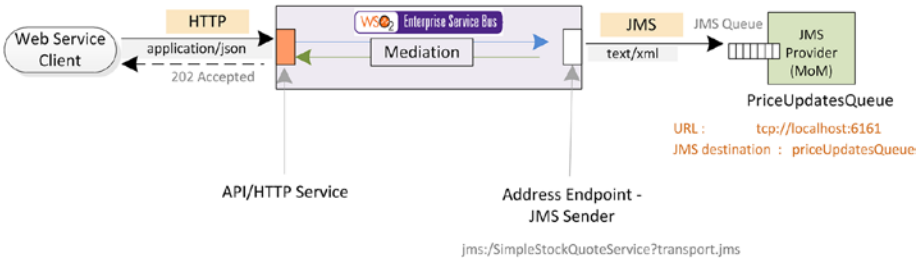

Figure 6-5. *Producing messages to a JMS queue using JMS sender*

As you learned in Chapter 5, the configuration of REST API converts the incoming REST-based messages to the required SOAP message format. The JMS producer related configuration comes into action when we send the messages through the address endpoint with the `jms:/..` protocol. When you use that, under the hood, it invokes the JMS Transport sender of WSO2 ESB (you need to uncomment and enable JMS Sender in `axis2.xml` prior to starting the server). You can find the relevant configuration in Listing 6-3, where you can give all the JMS related parameters as part of the JMS sender URL.

Listing 6-3. Using JMS Sender to Produce Messages to a JMS Queue

```
...
        <inSequence>
            <log level="full"/>
                <!-- [1] -->
            <property name="FORCE_SC_ACCEPTED" value="true" scope="axis2"/>
            <property name="OUT_ONLY" value="true"/>        <!-- [2] -->
            <call>
               <endpoint format="soap11" >        <!-- [3] -->
                   <address uri="jms:/PriceUpdatesQueue?transport.jms.Co
                   nnectionFactoryJNDIName=QueueConnectionFactory&amp;ja
                   va.naming.factory.initial=org.wso2.andes.jndi.Properti
                   esFileInitialContextFactory&amp;java.naming.provider.
                   url=repository/conf/jndi.properties&amp;transport.jms.
                   DestinationType=queue"
                                format="soap11"/>
               </endpoint>
            </call>
         </inSequence>
[1] You can specify this property to return HTTP 202 accept message
immediately after receiving the request.
[2] Since this is an asynchronous one way message, message flow doesn't need
to wait for a response. This flag is used to indicate it.
```

```
[3] When producing JMS messages, an address point can be used along with all
the required JMS connection parameters. Request is produced to JMS Broker
via JMS transport sender.
```

In this example, we have produced messages to a queue. Similarly you can publish messages to a topic by changing the destination type to a topic. As with the producer, you can produce messages to any JMS-compliant message broker/MoM product using JMS sender.

Two-Way JMS

In most of the messaging scenarios, JMS is used as the messaging API for asynchronous one-way messaging. But there can be messaging requirements to support request-response messaging semantics on top of JMS API. The two-way messaging with JMS can be implemented with JMS using another queue/topic to receive responses. JMS provides the `JMSReplyTo` message header field for specifying the destination (queue/topic) where a reply to a message should be sent. The `JMSCorrelationID` header field of the reply can be used to correlate the response with its original request. Therefore, when a JMS client sends a message to a JMS broker, it can also send the `ReplyTo` header along with the message, so that the broker can send back the response to the specified ReplyTo queue/topic. The JMS client can poll the response queue for the response messages.

In JMS two-way messaging scenarios, ESB can be used to expose a JMS interface to serve JMS request-response messaging scenarios or to send a message to an external JMS-based MoM and receive a response from it. To understand these scenarios further, let's consider the following example illustrated in Figure 6-6.

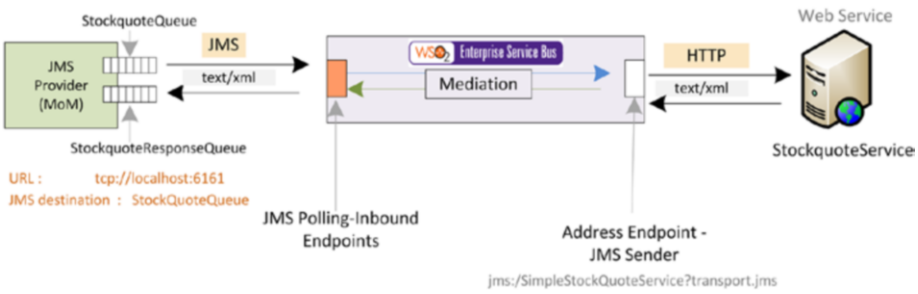

Figure 6-6. *JMS inbound endpoint polls the StockquoteQueue and injects the request to the mediation flow, which sends the response to the SOAP web service over HTTP. The response is sent back to the StockquoteResponseQueue.*

Suppose that a JMS client application places a stock quote request in a JMS queue (`StockquoteQueue`) and, along with the request, it sends the `JMSReplyTo` header, which specifies the queue (`StockquoteResponseQueue`), which will wait for the response message. But the backend service only supports SOAP web services over HTTP and it cannot directly accept requests from JMS MoM. That's where you can use ESB to fetch messages from the JMS queue (`StockquoteQueue`), send them to the SOAP web service, and place the response in the response queue (`StockquoteResponseQueue`).

You can use a JMS inbound endpoint in WSO2 ESB to pick up the message from the queue and send it to the SOAP backend service via the HTTP protocol. Since ESB receives the `JMSReplyTo` header with the response queue name, ESB knows where to send the response message. The configuration of the JMS inbound endpoint will be very similar to the previous JMS consumer scenario, but the response queue is read from the `JMSReaplyTo` header. If the client doesn't specify a response queue, you can configure `transport.jms.ReplyDestination` as part of the inbound configuration shown in Listing 6-4.

Listing 6-4. Using WSO2 ESB as a Two-Way JMS Consumer

```
<inboundEndpoint xmlns="http://ws.apache.org/ns/synapse"
                 name="JMSStockquoteTwowayConsumer"
                 sequence="stockquoteReqProcessorSeq"
                 onError="stockquoteReqErrorSeq"
                 protocol="jms"
                 suspend="false">
    <parameters>
        <parameter name="interval">1000</parameter>
        <parameter
                name="transport.jms.Destination">StockquoteQueue</parameter>
        ...
                <parameter
                  name="transport.jms.ReplyDestination">StockquoteResponseQ
                  ueue <!-- [1] -->
                </parameter>
        ...
    </parameters>
</inboundEndpoint>

<sequence name="stockquoteReqProcessorSeq">             <!-- [2] -->
    <log level="custom">
      <property name="Message-Flow" value="Stockquote Req-processor seq."
    </log>
    <header name="Action" value="urn:getQuote"/>
    <call>
        <endpoint>
            <address
              uri="http://localhost:7070/services/SimpleStockquoteService"
                     format="soap11"/>
        </endpoint>
    </call>
    <respond/>
</sequence>
[1] Configuring the response queue for sending the response message back
from inbound.
[2] Sequence that sends the request received from the JMS inbound endpoint
and sending the response back.
```

Now you have discovered how you can use JMS inbound endpoint to consume messages in a queue and send the response messages back to a response queue. In some JMS integration scenarios, you may need to do two-way messaging in the JMS producer side as well.

For example, let's consider the integration scenario depicted in Figure 6-7. In this scenario, the `stockquote` backend web service can only consume `stockquote` messages from a JMS queue (`StockquoteQueue`) and that service will place the `stockquote` responses in another JMS queue (`StockquoteResponseQueue`). But the client application is based on SOAP/HTTP and cannot directly talk to the backend service. Therefore, you can use the ESB to accept the request from the web service client over HTTP protocol and add the request to the JMS queue (`StockquoteQueue`). Since we need to receive the response via JMS into a separate queue, you can specify the `transport.jms.ReplyDestination` parameter when you are sending the request out via the JMS transport sender.

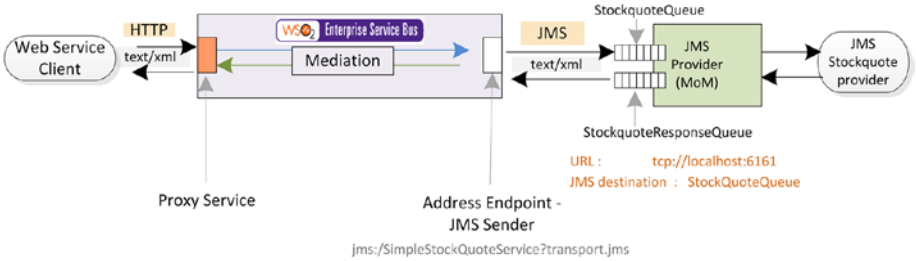

Figure 6-7. *ESB accepts the SOAP web service request over HTTP and sends that request to a JMS queue and specifies the JMS response queue. Once the response is available in the response queue, it is sent back to the web service client.*

The configuration of the two-way JMS producer scenario is similar to the previous JMS producer configuration; the only difference is that along with the `jms` endpoint URI, you need to specify the `transport.jms.ReplyDestination=StockquoteResponseQueue` parameter, as shown in Listing 6-5.

Listing 6-5. Using WSO2 ESB as a Two-Way JMS Producer

```
<proxy name="StockQuoteProxy" transports="http">
      <target>
          <inSequence>
              <property action="set" name="transport.jms.
ContentTypeProperty" value="Content-Type" scope="axis2"/>
              <call>
                 <endpoint format="soap11" >
                    <address uri="jms:/SimpleStockQuoteService?transp
                    ort.jms.ConnectionFactoryJNDIName=QueueConnection
                    Factory&amp;                    java.naming.factory.
                    initial=org.apache.activemq.jndi.ActiveMQInitialC
                    ontextFactory&amp;java.naming.provider.url=tcp://
```

```
                    localhost:61616&amp;transport.jms.DestinationType=que
                    ue&amp;transport.jms.ReplyDestination=StockquoteResponse
                    Queue"                          <!-- [1] -->
/>
                      format="soap11"/>
                  </endpoint>
               </call>
               <property
                                 action="remove" name="TRANSPORT_HEADERS"
                                 scope="axis2"/>                <!-- [2] -->
               <respond/>
            </inSequence>
      <target>
</proxy>
[1] Use transport.jms.ReplyDestination=StockquoteResponseQueue parameter to
specify the response queue.
[2] Removing all JMS transport header before sending the response back to
the web service client.
```

As you specified the response queue in the JMS connection URL, as soon as a new response is available, the sender returns the response to the mediation flow. In the mediation flow, you can remove all the JMS transport headers, as they are not required for the HTTP SOAP web service client.

So far we have discussed several use cases of integrating WSO2 ESB with JMS API-based enterprise messaging products. They are mainly based on the JMS inbound endpoint and JMS sender terminology and, with few simple variations in the configuration, you can implement these JMS integration scenarios. In such scenarios, you may also need to make sure JMS consuming or producing is done in a transactional manner. That's addressed in the JMS transaction specification and in the next section you learn how to use JMS transactions with WSO2 ESB.

Using JMS Transactions

JMS transactions are designed to ensure that the messages are received or sent in an all-or-nothing fashion. For example, when consuming messages in a JMS queue, when you are using a transactional consumption of messages, the JMS provider/server won't remove the messages from the queue if you sent a `commit()` message after processing the messages. If a failure occurs or a `rollback( )` is issued, then the JMS provider will attempt to redeliver the messages, in which case the messages will have the redelivered flag set. Similarly, you can use transactions at the JMS sender side too.

You can use JMS transaction with WSO2 ESB with both JMS consumer and JMS producer. In the following example, it shows a modified version of the supermarket scenario where the price update request must be consumed in transactional fashion. As illustrated in Figure 6-8, the JMS inbound endpoint is used as the transactional JMS consumer. The price update requests are fetched from the JMS queue and are processed and sent to the `PriceManagementService`. If the entire message processing and delivering is successful, then the consumer sends a commit request, which removes the message

from the JMS queue. But in case of an error, the fault sequence is invoked and inside the fault sequence, you can set the rollback flag, which will keep the messages in the queue and attempt to redeliver it.

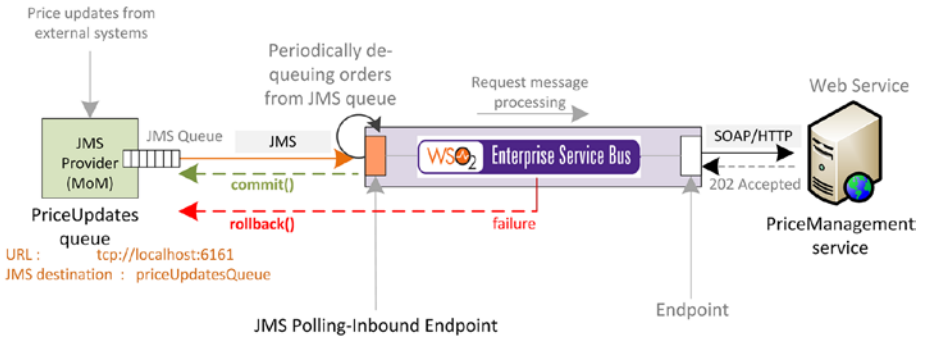

Figure 6-8. *Using JMS consumer-side transactions to get messages from a JMS queue, process them, and send to a backend service*

In order to configure a transactional JMS consumer scenario, you just need to enable the `SessionTransacted` property and add the rollback flag to the fault sequence, as shown in Listing 6-6. It is important to keep in mind that when you are using JMS transactions, you need to make sure you use Call mediator with the blocking mode to make sure that sending happens in the same thread, so that you have control over committing or rolling back the transaction.

Listing 6-6. Using JMS Consumer-Side Transactions with JMS Inbound Endpoints

```
<inboundEndpoint xmlns="http://ws.apache.org/ns/synapse"
                 name="JMSPriceUpdateConsumer"
                 sequence="stockquoteReqProcessorSeq"
                 onError="stockquoteReqErrorSeq"
                 protocol="jms"
                 suspend="false">
    <parameters>
        ...
        <parameter
                name="transport.jms.SessionTransacted">
            true</parameter>                              <!-- [1] -->
        <parameter name="transport.jms.SessionAcknowledgement">CLIENT_
ACKNOWLEDGE</parameter>
...
</inboundEndpoint>

...
...
<inSequence>
    <property name="OUT_ONLY" value="true"/>
```

```
    <call blocking="true">         <!-- [2] -->
        <endpoint>
            <address uri="http://localhost:8080/SimpleStockQuoteService"/>
        </endpoint>
    </call>

    <log level="custom">
      <property name="Transaction Action" value="Committed"/>
    </log>
</inSequence>
...

<sequence name=" stockquoteReqErrorSeq">
        <!-- [3] -->
    <property name="SET_ROLLBACK_ONLY" value="true" scope="axis2/>
    <log level="custom">
       <property name="Transaction Action" value="Rollbacked"/>
    </log>
</sequence>
[1] Configure the JMS Inbound consumer to use a transacted session and
session acknowledgement is set to client ack.
[2] Use call mediator with blocking mode true to make sure message sending
happens in the same thread.
[3] In the fault sequence, specify the rollback flag to revert the
transaction.
```

We used the `SessionAcknowledgement` to `CLIENT_ACKNOWLEDGE`. By default, JMS consumers use default acknowledgement mode, which is specified using the `AUTO_ACKNOWLEDGE` parameter. `CLIENT_ACKNOWLEDGE` parameters give consumers more control over when messages are acknowledged. A consumer can group a number of messages, and then invoke the acknowledge method of the message to instruct the JMS provider that the message and all other messages received until this point have been consumed.

Therefore, if something goes wrong in the message processing, the messages won't be removed from the queue and will be ready to be redelivered.

Store and Forward with Message Stores and Message Processors

When passing messages between two systems, there can be several constraints that prevent a smooth process. For example, when the message sender is ready to send messages, the recipient may be offline or the message consumption rate of the recipient may be very different from the sender. In such scenarios, we need to store messages in between the sender and the recipient and allow the message sender and recipient to work in full asynchronous manner. This messaging pattern is known as *store and forward messaging*.

The store and forward pattern is used to store incoming messages and later forward them to the intended recipient. The main advantage of this approach is that it allows you to send messages asynchronously and reliably to backend services. These messages can be stored in any reliable storage, such as databases and file systems, and a combination of both.

In WSO2 ESB, the store and forward messaging pattern is implemented by using two main message constructs known as message store and message processors. To understand the implementation of store and forward pattern with WSO2 ESB, let's consider the same supermarket software solution scenario. Here we have a web service client that sends the price updates to a price management service. The price updates are sent in ordered-fashion and hence the in-order delivery must be maintained across the messaging system. But the availability of the price management service is not guaranteed. And we have a strict requirement of not losing any price update and at the same time the order of the price update messages must be preserved. Now our challenge is to implement this scenario with WSO2 ESB. As depicted in Figure 6-9, you can use WSO2 ESB's message store and processor to realize this integration scenario.

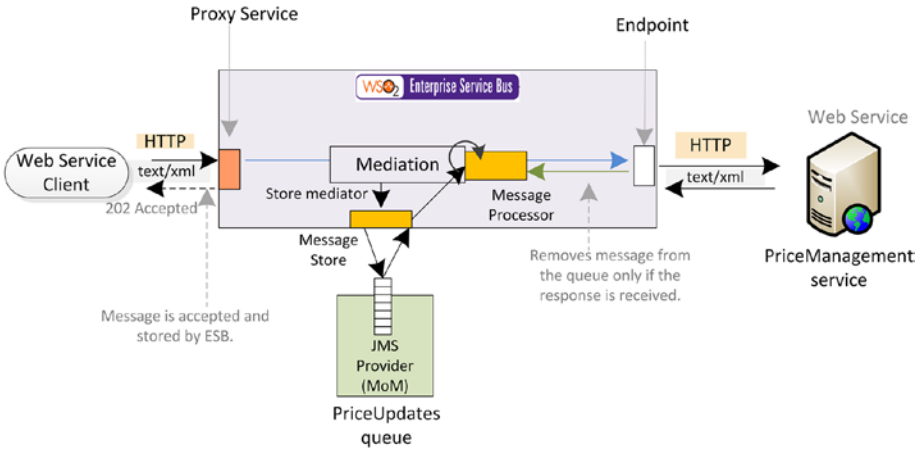

Figure 6-9. *Guaranteed message delivery between price update client and PriceManagement service by using message stores and processors*

In WSO2 ESB, you can define a message store that can be used to store the messages. But message store represents a virtual message store, which points to an actual message storage such as a JMS queue or a database table. There are multiple implementations of message stores and in this example we used a JMS message store. JMS message store can use any JMS provider as the underlying message storage and, when configuring the message store, you can provide all the connection details to the external JMS provider. In Listing 6-7, it shows the configuration of a JMS message store.

Listing 6-7. Defining a JMS Message Store

```
<messagestore
                class="org.apache.synapse.message.store.impl.jms.JmsStore"
                name="JMSMS">       <!-- [1] -->
   <parameter       <!-- [2] -->
        name="java.naming.factory.initial">
                org.apache.activemq.jndi.ActiveMQInitialContextFactory
                </parameter>
   <parameter
                name="java.naming.provider.url">tcp://localhost:61616
                </parameter>
   <parameter
                name="store.jms.JMSSpecVersion">1.1</parameter>
</messagestore>

[1] Name of the message store and the implementation of the message store
[2] Parameters required for JMS message store which connects to an external
JMS provider.
```

Once this JMS message store is deployed into ESB, it creates a queue in the JMS broker (in this case, ActiveMQ) with the name of the message store (JMSMS). Now you can add messages to the message flow from the mediation flow by using the `Store` mediator. In Listing 6-8, you can find a proxy service that accepts messages from a SOAP web service client and adds the received requests to the message store that we created previously.

Listing 6-8. Adding Messages to a Message Store from Mediation Flow

```
<proxy
                name="PriceUpdateManagerProxy"
                transports="http https" startonload="true" trace="disable">
   <target>
      <insequence>
         <property
                        name="FORCE_SC_ACCEPTED" value="true"
                                scope="axis2" type="STRING"></property>
                                <!-- [1] -->
         <property
                        name="OUT_ONLY" value="true"     <!-- [2] -->
                                scope="default" type="STRING"></property>
         <store messagestore="JMSMS"></store>            <!-- [3] -->
      </insequence>
   </target>
</proxy>
```

```
[1] This is to configure the proxy service to accepts the request and
immediately send back the HTTP 202 Accepted message.
[2] Since this is an one-way messaging scenario, you need to specify this
flag.
[3] Using store mediator, you can store the message in the JMS message store
which will add the current request to the JMS queue.
```

Now at this point, you have added the received request to the JMS message store and replied to the HTTP web service client with a HTTP 202 Accepted response. And all messages now reside in the JMS queue. When it comes to consuming messages stored in a message store, you can use the message processor. The message processor is used to consume messages from the message store and then deliver them to a configured backend. When a message processor sends messages to the backend, we have more control over the message sending part. For example, we can throttle the speed of the delivery of messages or we can reliably deliver messages to a configured backend.

Based on the nature of messaging sending from a message processor, there are two types of message processors available in WSO2 ESB.

- *Forwarding processor:* This is mainly used in guaranteed message delivery scenarios. Forwarding processor gets the messages from a message store and sends them to the configured endpoint. If the message is successfully delivered to the endpoint, then it's removed from the message store. If it's not delivered, it keeps on retrying the message sending based on the configured retry interval.
- *Sampling processor*: This is useful in rate matching scenarios where the producing rate of the client and the consuming rate of the consumer are different. This has better performance than the forwarding processor, but does not ensure guaranteed delivery.

In our price update scenario, we need to use the forwarding message processor, as we require the guaranteed delivery of the messages. Since we are using a queue for JMS store, it also ensures the in-order delivery. Therefore, you can configure a forwarding message processor as shown in Listing 6-9. In the forwarding message processor configuration, you can select the message store that it should fetch messages with and the target endpoint name.

Listing 6-9. Processing Messages in a Message Store Using the Forwarding Message Processor

```
<messageprocessor
        class="org.apache.synapse.message.processor.impl.forwarder.
ScheduledMessageForwardingProcessor"                        <!-- [1] -->
               name="PriceUpdateMP"
               targetendpoint=" PriceMgmtServiceEP"     <!-- [2] -->
                messagestore="JMSMS">                   <!-- [3] -->
   <parameter name="client.retry.interval">1000</parameter>  <!-- [4] -->
   <parameter name="interval">1000</parameter>              <!-- [5] -->
```

```
   <parameter name="is.active">true</parameter>
</messageprocessor>

<endpoint name="PriceMgmtServiceEP">                          <!-- [6] -->
   <address uri="http://localhost:7000/services/PriceManagementService">
                        </address>
</endpoint>
[1] Configure a forwaring message processor
[2] Name of the target endpoint which contains the backend service address.
[3] Associated message store instance.
[4] Duration between each message retry in case of a failure.
[5] Interval in milliseconds in which processor consumes messages
[6] Endpoint definition of the backend service.
```

You have seen how you can realize the store and forward messaging pattern with the use of message stores and processors using JMS-based persistent message store. But the usage of message stores and processors is not limited to the JMS protocol. You can also use other message store implementations such as JDBC message store, where you can configure a database to store messages from the mediation flow.

When implementing guaranteed delivery scenarios with message stores and processors, you may have to use failover capability at the message store level as well. The original message store may fail due to network failure, message store crash, or system shutdown for maintenance, and the failover message store is used as the solution for the original message store failure. So with the failover message store concept, the store mediator sends messages to the failover message store. Then, when the original message store is available again, the messages that were sent to the failover message store need to be forwarded to the original message store. The scheduled failover message forwarding processor is used for this purpose. The details of how you can use failover message store can be found at http://docs.wso2.com/enterprise-service-bus/Guaranteed+Delivery+with+Failover+Message+Store+and+Scheduled+Failover+Message+Forwarding+Processor.

Integrating with AMQP, MQTT and Kafka

So far we discussed enterprise messaging using JMS API. But there are quite a lot of other enterprise messaging standards and protocols that are increasingly popular. Therefore, in this section, you will discover some of the common enterprise messaging technologies that are popular in the enterprise integration space.

Using AMQP with RabbitMQ

With JMS, you can replace any JMS compliant message broker with any other JMS compliant broker. That's one of the key benefits of using JMS API. But as the name implies, JMS is inherently for Java applications. Therefore, in other words, JMS only allows us to have messaging interoperability within the Java platform, but not allow for cross-platform interoperability.

The main objective of AMQP (Advanced Message Queuing Protocol) is to enable cross-platform interoperability for enterprise messaging. For example, Figure 6-10 illustrates how JMS- and AMQP-based systems interoperate.

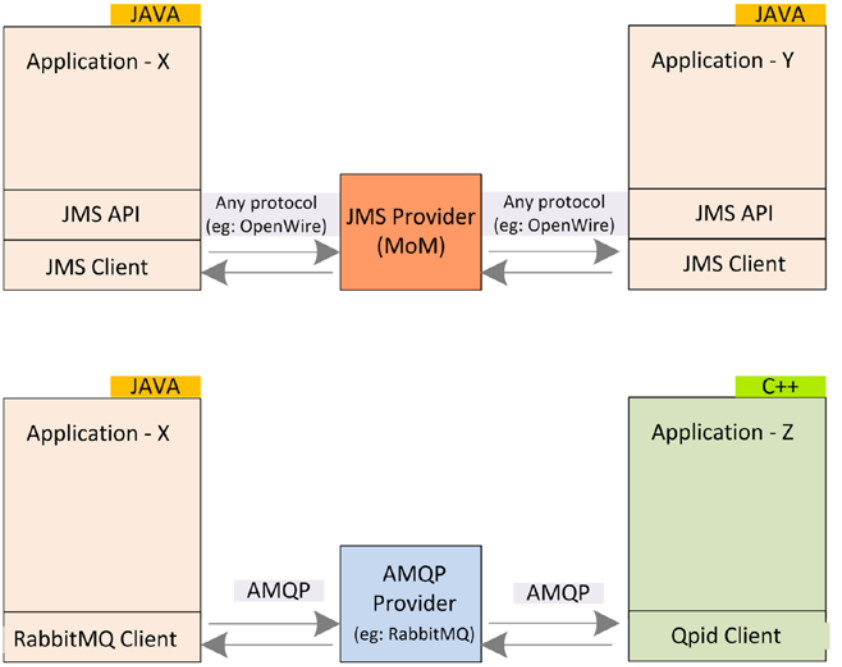

Figure 6-10. *JMS versus AMQP. With JMS, applications X and Y are implemented with Java and with AMQP, applications X and Y are implemented with Java and C++, respectively*

As you saw in the previous section, with JMS you can connect to any JMS compliant message broker/MoM using its client libraries via JMS API. But both applications need to be implemented using the Java platform. But with AMQP, you can connect an application based to any platform, as long as it is AMQP compliant. In Figure 6-10, you can observe a Java-based application connects to an AMQP provider (RabbitMQ) using its client libraries (RabbitMQ). Similarly, another application, which is based on C++ can connect to AMQP provider using its AMQP clients (Qpid).

AMQP provides a specification for an industry standard wire-level binary protocol to describe how the message should be structured and sent across the network. With AMQP, you can use whatever AMQP-compliant client library you want and any AMQP-compliant broker you want.

Now let's see how you use integrate enterprise-messaging systems, which are based on AMQP using WSO2 ESB. As an example, consider a lightly modified version of the supermarket price update scenario. Supposed that the price update requests are stored in a global message broker—RabbitMQ, which is based on AMQP. And price updates

requests should be fetched into each local RabbitMQ broker, which resides in each outlet. To realize this integration scenario, you can use WSO2 ESB as the AMQP consumer and producer, as shown in Figure 6-11.

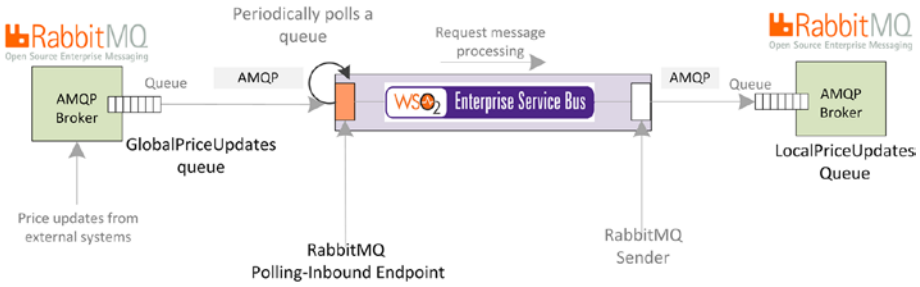

Figure 6-11. *RabbitMQ inbound endpoint can consume messages from the global price updates system and then add the messages into the local price updates system using RabbitMQ AMQP sender*

For the consumer side, you can use the RabbitMQ inbound endpoint which can consume messages from the `GlobalPriceUpdates` queue. The RabbitMQ sender can be used to send messages to the `LocalPriceUpdatesQueue` using the AMQP protocol. Since we are using AMQP, the global or local brokers or MoMs can be replaced with any AMQP-compliant broker. The configuration of this scenario (shown in Listing 6-10) is very similar to the JMS producer consumer scenarios that you learned about in the previous section.

Listing 6-10. AMQP Consumer and Producer Scenario with RabbitMQ

```
<inboundEndpoint xmlns="http://ws.apache.org/ns/synapse"
    name="RabbitMQConsumer"
    sequence="amqpPriceUpdateProcessorSeq"        <!-- [1] -->
    onError="amqpPriceUpdateErrorSeq"
    protocol="rabbitmq"                           <!-- [2] -->
    suspend="false">
      <parameters>
          <parameter name="interval">1000</parameter>  <!-- [3] -->
          <parameter name="rabbitmq.server.host.name">localhost</parameter>
          <!-- [4] -->
          <parameter name="rabbitmq.server.port">5672</parameter>
          <!-- [5] -->
          <parameter
                       name="rabbitmq.factory.recovery.interval">10000
                       </parameter>
          <parameter
                       name="rabbitmq.queue.name">GlobalPriceUpdatesQueu
                       </parameter> <!–[6] -->
```

```
          <parameter name="rabbitmq.exchange.name">exchange</parameter>
          <parameter
                       name="rabbitmq.connection.factory">
                              AMQPConnectionFactory</parameter>
      </parameters>
</inboundEndpoint>

<sequence name="amqpPriceUpdateProcessorSeq">        <!-- [7] -->
    <property name="FORCE_SC_ACCEPTED" value="true" scope="axis2"/>
        <property name="OUT_ONLY" value="true"/>
    <call>
     <endpoint>
        <address                                     <!-- [8] -->
        uri="rabbitmq:/AMQPProxy?rabbitmq.server.host.
        name=localhost&amp;rabbitmq.server.port=5672&amp;rabbitmq.queue.
        name=LocalPriceUpdates&amp;rabbitmq.queue.route.
        key=route&amp;rabbitmq.exchange.name=exchange"/>
     </endpoint>
    </call>
</sequence>
[1] Message processing sequence.
[2] Specifies the protocol as rabbitmq
[3] Polling interval
[4] Host name of the RabbitMQ server (AMQP server)
[5] Port of the RabbitMQ server
[6] Consumer destination points to GlobalPriceUpdates queue.
[7] Sequence that process the messages fetched from GlobalPriceUpdates
queue.
[8] RabbitMQ sender to produce messages to LocalPriceUpdates queue.
```

The configuration parameters are very similar to JMS apart from few AMQP/RabbitMQ specific configurations. There are several other parameters that you can use with either the RabbitMQ Inbound endpoint or RabbitMQ sender. You can find the details of each configuration parameters at https://docs.wso2.com/display/ESB490/WSO2+Enterprise+Service+Bus+Documentation.

As discussed earlier, the AMQP-based messages provide several benefits over JMS, such as the freedom of using any AMQP-compliant MoM with a single client library and the performance improvement of being a wire level protocol. Hence it's getting very popular in modern enterprise messaging standards. In the next section, you will discover another messaging standard—MQTT, which is very popular in the mobile and IoT (https://en.wikipedia.org/wiki/Internet_of_Things) space.

Integrating with MQTT

MQTT (Message Queuing Telemetry Transport) is a machine-to-machine (M2M)/"Internet of Things(IoT)" connectivity protocol. It was designed as an extremely lightweight publish/subscribe messaging transport. It is useful for connections with remote locations, where a small code footprint is required and/or network bandwidth is at a premium.

So, MQTT is all about IoT and devices, but why do we care about using an ESB with MQTT? In many IoT scenarios, it is often required to integrate conventional enterprise applications with MQTT-based systems and devices.

Let's take an example where we use WSO2 ESB as the MQTT consumer. Suppose that there is a temperature sensor the collects hourly temperature information and publishes them to a MQTT broker such as Mosquitto (`http://mosquitto.org/`). And there is a RESTful web service, called WeatherService, that needs to be updated on the latest temperature information sensed from the sensors. As shown in Figure 6-12, you can use WSO2 ESB to subscribe to the MQTT topic: `TemperatureData` from a MQTT Inbound endpoint. When there is new data published on that topic, it is fed into ESB. ESB takes care of sending that information to the WeatherService via HTTP/JSON.

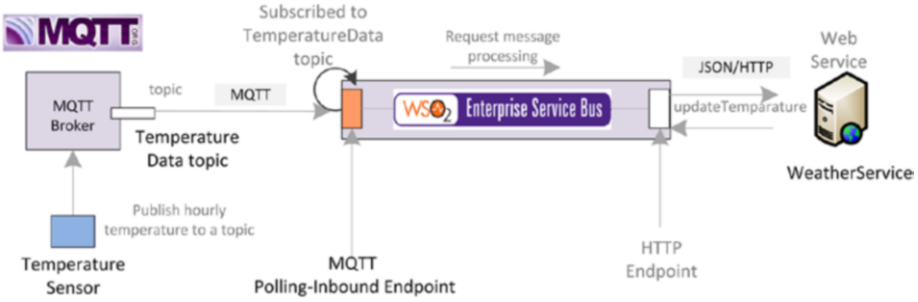

Figure 6-12. *Temperature data is published into a MQTT topic by the temperature sensor. An inbound MQTT endpoint is subscribed to that topic and it receives temperature updates to the mediation flow of WSO2 ESB. Then ESB sends that temperature data to the weather service over HTTP.*

The structure of the configuration of this MQTT inbound endpoint is again similar to any JMS/AMQP configuration, but contains a few MQTT protocol specific parameters. Listing 6-11 shows the configuration of the MQTT inbound endpoint.

Listing 6-11. Using MQTT Inbound Endpoint to Subscribe and Receive Messages from a MQTT Topic

```
<inboundEndpoint name="MQTTTemperatureDataConsumer"
          onError="TemperatureErrorSeq" protocol="mqtt"    <!-- [1] -->
      sequence="TemperatureDataProcessor" suspend="false">
      <parameters>
          <parameter name="mqtt.connection.factory">mqttFactory</parameter>
```

```
            <parameter name="mqtt.server.host.name">localhost</parameter>
            <!-- [2] -->
            <parameter name="mqtt.server.port">1886</parameter> <!-- [3] -->
            <parameter name="mqtt.topic.name">TemperatureData</parameter>
            <!-- [4] -->
            <parameter name="mqtt.subscription.qos">2</parameter>
            <parameter name="mqtt.client.id">client-id.143094112027
            </parameter>
            <parameter name="content.type">application/json</parameter>
            <!-- [5] -->
            <parameter name="mqtt.session.clean">false</parameter>
            <parameter name="mqtt.ssl.enable">false</parameter>
            <parameter name="mqtt.temporary.store.directory">store</parameter>
            <parameter name="mqtt.blocking.sender">false</parameter>
        </parameters>
</inboundEndpoint>
[1] Using MQTT as the protocol of the inbound endpoint.
[2] The host name of the MQTT broker.
[3] Port of the MQTT broker.
[4] The topic that the ESB inbound endpoint has subscribed to.
[5] Message format of the messages from the topic.
```

Similarly, ESB may be useful in scenarios where you need to publish MQTT messages to a topic. For examples, the scenario illustrated in Figure 6-13, the news alerts must be sent to all the mobile devices that are subscribed to it. In order to minimize the power consumption of the devices, they receive news alerts by subscribing to an MQTT topic. The REST web service client only knows JSON and sends the news alerts updates via HTTP. So, the task of ESB is to publish the received new alerts to the MQTT topic to which the mobile devices are subscribed.

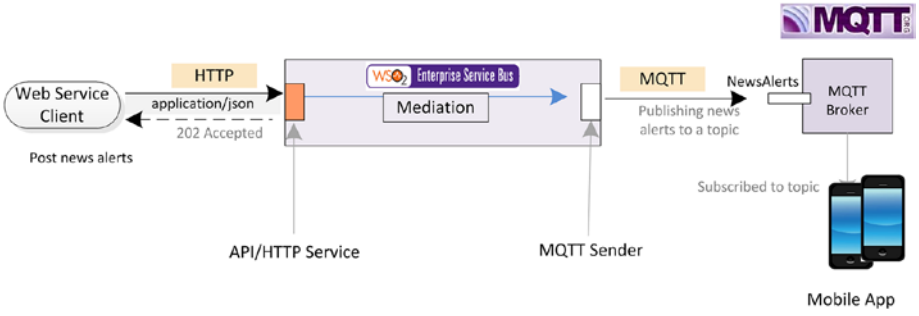

Figure 6-13. *ESB receives weather alerts via HTTP as JSON messages and ESB sends the message to a MQTT topic. All the other subscribers will receive it immediately*

To publish the content to the `NewsAlerts` topic, you can simply use the MQTT sender, as shown in Listing 6-11.

Listing 6-12. Publishing Messages to a MQTT Topic

```
...
<call>
    <endpoint>
        <address
uri="mqtt:/sender?mqtt.server.host.name=localhost&amp;mqtt.server.
port=1884&amp;mqtt.client.id=esb.test.sender&amp;mqtt.topic.
name=NewsAlerts&amp;mqtt.subscription.qos=2&amp;mqtt.blocking.sender=true"/>
    </endpoint>
</call>
...
```

Along with this MQTT sender, you can use the same mediation logic, which you used to do one-way messaging in previous scenarios. You can find more details about the MQTT related configuration parameters at http://docs.wso2.com/enterprise-service-bus.

So far you learned about a couple of popular messaging standards—AMQP and MQTT. It's noteworthy to mention that these two standards address two drastically different application domains, hence you should choose which approach suits you the most. With respective ESB, both these standards are equally supported. Apart from those two standards, there are a few other popular enterprise messaging standards such as Kafka.

Integrating with Kafka

Kafka is a distributed messaging system providing fast, highly scalable, and redundant messaging through a publisher-subscriber pattern. Message producers can write data to topics and consumers can read from topics. These topics are partitioned and replicated across multiple nodes and that makes Kafka a fully distributed message broker.

Kafka allows a large number of permanent or ad hoc consumers and supports high availability and automatic node recovery from failures. In real-world data systems, these characteristics make Kafka an ideal fit for communication and integration between components of large-scale data systems. The key difference between Kafka and other message brokers that use JMS, AMQP etc., is that Kafka is optimized for ordered publish subscribe, while the traditional brokers have a big feature set, which is rarely used but degrades performance. WSO2 ESB supports both message consuming capability and message producing capability with the Apache Kafka messaging system.

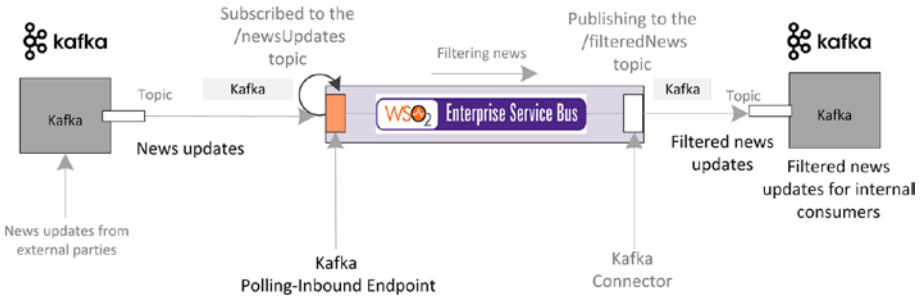

Figure 6-14. *Using WO2 ESB as Kafka consumer and producer*

WSO2 ESB provides a Kafka inbound endpoint that allows you to subscribe to a Kafka topic and inject the received messages to a sequence. This is similar to all other inbound endpoint that we have discussed so far. Listing 6-13 shows a sample configuration of the inbound endpoint to receive messages from a Kafka topic. Also, WSO2 ESB can be used to publish messages to a Kafka topic. The publishing can be done inside a ESB sequence using a Kafka connector. You can use WSO2 ESB's Kafka connector and initiate that inside an ESB message sequence. Once the initiation is done, you can publish messages to a given topic. The details related to ESB connector concept are discussed in Chapter 8.

Listing 6-13. Subscribing to a Kafka Topic from WSO2 ESB Using Kafka Inbound Endpoint, Filtering Messages Received for that Topic, and then Publishing Filtered Messages to Another Kafka Topic Through Kafka Connector

```
<inboundEndpoint xmlns="http://ws.apache.org/ns/synapse"
                 name="KakfaListenerEP"
                 sequence="requestHandlerSeq"
                 onError="inFaulte"
                 protocol="kafka"
                 suspend="false">
   <parameters>
      <parameter name="interval">100</parameter>
      <parameter name="coordination">true</parameter>
      <parameter name="sequential">true</parameter>
      <parameter name="zookeeper.connect">localhost:2181</parameter>
      <parameter name="consumer.type">highlevel</parameter>
      <parameter name="content.type">application/xml</parameter>
      <parameter name="topics">test,sampletest</parameter>
      <parameter name="group.id">test-group</parameter>
   </parameters>

...
   <inSequence>
          <kafkaTransport.init>
             <brokerList>localhost:9092</brokerList>
```

```
        </kafkaTransport.init>
        <kafkaTransport.publishMessages>
           <topic>nasdaq</topic>
        </kafkaTransport.publishMessages>
    </inSequence>
```

The enterprise messaging capabilities of WSO2 ESB can be used for various use cases and you can choose the message broker system that fits your use cases. Each messaging system offers a wide range of parameters that you can configure at the ESB inbound endpoint level or the transport sender/connector level. The details of all these parameters can be found at http://docs.wso2.com/enterprise-service-bus.

Summary

In this chapter you learned:

- The basics of Enterprise Messaging/Message Oriented Middleware (MoM).
- Using JMS, you can build interoperable Java application that can use any JMS-compliant MoM to interoperate with other applications.
- JMS is an API that allows you to write message broker agnostic code and your application can interact with any JMS-compliant MoM.
- You can use JMS inbound endpoint to consume messages from a queue or topic. Similarly, JMS sender can be used to produce messages to a JMS queue/topic.
- JMS is primarily designed for one-way asynchronous messaging but by using a separate queue to handle response (`JMSReplyTo` header), you can implement JMS request-response scenarios.
- JMS inbound endpoint can support two-way JMS and if the client doesn't send the `JMSReplyTo` header then you can specify the response queue name as part of the JMS inbound configuration.
- Similarly, with the JMS sender you can specify the reply destination and do two-way JMS on the producer sides.
- The store-and-forward message pattern is heavily used in enterprise integration space.
- WSO2 ESB provides two main constructs—message stores and processors. Message store is a logical representation of actual message storage and often uses JMS for JMS message store configuration. The forwarding processor is used for implementing guaranteed delivery scenarios and a sampling processor is used in controlling the message-processing rate.

- Apart from JMS, other messaging technologies such as AMQP and MQTT are also popular.
- AMQP tries to define a messaging standards, which is platform agnostic. MQTT is designed for device, machine-to-machine communication and often used in scenarios that have low power consumption and low bandwidth.
- Integration of both AMQP and MQTT is done using inbound endpoint for the consumer side and AMQP/MQTT senders for the producer side.

CHAPTER 7

File-Based Integration

Managed File Transfer has been around ever since the early stages of enterprise integration where one application writes a file that the other application reads and vice versa. The file formats, standards, location, privileges, and read/write coordination must be negotiated between the two applications beforehand. There are various message formats, such as comma-separated values (CSV), electronic data interchange (EDI), electronic protected health information (ePHI), etc., that are frequently used in this domain.

Software applications that are built on top of file transfer techniques have to be integrated with other systems that run on different protocols such as HTTP or other systems that use different file formats. Hence, ESB has to cater to a wide range of integration scenarios related to files, which is known as file-based integration.

In this chapter, you will learn about various use cases related to file-based integration with WSO2 ESB.

Reading Files

Reading files residing on a file system is one of the basic requirements of file-based integration. The files may reside in various file systems such as local, FTP, SFTP, and so forth. The file reading action may also trigger based on different requirement of the application. For instance, file reading can be a polling job (which polls for available files in a particular location in a file system) or an on-demand (access the file system and read the file content whenever the ESB needs content) job. Therefore, ESB has to support these different requirements in the file-based integration.

Reading a File from the Local File System

Assume that you want to build an integration scenario that polls to the file system and when there are new files in the specific file location, you read the contents of the file and write the content to the log. As depicted in Figure 7-1, you can implement this use case with the use of a File Inbound endpoint, which can poll the file system periodically and, when there's a new file in the file system location, it can inject the contents of the file as a message into a sequence. The sequence can apply arbitrary message mediation logic on top of that message.

K. Indrasiri, *Beginning WSO2 ESB*, DOI 10.1007/978-1-4842-2343-7_7

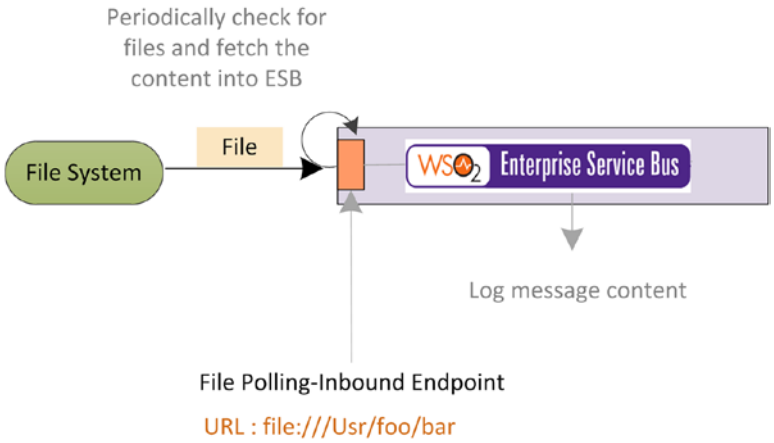

Figure 7-1. *File Inbound endpoint can pool the files in a given file system location. When there is a new file found in that location, the contents of the file is read and injected into a sequence. The sequence logs the contents of the file.*

With File Inbound endpoints, after processing the files, it moves them to a specified location or deletes them. Note that files cannot remain in the source directory after processing or they will be processed again. So if you need to maintain these files or keep track of the files that are processed, specify the option to move them instead of deleting them after processing.

Listing 7-1 shows the configuration of a File Inbound endpoint for this integration use case. As the main configuration elements of the File Inbound Endpoint, you can specify the message injecting sequence and error handling sequence. As the parameters of the inbound endpoint, you can specify the polling interval, location for processed files, content type of the created message payload and so forth.

Listing 7-1. Polling Files on the File Systems

```
<inboundEndpoint xmlns="http://ws.apache.org/ns/synapse"
                 name="OrderProcessing_File_Inbound_SFTP"
                 sequence="orderProcessingSeq"
                 onError="fault"
                 protocol="file"
                 suspend="false">
   <parameters>
      <parameter name="interval">1000</parameter>
      <parameter name="coordination">true</parameter>
      <parameter name="transport.vfs.ContentType">text/xml</parameter>
      <parameter name="transport.vfs.LockReleaseSameNode">false</parameter>
      <parameter name="transport.vfs.AutoLockRelease">false</parameter>
      <parameter name="transport.vfs.ActionAfterFailure">DELETE</parameter>
      <parameter name="transport.vfs.CreateFolder">true</parameter>
      <parameter name="sequential">true</parameter>
```

```
        <parameter name="transport.vfs.ActionAfterProcess">MOVE</parameter>
        <parameter name="transport.vfs.FileURI">/Users/kasun/development/
        deployment/maestro/wso2esb-4.9.0/sample_resources/file_reading/
        source</parameter>
        <parameter name="transport.vfs.DistributedLock">false</parameter>
        <parameter name="transport.vfs.Streaming">false</parameter>
        <parameter name="transport.vfs.MoveAfterProcess">/Users/kasun/
        development/deployment/maestro/wso2esb-4.9.0/sample_resources/file_
        reading/destination</parameter>
        <parameter name="transport.vfs.Locking">enable</parameter>
        <parameter name="transport.vfs.FileSortAscending">true</parameter>
        <parameter name="transport.vfs.FileSortAttribute">NONE</parameter>
        <parameter name="transport.vfs.Build">false</parameter>
    </parameters>
</inboundEndpoint>
```

■ **Note** It is important to note that, before introducing VFS inbound endpoints, WSO2 ESB supported file-based integration with proxy services with VFS transport. Therefore, if you are still using VFS transport, the same parameters can be used with file integration. However, if you need to support clustering/coordinating, it is recommended that you use VFS file inbound endpoint.

Failure Tracking

To track failures in file processing that can occur when a resource becomes unavailable, the VFS transport creates and maintains a failed records file. This text file contains a list of files that failed to process. When a failure occurs, an entry with the failed filename and timestamp is logged in the text file. When the next polling iteration occurs, the VFS transport checks each file against the failed records file, and if a file is listed as a failed record, it will skip processing and schedule a move task to move that file.

Similar to reading the files from the local machine, you can also read files from other file system types.

Reading Files from an FTP or FTP/s

Suppose the scenario in Figure 7-1 uses a file that's residing in an FTP location. In that case you can configure a File Inbound Endpoint in WSO2 ESB to poll and read files from a remote FTP server. The main difference is that the `FileURI` parameter that now contains the location of the FTP server and the `vfs.passive=true` parameter, which allows the FTP connection to stay open (once connected) and transfer a fairly large file.

Listing 7-2. Polling Files on the File Systems

```
<inboundEndpoint xmlns="http://ws.apache.org/ns/synapse"
                 name="OrderProcessing_File_Inbound_FTP"
                 sequence="orderProcessingSeq"
                 onError="fault"
                 protocol="file"
                 suspend="true">
   <parameters>
      <parameter name="interval">15</parameter>
      <parameter name="coordination">true</parameter>
      <parameter name="transport.vfs.ContentType">text/xml</parameter>
      <parameter name="transport.vfs.LockReleaseSameNode">false</parameter>
      <parameter name="transport.vfs.AutoLockRelease">false</parameter>
      <parameter name="transport.vfs.ActionAfterFailure">DELETE</parameter>
      <parameter name="transport.vfs.CreateFolder">true</parameter>
      <parameter name="sequential">true</parameter>
      <parameter name="transport.vfs.ActionAfterProcess">MOVE</parameter>
      <parameter name="transport.vfs.FileURI">vfs:ftp://kasun:password@ftp.
      server.org/test?vfs.passive=true</parameter>
      <parameter name="transport.vfs.MoveAfterProcess"> vfs:ftp://
      kasun:password@ftp.server.org/processed?vfs.passive=true </parameter>
      <parameter name="transport.vfs.DistributedLock">false</parameter>
      <parameter name="transport.vfs.Streaming">false</parameter>
      <parameter name="transport.vfs.Locking">enable</parameter>
      <parameter name="transport.vfs.FileSortAscending">true</parameter>
      <parameter name="transport.vfs.FileSortAttribute">NONE</parameter>
      <parameter name="transport.vfs.Build">false</parameter>
   </parameters>
</inboundEndpoint>
```

The rest of the parameters are more or less common to all file system types.

If the access to the FTP server is only through an SSL connection, then you need to configure File Inbound Endpoint with FTPs. Again you can simply change the URL to `ftps://` along with the username and password. In addition, you can configure SSL parameters if you are using FTP/s with certificates.

Listing 7-3. File URI of an FTPS with SSL

```
<parameter name="transport.vfs.FileURI">vfs:ftps://test:test123@10.200.2.63/
vfs/in?vfs.ssl.keystore=/home/user/openssl/keystore.jks&amp;vfs.
ssl.truststore=/home/user/openssl/vfs-truststore.jks&amp;vfs.ssl.
kspassword=importkey&amp;vfs.ssl.tspassword=wso2vfs&amp;vfs.ssl.
keypassword=importkey</parameter>
```

Reading Files from an SFTP

With WSO2 ESB File Inbound Endpoint, you can also read files from a remote SFTP. This is very similar to accessing a remote resource using SSH or SCP. Similar to the other configuration, the credentials can be provided (in encrypted form if required) in the connection URL, together with the absolute path of the resource.

Listing 7-4. File URI to poll an SFTP

```
<parameter name="transport.vfs.FileURI">vfs:sftp://kasun:password@host/
Users/kasun/development/orders
</parameter>
```

Similar to FTP/s you can use required certificates and key stores via vfs.ssl.* parameters.

FTP or SFTP Through a Proxy Server

It is often necessary to connect to FTP or SFTP through a proxy server. In such cases, you can configure your `FileURI` so that it has the required parameters to connect to FTP/SFTP via the proxy server. For example, you can configure a remote FTP file location via a proxy server as shown in Listing 7-5.

Listing 7-5. File URI of an FTP File Location Connected via a Proxy Server

```
ftp://username:password@127.0.0.1/home/wso2/res?proxyServer=201.10.0.11&pro
xyPort=3128&proxyUsername=proxyuser&proxyPassword=proxyPass&timeout=2500&re
tryCount=3
```

Here we used a proxy server address, port, and other parameters to specify the details of the proxy server. The same configuration constructs apply to SFTP as well.

Writing Files

From an ESB message mediation logic, you may have to trigger various file system operations including writing the content to a file, deleting a file, renaming, moving, and so on. WSO2 ESB provides two main ways that you can interact with the file system from a given mediation flow.

- *VFS Transport Sender:* Primarily supports writing contents to a file.
- *File Connector:* Supports various file operations such write, reads, renames, moves, etc.

VFS transport sender is the conventional way that most of the WSO ESB integration use cases are leveraged and File Connector is the latest addition to the WSO2 ESB's file integration capabilities. You can use either of these approaches based on your use case. Let's start with how you can use VFS transport sender.

Writing Files with VFS Transport

Let's learn how to use VFS transport sender to write content to the file system with a simple use case. Suppose that you want to design an HTTP service/REST API that can receive messages and write the message contents to the file system. To make the scenario less complex, assume that you can write the contents in the same format (here we use application/xml content) as you received it from the HTTP client.

As depicted in Figure 7-2, you can use WSO2 ESB's HTTP service/REST API construct to expose an HTTP interface to your client. So, the client can send the request to ESB's REST API with the message. And inside the mediation logic of the REST API, you don't really need to worry about transforming the message, as you only have to write the message to the file system in the same format. Therefore, you can simply use a call or send mediator with an address endpoint that has the VFS File URI specified.

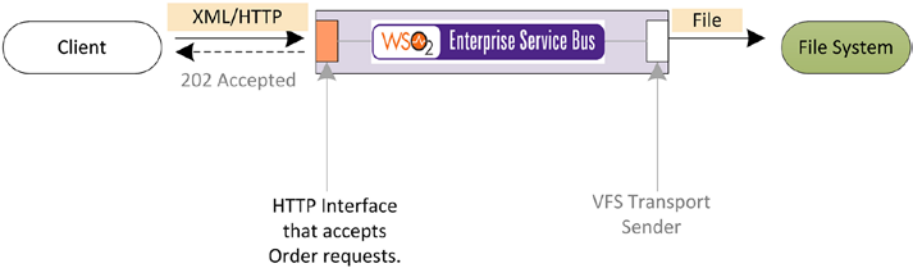

Figure 7-2. *Receiving messages via HTTP and writing the content to the file system*

When it comes to interacting with disparate file systems, File Transport Sender behaves in a similar way as the File Reader works for reading files. So you just have to enable the VFS Sender and specify the URI as the endpoint level with the the file system with which you want to interact.

■ **Note** You need to enable the VFS transport sender in the `axis2.xml` file as follows.

```
<transportSender name="vfs" class="org.apache.synapse.transport.vfs.
VFSTransportSender"/>
```

In Listing 7-6, it shows a configuration of the REST API. It uses the context and resource `orderproc/orders` and it's the resource URI that the client sends the request to. Therefore any request that comes to that particular URI goes through the message mediation logic, which logs a custom log message and then specifies the `OUT_ONLY` property to tell the engine that this is a one-way call. You can construct the name of the file that you want to write to the file system with the use of the `ReplyFileName` property that you set at the transport level. Here, we use a unique attribute of the message (which is message ID) to construct the filename.

Then you can use call mediator to send the message through the VFS transport sender with the use of an address endpoint that uses the `vfs:*` URI.

Listing 7-6. HTTP Service/REST API that Receives Messages via HTTP and Writes the Contents to the File System

```
<api xmlns="http://ws.apache.org/ns/synapse"
     name="OrderProcessor"
     context="/orderproc">
   <resource methods="POST" url-mapping="/orders">
      <inSequence>
         <log level="custom">
            <property name="Message Flow" value="== Order Received =="/>
         </log>
         <property name="OUT_ONLY" value="true" scope="default"
         type="STRING"/>
         <property name="transport.vfs.ReplyFileName"
                   expression="fn:concat(fn:substring-after(get-
                   property('MessageID'), 'urn:uuid:'), '.xml')"
                   scope="transport"/>
         <call>
            <endpoint>
               <address uri="vfs:file:///Users/kasun/development/deployment/
               maestro/wso2esb-4.9.0/sample_resources/file_writing"/>
            </endpoint>
         </call>
      </inSequence>
   </resource>
</api>
```

Similarly you can write message content to different file systems by specifying the required VFS URI as the address endpoint URI. For example, you can write the files to SFTP with a similar configuration shown in Listing 7-7.

Listing 7-7. Writing the Message Contents to the File System via SFTP

```
<call>
        <endpoint>
                <address uri="vfs:sftp://user_1:user_password@ftp.wso2.com/
foo/bar"/>
        </endpoint>
</call>
```

In some scenarios, you may have to access remote file systems via proxy servers. For example, assume that your FTP server has to be accessed via a proxy server, then you can configure the proxy server parameters as shown in Listing 7-8.

Listing 7-8. Writing the Message Content to an FTP through a Proxy Server

```
<call>
        <endpoint>
        <address uri="ftp://username:password@127.0.0.1/home/wso2/res?proxyS
        erver=127.0.0.1&proxyPort=3128&proxyUsername=proxyuser&proxyPassword
        =proxyPass&timeout=2500&retryCount=3
                "/>
        </endpoint>
</call>
```

Here, you can configure the parameter of the proxy server along with the URI of the address endpoint.

So far we have discussed how you can access the file system from an ESB message flow and write the content to the file system. However, you may have to access the file system and do other file operations other than write. For such use cases, you can use File Connector, which we will discuss in the latter parts of this chapter.

Transferring Files

Transferring or streaming files is another common requirement of file-based integration. For example, as depicted in Figure 7-2, assume that you want to poll a given file system location in your local file system and transfer it to an FTP, without processing the contents of the file.

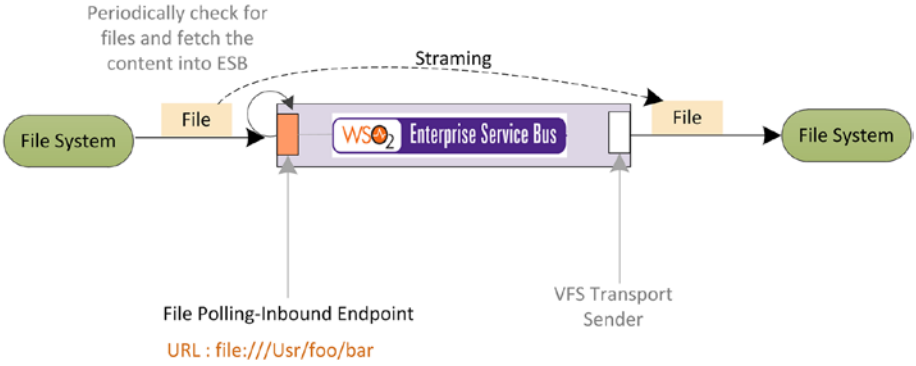

Figure 7-3. *Streaming Files from the local file system to an FTP*

This may seem to be quite obvious, in that you can just use a file inbound endpoint to read the file, then inject it into a sequence and an outbound endpoint and the VFS File URI writes the content to the destination. When you transfer large files, if you load the entire message in to the memory, it starts to consume more memory. If you don't really want to process the contents of the file, you can simply stream the content to achieve better performance when transferring large files. In such scenarios, you can enable binary builders in the `axis2.xml` as follows.

In <ESB_HOME>/repository/conf/axis2/axis2.xml, in the messageBuilders and formatters section as follows:

```
<messageBuilder contentType="application/binary" class="org.apache.axis2.
format.BinaryBuilder"/>

<messageFormatter contentType="application/binary" class="org.apache.axis2.
format.BinaryFormatter"/>
```

In the inbound endpoint, you have to specify the following parameter to keep the content as a stream in the message rather than building it.

```
<parameter name="transport.vfs.Streaming">true</parameter>
```

In the same proxy service, before the send mediator, add the following property:
You also need to add the following property so that your sequence holds the original thread until the send happens.

```
<property name="ClientApiNonBlocking" value="true" scope="axis2"
action="remove"/>
```

In Listing 7-9, you can find the full configuration of the inbound endpoint and the sequence that you can use to stream a file through the ESB.

Listing 7-9. Transferring Files from the Local File System to FTP

```
<inboundEndpoint xmlns="http://ws.apache.org/ns/synapse"
                 name="FileStreamingInboundEP"
                 sequence="fileStreamingSeq"
                 onError="fault"
                 protocol="file"
                 suspend="false">
   <parameters>
      <parameter name="interval">1000</parameter>
      <parameter name="coordination">true</parameter>
      <parameter name="transport.vfs.ContentType">application/binary
      </parameter>
      <parameter name="transport.vfs.LockReleaseSameNode">false</parameter>
      <parameter name="transport.vfs.AutoLockRelease">false</parameter>
      <parameter name="transport.vfs.ActionAfterFailure">DELETE</parameter>
      <parameter name="transport.vfs.CreateFolder">true</parameter>
      <parameter name="sequential">true</parameter>
      <parameter name="transport.vfs.ActionAfterProcess">MOVE</parameter>
      <parameter name="transport.vfs.FileURI">file:///Users/kasun/
      development/deployment/maestro/wso2esb-5.0.0-BETA/sample_resources/
      file_reading/streaming</parameter>
```

```
      <parameter name="transport.vfs.MoveAfterProcess"> file:///Users/kasun/
      development/deployment/maestro/wso2esb-5.0.0-BETA/sample_resources/
      file_reading/processed </parameter>
      <parameter name="transport.vfs.DistributedLock">false</parameter>
      <parameter name="transport.vfs.Streaming">false</parameter>
      <parameter name="transport.vfs.Locking">enable</parameter>
      <parameter name="transport.vfs.FileSortAscending">true</parameter>
      <parameter name="transport.vfs.FileSortAttribute">NONE</parameter>
      <parameter name="transport.vfs.Build">false</parameter>
      <parameter name="transport.vfs. Streaming">true</parameter>
   </parameters>
</inboundEndpoint>
<sequence xmlns="http://ws.apache.org/ns/synapse" name="fileStreamingSeq">
   <property name="OUT_ONLY" value="true" scope="default" type="STRING"/>
   <property name="transport.vfs.ReplyFileName"
             expression="fn:concat(fn:substring-after(get-
property('MessageID'), 'urn:uuid:'), '.png')"
             scope="transport"/>
   <log level="custom">
      <property name="Message Flow" value="== Order Received =="/>
   </log>

   <call>
      <endpoint>
         <address uri="vfs:file:///Users/kasun/development/deployment/
         maestro/wso2esb-5.0.0-BETA/sample_resources/file_writing/
         streaming"/>
      </endpoint>
   </call>
</sequence>
```

Now you have a good understanding about how you can integrate WSO2 ESB for reading and writing files. In the next few sections, let's dive deep into more file-based integration scenarios that you may encounter in real-world scenarios.

Message Transformation with File Integration

In most file-based integration scenarios, you will have to transform the file contents from one format to another. Let's take the example in Figure 7-2, where you have to implement HTTP interface on top of an existing file system. You expose an HTTP interface to your client, so that the client can send an HTTP request in JSON format to the ESB. Then ESB accepts the request, transforms it to CSV, and stores it in the file system.

As shown in Figure 7-4, the implementation of this scenario can be done with the concepts that you've already learned. You need to use a REST API/HTTP service and, for the message mediation logic, you need to use data mapper to convert the message from JSON to CSV and use VFS endpoint to store the CSV content to the file system.

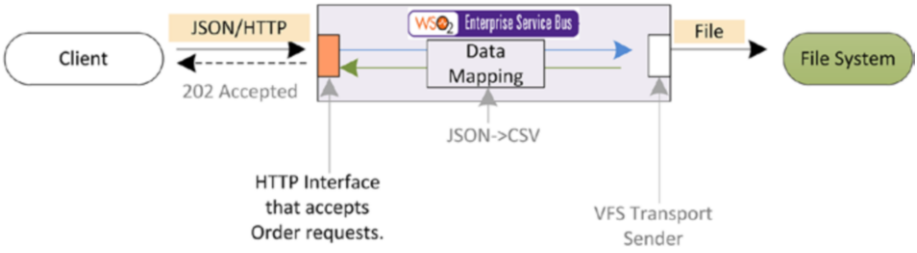

Figure 7-4. *Converting an HTTP/JSON request to CSV and storing it in a file system*

Listing 7-10 shows the configuration of the REST API. We configure the REST API so that it accepts the request and sends an HTTP 202 Accepted response by setting the `FORCE_SC_ACCEPTED` property. Then we use data mapper's configuration along with the outbound file-sending configuration that you already learned about in previous sections.

Listing 7-10. Transferring Files from the Local File System to FTP

```
<api context="/orderproc" name="JSON2CSV_OrderProcessor" xmlns="http://
ws.apache.org/ns/synapse">
    <resource methods="POST GET">
        <inSequence>
            <property name="FORCE_SC_ACCEPTED" scope="axis2" type="STRING"
            value="true"/>
            <datamapper config="gov:datamapper/orderProcMappingConfig.
            dmc" inputSchema="gov:datamapper/orderProcMappingConfig_
            inputSchema.json" inputType="JSON" outputSchema="gov:datamapper/
            orderProcMappingConfig_outputSchema.json" outputType="CSV"/>
            <property name="OUT_ONLY" scope="default" type="STRING"
            value="true"/>
            <property expression="fn:concat(fn:substring-after(get-
            property('MessageID'), 'urn:uuid:'), '.csv')" name="transport.
            vfs.ReplyFileName" scope="transport" type="STRING"/>
            <call>
                <endpoint>
                    <address uri="vfs:file:///Users/kasun/development/
                    deployment/maestro/wso2esb-4.9.0/sample_resources/file_
                    writing"/>
                </endpoint>
            </call>
        </inSequence>
        <outSequence/>
        <faultSequence/>
    </resource>
</api>
```

As you can see in the previous example, you can mix and match different mediation technologies along with file-based integration in real-world integration scenarios.

File Connector

So far what we have done related to file integration is read or write files. But in many scenarios you have to carry out file operations other than read or write and you may also have to carry out such operations inside a message flow. Before we dive into the details of a file connector, let's look at the *connector* concept in WSO ESB.

■ **Note** A *connector* allows you to interact with a third-party product's functionality and data from your ESB message flow, enabling you to connect to and interact with the APIs of services such as Twitter, Salesforce, and JIRA. A connector is independent from a given WSO2 ESB product and you have to download/install the connector as a separate entity. You can download and install WSO2 ESB connectors from WSO2 store at `https://store.wso2.com/store/assets/esbconnector`. All connectors are 100% free and open source. We will discuss ESB connectors in detail in the next chapter.

You can download and install file connectors in your WSO2 ESB runtime. Then you can call various file operations from your mediation flow. Installing can be done through the developer studio or by copying the connector ZIP file to the `ESB_HOME/repository/deployment/server/synapse-lib` directory (along with the import file that goes to the `/imports` directory).

To understand how you can use the file connector, let's consider the use case shown in the Figure 7-5. Assume that you need to build a REST API for an order processing system. Each order can be retrieved, added, modified, or deleted via the REST API. However, the backend system is a legacy system that only accepts orders in the form of files. So for each operation that's carried out, the REST API has a relevant file operation that you need to execute.

Figure 7-5. *Using File Connector to execute various file system operations from a mediation flow*

To build this integration scenario you can use a REST API/HTTP interface in WSO2 ESB and inside the message mediation logic you can call the required file operations. The usage of a connector inside mediation logic is similar to using the mediator that you have already learnt. Each connector operation can be called from the mediation logic. For

instance, if you want to read the contents of a file from the file system and add that as the current message inside a mediation flow, you can use the following logic.

```
<fileconnector.read>
        <source>/file_path/orders/A234R.json</source>
        <contentType>application/json</contentType>
</fileconnector.read>

<log level="full"/>
```

Once you invoke this particular sequence, you can see the contents of the file as the log message. In Listing 7-11, it shows the full configuration of the use case. Here we can use a REST API with a resource, and inside the in-sequence of that resource, we have a switch condition for various HTTP operations. In this example, we only use GET and POST for simplicity.

Listing 7-11. Implementing File System Operations with File Connector and Exposing Them as a REST API

```
<api xmlns="http://ws.apache.org/ns/synapse"
     name="OrderManagerHTTPService"
     context="/ordermanager">
    <resource methods="POST GET" url-mapping="/orders">
        <inSequence>
            <log level="custom">
                <property name="Message Flow" value="== Order Received =="/>
            </log>
            <switch source="$axis2:HTTP_METHOD">
                <case regex="POST">
                    <property name="order_id"
                              expression="json-eval($.orders.order[0].id)"/>
                    <property name="order_file_name"
                              expression="fn:concat('/Users/kasun/
                              development/deployment/maestro/wso2esb-5.0.0-
                              BETA/sample_resources/orders/',$ctx:order_id,
                              '.json')"/>
                    <log level="custom">
                        <property name="Message Flow" value="--- Order :
                        Create ---"></property>
                        <property name="Message Flow" expression="json-
                        eval($.orders.order[0])"></property>
                        <property name="Param" expression="$ctx:order_file_
                        name"></property>
                    </log>
                    <fileconnector.create>
                        <source>{$ctx:order_file_name}</source>
                        <inputContent>{json-eval($.orders.order[0])}
                        </inputContent>
```

```
                    </fileconnector.create>
                </case>
                <case regex="GET">
                    <log level="custom">
                        <property name="Message Flow" value="--- Order :
                        Get ---"></property>
                    </log>

                    <property name="order_file_name"
                              expression="fn:concat('/Users/kasun/
development/deployment/maestro/wso2esb-5.0.0-BETA/sample_resources/
orders/',$ctx:query.param.orderid, '.json')"/>

                    <fileconnector.read>
                        <source>{$ctx:order_file_name}</source>
                        <contentType>application/json</contentType>
                    </fileconnector.read>
                    <property name="messageType" value="application/json"
scope="axis2" type="STRING"/>
                    <log level="full"/>
                    <property name="NO_ENTITY_BODY" action="remove"
scope="axis2"/>
                    <respond/>
                </case>
            </switch>
        </inSequence>
    </resource>
</api>
```

As you can observe in Listing 7-11, for the POST condition we have extracted the contents from the HTTP POST message. Here is a sample JSON message that the client sends to the ESB's REST API.

```
{
  "orders": {
    "order": [
      {
        "id": "A234R",
        "name": "iPhone 6S",
        "description": "Apple Inc. 2016, unlocked",
        "unitPrice": "USD 700",
        "quantity": "1"
      }]
  }
}
```

As shown in Listing 7-11, the filename is constructed from the extracted content from the POST message and then we used the File Connector's `create` operation to create a file with the filename and the content that is passed as its argument. It creates a file that matches the HTTP POST request in the file system.

For the GET message, we extracted the name of the file from the query parameter, used that as the arguments for File Connector's `read` operation, and passed the contents of the file as the response payload. To make the use case less verbose, we've assumed that the file content format and HTTP messages use the same content type (JSON). Similarly, you can extend the use case to other HTTP verbs such as PUT and DELETE, and use the relevant File Connector operations to implement the mapping file system operations.

File Connector supports a number of other file operations such as:

- Append: Appends content to an existing file.
- Archive: Archives a file or folder.
- Copy: Copies a file or folder.
- Create: Creates a file or folder.
- Delete: Deletes a file or folder.
- isFileExist: Checks the existence of a file.
- listFileZip: Lists all files inside the ZIP file.
- Move: Moves a file or folder.
- Read: Reads the contents from a file.
- Search: Finds a file based on a file and directory pattern.
- Unzip: Decompresses a ZIP file.
- ftpOverProxy: Connects to a FTP server through a proxy.
- Send: Sends a file.

With these operations, you can invoke almost all the file operations from a mediation flow. In case you want to use these file operations in a periodic way rather than doing it on demand (when you get a request) inside a message flow, you can configure the file operation logic using the File Connector inside a sequence and invoke that sequence periodically through scheduled tasks.

Protocol Transformation from File to JMS

In some file integration use cases, you may have to integrate the file system with messaging queuing techniques. For instance, Figure 7-6 shows an integration scenario where the orders are received as files and you need to add all such orders to a JMS queue, so that some other system can process orders by consuming the queue. This scenario can be easily implemented by combing the file integration techniques you learned in this chapter and JMS integration techniques that we covered in Chapter 6.

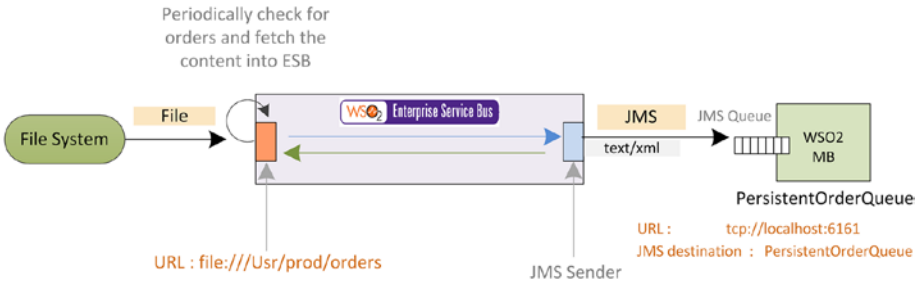

Figure 7-6. *The poll file system for order files and enqueuing those order requests inside a JMS queue resides in the WSO2 Message Broker (WSO2 MB)*

Let's assume that for messaging queuing, we use the WSO2 message broker. So, you can create a File Inbound endpoint so that you can poll the file system location to which you receive the orders. Then from the ESB mediation logic you can simply configure JMS sender and send the messages through a call mediator with an addressing endpoint with JMS URI.

The configuration of this integration scenario is shown in Listing 7-12. As you have already learned, a File Inbound endpoint is used to poll the file system for new orders that are created as files. Then the inbound endpoint injects the contents of the file as a message to a sequence. The sequence takes care of handling the message and it sends the message over to a JMS endpoint, which finally enqueues the message to the `PersistentOrderQueue` of the WSO2 Message Broker.

Listing 7-12. Transferring Files from a Local File System to FTP

```
<inboundEndpoint xmlns="http://ws.apache.org/ns/synapse"
                 name="OrderProcessing_File_Inbound_SFTP"
                 sequence="orderProcessingSeq"
                 onError="fault"
                 protocol="file"
                 suspend="false">
   <parameters>
      <parameter name="interval">1000</parameter>
      <parameter name="coordination">true</parameter>
      <parameter name="transport.vfs.ContentType">text/xml</parameter>
      <parameter name="transport.vfs.LockReleaseSameNode">false</parameter>
      <parameter name="transport.vfs.AutoLockRelease">false</parameter>
      <parameter name="transport.vfs.ActionAfterFailure">DELETE</parameter>
      <parameter name="transport.vfs.CreateFolder">true</parameter>
      <parameter name="sequential">true</parameter>
      <parameter name="transport.vfs.ActionAfterProcess">MOVE</parameter>
      <parameter name="transport.vfs.FileURI">file:///Users/kasun/
      development/deployment/maestro/wso2esb-5.0.0-SNAPSHOT_RC/sample_
      resources/file_reading/source</parameter>
```

```
<parameter name="transport.vfs.MoveAfterProcess"> file:///Users/kasun/
development/deployment/maestro/wso2esb-5.0.0-SNAPSHOT_RC/sample_resources/
file_reading/processed</parameter>

      <parameter name="transport.vfs.DistributedLock">false</parameter>
      <parameter name="transport.vfs.Streaming">false</parameter>
      <parameter name="transport.vfs.Locking">enable</parameter>
      <parameter name="transport.vfs.FileSortAscending">true</parameter>
      <parameter name="transport.vfs.FileSortAttribute">NONE</parameter>
      <parameter name="transport.vfs.Build">false</parameter>
   </parameters>
</inboundEndpoint>
<sequence xmlns="http://ws.apache.org/ns/synapse" name="orderProcessingSeq">
   <log level="full">
      <property name="MSG" value="== Received =="/>
   </log>
   <property name="OUT_ONLY" value="true"/>
   <call>
      <endpoint>
         <address uri="jms:/PersistentOrderQueue?transport.jms.Conne
ctionFactoryJNDIName=QueueConnectionFactory&amp;java.naming.factory.
initial=org.wso2.andes.jndi.PropertiesFileInitialContextFactory&amp;java.
naming.provider.url=repository/conf/jndi.properties&amp;transport.jms.
DestinationType=queue"/>
      </endpoint>
   </call>
</sequence>
```

Similarly, you can mix and match various protocol integration scenarios with File Integration techniques to solve real-world integration problems.

Summary

In this chapter, you learned:

- Why file-based integration is still used in the modern enterprises.
- How to poll, read files, and inject the content as messages into the ESB message flow using the File Inbound endpoint.
- How to wrote the message content to a file from an ESB message flow with VFS sender.
- How to stream a large file as binary content through the ESB to build file transfer integration scenarios.
- How to invoke various file operations from a mediation flow with File Connectors.
- How to use data mapping and protocol switching with file-based integration techniques.

CHAPTER 8

Integrating Applications, Cloud Services, and Data

As you learned in the first chapter, one of the main objectives of the ESB design is to facilitate various types of integration scenarios. Integration middleware such as ESBs facilitate the key features such as communication, mediation, orchestration, transformation, QoS, security, monitoring, administration and management to cater to these disparate integration needs. At the time when the ESB is emerging as an integration technology, the primary integration requirement was to support integrations of on-premise systems and services. However, with the proliferation of APIs, mobile devices and Software as a Service (SaaS), the ESBs have to support a much broader range of integration scenarios, ranging from conventional integrations to integration of cloud services and APIs. In addition to the types of integrations that ESB supports, the integration runtime could also be a cloud service. That means you can develop, deploy, and run your integration scenarios in the cloud. This is known as Integration Platform as a Server (iPaaS).

In this chapter, we focus on integration of proprietary systems that reside as on-premise applications and integrating clouds services with the use of WSO2 ESB. Also, we'll discuss how WSO2 ESB can be used as an Integration Platform as a Service (iPaaS), during the later part of the chapter.

Let's start our discussion with how you can integrate on-premise proprietary applications with WSO2 ESB.

Integrating Proprietary Systems

In any organization, there can be proprietary systems that help the organization run its key business functionalities. For example, most large organizations heavily use proprietary software systems such as ERP (Enterprise Resource Planning), CRM (Customer Relationship Management), HRM (Human Resource Management), etc., to run its key business activities. However, most of these systems are monolithic and inherently built with proprietary technologies. Often organizations have to integrate these proprietary systems with other types of software applications in the organization.

K. Indrasiri, *Beginning WSO2 ESB*, DOI 10.1007/978-1-4842-2343-7_8

For example, assume that a given organization uses a SAP system to manage its retail business and there are some other business functionalities that they want to integrate the SAP system with. However, those functionalities are built as web services. So, now the organization has to come up with a solution to integrate SAP system with these web services, which is a major effort when it comes to its software development. Therefore, integration middleware vendors incorporate such proprietary integration capabilities as part of the ESB, so that ESB takes care of all the heavy lifting of integrating proprietary systems.

WSO2 ESB comes with several adapters to connect with these proprietary systems. Let's start with SAP Adapter of WSO2 ESB.

SAP Integration

SAP ERP solutions provide reliable and efficient platforms to build and integrate enterprise or business-wide data and information systems with ease. SAP applications, based on their R/3 system, provide the capability to manage financial, asset, and cost accounting, production operations and materials, personnel, plants, and archived documents. When it comes to implementing various business functionalities, often organizations want to integrate SAP applications with non-SAP applications. For example, a given business functionality may be implemented by leveraging existing SAP applications and another software application that's built as a web service. In order to cater to this kind of integration requirements, WSO2 ESB provides a SAP adapter that allows you to connect your WSO2 ESB with your SAP applications, so that you can use WSO2 ESB as the communication bridge between SAP and non-SAP applications.

SAP integration requires the foundation knowledge on SAP, which is not covered within the scope of this book.

To integrate non-SAP systems with SAP applications, SAP provides a Java Connector, called "SAP Java Connector (SAP JCo)." WSO2 ESB uses SAP JCo as the Java library inside its SAP adapter. SAP applications and non-SAP applications can be integrated using IDoc or BAPI calls. WSO2 ESB uses the SAP JCo adapter to implement the WSO2 SAP adapter. You need to install the SAP JCo adapter into your WSO2 ESB distribution. Refer to the installation guide at `https://docs.wso2.com/display/ESB500/SAP+Integration`.

Let's start with how you can integrate SAP applications with WSO2 ESB using IDocs.

■ **Note** SAP inbound endpoints are not included in the GA release at that time this book is written. However, SAP integration support is available as a SAP transport for a proxy service. Therefore, you can use the same parameters with SAP transport with proxy services.

IDOC-Based Integration

Intermediate Documents (IDocs) are used to transfer data from SAP systems to external systems and external systems to SAP systems. An IDoc carries data about a business transaction from one system to another in the form of an electronic message. The transfer from SAP to non-SAP system is done via EDI (Electronic Data Interchange). EDI converts the data from IDoc into XML or an equivalent format. IDocs are used for asynchronous transactions. (For more information, refer to http://scn.sap.com/docs/DOC-34785.)

Receiving IDocs

Let's consider an IDoc-based integration use case with WSO2 ESB. Suppose that a SAP R/3 system wants to send some purchasing information to a third-party SOAP web service, a purchasing management service. The purchasing information can be transferred via IDoc asynchronous data exchange. In this case, the WSO2 ESB SAP adapter acts as the bridge between the SAP system and the purchasing management service.

As illustrated in Figure 8-1, you can use a SAP IDoc inbound endpoint as the communication interface between WSO2 ESB and the SAP R/3 system.

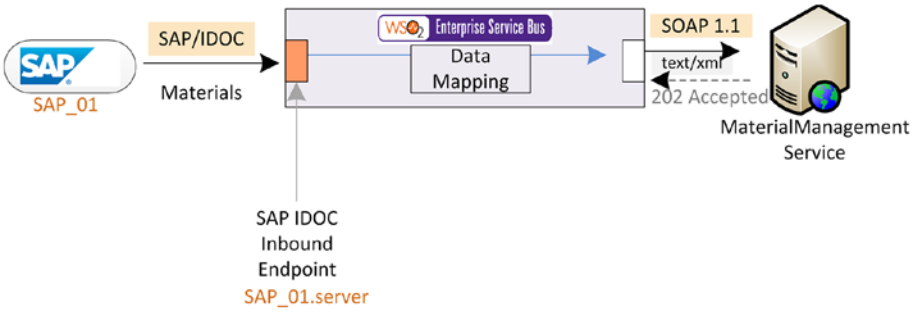

Figure 8-1. *WSO2 ESB SAP adapter receives an IDoc via its SAP IDoc inbound endpoint and then is converted and represented as a XML format inside the ESB. The communication between SAP and ESB is asynchronous one-way communication.*

In order to connect to the SAP system to receive IDocs, you need to configure the `.server` file inside `<ESB_HOME>/repository/conf/sap`.

This file should be named `<SAP-GWHOST>.server` and should define the relevant properties (Refer to https://docs.wso2.com/display/ESB500/SAP+Integration.) Here is a sample configuration of the `SAP_01.server` file.

```
jco.server.gwhost=/H/<IP>/S/3299/H/<IP>/S/3200
jco.server.gwserv=3300
jco.server.progid=IGS.CPT
jco.server.repository_destination=IGS.CPT
jco.server.name=IGS.CPT
jco.server.unicode=1
```

When you configure the SAP IDoc inbound endpoint, you need to refer to the respective server configuration with the parameter `transport.sap.serverName` (here we use `SAP_01`). Once you successfully start the SAP IDoc inbound, you should be able to see a log message with the SAP IDoc server name and the program ID. By looking at the SAP system, you could also see that there is a successful RFC connection established from the SAP system to ESB.

Listing 8-1 shows the SAP IDoc inbound endpoint configuration for this scenario. Once you send an IDoc over the RFC connection with this program ID, the ESB inbound endpoint will receive it. Then the message is injected into the specified sequence.

Listing 8-1. SAP IDoc Inbound Endpoint

```
<inboundEndpoint xmlns="http://ws.apache.org/ns/synapse"
                 name="SAP_IDoc_PurchasingDataInboundEP"
                 sequence="purchasingMaterialsIDocSeq"
                 onError="fault"
                 protocol="idoc"
                 suspend="true">
   <parameters>
      <parameter name="transport.sap.enableTIDHandler">enabled</parameter>
      <parameter name="transport.sap.serverName">SAP_01</parameter>
   </parameters>
</inboundEndpoint>
```

You can do the required data mapping inside that sequence to match the IDoc XML format to the required payload of the backend service. As discussed earlier, the entire IDoc receiving message flow is asynchronous so no response is sent back to the SAP system.

Sending IDocs

When you integrate SAP systems with non-SAP systems, you may have to send IDocs to the SAP system from non-SAP systems. WSO2 ESB facilitates IDoc sending with the transport sender that allows you to send an IDoc to any external SAP system.

Let's consider the integration scenario depicted in Figure 8-2. Suppose that you want to expose a REST API for receiving price updates of goods from external systems and then you want to update the SAP R/3 system based on those price updates. You can use a REST API to receive prices updates as JSON messages over HTTP. Now you want to convert these messages into the IDoc message format and send them across to the SAP system.

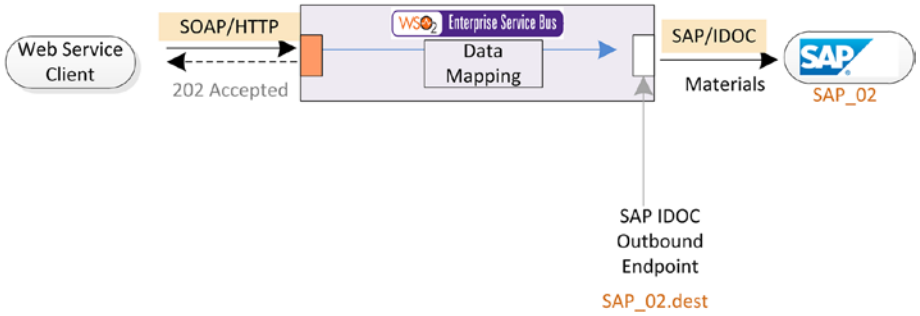

Figure 8-2. *WSO2 ESB SAP transport sender can be configured to send IDocs to a SAP R/3 system. You can configure all the parameters related to the destination SAP system by specifying them with a .dest file in repository/conf/sap. You can refer to this .dest file from your endpoint configuration.*

The first thing that you have to do here is that you need to configure the SAP R/3 system as a destination in the ESB configuration. This can be done by creating a configuration file `<SAP-GWHOST>.dest` inside the `<ESB_HOME>/repository/conf/sap`.

The `*.dest` properties file should be named `<SAP-GWHOST>.dest`. For example, if the name of your SAP gateway is `SAPSYS`, the name of the file should be `SAP_02.dest`.

(Refer to https://docs.wso2.com/display/ESB500/SAP+Integration.) Here is a sample configuration of the `SAP_02.dest` file.

```
jco.client.client=800
jco.client.user=wso2_user_sap02
jco.client.passwd=wso2pass14
jco.client.lang=en
jco.client.ashost=/H/<IP>/S/3299/H/<IP>/S/3200
jco.client.gwserv=3300
jco.client.sysnr=00
jco.client.idle_timeout=300
jco.client.logon=0
jco.client.msserv=3600
jco.client.trace=0
jco.client.getsso2=0
jco.client.r3name=CPT
```

You also have to enable the IDoc transport sender in the `axis2.xml` file as follows:

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

Once you have the transport sender side configuration completed, you need to decide on the type of IDoc that you want to create at the SAP system. For example, as per the use case in Figure 8-2, the price update request should have a specific IDoc type that you want to create at the destination SAP system. Once you have the format of the

IDoc that you want to create at the target SAP system, you can do the data mapping between the incoming JSON format and the IDoc XML format. Once the data mapping is completed, you can simply call the IDoc endpoint with a call mediator.

Listing 8-2. Sending IDoc to SAP System

```
<proxy xmlns="http://ws.apache.org/ns/synapse"
       name="PriceUpdateService"
       transports="http"
       startOnLoad="true"
       trace="enable"
       statistics="enable">
    <target>
        <inSequence>
            <log level="full"/>
            <!-- ... data mapper ... -->
            <call>
                <endpoint name="sapidocendpoint">
                    <address uri="idoc:/SAP_02"/>
                </endpoint>
            </call>
        </inSequence>
        <outSequence/>
    </target>
    <description/>
</proxy>
```

As with any Idoc-related messaging scenario, this is also a one-way asynchronous messaging scenario.

Since now you have a good understanding of how you can integrate WSO2 ESB with SAP R/3 systems through IDoc transport, let's move on to SAP integration with BAPI.

BAPI-Based Integration

BAPIs are standardized programming interfaces or methods that enable external applications to access the business functionalities and data in the SAP R/3 system. BAPIs are defined in the BOR (Business Object Repository) as methods of SAP business object types that carry out specific business functions. BAPIs can be considered a subset of the RFC-enabled function modules, especially designed as an API to the SAP business object. Unlike IDoc based communication, BAPIs are invoked synchronously and they send response data back to the client (in most cases).

Some BAPIs and methods provide basic functions and can be used for most SAP business objects.

```
BAPIs for Reading Data - GetList() , GetDetail() , GetStatus() ,
ExistenceCheck()
```

```
BAPIs for Creating or Changing Data- Create() ,Change(),Delete() and
Undelete() ,
BAPIs for Mass Processing -ChangeMultiple(), CreateMultiple(),
DeleteMultiple().
```

Let's consider a BAPI integration scenario where ESB has to invoke a BAPI residing in a SAP R/3 system.

Invoking BAPIs

Suppose that you want to expose a functionality, which is exposed as a BAPI from your SAP R/3 system as a REST API from WSO2 ESB. The REST client can invoke the REST API hosted in WSO2 ESB and ESB can take care of handling the complexity related to the BAPI integration (see Figure 8-3).

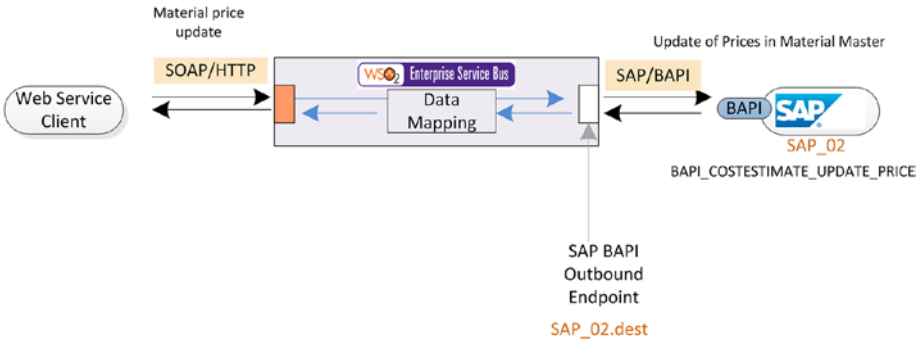

Figure 8-3. *WSO2 ESB SAP transport sender enables the remote BAPI invocation from ESB. Similar to the IDoc sending use case, you can configure a .dest file to specify the details of the SAP system and use that as the endpoint name of the BAPI endpoint.*

The first thing that you have to do is configure the SAP R/3 system in which these remote BAPIs exist, as a destination in the ESB configuration. This can be done by creating a configuration file `<SAP-GWHOST>.dest` inside `<ESB_HOME>/repository/conf/sap`.

The `*.dest` properties file should be named `<SAP-GWHOST>.dest`. For example, if the name of your SAP gateway is `SAPSYS`, the name of the file should be `SAP_02.dest`.

(Refer to https://docs.wso2.com/display/ESB500/SAP+Integration.) Here is a sample configuration of the `SAP_02.dest` file.

```
jco.client.client=800
jco.client.user=wso2_user_sap02
jco.client.passwd=wso2pass14
jco.client.lang=en
jco.client.ashost=/H/<IP>/S/3299/H/<IP>/S/3200
jco.client.gwserv=3300
jco.client.sysnr=00
```

```
jco.client.idle_timeout=300
jco.client.logon=0
jco.client.msserv=3600
jco.client.trace=0
jco.client.getsso2=0
jco.client.r3name=CPT
```

You also have to enable the BAPI transport sender in the `axis2.xml` file as follows:

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

Once you have the destination of SAP R/3 system configured, the next step is to decide the BAPI that you want to invoke at the remote SAP R/3 system. As the BAPI call is a remote function call (RFC), WSO2 ESB defines a mapping between the message and the RFC. That means if you have to invoke a BAPI in a SAP R/3 system, you need to configure the corresponding message format within the ESB to invoke that BAPI. As shown in Figure 8-4, you receive the JSON message over the REST API and the data mapping mediator is responsible for translating the incoming JSON message to the WSO2 ESB BAPI invocation message.

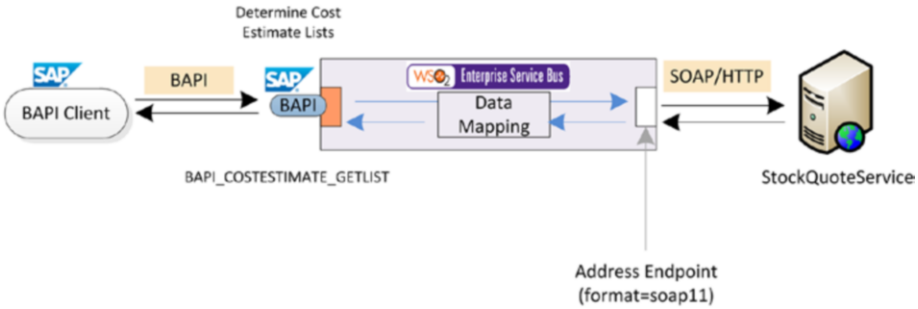

Figure 8-4. *Exposing a BAPI interface from WSO2 ESB so that the external BAPI client can consume one of the non-SAP business functionalities through a native SAP BAPI remote function call*

WSO2 ESB defines a generic message structure, which corresponds to a remote BAPI call. Listing 8-3 shows that structure. You can define the name of the BAPI that you want to invoke along with imports, structures, and tables. Therefore, this is the message format that you have to create inside the ESB message flow and then you can send it to a BAPI endpoint. Then ESB translates this message format to its corresponding BAPI remote function call.

Listing 8-3. Generic BAPI Structure that You Have to Create at the ESB Mediation Flow to Invoke a BAPI in a SAP R/3 System

```
<bapirfc name="{rfcFunctionName}">
    <import name="{jcoStructureName}">
        <structure>
            <field name="{fieldName}">{fieldValue_to_set_to_structure_
            field}</field>
        </structure>
        <field name="{fieldName}">{fieldValue}</field>
    </import>
    <tables>
        <table name="{tablename}">
            <row id="{rowId}"> <!-- collection of rows -->
                <field name="{fieldName}">{fieldValue}</field>
            </row>
        </table>
    </tables>
</bapirfc>
```

So, in you data mapping mediator, you should create a message similar to the one that's showed in Listing 8-3 as the output message. For example, a BAPI RFC call for `BAPI_COMPANYCODE_GETDETAIL` can be invoked from WSO2 ESB by constructing the message format shown in Listing 8-4 and then sending the message to the BAPI endpoint.

Listing 8-4. WSO2 ESB's Message Format that Corresponds to the RFC Call of BAPI_COMPANYCODE_GETDETAIL

```
<bapirfc xmlns="" name="BAPI_COMPANYCODE_GETDETAIL">
              <import>
                       <field name="COMPANYCODEID">XYZ</field>
               </import>
 </bapirfc>
```

You can configure the REST API as shown in Listing 8-5 to invoke the BAPI RFC.

Listing 8-5. Exposing a BAPI RFC Call as a REST API Through WSO2 ESB

```
<api xmlns="http://ws.apache.org/ns/synapse"
     name="BAPIInfoProviderAPI"
     context="/bapiinfo">
    <resource methods="POST">
        <inSequence>

            <!-- ... data mapping ... -->

            <call>
                <endpoint>
                    <address uri="bapi:/SAP_O2"/>
```

```
                </endpoint>
            </call>
            <property name="messageType"
                      value="application/json"
                      scope="axis2"
                      type="STRING"/>
            <respond/>
        </inSequence>
    </resource>
</api>
```

The response that you receive from the SAP BAPI invocation is also converted to the corresponding XML structure. For example, the response returned for a BAPI invocation of `BAPI_USER_GETLIST` has the following message format:

```
<BAPI_USER_GETLIST>
    <INPUT>
        <MAX_ROWS>0</MAX_ROWS>
        <WITH_USERNAME/>
    </INPUT>
    <OUTPUT>
        <ROWS>8</ROWS>
    </OUTPUT>
    <TABLES>
        <RETURN/>
        <SELECTION_EXP/>
        <SELECTION_RANGE/>
        <USERLIST>
            <item>
                <USERNAME>WSO2_User</USERNAME>
                <FIRSTNAME/>
                <LASTNAME/>
                <FULLNAME/>
            </item>
        </USERLIST>
    </TABLES>
</BAPI_USER_GETLIST>
```

You can complete the scenario by doing the reverse data mapping from BAPI response's XML structure to the expected JSON response of the client.

Exposing BAPI Interfaces

You can use WSO2 ESB to expose a BAPI on top of an existing web service or any other backend service. This may sound like a rare use case, but in the SAP world you may have to support a specific requirement of a BAPI client to consume one of the non-SAP business functionalities, as through a BAPI remote function call.

As you did with the IDoc receiving use case, you can configure a `.server` file, name it `<SAP-GWHOST>.server`, and define the relevant properties (Refer to https://docs.wso2.com/display/ESB500/SAP+Integration.) Here is a sample configuration of the `SAP_01.server` file.

```
jco.server.gwhost=/H/<IP>/S/3299/H/<IP>/S/3200
jco.server.gwserv=3300
jco.server.progid=IGS.CPT
jco.server.repository_destination=IGS.CPT
jco.server.name=IGS.CPT
jco.server.unicode=1
```

Then you can configure the SAP BAPI inbound endpoint. You need to refer to the respective server configuration with the parameter `transport.sap.serverName` (here we use `SAP_01`). Once you successfully start the BAPI endpoint, the external BAPI client can invoke a BAPI RFC residing in your WSO2 ESB. The request message is translated to the same generic XML format that you used in the previous use case when you invoked a BAPI from ESB.

Listing 8-6 shows the SAP BAPI inbound endpoint configuration of the scenario.

Listing 8-6. SAP BAPI Interface Exposed from WSO2 ESB Through a BAPI Inbound Endpoint

```
<inboundEndpoint xmlns="http://ws.apache.org/ns/synapse"
                 name="SAP_BAPI_UseInfo"
                 sequence="getUserDetails"
                 onError="fault"
                 protocol="bapi"
                 suspend="true">
   <parameters>
      <parameter name="transport.sap.enableTIDHandler">enabled</parameter>
      <parameter name="transport.sap.serverName">SAP_01</parameter>
   </parameters>
</inboundEndpoint>
```

When you want to send back the response, you have to create the same response XML message structure as in the previous use case. The BAPI inbound endpoint will convert the XML response message format back to a BAPI response.

HL7 Integration

Health Level 7 International (HL7) is a set of standards for transferring clinical and administrative data between hospital information systems. HL7 facilitate exchange, integration, sharing, and retrieval of electronic health information. The key idea of HL7 is to define a messaging standard for communication between disparate healthcare software applications to exchange clinical and administrative data.

There are several HL7 messaging standards defined under different versions.

- *HL7 Version 2.x:* HL7 versions 2.x have the aim to support hospital workflows. This standard uses a textual, non-XML message format based on delimiters.
- *Hl7 Version 3.x:* HL7 versions 3.x have the aim to support any and all healthcare workflows. This version uses XML-based message formats.

In the scope of this book, we focus on the HL7 v2.x-based integration scenarios with WSO2 ESB. The HL7 v2.x integration requirements are quite common, as most of the existing healthcare information systems still use HL7 v2.x. Support for HL7 v3.x in WSO2 ESB was not available at the time this book was written. However, there's on-going work related to supporting HL7 v3 on WSO2 ESB. So, for the rest of this chapter, we will focus on HL7 v.2 integration.

Let's look at how the HL7 messaging takes place between two healthcare information systems. Listing 8-7 shows a sample HL7 message.

Listing 8-7. Sample HL7 Message

```
MSH|^~\&|HL7SFOHealth|HIS|SFOHealth|HIS|201407271408||ADT^A04|1817457|D|2.5
.1|EVN|A04|AL
PID||0493575^^^2^ID 1|454721||Foo^BAR^^^^|FOO^BAR^^^^|19480203|M||B|254 E238
ST^^Howick^CA^3252^USA||(216)631-4359|||M|AGN|400003403~1129086|999-|
NK1||CONROY^MARI^^^^|SPO||(216)731-4359||EC|||||||||||||||||||||||||
PV1||O|O/R||||277^ALLEN^BONNIE^J^^^|||||||||| ||2688684|||||||||||||||||||
||||201407271408||||||002376853
```

So, this message format can be serialized or deserialized at the HL7 v.2.x consumers and producers. However, the standard doesn't specify a wire level protocol.

As the wire protocol between HL7 v2.x systems, the Minimal Lower Layer Protocol (MLLP) is used. MLLP protocol has a long history of use within the HL7 community, although it has never been formally part of the HL7 standard. The MLLP protocol is a minimalistic OSI-session layer framing protocol. It is assumed that the MLLP protocol will be used only in a network environment. So, in most cases, HL7 v2.x is used with MLLP, which is a text-based TCP socket-based protocol. With MLLP, special characters to form a block enclose the HL7 content. Listing 8-8 shows a sample HL7 message with MLLP. Note that `< SB>`, `< EB>`, and `< CR>` are used to denote the non-printable MLLP-framing single-byte values 0x0B, 0x1C, and 0x0D. (They are not XML tags.)

Listing 8-8. Sample HL7 Message with MLLP

```
<SB>
MSH|^~\&|ZIS|1^AHospital|||199605141144||ADT^A01|20031104082400|P|2.3|||
AL|NE|||8859/15|<CR>EVN|A01|20031104082400.0000+0100|20031104082400
PID||""|10||Vries^Danny^D.^^de||19951202|M|||Rembrandlaan^7^Leiden^^7301
TH^""
^^P||""|""||""|||||||""|""<CR>PV1||I|3w^301^""^01|S|||100^van den Berg^^A.S.
```

```
^^""^dr|""||9||||H||||20031104082400.0000+0100<CR>
<EB>
<CR>
```

So, when an HL7 source system sends a message to a target system, the source system will get an acknowledgement (ACK) from the target system with the receipt of the message. There are three types of ACKs defined in HL7 v2.x—Accept, Error, and Reject. WSO2 ESB HL7 integration support is implemented on top of the HAPI (see http://hl7api.sourceforge.net/) open source library.

So far, you learned about the fundamental concepts related to the HL7 protocol, so now it's time to dive deep into how different HL7 integration scenarios can be implemented with WSO2 ESB.

Receiving HL7 Messages

Integration between HL7 and non-HL7—based systems is a key part of HL7 integration. In this particular scenario, let's suppose that there is a healthcare information system, ABC Healthcare System, and it wants to send healthcare-related information to a SOAP-based healthcare service. Since these two systems are using disparate messaging protocols, we have to use the WSO2 ESB as the integration middleware.

Before proceeding into the actual integration scenarios, there are prerequisites that you need to configure. WSO2 ESB has to serialize and deserialize messages back and forth from HL7 to XML. For this purpose, the following message builders and formatter have to be enabled in `axis2.xml`.

```
<messageFormatters>
  <messageFormatter contentType="application/edi-hl7" class="org.wso2.
carbon.business.messaging.hl7.message.HL7MessageFormatter"/>
...
</messageFormatters>
...
<messageBuilders>
  <messageBuilder contentType="application/edi-hl7" class="org.wso2.carbon.
business.messaging.hl7.message.HL7MessageBuilder"/>
</messageBuilders>
```

In addition, you have to enable the transport sender for HL7 in `axis2.xml` as follows:

```
<transportSender name="hl7" class="org.wso2.carbon.business.messaging.hl7.
transport.HL7TransportSender">
    <!--parameter name="non-blocking">true</parameter-->
</transportSender>
```

The HL7 components required for WSO2 ESB are not shipped with the default ESB distribution (as it affects the distribution size), and you have to manually install those features from the WSO2 ESB feature repository. More details can be found at https://docs.wso2.com/display/ESB500/HL7+Transport.

Receiving HL7 Messages with Auto ACK

We can use WSO2 ESB's HL7 v.2 inbound endpoint as the HL7 message receiver at the ESB layer. The HL7 inbound endpoint essentially exposes an HL7 interface on top of the MLLP transport, so that the HL7 message sender can send messages through the MLLP protocol. The acknowledgement handling can be done in different ways, but for the use case shown in Figure 8-5, let's assume that ACK is sent once ESB receives the HL7 message at the inbound endpoint level.

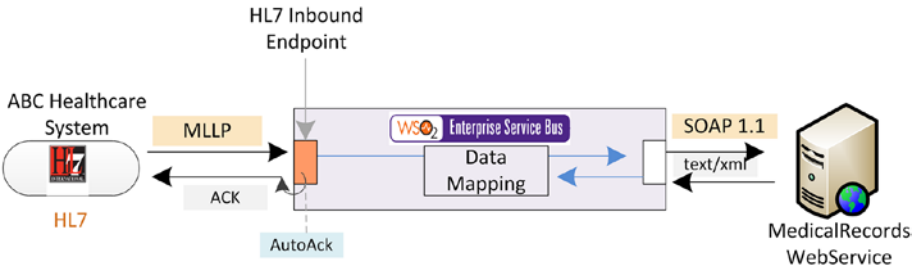

Figure 8-5. *Exposing an HL7 v.2 interface over MLLP protocol and transforming messages and sending them to a SOAP web service. The ACK message is sent when we receive the HL7 message at the HL7 inbound endpoint level.*

The HL7 inbound endpoint injects the HL7 message to the specified sequence and the message is converted to the XML canonical format at the ESB layer. So the data mapping is between the XML to the required SOAP message format of the backend web service. Since we have configured auto ACK, the response from the backend web service is not sent to the source HL7 system, but you can log it, drop it, or do any arbitrary process for it.

Listing 8-9 illustrates the configuration of HL7 inbound endpoint and the sequence that handles the incoming HL7 messages.

Listing 8-9. Receiving HL7 with Auto ACK and Sending Them to Backend Web Service

```
<inboundEndpoint xmlns="http://ws.apache.org/ns/synapse"
                 name="ABCHealthcare_HL7Receiver_AutoAck"
                 sequence="ABCHelathcareHL7_to_SOAP_with_AutoAck"
                 onError="fault"
                 protocol="hl7"
                 suspend="false">
   <parameters>
      <parameter name="inbound.hl7.Port">20000</parameter>
      <parameter name="inbound.hl7.AutoAck">true</parameter>
      <parameter name="inbound.hl7.ValidateMessage">true</parameter>
      <parameter name="inbound.hl7.TimeOut">10000</parameter>
      <parameter name="inbound.hl7.CharSet">UTF-8</parameter>
      <parameter name="inbound.hl7.BuildInvalidMessages">false</parameter>
```

```
      <parameter name="inbound.hl7.PassThroughInvalidMessages">false
      </parameter>
   </parameters>
</inboundEndpoint>

<sequence xmlns="http://ws.apache.org/ns/synapse" name="ABCHelathcareHL_to_
SOAP">
   <log level="full">
      <property name="MSG" value="== Received =="/>
   </log>
   <!-- ... Data Mapper ...-->
   <!-- Send to soap service -->

</sequence>
```

As the parameters of HL7 inbound endpoint you can provide a flag to set auto ACK, the TCP port in which it starts listening for incoming HL7 messages over MLLP protocol, timeout (when timeout expires an automatic NACK is sent to the source), flag to validate HL7 messages when you convert them to the XML infoset, etc. A list of supported parameters can be found at https://docs.wso2.com/display/ESB500/HL7+Inbound+Protocol.

Receiving HL7 Messages with Application ACK

The same scenario in the previous section can be configured with application acknowledgement. In this case, the ACK is sent back to the client only after we complete the entire sequence logic. For example, as shown in Figure 8-6, you can configure the WSO2 ESB so that it will send the ACK once we complete the injecting sequence.

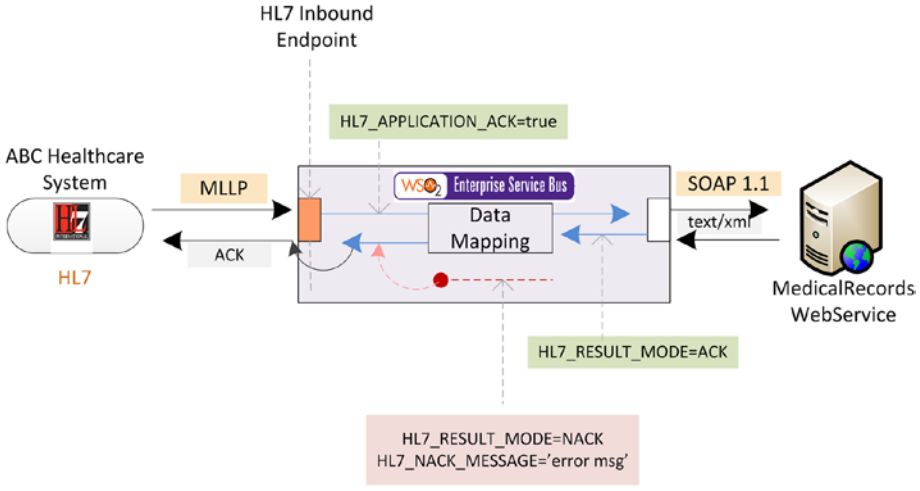

Figure 8-6. *Receiving HL7 messages and ACK is sent once we receive the response from the backend service. This is known as application ACK.*

The main idea is to specify that you want to do application ACK at the beginning (before sending message to the backend) of the injecting sequence with the property HL7_APPLICATION_ACK=true, and once you receive the response, you set HL7_RESULT_MODE=ACK. Application ACK can be a positive or negative ACK, which means if something fails when you invoke the backend service, your fault sequence will be triggered and you can configure the corresponding fault sequence to send back a NACK in that case. Listing 8-10 shows the configuration of the HL7 inbound endpoint and the injecting sequence that you can use to build an application ACK HL7 message receiving scenario.

Listing 8-10. Receiving HL7 with Application ACK

```
<inboundEndpoint xmlns="http://ws.apache.org/ns/synapse"
                 name="ABCHealthcare_HL7Receiver_AppAck"
                 sequence="ABCHelathcareHL7_to_SOAP_with_AppAck"
                 onError="fault"
                 protocol="hl7"
                 suspend="false">
   <parameters>
      <parameter name="inbound.hl7.Port">20001</parameter>
      <parameter name="inbound.hl7.AutoAck">false</parameter>
      <parameter name="inbound.hl7.ValidateMessage">true</parameter>
      <parameter name="inbound.hl7.TimeOut">10000</parameter>
      <parameter name="inbound.hl7.CharSet">UTF-8</parameter>
      <parameter name="inbound.hl7.BuildInvalidMessages">false</parameter>
      <parameter name="inbound.hl7.PassThroughInvalidMessages">false
      </parameter>
   </parameters>
</inboundEndpoint>

<sequence xmlns="http://ws.apache.org/ns/synapse" name="ABCHelathcareHL7_to_
SOAP_with_AppAck" onError="ABCHelathcareHL7_Fault">

   <log level="full">
      <property name="Status" value="== HL7 Message Received =="/>
   </log>
   <property name="HL7_APPLICATION_ACK" value="true" scope="axis2"/>
 <!-- data mapping -->
   <call>
      <endpoint name="MedicalRecordsWS">
         <address uri="http://localhost:6060/MedicalRecords"
format="soap11"/>
      </endpoint>
   </call>

   <log level="full">
      <property name="Status" value="== Successful : Sending Application ACK
=="/>
   </log>
```

```
   <property name="HL7_RESULT_MODE" value="ACK" scope="axis2"/>
   <property name="HL7_GENERATE_ACK" value="true" scope="axis2"/>
   <respond/>
</sequence>

<sequence xmlns="http://ws.apache.org/ns/synapse" name="ABCHelathcareHL7_
Fault">

   <log level="full">
      <property name="Fault Message : " value="Application NACK"/>
   </log>

   <property name="HL7_RESULT_MODE" value="NACK" scope="axis2"/>
   <property name="HL7_NACK_MESSAGE" value="error msg" scope="axis2"/>
   <respond/>

</sequence>
```

The configuration of the HL7 inbound endpoint is very similar to the previous use case, other than disabling the auto ACK flag. In the injecting sequence, we set `<property name="HL7_APPLICATION_ACK" value="true" scope="axis2"/>` to make sure that this is an application ACK scenario. Then you send the message to the backend service after doing the required data mapping. Once you receive the response, you are all set to send back the ACK to the original HL7 source. This can be done by setting the HL7 result mode, setting the property to generate an ACK, and responding with the ACK. Similarly, in a fault scenario, the on-error sequence will be triggered and a NACK can be triggered from the corresponding fault sequence.

Sending HL7 Messages

From ESB you can send HL7 messages to a backend HL7 system. WSO2 ESB can either work as the HL7 gateway or non-HL7 to HL7 message gateway. Suppose that we need to connect two different healthcare information systems that use HL7 as the communication protocol. And assume that you can't directly connect these two systems, as some message level transformations are required between these two systems or you merely want to use an HL7 gateway for monitoring and analytics purposes. So, as illustrated in Figure 8-7, you can use WSO2 ESB as the HL7 gateway between the two healthcare information systems—ABC Healthcare and XYZ Healthcare.

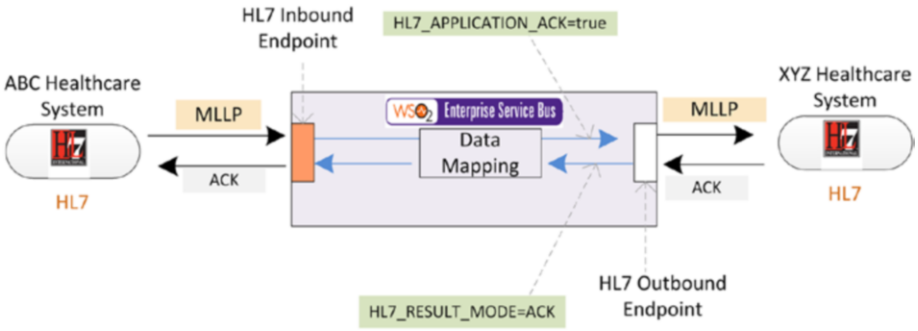

Figure 8-7. *Connecting two HL7-based healthcare systems with WSO2 ESB*

In this scenario, the HL7 transport sender is used to send the message out to the destination HL7 information system. The configuration of this scenario is shown in Listing 8-11. The HL7 messages are received through the HL7 inbound endpoint and the injecting sequence has the logic related to message sending to the destination HL7 system. Here we use the application ACK mode and data mapping is also used between two systems. The message type is configured to `application/edi-hl7`.

Listing 8-11. Receiving and Sending HL7 Messages Between Two HL7 Based Healthcare Information Systems

```
<inboundEndpoint xmlns="http://ws.apache.org/ns/synapse"
                 name="HL7_GW_Receiver"
                 sequence="HL7_GW_Seq"
                 onError="fault"
                 protocol="hl7"
                 suspend="false">
   <parameters>
      <parameter name="inbound.hl7.Port">20002</parameter>
      <parameter name="inbound.hl7.AutoAck">false</parameter>
      <parameter name="inbound.hl7.ValidateMessage">true</parameter>
      <parameter name="inbound.hl7.TimeOut">10000</parameter>
      <parameter name="inbound.hl7.CharSet">UTF-8</parameter>
      <parameter name="inbound.hl7.BuildInvalidMessages">false</parameter>
      <parameter name="inbound.hl7.PassThroughInvalidMessages">false</
      parameter>
   </parameters>
</inboundEndpoint>

<sequence xmlns="http://ws.apache.org/ns/synapse" name="HL7_GW_Seq"
onError="ABCHelathcareHL7_Fault">
```

```
   <log level="full">
      <property name="Status" value="== HL7 Message Received ==" />
   </log>
   <property name="HL7_APPLICATION_ACK" value="true" scope="axis2" />

   <!-- ... Data Mapper ...-->
   <property name="messageType" value="application/edi-hl7" scope="axis2" />

   <call>
      <endpoint name="XYZ_Healthcare_EP">
         <address uri="hl7://localhost:9988" />
      </endpoint>
   </call>

   <log level="full">
      <property name="Status" value="== Successful : Sending Application ACK
==" />
   </log>

   <property name="HL7_RESULT_MODE" value="ACK" scope="axis2" />

   <respond />

</sequence>
```

In addition to HL7 v2.x, WSO2 ESB offers support for other protocols widely used in the healthcare sector.

FHIR-Based Integration

Fast Healthcare Interoperability Resources (FHIR; see `hl7.org/fhir`) is a next generation standards framework created by HL7. FHIR combines the best features of HL7 v2, HL7 v3, and the CDA product lines while leveraging the latest web standards and applying a tight focus on implementability.

FHIR solutions are built from a set of modular components called "Resources." These resources can easily be assembled into working systems that solve real-world clinical and administrative problems at a fraction of the price of existing alternatives. FHIR is suitable for use in a wide variety of contexts—mobile phone apps, cloud communications, EHR-based data sharing, server communication in large institutional healthcare providers, and much more.

WSO2 ESB offers an FHIR connector (https://docs.wso2.com/display/ESBCONNECTORS/FHIR+REST+Connector) that can talk to FHIR-based systems. The details of how to use connectors with WSO2 ESB are covered in the latter part of this chapter.

WebSockets Support

WebSockets is by no means a proprietary protocol, but owing to its unique characteristics over other protocols, we have listed it in this section. Historically, creating web applications that need bidirectional communication (clients send requests to the server as well as the server can send requests to the client) has been a major challenge. The workarounds that were used to achieve this over existing protocols, such as HTTP, turned out to be inefficient.

- *Polling:* With this, the client polls the server periodically and the server immediately responds (with new data or no data). This method is extremely inefficient as it wastes CPU and bandwidth when there's no data available.
- *Long polling:* This is an improved version of polling in which the server doesn't respond immediately; rather it waits until it has some new data available. Once the new data is available, it sends it back to the client. The client immediately sends a request again and long polling continues. Again this method wastes the bandwidth and CPU for opening numerous connections over time.
- *Streaming:* With streaming, the client initiates the connection and sends the initial request. The server waits until it has new data to be sent. Once the new data is available, it keeps sending it to the client. The connection is open forever. This method allows servers to send events infinitely, but can't support client to server messages (hence, this is half-duplex).

Apart from the limitations associated with these methods, using HTTP headers in each message is also another major overhead. So, most cases a lot of bandwidth is wasted because of the unnecessary header data. On the other hand, connection establishment and closure is also extremely expensive. Therefore, it's necessary to have a standard protocol that can cater to these needs and it should be compatible with the existing protocols.

The WebSockets protocol addresses these requirements by using a single TCP connection for traffic in both directions and uses HTTP as the initial handshaking protocol, so that it can work with the existing infrastructure.

Before we jump into the WebSockets integration scenarios with WSO2 ESB, let's take a quick look at how the WebSockets protocol works in typical messaging scenarios.

As depicted in Figure 8-8, the protocol has two main parts, the handshake and the data transfer.

- The client initiates a connection to the server and sends an HTTP request. This is an HTTP GET request with a request of upgrading the connection to WebSockets. This is informed to the server by sending `Upgrade: websocket` and `Connection: Upgrade` by the client. The client also sends other HTTP headers related to the WebSockets handshake.

- The server responds with HTTP 101 with the switching protocols response. This completes the handshake process.
- Now the connection has been established and the client or server can send messages in the full-duplex manner using the same connection. There's no notion of request-response messaging during WebSockets data transfer.
- Either side can terminate the established connection.

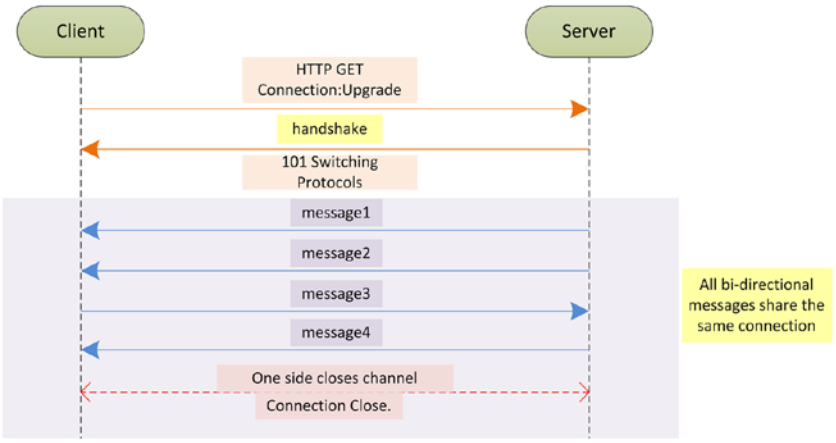

Figure 8-8. *WebSocket protocol overview*

The messages used in WebSockets can be in text or binary format.

Since you have a basic understanding of how the WebSockets protocol works, we can proceed to the WebSockets integration capabilities of WSO2 ESB.

WebSockets to WebSocket Integration

Suppose that you need to integrate two WebSockets-based systems—an e-commerce web portal and an order-management system. The internal WebSockets-based OrderManagement service cannot be directly exposed to the external clients and it has to be exposed via an intermediate WebSocket gateway. Therefore, the core requirement here is to implement a WebSocket gateway with WSO2 ESB, which facilitates the messaging between these two systems.

WSO2 ESB provides a WebSocket inbound endpoint, in which you can use to expose a WebSockets interface from WSO2 ESB. For outbound WebSocket invocations, you can use the WebSocket transport sender. In order to use the WebSocket sender, you need to enable it in the `axis2.xml` file.

As illustrated in Figure 8-9, a WebSocket interface is exposed to the e-commerce web portal through a WebSockets inbound endpoint. An outbound endpoint is used with the WebSockets transport sender to connect to the OrderManager WebSockets service.

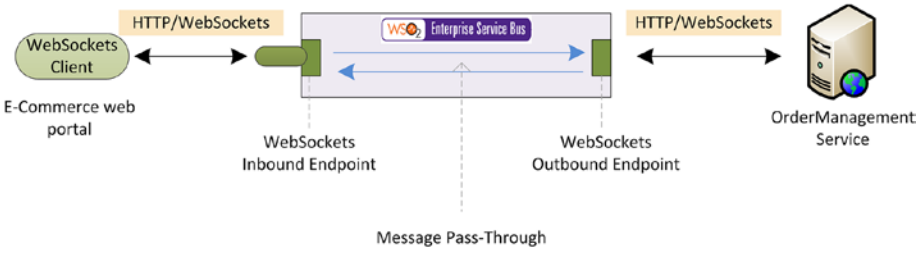

Figure 8-9. *Using WSO2 ESB as the WebSockets Message Gateway*

As you learned in the previous section, the WebSockets messaging pattern requires a handshake prior to data exchange. So, in the scenario shown in Figure 8-9, we configure the message flow so that all the handshake messages as well as data exchanging message flows through the sequences independently. However, there is an important aspect to consider. WebSockets is not a request-response protocol; therefore, there should be a specific way to handle server-initiated messages or *server pushes*. Let's take a closer look at the configuration of this integration scenario, which is shown in Listing 8-12.

Listing 8-12. WebSockets to WebSockets Integration with WebSockets Inbound Endpoint and WebSockets Transport Sender

```
<inboundEndpoint name="ECommerce_WebSockReceiver" onError="fault"
protocol="ws"
                 sequence="orderManagementDispatchSeq" suspend="false">
   <parameters>
      <parameter name="inbound.ws.port">9091</parameter>
      <parameter name="ws.outflow.dispatch.sequence">orderManagementServerP
      ushSeq</parameter>
      <parameter name="ws.client.side.broadcast.level">0</parameter>
      <parameter name="ws.outflow.dispatch.fault.sequence">fault</parameter>
   </parameters>
</inboundEndpoint>

<sequence xmlns="http://ws.apache.org/ns/synapse" name="orderManagementDisp
atchSeq">

   <log level="custom">
      <property name="Message Flow" value="WS-Request message"/>
   </log>

   <property name="OUT_ONLY" value="true"/>
   <call>
      <endpoint>
         <address uri="ws://localhost:8082/wsoc/OrderMgtService"/>
      </endpoint>
   </call>
</sequence>
```

```
<sequence xmlns="http://ws.apache.org/ns/synapse" name="orderManagementDisp
atchSeq">

   <log level="custom">
      <property name="Message Flow" value="ServerPush message received"/>
   </log>
   <respond/>
</sequence>
```

As shown in Listing 8-12, the WebSockets inbound endpoint has `port` as a parameter, the injecting sequence, which is the sequence that the WebSocket message is injected into, and a dispatching sequence to handle the server push messages. So, as we discussed earlier, since WebSockets is not a pure request-response protocol, all the messages that go from the server to the client go through the dispatching sequence.

- `inbound.ws.port`: The HTTP port where the WebSocket inbound endpoint starts listening for incoming messages.
- `sequence` (attribute of inbound Ep.): The generic inbound endpoint parameter that specifies the injecting sequence. WebSockets messages from the client to the server go through this sequence.
- `ws.outflow.dispatch.sequence`: All the messages that flow from the server to the client (server push messages) go through this sequence.
- `ws.client.side.broadcast.level`: We used this parameter to broadcast messages between different subscribers. (Value 0 means no broadcasting.) This will be explained further in the next use case.
- The outbound endpoint can specify the backend WebSockets service with the prefix `ws://`.

Also note that the injecting sequence has the `OUT_ONLY` parameter enabled, as the messaging is not request-response style. The server-push dispatching sequence handles the server send messages using a respond mediator (however, those messages are not really responses).

So, in this use case, we haven't really dealt with any of the WebSocket message content (data exchange). All the handshake messages and data exchanging messages are sent through the ESB layer.

When we are doing WebSockets to WebSockets integration, it is important to keep in mind that all the message exchanges between the participating WebSocket-based systems are sent through the WSO2 ESB. This includes handshaking messages as well as full-duplex WebSockets data frames. If you want to specifically filter out the handshake-related messages, you need to use a filter mediator that checks for the property `$ctx:websocket.source.handshake.present`. You can find a detailed use case for a similar scenario in the following subprotocol related use cases.

WebSockets to WebSocket Integration with Frame Broadcasting

The WebSockets protocol can be used to implement a publisher-subscriber message pattern so that we can broadcast WebSocket frames to WebSocket clients who are connected based on their connected subscriber path. This is quite useful when we want to distribute data frames over a set of connected clients over an inbound endpoint. To understand the use of subscriber path and frame broadcasting, consider the scenario depicted in Figure 8-10.

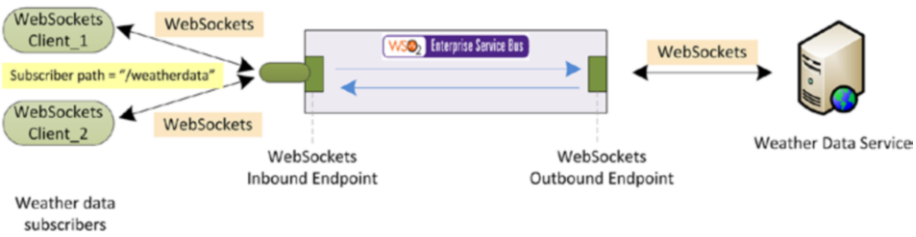

Figure 8-10. *WebSockets frames are broadcasted among the clients that have connected with subscriber path weatherdata*

In this scenario, there is a weather data backend service, which is based on WebSockets. Rather than exposing the weather data service as it is, there is a requirement to expose it through a separate WebSockets interface, which can also support publisher-subscriber messaging. This means that any WebSocket client that is connecting to the WebSocket interface of WSO2 ESB with a given subscriber path will receive messages sent to that particular subscriber path.

The implementation of this scenario is quite similar to the previous use case, but we need to set `ws.client.side.broadcast.level = 2`. By setting this parameter, all the clients that connected with same subscriber path will receive the WebSocket frame, except the one that publishes the frame to inbound. The use of broadcast levels can be summarized as follows:

- *Level 0:* Only the unique client may receive the frame from WebSockets inbound endpoint
- *Level 1:* All the clients that connected with the same subscriber path will receive the WebSocket frame
- *Level 2:* All the clients that connected with same subscriber path will receive the WebSocket frame, except for the one that publishes the frame to inbound

The corresponding configuration of the WebSockets inbound endpoint is shown in Listing 8-13. So, each client that connected to the WebSockets inbound endpoint with the subscriber path `http://host:port/weatherdata` will receive the WebSockets frames broadcasted for that particular subscriber path. The configuration of the injecting sequence and the dispatching (server-side message handling) sequence is quite similar to the previous use case.

Listing 8-13. WebSockets to WebSockets Integration with Subscriptions

```
<inboundEndpoint name="WeatherInfo_WebSockBroadcaster" onError="fault"
protocol="ws"
                 sequence="weatherDataReq_Seq" suspend="false">
   <parameters>
      <parameter name="inbound.ws.port">9092</parameter>
      <parameter name="ws.outflow.dispatch.sequence">weatherDataServerPush_
      Seq</parameter>
      <parameter name="ws.client.side.broadcast.level">2</parameter>
      <parameter name="ws.outflow.dispatch.fault.sequence">fault</parameter>
   </parameters>
</inboundEndpoint>

<sequence xmlns="http://ws.apache.org/ns/synapse" name="weatherDataReq_Seq">

   <log level="custom">
      <property name="Message Flow" value="WS-Request message"/>
   </log>

   <property name="OUT_ONLY" value="true"/>
   <call>
      <endpoint>
         <address uri="ws://localhost:8082/wsoc/OrderMgtService"/>
      </endpoint>
   </call>
</sequence>

<sequence xmlns="http://ws.apache.org/ns/synapse"
name="weatherDataServerPush_Seq">

   <log level="custom">
      <property name="Message Flow" value="ServerPush message received"/>
   </log>
   <respond/>
</sequence>
```

So far, the use cases that we have discussed are not processing the data that's transferred as WebSocket frames. Rather we simply pass through it from the ESB layer. In the next section, let's take a look at the scenarios where you need to read and process (i.e., transform) the content of WebSocket frames.

WebSockets to WebSocket Integration with Data Mapping

The WebSockets specification doesn't specify any details about content types of the frames that flow through WebSockets channels. Unlike HTTP, which is an application-level protocol, in the WebSocket protocol there is not enough information in an incoming

message to process these messages. These messages are either text or binary low-level frames.

Because of this, WSO2 ESB has defined a custom subprotocol over WebSockets. At the initial handshake of the WebSocket channel creation, we exchange the content type subprotocol type as an HTTP header. This will allow both client server parties to know the content type of frames they communicate.

> Subprotocols
>
> *The client can request that the server use a specific subprotocol by including the |Sec-WebSocket-Protocol| field in its handshake. If it is specified, the server needs to include the same field and one of the selected subprotocol values in its response for the connection to be established.*
>
> Source: https://tools.ietf.org/html/rfc6455#page-4

To understand the concept of content handling with WSO2 ESB's custom subprotocol, let's suppose that the e-commerce and OrderManagement service scenario that we discussed earlier has a requirement to do some data transformation of the WebSocket frames. In other words, the WebSocket frames sent from the e-commerce web portal cannot be directly sent to the OrderManagement service. Therefore, inside the ESB message flow, you need to transform it using the data mapper.

As shown in Figure 8-11, the message format of the content of WebSockets frames is JSON. The transformation required at the ESB layer is JSON-to-JSON transformation and the transformed messages are also sent in the form of WebSockets frames. When the e-commerce web portal sends a WebSocket frame to the WSO2 ESB, there should be a way to determine the content type of the WebSocket frame content. In order to support that WSO2 ESB requires having the subprotocol header (WSO2 ESB-defined) that is related to the content type during the handshake. That is `Sec-WebSocket-Protocol: synapse(contentType='application/json')`. When the handshake happens between the e-commerce web portal and the WSO2 ESB WebSockets listener, the e-commerce web portal should send this header.

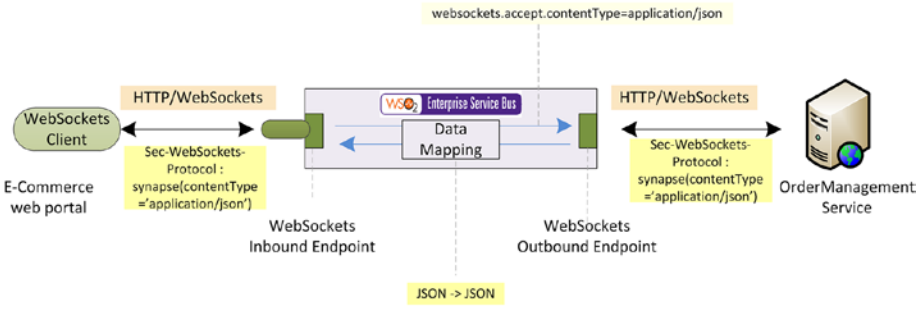

Figure 8-11. *WebSocket frames from a e-commerce web portal are transformed using the data mapper before we send them to the OrderManagement service. Both systems use the custom subprotocol header to transfer the content-type related data.*

Once the handshake is completed, the WebSocket inbound endpoint can build all the subsequent WebSocket frames based on the specified content type during the initial handshake. Therefore, inside the ESB message flow (injecting sequence), you can use the data mapper to map the message format to the message format of the `OrderManagementService`.

As mentioned earlier, all the messages related to WebSocket communication between the two systems flows through WSO2 ESB. This includes handshake-related messages as well as data frames. Therefore, the data mapping logic has to be explicitly done for WebSockets frames only. So, as shown in Listing 8-14, we have filtered the WebSockets frames using the `$ctx:websocket.source.handshake.present` property and done the data mapping accordingly.

In addition, when we send WebSockets frames from ESB to the OrderManagement system, we also have to send the WebSockets frames with a content type that we have transformed in our mediation flow (i.e., JSON in this use case). This is done by using the `websocket.accept.contenType` property defined at the `axis2` scope. We used this property to inform the WebSocket sender to build the frames with the JSON content type and to include the same subprotocol header that we used to determine the content of the WebSockets frames.

Listing 8-14. WebSockets to WebSocket Integration Using Subprotocols to Support Data Mapping

```
<inboundEndpoint name="ECommerce_WebSockReceiver" onError="fault"
protocol="ws"
                 sequence="orderManagementReqDataMapper_Seq"
suspend="false">
   <parameters>
      <parameter name="inbound.ws.port">9093</parameter>
      <parameter name="ws.outflow.dispatch.sequence">orderManagementServerPu
      shDataMapper_Seq</parameter>
      <parameter name="ws.client.side.broadcast.level">0</parameter>
      <parameter name="ws.outflow.dispatch.fault.sequence">fault</parameter>
   </parameters>
</inboundEndpoint>

<sequence xmlns="http://ws.apache.org/ns/synapse" name="orderManagementReqD
ataMapper_Seq">

   <property name="OUT_ONLY" value="true"/>
   <property name="websocket.accept.contenType" scope="axis2"
value="application/json"/>

   <switch source="$ctx:websocket.source.handshake.present">
      <case regex="true">
         <!-- no data mapping for handshake messages -->
      </case>
      <default>
         <!-- ... Data Mapping applied for all WebSockets data frames ...-->
```

```
      </default>
   </switch>

   <call>
      <endpoint>
         <address uri="ws://localhost:8082/websocket"/>
      </endpoint>
   </call>

</sequence>

<sequence xmlns="http://ws.apache.org/ns/synapse" name="orderManagementServe
rPushDataMapper_Seq">

   <log level="custom">
      <property name="Message Flow" value="ServerPush message received"/>
   </log>

   <switch source="$ctx:websocket.source.handshake.present">
      <case regex="true">
         <!-- no data mapping for handshake messages -->
      </case>
      <default>
         <!-- ... Data Mapping applied for all WebSockets data frames ...-->
      </default>
   </switch>

   <respond/>
</sequence>
```

As you have seen, one of the main requirements for building this kind of integration scenario is to support the custom subprotocol header that the WSO2 ESB defines, to cater to content-aware scenarios with WebSockets frames.

WebSockets to HTTP Integration

One of the main requirements in the WebSockets integration domain is to expose a non-WebSockets HTTP services as a WebSockets-enabled service through WSO2 ESB. Let's assume that the previous OrderManagement is not based on WebSockets, but rather based on conventional HTTP 1.x. Now you want to expose the same functionality via the WebSockets interface.

You can implement this use case with WSO2 ESB by simply introducing a WebSockets inbound endpoint. As illustrated in Figure 8-12, since you have to map the WebSockets data frames into an HTTP request, it is necessary to have the information related to the content type as part of a subprotocol header during the handshake.

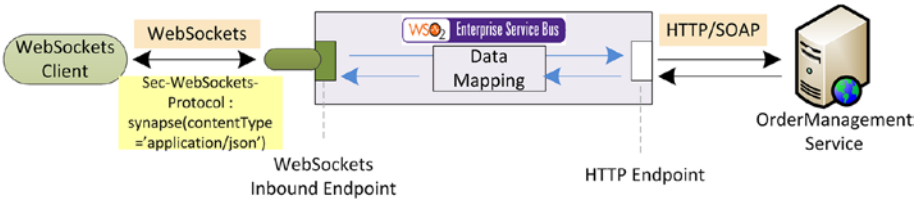

Figure 8-12. *Exposing a WebSockets interface for an existing non-WebSockets based HTTP service*

The injecting sequence can simply ignore the handshake related messages, as a handshake is not required to be propagated by the HTTP OrderManagement service, which doesn't use HTTP. As shown in Listing 8-15, the only significant logic that you require at the injecting sequence of WebSockets inbound endpoint is to filter out and drop the handshake messages. WebSockets frames are transformed and sent as HTTP requests to the backend service. The responses are translated back to server-sent messages at the WebSockets inbound endpoint.

Listing 8-15. Exposing HTTP Backend Service as a WebSocket Interface

```
<inboundEndpoint name="OrderManager_WebSockListener" onError="fault"
protocol="ws"
                 sequence="orderManagementReq_Seq" suspend="false">
   <parameters>
      <parameter name="inbound.ws.port">9091</parameter>
      <parameter name="ws.outflow.dispatch.sequence">orderManagementServer
      PushSeq</parameter>
      <parameter name="ws.client.side.broadcast.level">0</parameter>
      <parameter name="ws.outflow.dispatch.fault.sequence">fault</parameter>
   </parameters>
</inboundEndpoint>

<sequence xmlns="http://ws.apache.org/ns/synapse" name="OrderManager_
WSockToHTTP">
   <switch source="$ctx:websocket.source.handshake.present">
      <case regex="true">
         <!-- Ignoring handshake messages -->
         <drop/>
      </case>
      <default>
         <!-- ... Data Mapper ...-->
         <call>
            <endpoint>
               <http uri-template="http://localhost:9090/
               OrderManagementRESTAPI" method="POST"/>
            </endpoint>
         </call>
```

```
            <!-- ... Data Mapper ...-->
            <respond/>
        </default>
    </switch>
</sequence>
```

However, with this approach, your WebSocket interface is fully constrained by the underlying HTTP service. For instance, your WebSockets interface won't be sending any WebSockets frames that are initiated by the backend service. Rather, all messaging exchanges take place in a request-response style messaging.

HTTP to WebSockets Integration

Exposing an HTTP interface on top of an existing WebSocket service may sound like a rare requirement. Still there can be such use cases and hence we have included that as the last use case related to WebSockets. Unlike the previous scenario, where we implemented WebSockets to HTTP, this scenario is even more constrained due to the mismatch of these two protocols.

Suppose that a legacy client application wants to consume your WebSockets-based OrderManagement service. Since the client doesn't support the WebSockets protocol, you need to expose an HTTP 1.x interface as a REST API to the client.

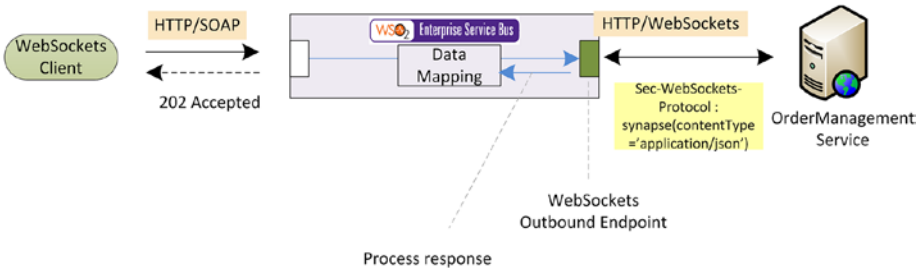

Figure 8-13. *Exposing an HTTP interface on top of an existing WebSockets service*

However, we can't implement a request-response messaging pattern in this particular scenario. That means that the request from the HTTP client should be a one-way message with 202 Accepted responses. So the HTTP client can only send one-way HTTP messages, which are translated back to ESB to OrderManagement service WebSockets frames. The related configuration is shown in Listing 8-16.

Listing 8-16. Exposing WebSockets Services as an HTTP 1.x Interface

```
<api xmlns="http://ws.apache.org/ns/synapse"
     name="OrderManagementAPI"
     context="/ordermgt">
    <resource methods="POST">
        <inSequence>
```

```
            <!-- ... data mapping ... -->
            <property name="OUT_ONLY" value="true"/>
            <property name="FORCE_SC_ACCEPTED" scope="axis2"
                      type="STRING" value="true"/>
            <property name="websocket.accept.contenType" scope="axis2"
            value="application/json"/>
            <call>
                <endpoint>
                    <address uri="ws://localhost:8082/websocket"/>
                </endpoint>
            </call>
        </inSequence>
    </resource>
</api>
```

You can simply call the WebSockets service through the WebSockets transport sender, and the inbound HTTP request is treated as a one-way inbound message with the use of the `FORCE_SC_ACCEPTED` and `OUT_ONLY` properties.

In-JVM Calls with Local Transport

There can be situation that ESB has to call its own proxy services or REST APIs from another sequence, proxy service, or REST API. In such a situation, if you call the proxy service or REST API with its URL (e.g., http://locahost:8280/services/MyProxy), it goes through the HTTP transport layer. Hence, even to call an internally hosted REST API or proxy service, there is a redundant network level call that you have to do.

WSO2 ESB supports `in-jvm`/local service calls with its local transport. You simply have to enable local transport in ESB by adding following transport sender and remove all "local" senders and receivers from `axis2.xml`.

```
<transportSender name="local" class="org.apache.axis2.transport.local.
NonBlockingLocalTransportSender"/>
```

You also have to configure the `$SESB_HOME/repository/conf/carbon.xml` file with the following `ServerURL`.

```
<ServerURL>https://${carbon.local.ip}:${carbon.management.port}${carbon.
context}/services/</ServerURL>
```

Once you have done this configuration you can simply call any proxy or REST API with its local/`in-jvm` URL. For example, suppose that you have a proxy service hosted in WSO2 ESB at http://locahost:8280/services/MyProxy. If you wanted to call that service through the local transport, you would use the following configuration.

```
<call>
    <endpoint name="ep2">
        <address uri="local://services/MyProxy"/>
    </endpoint>
</call>
```

It is recommended to use a local transport for all internal invocations of proxy services and REST APIs as it has less overhead since you don't have the redundant network-level innovation.

Integrating Cloud Services

As you learned in the previous section, integrating on-premise systems is a key requirement of an ESB, as any organization has such disparate IT applications. However, with the increasing adaptation of cloud-based software applications, most organizations run significant a portion of their IT businesses on cloud-based Software as a Service (SaaS) applications. For example, if we consider CRM (Customer Relationship Management) software, Salesforce is one of the most popular software solutions. It completely runs as a SaaS application with zero maintenance cost for the consumer organization. In addition to the increasing growth of SaaS solutions, most organizations leverage the business functionalities that are exposed as APIs. For example, popular day-to-day software tools such as Facebook, Twitter, etc. are heavily used along with other software applications. Most of the functionalities of these software solutions are exposed as APIs. So any enterprise might want to integrate their software with SaaS solutions or any public APIs out there.

In summary, the enterprise integration landscape has been moving into a much more dynamic and diverse space. Therefore, it is important for any ESB to support the wide range of applications, APIs, and SaaS solutions.

What is an ESB Connector?

Most of the SaaS solutions and APIs expose standard interfaces to the external parties to consume their business functionalities. For example, Salesforce provides rich SOAP (and REST) APIs for all its business functionalities, while Twitter provides a REST API to carry out all the Twitter-related activities outside its web site (`twitter.com`). So you can configure the ESB to directly consume those APIs based on standard technologies, such as REST or SOAP web services. However, if the ESB users have to build the integration solutions from the ground up using those cloud service APIs, they have to put more effort into dealing with the complexities of each API/SaaS, rather than focusing on their business use case.

Therefore, the ESBs has to provide a convenient and intuitive way to consume those cloud services/APIs from the ESB layer. On the other hand, there can be hundreds or thousands of APIs and SaaS solutions out there, and it is virtually impossible to include support for all those APIs/SaaS solutions in a single ESB product. We need a much more decoupled and scalable way of supporting cloud service integration in an ESB.

Considering all such requirements, WSO2 ESB introduced the concept of a *connector* with following characteristics.

- The WSO2 ESB connector provides a simple abstraction to access the underlying cloud service, SaaS, API, or an application from the ESB message flow. All the complexities of invoking the underlying cloud service are transparent to the ESB users.
- WSO2 ESB connectors are completely decoupled from the ESB runtime. Rather they are built using the extension points of WSO2 ESB and therefore are not coupled to a specific ESB version. They are developed and released independently of WSO2 ESB releases.
- There can be multiple connectors to the same API, which are based on different technologies that a given API is exposed to.

Before moving into the use cases of ESB connectors, it's important to understand how connectors are implemented with the use of the primitive ESB constructs. All the connectors for WSO2 ESB are 100% open source and you can download and use them from the WSO2 Connectors store at `https://store.wso2.com/store/assets/esbconnector`.

Structure of an ESB Connector

ESB connectors are built using sequence and endpoint templates, which you learned about in previous chapters. A connector is a logical entity that contains a collection of templates. Each template represents an operation associated with the underlying cloud service or application. For example, let's suppose we are using the ESB Twitter connector. The Twitter connector has multiple operations and for each operation there is a matching sequence template inside a connector. The sequence template contains the ESB mediation logic that is required to invoke the Twitter API. What you actually do when you use a connector is call these templates from your sequences. However, the way that you call a connector operation is different from the way you call a generic template. The templates available in a connector can be invoked in a different way.

Using an ESB Connector

Suppose that you want to invoke a Twitter update operation from one of your ESB message flows (sequences). You can simply do this by calling the `updateStatus` operation of the Twitter connector. Prior to that, you have to configure the `init` operation, which you pass all the Twitter-specific information of your account. It is also possible to store this information as a local entry and refer to the respective configuration from the Twitter operation.

In Listing 8-17, you can find how you initiate the Twitter connector and then do a status update with that particular account. Also the following example shows how you can use `cofigKey` attribute with a connector operation, which you can use with the connector configurations (things such as keys, secrets, etc.) with a connector operation. With that approach, you don't need to use the `init` operation prior to

invoking a connector operation. In the following use case, we first update the status using kasun's credentials and then the second `updateStatus` operation is done with the `otherTwitterAcc`'s Twitter account.

For each connector operation, you can pass message context variables (starting with `$ctx`) or any other hardcoded parameter. In Listing 8-17, we passed the several variables to the first `updateStatus` operation and the `status` is passed as a static parameter.

Listing 8-17. Using the Twitter Connector

```
<api xmlns="http://ws.apache.org/ns/synapse"
     name="TwitterInvokerAPI"
     context="/twitterinvoker">
    <resource methods="POST">
        <inSequence>
            <!-- Variable extraction -->
            <log level="full"/>
            <twitter.init>
                <consumerKey>{$ctx:consumerKey}</consumerKey>
                <consumerSecret>{$ctx:consumerSecret}</consumerSecret>
                <accessToken>{$ctx:accessToken}</accessToken>
                <accessTokenSecret>{$ctx:accessTokenSecret}
                </accessTokenSecret>
            </twitter.init>

            <twitter.updateStatus>
                <status>This is kasun's status update</status>
                <inReplyToStatusId>{$ctx:inReplyToStatusId}
                </inReplyToStatusId>
                <possiblySensitive>{$ctx:possiblySensitive}
                </possiblySensitive>
                <latitude>{$ctx:latitude}</latitude>
                <longitude>{$ctx:longitude}</longitude>
                <placeId>{$ctx:placeId}</placeId>
                <displayCoordinates>{$ctx:displayCoordinates}
                </displayCoordinates>
                <trimUser>{$ctx:trimUser}</trimUser>
                <mediaIds>{$ctx:mediaIds}</mediaIds>
            </twitter.updateStatus>

            <twitter.updateStatus configKey="otherTwitterAcc">
                <status>{$ctx:status}</status>
                <inReplyToStatusId>{$ctx:inReplyToStatusId}
                </inReplyToStatusId>
                <possiblySensitive>{$ctx:possiblySensitive}
                </possiblySensitive>
                <latitude>{$ctx:latitude}</latitude>
                <longitude>{$ctx:longitude}</longitude>
                <placeId>{$ctx:placeId}</placeId>
```

```
                <displayCoordinates>{$ctx:displayCoordinates}
                </displayCoordinates>
                <trimUser>{$ctx:trimUser}</trimUser>
                <mediaIds>{$ctx:mediaIds}</mediaIds>
            </twitter.updateStatus>
        </inSequence>
    </resource>
</api>
```

When you have to pass things such as passwords, secrets, keys, etc., it's not a good practice to have them be part of the plain configuration. You have to keep such information in a secured storage in WSO2 ESB. For this purpose, WSO2 ESB provides a Secure Vault tool to store your sensitive information in a secured storage in ESB and refers to those things using aliases.

For example, suppose that you want to securely store the `consumerSecret` parameter value of the Twitter operation. You can use Secure Vault tool to store the actual value of that parameter in WSO2 ESB (Refer to https://docs.wso2.com/display/ESB500/Working+with+Passwords to see how you can configure a secure vault.) Let's say you stored that value with the alias `kasun.consumer.secret` in the ESB secure vault. Now you can use that alias in your connector configuration, as shown in Listing 8-18.

Listing 8-18. Using Secure Vault with Connector Parameters

```
<twitter.init>
    <consumerKey>{$ctx:consumerKey}</consumerKey>
    <consumerSecret>{wso2:vault-lookup('kasun.consumer.secret')}
    </consumerSecret>
    <accessToken>{$ctx:accessToken}</accessToken>
    <accessTokenSecret>{$ctx:accessTokenSecret}</accessTokenSecret>
</twitter.init>
```

There can be some connector operations that have complex payloads that are more complicated than simple string-based parameters. Also, some connector operations may return payloads with complex message structures. For example, let's consider a Salesforce integration use case. Here we call a Salesforce query operation to retrieve a set of `Accounts` from Salesforce. The return message payload comprises a list of account information in a XML/SOAP message format. Therefor, after invoking the connector operation, you can apply any generic mediation logic inside your sequence.

Listing 8-19. Handling Message Payloads Returned from Connector Operations

```
<sequence xmlns="http://ws.apache.org/ns/synapse" name="salesforceInfoExtra
ctor">

   <log level="custom">
      <property name="Message Flow" value="Calling Salesforce..."/>
   </log>
```

```
   <salesforce.query configKey="fooSFCredentials">
      <batchSize>200</batchSize>
      <queryString>select id,name from Account</queryString>
   </salesforce.query>

   <!-- mediation logic : data mapping, filtering, splitting etc.  -->
</sequence>
```

Similarly, you can pass complex message formats as a part of the connector parameters. In the Salesforce `Upsert` operation shown in Listing 8-20, we used a payload factory to create the message payload that is required for the Salesforce `Upsert` operation.

Listing 8-20. Using Complex Message Format as a Connector Parameters

```
<sequence xmlns="http://ws.apache.org/ns/synapse" name="salesforceAddSeq">
   <log level="custom">
      <property name="Message Flow" value="Calling Salesforce upsert"/>
   </log>
   <payloadFactory>
      <format>
         <sfdc:sObjects xmlns:sfdc="sfdc" type="Account">
            <sfdc:sObject>
               <sfdc:Id>0019000000aaMkZ</sfdc:Id>
               <sfdc:Name>newname001</sfdc:Name>
            </sfdc:sObject>
            <sfdc:sObject>
               <sfdc:Name>newname002</sfdc:Name>
            </sfdc:sObject>
         </sfdc:sObjects>
      </format>
      <args/>
   </payloadFactory>
   <salesforce.upsert>
      <allOrNone>0</allOrNone>
      <allowFieldTruncate>0</allowFieldTruncate>
      <externalId>Id</externalId>
      <sobjects xmlns:sfdc="sfdc">{//sfdc:sObjects}</sobjects>
   </salesforce.upsert>

   <!-- mediation logic : data mapping, filtering, splitting etc.  -->
</sequence>
```

In order to use a connector and invoke connector operations, you don't really need to go through all the methods and parameters at the XML configuration level. Rather, if you use WSO2 ESB development tool, you can simply select the connector that you want to use, then select the operation that you want to use, and its respective parameters are shown in the visual editor. We will cover how to use connectors with ESB development tool as part of Chapter 10, in which we talk about the WSO2 ESB development process.

When you use connectors with data mapper, you can use the static scheme or dynamic schema available as part of the connector. The concept of a connector schema is applied for each connector operation, and it defines the message structure of the message that a connector operation accepts (incoming message format) and the message structure of the response (outcoming/returning message format). There are connectors that support static message formats as well as connectors such as Salesforce, which has a dynamic/variable schema. For those connectors, the developer tool connects to those APIs to retrieve the metadata and determine the message formats accordingly.

Inbound Connectors

You learned the concepts behind an inbound endpoint during the past chapters. The concept of an inbound connector is built on top of the inbound endpoint concept. Let's take an example from Salesforce to understand the concept of an inbound connector. The Salesforce standard API allows you to invoke various operations related to its business use cases. So, that is a pull API, where clients pull information from Salesforce. Salesforce also provides a "streaming" API that the consumers can use to receive notifications for changes to Salesforce data that match a SQL query you define, in a secure and scalable way.

To support such streaming APIs, WSO2 ESB introduced the concept of the inbound connector. An inbound connector is essentially an inbound endpoint, which is totally decoupled from ESB runtime and can be dynamically deployed into an ESB runtime. The runtime behavior of an inbound connector is almost the same as an inbound endpoint. You can configure the inbound connector and specify an injecting sequence to process the messages received through the streaming API.

In the following example, you can find a use case that we connect the WSO2 ESB to the Salesforce streaming API. The injecting sequence handles the streamed/pushed messages from Salesforce. Prior to using the streaming API, it was required to configure the "push topics" at the Salesforce API level. Listing 8-21 shows a sample configuration of a Salesforce streaming connector.

Listing 8-21. Salesforce Streaming Connector: Sample Configuration

```
<inboundEndpoint xmlns="http://ws.apache.org/ns/synapse"
                 name="SaleforceInboundEP"
                 sequence="saleforceInvoiceStreamSeq"
                 onError="fault"
                 class="org.wso2.carbon.inbound.salesforce.poll.
                 SalesforceStreamData"
                 suspend="false">
   <parameters>
    <parameter name="sequential">true</parameter>
    <parameter name="interval">10</parameter>
    <parameter name="coordination">true</parameter>
    <parameter name="connection.salesforce.userName">xxx@gmail.com
    </parameter>
```

```
    <parameter name="connection.salesforce.loginEndpoint">https://login.
    salesforce.com</parameter>
    <parameter name="connection.salesforce.password">xxxxYYYY</parameter>
    <parameter name="connection.salesforce.salesforceObject">InvoiceStatemen
    tUpdates</parameter>
    <parameter name="connection.salesforce.connectionTimeout">20000
    </parameter>
    <parameter name="connection.salesforce.readTimeout">110000</parameter>
    <parameter name="connection.salesforce.waitTime">10000</parameter>
    <parameter name="connection.salesforce.packageName">cometd</parameter>
    <parameter name="connection.salesforce.packageVersion">35.0</parameter>
    <parameter name="connection.salesforce.soapApiVersion">22.0</parameter>
 </parameters>
</inboundEndpoint>
```

Refer to https://docs.wso2.com/display/ESBCONNECTORS/Configuring+Salesforce+Streaming+Connector+Operations for more details on how to do this at the Salesforce level. Once you have done that, you can simply deploy the inbound connector on the ESB side and then the injecting sequence can process all the pushed/streamed messages from Salesforce.

Integrating Salesforce and SAP

Now you have a good understanding of the fundamentals of ESB connectors and how you can use connectors in ESB mediation flow. Let's next consider how a real-world integration use case can be implemented with ESB connectors. Suppose that you have to implement a scenario where you want to send information from the SAP system to your Salesforce CRM. Suppose that the customer information resides in SAP and has to be added and synced with the customer information of your Salesforce CRM. For that, you can use SAP Idoc-based integration from the SAP end and WSO2 ESB can be configured to receive the "Customer Master Idoc" at the ESB layer. Once we receive the customer master IDoc at the ESB level, we can configure the data mapping and then invoke the Salesforce update operation.

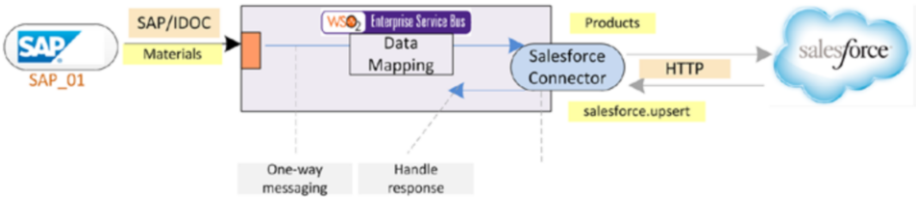

Figure 8-14. *Replicating customers available in SAP in Salesforce customers through SAP IDocs*

The configuration at the SAP IDoc integration is discussed during the SAP integration section in this chapter. Listing 8-22 shows a sample configuration of the injecting sequence to which the SAP inbound endpoint injects. The main highlights are the data mapping from the SAP IDoc XML format to Salesforce SOAP message format and the invocation of the relevant Salesforce connector operations.

Listing 8-22. Transferring Customers Available in SAP to Salesforce

```
<sequence xmlns="http://ws.apache.org/ns/synapse" name="SAP_to_Salesforce">
    <property description="tourID" expression="//TOUR_ID" name="tourID"
    scope="default" type="STRING"/>
    <property name="APOS" scope="default" type="STRING" value="'"/>
    <property description="queryString"
              expression="fn:concat('SELECT Name,ID,organization__c,us
              ername__c,queryType__c,mission__c,tourID__c,approved__c,
              orderID__c,orderItem__c FROM Request__c WHERE tourID__c
              =',syn:get-property('APOS'),get-property('tourID'),syn:get-
              property('APOS'))"
              name="QUERY" scope="default" type="STRING"/>
    <salesforce.query configKey="SFConfig">
        <batchSize>1</batchSize>
        <queryString>{$ctx:QUERY}</queryString>
    </salesforce.query>
    <property description="Id" expression="//sf:Id" name="Id"
    scope="default" type="STRING"
              xmlns:sf="urn:sobject.partner.soap.sforce.com"/>
    <property description="IdModified" expression="fn:substring
    (get-property('Id'),0,16)" name="IdModi" scope="default"
              type="STRING"/>
    <payloadFactory media-type="xml">
        <format>
            <sfdc:sobjects type="Request__c" xmlns:sfdc="sfdc">
                <sfdc:sobject>
                    <sfdc:tourID__c>$1</sfdc:tourID__c>
                    <sfdc:Name>$2</sfdc:Name>
                    <sfdc:Id>$3</sfdc:Id>
                    <sfdc:organization__c>$4</sfdc:organization__c>
                    <sfdc:username__c>$5</sfdc:username__c>
                    <sfdc:queryType__c>$6</sfdc:queryType__c>
                    <sfdc:mission__c>$7</sfdc:mission__c>
                    <sfdc:orderID__c>$8</sfdc:orderID__c>
                    <sfdc:orderItem__c>$9</sfdc:orderItem__c>
                    <sfdc:approved__c>1</sfdc:approved__c>
                </sfdc:sobject>
            </sfdc:sobjects>
        </format>
        <args>
            <arg evaluator="xml" expression="get-property('tourID')"/>
```

```
            <arg evaluator="xml" expression="//sf:Name"
            xmlns:sf="urn:sobject.partner.soap.sforce.com"/>
            <arg evaluator="xml" expression="get-property('IdModi')"/>
            <arg evaluator="xml" expression="//sf:organization__c"
            xmlns:sf="urn:sobject.partner.soap.sforce.com"/>
            <arg evaluator="xml" expression="//sf:username__c"
            xmlns:sf="urn:sobject.partner.soap.sforce.com"/>
            <arg evaluator="xml" expression="//sf:queryType__c"
            xmlns:sf="urn:sobject.partner.soap.sforce.com"/>
            <arg evaluator="xml" expression="//sf:mission__c"
            xmlns:sf="urn:sobject.partner.soap.sforce.com"/>
            <arg evaluator="xml" expression="//sf:orderID__c"
            xmlns:sf="urn:sobject.partner.soap.sforce.com"/>
            <arg evaluator="xml" expression="//sf:orderItem__c"
            xmlns:sf="urn:sobject.partner.soap.sforce.com"/>
        </args>
    </payloadFactory>
    <salesforce.update configKey="SFConfig">
        <allOrNone>0</allOrNone>
        <allowFieldTruncate>0</allowFieldTruncate>
        <sobjects xmlns:sfdc="sfdc">{//sfdc:sobjects}</sobjects>
    </salesforce.update>
    <drop/>
</sequence>
```

Since inbound connectors are essentially custom inbound endpoints, the clustering mechanism is exactly the same. (Inbound endpoint clustered deployed is discussed in detail in the deployment patterns section in Chapter 10.)

Data Integration

In any organization, the information is stored in some form of persistent storage. In most cases, the main information storage mechanism is databases. Therefore, it is often required to integrate the data that resides in databases with other systems in your enterprise IT landscape. In order to connect databases with other systems, WSO2 provides the capability to expose a database operation as a SOAP web service or as a RESTful service.

This is included with the WSO2 DSS feature and you can install the DSS feature inside ESB and start connecting to a database. Basically, the service exposed from the data services feature can be called from an ESB mediation sequence.

The data services feature offers a broad range of data integration capabilities than just exposing a SOAP or RESTful service, but in this section, we focus on that feature only. To understand the basic concepts of how to build a data service, consider the following configuration. Listing 8-23 shows a data service configuration of a data service called `OrderService`. Once you deploy the following configuration as a data service, it will deploy a SOAP web service in your WSO2 ESB + DSS instance (DSS features are installed into your ESB).

If you look at the data service configuration, you can find the name of service at the starting element, and a `config` section in which you configure all the details of the data source/database that you want to connect to. After the `config` section, you can see a query section, which contains the query that you want to run against your database tables. In this case, we are executing an `insert` operation and the required parameters are specified as part of the query. Then comes the `operation` section, where you configure the actual service operation, which is exposed by the service. From the `AddOrder` operation, the respective query is referred and parameters are specified, which will become the elements of the incoming SOAP message. Basically, once the service is deployed, you can invoke it by invoking the `AddOrder` operation with the SOAP message format relevant to the operation parameters that you specified.

Listing 8-23. Data Service Configuration of OrderService

```
<data name="OrderService" enableBatchRequests="true"
serviceNamespace="http://wso2.org/sample/shop/order">
   <config id="shopdatasource">
      <property name="org.wso2.ws.dataservice.driver">com.mysql.jdbc.
      Driver</property>
      <property name="org.wso2.ws.dataservice.protocol">jdbc:mysql://
      localhost:3306/SHOP_DB</property>
      <property name="org.wso2.ws.dataservice.user">root</property>
      <property name="org.wso2.ws.dataservice.password">root</property>
   </config>
   <query id="addOrderQuery" useConfig="shopdatasource">
      <sql>insert into ORDER_T values (:orderID, :customerID,
      :date,:price);</sql>
      <param name="orderID" sqlType="STRING" />
      <param name="customerID" sqlType="STRING" />
      <param name="date" sqlType="TIMESTAMP" />
      <param name="price" sqlType="DOUBLE" />
   </query>
   <operation name="AddOrder">
      <call-query href="addOrderQuery">
         <with-param name="orderID" query-param="orderID" />
         <with-param name="customerID" query-param="customerID" />
         <with-param name="date" query-param="date" />
         <with-param name="price" query-param="price" />
      </call-query>
   </operation>
</data>
```

You can learn more about WSO2's data service feature/server at https://docs.wso2.com/display/DSS350/WSO2+Data+Services+Server+Documentation.

You can use the data service feature for most of the real-world integration use cases. For example, consider the use case specified in Listing 8-23, where you need to integrate Salesforce's streaming API. All the pushed "invoice statement updates" have to be stored in an Oracle database.

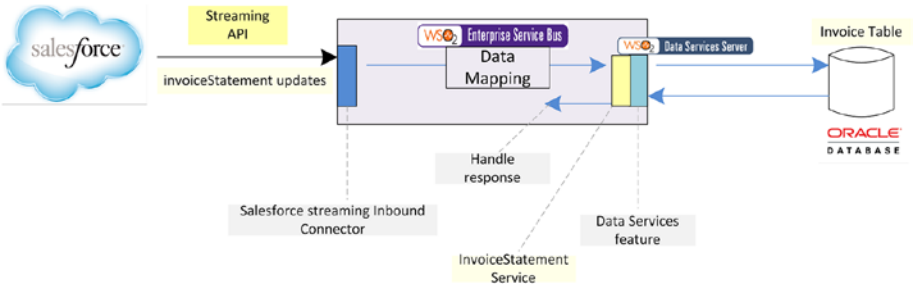

Figure 8-15. *Using the Salesforce streaming connector to receive Salesforce invoice statement updates and store them in a database using the data services feature*

So, you can use Salesforce streaming connector, which is an inbound endpoint. From that inbound connector, you can inject the messages to a sequence. Since you need to call the database through the data service feature, you need to install the data service features inside the ESB. Then you should create a data service with the operation required to insert the data into the database. Once you create the data service and deploy it, you can call the data service from the injecting sequence of the Salesforce stream connector. You will have to do the data mapping between the Salesforce message format and the message format required for the data service. The service call from the injecting sequence to the data service should go through the local transport.

Summary

In this chapter, you learned about the broad range of integration capabilities of WSO2 ESB. They are:

- SAP integration with IDoc and BAPI
- HL7 integration with HL7 v.2.x
- WebSockets integration
- What ESB connectors are and how to use connector operations inside a mediation flow
- Real-world use cases related to connectors and on-premise systems, such as SAP to Salesforce integration
- Using local transport for `in-jvm` service calls when you call ESB's locally hosted services from other services
- Exposing data residing in database as a SOAP or RESTful service using WSO2's data services feature
- Calling data services from a local transport

CHAPTER 9

Security in WSO2 ESB

In real-world integration scenarios, you may have to expose existing non-secured services as secured services to the consumers or you may have to invoke services that are secured in some form. When ESB is used as the messaging bus between consumers and services, it has to support a wide range of security patterns and standards.

In this chapter, we will mainly talk about the security patterns supported by WSO2 ESB with regard to the message and wire level security. This is a subset of enterprise security use cases that you may encounter in the real world. ESB's primary responsibility to address message- and transport-level security requirements while other security requirements such as identity federation, identity transformation, user provisioning, single-sign-on, etc., are part of identity and access management products. WSO2 offers the WSO2 Identity Server to cater to those requirements and it can seamlessly connect with WSO2 ESB as well.

Transport Level Security

When there's communication between client and server applications, most organizations want to do that in a secured manner. Rather than implementing a message-level security protocol, the entire communication channel can be secured.

Secure Sockets Layer (SSL) is used to secure communications over a network so that only the sender and receiver have access to the data that is transferred between them (to eliminate man-in-the-middle attacks and eavesdropping). Transport Layer Security (TLS) and its predecessor, Secure Sockets Layer (SSL), are both cryptographic protocols designed to provide communications security over a computer network.

Certificates and keys are used to enable SSL communication between the sender and the receiver. A certificate carries the information and a digital signature that identifies the client or server associated with that certificate. There are different types of keys such as private, public, and session, that you should use to enable secure communication between two entities.

Encryption uses a private key/public key pair, which ensures that the data can be encrypted by one key but can only be decrypted by the other key pair (Public-Key Infrastructure—PKI—Scheme). Trust is achieved through the use of certificates. Certificate trust can be thought of as a chain that starts with the Certificate Authority (or CA). A CA is a company or entity that issues SSL certificates. When SSL/TLS is being used for secure data exchange between client and server applications, there are two

K. Indrasiri, *Beginning WSO2 ESB*, DOI 10.1007/978-1-4842-2343-7_9

main mechanisms that are used—one-way and two-way SSL. Let's have quick look at each mechanism and then dive into WSO2 ESB capabilities for supporting SSL/TLS communication.

One-Way SSL (Server Authentication)

The high-level interaction of one-way SSL protocol is depicted in Figure 9-1. It starts with a client requesting access to a protected resource. Then the server presents its certificate to the client along with the server's public key. The client verifies the server's certificate and if the verification is successful, the client generates a session key and encrypts that key with the server's public key and sends it across. Since only the server can decrypt it, the session key can be used to encrypt all the message exchanges.

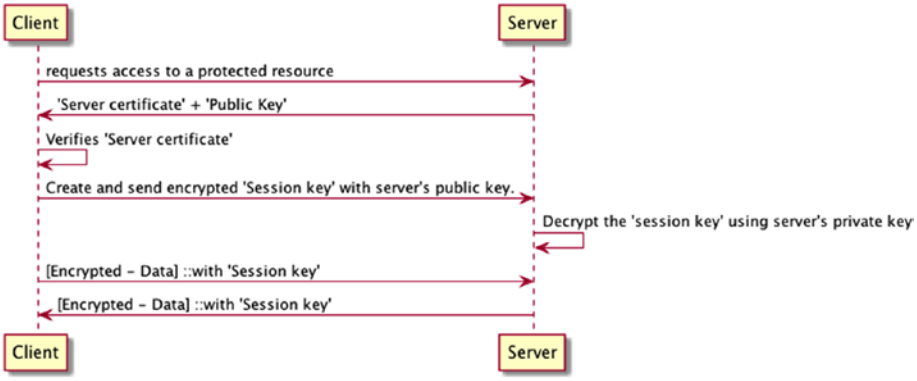

Figure 9-1. *One-way SSL*

Two-way SSL protocol is a slight extension of one-way SSL.

Two-Way SSL (Mutual/Client Authentication)

In two-way SSL, the client requests access to a protected resource. Then the server presents its certificate to the client. Once the client verifies the server's certificate, it sends its certificate to the server. The server verifies the client's credentials. Once this is done, the session key is exchanged between the client and server, and then data exchange starts with the use of the session key as the data encryption key. The high-level interactions of the two-way SSL protocol are illustrated in Figure 9-2.

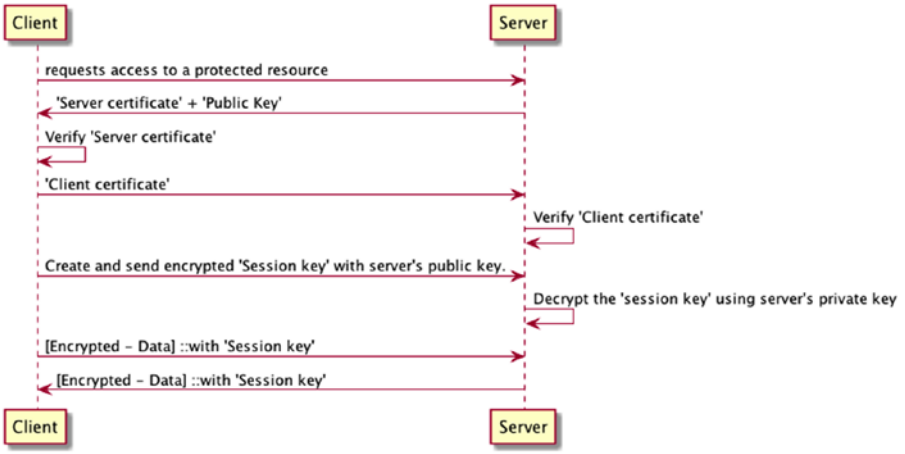

Figure 9-2. *Two-way SSL/mutual authentication*

Since now you have a basic understanding of how these SSL/TLS protocols work, let's move on to the SSL/TLS support offered by WSO2 ESB.

Using TLS/SSL with WSO2 ESB

In the context of WSO2 ESB, the majority of the TLS/SSL related use cases are there in the HTTP transport space. Hence, we will focus on the HTTP specific transport level security features only.

WSO2 ESB can be used to expose non-secured services as secured services using TLS/SSL enabled proxy service, the REST API, or inbound endpoints. The transport level security is configured at the transport receiver/listener side. In addition, HTTP inbound endpoints allow you to configure it at inbound endpoint configuration level so that you can use different transport configurations for each inbound endpoint.

Similarly, WSO2 ESB can invoke backend services, which are secured with TLS/SSL. In this case, the configuration has to be done at the transport sender side of the ESB.

Let's dive deep into the use cases related to TLS/SSL for both inbound and outbound messaging scenarios. Figure 9-3 illustrates the four main scenarios related to WSO2 ESB HTTP transport level security.

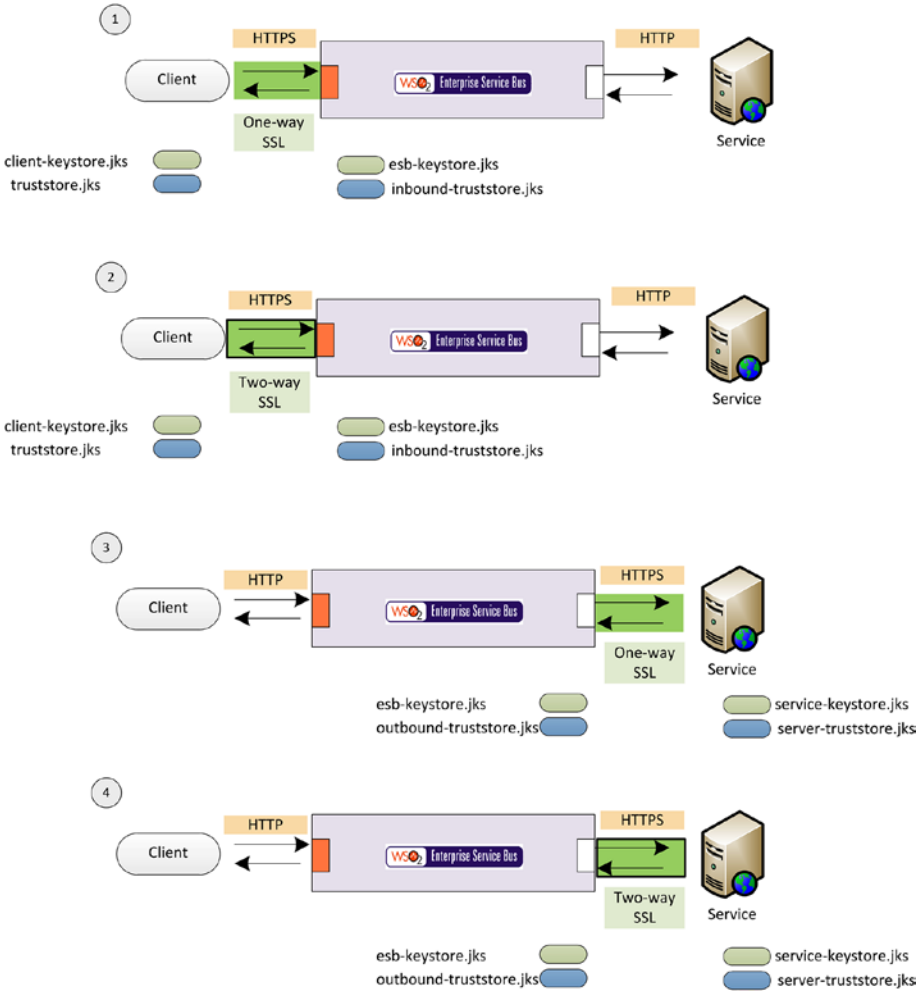

Figure 9-3. *Two-way SSL/mutual authentication*

Use Case 1: Client to ESB: One-Way SSL

In this use case, the HTTPs channel between the client and ESB is secured with one-way SSL. As shown in Figure 9-3, use case 1, suppose that ESB uses `esb-keystore.jks` as its keystore and `inbound-truststore.jks` is the trust store of ESB that will be used for incoming messaging channels. You can configure these keystores in your default transport configuration as follows.

You simply have to place keystores in the file system and refer them at the ESB transport receiver side configuration. Here, we have placed a keystore (`esb-keystore.jks`) and a trust store for inbound messages (`inbound-truststore.jks`) in the

repository/resources/security directory and referred it at the HTTPS transport receiver's configuration in axis2.xml.

Listing 9-1. One-Way SSL with HTTPs Transport

```
<transportReceiver name="https" class="org.apache.synapse.transport.
passthru.PassThroughHttpSSLListener">
        <parameter name="port" locked="false">8243</parameter>
        <parameter name="non-blocking" locked="false">true</parameter>
        <parameter name="HttpsProtocols">TLSv1,TLSv1.1,TLSv1.2</parameter>
        <parameter name="httpGetProcessor" locked="false">org.wso2.carbon.
        mediation.transport.handlers.PassThroughNHttpGetProcessor</
        parameter>
        <parameter name="keystore" locked="false">
            <KeyStore>
                <Location>repository/resources/security/esb-keystore.jks</
                Location>
                <Type>JKS</Type>
                <Password>wso2carbon</Password>
                <KeyPassword>wso2carbon</KeyPassword>
            </KeyStore>
        </parameter>
        <parameter name="truststore" locked="false">
            <TrustStore>
                <Location>repository/resources/security/inbound-truststore.
                jks</Location>
                <Type>JKS</Type>
                <Password>wso2carbon</Password>
            </TrustStore>
        </parameter>
        <!--<parameter name="SSLVerifyClient">require</parameter>
            supports optional|require or defaults to none -->
    </transportReceiver>
```

This is all you need to do at the ESB side to support one-way SSL. At the client side, you need to import the ESB's certificate to the client's trust store.

You can export ESB HTTPs receiver's certificate from the keystore using keytool as follows:

```
  >> keytool -export -keystore esb-keystore.jks -alias localhost -file wso2_
esb.crt
```

And import it to the client side trust store with:

```
  >> keytool -import -keystore truststore.jks -alias wso2carbon -file wso2_
esb.crt
```

Now you are all set for the secured one-way SSL communication between the client and the ESB. Make a note of the `SSLVerifyClient` parameter, which is disabled in a one-way SSL scenario. All these changes will be applied for all HTTPs inbound calls that use the default HTTP transport. However, if you want to configure a HTTP inbound endpoint, you have the freedom of using specific keystores for each inbound endpoints. The configuration of a sample HTTPs inbound endpoint is shown in Listing 9-2 and Figure 9-3.

Listing 9-2. One-Way SSL with HTTP Inbound Endpoints

```
<inboundEndpoint name="HttpListenerEP" protocol="https" suspend="false"
sequence="TestIn" onError="fault" >
    <p:parameters xmlns:p="http://ws.apache.org/ns/synapse">
        <p:parameter  name="inbound.http.port">8081</p:parameter>
        <p:parameter name="keystore">
            <KeyStore>
                <Location>repository/resources/security/wso2carbon.jks
                </Location>
                <Type>JKS</Type>
                <Password>wso2carbon</Password>
                <KeyPassword>wso2carbon</KeyPassword>
            </KeyStore>
        </p:parameter>
        <p:parameter name="truststore">
            <TrustStore>
                <Location>repository/resources/security/client-truststore.
                jks</Location>
                <Type>JKS</Type>
                <Password>wso2carbon</Password>
            </TrustStore>
        </p:parameter>
        <p:parameter name="HttpsProtocols">TLSv1,TLSv1.1,TLSv1.2
        </p:parameter>
        <p:parameter name="SSLProtocol">SSLV3</p:parameter>
        <p:parameter name="CertificateRevocationVerifier">
            <CertificateRevocationVerifier enable="true">
               <CacheSize>10</CacheSize>
               <CacheDelay>2</CacheDelay>
            </CertificateRevocationVerifier>
         </p:parameter>
     </p:parameters>
</inboundEndpoint>
```

Use Case 2: Client to ESB: Two-Way SSL

As shown in Figure 9-3, Use case 2, the communication between the client and ESB has to be secured with two-way SSL. As you learned in two-way SSL interactions that we explained earlier, the only difference in this scenario is that the client certificate has

to be verified by the server (ESB). Therefore, you need to enable the `SSLVerifyClient` parameter at the HTTP's transport receiver. Listing 9-3 shows the configuration with that parameter enabled at the HTTP's transport receiver.

Listing 9-3. Two-Way SSL with HTTP Transport Receiver

```
<transportReceiver name="http" class="org.apache.synapse.transport.passthru.
PassThroughHttpListener">
        <parameter name="port" locked="false">8280</parameter>
        <parameter name="non-blocking" locked="false">true</parameter>
        <!--parameter name="bind-address" locked="false">hostname or IP
        address</parameter-->
        <!--parameter name="WSDLEPRPrefix" locked="false">https://
        apachehost:port/somepath</parameter-->
        <parameter name="httpGetProcessor" locked="false">org.wso2.carbon.
        mediation.transport.handlers.PassThroughNHttpGetProcessor
        </parameter>
        <!--<parameter name="priorityConfigFile" locked="false">location of
        priority configuration file</parameter>-->
    </transportReceiver>

     <transportReceiver name="https" class="org.apache.synapse.transport.
     passthru.PassThroughHttpSSLListener">
        <parameter name="port" locked="false">8243</parameter>
        <parameter name="non-blocking" locked="false">true</parameter>
        <parameter name="HttpsProtocols">TLSv1,TLSv1.1,TLSv1.2</parameter>
        <!--parameter name="bind-address" locked="false">hostname or IP
        address</parameter-->
        <!--parameter name="WSDLEPRPrefix" locked="false">https://
        apachehost:port/somepath</parameter-->
        <parameter name="httpGetProcessor" locked="false">org.wso2.carbon.
        mediation.transport.handlers.PassThroughNHttpGetProcessor
        </parameter>
        <parameter name="keystore" locked="false">
            <KeyStore>
                <Location>repository/resources/security/esb-keystore.jks
                </Location>
                <Type>JKS</Type>
                <Password>wso2carbon</Password>
                <KeyPassword>wso2carbon</KeyPassword>
            </KeyStore>
        </parameter>
        <parameter name="truststore" locked="false">
            <TrustStore>
                <Location>repository/resources/security/inbound-truststore.
                jks</Location>
                <Type>JKS</Type>
                <Password>wso2carbon</Password>
```

```
            </TrustStore>
        </parameter>
         <parameter name="SSLVerifyClient">require</parameter>
        <!--<parameter name="SSLVerifyClient">require</parameter>
            supports optional|require or defaults to none -->
    </transportReceiver>
```

For HTTP inbound endpoints, you can enable the same parameter.

In addition to the keys that you have import/export in one-way SSL scenario, you need to export the client certificate from `client-keystore.jks` and import it to `inbound-truststore.jks` of ESB. This is the certificate that's required for ESB to authenticate the client.

Use Case 3: ESB to Service: One-Way SSL

This is the scenario that ESB has to use to invoke a backend service, which is secured with one-way SSL. We can use the same keystore—`esb-keystore.jks` for the ESB—and as the trust store, we can use `outbound-truststore.jks`. The outbound HTTP messaging channel can be configured at the HTTP transport sender side of ESB as follows.

Listing 9-4. One-Way SSL with HTTP Transport Sender

```
<transportSender name="https" class="org.apache.synapse.transport.passthru.
PassThroughHttpSSLSender">
        <parameter name="non-blocking" locked="false">true</parameter>
        <parameter name="keystore" locked="false">
            <KeyStore>
                <Location>repository/resources/security/esb-keystore.jks
                </Location>
                <Type>JKS</Type>
                <Password>wso2carbon</Password>
                <KeyPassword>wso2carbon</KeyPassword>
            </KeyStore>
        </parameter>
        <parameter name="truststore" locked="false">
            <TrustStore>
                <Location>repository/resources/security/outbound-truststore.
                jks</Location>
                <Type>JKS</Type>
                <Password>wso2carbon</Password>
            </TrustStore>
        </parameter>
        <!--<parameter name="HostnameVerifier">DefaultAndLocalhost
        </parameter>-->
            <!--supports Strict|AllowAll|DefaultAndLocalhost or the default
            if none specified -->
    </transportSender>
```

The certificate of the service has to be imported to the trust store of ESB (`outbound-truststore.jks`). Once you have done these configurations, you can invoke a backend through HTTP transport with one-way SSL.

Use Case 4: ESB to Service: Two-Way SSL

This is quite similar to use case 3, with the only difference of using ESB certificate at the service side. You need to export the ESB certificate from `esb-keystore.jks` and import the certificate at the service's trust store.

SSL Profiles

When you invoke backend services with one-way or mutual SSL, you may have to use different keystores/trust stores for different backend services. As an example, suppose that you have partner services and a customer service that uses TLS/SSL, and you want to maintain different keystores/trust stores for partners and customers. This can be done by specifying SSL profiles at the HTTPs transport sender in `axis2.xml` by specifying a `customSSLProfiles` parameter with the configuration shown in Listing 9-5.

Listing 9-5. SSL Profiles at the Transport Sender

```
<transportSender name="https" class="org.apache.synapse.transport.passthru.
PassThroughHttpSSLSender">
        <parameter name="non-blocking" locked="false">true</parameter>
        <parameter name="keystore" locked="false">
            <KeyStore> ...</KeyStore>
        </parameter>
        <parameter name="truststore" locked="false">
            <TrustStore>...</TrustStore>
        </parameter>

        <parameter name="customSSLProfiles">

            <profile>
                <servers>partner1.service.com:80, partner2.service.com:80,
                localhost:9444</servers>
                <KeyStore>
                    <Location>repository/resources/security/esb-keystore.
                    jks</Location>
                    <Type>JKS</Type>
                    <Password>password</Password>
                    <KeyPassword>password</KeyPassword>
                </KeyStore>
                <TrustStore>
                    <Location>repository/resources/security/partner-
                    truststore.jks</Location>
                    <Type>JKS</Type>
```

```
                <Password>password</Password>
            </TrustStore>
        </profile>

        <profile>
            <servers>customer1.service.com:80, customer2.service.
            com:80</servers>
            <KeyStore>
                <Location>repository/resources/security/esb-keystore.
                jks</Location>
                <Type>JKS</Type>
                <Password>password</Password>
                <KeyPassword>password</KeyPassword>
            </KeyStore>
            <TrustStore>
                <Location>repository/resources/security/customer-
                truststore.jks</Location>
                <Type>JKS</Type>
                <Password>password</Password>
            </TrustStore>
        </profile>
    </parameter>

</transportSender>
```

In this scenario, we used two different trust stores for the partner endpoint and the customer endpoint.

Application Level Security

Transport level security secures the entire communication channel between clients, ESB, and services. However, there can be situations where you want to have much more fine-grained security protocols implemented at each interface that ESB exposes or invokes. For example, a given REST API hosted in ESB can be secured with `Basic-Authentication` while proxy services are secured with `WS-Security`. Similarly, ESB may have to invoke services secured with OAuth as well as SOAP web services that leverage `WS-Security`/`WS-Security` Policy. These different requirements are addressed as part of the application level security use cases of WSO2 ESB.

Securing REST APIs

Let's start our discussion on how you can secure a REST API/HTTP service that you create in WSO2 ESB. You can use different types of API security techniques to do so.

The "basic" authentication scheme is based on the model that the client must authenticate itself with a username and a password for each REST API. Basic authentication transmits usernames and passwords across the network in

an unencrypted form, but you can use it on top of TLS/SSL to make a secured communication. The server will service the request only if it can validate the user ID and password for the protection space of the Request URI.

We can use `Basic-Auth` with WSO2 ESB REST APIs by incorporating a handler into the REST API configuration.

Listing 9-6. Securing REST API with a Basic-Auth Handler

```
<api name="StockQuoteAPI" context="/stockquote">
    <resource methods="GET" uri-template="/view/{symbol}" protocol="https" >
     ...
    </resource>
    <handlers>
      <handler class="org.wso2.rest.BasicAuthHandler"/>
    </handlers>
  </api>
```

With the API handler, you can either use the existing handler or extend it with your own requirement to handle different user stores, etc. Similarly, if you want to implement an OAuth 2.0 access token validation, it can be implemented as an API handler.

■ **Note** Although you can use API security techniques at the ESB layer, It is important to note that, in most cases, these APIs are exposed as managed APIs by API-Management solutions. So, security, throttling, caching, etc. are applied at that layer. Most WSO2 users implement non-secured services at the ESB layer and expose them as managed APIs through the WSO2 API Manager. Refer to https://docs.wso2.com/display/AM200/Getting+Started for more details.

Securing Proxy Services

The proxy services that you implement at the ESB layer can be secured using SOAP security techniques. Let's suppose that you want to implement a proxy service that uses `WS-Security` signing and encryption with `WS-Policy`. So, the first thing you have to do is figure out the required security policy that you want to apply at the proxy services level. There are various security policies that you can use and writing a security policy from scratch is a tedious task. Therefore, the WSO2 ESB development tool (details of the WSO2 development tool are discussed in Chapter 10) provides a wizard to configure the security scenario of your proxy service.

The WSO2 ESB developer tool provides 20 predefined, commonly used security scenarios to choose from. To create a security policy, create a Registry resource artifact (MySecurityPolicy-Resource type should be `WS-Policy`). Select design view for the `MySecurityPolicy.xml` file. You can see all the predefined security scenarios listed in the security wizard shown in Figure 9-4. In this example, I used the `UserNameToken` scenario. Under this scenario, we need to select user roles that are allowed to access this particular proxy service, so that users who have those roles can only access the service.

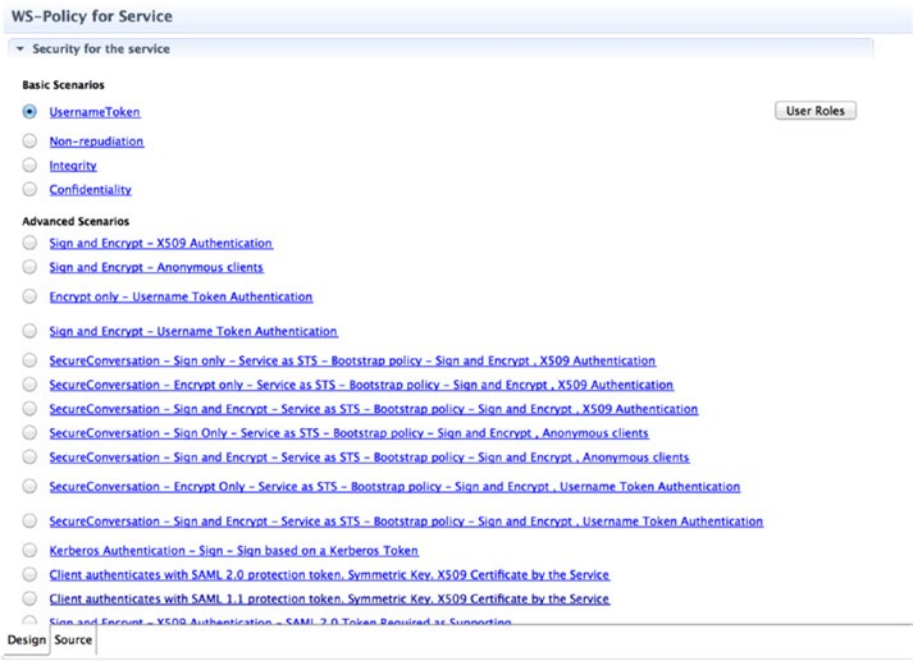

Figure 9-4. *Web service security wizard of the WSO2 ESB tool*

Once you create the policy file, you can create a proxy service and enable security of that proxy service. Listing 9-7 shows a proxy configuration with security enabled from the security policy that you created. Refer to https://docs.wso2.com/display/ESB500/Applying+Security+to+a+Proxy+Service for more details on how to use the security wizard. Because this example uses Apache Rampart for the backend security implementation, you might also need to download and install the unlimited strength policy files for your JDK before using Apache Rampart. Refer to http://docs.wso2.com/enterprise-service-bus/Securing+APIs for details.

Listing 9-7. Secured Proxy Service

```
<proxy name="SecuredStockQuoteService">
    <target>
        <inSequence>
            <header name="wsse:Security" action="remove"
                    xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/
                    oasis-200401-wss-wssecurity-secext-1.0.xsd"/>
            <call>
                <endpoint>
                    <address uri="http://localhost:9000/services/
SimpleStockQuoteService"/>
                </endpoint>
            </call>
```

```
            <respond/>
        </inSequence>
        <outSequence>
        </outSequence>
    </target>
    <publishWSDL uri="gov:/SecuredStockQuoteService.wsdl"/>
    <policy key="gov:/MySecurityPolicy"/>
    <enableSec/>
</proxy>
```

In this configuration, you can see that a reference to the policy resource is given in the proxy service configuration and, before sending the request to the non-secured service, we have removed unwanted security headers using the `header` mediator.

Invoking Secured Service

From the ESB mediation layer you may have to invoke the services, which are secured with application level security. Backend service/REST APIs may be secured from basic authentication, access tokens with OAuth, or using `WS-Security` standards.

Invoking Basic-Auth and OAuth 2.0-Based Services

Suppose that you want to invoke a service, which is secured with basic authentication. You can simply invoke the service through HTTP and generate the required "Authorization" header as a transport scope header. You need to use the `base64Encode` function to encode the username and password.

Listing 9-8. Invoking a Backend Service that's Secured with Basic-Auth, from ESB

```
<proxy name="OrderProcessingService">
    <target>
        <inSequence>
            ...
            <header name="Authorization"  expression="fn:concat('Basic ',
            base64Encode('my_username:my_password'))" scope="transport"/>
            <call>
                <endpoint>
                    <address uri="https://localhost:9888/services/
                    LocationService"/>
                </endpoint>
            </call>
            ...
            <respond/>
        </inSequence>
```

```
        <outSequence>
        </outSequence>
    </target>
    <publishWSDL uri="gov:/OrderProcessingService.wsdl"/>
</proxy>
```

Similarly, you access a resource that's secured with OAuth 2.0, and you using the required access token (with the prefix `Authorization: Bearer`).

Invoking WS-Security-Based Services

One of the common ways to secure SOAP-based web services is to use `WS-Security`/`WS-Security` Policy. Any service that leverages the SOAP security standard can be invoked from WSO2 ESB by specifying the security policy at the endpoint level. Listing 9-9 shows the configuration of this scenario. At the endpoint configuration, we have set up security policy resources with the `enableSec` parameter. Addressing is also enabled for outbound messaging.

Listing 9-9. Enabling Security for Outgoing Messages to Invoke a Secured Backend Service

```
<api name="StockQuoteAPI" context="/stockquote">
    <resource methods="POST" uri-template="/view/{symbol}" protocol="http" >
      <inSequence>
         ...
         <call>
            <endpoint>
               <address uri="http://localhost:6060/services/
               PizzaShopService" format="soap11">
                  <enableSec policy="gov:/PizzaShopServiceSecPolicy"/>
                  <enableAddressing/>
               </address>
            </endpoint>
         </call>

         <header name="wsse:Security" action="remove"
                    xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/
                    oasis-200401-wss-wssecurity-secext-1.0.xsd"/>
         <respond/>
      </inSequence>
    </resource>
</api>
```

In the response path, before sending back the response, we removed the security headers, as they are not required on the client side.

OAuth Mediator: Access Token Validation Inside a Mediation Flow

WSO2 ESB provides an OAuth mediator that can be used to validate the access tokens inside a mediation flow. Basically you can use OAuth mediation inside a sequence and the incoming access tokens are validated against an external OAuth service. In Listing 9-10, we used the OAuth mediator that calls the OAuth service available on the remoteServiceUrl and the required credentials to access the service are given as a parameter of the mediator. If the access token is validated successfully, the message flow will continue and, if that's not successful, then a fault message will be sent back to the client of that particular API.

Listing 9-10. OAuth Mediator for Access Token Validation

```
<api name="StockQuoteAPI" context="/stockquote">
    <resource methods="POST" uri-template="/view/{symbol}" protocol="https" >
      <inSequence>
         ...
         <oauthService xmlns="http://ws.apache.org/ns/synapse"
         remoteServiceUrl="https://localhost:9443/services" username="admin"
         password="admin" />
         <call>
             <endpoint>
                 <address uri="https://localhost:9888/services/
                 LocationService"/>
             </endpoint>
         </call>
         <respond/>
      </inSequence>
    </resource>
</api>
```

Entitlement Mediator: Policy Enforcement Inside a Mediation Flow

The entitlement mediator intercepts requests and evaluates the actions performed by the user against an eXtensible Access Control Markup Language (XACML) policy. You need to use an XACML Policy Decision Point (PDP), such as WSO2 Identity Server. The ESB message flow can act as the XACML Policy Enforcement Point (PEP) where the policy is enforced. Based on the policy enforcement, there can be three types of actions that you can take. Those are configured as part of the mediator configuration. In the configuration shown in Listing 9-11, we have configured the entitlement mediator inside a proxy service.

Listing 9-11. Policy Enforcement with the Entitlement Mediator

```
<proxy xmlns="http://ws.apache.org/ns/synapse"
       name="BusinessPolicyValidator"
       transports="https"
       startOnLoad="true"
       trace="disable">
   <description/>
   <target>
      <inSequence>
         <entitlementService remoteServiceUrl="https://localhost:9443/
         services"
                             remoteServiceUserName="admin"
                             remoteServicePassword=
"enc:kuv2MubUUveMyv6GeHrXr9il59ajJIqUI4eoYHcgGKf/BBFOWn96NTjJQI+wYbWjKW6r79S
7L7ZzgYeWx7DlGbff5X3pBN2Gh9yVOBHP1E93QtFqR7uTWi141Tr7V7ZwScwNqJbiNoV+vyLbsqK
JE7T3nP8Ih9Y6omygbcLcHzg="
                             callbackClass="org.wso2.carbon.
                             identity.entitlement.mediator.callback.
                             UTEntitlementCallbackHandler"
                             client="basicAuth">
            <onReject>
               <makefault version="soap12">
                  <code xmlns:soap12Env="http://www.w3.org/2003/05/soap-
                  envelope"
                        value="soap12Env:Receiver"/>
                  <reason value="UNAUTHORIZED"/>
                  <node/>
                  <role/>
                  <detail>XACML Authorization Failed</detail>
               </makefault>
                        <respond/>
            </onReject>
            <onAccept>
               <call>
                  <endpoint>
                     <address uri="http://localhost:8281/services/
                     BusinessService"/>
                  </endpoint>
               </call>
                <respond/>
            </onAccept>
            <obligations/>
            <advice/>
         </entitlementService>
      </inSequence>
```

```
   </target>
   <publishWSDL uri="http://localhost:8281/services/BusinessPolicyValidator
?wsdl"/>
</proxy>
```

The details of the PDP (WSO2 IS) are provided as the mediator configuration and we configured mediation logic under each entitlement result.

- *OnAccept*: The sequence to execute when the entitlement mediator returns a result of `Permit`. In the previous example, we invoke the backend service upon a policy permit.
- *OnReject*: The sequence to execute when the entitlement mediator returns a result of `Deny`, `Not Applicable`, or `Indeterminate`. We created a SOAP fault for the policy reject condition in the previous scenario.
- `Obligation` and `Advice`: When the XACML response contains an `Obligation` or `Advice` statement, the entitlement mediator clones the current message context, creates a new message context, sets the `Obligation` or `Advice` statement to the SOAP body, and then executes the sequence. It executes the `Obligation`'s sequence synchronously and therefore waits for its response. If the sequence returns `true`, the `OnAccept` sequence is executed; if `false`, the `OnReject` sequence is executed. The `Advice` sequence is executed asynchronously, so the mediator does not wait for a response.

The entitlement callback handler is used as a handler to get the subject (username) of the XACML request. The method of communication between the PEP and the PDP is specified by the entitlement service client parameter.

Summary

This chapter covered how to integrate WSO2 ESB with various security protocols.

- Transport level security is one of the most commonly used security techniques in WSO2 ESB.
- Various capabilities of handling SSL scenarios, such as one-way SSL, mutual SSL, and outbound SSL profiles, were discussed.
- Application level security features mainly fall under API security and web services security of hosted proxy service.
- REST APIs of ESB provide an extensible way to plug your API into handlers that process the security, such as `Basic-Auth`.

- Often API security is achieved at the API management layer and products such as WSO2 API Manager can support a wide range of API security techniques out of the box.
- SOAP web service security can be implemented by incorporating the `WS-Security` policy at the proxy service layer for inbound calls and at the endpoint layer for outbound calls.
- From mediation flows, you can use the `OAuth` mediator to validate access tokens and the entitlement mediator to enforce a policy.

CHAPTER 10

Development and Deployment Methodology

Throughout the entire set of chapters that you have read so far, we have covered most of the fundamentals of the ESB construct along with use cases. However, we have only focused on micro-level ESB mediation logic for those specific scenarios. Now it's time to learn the development and deployment process associated with WSO2 ESB and how you can leverage it and scale it to a large-scale integration project. So, during the first part of the chapter, we will focus on the development process and later we'll go into the details of how WSO2 ESB is deployed in real-world use cases to support high-availability and failover.

Development Methodology

The development methodology of WSO2 ESB is pretty much about how you develop integration scenarios with WSO2 ESB, maintain them, moving the configuration across different environments, and scale them to a large-scale development process, along with the associated tools that help you optimize the development.

WSO2 ESB comes with three main components—the runtime, the development tool, and analytics component. You have learned most of the things related to the ESB runtime. This chapter and the next are about the ESB tool and analytics runtime.

Using the WSO2 ESB Development Tool

WSO2 ESB comes with its Eclipse-based development tool, which plays a vital role when it comes to the development process of WSO2 ESB integration scenarios. Similar to the ESB product download, you can download the respective tooling distribution and install it in your development environment at `http://docs.wso2.com/enterprise-service-bus/Installing+WSO2+ESB+Tooling`.

You can use WSO2 ESB development tool to graphically design your integration flows, do data mapping between message formats, and debug your integration flows. Let's begin the discussion with how you can create an ESB artifact (proxy service, REST API, sequence, etc.) with the ESB developer tool.

K. Indrasiri, *Beginning WSO2 ESB*, DOI 10.1007/978-1-4842-2343-7_10

In the ESB developer tool dashboard (see Figure 10-1), you can find various integration project types. In this chapter, we cover the most commonly used development method. The most common project type is the "ESB Solution Project."

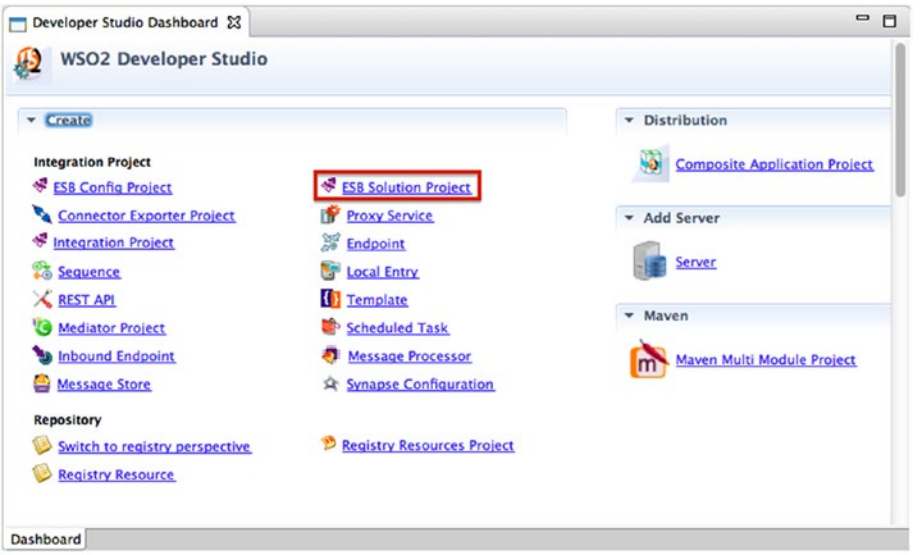

Figure 10-1. *ESB developer tool dashboard*

Suppose that you are building an integration scenario named `MyESBSolutionPrj` and once you create the project, it will have the structure shown in Figure 10-2.

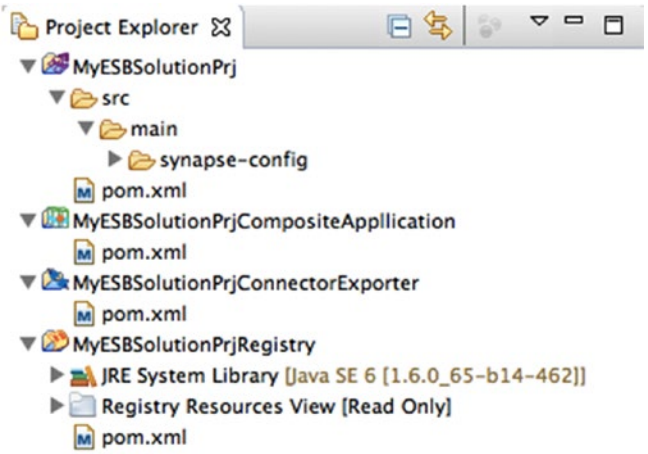

Figure 10-2. *ESB solution project structure*

As you can observe in Figure 10-2, there are several types of projects created when you create the ESB solution project.

The first one at the top is the ESB configuration project (`MyESBSolutionPrj`). This is where you can create and configure all ESB artifact-related configuration. As you can see, the main content of this project is a `synapse-config` directory where all the ESB constructs that you create will be placed.

Then you have the composite application project, which is capable of creating a deployable archive known as a composite application or `.car` file. For instance, when you want to deploy the ESB configuration that you create here, you need to export it as a .car archive using the composite application project.

The connector exporter is mainly used when you use connectors in your configuration. Often you fetch connectors from the WSO2 Connectors store and create ESB integration use cases using those connectors. So, when you deploy those integration scenarios, the associated connectors should also be part of the deployable archive. That's the purpose of using the "Connector Exporter" project. Finally, you have the registry resource project, which you can use to upload WSDL, XSLTs, policy files, etc. into the ESB registry and use those resources inside the ESB configuration project. The development process can be summarized as follows with the key steps involved in ESB development process.

- Create an ESB solution project. This will create four main projects in your developer tool.
- Build your integration scenario using ESB constructs (proxy services, APIs, sequences, etc.) inside the ESB config project.
- If you want to use any registry resources, create them inside the Registry resource project.
- If any connectors are to be used in the integration flow, they have to be imported into the tool and, inside the connector exporter, the respective connector should be included as part of the archive.
- When you are done with the development of the integration scenario, you simply have to export it as a composite application and the deployable archive will be a `.car` file.
- When you copy the created `*.car` to `repository/deployment/server/carbonapps` then ESB will dynamically deploy all the artifacts in that composite application.

Now let's go into the details of each project and find out how you can create each project configuration.

ESB Config Project

This is where you can create your ESB configuration logic. For that, you can use the graphical drag-and-drop tool or the source configuration. Figure 10-3 shows a screenshot of the graphical editor. You can select the required ESB constructs (mediators and endpoints) from the palette on the left.

Figure 10-3. *Graphical message flow editor*

You can click on each graphical elements and do the respective configuration in the properties window at the bottom.

ESB Registry Resource Project

You can save resources such as endpoints, data mapping configs, WSDLs, and XSLTs in a central repository as registry resources. All these registry resources need to be saved in a separate project called a "Registry Resources" project (see Figure 10-4).

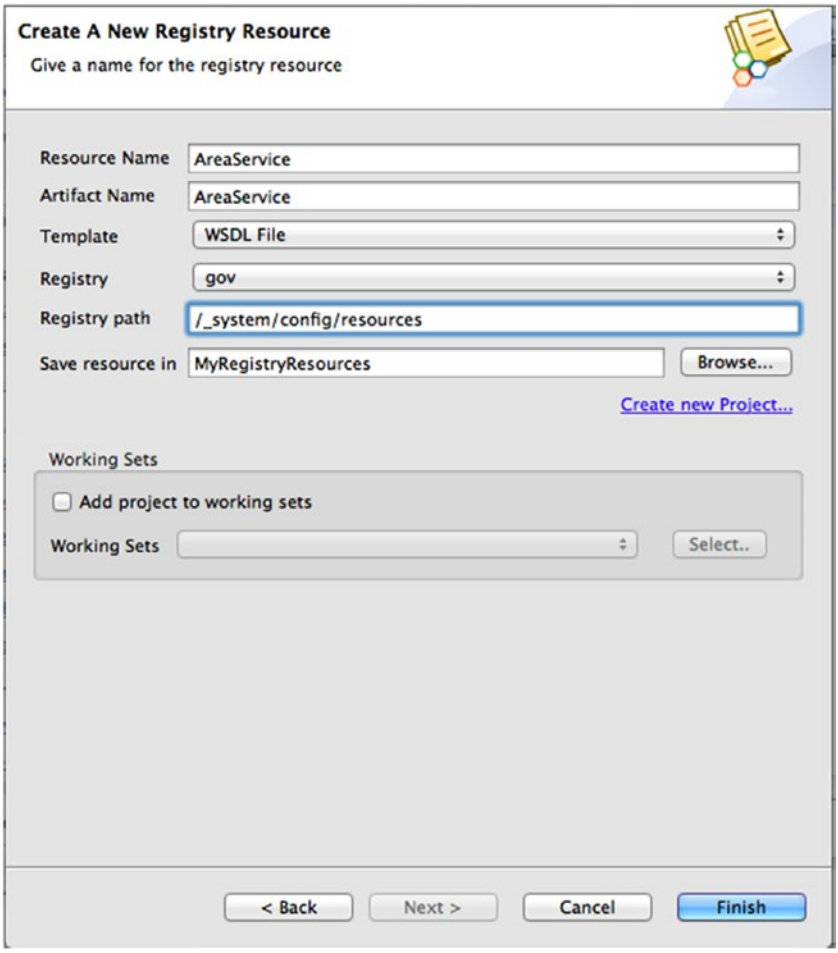

Figure 10-4. *Creating a registry resource inside inside the Registry Resource project*

The artifacts you create in the registry resource can be referred inside your ESB configuration that you create in your ESB configuration project. For example, the WSDL you create in the registry resource project can be referred from your proxy service configuration.

Importing Connectors to ESB Tool

In Chapter 8, you learned that ESB connectors are completely independent from the ESB release. Therefore, by default connectors are not shipped as part of the developer tool. What you have to do is import the connectors from the WSO2 Connector store into your developer tool. As shown in Figure 10-5, you can add or remove connectors inside your ESB Configuration project.

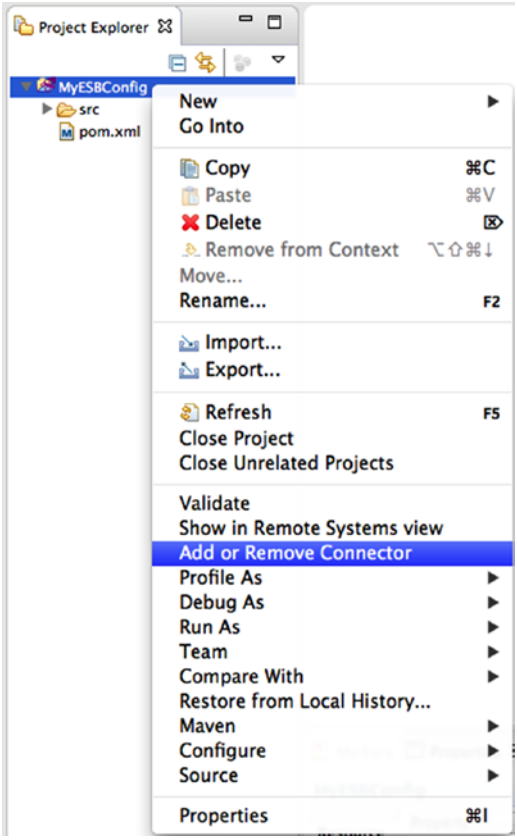

Figure 10-5. *You can add or remove connectors*

The connectors that you want to import can be selected, as shown in Figure 10-6.

Figure 10-6. *Selecting connectors to be imported into the developer tool*

Once you import the connectors, you can drag and drop connector operations into your ESB message flow. For example, as illustrated in Figure 10-7, you can see the operations related to the `Twilio` connector in the left pane. You just need to drag and drop the required operation into your sequence flow.

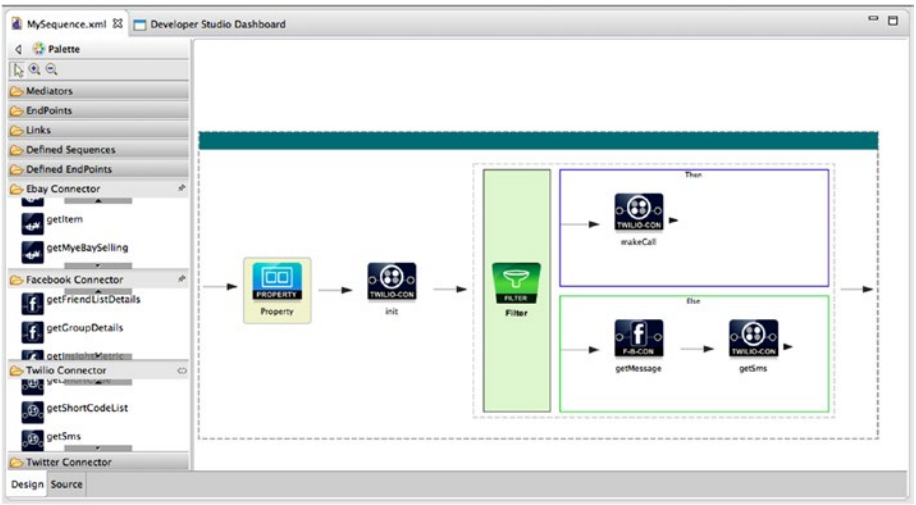

Figure 10-7. *You can drag and drop connector operations*

However, this only pulls the connectors into your developer tool but not into the ESB runtime, which you will finally deploy into. So, if you want to deploy the connector along with the ESB configuration, you need to add the imported connector into the "Connector Exporter" project. The "Connector Exporter" project is created when you create an ESB solution project (see Figure 10-8).

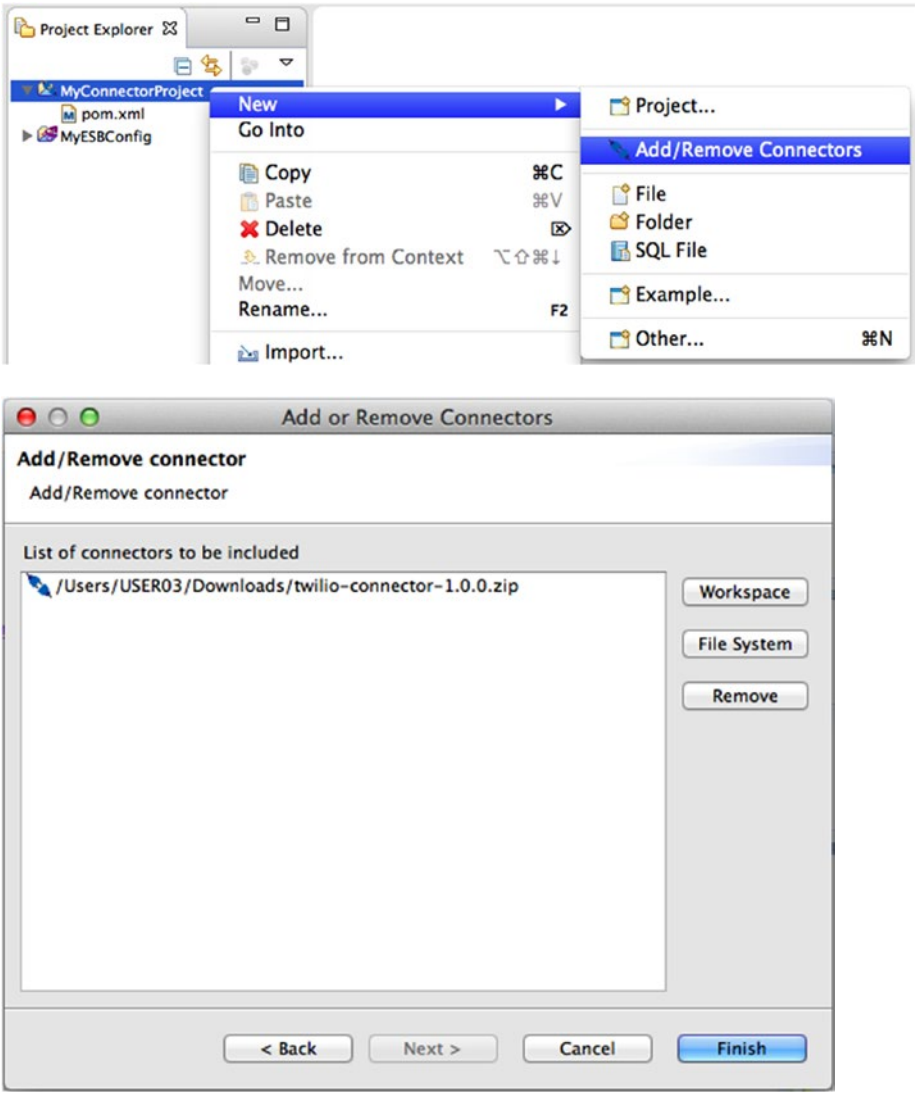

Figure 10-8. *Adding connectors to the Connector Exporter project*

So far we have discussed the development of various artifacts related to ESB, but not about how to deploy those into the runtime. That's where the Composite Application project comes into the picture.

Composite Application Project

Once you are done with the ESB integration project, you can go to the Composite Application project that is created as part of the ESB solution project and and then select the artifacts that you want to include in this particular deployable archive (see Figure 10-9).

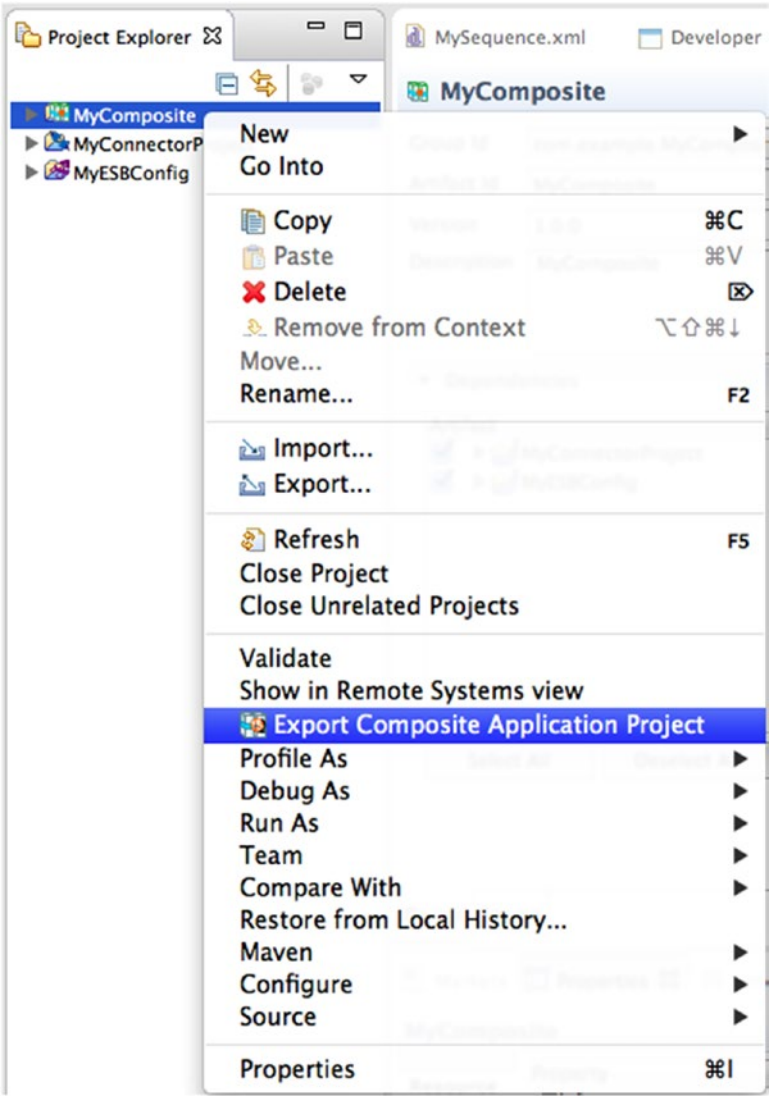

Figure 10-9. *Selecting the artifacts to be included in the .car file*

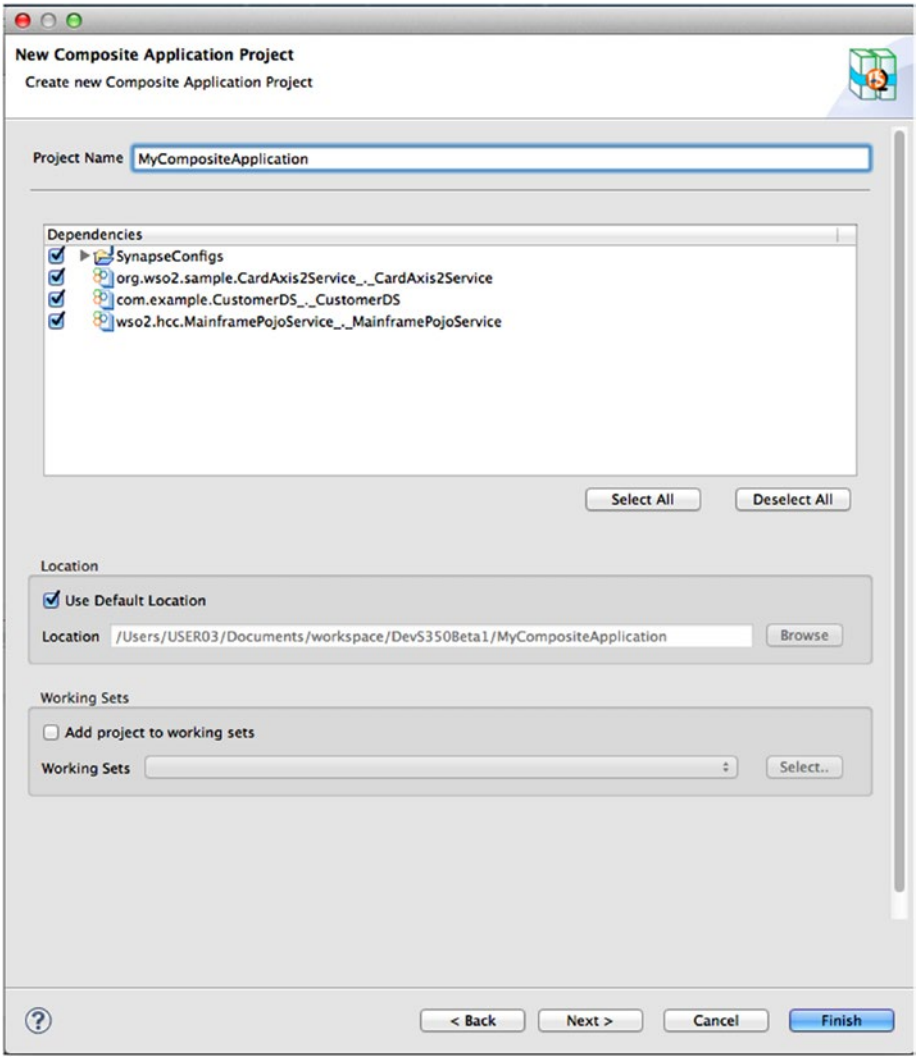

Figure 10-9. (*continued*)

With this you can export the `.car` file into your file system. You can also deploy it into your ESB runtime by just copying it over to `repository/deployment/server/carbonapps`.

Data Mapper

We briefly discussed data mapper in the initial chapters of this book. But let's dive deep into how you can use visual data mapping with WSO2 ESB. WSO2 ESB provides a data mapper (as a mediator) that you can use inside the ESB graphical editor, which you learned about in previous sections.

To understand the concept of the data mapper, suppose that you want to transform a message that's based on the JSON to SOAP message format. Suppose you have the sample message format of each message. So, you need to do this with the use of data mapper mediator in WSO2 ESB. Figure 10-10 illustrates the structure of the data mapper mediator.

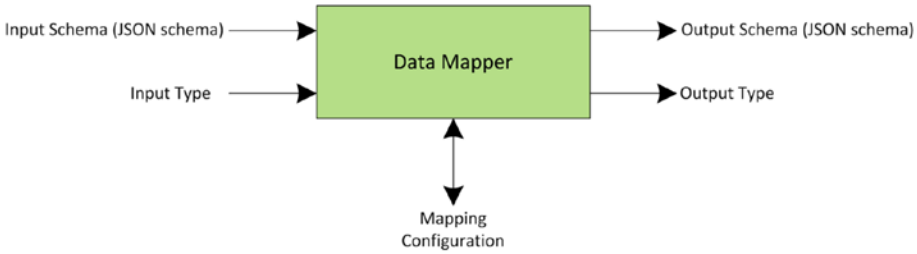

Figure 10-10. *Structure of the data mapper*

In order to transform JSON to XML, you need to provide an input schema (using JSON schema) of the incoming JSON message and the input type as JSON. As the output type, you need to provide the schema of the outgoing SOAP message (using JSON schema) and output type as XML. So once this data is available to the data mapper mediator, it can do the mapping between these two different message formats. The logic that you use to do the mapping is configured inside the mapping configuration. For example, if you want to do some string concatenation, that logic will be placed inside the mapping config.

So, those are the key concepts related to the data mapper, but that doesn't really help with learning how to use the data mapper. The following figures guide you on how you can graphically configure the data mapping for this scenario.

So the first thing you have to do here is drag and drop the data mapper into your sequence, as shown in Figure 10-11.

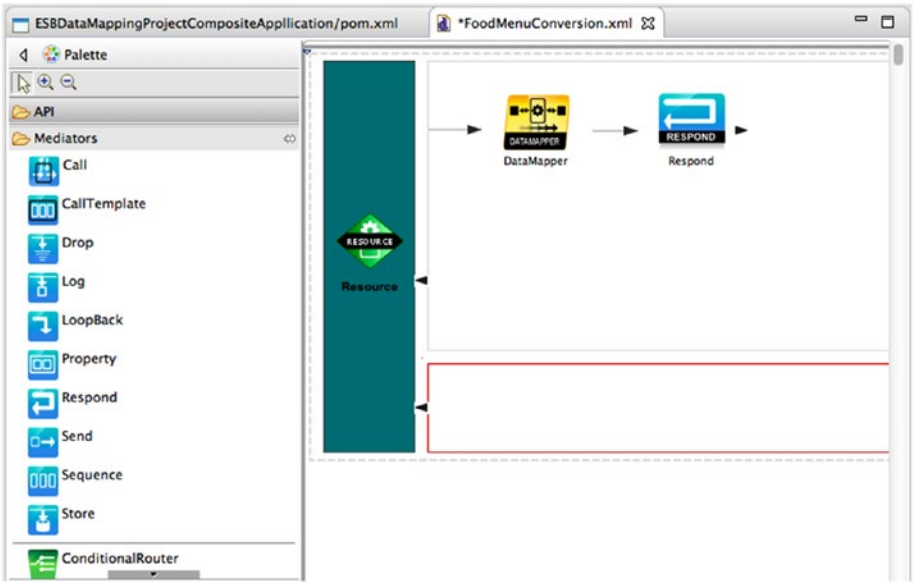

Figure 10-11. *Need caption*

Then when you double-click on the data mapper mediator, it prompts you to create the empty mapping configuration, which has to be created inside the registry resource project.

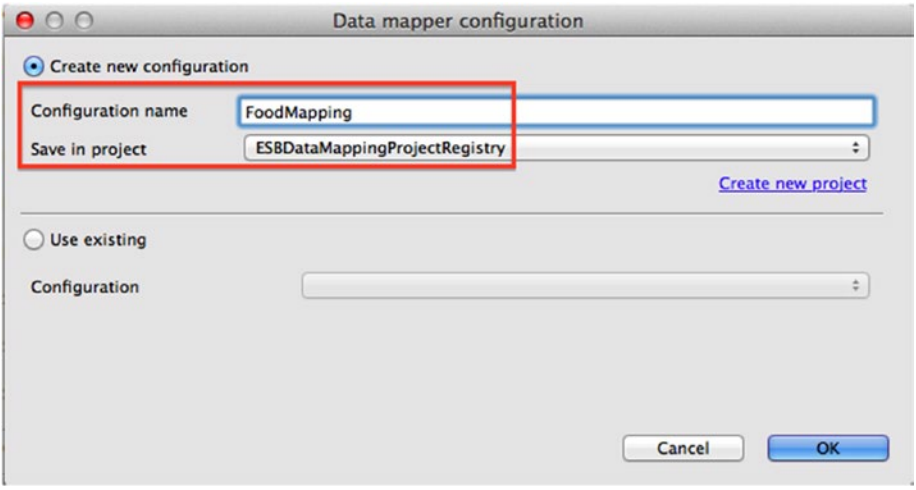

Figure 10-12. *Need caption*

Once you create the initial mapping config, you can observe an input box and an output box, which represents the incoming and outgoing messages from the data-mapping mediator.

Figure 10-13. *Need caption*

Now you can load the input/output schema automatically by loading the input/output sample messages.

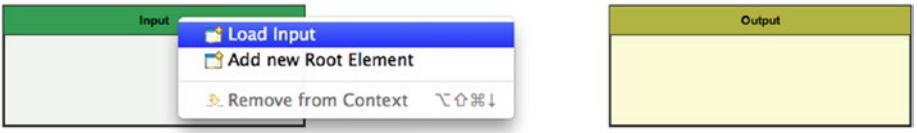

Figure 10-14. *Need caption*

The incoming/outgoing message structure will be graphically loaded into the data mapper, as shown in Figure 10-15.

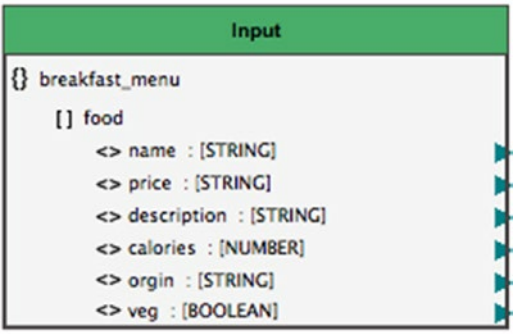

Figure 10-15. *Need caption*

Then you can simply drag and create the connectivity between the input and output messages.

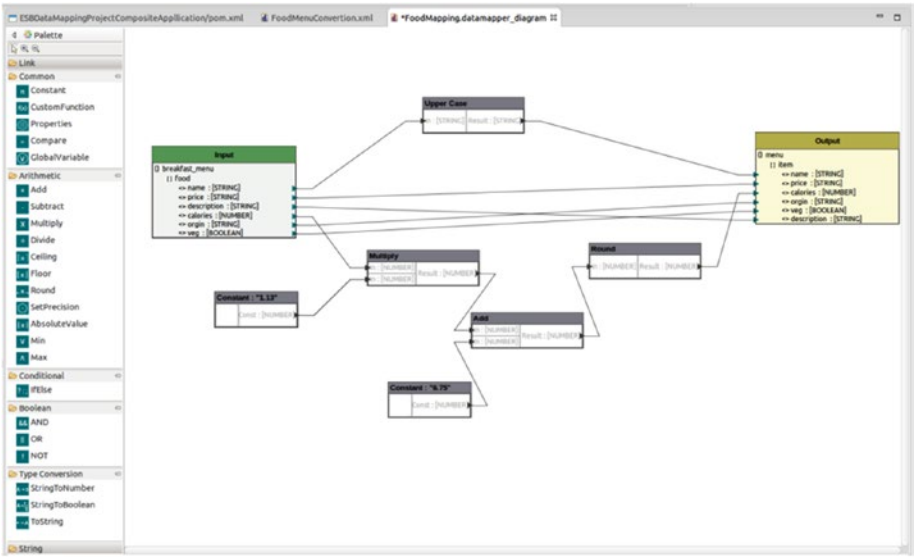

Figure 10-16. *Need caption*

There are quite a few data mapping operations that you could apply when you do the mapping between these formats and they are available in the left pane. Once you are done with the data mapping, the entire configuration can be deployed along with the other ESB artifacts.

Mediation Debugger

Debugging the mediation flows that you developed with WSO2 ESB is a key part of the development process. WSO2 ESB runtime facilitates the remote debugging and the debugging agent runs on the ESB development tool.

First you need to configure the ESB developer tool to work as the ESB debugger.

In the Java EE perspective of WSO2 ESB developer tool, click Run in the top menu of the WSO2 ESB Tooling Plugin, and then click Debug Configurations. Double-click ESB Mediation Debugger, as shown in Figure 10-17.

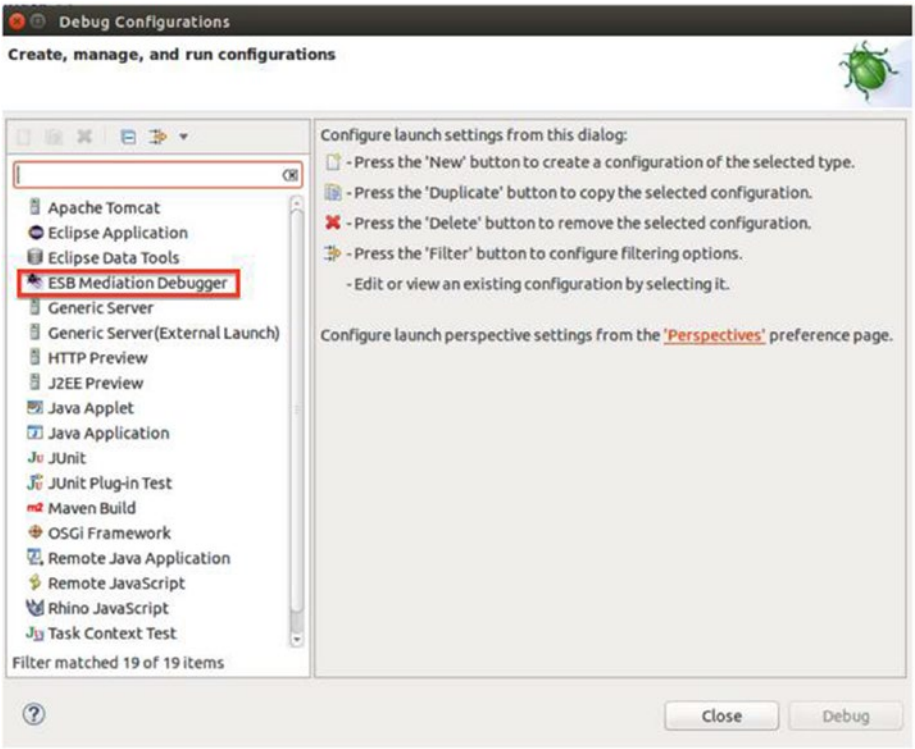

Figure 10-17. *Need caption*

Then you can configure the hostname of the ESB server and the debugging ports that are exposed by the ESB (when it is stared in the debugger mode).

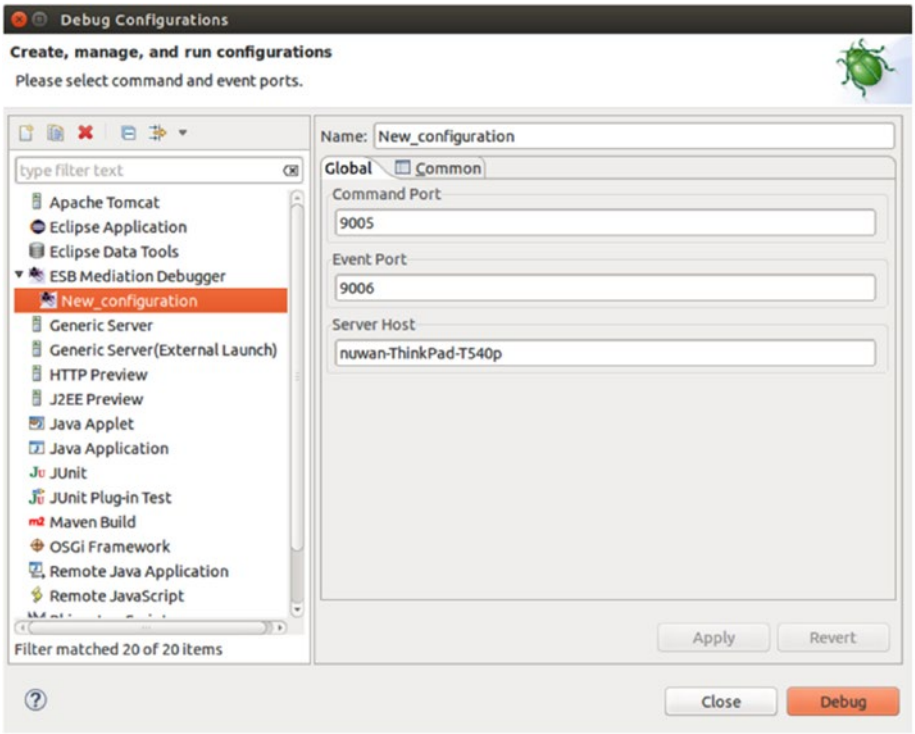

Figure 10-18. *Need caption*

Now the ESB tool is ready to be used as the debugger. So, ESB runtime has to be started with the debugging mode enabled. This can be done with an additional parameter to the starting script, `sh wso2server.sh -Desb.debug`. (You have approximately one minute to connect the WSO2 ESB developer tool with the remote debugging enabled ESB runtime. ESB will start if the developer tool connects to the server and ESB will start anyway when the waiting time expires.)

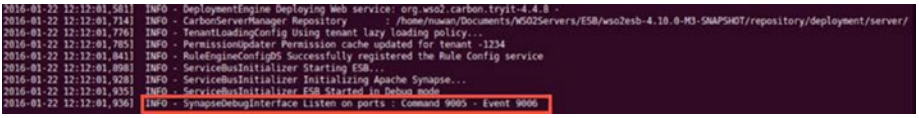

Figure 10-19. *Need caption*

Once you click Debug in the WSO2 ESB developer tool, it starts connecting to the ESB runtime. Now you can set all the required breakpoints in the ESB mediation flow by just right-clicking and choosing Toggle Breakpoint. It is require sync the ESB runtime and the developer tool by resending breakpoints when you do a change to the existing breakpoints. Now you can send a message through the configuration that you have set the

debug points and you will observe that those breakpoints are triggered when the message goes through the mediation logic.

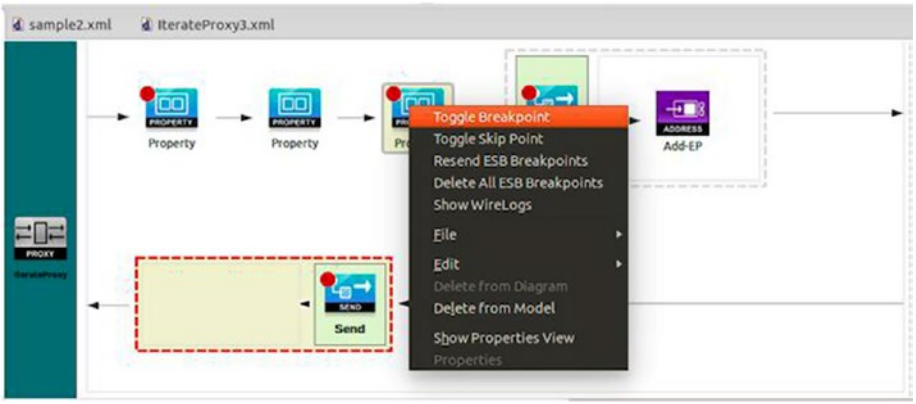

Figure 10-20. *Need caption*

When the breakpoint is triggered, you can see the message payload associated with that and the other attributes of the message.

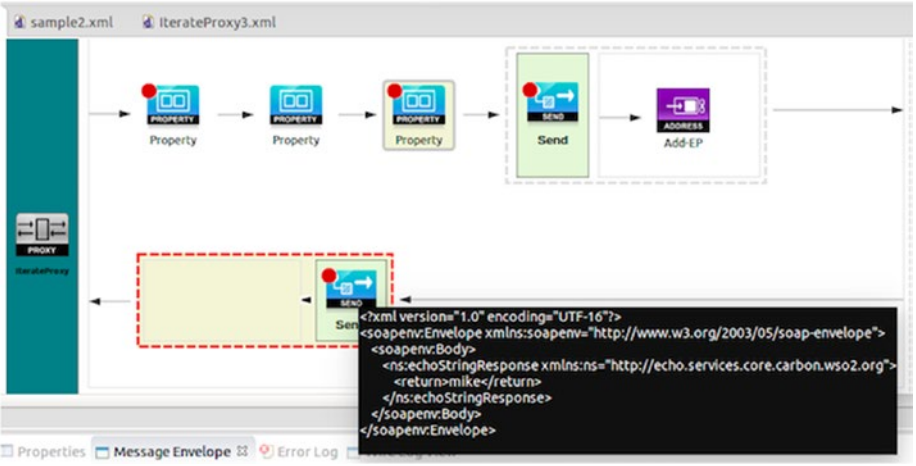

Figure 10-21. *Need caption*

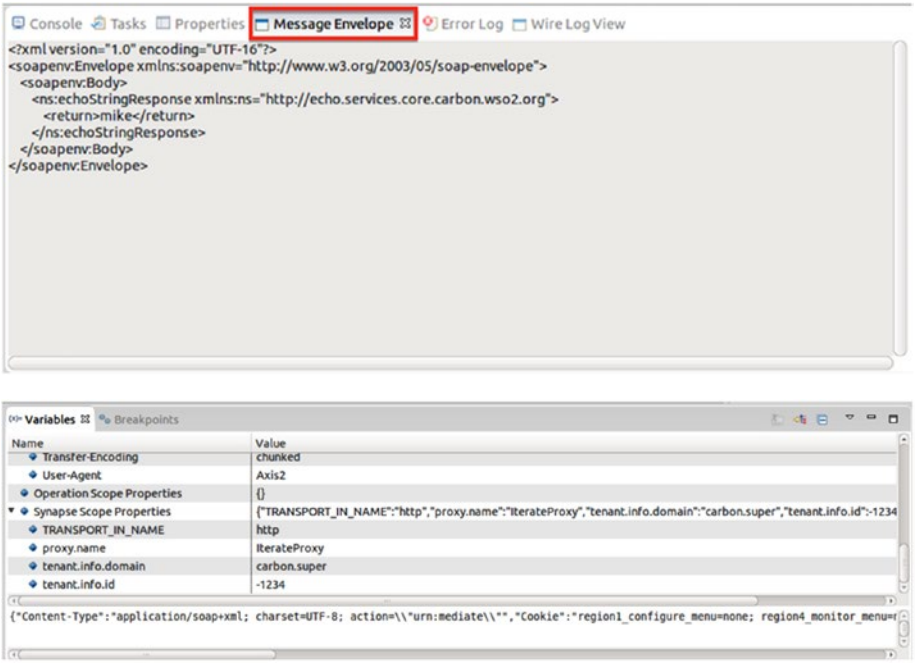

Figure 10-21. (*continued*)

You can also use the debugger to view the wire logs associated with the incoming/outgoing HTTP messages.

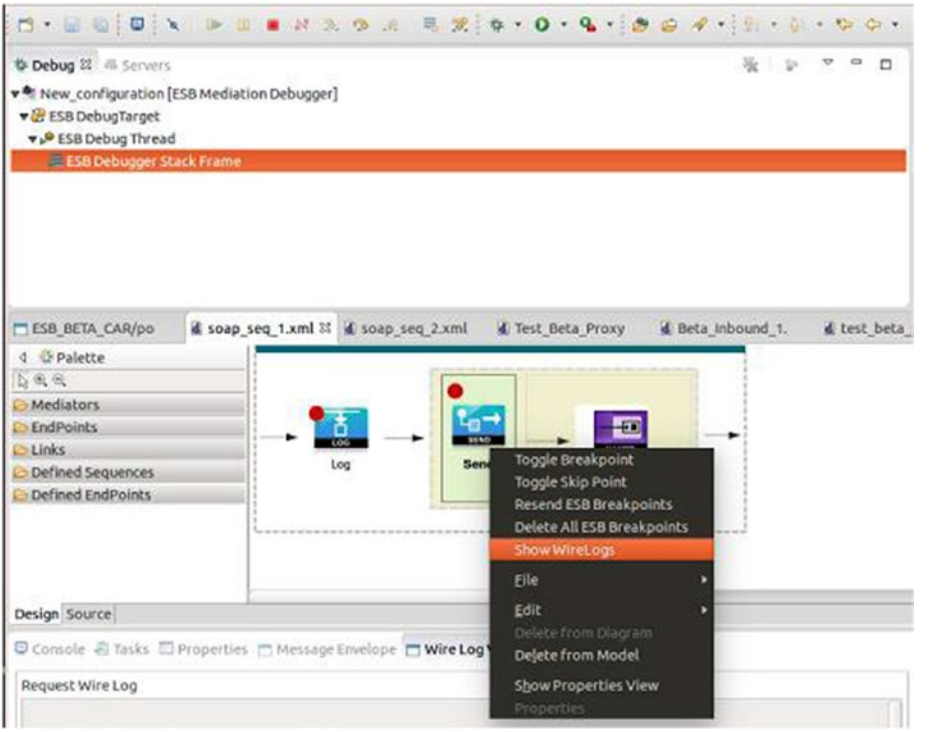

Figure 10-22. *Need caption*

The wire logs are shown in a separate tab at the bottom of the developer tool.

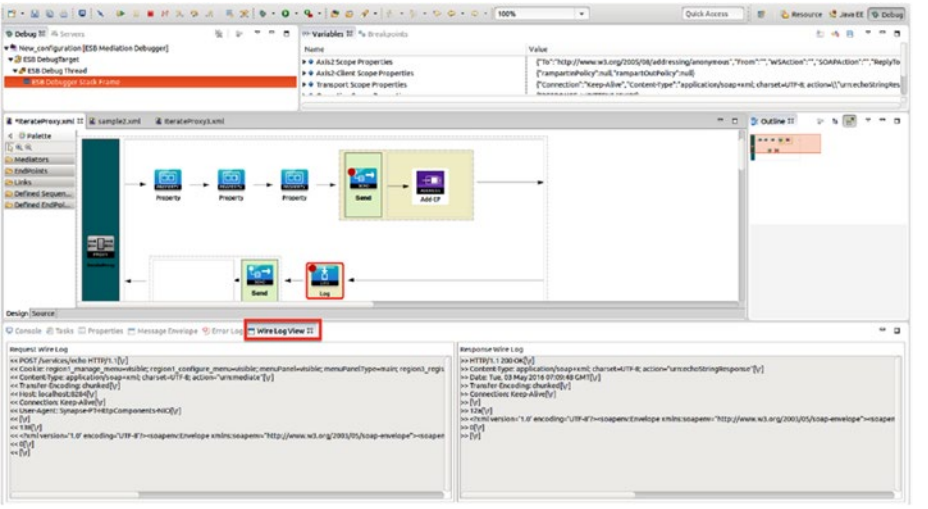

Figure 10-23. *Need caption*

It is also possible to modify and clear message attributes using the debugger tool. You can use this feature to modify different message properties and try out different use cases.

Deploying Artifacts Across Multiple Environments

When you are running WSO2 ESB in your organization, it is recommended to have a software development process that has dedicated resources for development, QA/Stress/Pre-Prod, and production environments. That means that the development of the integration scenarios starts from the development environment and then promotes to QA, then to any other intermediate environment, and finally to the production environment. So, it is really important for you to design and implement your integration scenarios so that the artifacts can be moved across these environments without much effort.

One of the fundamental design requirements of building such an integration scenario with WSO2 ESB is to identify the configuration parameters of your integration logic. That is the environmentally dependent and environmentally independent portion of it.

- *Environmentally dependent artifacts*: The most common environmental specific artifacts are the endpoints. The value of the endpoint drastically changes when you move the WSO2 ESB configuration across different environments.
- *Environmentally independent artifacts*: Most of the logics related to ESB mediation flow, data mapping configurations, etc. are independent from the environment.

Once you identify which part of your ESB logic contains the environment specific information, you should externalize them.

There are several ways that you can externalize such a configuration. One option that is provided along with the ESB is to externalize the endpoint as a registry resource. In this case, we create all the endpoints as registry resources and inside sequences we only refer them with the endpoint key. And we split the configuration into two different composite applications. One composite application contains the environmental independent artifacts, and hence it can be deployed across all environments without any issues. Then we have the environmental dependent composite application (which primarily contains registry resources) and we can keep several resources per each environment (with the same registry resource key).

Then we deploy respective composite application in the each environment. However, this approach is constrained to the things that you can keep as registry resources in WSO2 ESB (such as endpoint, XSLT files, etc.).

We can use the maven build tool to externalize the environmental dependent configuration using maven resource variables. As shown in Figure 10-24, you can use the maven resource variable concept to represent any environment specific information in your ESB configuration. And for each environment, you have to create a composite application by providing the required values for each variable.

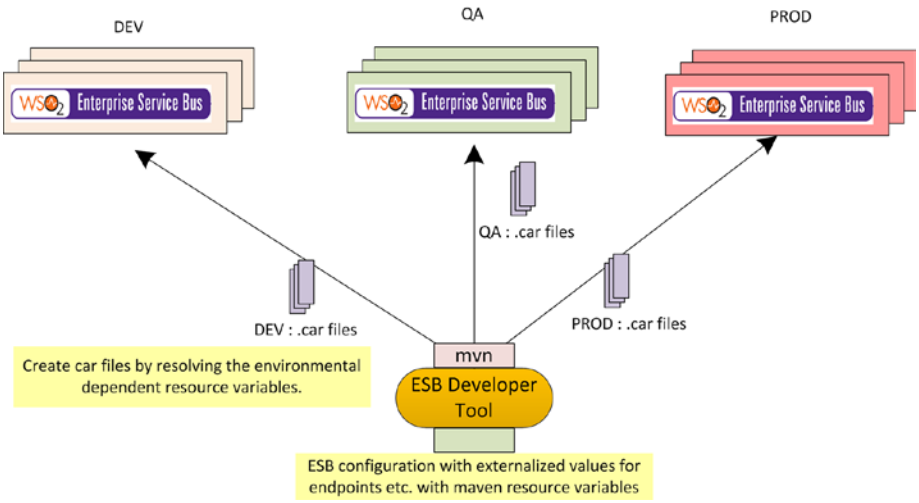

Figure 10-24. *Deploying artifacts across different environments using maven. Environmentally dependent configuration is represented as maven resource variables and they are getting substituted when we create a .car file through maven*

For instance, you can represent an endpoint address as follows in your sequences.

```
<call>
    <endpoint name="PizzaShoptRESTEndpoint">
         <address uri="${service.pizzashop.ep}"/>
    </endpoint>

</call>
```

Then, when you build the composite application using maven, you can pass actual environment specific values when you create the composite application for that particular environment.

```
 -Dservice.pizzashop.ep ="http://dev.pizzashop.com:6060/services/
PizzaShotRESTWS"
```

These values can come from system properties, your project properties, from your filter resources, and from the command line. So, with this approach, you can completely parameterize your ESB configuration.

In order to create a project that can build with maven, you need to create a "maven multi-module project" and include all the ESB configuration-related projects as a module of that. This is quite straightforward in the developer tool. During the creation of multi-module project, it asks for projects that you want to include.

Deployment Methodology

Once you have the development process in place, you need to think about the deployment patterns of WSO2 ESB. There are several patterns that you can use across different use cases. However, we will focus only on the simplest and most stable deployment patterns of WSO2 ESB.

The pattern illustrated in Figure 10-25, is primarily for all stateless use cases (where you don't use coordination features among ESB cluster nodes such as polling inbound endpoint coordination, message store/processor coordination, or task coordination). You can front a cluster of ESB nodes by any software or hardware load balancer. The ESB cluster is completely stateless and has no communication between each part. Health checking, monitoring, and adding, or removing nodes is totally up to the load balancer. Scaling the cluster is quite straightforward and it is just a matter of adding mode nodes to the cluster when you want to scale up (nothing needs to be done at the ESB layer, but at the load balancer level, you can add another ESB node).

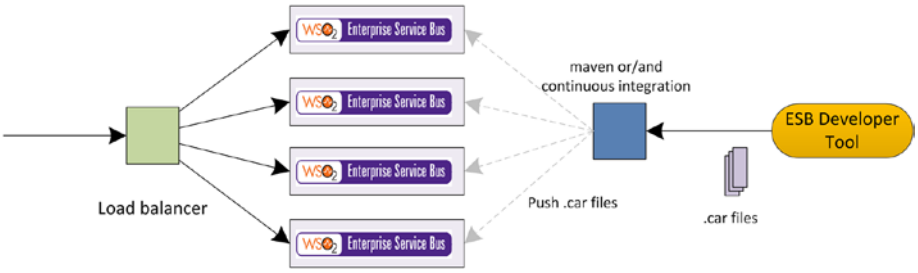

Figure 10-25. *Deploying artifacts across different environments using maven*

The ESB mediation configuration can be deployed into each cluster node by pushing the composite application archives (`.car` files) to each node. Often this can be automated and integrated to a continuous integration tool such as Jenkins so that `.car` files are auto-tested prior to pushing to ESB cluster nodes. The `.car` files has to be pushed into `$ESB_HOME/repository/deployment/server/carbonapps` of each cluster node.

Figure 10-26 shows a slight variation of the previous pattern. In this case, the only difference is that we are using a shared file system to synchronize configuration across the ESB cluster nodes. The ESB developer tool can be used to create the configuration and push the `.car` files to the shared file system through the maven/continuous integration tools.

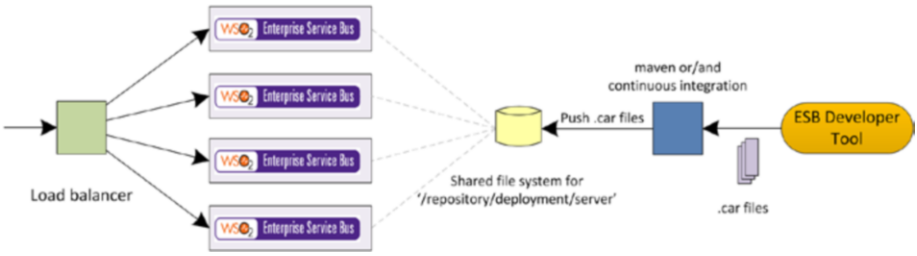

Figure 10-26. *Deploying artifacts across different environments using maven*

However, if you want to support a use case that requires coordination capability (polling inbound endpoints such as JMS, files, message stores and processors, and tasks), you need to create a stateful cluster. That means that cluster communication happens between each of the cluster nodes.

For these types of use cases, you need to deploy the cluster using the membership discovery style clustering mechanism supported in WSO2 ESB.

Here, we use Well Known Addressing (WKA)-based clustering of ESB nodes. With WKA, there is a set of well-known members and every node in the cluster knows about these members. When a new node wants to become a member of the cluster, it connects to one of the well-known members and declares its details. Then the well-known member provides all the information about the cluster and lets every member in the cluster know about the new node. This allows the node to become a member of the cluster. It is recommend to have more than two well-known members in a cluster for high-availability.

As shown in Figure 10-27, every ESB node in the cluster knows about the two well-known members. So, when a new node wants to connect to the cluster, it has to know the details of either of those well-known members. Let's look at how you configure an ESB node as a well-known member. In the `axis2.xml` file, you need to configure the clustering section as shown in Listing 10-1.

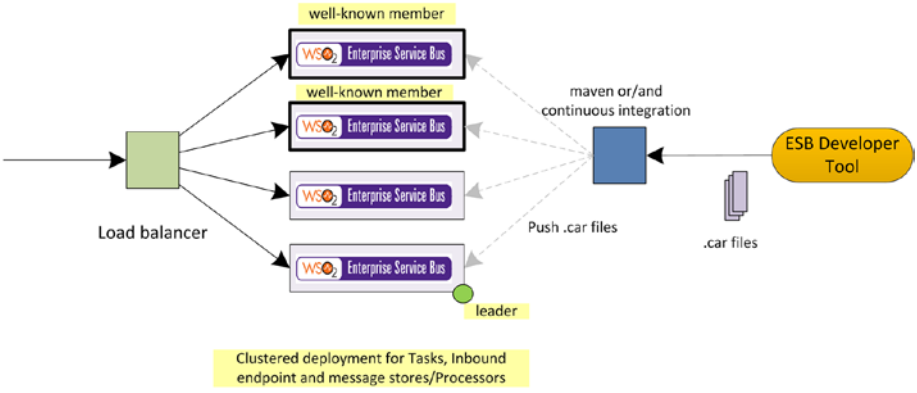

Figure 10-27. *Deploying artifacts across different environments using maven*

Listing 10-1. Configuring axis2.xml of the Well-Known member1

```
<!-- 1) Enable clustering -->
<clustering class="org.wso2.carbon.core.clustering.hazelcast.
HazelcastClusteringAgent" enable="true">
   <parameter name="AvoidInitiation">true</parameter>

   <!-- 2) Configure wka -->
   <parameter name="membershipScheme">wka</parameter>
   <!-- 3) Specify clustering domain. Clustering messages will only be sent
   to the members of that particular domain  -->
   <parameter name="domain">wso2.esb.domain</parameter>

   <parameter name="mcastPort">45564</parameter>
   <parameter name="mcastTTL">100</parameter>
   <parameter name="mcastTimeout">60</parameter>

   <!-- 4) Host name or IP of this well-known member -->
   <parameter name="localMemberHost">wkm1.wso2.com</parameter>

   <!-- 5) Set the port of this server to listen to cluster messages -->
   <parameter name="localMemberPort">4100</parameter>

   <parameter name="properties">
      <property name="backendServerURL" value="https://${hostName}:
      ${httpsPort}/services/"></property>
      <property name="mgtConsoleURL" value="https://${hostName}:
      ${httpsPort}/"></property>
   </parameter>

   <!-- 6) Specify the details of the other well-known member -->
   <members>
      <member>
         <hostname>wkm2.wso2.com</hostname>
         <port>4200</port>
      </member>
   </members>

   <groupmanagement enable="false">
      <applicationdomain agent="org.wso2.carbon.core.clustering.hazelcast.
      HazelcastGroupManagementAgent" description="ESB group" name="wso2.esb.
      domain" port="2222" subdomain="worker">
   </applicationdomain></groupmanagement>
</clustering>
```

Since we have two well-known members, they are pointing to each other for their well-known member. Hence the `axis2.xml-clustering` configuration of the other well-known member is shown in Listing 10-2.

Listing 10-2. Configuring axis2.xml of the Well-Known member2

```
<clustering class="org.wso2.carbon.core.clustering.hazelcast.
HazelcastClusteringAgent" enable="true">
   <parameter name="AvoidInitiation">true</parameter>

   <!-- 2) Configure wka -->
   <parameter name="membershipScheme">wka</parameter>
   <!-- 3) Specify clustering domain. Clustering messages will only be sent
   to the members of that particular domain  -->
   <parameter name="domain">wso2.esb.domain</parameter>

   <parameter name="mcastPort">45564</parameter>
   <parameter name="mcastTTL">100</parameter>
   <parameter name="mcastTimeout">60</parameter>

   <!-- 4) Host name or IP of this well-known member -->
   <parameter name="localMemberHost">wkm2.wso2.com</parameter>

   <!-- 5) Set the port of this server to listen to cluster messages -->
   <parameter name="localMemberPort">4200</parameter>

   <parameter name="properties">
      <property name="backendServerURL" value="https://${hostName}:
      ${httpsPort}/services/"></property>
      <property name="mgtConsoleURL" value="https://${hostName}:
      ${httpsPort}/"></property>
   </parameter>
   <!-- 6) Specify the details of the other well-known member -->
   <members>
      <member>
         <hostname>wkm1.wso2.com</hostname>
         <port>4100</port>
      </member>
   </members>

   <groupmanagement enable="false">
      <applicationdomain agent="org.wso2.carbon.core.clustering.hazelcast.
      HazelcastGroupManagementAgent" description="ESB group" name="wso2.esb.
      domain" port="2222" subdomain="worker">
   </applicationdomain></groupmanagement>
</clustering>
```

Now each non-well known member or dynamic member can connect to either of those well-known members. The `axis2.xml` clustering configuration of such a dynamic member is shown in Listing 10-3.

Listing 10-3. Configuring axis2.xml for a Dynamic Member

```
<clustering class="org.wso2.carbon.core.clustering.hazelcast.
HazelcastClusteringAgent" enable="true">
   <parameter name="AvoidInitiation">true</parameter>

   <!-- 2) Configure wka -->
   <parameter name="membershipScheme">wka</parameter>
   <!-- 3) Specify clustering domain. Clustering messages will only be sent
   to the members of that particular domain  -->
   <parameter name="domain">wso2.esb.domain</parameter>

   <parameter name="mcastPort">45564</parameter>
   <parameter name="mcastTTL">100</parameter>
   <parameter name="mcastTimeout">60</parameter>

   <!-- 4) Host name or IP of this well-known member -->
   <parameter name="localMemberHost">member1.wso2.com</parameter>

   <!-- 5) Set the port of this server to listen to cluster messages -->
   <parameter name="localMemberPort">4200</parameter>

   <parameter name="properties">
      <property name="backendServerURL" value="https://${hostName}:
      ${httpsPort}/services/"></property>
      <property name="mgtConsoleURL" value="https://${hostName}:
      ${httpsPort}/"></property>
   </parameter>

   <!-- 6) Specify the details of the other well-known member -->
   <members>
      <member>
         <hostname>wkm1.wso2.com</hostname>
         <port>4100</port>
      </member>
      <member>
         <hostname>wkm2.wso2.com</hostname>
         <port>4200</port>
      </member>
   </members>

   <groupmanagement enable="false">
      <applicationdomain agent="org.wso2.carbon.core.clustering.hazelcast.
      HazelcastGroupManagementAgent" description="ESB group" name="wso2.esb.
      domain" port="2222" subdomain="worker">
   </applicationdomain></groupmanagement>
</clustering>
```

The configuration of the dynamic members is quite similar, apart from specifying the members list. We put both `wkm1` and `wkm2` as part of the member list. Doing this automatically makes the other two members WKAs. There is no other special configuration to make a member a WKA member.

So, with this you can build an ESB cluster, which is stateful and can support coordination among the cluster members. Now suppose you deploy a JMS inbound endpoint with coordination enabled or task with coordination enabled. That task will be executed on one of the cluster nodes (this has nothing to do with the well-known members). We can call this node the leader, and as per Figure 10-27, and it's the ESB node at the bottom. For some reason, if that node is failed/shut down, then there will be a new leader election between these cluster nodes and a new leader will be elected. In this approach, the configuration can be pushed into cluster nodes or can use shared file storage.

Another popular deployment pattern is to deploy ESB nodes as containers. Container-based deployment is fully supported by WSO2 ESB. If you want to make the nodes even more lightweight, that can be done by removing unwanted features from ESB. Check out `https://docs.wso2.com/display/ESB500/Managing+Features` for details.

Summary

In this chapter you learned:

- The development process of implementing integration scenarios with WSO2 ESB.
- How to use WSO2 developer studio to build and deploy your integration scenario.
- How to use visual data mapping and mediation debugging.
- How to move your artifacts across multiple environments.
- Commonly used deployment patterns of WSO2 ESB.

CHAPTER 11

Administrating and Extending WSO2 ESB

This chapter is all about how you administrate your ESB deployment and how you can extend it for any custom requirement that is not support out of the box.

WSO2 ESB Analytics

You learned about three main components of the ESB product in the previous chapters—runtime, tool, and analytics. In this chapter, we focus on the ESB analytics component. In the same way you download the ESB product or tool, you can download the ESB analytics server. The ESB analytics provides two main functionalities related to your ESB runtime.

- *Statistics*: Coarse-grain and fine-grain statistics of your integration/mediation flow.
- *Message tracing:* Enable tracing for a selected message flow and trace through the message path.

The way that ESB runtime and analytics runtime works is that ESB publishes events/data to the analytics server via the Thrift protocol. The configuration of the connectivity between ESB runtime and the ESB analytics server is configured in `<ESB_HOME>/repository/deployment/server/eventpublishers/MessageFlowConfigurationPublisher.xml`. In order to collect ESB mediation statistics and message tracing data, you need to enable statistics and tracing as follows.

ESB analytics data collection can be configured by setting the following properties in the `<ESB_HOME>/repository/conf/synapse.properties` file. You can choose the option of collecting payload and properties if needed.

```
mediation.flow.statistics.enable=true
mediation.flow.statistics.tracer.collect.payloads=true
mediation.flow.statistics.tracer.collect.properties=true
```


K. Indrasiri, *Beginning WSO2 ESB*, DOI 10.1007/978-1-4842-2343-7_11

However, this configuration parameter doesn't enable statistics for all artifacts by default. If you need to collect statistics or/and tracing information for a particular artifact such as a proxy service, REST API, sequence, endpoint, etc., that has to be enabled at each artifact level (If you want to, you can use `mediation.flow.statistics.collect.all=true` to enable statistics for all artifacts, so that you don't need to enable it at each artifact level.) By default, when you download the ESB runtime and ESB analytics server, you just need to enable these parameters on the ESB side and they will automatically connect to your local machine. However, if the analytics servers are configured in a remote machine, they need to be configured in the event publisher configuration discussed earlier.

Suppose that you installed and enabled statistics and tracing at the `synapse.properties` file for the use case shown in Figure 11-1.

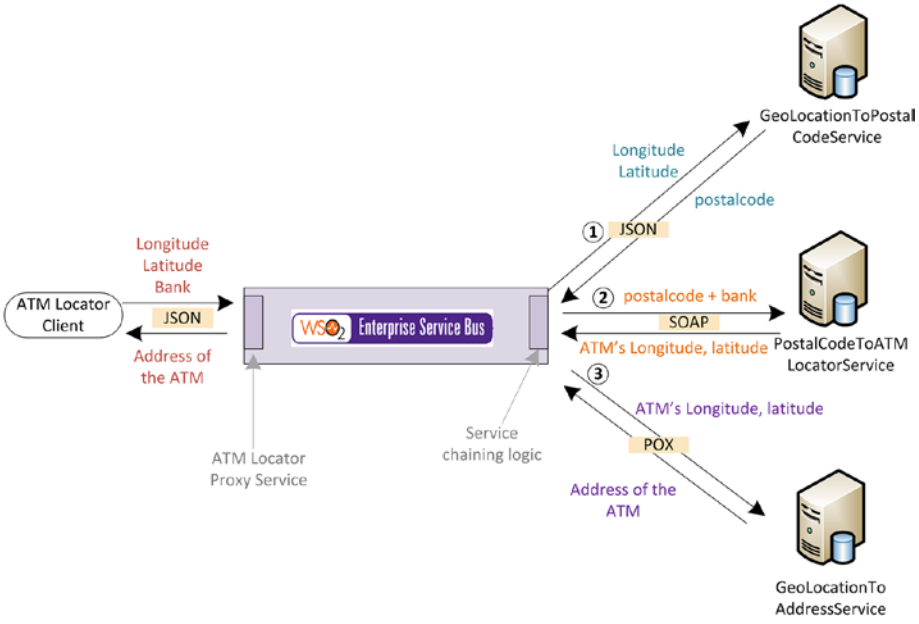

Figure 11-1. *A sample service orchestration scenario with numerous message transformations*

We have selected this use case as it has multiple endpoints and several message transformation logics. Now suppose that you enable statistics at the `synapse.properties` level and at the artifact level and enabled statistics and tracing. This is done by using `statistics="enable" trace="enable"`.

You can configure it to collect statistics or/and trace messages in a given artifact. Statistics collection is lightweight but message tracing consumes resources, as it keeps the message payload at each point where we change the payload.

Listing 11-1. ESB Configuration of the Service Orchestration Scenario

```
<api context="/atm/locator" name="ATMLocatorRESTAPI" statistics="enable"
trace="enable" xmlns="http://ws.apache.org/ns/synapse">
    <resource faultSequence="ATMLocatorFaultSeq" methods="POST">
        <inSequence>
            <log level="full">
                <property name="API_Log" value="=== Req Received at API
                Resource ==="/>
            </log>
            <datamapper config="gov:datamapper/Req2S1.dmc"
            inputSchema="gov:datamapper/Req2S1_inputSchema.json"
            inputType="JSON" outputSchema="gov:datamapper/Req2S1_
            outputSchema.json" outputType="JSON"/>
            <call>
                <endpoint key="Geo2PostalEP"/>
            </call>
            <datamapper config="gov:datamapper/Postalcode2ATMs.dmc"
            inputSchema="gov:datamapper/Postalcode2ATMs_inputSchema.json"
            inputType="JSON" outputSchema="gov:datamapper/Postalcode2ATMs_
            outputSchema.json" outputType="XML"/>
            <header name="Action" scope="default" value="urn:PostalcodeToAT
            MLocatorService"/>
            <call>
                <endpoint key="Postalcode2ATMEP"/>
            </call>
            <datamapper config="gov:datamapper/ATMLocation2Coordinates.
            dmc" inputSchema="gov:datamapper/ATMLocation2Coordinates_
            inputSchema.json" inputType="XML" outputSchema="gov:datamapper/
            ATMLocation2Coordinates_outputSchema.json" outputType="XML"/>
            <call>
                <endpoint key="ATMGeo2Address"/>
            </call>
            <datamapper config="gov:datamapper/ATMAddress2Res.dmc"
            inputSchema="gov:datamapper/ATMAddress2Res_inputSchema.json"
            inputType="XML" outputSchema="gov:datamapper/ATMAddress2Res_
            outputSchema.json" outputType="JSON"/>
            <respond/>
        </inSequence>
        <outSequence/>
    </resource>
</api>
```

Once you send requests through this REST API, you can log in to the ESB analytics server dashboard (https://localhost:9444/portal/dashboards/esb-analytics/) and observe the statistics and message-tracing portal.

The overview sections show the overall statistics of your ESB instance. Overall TPS, message count, successful and failed counts, usage of each ESB artifact, and so on, are shown in the overview section, as illustrated in Figure 11-2.

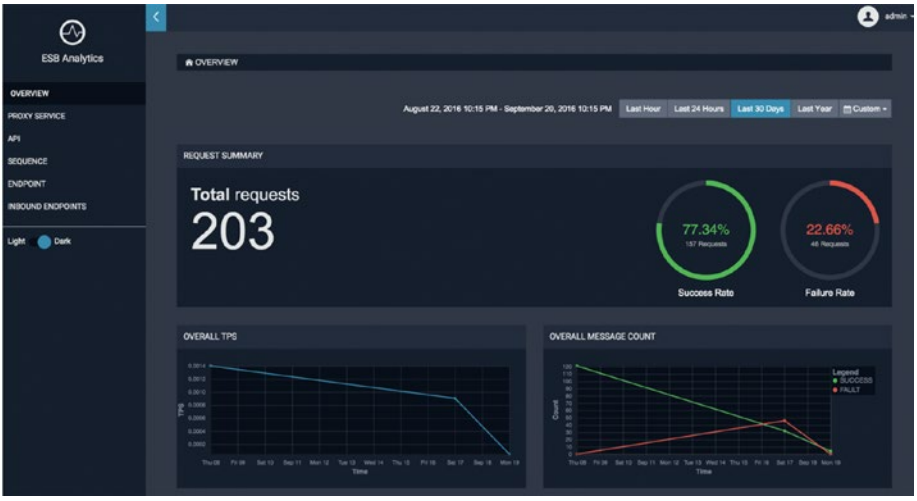

Figure 11-2. *ESB Statistics - Overview*

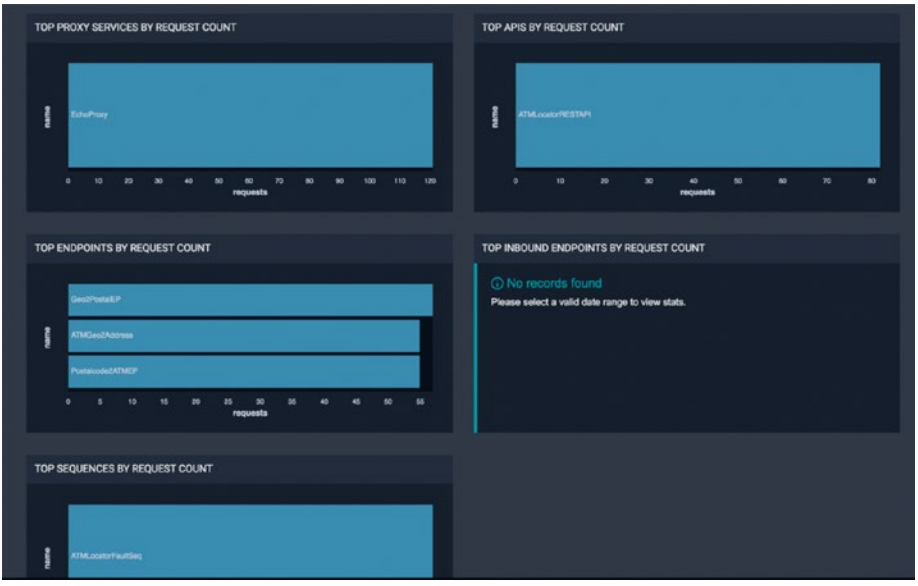

Figure 11-3. *Artifact level statistics*

Then you can further drill down to the analytics of each artifact type. Since we are using a REST API, we can select the REST API and observe the statistics collect against that API.

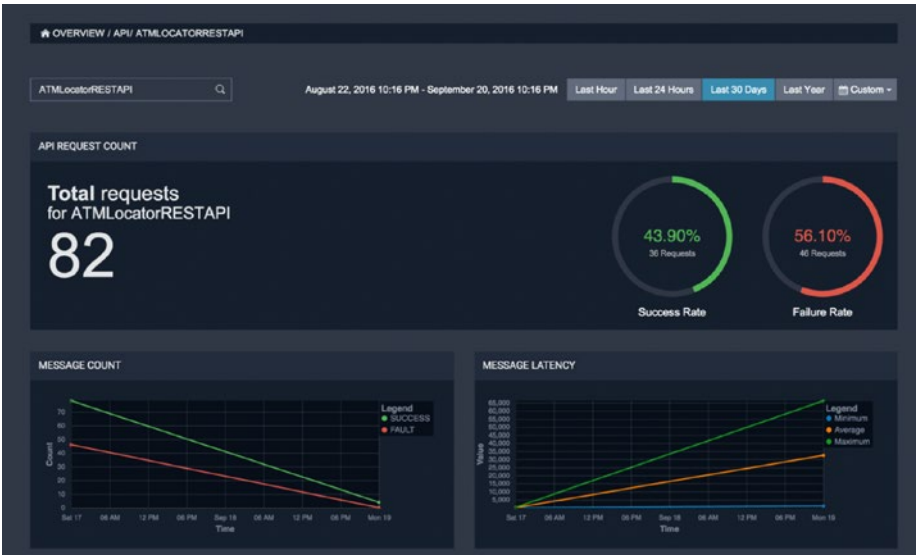

Figure 11-4. *Statistics of a given REST API/HTTP Service*

Also you can observe all the messages that go through the API along with its message ID with status (success or failed).

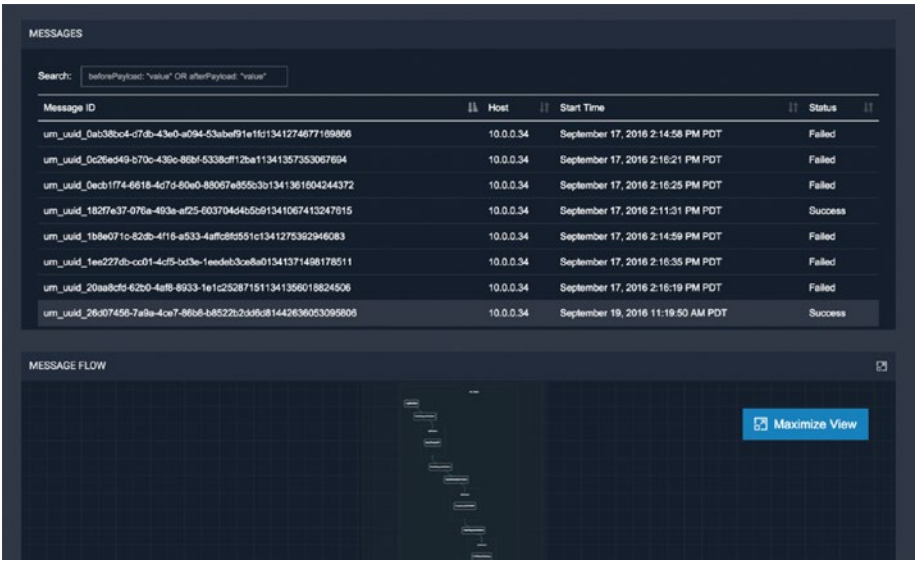

Figure 11-5. *List of messages that are processed by ESB and their message path through WSO2 ESB*

Then at the bottom, the overall message flow diagram is shown with the average processing time at each mediator/endpoint level.

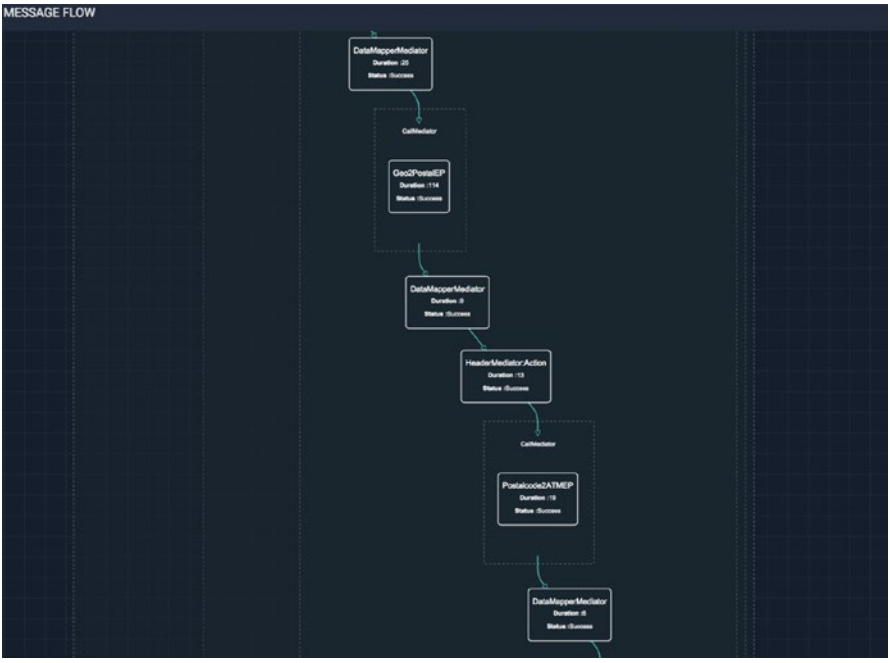

Figure 11-6. *Drill-down mediation statistics of a given message across different message processing component*

For any specific message ID, you can trace through the path by clicking on the message that you want to trace.

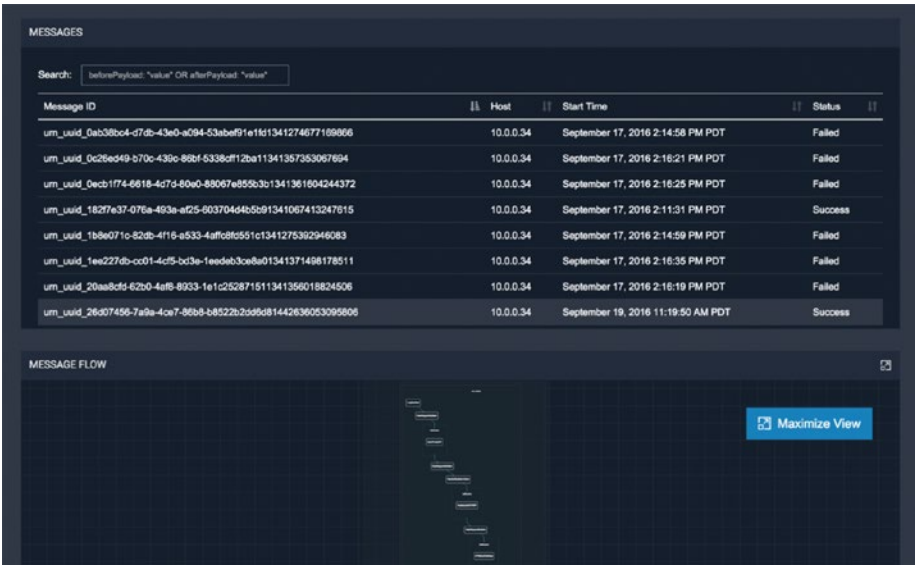

Figure 11-7. *Tracing the message flow of a selected message*

For example, if you observe one of the failure messages, you can find out that it failed during the invocation of the first service, at which point the fault sequence was triggered.

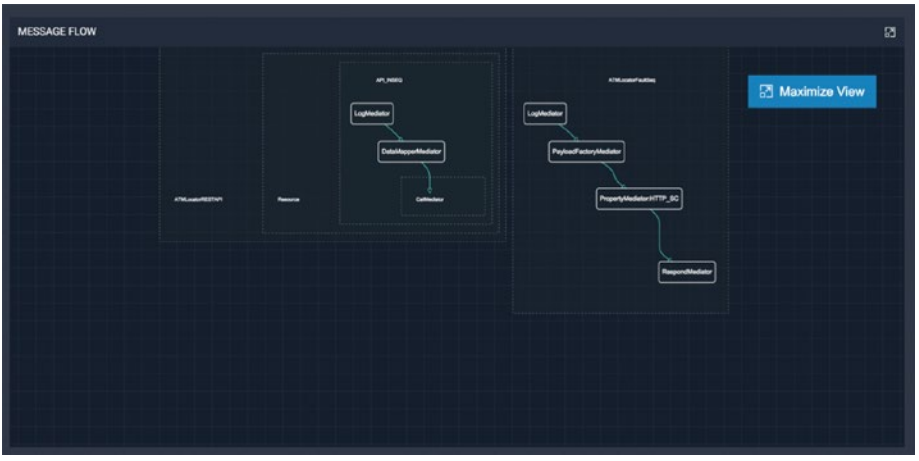

Figure 11-8. *Message tracing UI for a given message*

You can also view the payload associated with each mediator in that particular message path of the failed message.

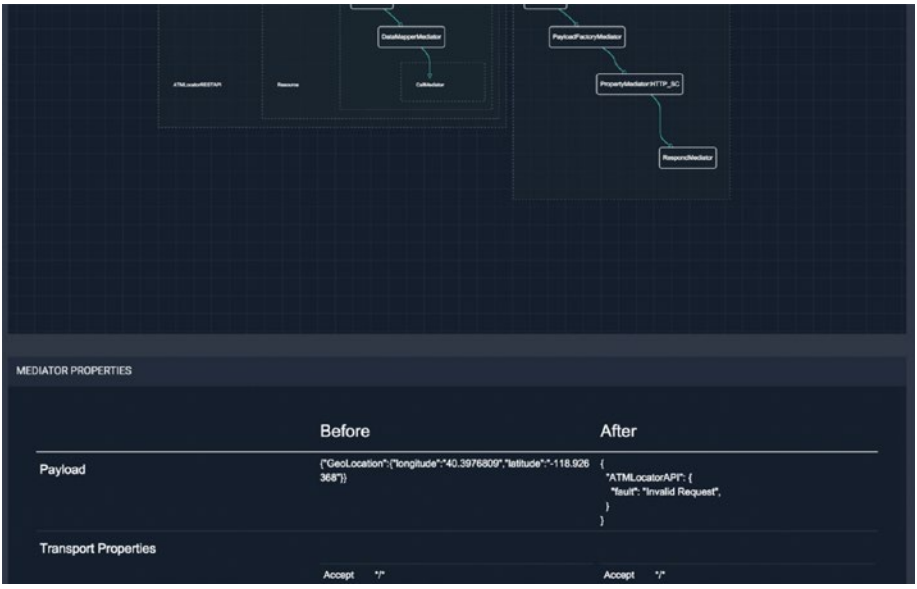

Figure 11-9. *Using message tracing of a given message ID and observer the message payloads and message attributes throughout the ESB message mediation flow*

You can enable statistics for all the artifacts while tracing can be applied for a selected artifact for a limited period of time (due to resource consumption). Often message tracing is useful for identifying the reason for various request failures in a production environment.

For high load, you may have to scale up the ESB analytics server. Since the ESB analytics server is based on the WSO2 Data Analytics Server (DAS), you can refer `https://docs.wso2.com/display/DAS310/Deployment+and+Clustering` for details on how to set up a high-available ESB analytics cluster.

Monitoring

When you run the ESB instance in your production environment, you have to monitor it for any abnormal behaviors that could affect the live traffic. For that, WSO2 ESB exposes the runtime status through JMX (Java Management Extension). JMX is a common method to manage and monitor the runtime parameters of a remote server. You can observe almost all the details related to ESB mediation flow statistics through JMX as well. You can use JConsole and connect to the ESB runtime to obtain this information.

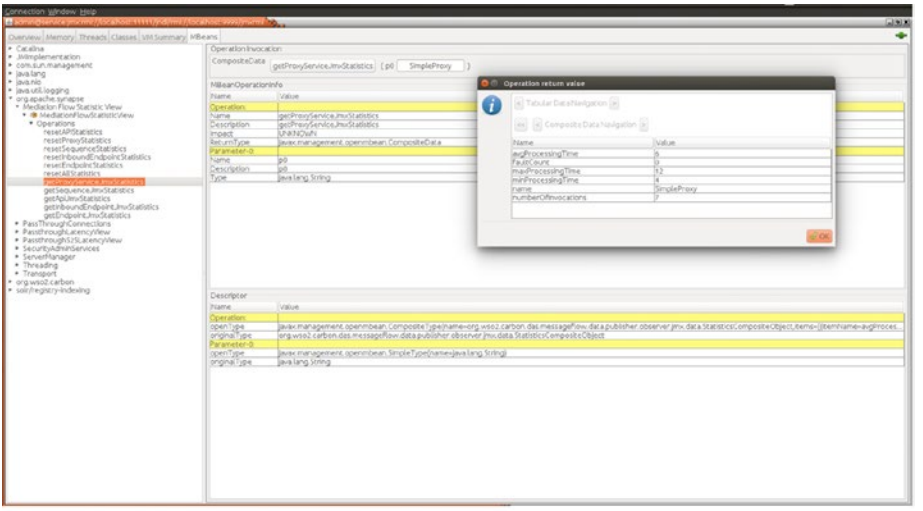

Figure 11-10. *Mediation statistics are also exposed through JMX*

In addition to the high-level statistics of ESB artifacts, WSO2 ESB also exposes the low-level latency information of its transports.

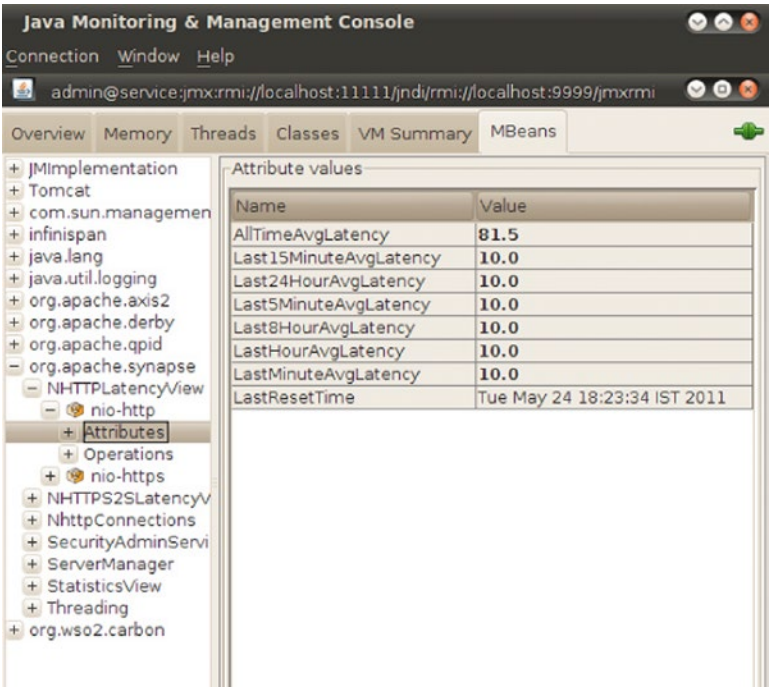

Figure 11-11. *Low-level HTTP transport details are exposed through JMX*

You can find more details of these MBeans at http://docs.wso2.com/enterprise-service-bus/JMX+Monitoring.

Extending WSO2 ESB

WSO2 ESB provides most of the standard features that you will need in most of the integration scenarios. However, it is not uncommon that you may come across certain requirements that are not supported out of the box by WSO2 ESB. This is where you will have to use extension points in WSO2 ESB.

Class Mediator

One of the most common extensions that you have to implement is the class mediator. It is quite useful when you want to inject custom message processing/mediation logic inside an ESB sequence.

The class mediator creates an instance of a custom-specified class and sets it as a mediator. The class must implement the `org.apache.synapse.api.Mediator` interface or extend the `AbstractMediator`. If any properties are specified as part of the class mediator configuration, the corresponding setter methods are invoked once on the class during initialization.

In order to create a class mediator, you can select Mediator Project and it will create a class mediator skeleton, as shown in Listing 11-2.

Listing 11-2. Class Mediator Structure

```
package kasun.panorama;

import org.apache.synapse.MessageContext;
import org.apache.synapse.mediators.AbstractMediator;

public class MyClassMediator extends AbstractMediator {

        public boolean mediate(MessageContext context) {
                // TODO Implement your mediation logic here
                return true;
        }
}
```

You can access all the information related to the incoming message through the message context object and you can implement any message processing logic inside the mediate method.

Script Mediator

The script mediator is used to invoke the functions of a variety of scripting languages such as JavaScript, Groovy, or Ruby. You can simply write the message processing logic using any of these languages and refer to the script from the script mediator inside a sequence.

Listing 11-3. Script Mediator Example

```
<log/>
<script language="js"
    key="repository/conf/sample/resources/script/test.js"
    function="testFunction"/>

// test.js (included as a registry resource)
function testFunction(mc) {
     var symbol = mc.getPayloadXML()..*::Code.toString();
     mc.setPayloadXML(
        <m:getQuote xmlns:m="http://services.samples/xsd">
           <m:request>
              <m:symbol>{symbol}</m:symbol>
           </m:request>
        </m:getQuote>);
}
```

Similarly, any other script can be included in this manner. There are specific methods such as `getPayloadXML()`, `getPayloadJSON()`, and so on, that are supported as part of message context API associated with script mediator. For all the available API methods and examples, refer to http://docs.wso2.com/enterprise-service-bus/Script+Mediator.

Custom Connector

Although there are hundreds of ESB connectors available with WSO2 ESB, you may have to address any specific custom requirement. For such requirements you can create your own connector to integrate with any cloud APIs or any internal legacy system. To build your own ESB connector, you can use the following Maven archetype:

```
mvn archetype:generate -DarchetypeGroupId=org.wso2.carbon.extension.
archetype -DarchetypeArtifactId=org.wso2.carbon.extension.esb.connector-
archetype -DarchetypeVersion=2.0.0 -DgroupId=org.wso2.carbon.connector
-DartifactId=org.wso2.carbon.connector.helloworld -Dversion=1.0.0
-DarchetypeRepository=http://maven.wso2.org/nexus/content/repositories/wso2-
public/
```

This creates the `org.wso2.carbon.esb.connector.helloworld` directory in the current location of your machine, with a directory structure similar to Figure 11-12.

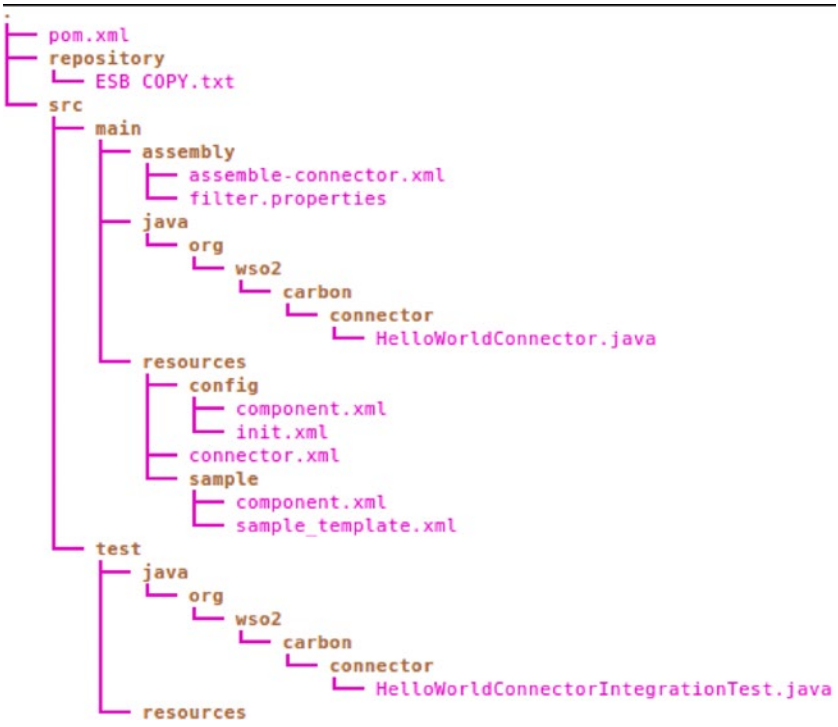

Figure 11-12. *Skeleton of the connector development project*

You simply have to create the required ESB templates and the required class mediators to handle the specific scenario. This will output a connector archive, which can be deployed into an ESB. Refer to https://docs.wso2.com/display/ESBCONNECTORS/Writing+a+Connector for more details.

Other Extensions

There are various other types of extensions you can use with WSO2 ESB. Tasks, custom message builders and formatters, custom inbound endpoints, etc. are some of the other commonly used extensions.

Error Handling

Any mediation logic that you develop should have proper error-handling semantics and logics. In many ESB artifacts, you can associate a fault-handling mechanism known as a *fault* sequence.

A fault sequence is a collection of mediators just like any other sequence, and it can be associated with another sequence or a proxy service. When the sequence, proxy

service, etc. encounters an error during mediation or while forwarding a message, the message that triggered the error is delegated to the specified fault sequence. Using the available mediators, it is possible to log the erroneous message, forward it to a special error-tracking service, and send a custom error message to the client. Proper error handling in the fault sequence prevents ESB from going to abnormal states and it also helps to maintain proper resource utilization.

Whenever an error occurs in WSO2 ESB, it attempts to provide as much information as possible about the error to the user through the `ERROR_CODE`, `ERROR_MESSAGE`, `ERROR_DETAIL`, and `ERROR_EXCEPTION` properties. These values can be added to logs via the log mediator. ESB has defined a set of error codes. For an example, if you have observed an error in your fault sequence with error code 101504, that means "connection timed out."

All the available error codes and the reasons for each error can be found at http://docs.wso2.com/enterprise-service-bus/Error+Handling.

Summary

In this chapter you learned:

- How to use the ESB analytics server with ESB runtime to obtain statistics of ESB mediation flow and to trace messages.
- JMX monitoring.
- Commonly used mediation extensions and how to implement them (class mediators, script mediators, and the custom connector).
- Error-handling techniques in WSO2 ESB.

Index

K. Indrasiri, *Beginning WSO2 ESB*, DOI 10.1007/978-1-4842-2343-7

H

I

J

K, L

M, N

O

P, Q

R

S

T, U, V

W, X, Y, Z