Apache Axis2 Web Services 2nd Edition

Create secure, reliable, and easy-to-use web services using Apache Axis2

Deepal Jayasinghe

Afkham Azeez

[PACKT] open source
community experience distilled
PUBLISHING

BIRMINGHAM - MUMBAI

Apache Axis2 Web Services
2nd Edition

First published: May 2008

Second Edition: February 2011

Production Reference: 1110211

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

ISBN 978-1-849511-56-8

www.packtpub.com

Cover Image by Ed Maclean (edmaclean@gmail.com)

Table of Contents

Preface

SOA, in practicality web services, is becoming the enabler for application integration. Since the introduction of web services, Apache Software Foundation has played a major role and produced several good web services frameworks. This book covers the defector Java Web Service framework, also known as Apache Axis2. This book covers several important facts that you would want to know about web services and writing, from simple web services to complex web services. By the end of this book, you will have learned about Axis2, its architectures and features, writing and deploying a simple service, writing service extensions and quality of services, POJO and JAX-WS services, clusters, and secure reliable web services.

What this book covers

Chapter 1, *Apache Web Services and Axis2* - Gives you an introduction to web services and the web service stack at Apache.

Chapter 2, *Looking inside Axis2* - Learn about Axis2 architecture and the importance of its components.

Chapter 3, *Axis2 XML Model (AXIOM)* - Learn about the heart of a web service framework and learning more about XML processing in Axis2.

Chapter 4, *Execution Chain* - Learn how to extend the core functionality of the framework though handlers.

Chapter 5, *Deployment Mode* - Learn about the new and user friendly deployment model and several ways of deploying a service in Axis2.

Chapter 6, *Information Model* - Learn how Axis2 stores it static and dynamic data and the importance of it.

Chapter 7, *Writing an Axis2 Service* - Learn how to write a simple-complex service using Axis2 and how to deploy it.

Chapter 8, *Writing an Axis2 Module* - Learn how to extend Axis2 core functionality through a self-contained package.

Chapter 9, *The Client API* - Learn how to use Axis2 to invoke other services, available APIs, and how to use them.

Chapter 10, *Session Management* - Go beyond single invocation and learn how to use Axis2 features to provide better and more efficient statefull service.

Chapter 11, *Developing JAX-WS Web Services* - Learn the fundamentals of developing JAXWS based web services, the most popular web service development technology used by Java developers.

Chapter 12, *Axis2 Clustering* - Learn about clustering Apache Axis2, which will allow you to deploy Axis2 in large scale production deployments.

Chapter 13, *Enterprise Integration Patterns* - Learn about some enterprise SOA deployment patterns that make use of the underlying Axis2 clustering infrastructure.

Chapter 14, *Axis2 Advanced Features and Usage* - Go beyond simple features and learn about REST, MTOM, and several other advanced features.

Chapter 15, *Building a Secure Reliable Web Service* - Learn how to use Axis2 and related components to make your service more secure and reliable.

What you need for this book

- Java 5 or above (Axis2 support only JDK 1.5 and above)
- Latest version of Axis2 (Axis2 1.5.4)

Who this book is for

This book is for Java developers who are interested in building web services using Apache Axis2. Familiarity with web standards such as SOAP, WSDL, and XML parsing is assumed.

Conventions

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

Code words in text are shown as follows: "The `AxisEngine` or driver of Axis2 defines two methods, `send()` and `receive()`, to implement these two pipes."

A block of code is set as follows:

```
//First, create the parser
XMLStreamReader parser = XMLInputFactory.newInstance().
createXMLStreamReader(new FileInputStream(file));
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
//the plain OMBuilder
        StAXBuilder builder = new StAXOMBuilder(byteArrayInputStream);
        //return the root element.
        OMElement root = builder.getDocumentElement();
        root.serialize(System.out);
```

New terms and **important words** are shown in bold: "Axis2 now comes handy with the flexibility to support **Message Exchange Patterns (MEPs)** with in-built support for basic MEPs defined in WSDL 2.0."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

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

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

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

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

Customer support

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

Downloading the example code for this book

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

Errata

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

Piracy

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

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

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

Questions

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

1
Apache Web Services and Axis2

Apache Axis2 is the next generation web service framework from Apache. The Apache software foundation started Apache SOAP as its first web service framework. Next, they developed Apache Axis, which became one of the very successful projects at Apache and is still used heavily in the industry. Due to rapid changes in the industry and demands from the user community, Apache Axis alone was not able to fulfill those requirements, thus the Apache Web Service community initiated Apache Axis2 project in 2004. In a short period of time, Apache Axis2 has become the de facto open source Java Web Service framework, which is now heavily used in both the industry and in academia. Axis2, the next generation of the Apache Web Service stack, takes one more step closer to the first production version, by releasing another developer version.

In this chapter, we will learn more about web services, its history, standards, as well as the components of web services. At the end of the chapter, we will discuss the need for a new web service engine, and finally how to install and run Axis2.

Here, we focus more on the web services and related technologies. In particular, we will cover:

- Service Oriented Architecture
- Overview of web services
- Web services standers and standard bodies
- Apache Web Service stack
- Getting started with Axis2

Service Oriented Architecture (SOA)

The era of isolated computers is over. Now "*connected we stand, isolated we fall*" is becoming the motto of computing. Networking and communication facilities have connected the world in a way as never before. The world has hardware that could support the systems that connect thousands of computers, and these systems have the capacity to wield power that was once only dreamed of.

Yet, computer science lacked the technologies and abstraction to utilize the established communication networks. The goal of distributed computing is to provide such abstractions. RPC, RMI, IIOP, and CORBA are a few proposals that provide abstractions over the network for the developers to build upon.

These proposals fail to consider one critical nature of the problem. The systems are a composition of numerous heterogeneous subsystems, but these proposals require all the participants to share a programming language or a few languages. **Service Oriented Architecture** (**SOA**) provides the answer by defining a set of concepts and patterns to integrate homogenous and heterogeneous components together. SOA provides a better way to achieve loosely coupled systems, and hence more extensibility and flexibility. In addition, similar to object-oriented programming (OOP), SOA enables a high degree of reusability. There are three main ways one can enable SOA capabilities in their systems and applications:

- Existing messaging systems: for example, IBM MQSeries, Tibco, JMS, and so on
- Plain Old XML (POX): for example, XML/HTTP, REST, and so on
- Web services: for example, SOAP, WSDL, WS-*

Among the commonly used messaging systems, **Java Messaging Service** (**JMS**) plays a major role in the industry and has become a common API for messaging systems. We can find a number of different message types of JMS, such as Text, Bytes, Name-Value pair, Stream, and Object. One of the main disadvantages of these types of messaging systems is that they do not have a single wire format (serialization format). As a result, interoperability is a big issue: if two applications are using JMS to communicate, then they must be on the same implementation. Sonic, Tibco, and IBM are the leaders in the commercial markets, and JBoss, Manta, and ActiveMQ are the commonly used open source implementations.

Plain Old XML or POX is another way of exposing functionality and enabling SOA in the system. With the widespread use of the Web, the POX approach has become more popular. Most of the web applications expose the XML APIs, where we can develop components and communicate with them. Google Maps, Auto complete, and Amazon services are a few examples of applications that heavily use XML APIs to expose the functionality. In most cases, POX is used in combination with REST (Representational State Transfer). **REST** is a model of an underlying architecture of the Web, and it is based on the concept that every URL identifies resources. GET, PUT, POST, and DELETE are the verbs that are used in the REST architecture. REST is often associated with the theoretical standpoints, and for this reason, REST is generally not used for complex interactions.

Among the three commonly used methods to enable SOA, a web service can be considered as the most standard and flexible way. Web services extend the idea of POX and add additional standards to make the communication more organized and standardized. There are several reasons behind the web services being the most popular SOA-enabled mechanism, as stated here:

- Web services are described using WSDL, and WSDL can capture any complex application and the required quality of services.
- Web services use SOAP as the message transmission mechanism, as SOAP is a special type of XML. It gains all the extensibility features from XML.
- There are a number of standard bodies to create and enforce the standards for web services.
- There are multiple open source and commercial web service implementations. By using the standards and procedures, web services provide application and programming language-independent mechanism to integrate and communicate. Different programming languages may define different implementations for web services, yet they interoperate because they all agree on the format of the information they share.

Web service overview

The Internet is revolutionizing business by providing an affordable and efficient way to link companies with their partners as well as customers. However, there are issues that reduce the productivity of the Internet. Among the issues, incompatible applications and frameworks that cannot interoperate or exchange business data are major concerns. Particularly, when using REST-based application, marshalling and unmarshalling data, as well as adding quality of support, is a major concern. Web Service is a new model for e-business that is expected to change the way business applications are developed, integrated and interoperate. Web Services are a self-describing, self-contained, modular application accessible over the Web. It is exposed as an XML interface, as well as it communicates with other services using XML messages over standard web protocols.

The fundamental concept behind web services is the SOA where an application is no longer a large monolithic program, but it is divided into smaller, loosely coupled programs. The provided services are loosely coupled together with standardized and well-defined interfaces. These loosely coupled programs make the architecture very extensible due to the possibility to add or remove services with limited costs. Therefore, new services can be created by combining existing services. To understand loose coupling clearly, it is better to understand the opposite, which is tight coupling, and its problems:

- Errors, delays, and downtime spread through the system
- The resilience of the whole system is based on the weakest part
- Cost of upgrading or migrating spreads
- It's hard to evaluate the useful parts from the dead weight

In web services, there are three main standard bodies that helped to improve the interoperability, quality of service, and base standards:

- WS-I
- OASIS
- W3C

The main functionality of WS-I is to provide standards and specifications to ensure interoperability, composablity, and profiling. In other words, to create standards and procedures to enforce the required level of interoperability among various web service frameworks. OASIS's main goal is to improve the quality of services of web services, which include security, reliability, transaction, and resource management. W3C, one of the web services standard bodies, defines a web service as a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a format that can be processed by machine and read by human. The format is known as **Web Services Description Language** (**WSDL**). Other applications communicate with the web service in a manner prescribed by its description using **Simple Object Access Protocol** (**SOAP**) messages, typically conveyed using HTTP with an XML serialization, in conjunction with other web-related standards.

A web service is a well-known open technology standard, which provides a number of benefits as listed here:

- Increased interoperability, resulting in lower maintenance costs
- Increased reusability and composablity (for example, use publicly available services and reuse them or integrate them to provide new services)
- Increased competition among vendors, resulting in lower product costs
- Easy transition from one product to another, resulting in lower training costs
- Greater degree of adoption and longevity for a standard, a large degree of usage from vendors and users leading to a higher degree of acceptance

One can argue that the web service concept is the logical evolution from object-oriented systems to systems of services. As in object-oriented systems, some of the fundamental concepts in web services are encapsulation, message passing, and dynamic binding. However, the service-based concept is extended beyond method signatures, as information related to what the service does, where it is located, how it is invoked, the quality of service, and security policy related to the service can also be published in the service interface (WSDL).

How do organizations move into web services?

There are three main ways in which an organization could possibly use to move into the web services, listed next:

- Create a new web service from scratch. The developer creates the functionalities of the services as well as the description.

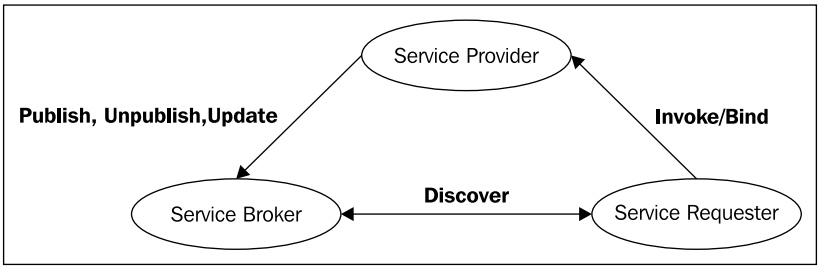
- Expose the existing functionality through a web service. Here the functionalities of the service already exist. Only the service description needs to be implemented.
- Integrate web services from other vendors or business partners. There are occasions when using a service implemented by another is more cost effective than building from the scratch. On these occasions, the organisation will need to integrate others' or even business partners' web services.

The real usage of web service concepts is for the second and third methods, which enables other web services and applications to use the existing applications.

Web services describe a new model for using the web; the model allows publication of business functions to the Web and provide universal access to those business functions. Both developers and end users benefit from web services. The web service model simplifies business application development and interoperation.

Web services model

Web services model consists of a set of basic functionalities such as describe, publish, discover, bind, invoke, update, and unpublish. In the meantime, the model also consists of three actors—service provider, service broker, and service requester. Both the functionalities as well as actors are shown in the next figure:

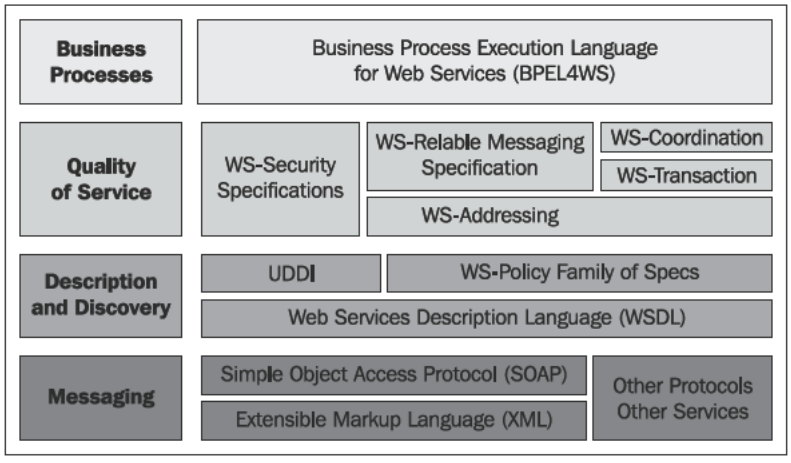


- **Service provider** is the individual (organisation) that provides the service. The service provider's job is to create, publish, maintain, and unpublish their services. From a business point of view, a service provider is the owner of the service. From an architectural view, a service provider is the platform that holds the implementation of the service. Google API, Yahoo! financial services, Amazon services, and weather services are some examples of service providers.

- **Service broker** provides a repository of service descriptions (WSDL). These descriptions are published by the service provider. Service requesters will search the repository to identify the required service and obtain the binding information for these services. Service broker can be either public, where the services are universally accessible, or private, where only a specified set of service requesters are able to access the service.
- **Service requester** is the party that is looking for a service to fulfil its requirements. A requester could be a human accessing the service or an application program (a program could also be a service). From a business view, this is the business that wants to fulfil a particular service. From an architectural view, this is the application that is looking for and invoking a service.

Web services standards

So far we have discussed SOA, standard bodies of web services, and the web service model. In this section, we are going to discuss more about standards, which make web services more usable and flexible. In the past few years, there has been a significant growth in the usage of web services as application integration mechanism. As mentioned earlier, a web service is different from other SOA exposing mechanisms because it consists of various standards to address issues encountered in the other two mechanisms. The growing collection of WS-* (for example, Web Service security, Web Service reliable messaging, Web Service addressing, and others) standards, supervised by the web services governing bodies, define the web service protocol stack shown in the following figure. Here we will be looking at the standards that have been specified in the most basic layers: messaging and description, and discovery.



The messaging standards are intended to give the framework for exchanging information in a distributed environment. These standards have to be reliable so that the message will be sent only once and only the intended receiver will receive it. This is one of the primary areas where research is being conducted, as everything depends on the messaging ability.

Next, we are going to discuss some of these standards in details. Firstly, we will start with XML-RPC, which later evolved into SOAP.

XML-RPC

The **XML-RPC** standard was created by Dave Winer in 1998 with Microsoft. The existing RPC systems were very bulky. Therefore, to create a light-weight system, the developer simplified it by specifying only the essentials and defined only a handful of data types and commands. This protocol uses XML to encode its calls to HTTP as a transport mechanism. The message is sent as a POST request in which the body of the request is in XML. A procedure is executed on the server and the value it returns is also formatted into XML. The parameters can be scalars, numbers, strings, dates, as well as complex record and list structures.

As new functionalities were introduced, XML-RPC evolved into what is now known as SOAP, which is discussed next. Still, some people prefer using XML-RPC because of its simplicity, minimalism, and the ease of use.

SOAP

Initially, SOAP was defined as Simple Object Access Protocol. However, the latest version (SOAP 1.2) has moved beyond its original definition. The SOAP standard was originally designed by four developers with the backing of Microsoft as an object-access protocol. The protocol specifies exchange of XML-based messages over computer networks in transport independent manner. The developers had chosen XML as the standard message format because of its widespread use by major organizations and open source initiatives. Also, there is a wide variety of freely available tools that ease the transition to a SOAP-based implementation.

The concept of **SOAP** is a stateless, one-way message exchange. However, applications can create more complex interaction patterns—such as request-response, request-multiple responses, and so on—by combining such one-way exchanges with features provided by an underlying protocol and application-specific information. SOAP is silent on the semantics of any application-specific data it conveys as it is on issues such as routing of SOAP messages, reliable data transfer, firewall traversal, and so on. However, SOAP provides the framework by which application-specific information may be conveyed in an extensible manner.

Web Services Addressing (WS-Addressing)

It would have been quite useful if there was a standard way to express where a message should be delivered in a web services network. This could reduce the work load of the developers when they are able to simplify web services communication and development, and avoid the need to develop costly solutions ad hoc that are often difficult to interoperate across platforms. When interacting with human applications, we enter the address (or URL) in the browser and then navigate the page. We click on the internal links and it takes us to a new page. When it comes to application-application communication, such as web services, there should be a standard way of specifying those addresses. Thus, WS-Addressing enables organizations to build reliable and interoperable web service applications by defining a standard mechanism for identifying and exchanging Web Services messages between multiple end points.

The standard provides transport independent mechanisms to address messages and identifies web services, corresponding to the concepts of *address* and *message correlation* described in the web services architecture. The standard defines XML elements to identify web services endpoints and to secure end-to-end endpoint identification in messages. This enables messaging systems to support message transmission through networks that include processing nodes such as endpoint managers, firewalls, and gateways in a transport-neutral manner.

Service description

When we buy a product we can find the operational manual for it or when we buy a software application, we should have the manuals or documentation to use it. When it comes to OOP, we have interfaces that describe the public operation and in the same way Java docs provide information about the available methods and how to use them. Hence, it is important to note that the description of a web service is essential for classifying, discovering, and using a service. The description should be understandable to both humans as well as for applications. They further mention that the web service description is required to be at both semantic and syntactic level. Semantic information has to contain details about the service provider, what the service does, and characteristics such as reliability, security, and sequencing of messages. The semantic information enables service requesters to decide whether a service satisfies their needs or not. Also, brokers can use the semantic information to categorize the service. Syntactic information describes how to use the service and may also concern non-functional requirements, such as reliability, security, and transactions. Above all, service description becomes the external documentation to read and understand more about the service.

Web Services Description Language (WSDL)

WSDL, developed by IBM, Ariba, and Microsoft, is an XML-based language that provides a model for describing web services. The standard defines services as network endpoints or ports. WSDL is normally used in combination with SOAP and XML schema to provide web services over networks. A service requester who connects to a web service can read the WSDL to determine what functions are available in the web service. Special data types are embedded in the WSDL file in the form of XML Schema. The client can then use SOAP to call functions listed in the WSDL.

The standard enables one to separate the description of the abstract functionality offered by a service from the concrete details of a service description such as how and where that functionality is offered. This specification defines a language for describing the abstract functionality of a service as well as a framework for describing the concrete details of a service description. The abstract definition of ports and messages is separated from their concrete use, allowing the reuse of these definitions. A port is defined by associating a network address with a reusable binding and a collection of ports define a service. Messages are abstract descriptions of the data being exchanged and port types are abstract collections of supported operations. The concrete protocol and data format specifications for a particular port type constitutes a reusable binding where the messages and operations are then bound to a concrete network protocol and message format.

There are two main WSDL standards—WSDL 1.1 and WSDL 2.0. However, most of the Web Service frameworks available today still use WSDL 1.1, and a framework such as Apache Axis2 has support for both WSDL 1.1 and WSDL 2.0.

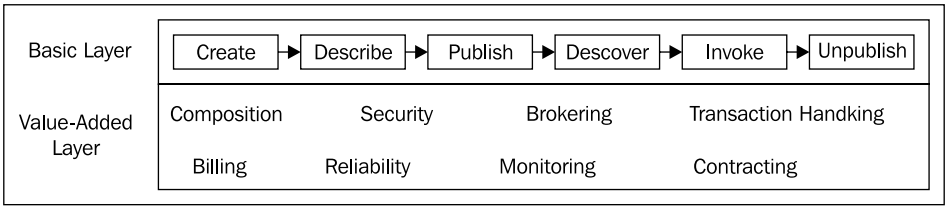
Web services lifecycle

As shown in the following figure, web services consist of a number of activities. These activities can be divided into two layers: the basic layer, which consists of the main activities that have to be supported by any web service, and the value-added layer, which brings value and enhances the performance of the web service.

- Create: The first activity in the service life cycle is the creation of the web service. This can be achieved either by building from scratch or by integrating existing web services.
- Describe: After creating the web service, it has to be described so that others can access it.
- Publish: After the description, it has to be published on the Web.

- Discover: Discovering a web service can be facilitated by a service broker, which will support requirement analysis and description of requester's need, matching needs to available web services, negotiation, and binding. As an alternative to discovery, often commercial agreements are made based on a supplied WSDL. This forms part of the contract between organizations.
- Invoke: Once the service is discovered, use tools and procedures to invoke the service.
- Unpublish: Finally, the service can be unpublished if it is no longer available or needed.

After the discovery is made and it is decided to use a certain web service, a number of activities related to contracting take place. During the lifetime of a web service, it will be updated and maintained throughout by the service provider. If the web service description is changed, this will be updated at the service broker's end.



Apart from these basic activities some value added activities need to take place for a web service to function effectively. Activities such as monitoring, billing, reliability, and security have to be implemented.

These web service activities can take place only at different sites, that is, some of these activities will take place at the service provider's site, while some will be at the service broker's site and the rest will be at the service requester's site. This does not mean that a particular site can only play one role; it can play multiple roles.

Apache Web Service stack

In this section, we are going to discuss web service frameworks, particularly Web Services frameworks at Apache. Increasing popularity of web services and their usage have built a new competition in the market. This competition has led to produce a set of good web service frameworks in the open source and commercial domains. In the open source domain, especially when it comes to web services, undoubtedly Apache has the most commonly used Web Service framework. Over the years, Apache has produced four main Web Service frameworks, and we are going to discuss three of them here. As far as the history of web services is concerned, several generations are clearly visible. The first generation web services were highly controlled interactions and can be considered as mere tests of feasibility. Apache SOAP was one of the notable SOAP engine in the first generation, and that was mainly meant to be "proof of concept" and not at all that concerned about performance. Apache SOAP was initially developed at IBM and later they donated it to Apache and started the Web Service project in Apache. The whole idea of the first generation SOAP engines was to convince the world that web services are feasible.

Soon the toll of these first generation SOAP engines paid off. More companies started showing their interest and the SOA started taking shape. This stage can be called as the second generation of web services and required better SOAP engines that were faster. Aspects such as discovery and definition were already standardized and SOAP engines also needed to support these standards. Apache Axis (Axis1) was born as one of these second generation SOAP engines.

Apache Axis project is one of the most successful projects in Apache, and it gained huge market awareness. A lot of companies use it as the core Web Service framework. Apache Axis project introduces a number of new concepts into Apache Web Services, and among them Handler framework can be considered as one of the most important and useful features. Handler framework enables us to extend core web service features and add additional quality of services, such as security and reliability. Axis uses DOM as internal XML (SOAP) representation mechanism, and it comes with comprehensive support for code generation and data binding. Axis project has better support to quality of service, including reliability, security, and transaction. Additionally, it has support for binary data. Axis has two implementations, one for Java and the other for C++. Axis project is still popular and very stable and a lot of companies use Axis as the web service framework.

Now the second generation of web services is also coming to an end. Web services are becoming highly demanding and a large number of players have entered the web service arena. Aspects governing different facets of web service interactions have been standardized. The third generation of web services requires faster, far more robust SOAP engines and the existing Axis is not enough for this. Axis2 was made to fill this gap.

Why Axis2?

As we discussed earlier, web services are moving rapidly and a lot of organizations have moved to web services field. As a result of this, new requirements are encountered and new standards are defined. Especially, the widespread use of Cloud computing also introduces a number of interesting challenges to the web service world. Most of the commonly used commercial Clouds (for example, Amazon, EC2) use a web service as the main management interface. Now people are looking for web services, which perform fast, have reliability, security, and transaction, and above all usability. In addition to the requirements, new WS-* specifications have defined and web service engine need to support those.

As we discussed in the previous section, Apache Axis1 is one of the most stable and commonly used web service frameworks. So changing Axis1 architecture to support new requirements and new Web service standards was not a feasible option. In the meantime, any software has its own lifecycle: it can evolve up to a certain point and after that revolution is needed. Same theory was applicable for Axis1 as well. As a result, Apache Web Service development project management team decided to start a fresh project to support those new standards and user requirements.

Today, we live in fast moving society where time is very critical for all of us. At a time when we start to improve time in nanosecond level, applications such as stock market millisecond make a huge difference. As a result, performance is a critical factor and having features and providing usability are not enough. Thus, in addition to the new requirements and WS-* specifications, performance was a major concern for Axis2. Axis1 uses DOM as its XML representation mechanisms. As a result of that, complete messages need to load into memory before starting to process, which slows down the system as well as increases the memory usage. Therefore, one of the key challenges was to improve the XML processing and from that improve the overall processing time and memory footprint. To provide better support and faster processing of XML, Axis2 introduced new XML processing framework known as Axiom. It uses pull parsing technology to achieve its requirements.

The Apache Web Service community discussed and agreed to introduce a new web service framework called **Axis2**, with a number of new requirements as well as a very flexible and easily extensible architecture, to support the current WS-* standards as well as for future standards. That is how Axis2 or the Apache third generation web service engine cames to the industry.

Apache Axis2 not only supports SOAP 1.1 and SOAP 1.2, but it also has integrated support for the widely popular REST-style of web services. The same business logic implementation can offer both a WS-* style interface as well as a REST/POX style interface simultaneously.

Apache Axis2 is more efficient, more modular, and more XML-oriented than the older version. It is carefully designed to support the easy addition of plugin "modules" that extend their functionality for features such as security and reliability:

- WS-ReliableMessaging: supported by Apache Sandesha2
- WS-Coordination and WS-Atomic Transaction: supported by Apache Kandula2
- WS-Security: Supported by Apache Rampart
- WS-Addressing: module included as part of Axis2 core

Axis2 comes with many new features, enhancements, and industry specification implementations:

- Speed: Axis2 uses its own object model and StAX (Streaming API for XML) parsing to achieve significantly greater speed than earlier versions of Apache Axis.
- AXIOM: Axis2 comes with its own lightweight object model, AXIOM, for message processing, which is extensible, performs highly, and is developer-convenient.
- Hot deployment: Axis2 is equipped with the capability of deploying web services and handlers while the system is up and running. In other words, new services can be added to the system without having to shut down the server. Simply drop the required web service archive into the services directory in the repository, and the deployment model will automatically deploy the service and make it available for use.
- Asynchronous web services: Axis2 now supports asynchronous web services and asynchronous web services invocation using non-blocking clients and transports.
- MEP support: Axis2 now comes handy with the flexibility to support **Message Exchange Patterns** (**MEPs**) with in-built support for basic MEPs defined in WSDL 2.0.
- Flexibility: The Axis2 architecture gives the developer complete freedom to insert extensions into the engine for custom header processing, system management, and anything else that you can imagine.
- Stability: Axis2 defines a set of published interfaces, which change relatively slowly, as compared to the rest of Axis.
- Component-oriented deployment: You can easily define reusable networks of Handlers to implement common patterns of processing for your applications or to distribute to partners.

- Transport framework: Axis2 has a clean and simple abstraction for integrating and using Transports (that is, senders and listeners for SOAP over various protocols such as SMTP, FTP, message-oriented middleware, and so on), and the core of the engine is completely transport-independent.
- WSDL support: Axis2 supports the Web Service Description Language (versions 1.1 and 2.0), which allows you to easily build stubs to access remote services, and also to automatically export machine-readable descriptions of your deployed services from Axis2.
- Add-ons: Several web service specifications have been incorporated, including WSS4J for security (Apache Rampart), Sandesha for reliable messaging. Kandula is an encapsulation of WS-Coordination, WS-AtomicTransaction, and WS-BusinessActivity.
- Composition and extensibility: Modules and phases improve support for composability and extensibility. Modules support composability and can also support new WS-* specifications in a simple and clean manner. They are, however, not hot deployable, as they change the overall behavior of the system.

Downloading and installing Apache Axis2

Apache Axis2 version 1.0 was released in 2006 and after that there were a number of releases. The current stable release is 1.5. The previous version of this book, "*Quickstart Apache Axis*", *Deepal Jayasinghe*, *Packt Publishing*, was based on version 1.2 of Axis2. This one is based on version 1.5. From 1.2 to 1.5, Axis2 introduced a set of new features as well as API changes. One of the key changes was moving from JDK 1.4 to 1.5. The current version of Axis2 only supports JDK 1.5 and higher. Nevertheless, most of the commonly used APIs still remain the same.

There are three types of software: free software, open source software, and commercial software. The main idea of free software is that you can download the software for free; however, you do not get access to the source code or the development work. On the other hand, open source software is designed and developed by an open community; anyone can participate in the discussions and contribute to the project, and finally, once the product is released, the user can have access to both the product and the source code. The user can modify the source code, fix issues, redistribute, and so on. As Axis2 is also an open source project, you can download Axis2 and get access to both binary and source codes. However, in the proprietary software, the license agreement is very restricted, and usually the user does not get to see the source code; the user only gets the binary.

You can download the latest version (or 1.5) from the Axis2 official website or any mirror site. The download link is shown here:

```
http://ws.apache.org/axis2/download/1_5_1/download.cgi
```

Once you go the download page, you can find four different distributions (artifacts):

- Binary distribution
- WAR distribution
- Source distribution
- Document distribution

In addition, you can also download IDE plugins from Axis2's official website (IDE tools include both Eclipse and IntelliJ IDEA).

Binary distribution

Axis2 binary distribution consists of all the relevant third-party libraries, a set of samples, and Axis2 runtime. Installing binary distribution is just a matter of extracting ZIP archive files into a location where you want. Once you download and extract the binary distribution, you will be able to see a set of subdirectories inside it (`bin`, `lib`, `samples`, `repository`, `webapp`, and `conf`). A typical structure of an extracted binary distribution is shown below:

Axis2 binary distribution is a complete package where you can deploy services and expose them using SimpleAxisServer. **SimpleAxisServer** is a fully functional server that can be used as the backend server to expose the web service. It supports all the features that the servlet version supports, including session management, thread management, auto WSDL generation, and others.

We can also use Axis2 binary distribution to invoke remote services. For this you need to add Axis2 and other related libraries into the class path and use Axis2 Client APIs to invoke the service. One of the commonly used approaches is to add those libraries into your IDE and client applications developed from that.

Binary distribution can also be used to generate Stubs and Skeleton from a given WSDL or to generate WSDL from a given Java class. In the later chapters, we will discuss how to use these tools to generate code.

Starting Axis2 as a standalone server is just a matter of running either `.bat` or a script file in the `bin` directory. Once we run the `axis2server.sh` (or `.bat`) and type `http://localhost:8080/axis2`, we can see the list of available services in the system, and these indicate the server is up and running.

WAR distribution

One of the easiest ways to expose a web service is to integrate it with an existing application or enable the web service through an available application server. In such scenarios, it is very useful to have a separate WAR (Web Archive) distribution, which helps the users to download, deploy, and access very easily. Assume that you have already downloaded Axis2 WAR distribution, and further assume you have Apache Tomcat running. To deploy Axis2, you need to copy the `axis2.war` file into the `webapps` directory. Next, if Tomcat is running on port 8080, you can access Axis2 by going to the following URL:

```
http://localhost:8080/axis2
```

If everything has gone well, you will get the following page:

Next, we can try to invoke the version service (a default service comes with Axis2 distribution) using the following URL:

```
http://localhost:8080/axis2/services/Version/getVersion
```

After this, it should display the version string of Axis2 you have downloaded. Once you see the version number, we have successfully deployed Axis2 into Tomcat. Next, you can upload or deploy a new service into Axis2.

```
<ns:getVersionResponse>
  <ns:return>Hi - the Axis2 version is SNAPSHOT</ns:return>
</ns:getVersionResponse>
```

In Axis2, to add a new web service, we need to add corresponding resources (service archive file) into the extracted location of the WAR file. However, most of the application servers do not unpack the WAR file. In those cases, you need to follow additional steps to configure your WAR file. For that, you need to extract the WAR file and do the required modifications to the `web.xml` file and recreate the WAR file and deploy. If your application server unpacks the WAR file, we can drop our new web service into the unpack location. This method is only used for quick testing—real development and production use. It is recommended to use Ant (or Maven) to produce WAR files for specific deployments, that is, not manually adjusting the contents of WAR files. We will learn about Axis2 Web Service in detail later in this book. There, we will be able to understand the meaning of adding a new web service.

Installing WAR distribution consists of the following few steps:

1. Install application server: If you do not have any application server in your machine, then you need to download and install the application server. Among the available application servers Apache Tomcat can be considered as one of the best application servers (does not support all the features of J2EE). So let's download Tomcat (5.x or above) and install.
2. Depending on the application server, you can find the location where you need to deploy the WAR files. If you take Tomcat as an example, you need to put the WAR file into the `webapps` directory. So let's drop Axis2 WAR distribution into the `webapps` directory of the application server.
3. As the final step, open a browser and type `http://localhost:8080/axis2`. We can see Axis2 web application (as shown in the previous figure) home page (here the URL may differ, depending on the application server).

Source distribution

As the name implies, source distribution consist of the source code that is used to build binary distribution. As Apache Axis2 is released under Apache license, we can do anything with the source code. The idea is that the user can use Axis2 and modify to suit their requirements. Additionally, if they think that a modification is useful for other people, they can submit a patch and ask the project community to merge the changes to the main source repository.

When we develop a real application, it is always useful to have the source code around in addition to the documentation that helps to debug the application as well. In the meantime, source distribution consists of Maven scripts (`http://maven.apache.org`) and we can use them to create either binary distribution, WAR distribution, or even JAR distribution.

Document distribution

The document distribution provides all the necessary resources to understand the different features and functionality in the project. It provides Java documentation and tutorials, explaining how to use different features,

JAR distribution

One of the key steps of a web service is developing the Services and Clients. In that stage, we need to have access to the web service framework and their libraries. To access the APIs and other relevant resources, it is a common practice to add the required libraries into the **Integrated Development Environment** (**IDE**). Axis2 JAR distribution provides a convenient way to download and embed Axis2 runtime into the IDE. You can download Axis2 library files separately from the following links, or you can get those from Axis2 binary or WAR distribution:

`http://people.apache.org/repo/m2-ibiblio-archived/org/apache/axis2/`

Summary

Is the web service concept a new revolutionary technology? The idea of splitting large programs into small modules is an established principle of higher-level programming languages. Even in the time of assembler programming, procedures were separated from the other program parts for a better reusability. In fact, the popularity of web services is caused by the standardisation of techniques and by the standards that are open enough to be adapted to different situations. So the developer is not committed to a concrete realisation of these concepts. There always exist more solutions to realise the concepts of web services.

In this chapter, we discussed the main ways of enabling the SOA functionally in your organization. There we discussed three main approaches—existing messaging system, plain old XML, and web services. We then looked at the standard bodies of web service and the web service model. Finally, we discussed Apache Axis2 and how to download and use it.

In the next chapter, we will discuss Axis2 architecture, key components, and key features.

2
Looking inside Axis2

There are a number of factors that make the process of understanding an application much easier. When it comes to middleware application such as Apache Axis2, having a good understanding of the underlying architecture, the key modules, and how they are connected helps to use the application with a lot of confidence. Hence, in this chapter, the main focus is to discuss Axis2 architecture and core components in the application. In addition, flexibility and extensibility are the two main design criteria that the software designers would like to have in their applications. Hence, Axis2 also features extensibility and flexibility.

In this chapter, we will discuss more about Axis2 architecture, core components, and how they are related. We will cover:

- Axis2 architecture
- Core modules
- Client API
- Extensibility

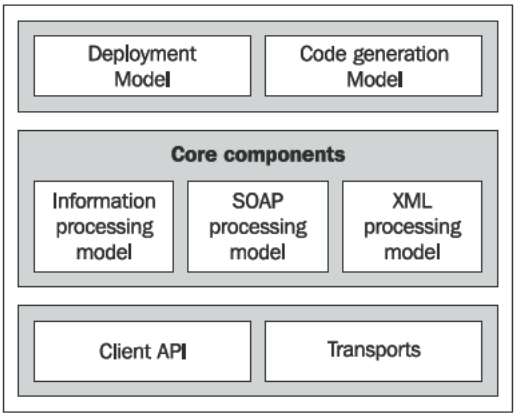
Axis2 architecture

Axis2 uses modular architecture and the advantages associated with it. In general, modular architecture allows any of the components to be upgraded or replaced independently as the requirements or the technologies change. Axis2 consists of a set of core as well as a set of non-core modules. In addition, the Axis2 core system is a pure SOAP-processing system with no JAX-PRC concept burnt into the core. Every message coming into the system has to be transformed into a SOAP message before it is handed over to the core engine. An incoming message can either be a SOAP message or a non-SOAP message (REST, JSON). But at the transport level, it will be converted into a SOAP message.

When Axis2 was designed, the following key rules were incorporated into its architecture. These rules were mainly applied to achieve a highly flexible and extensible SOAP processing engine:

- Separation of logic and state to provide stateless processing mechanism. This is because web services are stateless.
- Single information model, in order to enable the system to suspend and resume.
- Ability to extend support to newer web service specifications with minimal changes made to the core architecture.

The following figure shows all the key components in Axis2 architecture (including the core components as well as non-core components):



Core modules

The term **core** means to categorize key components of Axis2. The set of core modules consists of the minimal number of modules that need to set up an Axis2 system. In other words, once you have all the core modules (`.jar` files) and other dependency libraries, you can use Axis2 to deploy a service and to invoke a remote service. The following is the list of core modules in Axis2:

- XML processing model: One of the key components of any messaging system is the message processing and manipulating system. System usability and performance wholly depends on the efficiency of the message processing component. As mentioned earlier, Axis2 was built as a SOAP processing framework and SOAP is a specialized version of XML. Hence, any XML processing framework can be employed to process SOAP messages. Axis1 uses DOM as its message representation mechanism. However, Axis2 introduced a fresh, XML InfoSet-based representation for SOAP messages. It is known as **AXIs Object Model** (**AXIOM**). AXIOM encapsulates the complexities of the efficient XML processing within the implementation.
- SOAP processing model: As mentioned in the previous item, SOAP is a specialized version of XML. In other words, SOAP is XML, but comes with a standard structure. In Axis2, the SOAP processing module involves the processing of an incoming SOAP message. The model defines the different stages (phases) that the execution will walk through. The user can then extend the processing model to specific places.
- Information model: It is always a good idea to keep that data and logic separately. A classic example is DBMS (Database Management System), which does this. Thus, one can change without affecting the other. In Axis2 domain, an information model keeps both static as well as dynamic states. And there is a separate component to process the corresponding logic. The information model consists of two hierarchies to keep static and runtime information separate. Session management and Service lifecycle management are two core features of the information model.
- Deployment model: The deployment model allows the user to easily deploy the services, configure the transports, and extend the SOAP processing model. It also introduces newer deployment mechanisms in order to handle hot deployment, hot updates, and J2EE-style deployment.
- Client API: This provides a convenient API for users to interact with the web services using Axis2. The API consists of two sub-APIs, for average users and for advanced users. Axis2 default implementation supports all the eight **Message Exchange Patterns** (**MEP**) defined in WSDL 2.0. The API also allows easy extension to support custom MEPs.
- Transports: Axis2 defines a transport framework that allows the user to use and expose the same service in multiple transports. The transports fit into specific places in the SOAP processing model. The implementation, by default, provides a few common transports (HTTP, SMTP, JMS, TCP, and so on). However, the user can write or plug in custom transports, if needed.

XML processing model

As mentioned in *Chapter 1*, *Apache Web Services and Axis2*, Axis2 is built on a completely new architecture as compared to Axis 1.x. One of the key reasons for introducing Axis2 was to have a better and an efficient XML processing model. Axis 1.x used DOM as its XML representation mechanism, which required the complete objects hierarchy (corresponding to incoming messages) to be kept in the memory. This will not be a problem for small-sized messages. But when it comes to large-sized messages, it becomes an issue. To overcome this problem, Axis2 has introduced a new XML representation.

AXIOM (AXIs2 Object Model) forms the basis of the XML representation for every SOAP-based message in Axis2. The advantage of AXIOM over the other XML InfoSet representations is that it is based on pull parser technique, whereas most others are based on PUSH parser technique.

The main difference of pull over push is that in pull technique, the invoker has full control over the parser, and it can request for the next event and act upon that, whereas in the case of PUSH, the parser has limited control and delegates most of the functionality to handlers that respond to the events that are fired during the course of processing the document.

As AXIOM is based on pull parser technique, it has "on-demand-building" capability wherein object model will be built only if it is asked to do so. If required, one can directly access underline pull parser from AXIOM, and use that rather than build an **Object Model** (**OM**).

SOAP processing model

Sending, receiving, and processing a SOAP message is one of the key jobs of the SOAP-processing engine. The architecture in Axis2 provides two pipes (or flows) in order to perform two basic actions. Old messaging systems such as RPC were built around request and response. In Axis, it goes beyond that and introduces one-way message processing. With this, there can be request without any responses.

The `AxisEngine` or driver of Axis2 defines two methods, `send()` and `receive()`, to implement these two pipes. The two pipes are named **InFlow** and **OutFlow**. The complex Message Exchange Patterns (MEPs) are constructed by combining these two types of pipes. It should be noted that in addition to these two pipes, there are two other pipes as well, which help in handling incoming and sending *fault* messages.

As mentioned earlier, the key job of the SOAP processing framework is to process the SOAP message. However, just processing the message is not enough. It is necessary to provide additional functionalities, such as quality of service. Thus, it is required to have extensibility built onto the system. Extensibility of the SOAP processing model is provided through handlers. When a SOAP message is being processed, the handlers that are registered will be executed. The handlers can be registered in global, service, or in operation scopes, and the final handler chain is calculated by combining the handlers from all the scopes.

The handlers act as interceptors and they process the parts of the SOAP message and provide the *quality of service* features (a good example of quality of service could be security or reliability). Usually handlers work on the SOAP headers, yet they may access or change the SOAP body as well.

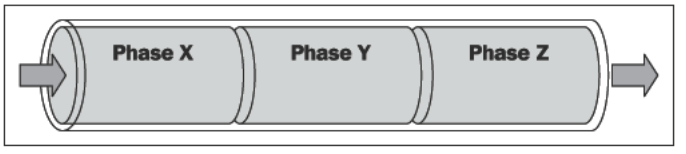
The concept of a flow is very simple and it consists a series of phases wherein a phase is referred as a collection of handlers. Depending on the MEP for a given method invocation, the number of flows associated with it may vary.

In-Only MEP: In the case of an In-Only MEP, the corresponding method invocation has only one pipe, that is, the message will only go through in-pipe (inflow).

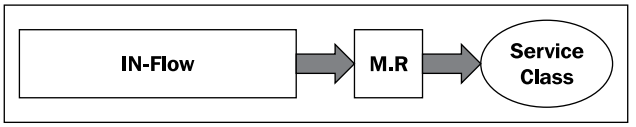
In-Out MEP: On the other hand, in the case of an In-Out MEP, the message will go through two pipes, that is, in-pipe (inflow) and out-pipe (outflow). When a SOAP message is being sent, an `OutFlow` would begin.

Fault-Flow: The `OutFlow` invokes the handlers and ends with a **transport sender** that sends the SOAP message to the target endpoint. The SOAP message is received by a **transport receiver** at the target endpoint, which reads the SOAP message and starts the `InFlow`. The `InFlow` consists of handlers and ends with the **message receiver**, which handles the actual business logic invocation.

A phase is a logical collection of one or more handlers, and sometimes a phase itself acts as a handler. Axis2 introduced the phase concept as an easy way of extending core functionalities. In Axis 1.x, we needed to change the global configuration files if we want to add a handler into a handler chain. But Axis2 makes it easier by using the concept of phases and phase rules. Phase rules specify how a given set of handlers inside a particular phase are ordered. The following figure illustrates a flow and its phases:



If the message has gone through the execution chain without having any problem, the engine will hand over the message to the message receiver in order to do the business logic invocation. After this, it is up to the message receiver to invoke the service and send the response if necessary. The following figure shows how the message receiver fits into the execution chain:



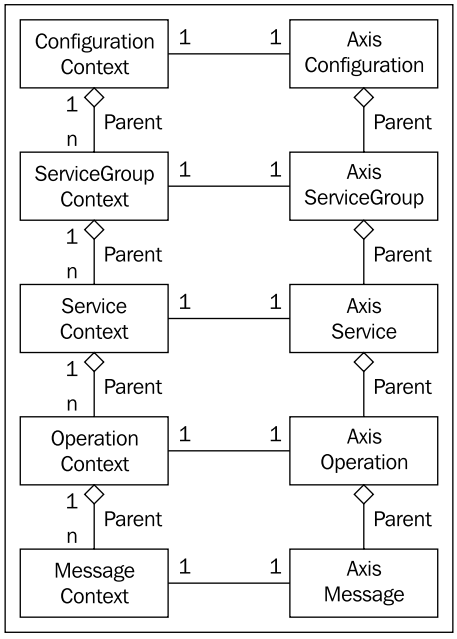
The two pipes do not differentiate between the server and the client. The SOAP processing model handles the complexity and provides two abstract pipes to the user. The different areas or stages of the pipes are named as **phases** in Axis2. A handler always runs inside a phase, and the phase provides a mechanism to specify the ordering of handlers. Both pipes have built-in phases, and both define the areas for *user phases*, which can be defined by the user as well.

Information model

As shown in following figure, the information model consists of two hierarchies—description hierarchy and context hierarchy. The **description** hierarchy represents the static data that may come from different deployment descriptors. If the hot deployment is turned off, the description hierarchy is not likely to be changed. If the hot deployment is turned on, you can deploy the service while the system is up and running. In this case, the description hierarchy is updated with the corresponding data changes made by the services. The **context** hierarchy keeps the runtime data. Unlike in the description hierarchy, the context hierarchy keeps on changing when the server starts receiving messages.

These two hierarchies create a model that provides the ability to search for the key-value pairs. When the values are to be searched at a given level, they are done while moving up the hierarchy until a match is found. In the resulting model, the lower levels override the values present in the upper levels. For example, when a value has been searched in the message context and is not found, then it would be searched in the operation context, and so on. The search is first done *up* the hierarchy, and if the starting point is a context, it would search in the description hierarchy as well.

This allows the user to declare and override values, resulting in a very flexible configuration model. The flexibility could be the Achilles' heel of the system, as the search is expensive, especially for something that does not exist.



We will discuss this figure in more detail in *Chapter 6*, *Information Model*, where we will discuss each item separately and also provide a detailed description of how they are connected.

Deployment model

The previous versions of Axis (Axis 1.x) addressed the requirements presented on development stages of the product. However, once the SOA and web service became popular, a number of new requirements came into the picture. Among them, usability is one of the most important factors to be considered.

In Axis 1.x the user has to manually invoke the admin client and update the server `classpath`. After this, the user needs to restart the server in order to apply the changes. This burdensome deployment model was a definite barrier for beginners. Axis2 was engineered to overcome this drawback, and provided a flexible, user-friendly, and easily configurable deployment model.

Axis2 deployment introduced the notion of J2EE-like deployment mechanism, wherein the developer can bundle all the class files, library files, resources files, and configuration files together as an archive file, and drop it in a specified location in the file system. In addition, Axis2 has also introduced the concept of deployer, where it provides an easy way of deploying anything Axis2 (Data Service, Mashups, and others).

The concept of hot deployment and hot update is not new to technical paradigm and particularly to the web service platform. However, having a very good support for hot-deployment is a new concept to the Axis community. Therefore, Axis2 was developed by giving room for *hot* deployment features.

- Hot deployment: This refers to the capability of deploying service while the system is up and running. In a real-time system or in a business environment, the availability of the system is very important. If the processing of the system is slow even for a moment, the loss might be substantial and it may affect the business. In the meanwhile, it is required to add new services to the system. If this can be done without having the need to shut down the servers, it would be a great achievement. Axis2 addresses this issue and provides web service hot deployment ability, wherein you do not need to shut down the system to deploy a new web service. All that needs to be done is to drop the required web service archive into the services directory in the repository. The deployment model will automatically deploy the service and make it available.
- Hot update: This refers to the ability to make changes to an existing web service without even shutting down the system. This is an essential feature that is best suited in a testing environment. It is not advisable to use hot updates in a real-time system, because a hot update could lead a system into an unknown state. Additionally, there is the possibility of loosening the existing service data of that service. To prevent this, Axis2 comes with the hot update parameter set to `FALSE` by default.

Client API

Nowadays, asynchronous or non-blocking web service invocation is a key requirement. A classic example is AJAX and Web 2.0 concept, where you have a number of asynchronous components in the web page. There are currently a few approaches to invoking a web service in a non-blocking manner. Two of them are listed here:

- The client programming model, where a client invokes the service in a non-blocking manner

- The transport level non-blocking invocation where invocation occurs in two transports (it could either be two single-channel transports, such as SMTP, or two double-channel transports such as HTTP)

Axis2 client API supports both non-blocking invocation scenarios.

Axis2 introduces a very convenient client API for invoking a service that consists of two classes called `ServiceClient` and `OperationClient`. `ServiceClient`. API is intended for regular usage when you just require sending and receiving XML. On the other hand, Operation client is meant for advanced usage, when there is a need to work with SOAP headers and some other advanced tasks. With `ServiceClient`, you can only access SOAP body or the pay load. Although we can add SOAP headers, you do not have any way to retrieve the SOAP header by using the `ServiceClient`. We need to use an operation client for such a function.

`ServiceClient` has the following API for invoking a service:

- **sendRobust**: The whole idea of this API is to send an XML request to the web service and not care about its response. However, if something goes wrong, you would be required to know that too. So this API invokes a service where it does not have a return value but would throw an exception.
- **fireAndForget**: This API is for just sending an XML request and not caring about neither the response, nor the exception. Hence, this is useful in invoking an In-Only MEP.
- **sendReceive**: This invokes a service that has a return value. This is one of the most commonly used APIs. Hence, this is used for invoking an In-Out MEP.
- **sendReceiveNonBlocking**: This invokes a service in a non-blocking manner. This method can be used when the service has a return value. In order to use this method, we have to pass a callback object that is called once the invocation is complete.

As mentioned earlier, the `OperationClient` is for advanced users, and working with `OperationClient` requires you to know Axis2 in depth. In `ServiceClient`, you do not have to know anything about SOAP envelope, message context, and others. But when it comes to `OperationClient`, the users have to create these by themselves before invoking a service. Creating and invoking a service using `OperationClient` involves the following steps:

1. Create a `ServiceClient`.
2. After that, create `OperationClient` with the use of `ServiceClient` that you have created.
3. Create a SOAP envelop.
4. Create message context.

5. Add the SOAP envelope to the message context.
6. Add the `MessageContext` to `OperationClient`.
7. Invoke the `OperationClient`.
8. If there is a response, get the response message context from the `OperationClient`.

Transports

In Axis2, each and every transport consists of two parts, namely transport senders and transport receivers. You can define transports along with senders and receivers in Axis2 global configuration. The transport receiver is the one through which the `AxisEngine` receives the message, whereas the transport sender is the one that sends out the message. One of the important aspects of Axis2 is that its core is completely independent of the transport sender and receiver.

Axis2 is built in order to support the following transport protocols:

- HTTP/HTTPS: In HTTP transport, the transport listener is a servlet or `org.apache.axis2.transport.http.SimpleHTTPServer` provided by Axis2. The transport sender uses a common HTTP client for connection and sends the SOAP message.
- TCP: This is the simplest transport and it needs WS-Addressing support in order to be functional.
- SMTP: This requires a single e-mail account. The transport receiver is a thread that checks for e-mails at fixed intervals of time.
- JMS: This provides a way to invoke a web service by using the JMS way.
- XMPP: This provides a standard way to communicate with the Jabber server and to invoke web services using XMPP protocol.

Other modules

The idea of other modules is optional modules that provide additional functionalities or utilities: for example, modules that need to generate Axis2 services and service proxies. Axis2 does not use these modules at the runtime, and as mentioned earlier, those are only to provide additional features. Axis2 uses them as tooling modules to generate service-side code and client-side code.

- Code generation: Axis2 provides a code generation tool that generates server-side (skeleton) and client-side code (stub or proxy) along with descriptors and a test case. The generated code would simplify the service deployment and the service invocation. This would increase the usability of Axis2.

- Pluggable data binding: The basic client API of Axis2 lets the user process SOAP at the XML InfoSet level, whereas data binding extends it to make it more convenient for the users by encapsulating the InfoSet layer and providing a programming language-specific interface.

Code generation

In the traditional RPC system, there is a concept called proxy and skeleton generation. The idea is to use the remote interface and generate proxy code to invoke the remote service, thus local application talks to the proxy and proxy internally calls to the remote service and gets the response. Java RMI is a good example for this. One of the issues the proxy is that it is fully application-dependent. Thus, to write a proxy, good knowledge about the application is required. With the code generation, it resolves that issue by providing tools to generate proxy.

In Axis2 world, the code generation is simply to use WSDL document and generate either (or both) Client proxy and Server skeleton. In Axis2, it uses a very extensible approach by combining XSL technologies with the code generation.

Data binding

Data binding for Axis2 is implemented in an interesting manner. Data binding has deliberately not been included in the core, and hence the code generation allows different data binding frameworks to be plugged in. This is done through an extension mechanism, where the codegen engine calls the extensions first and then executes the core emitter. The extensions plot a map of QNames versus class names, which is passed to the code generator wherein the emitter operates.

Axis2 supports the following data binding frameworks, including its own data binding framework called Axis2 Data Binding (ADB):

- **ADB:** This is a simple and lightweight framework that works off StAX and is fairly performant
- **XMLBeans:** If we want to use full schema support, this is preferred, as XMLBeans claims that it supports complete schema specification
- **JaxMe:** JaxMe support has been added to XMLBeans, and it serves as another option for the user
- **JibX:** This is the most recent addition to the family of data binding.
- **JaxBRI:** This provides additional Marshaller properties that are not defined by the JAXB specification

Extensible nature of Axis2

In Axis2, there are many ways to extend the functionalities. In this book, we will be discussing a few of them, which are listed here:

- Service extension of the module
- Custom deployers
- Message receivers

Service extension or the module

Both Axis1 and Axis2 have the concept of handlers. But when compared to Axis 1.x, there are few changes in the way Axis2 specifies and deploys handlers. In Axis 1.x, if you want to add a handler, then you need to change the global configuration file and then restart the system. In the meantime, it does not have a way to add or change handlers dynamically.

To overcome this problem as well as to add new features, Axis2 introduced the concept of web service extensions or a module where the main purpose of a module is to extend the core functionality. It is similar to adding handler chains in Axis1.x. The advantage of the Axis2 module over the Axis 1.x handler chain is that you can add new modules without changing any global configuration files.

A **module** is a self-contained package that includes handlers, third-party libraries, module-related resources, and a module configuration file.

A module can be deployed as an archive file. Axis2 came up with a new file extension for modules called `.mar`. The most important file in a module archive file is the module configuration file or `module.xml`. A module will not be functional unless it has a `module.xml` file. A module configuration file mainly specifies handlers and their phase rules. So once we engage a module depending on the phase rule, the handlers will be placed in different flows (inflow, outflow, and so on).

The idea of modules is very simple. To implement support for WS-Addressing or WS-Security in our services, we need to download the corresponding module and drop it into the modules directory of the Axis2 repository. We can engage the module at the time of deployment by adding `<module ref="module name"/>` to `axis2.xml` (global configuration file). In addition to that, if we want to engage a module at runtime, we can do that in many ways, such as by using Axis2 web admin, handlers, and so on.

Custom deployers

We can deploy a service in many ways. One could deploy a service as an archive file (Axis2 default) by creating a service using a database or by creating a web service using a text file. The idea of custom deployers is to open avenues for supporting any kind of deployment mechanisms. Axis2 has in-built support for:

- Archive-based deployment (`.aar` and `.mar` concept)
- POJO deployment (`.class` or `.jar`)

But if someone wants to deploy a service or a module, he or she can achieve that goal with the use of custom deployers. We will discuss custom deployers in more detail in *Chapter 5*, *Hacking Deployment*.

Message receivers

As we have discussed, the Axis2 execution chain is a collection of phases wherein each phase is a logical group of handlers. The message receiver is a handler in itself. But it is different from others because Axis2 treats it differently. If the message has gone through the inflow with no issues, or in other words, no exceptions have occurred in the middle of the chain, the engine hands over the message to the message receiver so as to invoke the associated business logic.

Message receivers interact directly with both the actual service implementation class and the `AxisEngine`. However, there can be some instances wherein there are no service implementation classes and all the logic is handed to the message receiver. The message receiver is the last component in the inflow process. Axis2 has got nothing to do with it once the message is handed over to the message receiver.

Summary

Axis2 is enterprise-ready. Web service engine provides a better SOAP processing model, with a considerable increase in performance of both speed as well as memory, with respect to Axis 1.x and other existing web service engines. In addition, it provides the user with a convenient API for deployment service, extending the core functionality, and thus acting as a new client programming model. In this chapter, we have learned about the internals and architecture of Axis2. We have learned how Axis2 architecture helps in attaining a more flexible and extensible Axis2.

In the next chapter, we cover Apache Axiom, the XML processing, and manipulation of the framework of Axis2. There we introduce Axiom and its features and discuss how to use them by giving code samples.

3
Axis 2 XML Model (AXIOM)

AXIOM stands for AXis2 Object Model and refers to the XML InfoSet model that was initially developed as part of Apache Axis2, but later it moved as a WS commons project so that projects other than Axis2 were able to benefit from it. XML InfoSet refers to the information included inside the XML, and for programmatic manipulation it is convenient to have a representation of this XML InfoSet in a language-specific manner. For an object-oriented language, the obvious choice is a model made up of objects. DOM and JDOM are two such XML models. AXIOM is conceptually similar to such an XML model by its external behavior, but deep down it is very much different. At the end of this chapter, you will understand the basics of AXIOM and the best practices to be followed while using AXIOM.

This chapter will cover:

- AXIOM architecture
- Pull parsing
- Working with AXIOM
- Advanced operations with AXIOM

Overview of AXIOM and its features

Apache Axiom was designed to provide fast and better XML, which became the heart of any given XML-processing system. Axiom is a lightweight implementation built on deferred parsing technology. More concretely, it is an implementation of StAX (JSR 173), the standard streaming pull parser API. One of the main intentions of this was to provide a standard manipulation API. Thus the AXIOM object model can be manipulated flexibly as any other object model (for example, JDOM), but underneath, the objects will be created only when they are absolutely required (that is, on-demand building). This leads to less memory-intensive programming.

Looking at the features of AXIOM, deferred building can be considered as one of the best. In addition, providing deferred building was also one of the design goals. If you look at Axis1, one of the drawbacks it had was its XML representation. In Axis1, it uses **Document Object Model** (**DOM**) as the XML representation, thus needing to load the complete message into the memory before the starting process. This leads to memory and performance overhead. AXIOM was introduced to solve those issues, and in addition to that, it has the following key features as well:

- **Lightweight**: Axiom was built using the previous experiences gained through the Apache Axis1 project. Hence, one of the design goals was to make it lightweight. This is achieved by reducing the depth of the hierarchy, number of methods, and the attributes enclosed in the objects. This makes the objects less memory intensive.
- **Deferred building**: This is one of the most important features that Axiom provides. This means building the object model only when it is absolutely needed. The objects are not made unless a need arises for them. This passes the control (of building) over to the object model, rather than an external builder. In other words, deferred building refers to the building the object model only when it is required.
- **Pull-based manipulation**: In deferred building, Axiom uses pull parsing APIs. As mentioned earlier, AXIOM is based on StAX, the standard pull parser API. To provide more control to the user, AXIOM exposes pull API, thus applications that use AXIOM can directly work with pull APIs.

What is pull parsing?

We encountered pull parsing several times in this chapter. It is essential to understand the meaning and the concept of pull parsing. Pull parsing is a popular technology for processing XML at the time Axiom was designed. In addition, pull parsing has shown success in several academic and industrial works. Before pull parsing, most of the XML processing frameworks such as DOM were push-based: in other words, populating the object structure completely depends on the underlying parser. There, first the user creates the parser and starts the process. Then the parser keeps on firing the events based on the elements (for example, `start` event for `start` element, `end` event for `end` element). With the push-based techniques, the user does not have control over the parser, and the user has to keep processing the events generated by the parser and act accordingly. Push-based approaches are easy to use; however, when it comes to applications that are dominated by XML processing, efficient mechanisms are needed. For example, push-based parsing require populating everything in the memory, but applications such as **Enterprise Service Bus** (**ESB**) only require to process a single part of the message and forward that to the appropriate target.

With pull parsing, the user has full control over the parser. The user can ask for the next event process and for the next event. In other words, in pull parsing, the parser only proceeds at the user's command. The user can decide to store or discard events generated from the parser. This is similar to a water tap where you open the tap, get the water, and close the tap. When you want the water again, you open the tap again and continue the process. Thus, the user has full control over the flow and process of the water. The same thing happens with pull parsing where the user creates the parser and asks for the events and then it gives the event. The user processes the events and asks for the next event, until it reaches the end of the file.

Architecture

When we consider the AXIOM architecture, it is not as complex as you may think. As mentioned earlier, many APIs are very similar to JDOM APIs. Notably, AXIOM has native support for binary data and MTOM, which is important for a SOAP processing framework such as Axis2. AXIOM also has full support for XML processing and enhanced support for SOAP processing.

AXIOM is also known as **Object Model** (**OM**), and the OM Builder wraps the raw XML character stream through the StAX reader API. Hence, the complexities of the pull event stream are transparent to the user.

As mentioned previously, deferred building is one of the main features of AXIOM. In addition, caching and non-caching can also be considered as another useful feature. The idea of caching depends upon whether the object model is stored in the memory or not. If the object model is stored in the memory, it is cached, otherwise it is non-cached. The reason why this is so important is because caching can be turned off in certain situations. If this happens, the parser proceeds without building the object structure. Users can extract raw pull stream from AXIOM and use that instead of the AXIOM. In this case, it is sometimes beneficial to switch off caching.

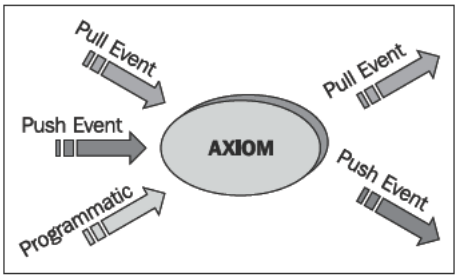
Working with AXIOM

You can either download AXIOM binary or you can build the binary using the source distribution (or from source repository). As you already know, though AXIOM was started as a part of Axis2, now it has its own release cycle. Therefore, you can either download AXIOM binary from AXIOM release or you can find AXIOM binary in the Axis2 release.

Once you have AXIOM binary, the next step is to add the binary into your `classpath` (and the dependent binary files as well); only then can you start to work with AXIOM. If your application has a build system such as Maven, you can add the dependency to that and let it download AXIOM JARs automatically.

Creating Axiom

You can create AXIOM (instance of object model) in three ways, as shown in the following figure. First, you can create Axiom using a pull event stream. Second, you can create Axiom using a push event stream, or you can create AXIOM programmatically. In this chapter, you will learn how to create Axiom using a pull event stream and also programmatically because these are the two most common methods you use to create Axiom.



First, let us look at how to create Axiom (we will be using AXIOM and Axiom interchangeably) using a pull event stream. Axiom provides a notion of a factory and a builder to create objects. The factory helps to keep the code at the interface level and the implementations separately. Axiom is tightly bound to StAX API (API for pull parser), thus a StAX-compliant reader should be created first with the desired input stream. One can then select one of the many builders available in AXIOM. In Axiom, you can find different types of builders as well, and those are mainly for user convenience. Axiom has OM builders (pure XML processing) as well as SOAP builders (SOAP processing optimized for SOAP), so you can use the appropriate builder for your requirement. **StAXOMBuilder** will build a pure XML InfoSet-compliant object model, while the **SOAPModelBuilder** returns SOAP-specific objects (such as the `SOAPEnvelope`, which are subclasses of the `OMElement`) through its builder methods.

Creating Axiom from an input stream

The following piece of code shows the correct method of creating an Axiom document from a file input stream (input stream can be file stream, socket stream or any other stream):

```
//First, create the parser
XMLStreamReader parser = XMLInputFactory.newInstance().
createXMLStreamReader(new FileInputStream(file));
//create an OM builder, the most recommend approach is to use the //
factory, but here we simply create it
```

```
StAXOMBuilder builder = new StAXOMBuilder(parser);
//get the root element (in this case the envelope)
OMElement documentElement = builder.getDocumentElement();
```

When you want to read an input stream using Axiom, the first step is to create a parser with the input stream (in this case, we create a `FileInputStream`). Next, you need to create a builder by giving parser as an argument. The builder uses parser internally to process the input stream. Finally, you can get the root element; remember, when you ask for the document element from the builder, it will give you the pointer to the Axiom wrapper. But the XML stream is still in the stream and no object tree is created at that time. The object tree is created only when you navigate or build the Axiom.

Creating Axiom using a string

Now let us try to create an Axiom document from a string, which is also very straightforward.

```
String xmlString = "<book>" +
                   "<name>Qucik-start Axis</name>" +
                   "<isbn>978-1-84719-286-8</isbn>" +
                   "</book>";
ByteArrayInputStream xmlStream = new
  ByteArrayInputStream(xmlString.getBytes());
  //create a builder. Since you want the XML as a plain XML, you can
  just use
  //the plain OMBuilder
StAXBuilder builder = new StAXOMBuilder(xmlStream);
  //return the root element.
builder.getDocumentElement();
```

As you can see here, when creating an Axiom from a string, first you get an input stream from that and then follow the same procedure stated here. By looking at the example, it is clear that creating an Axiom from an input stream or from a string is pretty straightforward. However, elements and nodes can also be created programmatically to modify the structure of the AXIOM element you created here. The recommended way to create Axiom objects programmatically is to use one of the factory APIs that come with AXIOM.

The `OMAbstractFactory.getOMFactory()` method will return the proper factory and the creator methods for each type that should be called.

Creating Axiom programmatically

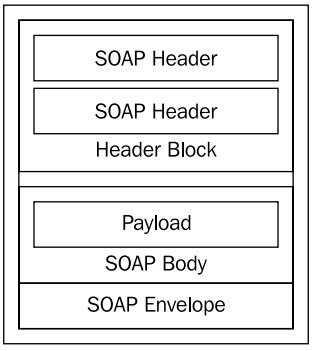
Creating an Axiom programmatically requires a number of additional steps compared to the mentioned approaches. The process of creating Axiom programmatically involves the following steps:

```
//Obtain a factory
OMFactory factory = OMAbstractFactory.getOMFactory();
//use the factory to create two namespace object
OMNamespace axis2 = factory.createOMNamespace("axis2","ns");
//use the factory to create three elements to represent the book
element
OMElement root = factory.createOMElement("book",axis2);
OMElement name = factory.createOMElement("name",axis2);
OMElement isbn = factory.createOMElement("isbn",axis2);
```

As you can see, the factory has a set of `factory.create` methods. This is mainly to cater to different implementations while keeping the programmer's code intact. When you use Axiom, it is always good practice to use the factory for creating Axiom objects. This will ease the switching of different Axiom implementations. Several differences exist between a programmatically created `OMNode` and a conventionally created `OMNode`. The most important difference is that the former will have no builder object enclosed, whereas the latter always carries a reference to its builder.

As we discussed earlier in this chapter, the object model is built as and when required. Therefore, each and every `OMNode` should have a reference to its builder. If this information is not available, it is due to the object created without a builder. This difference becomes evident when the user tries to get a non-caching pull parser from the `OMElement`.

The SOAP object hierarchy (see the following figure) is made in the most natural way for a web service programmer. An inspection of the API will show that it is quite close to the SAAJ API but with no bindings to DOM or any other model. The SOAP classes extend basic OM classes (such as `OMElement`); hence, one can access a SOAP document either with the abstraction of SOAP or drill down to the underlying XML object model with a simple casting.



Adding child nodes and attributes

So far you have learned how to create Axiom programmatically and by using StAX API, but it is not enough to work with Axiom. You also need to learn how to create and add child nodes to Axiom.

Addition and removal methods are primarily defined in the `OMElement` interface. The following are the important methods for adding nodes:

```
public void addChild(OMNode omNode);
public void addAttribute(OMAttribute omAttribute);
```

Now let us try to complete the book element you previously created by adding `child elements name` and `isbn` to the root element:

```
root.addChild(name);
root.addChild(isbn);
```

- The `addChild` method will always add the child as the last child of the parent.
- A given node can be removed from the tree by calling the `detach()` method. A node can also be removed from the tree by calling the remove method of the returned iterator, which will also call the detach method of the particular node internally.
- Namespaces are a tricky part of any XML object model and is the same in Axiom. However, the interface to the namespace has been made very simple. `OMNamespace` is the class that represents a namespace with intentionally removed setter methods. This makes the `OMNamespace` immutable and allows the underlying implementation to share the objects without any difficulty.

Working with OM namespaces

As we just discussed, namespace handling is one of the key parts of XML processing. Hence Axiom provides a set of APIs to handle namespaces:

```
public OMNamespace declareNamespace(String uri, String prefix);
public OMNamespace declareNamespace(OMNamespace namespace);
public OMNamespace findNamespace(String uri, String prefix) throws
OMException;
```

As you can see there are two `declareNamespace` methods and they are fairly straightforward. Note that a namespace declaration that has already been added will not be added twice. `findNamespace` is a very handy method to locate a namespace object higher up the object tree. It searches for a matching namespace in its own declarations section and jumps to the parent if it's not found. The search progresses up the tree until a matching namespace is found or the root has been reached.

During the serialization, a directly created namespace from the factory will only be added to the declarations when that prefix is encountered by the serializer.

You will learn how to serialize an Axiom element later in this chapter in the section *Serialization*. However, if you serialize the element you created, then you get the following output:

```
<ns:book xmlns:ns="axis2"><ns:name></ns:name><ns:isbn></ns:isbn></
ns:book>
```

Working with attribute

Let us now see how to create and add attributes to the book element:

```
OMAttribute type = factory.createOMAttribute("type",null,"web-
                   services");
root.addAttribute(type);
```

If you serialize the element again, then you will see following output:

```
<ns:book xmlns:ns="axis2" type="web-
    services"><ns:name></ns:name><ns:isbn></ns:isbn></ns:book>
```

Traversing the Axiom tree

In the previous sections, you learned how to create Axiom elements, to create and add child nodes, to create namespaces, and to create attributes. Now let us try to traverse the Axiom tree.

When traversing an Axiom object, you can use the conventional way of navigating the tree, where you get a node and look for its children, and then for each child node you do the same. Notably, in Axiom you do not get a list, rather you get an iterator: the idea is to encapsulate the deferred building by using an iterator. The following code sample shows how the children can be accessed. The children are of the type `OMNode` and can either be of type `OMText` or `OMElement`.

```
Iterator childNodes = root.getChildren();
while(childNodes.hasNext()){
  OMNode node = (OMNode) childNodes.next();
  }
```

This method shows how to access a list of children using an iterator; in addition to this, once you process a particular element, you can get its siblings as well. Moreover, you can access previous siblings and next siblings using `nextSibling()` and `previousSibling()` methods respectively.

You can also filter only a set of nodes (for example, elements with a certain name) using `getChildrenWithName(QName)` methods, where `getChildrenWithName(QName)` methods returns an iterator with all the child nodes, which have the given `Qname`.

Serialization

Now you have a good understanding of creating and traversing the Axiom tree. Therefore, the next most important part is to learn how to serialize or write Axiom object model into an output stream.

Axiom can be serialized either as the pure object model (for example, a set of Java objects) or the pull event stream (for example, writing to a file or any other output stream). The serialization uses an `XMLStreamWriter` object to write out the output and hence the same serialization mechanism can be used to write different types of outputs (such as text, binary, and others). Although not discussed in the book, the current version of Axis2 uses this and provides a number of different message formats (for example, REST, JSON, and so on).

In Axiom, it provides a way to navigate the tree with or without building the object structure. A caching flag is provided by Axiom to control this behavior. The `OMNode` has two methods: `serializeAndConsume` and `serialize`. When `serializeAndConsume` is called, the cache flag is reset and the serializer does not cache the stream. Hence, the object model will not be built if the cache flag is not set. In this case, it serializes the XML stream directory to the output stream (without creating object model). If you call the `serializeAndConsume` method, you can serialize the Axiom tree only once, as it does not build the Axiom tree into memory. However, you can call the serialize method any number of times. You will learn the differences between the two later in this section. The following code segment shows how to use the serializer and write output to the standard output:

```
//creating a writer to write into the standard output
XMLStreamWriter writer =
    XMLOutputFactory.newInstance().createXMLStreamWriter(System.out);
//write the content to the console with caching (i.e., build the //
tree)
  root.serialize(writer);
  writer.flush();
```

Now let us try to understand the difference between the methods `serializeAndConsume()` and `serialize()`. First, let us try to call the serialize method twice in the Axiom element, as shown in the following sample code:

```
String xmlStream = "<ns:book xmlns:ns=\"axis2\" type=\"web-
services\"><ns:name></ns:name><ns:isbn></ns:isbn></ns:book>";
        //Create an input stream for the string
        ByteArrayInputStream byteArrayInputStream = new
ByteArrayInputStream(xmlStream.getBytes());
        //create a builder. Since you want the XML as a plain XML, you
can just use
        //the plain OMBuilder
        StAXBuilder builder = new StAXOMBuilder(byteArrayInputStream);
        //return the root element.
        OMElement root = builder.getDocumentElement();
        root.serialize(System.out);
        root.serialize(System.out);
```

If you run this sample code, then you will see the following output in the console:

```
<ns:book xmlns:ns="axis2" type="web-services"><ns:name></
ns:name><ns:isbn></ns:isbn></ns:book>
<ns:book xmlns:ns="axis2" type="web-services"><ns:name></
ns:name><ns:isbn></ns:isbn></ns:book>
```

However, if you call `serializeAndConsume()` first and then call `serialize()`, you will get an exception. This is because once you have called the `serializeAndConsume()` method, the Axiom tree will not be built and the cache flag is reset. So the next time you try to call `serialize`, you have nothing left to serialize and you will get an exception.

```
String xmlStream = "<ns:book xmlns:ns=\"axis2\" type=\"web-
    services\"><ns:name></ns:name><ns:isbn></ns:isbn></ns:book>";
        //Create an input stream for the string
        ByteArrayInputStream byteArrayInputStream = new
    ByteArrayInputStream(xmlStream.getBytes());
        //create a builder. Since you want the XML as a plain XML, you
can just use
        //the plain OMBuilder
        StAXBuilder builder = new
    StAXOMBuilder(byteArrayInputStream);
        //return the root element.
        OMElement root = builder.getDocumentElement();
        root.serializeAndConsume(System.out);
        root.serialize(System.out);
```

Advanced operations with Axiom

Now you know how to create and serialize Axiom. However, that is not sufficient if you are going to do advance work with Axis2 or Axiom. There are a few more things that you need to learn:

- Using `OMNavigator` for traversing
- Xpath navigation
- Accessing pull parser
- Using SOAP support

Xpath navigation

Axiom has Xpath navigation support through Jaxen, so you can write Xpath query and invoke them in Axiom. Let us write an Xpath query to get the ISBN number from the book element you created. If you run the following sample, you will get `56789` as output in the console:

```
String xmlStream = "<book type=\"web-services\"><name></
name><isbn>56789</isbn></book>";
        ByteArrayInputStream byteArrayInputStream = new
ByteArrayInputStream(xmlStream.getBytes());
        StAXBuilder builder = new StAXOMBuilder(byteArrayInputStream);
```

```
            OMElement root = builder.getDocumentElement();
            AXIOMXPath xpath = new AXIOMXPath("/book/isbn[1]");
            OMElement selectedNode = (OMElement) xpath.
    selectSingleNode(root);
            System.out.println(selectedNode.getText());
```

Accessing the pull parser

As we discussed in the previous sections, Axiom is tightly integrated with pull parsing techniques. In other words, Axiom is an implementation of StAX APIs. To provide a high degree of control to the user, Axiom has exposed its internal pull parser to the user through the following two APIs:

- `getXMLStreamReader()`
- `getXMLStreamReaderWithoutCaching()`

In the first case, the user can get the parser, but when the user proceeds, it will automatically build the object structure. However, in the second case, Axiom will not build the object structure. So if the user misses some events, there is no way to get them back. This `XMLStreamReader` instance has a special capability of switching between the underlying stream and the Axiom object tree if the cache setting is off. However, this functionality is completely transparent to the user.

As we mentioned previously, Axiom uses cache flag to control the building of in-memory object tree. If the caching is on, then it should build the in-memory object structure. On the other hand if the caching is off, it should not build the in-memory data structure. As mentioned previously, this behavior is very useful in applications such as message proxies or ESB2 because those applications only need to read a certain part and building the object structure is not needed for forwarding the message .

Axiom and SOAP

As discussed, Axiom was developed as an XML representation mechanism in Axis2. Therefore, as a SOAP-processing framework, Axis2 needs to work with SOAP. You know that SOAP is also XML, but it has its own structure to be a special type of XML. So it is easy if you can get SOAP level API from Axiom. Therefore, Axiom has an in-built support for SOAP representation and navigation. You can create SOAP 1.1 and 1.2 documents easily with Axiom and navigate them. When you navigate SOAP, Axiom has API to get the headers and body. Therefore, you do not need to get an iterator and navigate. The following two samples show how to create SOAP 1.1 and 1.2 documents easily with Axiom.

Creating a SOAP 1.1 document

As you can see in the following code, first you create a book element as the body of the SOAP message and then you create a default SOAP 1.1 envelope and add the created book element as the body:

```
OMFactory factory = OMAbstractFactory.getOMFactory();
        OMNamespace axis2 = factory.createOMNamespace("axis2", "ns");
        OMElement root = factory.createOMElement("book", axis2);
        OMAttribute type = factory.createOMAttribute("type",null,"web-
services");
        root.addAttribute(type);
        OMElement name = factory.createOMElement("name", axis2);
        OMElement isbn = factory.createOMElement("isbn", axis2);
        root.addChild(name);
        root.addChild(isbn);
        SOAPFactory soapFactory = OMAbstractFactory.
getSOAP11Factory();
        //get the default envelope
        SOAPEnvelope env = soapFactory.getDefaultEnvelope();
        //add the created child
        env.getBody().addChild(root);
        System.out.println( env);
```

As you can see, first you create book element as the body of the SOAP message and then you create default SOAP 1.1 envelope and add the created book element as the body.

Creating a SOAP 1.2 document

Creating a SOAP 1.2 document is almost the same as in the previous example, except for the factory; here you need to use 1.2 factories instead of 1.1 factories:

```
OMFactory factory = OMAbstractFactory.getOMFactory();
        OMNamespace axis2 = factory.createOMNamespace("axis2", "ns");
        OMElement root = factory.createOMElement("book", axis2);
        OMAttribute type = factory.createOMAttribute("type",null,"web-
services");
        root.addAttribute(type);
        OMElement name = factory.createOMElement("name", axis2);
        OMElement isbn = factory.createOMElement("isbn", axis2);
        root.addChild(name);
        root.addChild(isbn);
        SOAPFactory soapFactory = OMAbstractFactory.
getSOAP12Factory();
        //get the default envelope
        SOAPEnvelope env = soapFactory.getDefaultEnvelope();
        //add the created child
        env.getBody().addChild(root);
        System.out.println( env);
```

Summary

In this chapter, we discussed a little history of Axiom, why it was introduced, and the standard specifications it was built on. We then discussed some of the features it provides and demonstrated them with code samples. Although we did not explicitly mention it, Axiom was initially designed only to handle SOAP, but later it was improved to handle any type of XML messages. Thus, currently Axiom is a complete XML-processing framework, which you can use outside Axis2.

In the next chapter, we will discuss one of the fundamental features of Axis2—the execution framework. There we will discuss the handler chain, handler, phase, phase rules, and how everything provides extensible flavor to Axis2.

4
Execution Chain

The key functionality of any given web service framework is processing an incoming message and delivering it to the targeted application (service), and once the result is produced, delivering it to the client. Today, for business critical applications, we need a lot more than just processing the messages. Thus, support, for example reliability, security, transactions, and throttling is required. In addition, a framework should also be easily extensible to support new customer requirements and other quality services. To provide those features, most of the SOAP processing frameworks utilize the concept of pipes, where any incoming or outgoing message goes through the pipe and the extensible pipe is divided into small pieces. Such a piece is known as an **interceptor**. One can add new interceptors, change them, or delete them, to cope with the requirements.

In this chapter, we will discuss the following topics in more detail:

- Handlers
- Phase
- Phase rules
- Concept of flows
- Some of the default handlers that come with Axis2 distribution

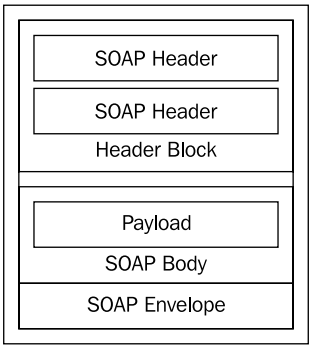
Handler

A handler is one of the most important and useful features introduced by the Axis1 project. In fact, the idea of a handler has been used in the industry for a long time. Some refer to handler as the message interceptor. In any messaging system, the interceptor has its factual meaning in the context of messaging too, which intercepts the messaging flow and does whatever task it is assigned to do. In fact, the interceptor is the smallest execution unit in a messaging system so that, as the interceptor in Axis, handler does the same thing.

Handlers, in Axis, are *stateless*, meaning they do not keep their past execution states in memory. A handler can be considered as a logic invoker with the input for the logic evaluation taken only from the `MessageContext`. Handler has both read and write access to **MessageContext** (**MC**) and to incoming SOAP messages. Thus, a handler can read SOAP messages, remove elements from the message (mostly headers), add new elements (headers), or modify elements as well as add, delete, or modify content from the `MessageContext`.

For continuation purposes, we can consider `MessageContext` as a property bag that keeps incoming or outgoing messages and other (maybe both) required parameters and properties to carry the message through the execution chain. On the other hand, via the MC, we can access the whole system, such as system runtime, global parameters, properties services, and operations. We will discuss more operations associated with MC in *Chapter 6*, *Information Model*.

In most of the cases, a handler only touches the header block of the SOAP message, which will read a header (or headers), add header(s), or remove header(s). However, it does not mean that the handler cannot touch the SOAP body. There are situations where handlers also process the body; a good example is encryption. In the process of reading, if a header is targeted to a handler and if it cannot execute properly (message might be faulty), it should throw an exception and the next chain driver (in Axis2, it is engine) would take the necessary action. A typical SOAP message with some headers is shown in the following figure:



Any handler in Axis2 has the capability to pause the message execution, meaning that the handler can stop the message flow if it cannot continue. **Reliable Messaging** (**RM**) is a good example, or a use case for that scenario, that needs to pause the flow depending on some pre and post conditions. In the case of RM, it works on the message sequence. If a service invocation consists of more than one message, and if the second one comes before the first, RM handler will stop (rather pause) the execution of the message invocation corresponding to the second message until it gets the first message. When it gets the first message, RM will invoke that, and then after that, it will invoke (or resume) the second message.

Writing a simple handler

To understand the concept better, you need to put them into practice. Writing a handler in Axis2 is very simple. The only thing we need to remember is that it has to either extend from `AbstractHandler` or implement the `Handler` interface. A simple handler that extends the `AbstractHandler` will look as follows:

```
public class SimpleHandler extends AbstractHandler {

    public SimpleHandler() {
    }

    public InvocationResponse invoke(MessageContext msgContext) throws
AxisFault {
        //Write the processing logic here
        // DO something
        return InvocationResponse.CONTINUE;
    }
}
```

One thing to note here is the return value of the `invoke` method, which determines the continuation of the message flow. We can have three values (described as follows) as the return values of the `invoke` method:

- **Continue**: If you, as a handler, think that the message is ready to forward
- **Suspend**: A handler thinks that the message cannot be sent forward since some conditions are not satisfied yet, so suspend the execution
- **Abort**: A handler thinks that there is something wrong with the message therefore it cannot allow the message to go forward

In most of the cases, handlers will return `InvocationResponse.CONTINUE` as the return value.

When a message is received by the Axis engine, it will call the `invoke` methods of each handler by passing argument as the corresponding `MessageContext`. As a result, you can implement all the processing logic inside that method. A handler author has full access to a SOAP message and all the required properties to process the message via the `MessageContext`. In addition to that, if the handler finds some pre-condition and is not satisfied with the invocation, the invocation can be paused, as mentioned earlier (`Suspend`).

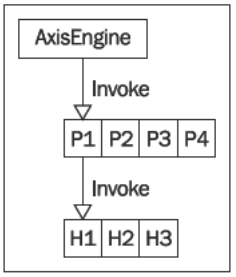
If some handler suspends the execution, it is its responsibility to store the message context, and when the conditions are satisfied, forward the message. As an example, RM handler performs the exact same scenario.

Phase

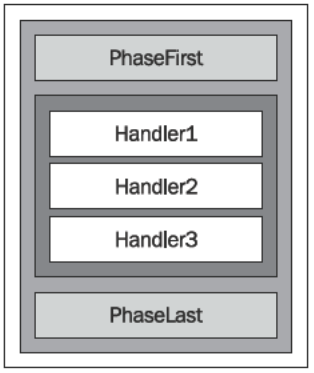
The concept of a **phase** is introduced by Axis2 and it was mainly to support the dynamic ordering of handlers to provide better extensibility, and better flexibility of the handler chain. A phase can be defined in various ways:

- It can be considered as a logical collection of handlers
- It can be considered as a specific time interval in the message execution
- It can be considered as a bucket into which one can put his/her handler
- One can consider a phase as a handler too

There is another term called flow (pipe). A **flow** is the message pipe where the message enters from one end of the flow and leaves from the other end of the flow. A flow or execution chain can be considered as a collection of phases. Although it has been mentioned earlier that the Axis engine will call the `invoke` method of a handler. That is not totally correct. At a very high level, what the engine really does is call the `invoke` method of each phase in a given flow, and then the phase will sequentially invoke all the handlers in it (the following figure illustrates how an engine calls a phase and then a phase calls handlers). As you know, you can extend `AbstractHandler` and create a new handler. In the same way, one can extend the `Phase` class and create a new phase too. But remember, you do not always need to extend the `Phase` class to create a new phase; you can do that by just adding an entry into `axis2.xml` (adding a phase into `axis2.xml` is described in *Chapter 6*, *Information Model*). A phase has two important methods—pre-condition checking and post-condition checking. Therefore, if you are writing a custom phase, those are the two methods you need to consider. However, writing a phase is a very rare scenario (user requirements can be mostly satisfied using default phases provided by Axis2); it is handlers that are important.



A phase is designed to support phase rules. Phase rules tell Axis2 where to put handles in the execution chain. As we will see later, there are rules like `phaseFirst` and `phaseLast`. So in a `Phase`, there are reserved slots to hold both the phase first and phase last handlers. The rest of the handlers will be kept in a different list. A phase can be graphically represented as follows:



Once the engine calls the `invoke` method of the phase, it has the following exception sequence:

1. First check whether the precondition is satisfied.
    - If not, throw an exception and stop the message from processing
2. Then check whether the phase first handler is there. If it is, invoke it.
3. Next, invoke the rest of the handlers, excluding the phase last handler.
4. If the phase last handler is there, then invoke it.
5. Finally, it will check whether the post condition is satisfied to forward the message.
    - If not, stop the execution and throw an exception

Types of phases

There are two types of phases defined in `axis2.xml`. However, they are not different in terms of the way they are implemented. The only difference is a semantic one, based on the location of the message flow; one type of phase gets executed for all the messages and another type does not. The following are the two types of phases that you can find in Axis2:

- Global phase
- Operation phase

Global phases

Global phases are the phases that are invoked irrespective of the service. In simple terms, whenever a message comes into the system, it will go through the global phases. The whole idea of defining a set of global and operation phases in `axis2.xml` is to make it easy for the module authors. As we shall see in *Chapter 8*, *Writing an Axis2 Module*, the module's author will create his module with the module descriptor file, and that module descriptor will make assumptions about the phases defined in the `axis2.xml`.

If we consider the default `axis2.xml`, which we can find inside our download directory, it has a set of global and operation phases. The splitting point of the global and operation phases is the **dispatch** phase. All the phases up to the dispatch phase (including the dispatch phase) are considered as global phases and the rest are considered as operation phases.

The phase section of the default `axis2.xml` is as follows. It is a bit complicated, but just focus on the phase keyword only.

```
<phaseOrder type="InFlow">
        <!--  System predefined phases       -->
        <phase name="Transport">
            <handler name="RequestURIBasedDispatcher"
                     class="org.apache.axis2.engine.
RequestURIBasedDispatcher">
                <order phase="Transport"/>
            </handler>
            <handler name="SOAPActionBasedDispatcher"
                     class="org.apache.axis2.engine.
SOAPActionBasedDispatcher">
                <order phase="Transport"/>
            </handler>
        </phase>
        <phase name="Security"/>
```

```
        <phase name="PreDispatch"/>
        <phase name="Dispatch" class="org.apache.axis2.engine.
DispatchPhase">
            <handler name="RequestURIBasedDispatcher"
                     class="org.apache.axis2.engine.
RequestURIBasedDispatcher"/>

            <handler name="SOAPActionBasedDispatcher"
                     class="org.apache.axis2.engine.
SOAPActionBasedDispatcher"/>

            <handler name="AddressingBasedDispatcher"
                     class="org.apache.axis2.engine.
AddressingBasedDispatcher"/>
            <handler name="RequestURIOperationDispatcher"
                     class="org.apache.axis2.engine.
RequestURIOperationDispatcher"/>

            <handler name="SOAPMessageBodyBasedDispatcher"
                     class="org.apache.axis2.engine.
SOAPMessageBodyBasedDispatcher"/>

            <handler name="HTTPLocationBasedDispatcher"
                     class="org.apache.axis2.engine.
HTTPLocationBasedDispatcher"/>
        </phase>
        <!--  System predefined phases       -->
        <!--   After Postdispatch phase module author or service
author can add any phase he want      -->
        <phase name="OperationInPhase"/>
        <phase name="soapmonitorPhase"/>
    </phaseOrder>
```

According to this XML segment, there are four global phases:

- Transport
- Security
- PreDispatch
- Dispatch

`OperationInPhases` and `soapmonitorPhase` are in the operation-specific phases.

All the global phases have semantic meanings from their names as well:

- Transport phase: This consists of a handler that performs tasks that depend on the type of transport.
- Security phase: WS-security implementation will add their handlers, but they are not limited to it, and any other modules or users can also add handlers.
- PreDispatch: As the name implies, this has a set of handlers that perform tasks that are needed for dispatching. Handlers, like WS-addressing, will always be in this phase.
- Dispatch phase: This is the phase that does the dispatching by simply finding the corresponding service and the operation for the incoming message. Therefore, the dispatch phase consists of dispatching handlers. We can also add new global phases and then handlers into it, which requires modification of the `axis2.xml` file.

  Note that if necessary the user can easily add new phases by editing `axis2.xml`.

Operation phases

Say, for instance, we have a handler and we do not need to run that for every message coming to the system, but we need to run that for selected operations. This is where the operation phase comes into the picture. Operational phases are the phases that come after the dispatch phase, and the user can add any number of phases. Adding operation phases does not change the structure of the other operations in their chains.

Phase rules

The main idea of phase rules is to correctly locate a handler relative to the one inside a phase, maybe at the deployment time or at the runtime. Axis1 did not have the concept of phases or phase rules. What it had was a global configuration file where you go and define you handlers. But that had a number of limitations; in particular, you lose the dynamic nature of the handler chain. Therefore, one aspect of phase rules is to address the issues of dynamic execution chain building capability.

Characterizing a phase rule

Characterizing a phase rule can be based on one or more of the following properties:

- Phase name: Name of the phase that the handler must be placed in
- First phase (`phaseFirst`): The first handler of the phase

- Last phase(`phaseLast`): The last handler of the phase
- Before (`before`): Positions the handler before another handler
- After (`after`): Positions the handler after another handler
- Before and after: Places the handler between two handlers

Phase name

`phase` is a compulsory attribute for any phase rule which gives the name of the phase in which the handler must fit. In order for the rule to be a valid phase name, it should be known to the system, which must be either a global phase name or an operation specific phase.

phaseFirst

As the name implies, if you want a handler to be invoked as the first handler in a given phase, irrespective of other handlers in the phase, you have to set `phaseFirst` attribute to `true`. A handler which is having the phase rule only with `phaseFirst` and a phase looks as follows:

```
<handler name="simple_Handler " class="org.apache.axis.handlers.
SimpleHandler ">
            <order phase="userphase1" phaseFirst="true"/>
</handler>
```

phaseLast

Like `phaseFirst`, if one wants the handler to be run last in a given phase, irrespective of other handlers, one has to set `phaseLast` to `true`. So the handler with `phaseLast` will look as follows:

```
<handler name="simple_Handler " class="org.apache.axis.handlers.
SimpleHandler ">
            <order phase="userphase1" phaseLast="true"/>
</handler>
```

If there is a phase rule with both `phaseFirst` and `phaseLast` set to `true`, then that phase cannot have any more handlers. In other words, the phase has only one handler.

before

There may be situations when a handler should always run before some other handler, no matter what the exact location is. A real-time use case for this can be the security handler that wants to run before the RM handler. The logic has to be written as follows and the value of the `before` attribute is the referred handler name:

```
<handler name="simple_Handler2 " class="org.apache.axis.handlers.
SimpleHandler2 ">
            <order phase="userphase1" before=" simple_Handler "/>
</handler>
```

The Axis2 phase rule processing logic is implemented in such a way that if the handler referred by the `before` attribute is not available in the phase at the time rule is been processed, it just ignores the rule and places the handler immediately after the `phaseFirst` handler (if it is available, it will place the handler somewhere in the phase).

after

Like `before`, if a handler always runs after some other handler, phase rule can be written using the `after` attribute and it should look like the example that follows. The value of the `after` attribute is the referred handler name:

```
<handler name="simple_Handler3 " class="org.apache.axis.handlers.
SimpleHandler3 ">
            <order phase="userphase1" after=" simple_Handler2"/>
</handler>
```

after and before

If a handler needs to be run in between two different handlers, the phase rule can be written using both `before` and `after` attribute. The values of both `before` and `after` attributes are the names of referred handlers. The correct way of writing a phase rule will be as follows:

```
<handler name="simple_Handler4" class="org.apache.axis.handlers.
SimpleHandler4">
            <order phase="userphase1" after=" simple_Handler1"
before=" simple_Handler2"/>
</handler>
```

Invalid phase rules

Validity of a phase rule is an important factor in Axis2. There can be many ways to get the same handler order by using different kinds of phase rules. However, while writing a phase rule, it is required to check whether the rule is a valid rule. There may be many ways in which a phase rule becomes an invalid rule. Some of them are as follows:

- If there is a phase rule of a handler with either `phaseFirst` or `phaseLast` attributes set to `true`, then the handler can have neither before nor after appearing in the phase rule. If they do, then the rule is invalid.

- If there is a phase rule of a handler with both `phaseFirst` and `phaseLast` set to `true`, then that particular phase cannot have more than one handler. If someone tries to write a phase rule that inserts a handler into the same phase, then the second phase rule is invalid.
- There cannot be two handlers in one phase with their `phaseFirst` attribute set to `true`.
- There cannot be two handlers in one phase with their `phaseLast` attribute set to `true`.
- If the rule is such that the `before` attribute is referred to as the `phaseFirst` handler, then the rule is invalid.
- If the rule is such that the `after` attribute is referred to as the `phaseLast` handler, then the rule is invalid.

```
<handler name="simple_HandlerError " class="org.apache.axis.
handlers.SimpleHandlerError ">
            <order phase="userphase1" before=" simple_Handler"
phaseFirst="true"/>
</handler>
```

Phase rules are defined per basic handler, and any handler in the system must fit into a phase in the system.

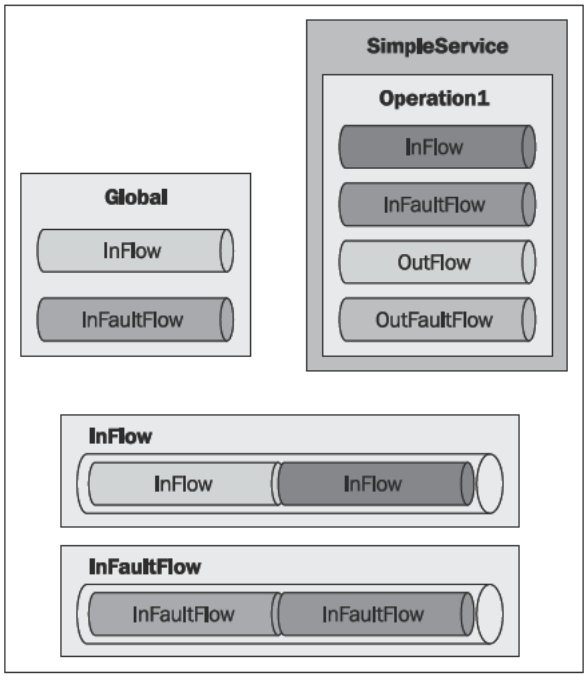
Flow

Flow is simply a collection of phases, and the order of phases inside a flow is defined in `axis2.xml`. As a phase is a logical collection and, which is, in fact, a virtual concept, a flow can be assumed as the execution chain (a collection of handlers). There are four types of flows in Axis2:

- `InFlow`: When a message comes in (request message), the message has to go via the `InFlow`. Then all the handlers in the `InFlow` will be invoked. `InFlow` is somewhat different from the `OutFlow`. A flow consists of two parts. The first part is from the beginning to the dispatcher (up to and including the dispatch phase). The second part will be there only if a corresponding service is found at the end of the dispatch phase. Therefore, the second part of the flow is `InFlow` of the corresponding operation for the incoming message. So the `InFlow` consists of a global part and an operation part.
- `InFaultFlow`: This flow will be invoked if the incoming request is faulty (request with HTTP status code 500).
- `OutFlow`: When a message is moving out from the server (say a response), this is invoked. As a result, the outgoing message is always bound to an operation, and there is nothing similar to dispatching in the out path.

- `OutFaultFlow`: If something goes wrong in the out path, then this will be invoked.

The following figure shows different flows and how they combine at runtime. For example, the lower part of the figure shows how global flow and operational flow are combined to complete the flow.



Module engagement and dynamic execution chain

In *Chapter 8*, *Writing an Axis2 Module*, we will learn that an Axis2 module can be considered as a collection of handlers. So by engaging a module either globally to a service or to an operation, handlers will be placed in the corresponding phases, depending on the phase rules. If a module is engaged globally, all the services in the system will be affected, and there is a probability of changing both global and operations flows (every operation in all the services). If a module is engaged to a service, flows belonging to all the operations in that particular service will be changed. If a module is engaged to an operation, every flow in that operation may be changed.

The only way of changing a flow is by adding a handler(s) to a phase in the flow. Therefore, module engagement will cause a change to the handler chain dynamically. A module can be engaged dynamically (at runtime) or statically (at deployment time). If you want to engage a module statically, you need to specify it in the description files. If it is to be a service or to be an operation, then do so in `services.xml`. If it is to be engaged globally, then in `axis2.xml`.

Special handlers in the chain

When you consider the execution chain, you can find four types of special handlers in the execution chain:

- Transport receiver
- Dispatcher(s)
- Message receiver
- Transport sender

Transport receiver

Whenever a message comes into a system first, it will reach a transport receiver. A transport receiver can be considered as something that is waiting to accept an incoming message (in the case of the application server, a transport receiver could be `Servlet`). Therefore, `InFlow` of an execution chain always starts with the transport receiver.

Dispatchers

As we discussed earlier, one of the fundamental goals of a SOAP processing framework is to deliver an incoming message to the targeted application. The process of finding the correct targeted application is called **dispatching**. In Axis2, dispatching will be taking place in the middle of the incoming execution chain. So, dispatchers are handlers in the chain. In Axis2, there are a number of ways to perform the dispatching:

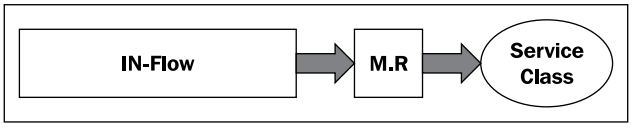
- Using transport headers and transport level data
- Using WS-Addressing information
- Using an incoming SOAP message

To cater to the mentioned types of dispatching, Axis2 has a set of default dispatchers, and you can change the order of their execution using `axis2.xml`. If you look at the `InFlow` element, which we have discussed in the *Global phases* section, you can find the set of available dispatchers in the dispatch phase.

- `RequestURIBasedDispatcher`: Try to find a service and operation using transport URI
- `SOAPActionBasedDispatcher`: Try to find the operation using the SOAP action
- `AddressingBasedDispatcher`: Uses addressing information in the WS-A header to find the service and operation
- `SOAPMessageBodyBasedDispatcher`: Uses and navigates the SOAP message, especially the body to find the operation.
- `HTTPLocationBasedDispatcher`: This will be used to dispatch WSDL 2.0-related SOAP messages

Message receiver

The message receiver is a handler in itself, but the only difference is that Axis2 treats this handler differently than others. If the message has gone through the execution chain without having any problem (no exceptions have occurred in the middle of the chain), the engine will hand over the message to the message receiver to do the business logic invocation. The following figure shows the location of the `MessageReceiver` in the execution chain:



On the other hand, the message receiver is the one who directly interacts with both the actual service implementation class and Axis Engine (there may be instances where a service would be a message receiver). Axis 1.x has the concept of **pivot point**, where the request path and response path are met together and where actual service invocation takes place. As mentioned earlier, the message receiver is the end of `InFlow` that interacts with the service `impl` class. Therefore, Axis2 does not care about the message after handing it over to the message receiver. Notably, it is up to the message receiver to process the message and decide whether to send the response message or not, if there is any.

Axis2 distribution consists of a set of message receivers to support XML in XML out cases as well as to support the JavaBeans case.

- `RawXMLINOnlyMessageReceiver`: XML in only scenario
- `RawXMLINOutMessageReceiver`: XML in XML out scenario
- `RPCInOnlyMessageReceiver`: Java bean in only scenario
- `RPCMessageReceiver`: Java bean in out scenario

Transport sender

As we discussed earlier, the transport receiver is the starting handler of the inflow. In contrast, transport sender is the one that runs in the OutFlow as the last handler of the outflow. You can have different types of transport senders for different transports. For example, Axis2 has transport senders for HTTP, SMTP, TCP, and others. When you send the message in HTTP transport, the HTTP sender will be invoked. On the other hand, if you are sending the message via SMTP, the SMTP transport sender will be invoked.

Summary

Axis2 is good enough to provide web service interaction with dynamic and flexible execution frameworks. Flexibility is achieved using the concepts of phases and phase rules, and the dynamic nature of the execution chain has been achieved by runtime module engagement. In this chapter, we discussed the concept of handlers, phases, and how to use them (with an example). We also discussed the phase rules, how one can use them to locate a handler in a given flow, relatively or absolutely. At the end of the chapter, we discussed the special types of handlers in Axis2 called transport receivers, dispatches, message receivers, and transport senders.

The next chapter is one of the most interesting ones, where we discuss the Axis2 deployment model. The Axis2 deployment model provides several new features compared to Axis1, and the new deployment model is highly extensible and user friendly.

5
Deployment Model

Now the trend is not just to have features, but to have them in a very user-friendly manner. Time is the only thing they care about, and no one likes to spend days doing a small thing. In the previous versions of Apache Axis, user friendliness did not have a high priority because those were mainly to prove the web service concepts. Therefore, in Axis 1.x, the user has to invoke the admin client manually, update the server `classpath`, and then restart the server to apply the changes. This burdensome deployment model was a definite barrier for beginners. Therefore, Axis2 was engineered to overcome this drawback and provide a flexible, user-friendly, and an easily configurable deployment model.

In this chapter, we will be discussing:

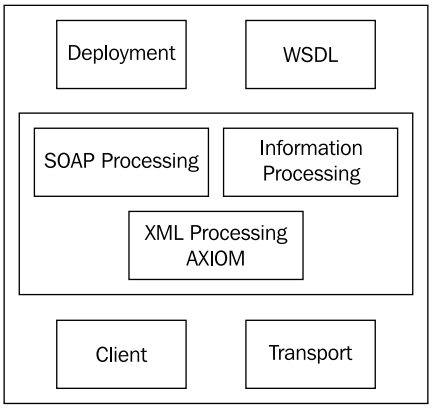
- The Axis2 deployment model, specifically the archive-based deployment model
- Hot deployment and hot update
- The concept of repository
- The different deployment descriptors
- A few hands on examples

What is new in Axis2 deployment?

As mentioned in *Chapter 1*, *Apache Web Services and Axis2*, one of the main goals of the Axis2 design is to provide more user friendliness; in the meantime, providing better extendibility and flexibility to the system. When it comes to user friendliness, service deployment is one area where the user needs less work. As a result, Axis2 supports a very convenient deployment model with a number of new features, compared to Apache Axis1. Some of the commonly used and useful sets of features are shown here:

- J2EE-like deployment mechanism (archive-based)
- Hot deployment and hot update
- Idea of repository
- Change in the way of deploying handlers (modules)
- Deployment descriptors
- Deployment options

The following figure shows the J2EE-like deployment mechanism:



In any J2EE application server, you can deploy an application as a self-contained package, where you can bundle all your resources, configuration files, and binary files together into one file, and deploy it.

Isn't that easy and useful? Well, the obvious answer is *yes, it is useful*.

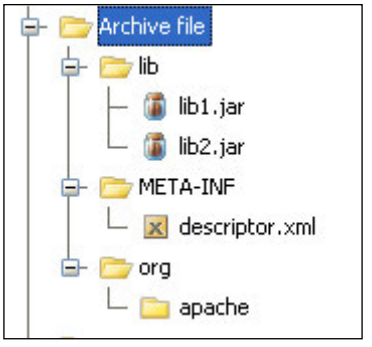
This is why Axis2 has introduced the same mechanism to deploy services (and modules) as well in a very convenient manner.

Let's think about a scenario where you have a service with several third-party dependencies and a number of property files. Further, assume that you do not have a J2EE-like deployment mechanism. Then what you have to do is put all those dependent JAR files and property files into the application `classpath`. This work is doubled if you have one or two servers, but what will happen if you are in a clustered environment with hundreds of replicates? In that case, it won't be practical to go and add the dependent JAR files and other resources into the `classpath` of each and every replica. So when you have the J2EE-like deployment mechanism, you do not need to worry about such issues. You can just drop the self-contained package—in this case, the service archive file—into the replicates. This definitely reduces your work and prevents common human errors as well.

The internal structure of the Axis2 self-contained package (or archive file) is as shown in the following figure. Both the Axis2 services archive and the module archive have a very similar structure, with only minor disparities.

In the case of the Axis service archive, the `descriptor.xml` file becomes `services.xml`, and in the other case, it becomes `module.xml`.

File extension for the Axis2 service archive is `.aar` and that for the module is `.mar` (the service archive or module archive is just a ZIP file with a changed file extension to `.aar` or `.mar`).



As mentioned earlier, in the case of a service the `descriptor.xml` file would be `services.xml`. So in the service archive file, you can find a file called `services.xml` inside the `META-INF` directory. On the other hand, for the module archive, `META-INF` will have a file called `module.xml`.

- For a service archive: `descriptor.xml` ---> `services.xml`
- For a module archive: `descriptor.xml` ---> `module.xml`

Hot deployment and hot update

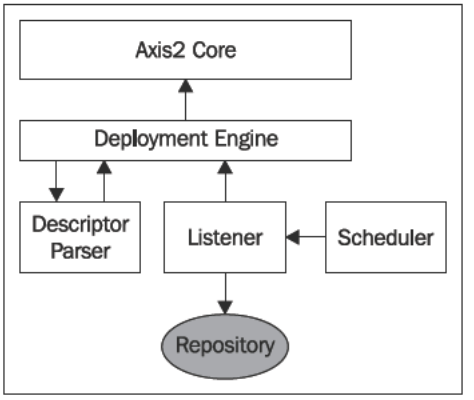
Availability is a big concern when it comes to enterprise-level applications, and in such a situation, even a fraction of time is highly valuable. Therefore, restarting a server is not a realistic option, and what is required is to update and change the system without shutting it down. This is where the hot deployment and hot update come into the picture. When your application has those features, you do not need to shut the system down in order to update the system.

Though the concepts of hot deployment and hot update are not new terminologies to the technical paradigm, these are new features in the Apache web service stack Axis.

In Axis2, hot deployment and hot update work by constantly monitoring the changes in the repository by a timer. More specifically, when the user changes the last modified date of a given file, it will treat it as a hot update. In contrast, when it finds a new file it treats it as a hot deployment. The architecture of hot deployment is shown in the figure in the next section.

Hot deployment

Hot deployment is the capability of deploying new services while the system is up and running. As an example, let's say that you have two services, *service1* and *service2*, up and running, and you deploy a new service called *service3* without shutting the system down. The system then makes *service3* a running service as well. This particular scenario is called hot deployment.



As a system administrator, if you do not like the hot deployment of a service, you can turn that off easily by changing the Axis2 global configuration file called `axis2.xml`. Changing the global configuration is just changing a parameter, shown as follows:

```
<parameter name="hotdeployment">false</parameter>
```

Hot update

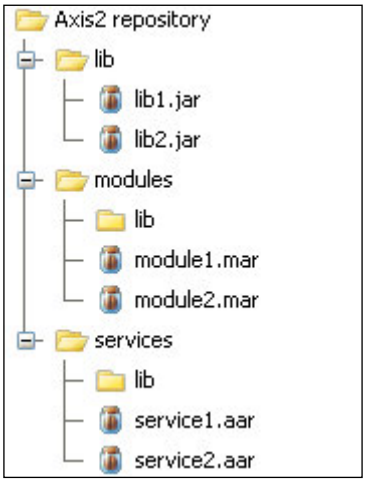
A hot update is the ability to make changes to an existing web service without shutting down the system. This is an important feature and is required in a testing environment. However, it is not advisable to use a hot update in a real-time system, because a hot update could result in the system leading into an unknown state. Additionally, there is a possibility of loosening the existing service data of that service. To prevent this, Axis2 comes with the hot update parameter set to `FALSE` by default, and if you want to have this feature, you could do that by changing the configuration parameter as follows:

```
<parameter name="hotupdate">true</parameter>
```

Repository

The Axis2 repository is just a directory in the filesystem with a specific structure. On the other hand, the repository can be located locally or in a remote machine. The idea of a repository was introduced to support archive-based deployment and hot deployment features in a very convenient manner.

The repository directory consists of two main sub directories called `services` and `modules`. Also, you may have an optional sub directory called `lib` as well. If you want to deploy a service, you need to drop the service archive file into the `services` directory. Similarly, if you want to deploy a module, then you need to drop a module archive file into the `modules` directory. The idea behind the `lib` directory is to store the third-party libraries that are going to be shared across both services and modules. Refer to the next screenshot to see the directory structure:



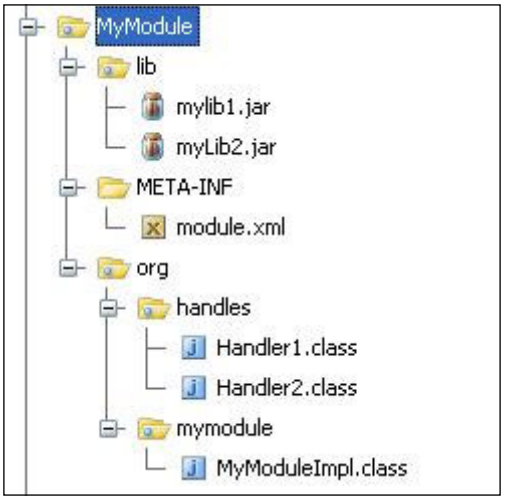
If one or more modules in the `modules` directory are required to share some resources, then they can add those resources into the `lib` directory inside the `modules` directory. Similarly, if the services in the `services` directory want to share some common resources, the proper place is the `lib` directory inside the `services` directory.

Change in the way of deploying handlers (modules)

The concept of **service extension** is a new feature to the Apache Axis paradigm, but the developers have achieved the same goal doing a hard job in Axis 1.x. So the idea is to extend the core functionality of the system or to provide quality of services. In the case of Axis 1.x, if you need to extend its core functionality, you need to write a handler (the smallest unit in the execution chain), change the global configuration files to add the handler, and finally restart the system.

A module does the same bit of work, but reduces the amount of work you need to do. In the meantime, a module can have one or more handlers alone with a module descriptor called `module.xml`. Most of the time, a module is an implementation of a specific WS specification. For example, an Axis2 addressing module is an implementation of WS addressing; Sandesha is an implementation of WS-Reliable Messaging.

As mentioned earlier, you can deploy a module as an archive file, and the structure of the module archive file is shown in the following screenshot:



Deployment descriptors

The flexibility and extensibility of Axis2 is focused on its deployment descriptors as well. Rather than working with one configuration file, Axis2 has different configuration files for different levels of configuration. For example, let's say that you want to have different types of configuration for different levels; then, having multiple configuration files for different levels solves the problem for you. There are three types of descriptors or configuration files in Axis2, namely:

- Global descriptor (`axis2.xml`)
- Service descriptor (`services.xml`)
- Module descriptor (`module.xml`)

Global descriptor or axis2.xml

As mentioned earlier, the configuration in Axis2 can be specified using XML descriptors. This gives you much more flexibility for extending and changing Axis2. No need to go and change the code to have a different configuration. Even for core functionality, it's the same. If you consider Axis2's global configuration file—`axis2.xml`—it has all the minimal configurations that are needed to run Axis2. Axis2's minimal configuration includes:

- Configuration parameters
- Transport senders
- Transport listeners
- Execution chains and phases
- Default dispatchers
- Default message receivers
- Default client-side configurations
- Global modules
- WS-Policy (global level policy)

Some of the given terms are not familiar to you, but you do not need to worry about them now. We will discuss these terms in the following chapters. Axis2 comes with default `axis2.xml`, and it has the minimum configuration required for starting Axis2. However, you can change it as you wish and can start Axis2 with your own `axis2.xml`. The key thing to remember here is that if you make any changes to `axis2.xml`, you have to restart the system in order to make those changes effective.

Service descriptor (services.xml)

As we discussed in the previous section, `axis2.xml` specifies the configuration that affects the whole system. However, `services.xml` is for configuring a particular service or a service group. In the next section, we will look at the available ways of deploying services in Axis2. Archive-based deployment and directory-based deployment can be considered as the two most commonly used techniques. In these two cases, for a service to be a valid service, it is required to have a `services.xml` file. A service descriptor is used to specify the following types of configurations (some of these are optional):

- Name of the service
- Target name spaces of the service
- Session scope

- Expose transports
- Service level and operation level parameters
- Message receivers
- Service level modules
- Operations—expose operations as well as non expose operations
- Bean mapping
- Object suppliers
- Service and operation level policy

We will learn more about `services.xml` and how to write and create services in *Chapter 7*, *Writing an Axis2 Service*.

Module descriptor or module.xml

It is so obvious now that `module.xml` is for configuring Axis2 modules; so it also has different types of configurations:

- Handlers and their phase rules
- Module parameters
- End points
- WS-Policy

In *Chapter 8*, *Writing an Axis2 Module*, we will learn more about Axis2 modules and how to write them.

Available deployment options

In the initial stage of Axis2, it only had archive-based deployment. However, later it introduced a number of convenient deployment options. This relay made the service authors' jobs easier. Adding them all together, Axis2 has the following deployment options for service deployment:

- Archive-based deployment
- Directory-based deployment
- Deploying programmatically using archive files
- Programmatically making the `Java` class into a web service
- **Plain Old Java Object** (**POJO**) deployment support alone with annotation
- Deploy and start Axis2 in one line

Archive-based deployment

The most common and recommended approach for deploying a service in Axis2 is **archive-based deployment**. There you get many configuration options and more flexibility compared to other types. In this chapter, we will discuss more about archive-based deployment with samples.

Directory-based deployment

Directory-based deployment is almost identical to archive-based deployment; the only difference is that rather than creating an archive file, you can deploy a service as a directory. The structure of the directory is identical to that of an archive file.

Deploying a service programmatically

To deploy a service programmatically using an archive file is not really a user requirement; rather it is module author's requirement, where some modules require a web service to be deployed at runtime in order to provide the full functionality of that particular module.

To create a service (`ServiceGroup`) programmatically, you need to have the file object representing the service archive file and a pointer to an Axis2 runtime or `ConfigurationContext`. Once you have those two, we can create a web service, as shown in the following code fragment. The advantage of this approach is that you do not need to copy your service archive file into the repository and it is only visible at the runtime of your service.

```
//Need to have a reference to ConfigurationContext
ConfigurationContext configContext = getConfigurationContext();
File serviceArchiveFile = new File("Location of the file");
//Now let's create AxisServiceGroup which contains the service we want
to have
AxisServiceGroup serviceGroup = DeploymentEngine.loadServiceGroup(
         serviceArchiveFile,
         configContext);
```

Once you have created a service, the next step is to add the service into the system. You can do that as follows:

```
//Getting a pointer to AxisConfiguration
AxisConfiguration axiConfiguration = configContext.
getAxisConfiguration();
//Adding the created service
axiConfiguration.addServiceGroup(serviceGroup);
```

POJO deployment

To continue the discussion on the other deployment options, first you need to create a `Java` class, which you want to expose as a service. Let's assume that you have a web service with two methods, namely, `sayHello` and `add`, then your service Java class will look like this:

```
public class MyService {

    public String sayHello(String name) {
        return "Hello " + name;
    }

    public int add(int a, int b) {
        return a + b;
    }
}
```

Making a Java class into a web service is a very handy feature in Axis2 and it is very useful when debugging in developing web services. In this case, you do not need to know anything about the archive file concept, `services.xml`; you just need to have a pointer to `AxisConfiguration`. You can then make a web service using the Java class as follows:

```
//Need to have a pointer to AxisConfiguration
AxisConfiguration axiConfiguration = getAxisConfiguration();
//Creating a service using java class
AxisService service = AxisService.createService(
                MyService.class.getName(),
                axiConfiguration);
// Adding the created Service in to AxisConfiguration
axiConfiguration.addService(service);
```

The given deployment mechanism is one way of doing POJO deployment. There are other ways of deploying POJO, for example by copying `.class` files into the `repository` or `pojo` directory. One thing to note here is that you can deploy a service as mentioned if, and only if, you have a pointer to an `AxisConfiguration`.

However, in order to use this mechanism, you need to have `AxisConfiguration`, otherwise you cannot use this mechanism. Instances where you do not have a way to access the `AxisConfiguration`, you need to find some other way of achieving your goal. So, the other type of POJO deployment mechanism will help you in this case. One easy way is to deploy `.class` files into a directory called `pojo` in the Axis2 repository. Then Axis2 will process that `.class` file and make that into a web service for you. Now compile your Java class to get the `MyService.class` file. You first need to create a directory inside the `repository` (a `repository` directory inside the place where you unpack the Axis2 binary distribution) called `pojos` and that should be in the same level as `services` and the `modules` directory. So now you have the repository as follows:

```
Axis2

  -repository

    -services

    -modules

    -pojos
```

Now drop the `MyService.class` file into the `pojo` directory. If the server is not running, you need to start Axis2 and enter the following URL in the browser and see what you get:

```
http://localhost:8080/service
```

This gives you a hint that your service is up and running. Now let's try to invoke the service and see whether it is working. As you have not learnt about the Axis2 client programming model, let's try to invoke the service in the REST way. Enter the following in a browser and see what you get:

```
http://localhost:8080/axis2/services/MyService/sayHello?name=Axis2
```

You will see the following:

```
<ns:sayHelloResponse>
   <return>Hello Axis2</return>
</ns:sayHelloResponse>
```

This simply tells you that you have actually invoked the service. Let's try to invoke the `add` method and see:

```
http://localhost:8080/axis2/services/MyService/add?a=10&b=15
```

```
<ns:addResponse>
   <return>25</return>
</ns:addResponse>
```

This is the exact addition of the two given numbers. Now you are sure that we expose and invoke our Java class as a web service.

If you want to test this with the Axis2 web distribution, then you can do that by copying the `MyService.class` file in the path `TOMCAT_HOME/webapps/axis2/WEB-INF/pojo`.

After this, you need to follow the steps and see what you get.

Deploying and running a service in one line

Among all the mentioned deployment options, this option can be considered as the most convenient way of deploying a service and starting the server. You do not need to have a repository, `services.xml`, or anything else of the sort. The only thing you need to have is the Axis2 library file (`axis2-1.3.jar`) and its dependent libraries. You can then deploy and start the Axis2 server as follows. This way of deploying and running the server is very useful when you're debugging and developing a service.

```
new AxisServer().deployService(MyService.class.getName());
```

When you start AxisServer, it will start SimpleHttpServer in the port specified in the Axis2 default configuration file and that would be port 6060.

So now if we type `http://localhost:6060`, we will see the following:

If you run the following URLs in the browser, you will get the same result as seen earlier:

```
http://localhost:6060/axis2/services/MyService/sayHello?name=Axis2
<ns:sayHelloResponse>
   <return>Hello Axis2</return>
</ns:sayHelloResponse>
```

Now, if you try to access the `add` method using the following URL, you should get the same results as before:

```
http://localhost:6060/axis2/services/MyService/add?a=10&b=15
<ns:addResponse>
   <return>25</return>
</ns:addResponse>
```

Summary

In this chapter, we discussed how Axis2 deployment works, the available types of deployment descriptors, and their structures. In addition, we also discussed the different types of configuration files available in Axis2. At the end of the chapter, we learned the most important thing, that is, the available deployment options in Axis2. We tested those options with a sample as well. The next step would be to create a few more services and see what happens.

In the next chapter, we will discuss more about the Axis2 information model. There we will discuss about static and dynamic data hierarchies, different entities in the hierarchy, how they are created, and which descriptors are responsible for the different entities.

6
Information Model

Since the last decade, **Service Oriented Architecture** (**SOA**) has been gaining a lot of popularity in the information technology industry. In today's industry, most of the applications try to enable SOA APIs on their applications, for example, Google APIs and Amazon Web Services to name a few. There are a number of reasons behind this trend:

- SOA is easy to use
- There are standard bodies that help to improve every type of application to interoperate well
- Free and open source SOA frameworks

The web service landscape is changing very rapidly, and hence, to support these changes and to facilitate user requirements, a web service framework has to be flexible and extensible enough. Applications, such as Internet commerce (e-commerce), need to manage two types of data:

- Static data (for example, information, product information)
- Dynamic data (for example, transaction data, shipping information)

As a result, most applications need to have better support for static and dynamic data. Furthermore, having static and dynamic data separate leads to better flexibility and extensibility in the system. Notably, most web service frameworks (for example, Axis2) have the notion of **stateless**, which means those frameworks do not maintain the sessions. Nevertheless, building complex systems is a tedious task in the absence of session support. Thus, Axis2 tries to provide stateless as well statefull services in a convenient manner, by having two data hierarchies—one to manage static data and the other to manage dynamic data.

In this chapter, we will discuss the Axis2 information model, and in particular, two data hierarchies, namely, static data and dynamic data. We will discuss more about the following topics:

- The concepts of Axis2 static and dynamic data
- Axis2 static data and their types
- Parameters and their hierarchy
- Formatting and building messages through `MessageFormatters` and `MessageBuilders`
- Transport senders and transport receivers
- Static data hierarchy focusing on `AxisOperating`, `AxisService`, `AxisServiceGroup`, and `AxisModule`
- Dynamic data hierarchy, including `ConfigurationContext`, `MessageContext`, `OperationContext`, and `ServiceContxt`

Axis2 static data

As mentioned earlier, for any given application, it is common to have two types of data called **static** and **dynamic**. A classic example of this is a banking application. It needs to keep the data such as user information (for example, name, address, date of birth, phone number, and other such information) static (changes that are infrequent). Now, let's consider an application that provides XML-based configuration (for example, Apache Tomcat, Apache Axis2, and so on). At the time of system initialization, it is required to load these configurations and store them in some structure (or use them to configure the application). Once the application stores all the configuration information in a static structure, it can use that information whenever it's required with minimum or zero cost. In contrast, if the application is written in such a way that, whenever it needs to read some information, it reads the XML file and retrieve data, it leads to higher performance issues.

Performance is a major consideration when it comes to web services, loading configuration data at runtime from a secondary storage (when required) is not an option. So it may be better to keep such data in memory, ready for use whenever required. On the other hand, if we have to contain all the configuration data in one large object, it can also add to the performance overhead (Hot pots). To address this problem in a more efficient manner, Axis2 has an object hierarchy to store configuration data more organized manner. Some of the objects in the hierarchy will be created at deployment time and some at runtime, depending on the deployment options employed. Now let us discuss the different types of objects in this hierarchy and try to understand how and when they are created.

Axis2 has three types of configuration files (deployment descriptors) that populate and configure the object hierarchy. Those types are as follows:

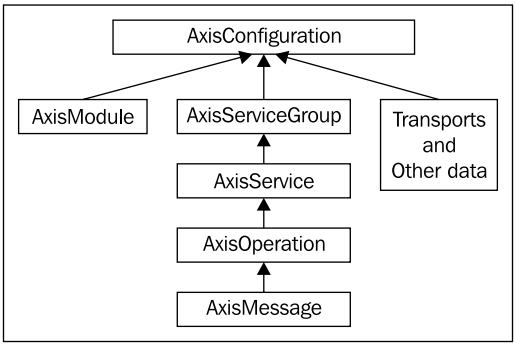
- Global level configuration file – `axis2.xml`
- Service level configuration file – `services.xml`
- Module or service extension configuration file – `module.xml`

The global configuration file is called `axis2.xml` and contains the bare minimum configuration data that is needed to start an Axis2 Web Service framework, either as a server or as a client. As we will discuss in this chapter, a user can modify `axis2.xml` to suit the user requirements and start Axis2 with the modified file. This is the whole point of providing a configuration file. In the web service domain, there are a large number of parameters that users wish to configure such as the default SOAP version to be used, default HTTP version, namespaces, platform-specific configurations, other user data, data binding information, and so on. Being a very configurable web service framework, this holds true for Axis2. A typical `axis2.xml` that we can use to run Axis2 has the following set of configuration options:

- Deployment configuration data
- Transport senders
- Transport receivers
- Execution chains
- Phases
- Parameters
- Message formatters and message builders

We have already encountered some of the mentioned terms and the rest will be discussed soon in this chapter as well as throughout this book.

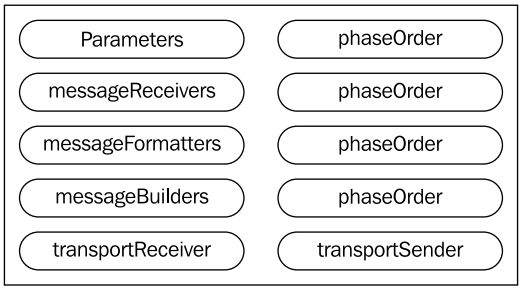
The following figure shows the relationship among various types of descriptions or metadata in Axis2:



As we can see, the top-most component in the hierarchy is `AxisConfiguration` that keeps track of all the configuration data, either directly or indirectly. There are three major types of objects shown in the figure. Firstly, `AxisModule` originates from a descriptor file called `module.xml`, so that when we deploy a module in Axis2, there will be a new `AxisModule` object to keep track of that particular module's configuration data. Secondly, the middle object hierarchy is created when we deploy a service in Axis2. Finally, there are the transports and other data that are read directly from `axis2.xml`.

AxisConfiguration

`AxisConfiguration` is the top-most component of the static data hierarchy. Nevertheless, it should be noted that although we call `AxisConfiguration` as a static hierarchy, it does not mean that it is immutable. There are a number of instances where it changes the content, but these changes are very infrequent. The whole `AxisConfiguration` object is effectively a collection of data coming from `axis2.xml`, a set of `module.xml` files, and a set of `services.xml` files. There are many ways to *create* `AxisConfiguration` as well. One could create an `AxisConfiguration` using the local filesystem, using a remote repository, or even by using a database. We will discuss these options in *Chapter 15, Building a Secure Reliable Web Service*. In this chapter, the focus will be on creating an `AxisConfiguration` using the default `axis2.xml` file from the local filesystem. A typical `axis2.xml`, which has the bare minimum configuration data to start an Axis2 server, is shown in the following figure (a detailed explanation of the element structure is given after the figure):



Parameters

As you can see in the previous figure, the `axis2.xml` file has parameters, and these can be defined at different levels in the document as well. As illustrated, we have parameters at the top level as well as inside the transports. The main use of parameters is to configure the system and to provide configuration data that is needed at runtime. For example, if we need to log some request to a particular location, that location can be provided using a parameter. Parameters are designed to store primitive data types (for example, `string`, `int`, `double`, and so on) and `OMelements`, but *not* any type of objects. The following code snippet shows how to define a parameter in any of the configuration files in Axis2.

Each parameter has an optional attribute called `locked`, as shown here:

```
<parameter name="name" locked="true/false"> value </parameter>
```

The idea of the `locked` attribute is to provide a control mechanism to make sure that none of the child nodes override that parameter. For example, let's say we have a parameter, as mentioned here, as an immediate child of `axis2.xml`:

```
 <axisconfig name="AxisJava2.0">
       <parameter name="port" locked="true">6060</parameter>
       ....................
</axisconfig>
```

This will ensure that any other description in Axis2 cannot have a parameter with the same name, here `port`. This phenomenon is illustrated in the following section. Here, `axis2.xml` becomes invalid, as it tries to override a locked parameter. Furthermore, we cannot have the parameter with the name `port` in any of the other two descriptors, which are `services.xml` and `module.xml`. If they exist, they become invalid.

```
<axisconfig name="AxisJava2.0">
       <parameter name="port" locked="true">6060</parameter>
      <transportReceiver name="http"
```

```
                        class="o.a.a.t.h.SimpleHTTPServer">
            <parameter name="port">6060</parameter>
            </transportReceiver>
    </axisconfig>
```

There is a scope associated with a parameter. A parameter defined in `axis2.xml` as an immediate child can be accessed by any of the descriptors in the system. If a parameter is defined inside a first level child of `axis2.xml` (for example, `transportSender`), then that parameter can only be accessed inside that particular child, in this case that specific `transportSender`.

When accessing a parameter, Axis2 first checks whether the parameter is defined in the current description where we are. If not, it checks the immediate parent to see whether the parameter exists there. If found, the parameter will be returned; otherwise, the parent's parent will be checked, and so on. In this manner, it will search the hierarchy when a request for a parameter is made.

MessageReceiver

In *Chapter 4*, *Execution Chain*, we discussed more about `MessageReceiver` and its usage. All the message receivers we are going to use for our services should be specified in `axis2.xml`. You can have as many message receivers as you want and use the right one at the `services.xml`. The way to specify the message receiver in `axis2.xml` is shown next:

```
        <messageReceivers>
            <messageReceiver mep="MPE"
                             class="o.a.a.r.RawXMLINOnlyMessageReceiver"/>
        </messageReceivers>
```

Note: In the preceding XML element, MEP stands for the Message Exchange Patterns that the `MessageReceiver` can handle. Class attribute is to specify the actual implementation class of the MR. You can register any number of message receivers alone with a unique MEP. For example, WSDL 2.0 allows 8 types of MEPs.

MessageFormatters and MessageBuilders

In HTTP, we use a content-type header to specify the type of data in the message body. Moreover, depending on the content type, the wire format of the message varies, for example, XML, JSON, base64, and so on. Axis2 also supports a number of different message types, thus it requires serializing and deserializing a message into the correct format. To facilitate, Axis2 has introduced `MessageFormatters` and `MessageBuilders`; the first one is to serialize the message into the wire format and the second one is to build the SOAP message from the incoming message stream. We already know that any kind of message is represented in Axis2 using Axiom, and when we serialize the message, it needs to be formatted, based on the content type. `MessageFormatters` exist to do that job for us. We can specify `MessageFormatters` along with the content type in `axis2.xml`. On the other hand, a message coming into Axis2 may or may not be XML. However, for it to go through Axis2, an Axiom element needs to be created. Therefore, `MessageBuilders` are employed to construct the message, depending on the content type.

These two types of descriptions can be considered complex, and as users are not likely to change them, users can happily live with the default `axis2.xml`, as it will have configured all the commonly used content types along with their corresponding builders and formatters. The structure of XML elements in `axis2.xml` is shown next:

```
<messageFormatters>
    <messageFormatter contentType="application/x-www-form-
      urlencoded"
      class="o.a.a.t.h.XFormURLEncodedFormatter"/>
</messageFormatters>

<messageBuilders>
    <messageBuilder contentType="application/xml"
      class="o.a.a.b.ApplicationXMLBuilder"/>
</messageBuilders>
```

As shown here, you can specify message builders and formatters with content type and the implementation class. At runtime, Axis2 picks the right one to serialize and de-serialize the message.

TransportReceiver and TransportSender

Axis2 is transport independent, and hence one can communicate with Axis2 using a number of transports. For example, Axis2 has inbuilt support for HTTP, TCP, SMTP, and JMS. All of these are configurable through axis2.xml.

The job of a transport sender is to serialize and handle the message exchanges, depending on the underlying protocol. On the other hand, the transport receiver's job is to deserialize an input stream into Axiom and respond to the client according to the protocol. Axis2 comes with a better configuration for all the transport. However, transport such as SMTP requires some user involvement to provide correct server names, for example, POP and SMTP. Other than that, a user does not need to change the original transport definition. The structure of XML elements in `axis2.xml` is shown next:

```
<transportReceiver name="http"
                   class="o.a.a.t.h.SimpleHTTPServer">
    <parameter name="port">6060</parameter>
</transportReceiver>

<transportSender name="http"
                 class="o.a.a.t.h.CommonsHTTPTransportSender">
    <parameter name="PROTOCOL">HTTP/1.1</parameter>
    <parameter name="Transfer-Encoding">chunked</parameter>
</transportSender>
```

As shown here, you can add new transport senders and receivers by specifying name (a.k.a protocol) and the implementation class. In addition, you can also add transport specific parameters.

Flows and phaseOrder

In *Chapter 4*, *Execution Chain*, when we discussed the Axis2 execution chain, we talked about the use of flows and `phaseOrders`. As mentioned, Axis2 comes with a set of predefined phases and flows such as in-flow, out-flow, and so on. A user does not need to change them unless they have some specific requirements. However, if it is necessary to change the configurations, you have learned how to do that as well in *Chapter 4*. The structure of four different flows and the way to specify them in `axis2.xml` is shown here:

```
<phaseOrder type="InFlow">
     <phase name="Transport">
        <handler name="RequestURIBasedDispatcher"
                 class="o.a.a.d.RequestURIBasedDispatcher">
            <order phase="Transport"/>
        </handler>
         ..........................................
    </phase>
    <phase name="Security"/>
```

```
    ..................................................
</phaseOrder>
<phaseOrder type="OutFlow">
    <phase name="OperationOutPhase"/>
    ..................................................
</phaseOrder>
<phaseOrder type="InFaultFlow">
    <phase name="PreDispatch"/>
   ..........................................
</phaseOrder>
<phaseOrder type="OutFaultFlow">
    <phase name="OperationOutFaultPhase"/>
   ..........................................................
</phaseOrder>
```

So far, we have discussed the different types of configuration data that comes from the `axis2.xml`. Now let us look at the other types of descriptions.

AxisModule

In simple terms, an `AxisModule` is a runtime representation of a `module.xml`. So all the configuration data found in `module.xml` is in `AxisModule`. A typical module configuration file or `module.xml` contains the following data:

- Module name
- Module description
- Handlers and phase rules
- End point and operations
- WS-Policy
- Parameters

At the time of deployment, an `AxisModule` is populated using the data from a `module.xml`. At runtime, any part of this data can be retrieved via the same `AxisModule`. Once the data is populated and the module is complete, the parent description of the `AxisModule` becomes the `AxisConfiguration`.

Service description hierarchy

As we can see in the first figure, in the middle we have an object hierarchy called the `AxisService` hierarchy. This particular hierarchy is created using a `services.xml` file or the service descriptor and the hierarchy contains four types of descriptions.

When we deploy a service into Axis2, an object hierarchy will be created and added to `AxisConfiguration`. Therefore, unless we have services deployed in Axis2, we do not have the service objects hierarchy in the `AxisConfiguration`. Unlike `AxisModules` and other descriptions (for example, transports and message formatters), the service description hierarchy is likely to be changed at runtime, depending on the deployment options. A typical `services.xml` is shown next to help explain the object hierarchy in a more specific manner better:

```
<serviceGroup>
 <parameter name="name">value</parameter>
 <service name="Foo">
     <parameter name="name">value</parameter>
     <operation name="bar">
         <parameter name="name">value</parameter>
        <message label="in"></message>
     </operation>
 </service>
 <service name="XYZ">
 .....
 </service>
</serviceGroup>
```

AxisServiceGroup

`AxisServiceGroup` is the top-most component of the service description hierarchy and is the child of `AxisConfiguration`. `AxisServiceGroup` can be considered as the parent of a set of `AxisServices` that are defined in `services.xml`. Once we define a parameter in `AxisServiceGroup`, that parameter can be accessed from any `AxisService`, `AxisOperation`, or `AxisMessage`, lower in the hierarchy. In addition to parameters, an `AxisServiceGroup` may contain collections of modules engaged to this particular `AxisServiceGroup`.

Note: The idea of a service group is a logical concept; the main goal of having service group is to deploy a set of related services together and share static and dynamic data across the services in the group.

AxisService

An `AxisServiceGroup` should contain one or more `AxisServices` as children. Therefore, any configurations (for example, parameters) defined in `AxisServiceGroup` or `AxisConfiguration` are accessible inside an `AxisService`.

`AxisService` has the following data and configurations:

- `AxisOperation`
- `Parameters`
- `Exposed transports`
- `Engaged modules`
- `Namespaces`
- `Exposed transports`
- `Description about the service`
- `Message receivers`
- `WS-Policy`

In the next chapter, we will discuss these terms in detail and explain how to create and deploy services in Axis2.

AxisOperation

`AxisOperation` is the runtime description representation of an exposed web service operation. As an example, let's say we have published an operation called `bar` in the Web service `Foo`. Then there should be an `AxisService` object called `Foo` and that object should have an `AxisOperation` object called `bar`. The parent description of an `AxisOperation` is the `AxisService`, and any parameter defined in the parent's descriptions can be accessed inside the child, in this case, the `AxisOperation`. So any parameter in `AxisConfiguration`, `AxisServiceGroup`, and `AxisService` can be accessed and used inside `AxisOperation`. In addition to parameters, `AxisOperation` contains:

- AxisMessage
- Engaged modules
- Operation name
- Soap actions
- WS-Policy

AxisMessage

`AxisMessage` is the leaf element of the service hierarchy, and the immediate parent of the `AxisMessage` is an `AxisOperation`. As we discussed, `AxisService` can have any number of `AxisOperation`. However, `AxisOperation` cannot have any number of `AxisMessage` elements. The number of `AxisMessages` in an operation is determined by the message exchange pattern. As an example, if the operation is in-out, there will be only two `AxisMessage` elements in it, one to represent the in message configuration and another to represent the out message configuration. `AxisMessage` has the following set of data:

- Parameters
- WS-Policy
- Message label
- Element Qname of the corresponding schema element (optional)

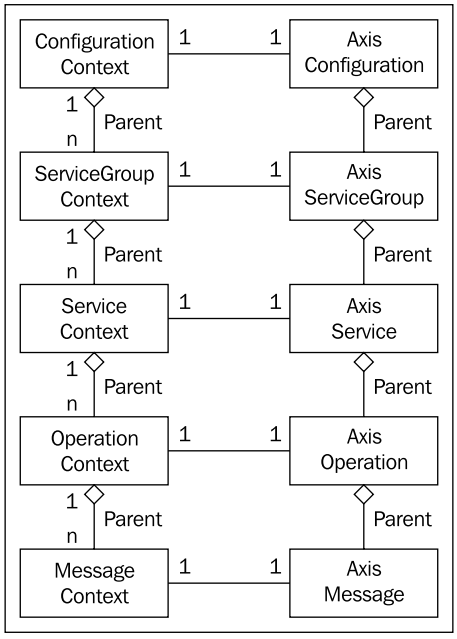
Note: A message element is an optional element, thus, we can have operation elements with zero message elements. You need to add message elements only if you are going to override the message, such as adding a new policy.

Now, we have a good understanding of Axis2 static data and how to change them, hence the next step is to learn about dynamic data hierarchy and the relationship between static data and runtime data. The following figure shows the relationship between the descriptions hierarchy and the contexts or the runtime data hierarchy.

Axis2 contexts

The Axis2 context hierarchy is the runtime data representation of Axis2. Runtime data comes into the picture only when Axis2 receives a message, (unlike descriptions). The runtime data is used to share data across multiple invocations or among the handlers in one invocation. When we were discussing the description hierarchy, we talked about how parameters act as the main configuration mechanism. In context, the main configuration or data sharing mechanism is the use of `Properties`. Unlike parameters, we do not need to define properties anywhere, as we can create them on-the-fly and use them. Properties are stored as name-value pairs in the context hierarchy. Hence, if we add a property to the context, then that property can be accessed and overridden by any other of its child contexts.

The following figure shows the relationship between static data and runtime data in Axis2:



As you can see on the left-hand side, the top most component of the hierarchy is `ConfigurationContext`. The only difference, when `ConfigurationContext` is compared to other contexts, is that it is the only context that exists in the system before receiving a message. `ConfigurationContext` has a reference to an `AxisConfiguration`, and to create `ConfigurationContext`, it is required to have an `AxisConfiguration` available.

ConfigurationContext

A `ConfigurationContext` is a runtime representation of the whole system. To start Axis2, you need to have a configuration context. The lifetime of the configuration context will be the lifetime of the system, so if we store a state (a property), it stays there forever (that is, until the system is shut down). `ConfigurationContext` is not only the parent of all other context, it can also be considered as an originator element of the entire Axis2 system.

ServiceGroupContext

When a message is received by Axis2, a `ServiceGroupContext` is created to store and shared data. To create a `ServiceGroupContext`, it is required to have an `AxisServiceGroup` object available. One `AxisServiceGroup` may have one or many `ServiceGroupContext`. Nevertheless, there is only one `AxisServiceGroup` associated with the context. The lifetime of the `ServiceGroupContext` depends on the service scope (for example, application, transport, SOAP session, and request). If the service scope is *application*, then the lifetime will be the same as system lifetime. However, if the service scope is "request", then there will be a `ServiceGroupContext` created for each and every invocation. If we want to share data across multiple services in one service group at runtime, then the `ServiceContext` provides the means to do so.

ServiceContext

A `ServiceContext` represents the runtime data for a given service. To create a `ServiceContext`, we need to have a `ServiceGroupContext` object and an `AxisService` object available. The lifetime of the `ServiceContext` depends on the scope of the service. If we want to share data across multiple invocations of the same service, then `ServiceContext` can be used as the placeholder to store that data.

For example, let's say we have a service with three operations—`login`, `doSomething`, and `logout`. Also, assume that we need to share data across these three operations. Here, `ServiceContext` can be used to achieve these objectives. The number of `ServiceContexts` in one `ServiceGroupContext` is dependent on the number of `AxisServices` in the corresponding `AxisServiceGroup`.

OperationContext

An `OperationContext` represents the lifetime of an MEP. More often than not, the lifetime of an operation context is less than the lifetime of the service context. We can use `OperationContext` to share data among messages in an MEP. For example, if we want to share data between a request and the response, then `OperationContext` can be used to achieve that. The number of `OperationContext` contexts in a `ServiceContext` is not related to the number of `AxisOperation` operations in an `AxisService`; it only depends on the number of invocations, for example, one `OperationContext` for each invocation.

MessageContext

When a message is received by any transport, the first thing it does is create a `MessageContext` to represent the incoming message. To create a `MessageContext`, we need to have `ConfigurationContext` available. The lifetime of the message context is the same as the processing time of the message, for example, the lifetime of an incoming `MessageContext` is the time taken for a message to travel from the transport receiver to the message receiver. In the case of an outgoing message, the lifetime will be the time taken by a message to reach the transport sender, from the moment it left the message receiver.

Although we can see a hierarchy in the second figure in the case of incoming messages, the hierarchy will not be complete until the message has passed the dispatchers. Until that happens, it has only the `ConfigurationContext`. In the case of outgoing messages, the complete hierarchy is available.

Summary

In this chapter, we discussed the Axis2 runtime data hierarchy and static data hierarchy. We discussed how and when these are created. Most of the things we discussed here are important, if you plan to create a complex service such as a session-aware service or when writing handlers. If you are going to use Axis2 to simply deploy and invoke a service, you do not need to worry too much about the facts discussed here.

Now that we have gained a good understanding of the Axis2 concepts' usefulness, it is time to use these concepts practically. In the next chapter, we will discuss how to use Axis2 in your web service application. It also covers writing web services using Axis2.

7
Writing an Axis2 Service

In the previous chapters, we introduced and discussed various Axis2 concepts and highlighted the importance of them. Starting from this chapter, we are going to discuss how to use those concepts in the real world. Each chapter provides you with a comprehensive set of examples, which help you to understand the concepts clearly. Repeating the example in your own environment would help you to gain the most out of each chapter. Axis2 introduces several new features to its web services. For example, annotation support, session support, and ways to store session-aware data in the information hierarchy, POJO, and Spring-based web services.

As we have already discussed in the previous chapters, there are two main aspects to any given web service framework; firstly, to provide a hosting environment for web services, and secondly, to provide an invocation framework to access a remote service. Hosting a service includes implementing a service, deployment of the service, and lifecycle management of the service. In Chapter5, we briefly discussed a number of different ways to deploy a service. Here, in this chapter, we are going to discuss how to implement a service and deploy it on the Axis2 Web Service hosting environment. Moreover, in this chapter, we will cover the two fundamental approaches that are commonly used in the industry to create web services. At the end of this chapter, you will have a good understanding of how to create a web service and deploy it in Axis2. In *Chapter 9*, *The Client API*, we will discuss how to consume a web service deployed in Axis2 or elsewhere.

Creating a web service

When it comes to problem solving, there are two main approaches that we commonly use—the top-down approach and the bottom-up approach. Not surprisingly, those approaches are applicable in the web service world as well. In the web service world, the two approaches have been given different names but the concepts remain the same:

- **Code first approach**: It is the same as the bottom-up approach, where we first start with the source code and eventually expose our source code as a web service. More concretely, the code first approach helps to easily convert an existing application into a web service. In general, this approach reduces the learning curve, where a user does not need to have a very good understanding of the web service concepts, having a fair understanding of how a specific framework works would help to easily achieve the goal. POJO or Plain Old Java Object is one of the very good examples of the code first approach. Notably, with the code first approach, you might not be able to get the full power of the framework and web service, but you can still achieve your goal.
- **Contract first approach**: As we discussed in the previous section, the code first approach is analogous to the bottom-up approach. In contrast, the contract first approach is analogous to the top-down approach. In this process, we first write the **Web Service Description Language** (**WSDL**) document according to the service contract, its meaning, and so on. Entities participating in the service invocation come up with a set of APIs and then map that into the WSDL document. Once we have the WSDL document, we can use the tool supported by the web service framework to convert the WSDL document in the framework-dependent code (skeleton) and then complete the business logic.

Considering both the approaches together, we can get an idea of the pros and cons associated with each. By the end of this chapter, you will be able to understand why one approach may be picked over the other, depending on the scenario at hand.

As mentioned before, in the code first approach, we first write the service implementation class and other relevant classes. The important factor to note here is that with the code first approach, we do not need to have an understanding of the concepts behind the web services beforehand (for example WSDL, XML, SOAP, and so on). All you need to do is write your service implementation class with a set of public methods that you are planning to expose as web service operations. So using this approach, even someone who has no knowledge of WSDL or SOAP can deploy a service and easily access it as a web service. Sometimes, this approach is also known as the **POJO approach**. One of the key advantages of this approach is that you can change your service class as and when you wish to, as you do not need to worry too much about method parameters, methods names, and so on. So the key objective is to meet a customer's requirements very conveniently.

On the other hand, in the contract first approach, we first start with an agreement between the two or more parties who come up with one or more mutually agreed upon contracts. These contracts might be based on underlying protocols, security considerations, and other factors such as policies. In most web service frameworks, the contract is written using WSDL. Using the prepared WSDL document, one party can create a service while others can use the same document to create clients to invoke that service.

In almost all the web service frameworks available today, you can find integrated tools to create both client-side and server-side code using the given WSDL document. The process of creating a service and client code using a WSDL document is called **code generation** and the generated classes are called **data bound classes**. In this approach, the advantage is that you can use the tools available in the distribution to create service skeletons or the service proxies (stub), which you can fill in to implement the desired business logic. Therefore, manual code writing is greatly reduced. The major disadvantage here is that if ever you needed to change the WSDL document, then you will have to generate code again; whereas in the code first approach, changing a service class is a pretty straightforward task. However, once two parties agree upon a set of policies and come up with the WSDL document, it is not likely that the WSDL will undergo changes.

If you are new to the web services area, then it might be best to start with the POJO approach, as all it involves is writing Java code (e.g., in the case of Java based Web services). Despite this fact, the end result is a service which can be exposed and consumed universally. So let us first discuss the code first approach and see how you can start writing an Axis2 compatible web service using a plain old Java class.

The code first approach

In the code first approach, we start by writing the service implementation class, that is, the class that provides the service. We will begin by writing a simple web service that says *Hello* <*name*> when invoked. Note that <name> here will be your name given as an input parameter to the service.

Single class POJO approach

Let us first write our service implementation class. Since Axis2 is a Java-based web service framework, we are going to use Java as the primary language for implementing the service. Hence, the Java code for your service class should be somewhat like the following. In this case, let us assume that the class does not have any package name declared, that is, the class is in the default package.

```
public class HelloWorld {
    public String sayHello(String name) {
        return "Hello " + name;
    }
}
```

As you can see, even this minimal amount of code is enough to be a service on its own. We already discussed in *Chapter 5*, *Deployment Model*, more about the available deployment options in Axis2. As we discussed, there are a number of ways of deploying a service. More importantly, even for POJO deployment, there are a few ways in which you can make your Java class into a web service.

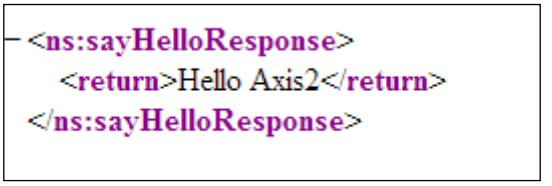
Let us start with the simplest approach; here, we assume that you are going to deploy the service in the Tomcat application server. If you are using some other application server, then you need to change the following path accordingly. Once you've followed the steps listed here, you will have deployed and invoked the service as well:

1. Compile the Java class, which will produce the `HelloWorld.class` file.
2. Go to `<TOMCAT_HOME>/webapps/axis2/WEB-INF`.
3. Create a directory named `pojo` inside the WEB-INF directory.
4. Then copy the `HelloWorld.class` into the `pojo` directory, created in the previous step.
5. Start Tomcat.
6. Go to `http://localhost:8080/axis2/services/listServices`, where you will be able to find a service named `HelloWorld` listed.

- Now open your browser and type the following URL and see what you get. It should be noted here that depending on the port that you have specified in Tomcat, you need to change 8080.

```
http://localhost:8080/axis2/services/HelloWorld/
sayHello?name=Axis2""
```

Now you should be able to see Hello Axis2 in the browser. You can experiment by changing the `name` parameter, and you will find that the service responds with a `Hello` to any name entered.



What all of this means is that you have successfully deployed your service more concretely; you have invoked the service in a REST (Representational State Transfer) manner.

As mentioned, the previous approach can be considered as one of the easiest ways to write and deploy services in Axis2. Though the sample we chose was quite simple, that is not to say you can't write very complicated services in this manner. However, the only limitation is that you cannot have the service class in any package other than the default package. If you want to have the service class in your own package structure, you will have to follow a slightly different path, which we will discuss soon, later in this chapter.

Considering the POJO support provided by Axis2, you should know that you can deploy both annotated Java classes and non annotated (plain) Java classes. It should be mentioned here that Axis2 has support for the JSR 181 annotation specification. If you are not familiar with Java annotations, it would suffice to say that annotations are a mechanism that provides metadata when using POJOs. A simple Java class with basic JSR 181 annotation can be written as follows, and it should be noted that it is possible to provide large amounts of data and create complicated applications using this feature.

```
import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.WebParam;

@WebService (targetNamespace = "http://sample.org/helloWorld", name =
"HelloWorld")
public class HelloWorld {
```

```
    @WebMethod (action = "urn:sayHello" ,operationName = "sayHello")
    public String sayHello(@WebParam (partName = "name") String name)
{
        return "Hello " + name;
    }
}
```

Annotation is only supported in JDK 1.5 or higher. Also, a lack of expertise in the usage of annotations should not be a concern for you at this point. Your main objective here should be learning the Axis2 concepts, after which, applying annotations should be pretty straightforward.

POJOs with packages

As we discussed in the previous section, when deploying a POJO as a single `.class` file, you cannot have the Java class declared inside a custom package. But if you need it to be so, or if there are several classes that are required for the working of your POJO, and you cannot use the single class POJO approach any longer, then you need to follow the approach we are going to discuss here.

Now let us consider a class like the following:

```
package book.sample
import javax.jws.WebService;
@WebService
public class AddressService {
    public Address getAddress(String name) {
         Address address = new  Address();
         address.setStreet("Street");
         address.setNumber("Number 15");
        return address;
    }
}
```

You can see that we have annotated the `AddressService` class. The corresponding `Address` class is given here:

```
package book.sample
public class Address {
    private String street;
    private String number;
    public String getStreet() {
        return street;
    }
```

```
    public void setStreet(String street) {
        this.street = street;
    }
    public String getNumber() {
        return number;
    }
    public void setNumber(String number) {
        this.number = number;
    }
}
```

Now you have to compile the source code and create a **Java ARchive** file **(JAR)**, that is, a `.jar` file from the `.class` files. There are multiple ways to create a JAR file. You can use Java APIs or you can use some of the third-party archive tools to accomplish this task.

Next, you will need to edit your `axis2.xml` to add a deployer to handle `.jar` files. You can do this by adding the following entry to the `axis2.xml` file in:

```
<TOMCAT_HOME>/webapps/axis2/WEB-INF/conf/axis2.xml
<!--.jar handler-->
<deployer extension=".jar" directory="pojo" class="org.apache.axis2.
deployment.POJODeployer"/>
```

The next step is to drop the `.jar` file into `<TOMCAT_HOME>/webapps/axis2/WEB-INF/pojo`.

A `.jar` file will, more often than not, contain more than one `.class` file in it. So there should be a way to identify the service class (or classes) in it. That is why we have annotated the POJO class with @WebService. Provided this annotation is applied, Axis2 will identify that class as the service implementation class and expose it as a web service.

To see what has happened (remember to restart Tomcat if it is already running), go to `http://localhost:8080/axis2/services/listServices`, where you should find a service called `AddressService` listed.

If you have turned hot deployment on, you can change either the `.class` files or the `.jar` file and redeploy it. Axis2 will pick up the changes you made.

Deploying services using a service

The POJO deployment method, though the simplest, cannot be considered the most suitable or the most flexible method when considering the different options available in the code first approach. As we already discussed, when you have a service with its own package names, the POJO-based deployment becomes a little complicated. As a solution to this, Axis2 has introduced yet another deployment mechanism called **archive-based deployment**. It should also be noted here that the most recommended approach for deploying a service in Axis2 is not the POJO approach, it is the archive-based deployment approach. However, you can feel free to use the POJO approach as the initial stage of the service development, as it can be very convenient. Now let's see how you can go about creating a service archive file from the `HelloWorld.java` class.

Writing the services.xml file

When deploying your service as an archive file, in order for it to be a valid service in Axis2, you need to have a service deployment descriptor document named `services.xml` inside the service archive. This service deployment descriptor will tell the deployment module how to configure and deploy the service. Writing a `services.xml` for the service you developed previously is very simple and pretty straightforward.

There are a few things that you have to keep in mind when writing files such as `services.xml`:

- The fully qualified class name of the service implementation class (for example, if you have a class called `MyService` inside the `foo.bar` package, the fully qualified name becomes `foo.bar.MyService`)
- Message receiver or receivers that you are going to use

Axis2 has a set of inbuilt message receivers. Some of these can only handle XML-in/XML-out scenarios and are called **RawXML message receivers**. Also, there are message receivers that can handle any kind of JavaBeans/simple Java types/XML and these are called **RPC message receivers**. According to the earlier sample code, it may be obvious to you that you cannot use any of the RawXML message receivers for this particular service. So, RPC message receivers would be the order of the day here.

There are different methods you can use when writing a `services.xml`, and the one you choose may vary depending on the way you specify the message receivers, operation overriding, and so on. Let's start with a very basic `services.xml` to get an understanding of the concepts. The `services.xml` corresponding to the given service can be written as follows:

```
<service name="HelloService">
    <description>
        This is my first service, which says "Hello!"
    </description>
    <parameter name="ServiceClass">HelloWorld</parameter>
    <operation name="sayHello">
        <messageReceiver
            class="org.apache.axis2.rpc.receivers.
RPCMessageReceiver"/>
    </operation>
</service>
```

As you can see, I have highlighted a few lines in the given XML snippet. Those are the important XML tags that you need to remember to include when writing a `services.xml`.

Service implementation class

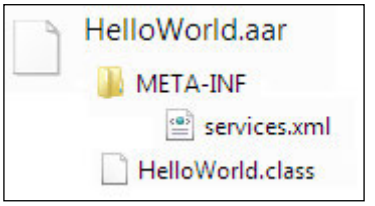
To specify the corresponding service implementation class, you need to add a parameter with the name `ServiceClass` and the value of that parameter should be the fully qualified name of the service implementation class. In this case, it is just `HelloWorld` (class is declared in the default package).

Specifying the message receiver

There are a few ways to specify the message receiver for a given service. One of them is to add an `operation` tag with the actual Java method name and include the message receiver inside this `<operation>` tag. The `services.xml`, mentioned earlier, has followed this approach. As you can see, the service implementation class has a method called `sayHello`, and the `services.xml` has an operation tag with the same name. Inside the `operation` element, we have added the message receiver element.

Creating a service archive file

The next step is to create a service archive file containing the compiled bytecode of the service implementation class and the `services.xml`. The inner folder structure of a service archive file should look like the following screenshot:



Deploying the service is just a matter of dropping the service archive file into the `axis2/WEB-INF/services` directory in your aixs2 server's repository. In addition, you can also upload your service archive file using Axis2's administration console. Irrespective of the way you deploy your service, the name of the service becomes `HelloService`.

Different ways of specifying message receivers

As mentioned earlier, there are several ways of specifying message receivers for a given service:

- Specify the message receiver at the operation level for each operation
- Specify all the message receivers at the service level for the whole service
- Specify a service level message receiver and override it for individual operations as and when required

Specifying the message receiver at the operation level

The example we discussed earlier used this approach, where the message receiver was specified at the operation level. The advantage of this approach is that you can have different message receivers for different operations; for example, you can have an operation which uses a simple Java class and some other operations using `OMElements`. In that case, you can configure one operation with the RPC message receiver and another with the XML message receiver.

Specifying message receivers at the service level for the whole service

Think of a scenario where you have a large number of operations to be published in the `services.xml` file. In this case, adding message receivers for each and every operation seems like a headache. However, if you can specify a message receiver for the whole service, it would make the service author's job easier and would simplify the `services.xml` as well.

Axis2 has inbuilt support for all of the eight **Message Exchange Patterns** (**MEPs**) defined in W.S.D.L. 2.0. In a `services.xml`, you can specify an MEP and the corresponding message receiver, and then, depending on the MEP the operation belongs to, Axis2 will automatically pick up the message receiver and set it to the operation.

Inside the `operation` tag, you can add an attribute to specify the MEP of the operation as follows:

```
<operation name="sayHello" mep="http://www.w3.org/2004/08/wsdl/in-out"
/>
```

An example of defining service level message receivers for a given service is shown here:

```
<service>
    <Description>
        This is my first service, which will say Hello.
    </Description>
    <messageReceivers>
        <messageReceiver mep="http://www.w3.org/2006/01/wsdl/in-only"
class="org.apache.axis2.rpc.receivers.RPCInOnlyMessageReceiver"/>
        <messageReceiver mep="http://www.w3.org/2006/01/wsdl/in-out"
class="org.apache.axis2.rpc.receivers.RPCMessageReceiver"/>
    </messageReceivers>
<parameter name="ServiceClass" locked="false">HelloWorld</parameter>
<operation name="sayHello" mep="http://www.w3.org/2004/08/wsdl/in-out"
/>
</service>
```

According to the `services.xml`, `org.apache.axis2.rpc.RPCMessageReceiver` is the message receiver for all of the in-out operations (any operation that belongs to the in-out MEP will assign this message receiver as its message receiver) in the service. Whereas, the service level message receiver for an in-only MEP is `RPCInOnlyMessageReceiver`. If you redeploy the `HelloWorld` service with the new `services.xml`, you will definitely get the same result when you invoke the service again.

Specifying the service level message receiver and overriding them through operations

There may be an instance where a service author wants to use different message receivers for some of the operations while he/she has already defined service level message receivers. The overriding of a service level message receiver by an operation can be easily achieved by just adding a message receiver element to the operation that you want to assign a different message receiver to. A sample `services.xml` that follows this technique is shown next:

```
<service>
    <Description>
        This is my first service, which will say Hello.
    </Description>
    <messageReceivers>
        <messageReceiver mep="http://www.w3.org/2006/01/wsdl/in-only"
                         class="org.apache.axis2.rpc.receivers.
RPCInOnlyMessageReceiver"/>
        <messageReceiver mep="http://www.w3.org/2006/01/wsdl/in-out"
                         class="org.apache.axis2.rpc.receivers.
RPCMessageReceiver"/>
    </messageReceivers>
    <parameter name="ServiceClass" locked="false">HelloWorld</
parameter>
    <operation name="sayHello">
        <messageReceiver class="org.apache.axis2.receivers.
RawXMLINOutMessageReceiver"/>
    </operation>
</service>
```

Operation `sayHello` uses a different message receiver than its service level message receivers.

All the public methods in the service implementation class are exposed whether you specify them in the `services.xml` or not. Axis2 calculates the MEP of an operation by checking its corresponding Java method. If the method's return type is void, then the MEP will be in-only, or else it will be in-out. Depending on the MEP, the correct message receiver will be set.

Note: If you do not want to expose some of the operations in your service class, you can also do that by adding the following tag:

```
<excludeOperations>
    <operation>op1</operation>
</excludeOperations>
```

Service group and single service

There are many instances where you might want to deploy multiple services (logically related or otherwise) together in a single service archive file. For that, Axis2 has the concept of a service group. Here you can have multiple service implementation classes and only one `services.xml` file to describe all the services. The only difference is that the root element of the `services.xml` is changed to `serviceGroup` instead of `service`. As an example, say you want to deploy two services together in a single service archive file and further assume that their names are `MyService1` and `MyService2` respectively. Then the `services.xml` can be written as follows:

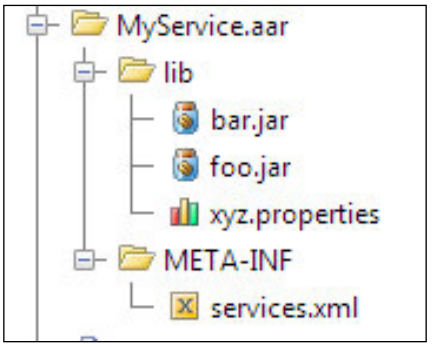
```
<serviceGroup>
    <service name="MyService1">
       ......................
    </service>
    <service name="MyService2">
       ......................
    </service>
</serviceGroup>
```

Comparing this `services.xml` with the `services.xml` from the earlier example (`HelloWorld`), the only difference in the service elements is that here the service element has an additional attribute called `name`. If you want to have multiple service elements in the `services.xml` file, then it is compulsory to have a name attribute in each and every service element.

Adding third-party resources

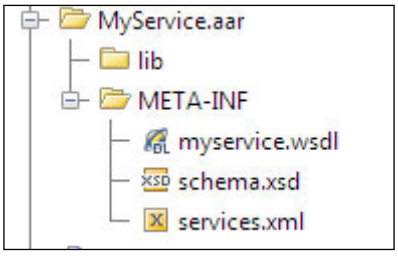
There are many instances where you will want to use third-party libraries in your web service. Also, by now, you should know that each service and module in Axis2 is isolated. So each service gets its own class loader. When you have third-party libraries that you want to include with your web service, Axis2 has a mechanism to do just that. This is done by simply creating a folder named `lib` inside the service archive file and dropping the libraries or resources inside it.

Imagine you have a service that needs to use libraries named `foo.jar` and `bar.jar` and a resource named `xyz.properties` in your service. Your service archive file should then look like the following screenshot:



Service WSDL and schemas

When you start writing relatively more complex applications, you will feel the need for having a WSDL document and schema files inside the service archive file. When considering enterprise level applications, you cannot let clients rely on an auto-generated WSDL document. In order to have a solid and well defined WSDL document, it is necessary to deploy the service along with the WSDL document file and the corresponding XML Schema files. Therefore, when you want to add a WSDL document file, the place to put it would be inside the META-INF directory of a service archive. The only thing you have to remember when doing so is that the service name given in the `services.xml` and the service name defined in the WSDL document should be the same. A service archive file with the included WSDL file will have the following structure:



When you have a WSDL file and `services.xml`, the relationship between them will be as follows:

WSDL file

```
<wsdl:definitions xmlns:wsdl=http://schemas.xmlsoap.org/wsdl/ …..>
<wsdl:types>
<xs:schema targetNamespace="http://org.apache.axis2" … >
</xs:schema>
</wsdl:types>
<wsdl:portType name="MyPort">
</wsdl:portType>
<wsdl:binding name="MyBiding" type="tns: MyPort">
</wsdl:binding>

<wsdl:service name="MyService">
     <wsdl:port name=" MyPort " binding="tns: MyBiding ">
        <soap:address location="http://127.0.0.1:8080/axis2/services/
MyService " />
    </wsdl:port>
</wsdl:service>
Services.xml
<service name="MyService">
   ...
</service>
```

Contract first approach – starting from the WSDL

The easiest and a more suitable way of creating a web service is to start from the WSDL document. This is the method most followed when it comes to most of the enterprise level applications. As enterprise level applications usually have well defined business scenarios and corresponding business contracts, which can be presented in the form of a WSDL document, it makes sense to start from there. The point to note here is that once the client and the producer (service) have the WSDL document, it acts as a contract according to which the development should take place.

Axis2 has inbuilt support for the generation of service and client code, once you have the WSDL document. So in this case, as a service author, you only need to do the following:

1. Generate the service code (service skeleton).
2. Fill in the service skeleton according to the business logic.
3. Run the generated Ant build file.
4. Deploy the service archive file created by Ant into your application server where the Axis2 is running.

Generating code

Axis2 comes with a set of tools and IDE plugins for code generation (WSDL2Code) in order to make web services' development work easier. So you can choose from any of these code generation tools to generate the service skeleton. In addition, there is a set of databinding frameworks that are supported, and out of which, you can select one, depending on your requirements. To name a few, you can use XMLBeans, ADB, or any other available databinding frameworks such as JiBX, JaxMe, and others.

When you generate server-side code, it creates the following artifacts:

- Service skeleton class
- Message receivers (most of the time one or two)
- `services.xml`
- `services.wsdl`
- Ant build file (`build.xml`)

Filling in the service skeleton

Axis2 generates a service skeleton class with methods that can throw `UnsupportedOperation`. All that we have to do is implement the service skeleton class according to the business logic you want to provide.

Running the ant build file

After completing the service skeleton, the next step is to create a service archive file using the modified generated code. To make this job simple, Axis2 generates an Ant build file to create the service archive file for you. What you have to do is open a command line console and go to the folder where you output the generated code and then type `ant build` in the console to run the Ant build file. The service archive will now be created for you.

In this chapter, we only skimmed through the contract first approach. You will learn more about code generation and databinding in detail later in this book. We will also be covering a number of samples as well as the available tools.

Summary

Turning a Java class into a web service is very straightforward in Axis2. Once you know how to write a `services.xml` file correctly, you can create more complex applications rather than just POJOs. Deploying a service is just a matter of creating a service archive file and dropping it into the `services` directory in the Axis2 repository. WSDL's first approach is the easiest way of creating a service since Axis2 has inbuilt support for code generation and it has a set of tools to make the job even easier.

In this chapter, we discussed how to convert a simple class file into a web service and how to access that in a REST manner. We discussed what happens when the service becomes a little complex by adding a package name. We also discussed how to use the archive-based deployment mechanism. Finally, we briefly discussed the contract first approach.

In the next chapter, we will discuss how to extend Axis2's core functionality and provide additional service quality using the Axis2 module.

8
Writing an Axis2 Module

Web services are gaining a lot of popularity in the industry and have become one of the major enabler for application integration. In addition, due to the flexibility and advantages of using web services, everyone is trying to enable web service support for their applications. As a result, web service frameworks need to support new and more custom requirements. As we have already discussed in the previous chapters, one of the major goals of a web service framework is to deliver incoming messages into the target service. However, just delivering the message to the service is not enough; today's applications are required to have reliability, security, transaction, and other quality services.

Due to the popularity of web services, standard bodies are producing new web service standards, and it is hard to support those new standards if the web service framework is not flexible enough. From the very beginnings of Axis2, flexibility and extensibility were the two main design considerations. The idea of Axis2 modules is to extend the core functionality of the system without performing any changes to the core system. For example, Axis2 supports reliability and security as two separate modules, and the core engine is fully independent of those two qualities of service modules.

In this chapter, we will discuss the power of Axis2 modules and how to use them to extend Axis2 to support for your own requirements. In particular, we will discuss the following items:

- Brief history of the Axis2 module
- Introducing module concept
- Structure of the module
- Module configuration file (`module.xml`)
- Optional module implementation class
- Steps to writing a `module.xml` file
- Deploying and engaging a module
- Brief overview of the WS-Policy and its usage in modules

In our approach, we will be using code sample to help us understand the concepts better.

Brief history of the Axis2 module

Looking back at the history of Apache Web Services, the Handler concept can be considered as one of the most useful and interesting ideas. Due to the importance and flexibility of the handler concept, Axis2 has also introduced it into its architecture. Notably, there are some major differences in the way you deploy handlers in Axis1 and Axis2. In Axis1, adding a handler requires you to perform global configuration changes and for an end user, this process may become a little complex. In contrast, Axis2 provides an easy way to deploy handlers. Moreover, in Axis2, deploying a handler is similar to deploying a service and does not require global configuration changes.

At the design stage of Axis2, one of the key considerations was to have a mechanism to extend the core functionality without doing much. One of the main reasons behind the design decision was due to the lesson learned from supporting WS reliable messaging in Axis1. The process of supporting reliable messaging in Axis1 involved a considerable amount of work, and part of the reason behind the complex process was due to the limited extensibility of Axis1 architecture. Therefore, learning from a session in Axis1, Axis2 introduced a very convenient and flexible way of extending the core functionality and providing the quality of services. This particular mechanism is known as the module concept.

Module concept

In *Chapter 4*, *Execution Chain*, we introduced and discussed the handler concepts and how to use handlers. As we discussed there, one of the main ideas behind a handler is to intercept the message flow and execute specific logic. In Axis2, the concept of a module is to provide a very convenient way of deploying service extension. We can also consider a **module** as a collection of handlers and required resources to run the handlers (for example, third-party libraries). One can also consider a module as an implementation of a web service standard specification. As an illustration, Apache Sandesha is an implementation of WS-RM specification. Apache Rampart is an implementation of WS-security; likewise, in a general module, is an implementation of a web service specification. One of the most important features and aspects of the Axis2 module is that it provides a very easy way to extend the core functionality and provide better customization of the framework to suit complex business requirements. A simple example would be to write a module to log all the incoming messages or to count the number of messages, if requested.

Module structure

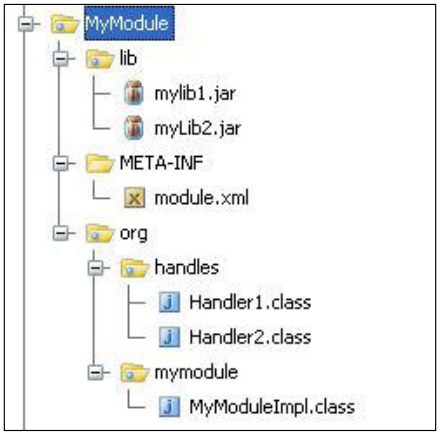
As mentioned a few times before in previous chapters, Axis1 is one of the most popular web service frameworks and it provides very good support for most of the web service standards. However, when it comes to new and complex specifications, there is a significant amount of work we need to do to achieve our goals. The problem becomes further complicated when the work we are going to do involves handlers, configuration, and third-party libraries. To overcome this issue, the Axis2 module concept and its structure can be considered as a good candidate. As we discussed in the deployment section, both Axis2 services and modules can be deployed as archive files. Inside any archive file, we can have configuration files, resources, and the other things that the module author would like to have.

It should be noted here that we have hot deployment and hot update support for the service; in other words, you can add a service when the system is up and running. However, unfortunately, we cannot deploy new modules when the system is running. You can still deploy modules, but Axis2 will not make the changes to the runtime system (we can drop them into the directory but Axis2 will not recognize that), so we will not use hot deployment or hot update. The main reason behind this is that unlike services, modules tend to change the system configurations, so performing system changes at the runtime to an enterprise-level application cannot be considered a good thing at all.

As we discussed earlier, adding a handler into Axis1 involves global configuration changes and, obviously, system restart. In contrast, when it comes to Axis2, we can add handlers using modules without doing any global level changes. There are instances where you need to do global configuration changes, which is a very rare situation and you only need to do so if you are trying to add new phases and change the phase orders. You can change the handler chain at the runtime without downer-starting the system.

As mentioned earlier, changing the handler chain or any global configuration at the runtime cannot be considered a good habit. This is because in a production environment, changing runtime data may affect the whole system. However, at the deployment and testing time this comes in handy.

The structure of a module archive file is almost identical to that of a service archive file, except for the name of the configuration file. We know that for a service archive file to be a valid one, it is required to have a `services.xml`. In the same way, for a module to be a valid module archive, it has to have a `module.xml` file inside the `META-INF` directory of the archive. A typical module archive file will take the structure shown in the following screenshot. We will discuss each of the items in detail and create our own module in this chapter as well.



Module configuration file (module.xml)

As we have already discussed, the module archive file is a self-contained and self-described file. In other words, it has to have all the configuration required to be a valid and useful module. Needless to say, that is the beauty of a self-contained package. The Module configuration file or `module.xml` file is the configuration file that Axis2 can understand to do the necessary work.

A simple `module.xml` file has one or more handlers. In contrast, when it comes to complex modules, we can have some other configurations (for example, WS policies, phase rules) in a `module.xml`. First, let's look at the available types of configurations in a `module.xml`. For our analysis, we will use a `module.xml` of a module that counts all the incoming and outgoing messages. We will be discussing all the important items in detail and provide a brief description for the other items:

- Handlers alone with phase rules
- Parameters
- Description about module
- Module implementation class
- WS-Policy
- End points

Handlers and phase rules

As we discussed, a module is a collection of handlers, so a module could have one or more handlers. Irrespective of the number of handlers in a module, `module.xml` provides a convenient way to specify handlers. Most importantly, `module.xml` can be used to provide enough configuration options to add a handler into the system and specify the exact location where the module author would like to see the handler running. In *Chapter 4*, *Execution Chain*, we learnt more about phase rules as a mechanism to tell Axis2 to put handlers into a particular location in the execution chain, so now it is time to look at them with an example.

Before learning how to write phase rules and specifying handlers in a `module.xml`, let's look at how to write a handler. There are two ways to write a handler in Axis2:

- Implement the `org.apache.axis2.engine.Handler` interface
- Extend the `org.apache.axis2.handlers.AbstractHandler` abstract class

In this chapter, we are going to write a simple application to provide a better understanding of the module. Furthermore, to make the sample application easier, we are going to ignore some of the difficulties of the Handler API. In our approach, we will extend the `AbstractHandler`. When we extend the abstract class, we only need to implement one method called invoke. So the following sample code will illustrate how to implement the `invoke` method:

```
public class IncomingCounterHandler extends AbstractHandler implements
CounterConstants {
public InvocationResponse invoke(MessageContext messageContext) throws
AxisFault {
        //get the counter property from the configuration context
 ConfigurationContext configurationContext = messageContext.
getConfigurationContext();
        Integer count =
                (Integer) configurationContext.getProperty(INCOMING_
MESSAGE_COUNT_KEY);
        //increment the counter
        count = Integer.valueOf(count.intValue() + 1 + «»);
        //set the new count back to the configuration context
        configurationContext.setProperty(INCOMING_MESSAGE_COUNT_KEY,
count);
        //print it out
        System.out.println(«The incoming message count is now « +
count);
        return InvocationResponse.CONTINUE;
    }
}
```

As we can see, the method takes `MessageContext` as a method parameter and returns `InvocationResponse` as the response. You can implement the method as follows:

1. First get the `configurationContext` from the `messageContext`.
2. Get the property value specified by the property name.
3. Then increase the value by one.
4. Next set it back to `configurationContext`.
5. In general, inside the `invoke` method, as a module author, you have to do all the logic processing, and depending on the result you get, we can decide whether you let `AxisEngine` continue, suspend, or abort. Depending on your decision, you can return to one of the three following allowed return types:
    - `InvocationResponse.CONTINUE`

      Give the signal to continue the message
    - `InvocationResponse.SUSPEND`

      The message cannot continue as some of the conditions are not satisfied yet, so you need to pause the execution and wait.
    - `InvocationResponse.ABORT`

      Something has gone wrong, therefore you need to drop the message and let the initiator know about it
6. The message cannot continue as some of the conditions are not satisfied yet, so you need to pause the execution and wait.
7. `InvocationResponse.ABORT`.
8. Something has gone wrong, therefore you need to drop the message and let the initiator know about it.

The corresponding `CounterConstants` class a just a collection of constants and will look as follows:

```
public interface CounterConstants {
    String INCOMING_MESSAGE_COUNT_KEY = "incoming-message-count";
    String OUTGOING_MESSAGE_COUNT_KEY = "outgoing-message-count";
    String  COUNT_FILE_NAME_PREFIX = "count_record";
}
```

As we already mentioned, the sample module we are going to implement is for counting the number of request coming into the system and the number of messages going out from the system. So far, we have only written the incoming message counter and we need to write the outgoing message counter as well, and the implementation of the out message count hander will look like the following:

```
public class OutgoingCounterHandler extends AbstractHandler implements
CounterConstants {
    public InvocationResponse invoke(MessageContext messageContext)
throws AxisFault {
        //get the counter property from the configuration context
        ConfigurationContext configurationContext = messageContext.
getConfigurationContext();
        Integer count =
                (Integer) configurationContext.getProperty(OUTGOING_
MESSAGE_COUNT_KEY);
        //increment the counter
        count = Integer.valueOf(count.intValue() + 1 + «»);
        //set it back to the configuration
        configurationContext.setProperty(OUTGOING_MESSAGE_COUNT_KEY,
count);
        //print it out
        System.out.println(«The outgoing message count is now « +
count);
        return InvocationResponse.CONTINUE;
    }
}
```

The implementation logic will be exactly the same as the incoming handler processing, except for the property name used in two places.

Module implementation class

When we work with enterprise-level applications, it is obvious that we have to initialize various settings such as database connections, thread pools, reading property, and so on. Therefore, you should have a place to put that logic in your module. We know that handlers run only when a request comes into the system but not at the system initialization time. The module implementation class provides a way to achieve system initialization logic as well as system shutdown time processing. As we mentioned earlier, module implementation class is optional. A very good example of a module that does not have a module implementation class is the Axis2 addressing module. However, to understand the concept clearly in our example application, we will implement a module implementation class, as shown below:

```
public class CounterModule implements Module, CounterConstants {
    private static final String COUNTS_COMMENT = "Counts";
    private static final String TIMESTAMP_FORMAT = "yyMMddHHmmss";
    private static final String FILE_SUFFIX = ".properties";

    public void init(ConfigurationContext configurationContext,
```

```
                        AxisModule axisModule) throws AxisFault {
        //initialize our counters
        System.out.println("inside the init : module");
        initCounter(configurationContext, INCOMING_MESSAGE_COUNT_KEY);
        initCounter(configurationContext, OUTGOING_MESSAGE_COUNT_KEY);
    }

    private void initCounter(ConfigurationContext
configurationContext,
                             String key) {
        Integer count = (Integer) configurationContext.
getProperty(key);
        if (count == null) {
            configurationContext.setProperty(key, Integer.
valueOf("0"));
        }
    }

    public void engageNotify(AxisDescription axisDescription) throws
AxisFault {
        System.out.println("inside the engageNotify " +
axisDescription);
    }

    public boolean canSupportAssertion(Assertion assertion) {
        //returns whether policy assertions can be supported
        return false;
    }

    public void applyPolicy(Policy policy,
                            AxisDescription axisDescription) throws
AxisFault {
        // Configuure using the passed in policy!
    }

    public void shutdown(ConfigurationContext configurationContext)
throws AxisFault {
        //do cleanup -  in this case we'll write the values of the
counters to a file
        try {
            SimpleDateFormat format = new SimpleDateFormat(TIMESTAMP_
FORMAT);
            File countFile = new File(COUNT_FILE_NAME_PREFIX + format.
format(new Date()) + FILE_SUFFIX);
            if (!countFile.exists()) {
                countFile.createNewFile();
            }
```

```
            Properties props = new Properties();
            props.setProperty(INCOMING_MESSAGE_COUNT_KEY,
                    configurationContext.getProperty(INCOMING_MESSAGE_
COUNT_KEY).toString());
            props.setProperty(OUTGOING_MESSAGE_COUNT_KEY,
                    configurationContext.getProperty(OUTGOING_MESSAGE_
COUNT_KEY).toString());
            //write to a file
            props.store(new FileOutputStream(countFile), COUNTS_
COMMENT);
        } catch (IOException e) {
            //if we have exceptions we'll just print a message and let
it go
            System.out.println("Saving counts failed! Error is " +
e.getMessage());
        }
    }
}
```

As we can see, there are a number of methods in the previous module implementation class. However, notably not all of them are in the module interface. The module interface has only the following methods, but here we have some other methods for supporting our counter module-related stuff:

- `init`
- `engageNotify`
- `applyPolicy`
- `shutdown`

At the system startup time, the `init` method will be called, and at that time, the module can perform various initialization tasks. In our sample module, we have initialized both in-counter and out-counter.

When we engage this particular module to the whole system, to a service, or to an operation, the `engagNotify` method will be called. At that time, a module can decide whether the module can allow this engagement or not; say for an example, we try to engage the security module to a service, and at that time, the module finds out that there is a conflict in the encryption algorithm. In this case, the module will not be able to engage and the module throws an exception and Axis2 will not engage the module. In this example, we will do nothing inside the `engageNotify` method.

As you might already know, WS-policy is one of the key standards and plays a major role in the web service configuration. When you engage a particular module to a service, the module policy should be applied to the service and should be visible when we view the WSDL of that service. So the `applyPolicy` method sets the module policy to corresponding services or operations when we engage the module. In this particular example, we do not have any policy associated with the module, so we do not need to worry about this method as well.

As we discussed in the `init` method, the method `shutdown` will be called when the system has to shut down. So if we want to do any kind of processing at that time, we can add this logic into that particular method. In our example, for demonstration purposes, we have added code to store the counter values in a file.

Writing the module.xml file

So far, we have written two handlers and module implementation classes. Now, the only remaining thing to do is to write the module descriptor file. When writing `module.xml`, we have to use phase rules to specify the location of handlers and we have discussed phase rules before (it is time to refresh our mind about the phase rule). The most simple `module.xml` file for our module is shown here:

```
<module name="counterModule" class="org.apache.axis2.sample.module.
request.CounterModule">
  <Description>
    Counts the incoming and outgoing messages
  </Description>
  <InFlow>
    <handler name="IncomingMessageCountHandler"
                class="org.apache.axis2.sample.module.request.
IncomingCounterHandler">
      <order phase="Transport" after="RequestURIBasedDispatcher"
                 before="SOAPActionBasedDispatcher"/>
     </handler>
  </InFlow>
  <OutFlow>
    <handler name="OutgoingMessageCountHandler"
                class="org.apache.axis2.sample.module.request.
OutgoingCounterHandler">
      <order phase="MessageOut"/>
    </handler>
  </OutFlow>
</module>
```

In the file `module.xml`, we have specified the description of the module as "Counts the incoming and outgoing messages"; in the meantime, it has specified the two handlers we implemented earlier with phase rules.

As you can see, we try to put our incoming message counter into the transport phase and the exact location is after `RequestURIBasedDispatcher` and before `SOAPActionBasedDispatcher`. If you look at the default `axis2.xml` file, you can see the two handlers in the `inFlow`. Meanwhile, in the outgoing message, a counter is added to the message-out phase and this does not specify the exact location.

If you look carefully, you can see in the root element that there is an attribute called "class", which specifies the module interface class. We need to remember that this attribute is an optional one and some modules may or may not have this attribute.

There are instances where the module author needs to use a new phase for the module. In such a situation, one can edit the `axis2.xml` and specify the new phase(s) one wants. Once specified, phases can be used inside the `module.xml`.

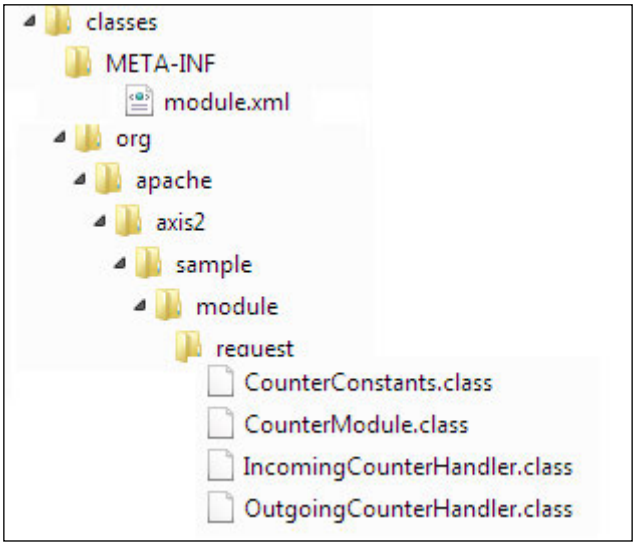
In the `module.xml` file, we have not discussed how to add the fault handlers, as we already know, there are two flows for fault processing: `InFaultFlow` and `OutFaultFlow`. The `InFaultFlow` is executed when there is an incoming fault. Similarly, `OutFaultFlow` is executed when there is an outgoing message. If you want to add handler(s) to any of those phases, you can do that just like you did with the other two flows.

Deploying and engaging the module

Now we have written everything you need for a valid module, the only remaining thing is to create the module archive file and deploy it to the repository. First we compile our source code and then, as we know, it creates `.class` files.

Assuming `org.apache.axis2.sample.module.request` is the package name of our source file, we can find all the .class files under `classes/org/apache/axis2/sample/module/request`.

Now create a directory called `META-INF` under the classes directory and copy the `module.xml` file into it. Then our `classes` directory should look like the following screenshot:



Now create a ZIP file from the classes directory and rename the ZIP file into the counter-`module.mar`.

Deploying the module is as simple as dropping the file into the `TOMCAT_HOME/webapps/axis2/WEB-INF/modules` or `repository/modules` directory. In this case, let's focus on deploying the module in Tomcat or your favorite web application server. As we know, Axis2 does not support module hot deployment, so just dropping the module will not make any changes to the runtime system. To deploy our module and apply the necessary changes to runtime, we need to restart Axis2; in other words, we need to restart the application server.

It should be noted that just deploying a module does not add its handlers into the handler chain; to add handlers into the system, it is necessary to engage the module. Now, let's see how we engage a module in Axis2. For this purpose, we can use the Axis2 web administration console. Moreover, to make the story simple, let's try to engage the module to all the services in the system using the web administration console (we can engage to a single service also using the administration console). We can engage the module to the system by carrying out the following steps:

1. Go to `http://localhost:8080/axis2/`.
2. Click on the `Administration` tab. This will open up a new page and ask for the username and password.

3. Enter `admin` as the username and `axis2` as the password. This will open up the administration console.
4. After that, click on the **Available Modules** in the side navigation menu on the left-hand side. There we will find our Counter module as

   `counterModule: Counts the incoming and outgoing messages.`
5. Now go to `Engage Module\For all Services` and this will open up a new page with a drop-down menu.
6. Select **counterModule** from that and click engage.
7. Now we should see the message **counterModule module engaged globally successfully**.
8. Go to **Global Chains** and you will be able to see the `IncomingMessageCountHandler` handler in the transport phase between `RequestURIBasedDispatcher` and `SOAPActionBasedDispatcher`.

Now, this simply tells us that we have engaged the module successfully and we have added the handler into the correct phases to invoke the version server, enter the following into your browser:

```
http://localhost:8080/axis2/services/Version/getVersion
```

Then, in the console, you will see the following:

```
The incoming message count is now 1
The outgoing message count is now 1
```

This simply tells us that the request has gone though the incoming counter handler and the response has gone though the outgoing counter handler. Now, let's invoke the service one more time and see what we get in the console. We will see the following:

```
The incoming message count is now 2
The outgoing message count is now 2
```

Now we know how to write a very simple module, deploy it, and engage it. As an exercise, we can change the `module.xml` and see what happens. Also, we can change the phase rules and see whether it puts the handlers into the correct locations. In the meantime, we can restart Tomcat, without engaging the module. Here, we should not see any output in the console. This will help us to understand that handlers are invoked only when we engage the module.

There are other ways of engaging a module, for example, to engage a module globally, a user can edit `axis2.xml` and add the `<module ref="modulename"/>` element. Moreover, if a user wants to engage it to particular services, he/she can do that by editing the `services.xml` and adding the same tag mentioned earlier.

Advanced module.xml

Here, we looked at a very simple `module.xml`, but when it comes to very complex applications, we need to have many more configurations in a `module.xml` file. We might need to have parameters, WS-Policy, and endpoints. Now let's look at them one at a time in brief.

Parameters

Adding a parameter here is the same as adding a parameter in `services.xml` or `axis2.xml`. We just need to add the following tag into `module.xml` and Axis2 will do the right thing for us:

```
<parameter name="foo">bar</parameter>
```

We can have any number of parameters in a `module.xml`, and when we want to access the parameter, we can do that by carrying out the following steps:

- First we need to get the `AxisModule`. We can do that either by using the `init` method (Axis2 passes the `AxisModule`) or we need to get the corresponding `AxisModule` from the `ConfigurationContext` (inside the Module implementation class) or from `messageContext` (inside a handler).
- After this, we can ask for the parameter from the `AxisModule`.

WS-Policy

Specifying a WS-Policy element in `module.xml` is one way of configuring a module. We can add WS-Policy element inside a `module.xml`. If we consider the Sandesha2 (reliable message implementation) module, we can find the following policy element:

```
<wsp:Policy xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
                xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/
oasis-200401-wss-wssecurity-utility-1.0.xsd"
                xmlns:sandesha2="http://ws.apache.org/sandesha2/
policy" wsu:Id="RMPolicy">
          <sandesha2:RMAssertion>
                 [REST OF THE FILE]
          </sandesha2:RMAssertion>
</wsp:Policy>
```

Endpoints

In Axis2, an endpoint is an operation of a web service. So, adding an endpoint is the same as adding an operation. The key question is—why do we need to add an endpoint to a module? To understand this, let's assume that we have a module and this module has a set of control operations. The most suitable example is reliable messaging, as it has a number of control messages (for example, create sequence, last message, terminate sequence). Say, we need to invoke a service in a reliable manner. It first needs to set up a sequence with the service. For this, it will send the control message called `createSequence` to the service we need to access. But that service does not have that method, so if we try to send the `createSequence` message without adding the method, Axis2 will throw an exception saying **Unable to dispatch**. Therefore, adding an endpoint will solve that issue. Hence, when we engage Sandesha, it adds a method called `createSequence` to the service at the time we engage the module (or all the services, if we engage that to the whole system). When a request comes, Axis2 will dispatch without having any problem. This operation or the endpoint has its own message receiver to do the right thing.

So, it is obvious that when a module needs control operations exchange, it is required to add endpoint to represent those operations. Adding an endpoint is very simple. What we need to do is add an operation element(s) along with a message receiver and a set of action mapping. To get an idea about that, let's take the Sandesha module as a reference. Its `module.xml` has the following operation tag to add the control operations. Remember, when we have the operation tag in `module.xml`, Axis2 will do all the processing, including creating `AxisOperation` and adding that. As a module author, all we need to do is specify them in the `module.xml` file:

```
<operation name="Sandesha2OperationInOut" mep="http://www.
w3.org/2006/01/wsdl/in-out">
      <messageReceiver class="org.apache.sandesha2.msgreceivers.
RMMessageReceiver"/>
        <!-- namespaces for the 2005-02 spec -->
        <actionMapping>http://schemas.xmlsoap.org/ws/2005/02/rm/
CreateSequence</actionMapping>
        <actionMapping>http://schemas.xmlsoap.org/ws/2005/02/rm/
AckRequested</actionMapping>
        ..................
    </operation>
```

If we look at the Sandesha `module.xml`, we will be able to learn and find out more about writing a `module.xml` file.

Summary

In this chapter, you learned about the importance of the Axis2 module. We also learned that the Axis2 module provides a very flexible way to extend the Axis2 core functionality and provides quality service. Moreover, we discussed the module and related concepts by writing a sample module and demonstrating most of the commonly used configuration settings. In our sample application, we discussed how to write handlers, how to write module implementation classes, and finally, how to put everything together and deploy the module. At the end of the chapter, we learned how to engage a module to Axis2.

In this chapter, we discussed how to write and deploy services and how to write and deploy modules. In the next chapter, we are going to discuss how to invoke a remote service. There, we will discuss how to use Axis2 as a client and invoke services in a number of ways.

9
The Client API

Web service is one of the commonly used approaches for applications integration (composition), when integrating applications; an application can act as either a consumer (client) or a producer (server). A web service framework should be able to deploy services as well as access the services. So far, we have discussed the deployment side of it. Thus, in this chapter, we will focus on the client side and how to use Axis2 to access remote services. In high level, Axis2 runtime does not differentiate between the client side and the server side. Moreover, it uses the same execution framework at the server side as well as the client side. As we already discussed, in the server side, we have services. Thus, to keep the symmetry in the client side as well, Axis2 creates a dummy service when we use the client API. In this chapter, we will look at the various aspects of the client API, with the example use case.

In this chapter, we will discuss most of the commonly used APIs of the Axis2 client, as well as explaining how to use them with sample code. First, we will discuss the various ways of creating the Axis2 client instance. Then, we will discuss the available APIs and finally, we will discuss the more advanced uses of Axis2 client APIs. Particularly, we will be covering the following items in detail:

- The idea of the web service client
- Synchronous and asynchronous service invocation
- Axis2 client APIs, namely, `ServiceClient` and `OperationClient`
- Demonstrate `ServiceClient` APIs with examples
- Demonstrate `OperationClient` APIs with examples

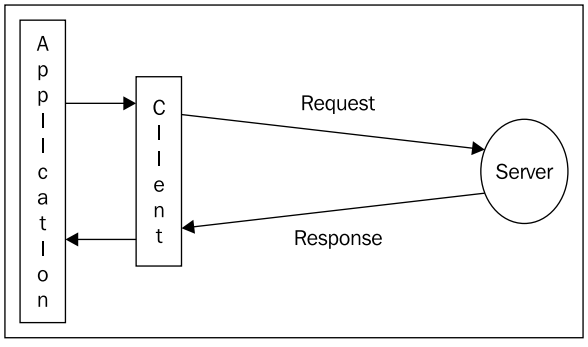
Web service client

What is special about the web service client? How is it different from the usual web page? In the case of accessing a web page by using the web browser, a human interacts with it and clicks the page and navigates. In contrast, when application-application communication occurs, the application has to perform the action that the human performs. In a web service, the client can be considered as the component that lets us click and navigate the service.

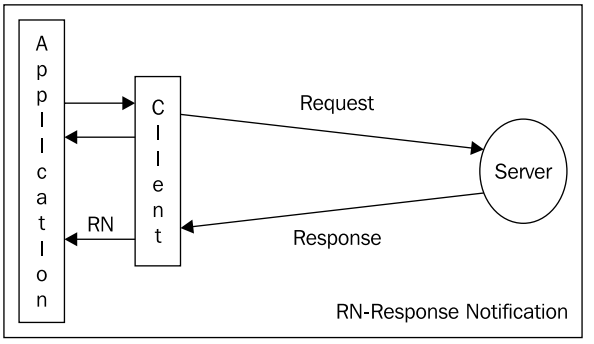
As we have discussed earlier, one of the key advantages of web services is that they provide a way to achieve application-to-application communication. Hence, web service client plays a major role in the application integration. In addition, when it comes to web service clients, there are multiple factors we need to consider. Among them, invoking a service in a non-blocking, asynchronous manner is very useful and a critical feature that most people are looking for. In the past, most of the web service frameworks have only focused on blocking the invocation pattern (rather RPC-centric). Now the trend is totally different. What users are looking for is an asynchronous or non-blocking way when invoking the service—not only in web services, but also in web-based applications—they used techniques such as AJAX to have the asynchronous invocation support. In Axis2, we have two types of asynchronous manner as well, while it has support for WSDL 2.0 basic MEPs.

Blocking and non-blocking invocation

As we just discussed, there are two main ways of utilizing a web service—in a synchronous manner and asynchronous manner (blocking and non-blocking). In case of synchronous, the user invokes the service and waits until he/she gets the response. From a programming point of view, the application blocks until the user gets the response. The following figure illustrates how synchronous web service utilization takes place:



In the case of asynchronous type invocation, the user application (for example, GUI) does not block, so the user can continue working. Axis2's way of utilizing a service in asynchronous is shown in the following figure:



The preceding figure shows how a typical asynchronous message works, where the application uses Axis2 client and asks to send the message. When calling the Axis2 client, the application creates a callback object and passes that to the call. Axis2 uses the callback and returns the control back to the client. Thus, the client does not get blocked. When the reply comes, the Axis2 client uses the callback and notifies it, by which, the application gets to know about the message arrival as well and acts upon it.

Looking into Axis2 client API

In today's world, people do not like and do not want to waste time to get a simple thing done. Thus, user friendliness is one of the key considerations for any product. In Axis2, one of the key designs of the Axis2 client API is to provide asynchronous web service invocation support. Meantime, Axis2 has the key consideration of user friendliness, so combining both together, the new client API is very convenient to work with. To make the web consumer's or the end user's job easier, the Axis2 client API consists of two sub APIs called `ServiceClient` and `OperationClient`, one for average users and the other for advanced users, respectively.

ServiceClient API

As we mentioned earlier, `ServiceClient` is mainly designed for average users or the users who have just moved to the web services field. Nevertheless, it has the notion of interacting with a service and providing all the necessary support to invoke any complex type of web service. Let's assume a calculator as an example of a web service, and further assume that the service has operations such as "add", "deduct", "multiply", and "divide. So the idea of a service client is to provide an API to invoke any of those operations in a very convenient manner.

Available options to create a ServiceClient

There are multiple ways to create a `ServiceClient` instance, and of course, we can choose the most suitable constructors for us. No matter how we create a `ServiceClient`, it is required to have an an Axis2 runtime (`ConfigurationContext`) to invoke the service. Either user explicitly provides an instance of configuration context or Axis2 creates it internally.

Earlier, we discussed `ConfigurationContext` and how to create it. So we can use those techniques to create `ConfigurationContext` when creating a `ServiceClient`, or as we will see, we can create `ServiceClient` with `ConfigurationContext` as null and let Axis2 create `ConfigurationContext` for us.

Type 1: Creating a ServiceClient using its default constructor

The easiest way to create a `ServiceClient` is to use its default constructor, as shown below. In this case, it creates a `ConfigurationContext` using the Axis2 default configuration file, which is available in the Axis2 JAR file. In the meantime, it creates an anonymous (dummy service) service with three operations (three operations to support WSDL 2.0 MEPs). Even though we create a service client this manner, we can use the created client to asses any web service.

```
ServiceClient serviceClient = new ServiceClient();
```

When we are trying to create `serviceClient` inside an Axis2 system (as an example, a handler tries to create a `ServiceClient` to invoke a service, or if service tries to invoke some other service), then it is created using server's `ConfigurationContext`. In this case, all the properties, transports, and modules in the server are accessible to the `ServiceClient`, and that scenario is called as a client running inside a server.

Type 2: Creating a ServiceClient with your own ConfigurationContext

When we want to create a `ServiceClient` with our own configuration data, we can use the following constructor. As we know by now, there are a number of ways to create `ConfigurationContext`. At the same time, there are many instances where we want to create `ServiceClient` with our own `AxisService`, which might have configured with custom QoS (Quality of Service), parameters, WS-Policy, and so on.

```
ServiceClient serviceClient =
  new ServiceClient (configContext, axisService);
```

In this case, either (or both) of the arguments can be null. If both are null, that is obviously equivalent to the default constructor case of `ServiceClient`. If `ConfigurationContext` is null, depending on the location you are trying to create the `ServiceClient`, either a new one will be created using the Axis2 default configuration or a server's `ConfigurationContext` will be used. Similarly, if `AxisService` is null, an anonymous service is created.

Type 3: Creating a dynamic client (client on the fly)

The idea behind a dynamic client is to create a `ServiceClient` on the fly, or simply create a client for a given WSDL at runtime and use the created `ServiceClient` to invoke the corresponding service. When we create the `ServiceClient` in this manner, the corresponding AxisService object is configured according to the WSDL document. We will see the advantages of the dynamic client in the middle of this chapter. The constructor for creating a dynamic client is given as follows:

```
ServiceClient dynamicClient =new ServiceClient(
  configContext,wsdlURL, wsdlServiceName, portName);
```

- `configContext`: `configurationContext` can be null; if it is null, the logic mentioned in Option 2 will be applied.
- `wsldURL`: This argument should not be null as it specifies the URL for the WSDL file.
- `wsdlServiceName`: As you know, the WSDL document might have multiple service elements. So if you want to pick a specific service element, you can pass the QName of that service element. The value of this argument can be null. If it is null, the first one from the service element list will be considered as the service element.
- `portName` : As you know, a service element in a WSDL file might have multiple ports as well. So if you want to select a specific port, you can pass the name of the port as the value of this argument. Then, if the value is null again, the first one from the port list will be selected as the port.

A sample WSDL to understand what the mentioned service elements can be is shown here (the ports are shown here as well):

```
<wsdl:service name="MyService">
     <wsdl:port name="ServicePort1" binding="axis2:ServiceBinding1">
     <soap:address location="http://127.0.0.1:8080/axis2/services/
                            MyService"/>
      </wsdl:port>
      <wsdl:port name="ServicePort2" binding="axis2: ServiceBinding2 ">
```

```
      <soap12:address location="http://127.0.0.1:8080/axis2/services/
                                MyService"/>
       </wsdl:port>
</wsdl:service>
```

ServiceClient with working samples

Now we know the available ways for creating a service client, but there is nothing like code to explain those. So the best way to understand `ServiceClient` API is to write a few real samples. But first, we need to start the Axis2 server and deploy the following sample service in the server. Only after that can we write a useful client to invoke the service. We can create and deploy a service by using the following steps:

1. Open your favorite IDE and write the following Java class or the service implementation class.

   ```
   public class MyService {
     //method which has a return value
     public String echo(String value) {
       return value;
     }
     // does not have a return value
     public void update(int value) {
     System.out.println("value is :" + value);
     }
   }
   ```

   Sample service implementation class has two methods—one has a return value and the other does not.

2. Writing the `services.xml` file for the service, your `services.xml` will look like the following. See the references section to learn about Axis2 deployment.

   ```
   <serviceGroup>
     <service name="MyService">
     <messageReceivers>
       <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-only"
    class="org.apache.axis2.rpc.receivers.RPCInOnlyMessageReceiver"/>
     <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-out"
    class="org.apache.axis2.rpc.receivers.RPCMessageReceiver"/>
     </messageReceivers>
       <parameter name="ServiceClass" locked="false">
         MyService
       </parameter>
     </service>
   </serviceGroup>
   ```

3. Create a service archive file and deploy the service in Axis2 server.

   The first step of creating the service archive file is to compile the source code. Next, go to the compile directory and create a directory called `META-INF`, and copy the created `services.xml` file into it. Next, use any of the archive tools and create a ZIP archive (you can also use the JAR file creation), and name it as `myservice.aar`.

Axis2 comes with IDE tools and Maven plugins that help you to create a service archive file.

Once you have the service archive file, you can deploy it using any of the mechanism we discussed in the *Deployment* section.

Now that we have deployed the service in the server, what remains is to invoke the service. Here we will look at a number of scenarios to understand the concept clearly.

Scenario 1: Invoking a service in a blocking manner (sendReceive())

The most commonly used service invocation pattern is the request response invocation pattern (in-out MEP in WSDL 2.0 terminology). Most of the services are written in a manner such that they have an input(s) and an output. So we need to use a request response invocation pattern. In Axis2, that can be done in two ways—a blocking manner or non-blocking manner. The first sample demonstrates invoking a service in the blocking manner:

1. Create a `ServiceClient` using any of the constructors we mentioned earlier.
2. Create an `OMElement` for payload (first child of SOAP body). In Axis2, XML representation is built on AXIOM. That is why you need to create `OMElement` (See the references section to learn about AXIOM).

   We can use the following code snippet to create the payload that we need to invoke the service. If you look at the service WSDL, then you can understand how to create the request element.

```
public OMElement createPayLoad() {
  OMFactory fac = OMAbstractFactory.getOMFactory();
  OMNamespace omNs = fac.createOMNamespace(
                   "http://ws.apache.org/axis2", "ns1");
  OMElement method = fac.createOMElement("echo", omNs);
  OMElement value = fac.createOMElement("value", omNs);
  value.setText("Hello , my first service utilization");
  method.addChild(value);
```

```
    return method;
}
```

3. Before invoking the service, we need to create a metadata (property bag) object called `Options` and set that to `ServiceClient`. `Options` object contains properties such as target **EPR** (**End Point Reference**), SOAP Action, transport data, and so on, to configure the client side for the service invocation. The following code snippet shows how to create an `Option` object and fill it:

```
ServiceClient sc = new ServiceClient();
// create option object
Options opts = new Options();
//setting target EPR
opts.setTo(new EndpointReference(
    "http://127.0.0.1:8080/axis2/services/MyService"));
//Setting action
opts.setAction("urn:echo");
//setting created option into service client
sc.setOptions(opts);
```

   If we create a `ServiceClient` as a dynamic client, we do not need to worry about creating the `Options` object. It will automatically create everything for us.

4. `sendReceive` is the API for invoking a service in a blocking manner. When we use this API, the program blocks until it gets the response.

```
OMElement res = sc.sendReceive(createPayLoad());
System.out.println(res);
```

   So once we run this sample code, we will get the following output:

```
<ns:echoResponse
  xmlns:ns="http://ws.apache.org/axis2/xsd">
  <return>
    Hello This is my first service
  </return>
</ns:echoResponse>
```

5. `sendReceive` is the API for invoking a service in a blocking manner. Remember that `sc.sendReceive(createPayLoad())` only works if we create `ServiceClient` using either its default constructor or passing a null value as the `AxisService` parameter for any other constructors. When we create `ServiceClient` using our own `AxisService` or when we create it as a dynamic client, we have to use the following method with the correct operation name:

   ```
   sendReceive(QName operation, OMElement elem);
   ```

   For example, if we create `ServiceClient` using the service WSDL as shown, we have to use the operation name as shown in the code:

   ```
   ServiceClient sc = new ServiceClient(null, new URL("http://
   localhost:8080/axis2/services/MyService?wsdl"),null,null);
   sc.sendReceive(new QName("http://ws.apache.org/axis2","echo"),cre
   atePayLoad());
   ```

   As we can see, we only need two lines here to configure and invoke a service.

Scenario 2: Utilizing a service in a non-blocking manner (sendReceiveNonBlocking())

To invoke an in-out MEP in an asynchronous manner, we can use this API. There are two mechanisms of implementing asynchronous type invocation—callback and pooling. Axis2 uses callback mechanism to provide asynchronous support. Therefore, in order to use non-blocking API, we need to implement `org.apache.axis2.client.async.AxisCallback` and pass that object as the method parameter.

In this case, we can follow step 1 through step 3 in Scenario 1, without any modification. But we need to change step 4 to the following:

```
ServiceClient sc = new ServiceClient();
Options opts = new Options();
opts.setTo(new EndpointReference(
  "http://127.0.0.1:8080/axis2/services/MyService"));
opts.setAction("urn:echo");
sc.setOptions(opts);
//creating callbakc object
AxisCallback callback = new AxisCallback() {
  public void onMessage(MessageContext msgContext) {
    System.out.println(
      msgContext.getEnvelope().getBody().getFirstElement());
      complete = true;
}
  public void onFault(MessageContext msgContext) {
    System.err.print(msgContext.getEnvelope().toString());
}
```

```
    public void onError(Exception e) {
      e.printStackTrace();
  }
    public void onComplete() {
      complete = true;
    }
  };
  //invoking the service
  sc.sendReceiveNonBlocking(createPayLoad(), callback);
  System.out.println("-------Invoke the service---------");
  int index = 0;
  //wait till you get the response, in real applications you do not need
  //to do this, since once the response arrive axis2 will notify //
  callback,
  // then you can implement callback to do whatever you want, may be to
  //update GUI
  while (!complete) {
    Thread.sleep(1000);
    index++;
    if (index > 10) {
      throw new AxisFault("Time out");
    }
  }
```

Key differences in Scenario 1 and Scenario 2 are as follows:

- It's necessary to create the `Axiscallback` object
- `sendReceiveNonBlocking` is a void operation

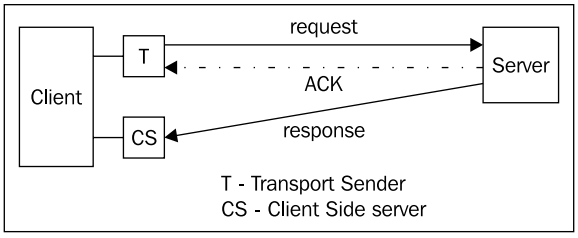
Once you run the code sample, you would get the following output:

```
-------Invoke the service---------
<ns:echoResponse
  xmlns:ns="http://ws.apache.org/axis2/xsd">
<return>
  Hello , my first service utilization
</return>
</ns:echoResponse>
```

Similar to Scenario1, if we create `ServiceClient` using our own `AxisService` or as a dynamic client, then we need to use the following method with a qualified operation `name.sendReceiveNonBlocking`

```
(QName operation, OMElement elem,AxisCallback callback);
```

Scenario 3: Utilizing a service using two transports



In this case, the request goes through one transport and is received through another transport. For example, request can go though HTTP and come through TCP; or thr request can go though one HTTP connection and come via another HTTP connection. To invoke a service in the given manner, we need to have WS-Addressing support. Therefore, we need to engage the addressing module in both client and server sides. In Scenario 2, what we had was only application-level asynchronous support, but here we have transport level asynchronous support.

By changing step 3 in Scenario 2 in the following manner, we can invoke the service via two transports. Let's send request via HTTP and try to get the response via TCP:

```
ServiceClient sc = new ServiceClient();
Options opts = new Options();
opts.setTo(new EndpointReference(
           "http://127.0.0.1:8080/axis2/services/MyService"));

//engaging addressing module
sc.engageModule(new QName("addressing"));
//I need to use separate listener for my response
opts.setUseSeparateListener(true);
// Need to receive via TCP
opts.setTransportInProtocol(Constants.TRANSPORT_TCP);

opts.setAction("urn:echo");
sc.setOptions(opts);
```

It should be noted here that we can send a message using HTPP and receive a message via HTTP as well. In this case, Axis2 will start up a new HTTP listener to receive the incoming message.

Scenario 4: Utilizing an in-only MEP (fireAndForget)

If we want to send data to a server but we worry neither about response nor exceptions, then we could use this API. In WSDL 2.0 terminology, this API is to invoke an in-only MEP. Let's try to invoke the `update` operation in MyService.

1. Create a `ServiceClient`.
2. Create the payload `OMElement`, you need to create a new payload since the method name is different in this case.

```
public OMElement createPayLoad() {
  OMFactory fac = OMAbstractFactory.getOMFactory();
  OMNamespace omNs = fac.createOMNamespace(
                   "http://ws.apache.org/axis2", "ns1");
  OMElement method = fac.createOMElement("update", omNs);
  OMElement value = fac.createOMElement("value", omNs);
  value.setText("10");
  method.addChild(value);

  return method;
}
```

3. When we run the following code, you can see the `value is :10` in the server's console, and we don't get any response or exception even if something goes wrong in the server:

```
ServiceClient sc = new ServiceClient();
Options opts = new Options();
opts.setTo(new EndpointReference(
    "http://127.0.0.1:8080/axis2/services/MyService"));
opts.setAction("urn:update");
sc.setOptions(opts);
sc.fireAndForget(createPayLoad());
```

Replace `setTo` with an invalid one and see whether you are getting any exception; obviously, you will get nothing.

Scenario 5: Utilizing an in-only MEP (sendRobust)

This API is also used to invoke a one way operation, but the only difference from Scenario 4 is if something goes wrong in the server, the client will be informed about that. We can use the Scenario 4 code with minor changes to invoke the service in a robust manner, step 1 and step 2 remain unchanged; however, step 3 needs to change in the following manner:

```
ServiceClient sc = new ServiceClient();
Options opts = new Options();
```

```
opts.setTo(new EndpointReference(
    "http://127.0.0.1:8080/axis2/services/MyService"));
opts.setAction("urn:update");
sc.setOptions(opts);
sc.sendRobust(createPayLoad());
```

Replace `setTo` with an invalid one and see whether you are getting any exception.

Working with the OperationClient

As we already know, with `ServiceClient`, we have only access to payload in both, the sending and receiving sides. But that is not enough if we are trying to implement the enterprise level web applications. There, we need to have more control. In those cases, we might want to add custom headers into an outgoing SOAP message as well as we need to access the incoming SOAP process directly or we would be required to access incoming and outgoing message contexts. With `ServiceClient`, we can do none of these (however, we can get the current `OperationContext` once we invoke the service, and from this, we can access both, the request and the response `MessageContexts`). The solution is to use the operation client for these scenarios. Let's invoke the `echo` operation using the operation client to understand the API:

1. Create a `ServiceClient` instance.

   ```
   ServiceClient sc = new ServiceClient();
   ```

2. Create an OperationClient (to create an operation client, we need to pass the full qualified operation name , as it is in the dynamic client case)

   ```
   OperationClient opClient = sc.createClient(
                              ServiceClient.ANON_OUT_IN_OP);
   ```

   When we create a service client using its default constructor, it creates the antonyms service with three operations, and the constant `ServiceClient.ANON_OUT_IN_OP` is one of them.

3. Create a message context and set properties to its `Option` object:

   ```
   //creating message context
   MessageContext outMsgCtx = new MessageContext();
   //assigning message context's option object into instance variable
   Options opts = outMsgCtx.getOptions();
   //setting properties into option
   opts.setTo(new EndpointReference(
              "http://127.0.0.1:8000/axis2/services/MyService"));
   opts.setAction("urn:echo");
   ```

4. Create `SOAPEnvelope` and add that to the message context (not how we added in the previous cases). Here, you need to create a full SOAP envelop.

```
outMsgCtx.setEnvelope(creatSOAPEnvelop());
```

   The `creatSOAPEnvelop` method will look like the next snippet of code:

```
public SOAPEnvelope creatSOAPEnvelop() {
  SOAPFactory fac = OMAbstractFactory.getSOAP11Factory();
  SOAPEnvelope envelope = fac.getDefaultEnvelope();
  OMNamespace omNs = fac.createOMNamespace(
                     "http://ws.apache.org/axis2", "ns1");
  OMElement method = fac.createOMElement("echo", omNs);
  OMElement value = fac.createOMElement("echo", omNs);
  value.setText("Hello");
  method.addChild(value);
  envelope.getBody().addChild(method);
  return envelope;
}
```

   In this sample, what we have is the default `SOAPEnvelop` with a sample payload, but we can create a complex `SOAPEnvelop`, depending on our requirements.

5. Add a message context into the operation client:

```
opClient.addMessageContext(outMsgCtx);
```

6. To send the message, we need to call the `execute` method in the operation client:

```
opClient.execute(true);
```

   The Boolean method argument is to say whether we want to invoke it in a blocking manner or a non-blocking manner. If the value is true, invocation will be a blocking one.

7. To access a response message context and response `SOAPEnvelop`:

```
//pass message label as method argument
MessageContext inMsgtCtx = opClient.getMessageContext("In");
SOAPEnvelope response = inMsgtCtx.getEnvelope();
System.out.println(response);
```

When we are invoking an in-out MEP, as in this sample, the message label of the request is `Out` and value of the response is `In`; that is why we have to pass `In` as the message label value to get the response message context.

Once we have the message context, we can use that to access the `SOAPEnvelop`, properties, transport headers, and so on. And when we run this code sample, we should get the following as the console output:

```
<?xml version='1.0' encoding='utf-8'?>
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
    <soapenv:Header />
    <soapenv:Body>
         <ns:echoResponse
             xmlns:ns="http://ws.apache.org/axis2/xsd">
             <return>
                   Hello
              </return>
         </ns:echoResponse>
    </soapenv:Body>
</soapenv:Envelope>
```

Here, we discussed `OperationClient` as the means of accessing the incoming and outgoing `MessageContext`. One of the most useful use cases of accessing outgoing `MessageContext` or the `SOAPEnvelop` is to add headers. However, the `ServiceClient` API can also be used to add SOAP headers, as shown here:

```
sc.addHeader(SOAPHeaderBlock);
```

Or we could add headers, as shown here:

```
sc.addStringHeader(new QName("http://sample.org/header","MyHeader"),
"headervalue");
```

If we intercept the message using the TCP monitor or similar mechanisms, we can see the SOAP header in the SOAP message:

```
<soapenv:Header>
  <axis2ns1:MyHeader xmlns:axis2ns1=
    "http://sample.org/header">headervalue</axis2ns1:MyHeader>
</soapenv:Header>
```

In the meantime, we can access the last `OperationContext` using `ServiceClient`, as shown here:

```
OperationContext operationContext = sc.getLastOperationContext();
```

From `operationContext`, we can get either `In messageContexts` or `Out messageContexts`.

Summary

In this chapter, we discussed one of the key components of Axis2 known as Axis2 client API. We discussed how to create the Axis2 client instances in multiple ways as well as various ways of configuring them.

Next, we discussed the most commonly used APIs and we used code samples to explain the scenarios better. The Axis2 client API is very convenient and it has a number of cool features such as asynchronous web service invocations, multiple transport selection, and so on. After running the sample, we can easily understand the basis of the Axis2 client API; understating the rest of the API required us to write complex samples.

10
Session Management

By design, web services are said to be stateless, so there is no notion of keeping state in a web service. From the programming point of view, having one or more instances does not make any difference at execution. Though web services are stateless, when it comes to complex or enterprise applications, it is hard to support the required functionality without maintaining state or having the session management support. As an example, consider a banking application where you go to an ATM, insert the ATM card, and start performing some transaction. Internally, it maintains a secure session for you. So operations in a given transaction belong to the same secure session. As mentioned in previous chapters, the Axis2 Web Service framework provides better support for any type of enterprise application. Hence, Axis2 has better support for session management at different levels, which we will discuss during this chapter.

In this chapter, we will discuss Axis2 session management and how to use it. In particular, we will cover the following items:

- Stateless nature of Axis2
- Available session types
- Session-related operations
- Request session
- SOAP session
- Transport session
- Application session
- How to use session support at the client side

As stated earlier, by design, web services are said to be stateless. However, it is difficult to implement complex applications without having session management support. To understand the neediness of session support, let us consider a typical bank application. If we consider the sequence of events associated with a typical banking application, we get an idea as to why we need sessions:

- First the user logs in to his/her account (invoking the `login` method)
- Withdraw money (invoking some operation on his/her account)
- Log out (complete the transaction)

We can easily understand that the three operations stated earlier are interrelated and the same user does all those invocations. So this means that someone needs to keep a track of the user and the user data throughout the invocation of the methods. This simply indicates the requirement of session management to implement the banking application in the web-services way. Of course, there will be alternative ways of implementing the same application; then there is a need to have some other mechanism in place to identify and authenticate the user.

Stateless nature of Axis2

As we know by now, Axis2 architecture keeps logic and data separate. In the meantime, Axis2 has two types of data models—**static data** and **runtime data**. We have already discussed these in detail. In this chapter, we will discuss more about Axis2 dynamic data hierarchy because session management fully depends on the runtime data. The samples in this chapter will help you understand more about runtime and static data, and most importantly, where to use what.

In Axis2 framework, each individual component is said to be **stateless**. In other words, service implementation class or handlers should not try to store any local variable. Having stateless instances provides better support to reduce the concurrency control issues; this is due to the race conditions. In Axis2 handlers, `MessageReceivers`, `TransportSenders`, `TransportReceivers`, and even `AxisEngine`, are said to be stateless. So they do not keep any state in those classes. As a result of that, we do not mind whether we have one instance or a number of instances of the same handlers. As Axis2 has the notion of stateless nature in handlers, when you write handlers you need to write it in such a way that it does not keep any state in it. For example, we cannot consider the following handler implementation as a good approach. As we can see, it has a class variable to store the `messageContext`. So when we deploy Axis2 in a concurrent environment, we will definitely have issues. So, as the best practice, we need to keep in mind not to use any class variables inside our code.

```
public class InvalidHandler extends AbstractHandler {

    //Class variable to keep current MC
    private MessageContext currentMessageContext;

    public InvocationResponse invoke(MessageContext msgContext) throws
AxisFault {
        currentMessageContext = msgContext;
        //We need to write whatever the log we need to have here
        return InvocationResponse.CONTINUE;
    }
}
```

The preceding code sample shows that the handler that keeps the class variable calls `currentMessageContext`. This has serious issues when multiple concurrent requests are being handled. Thus the correct approach would be to *not* use class variables. You can solve this by either using a method variable or by storing it in some other context; we will discuss this later in this chapter. The following code sample demonstrates the correct way of handling variables:

```
    public InvocationResponse invoke(MessageContext msgContext) throws
AxisFault {
        MessageContext currentMessageContext = msgContext;
        //We need to write whatever the log we need to have here
        return InvocationResponse.CONTINUE;
    }
}
```

This does not mean that we cannot maintain state in Axis2; it simply tells us that keeping states in an implementation class is not the right approach. The most recommended way is to use the context hierarchy to store necessary data and manage sessions.

The available type of sessions in Axis2

As mentioned earlier, it is very difficult to implement enterprise level applications using a web service without having proper session management support. On the other hand, it is not a mandatory requirement to have session management support in a web service framework. We might have services that do not require session management at all.

In most cases, adding so many features tends to slow down the overall system performance. For example, if we have to check ten conditions as opposed to checking thousand conditions, it would show low performance. Hence, adding a new feature should make sure to add a minimum cost to the overall system runtime. it is the same with session management support. Not that when we need to keep the session-related data in the memory, it definitely increases the memory footprint. Nevertheless, we know that memory is not a big issue in the computer industry today. However, keeping so much data on memory causes a number of issues. If the memory footprint is too large, it will cause the collection of additional garbage and then that will reduce the system performance. As a result, it is required to have a tradeoff, keep data in the memory but for a pre-defined interval of time. That is one of the main design goals of the context hierarchy, where each level has a different life span.

When we consider session management, we see the requirement for different types of sessions and the lifetime of the session needs to vary from one to another. Some sessions last for a few seconds while some others last for the lifetime of the whole system. Axis2 architecture has been designed to support four types of sessions, and obviously, there are minor differences from one type to another. Considering the different types of use cases, Axis2 has the following four types of session scopes:

- Request
- SOAPSession
- Application
- Transport

When we were discussing Axis2 runtime data, we mentioned that we need to check the runtime data or context hierarchy for session management. Therefore, we need to memorize what we learn here to get a better understanding of session management.

Here we will learn about the five types of contexts in the hierarchy, which have been listed below with a brief explanation:

- `ConfigurationContext`: This is the runtime representation of the whole system. To start the Axis2 system, we need to have the configuration context. The lifetime of the configuration context will be the lifetime of the system, so if we store some state (a property), it will last forever (until system shutdown).
- `ServiceGroupContext`: In Axis2, we can deploy multiple services together as a service group, and then the runtime representation of that is called as a ServiceGroupContext.

- `ServiceContext`: This will represent the runtime of one service and the context lifetime will be the lifetime of the session. There can be one or many service contexts in the system, depending on the session scope of the corresponding service.
- `OperationContext`: This context is to represent the life time of a MEP (Message Exchange Patterns). The lifetime of an operation context is mostly less than the lifetime of the `ServiceContext`.
- `MessageContext`: The lifetime of an incoming message is represented by the message context. If two handlers in a given execution chain want to share data, then the best way is to store them in the message context. One `OperationContext` may have one or more `MessageContexts`.

Session initializing and session invalidating

When talking about the session management, one thing that comes to mind is the lifetime of the session. Most of the Web applications, like e-commerce, banking, hotel reservation, and so on, have a predefine lifetime. If you are not active for that time, it will automatically terminate your session and prompt you to log in again. Hence, there is a particular time when a session gets started and another when the session finishes. So whoever writes a session aware service might need to know when a session starts and when it finishes. To facilitate this, Axis2 uses Java reflection and optional interfaces to inform the service implementation class. Actually, there are two ways that a service author is notified when a session starts and finishes, irrespective of the session scope.

Java reflection

In this case, the service author has to implement the following two methods in his/her service implementation class, if he/she wants to be notified as to when the session starts and finishes. At runtime, when a session starts, Axis2 looks to see whether the following methods are there in the service implementation class; if so, it calls the right method:

```
//This method will be called when a session start
public void init(ServiceContext serviceContext) {
 // Our code goes here
}
//This method will be called when a session finishes.
public void destroy(ServiceContext serviceContext) {
  // Our code goes here
}
```

Using the optional interface

When we use Java reflection, there is a very slight probability of making mistakes in the method's name and parameters, so it is good to user interface. That would lead you to use the method in the correct manner and avoid mistakes. The corresponding interface that Axis2 provides is called `org.apache.axis2.service.Lifecycle`, which has the same two methods we discussed earlier. If you want to get a notification as to when a session starts, you should write your service implementation class to implement that particular interface; then Axis2 will automatically call the right method. So we can write our service implementation class as shown below:

```
public class MyService implements Lifecycle {
    public void init(ServiceContext context) throws AxisFault {

    }
    public void destroy(ServiceContext context) {

    }
}
```

Accessing MessageContext

In the first part of the chapter, we discussed that keeping class variables anywhere in Axis2 is not a good practice (for example, service implementation class, handler class). There, we also mentioned the fact that keeping local variables also leads to issues in the concurrent environments. Hence, it is not a good idea to keep variables inside the service implementation class. When managing sessions, the right way is to store the data in one of the context and access them whenever needed. Though Axis2 passes `ServiceContext` when the session starts, storing it inside the service implementation is not the right approach (storing as a class variable). Therefore, we need to have a mechanism to access those contexts. As we learned before, if you have access to `MessageContext`, then you can access the entire context hierarchy using that. Thus, if we can get access to the current `MessageContext`, we can consider accessing the entire context hierarchy. So the next question is how do we get the current `MessageContext` inside to the service class? That is very straightforward; Axis2 sets `MessageContext` into `ThreadLocal`, so from that, we can access the `MessageContext`. Let's say that we want to access the `MessageContext` inside the method called `foo`. We can do that as follows:

```
public void foo(){
        MessageContext messageContext = MessageContext.
getCurrentMessageContext();
 }
```

This can be used to access the current `MessageContext`, irrespective of the session scope we use.

Having discussed all the necessary pieces of session management, now it is time to discuss about the different types of available sessions in Axis2. As mentioned above, there are four different types of sessions. We will discuss all of them in detail here.

Request session scope

Request session scope is the default session scope in Axis2. When we deploy a service without knowing anything about session management, our service will be deployed in the request session scope. The lifetime of this session is limited to the method invocation lifetime or the request processing time. When we deploy a service in the request scope, it simply means that we are not going to worry about the session management at all. So it is like having no session management.

Once we deploy a service in the request session scope, for each and every invocation, a new service implementation class will be created. Say we have to deploy a service called `Foo` in the request scope; then if a client invokes the service ten times, 10 instances of the service class will be created.

If we want to specify the scope explicitly, we can still do that by adding a scope attribute to the service element in `services.xml`, as follows. However, as mentioned earlier, deploying a service in the request scope does not require any modifications (the following is optional):

```
<service name="Foo"  scope=" request">
</service>
```

To get an understanding about request scope, create a service using the following service class, deploy it, and invoke it:

```
public class MyService implements Lifecycle {
    public void init(ServiceContext context) throws AxisFault {
        System.out.println("I'm inside init method ");
    }
    public void destroy(ServiceContext context) {
        System.out.println("I'm inside destroy method");
    }
    public String foo(String foo) {
        return foo;
    }
}
```

To invoke the service, we just type the following in the browser:

```
http://localhost:8080/axis2/services/MyService/foo?foo=foo
```

We will see the following in the server console:

```
[INFO] Deploying Web service: HelloWorld.aar - file:/C:/Program Files/Apache Sof
tware Foundation/Tomcat/webapps/axis2/WEB-INF/services/HelloWorld.aar
Oct 19, 2010 9:32:51 AM org.apache.coyote.http11.Http11AprProtocol start
INFO: Starting Coyote HTTP/1.1 on http-127.0.0.1-8080
Oct 19, 2010 9:32:51 AM org.apache.coyote.ajp.AjpAprProtocol start
INFO: Starting Coyote AJP/1.3 on ajp-8009
Oct 19, 2010 9:32:51 AM org.apache.catalina.storeconfig.StoreLoader load
INFO: Find registry server-registry.xml at classpath resource
Oct 19, 2010 9:32:51 AM org.apache.catalina.startup.Catalina start
INFO: Server startup in 11781 ms
I'm inside init method
```

If we continue to do so, we will see it getting printed every time we invoke the service.

It should be noted here that even if we deploy a service in a request scope, there are many ways of keeping our service as a state full service. One way is to store the state in Axis2 global runtime (`ConfigurationContext`) and retrieve whenever necessary. Notably, this will add more overhead to the system, as there is only one `ConfigurationContext`, and if everything tries to access it, it will be a bottleneck.

SOAP session scope

The idea of SOAP session is to have a transport-independent way of managing a session between two SOAP nodes; obviously, between the client and the server. Here, Axis2 uses SOAP headers in order to manage the session. In the SOAP session scope, it has a slightly longer lifetime than the request session scope, and deploying a service in a SOAP session is required to change `services.xml` as well. Managing a SOAP session requires both the client and the service to be aware of the sessions, that is, the client has to send the session-related data if he/she wants to access the same session and the service has to validate the user using the session-related data.

SOAP-based session management is not interoperable; part of the reason is that there's no agreement upon a standard specification to manage the session. There is a standard specification called WS-Context, but none of the commonly used web service frameworks have implemented it.

In order to manage a SOAP session, a client has to send an additional reference parameter in the SOAP header, which is named as `serviceGroupId` (it will be sent to the client when he invokes a service that deploys in SOAP session the first time). In the meantime, a SOAP session has to provide a way to manage sessions across not only a single service invocation, but also for multiple services in a service group. As long as we are in the same SOAP session, we can manage service related data in `ServiceContext` and if we want to share data across other services in the group, then we can use `ServiceGroupContext` to store the session-related data.

When we deploy a service in a SOAP session and when a client tries to access the service the first time, Axis2 will generate `serviceGroupId` and send that to the client as a reference parameter in `wsa:ReplyTo`, as shown next. However, we should mention here that to have SOAP session support both the client and the server, it has to have WS-addressing support.

```
<wsa:ReplyTo>
       <wsa:Address>http://www.w3.org/2005/08/addressing/anonymous</
        wsa:Address>
       <wsa:ReferenceParameters>
<axis2:ServiceGroupId xmlns:axis2="http://ws.apache.org/namespaces/axi
s2">urn:uuid:65E9C56F702A398A8B11513011677354</axis2:ServiceGroupId>
       </wsa:ReferenceParameters>
 </wsa:ReplyTo>
```

So if a client wants to live in the same session, then he/she has to copy that reference parameter and send it back to the server when he/she invokes the service the second time. As long as a client sends the valid `ServiceGroupId`, he/she can use the same session, and a service can maintain the session-related data. Unlike request session scope, SOAP session has a default timeout period, if the client does not touch the service for a period of 30 seconds, then the session will expire, and if the client sends the old `serviceGroupId`, he/she will get an AxisFault. We can change the default timeout period by changing the server's `axis2.xml` file as follows:

```
<parameter name="ConfigContextTimeoutInterval">30000</parameter>
```

By changing the parameter value, we can have a timeout interval that we want.

As mentioned earlier, deploying a service in a SOAP session requires that we change `services.xml` as follows:

```
<service name="MyService"  scope=" soapsession">
</service>
```

It should be noted here that though we discussed about copying the references parameter, once we use the Axis2 client, we can configure it to copy the parameter automatically, and we do not need to worry about doing it manually.

Just to get an idea about the SOAP session management, let's write the following service class and deploy it. If you look at it carefully, what the service class does add the current passes value to the previous value and sends the result. So once we keep on doing that, we should get the incrementing values. First write the service and deploy that in the SOAP session while doing the necessary changes to `services.xml`:

```
public class MyService {
    public int add(int value) {
        MessageContext messageContext = MessageContext.
getCurrentMessageContext();
        ServiceContext sc = messageContext.getServiceContext();
        Object previousValue = sc.getProperty("VALUE");
        int previousIntValue = 0;
        if (previousValue != null) {
           previousIntValue = Integer.parseInt((String)previousValue);
        }
        int currentValue = previousIntValue + value;
        sc.setProperty("VALUE","" + currentValue);
        return currentValue;
    }
}
```

Now let's use the following code to invoke the service. As you can see, we have engaged the addressing module but we have not done anything to manage the sessions:

```
ServiceClient sc = new ServiceClient();
sc.engageModule("addressing");
Options opts = new Options();
opts.setTo(new EndpointReference(
    "http://127.0.0.1:8080/axis2/services/MyService"));
opts.setAction("urn:add");
sc.setOptions(opts);
OMElement ele = sc.sendReceive(createPayLoad(10));
System.out.println(ele.getFirstElement().getText());
ele = sc.sendReceive(createPayLoad(10));
System.out.println(ele.getFirstElement().getText());
```

The `createPayLoad` method is shown here:

```
public static OMElement createPayLoad(int intValue) {
  OMFactory fac = OMAbstractFactory.getOMFactory();
  OMNamespace omNs = fac.createOMNamespace(
                   "http://ws.apache.org/axis2", "ns1");
        OMElement method = fac.createOMElement("add", omNs);
        OMElement value = fac.createOMElement("args", omNs);
        value.setText("" + intValue);
        method.addChild(value);
        return method;
    }
```

Once we run the code, we will see following output on the client side:

```
10
10
```

This means though that we invoke the service twice as we have gotten the same output. This simply means no session has taken place, so now let's change our client code a bit and see what we are getting. Just add the following line of code and run the client again:

```
opts.setManageSession(true);
sc.setOptions(opts);
```

Now you should see the following output on the client side:

```
10
20
```

So this simply tells us that we have invoked the service in a session-aware manner.

Transport session scope

Transport session management is very similar to session management in Web domain, where as long as you keep the browser open or as long as you have the cookies, you can stay in the same session. In the case of transport session, Axis2 uses transport related session management techniques to manage session, as an example in the case of HTTP, it uses HTTP cookies to manage a session. One of the important factors with transport level session management is the lifetime of the session control by the transport, and not by Axis2. What Axis2 does is it stores service context and ServiceGroupContext in the transport session object, so that a service can access those contexts as long as the session lives.

One of the key advantages of a transport session over other sessions is that we can talk to multiple service groups within one transport session. In a SOAP session, we don't have a way to communicate between two service groups, but with the transport session, we have that capability too. In this case, the number of service instances created depends on the number of transport sessions created.

Deploying a service in the transport session requires us to change `services.xml` as follows:

```
<service name="MyService"  scope=" transportsession">
</service>
```

Now, let's change our previous sample to have the scope as transport and redeploy the service. Next we'll look at several ways to invoke the service.

Option 1: Using the browser

`http://localhost:8000/axis2/services/MyService/add?value=10`

If we keep on typing that, we will get the following output:

```
<ns:echoResponse>
  <return>10</return>
</ns:echoResponse>

<ns:echoResponse>
  <return>20</return>
</ns:echoResponse>

<ns:echoResponse>
  <return>30</return>
</ns:echoResponse>
```

Option 2: Using the service client

When we use the service client, once we set the session management flag to true, it will send back the transport cookie as well. So if we run the following code, we will see the following outcome:

```
ServiceClient sc = new ServiceClient();
Options opts = new Options();
opts.setTo(new EndpointReference(
  "http://127.0.0.1:8000/axis2/services/MyService"));
opts.setAction("urn:add");
opts.setManageSession(true);
sc.setOptions(opts);
OMElement ele = sc.sendReceive(createPayLoad(10));
System.out.println(ele.getFirstElement().getText());
ele = sc.sendReceive(createPayLoad(10));
System.out.println(ele.getFirstElement().getText());
```

```
10
20
```

It simply tells us that we have to invoke the service deployed in the transport session (invoked in a session-aware manner). In this case, we do not need to have the addressing module.

Application scope

Application scope has the longest lifetime compared to all the others and the lifetime of the application session is equal to the lifetime of the system. If we deploy a service in the application scope, there will be only one instance of that service and obviously there will be only one `ServiceContext` for that service too. In the world of Axis2, if we consider the memory footprint and if we don't want to manage a session, then a good idea is to deploy the service in the application scope.

When we deploy a service in the application scope, a client does not need to send any additional data to use the same session.

To deploy a service in the application scope, we need to change `axis2.xml`, as shown here:

```
<service name="foo"  scope=" application">
</service>
```

Service deployed in the application session scope is not guaranteed to be threading-safe.

Managing sessions using ServiceClient

As we know by now, managing a session in the client side involves bit of a work. As mentioned earlier, both in the SOAP session and transport session, a client has to send the session-related data, if he/she wants to live in the same session. Maybe he can do that for a SOAP session by coping with the required reference parameters. However, with transport session, how can a user get access to transport, to copy, and send cookies?

To make life easier, Axis2 has the inbuilt capability of managing sessions in the client session by just setting a flag, and we have used that already. Then, depending on the service-side session, it will send the corresponding data as long as we use the same service client. So, the main requirement is to use the same service client to invoke the service if you want to live in the same session.

If we want to live in the same session, we can create a service client, shown as follows, and re-use the created service client object to invoke the service:

```
Options options = new Options();
options.setManageSession(true);
ServiceClient sender = new ServiceClient();
sender.setOptions(options);
```

Once we create the `ServiceCient`, as shown here, if the service deploys in a SOAP session, it will copy the `serviceGroupId` and send that from the second invocation onwards. If the server sends the session ID, such as HTTP cookies, it will copy that to `ServiceContext` (in the client side) and send it back to the server when the client invokes the service for the second time.

Summary

Stateless nature is one of the main characteristics of web services. However, this is a limitation for the advanced web services developers. Developing an enterprise level application using web services is not easy, unless we have a session management layer. Axis2 has the level four of sessions to address enterprises level web service development issues. In this chapter, we discussed different types of sessions available in Axis2. After that, we used a sample service and a sample client to demonstrate the different sessions and how they work.

In the next chapter, we will focus on another very good web service specification, called JAXWS. There, we covered the JAXWS annotations, deployment, and some examples to illustrate how to use it.

11
Developing JAX-WS Web Services

Java API for XML-based web services is a popular standard, targeting Java developers who write web services and web service clients. The conventional **contract first** approach, which consists of defining the web service contract in a WSDL and then generating the appropriate code from those WSDLs, is not the most natural way of programming for Java developers. The JAX-WS specifications allow you to write a regular Java code and annotate the code appropriately, in order to make them into web services. Therefore, JAX-WS is a **code first** approach for developing web services and clients in Java. The ease of development is achieved through writing everything as Java classes and not having to write any specific deployment descriptor file such as `services.xml` files utilized by Axis2 AAR services. Older Java Web Service specifications such as JAX-RPC required the web service classes to implement a specific interface. With JAX-WS, there is no such restriction and regular POJOs can be made into web services with minimal changes. JAX-WS also allows contract first development. You will see how JAX-WS-annotated web services can be generated starting from WSDL using the wsimport tool later in this chapter. At the time of writing, Axis2 only supports the JAX-WS 2.0 specification.

In this chapter, we will be looking at:

- JAX-WS Annotations
- Server-side JAX-WS
- Client-side JAX-WS

By the end of this chapter, you will have a good understanding about JAX-WS annotations and how to develop JAX-WS services and clients in Apache Axis2.

Writing a simple JAX-WS web service

First, let us look at how easy it is to write a simple web service using JAX-WS annotations:

```
import javax.jws.WebService;
import javax.jws.Oneway;
import javax.jws.WebMethod;

@WebService
public class HelloWorld {
    public String hello(String name) {
        return "Hello " + name;
    }
}
```

As you can see, the simplest method to make a Java class into a web service is by adding the `WebService` annotation. The `@WebService` annotation will mark the `HelloWorld` class as a web service. The `hello` method will be transformed into a web service operation that takes in a single parameter. This class can be compiled and archived as a JAR (Java ARchive) file, and then can be deployed on an Axis2 server.

Let us go through the widely used JAX-WS annotations in detail.

JAX-WS annotations

The JAX-WS specifications define a number of annotations. In this section, you will see the purpose and usages of some of the widely used JAX-WS annotations. Annotations can be categorized into two types—annotations that are used in mapping Java to WSDL and schema and annotations that are used at runtime to control how the JAX-WS runtime processes and responds to web service invocations. We will be looking at a set of annotations coming from several specifications:

- JSR 181 (Web Service Metadata)
- JSR 224 (JAX-WS)
- JSR 222 (JAXB)
- JSR 250 (Common Annotations)

JSR 181 (Web Service Metadata) annotations

In this section, we will be looking at the annotations introduced in the JSR 181 (Web Service Metadata) specification. We will be looking at the following annotations:

- `javax.jws.WebService`
- `javax.jws.WebMethod`
- `javax.jws.OneWay`
- `javax.jws.WebParam`
- `javax.jws.WebResult`
- `javax.jws.soap.SOAPBinding`

javax.jws.WebService

This annotation marks a Java class as defining a web service interface. By default, all public methods in this Java class will be exposed as web service operations. Web service implementation classes must have a `WebService` annotation. This annotation is defined as follows:

```
@Retention(value=RetentionPolicy.RUNTIME)
@Target({TYPE})
public @interface WebService {
    String name() default "";
    String targetNamespace() default "";
    String serviceName() default "";
    String wsdlLocation() default "";
    String endpointInterface() default "";
    String portName() default "";
}
```

Let us look at the properties of the `WebService` annotation.

name

This property defines the name of the `wsdl:portType` in the WSDL 1.1 document. The default value of this property is the unqualified name of the relevant Java class or interface annotated with the `WebService` annotation.

targetNamespace

This property defines the XML namespace of the WSDL and some of the XML elements generated from this Java class or interface annotated with the `WebService` annotation. The default value is the namespace derived from the package name of this web service. For example, if the web service is part of the `org.apache.axis2.sample` package, the default namespace will be `http://sample.axis2.apache.org`.

serviceName

This property defines the name of the `wsdl:service` in the WSDL 1.1 document. The default value is the unqualified name of the annotated Java class or interface, suffixed with the string "`service`".

endpointInterface

This property defines the qualified name of the **Service Endpoint Interface (SEI)**. This annotation allows the separation of the interface contract from the implementation. The endpoint implementation class is not required to implement the `endpointInterface`.

If `endpointInterface` property is specified, all other `WebService` properties are ignored as are all other JSR 181 annotations. Only the annotations on the service endpoint interface will be taken into considerations.

portName

This property defines the `wsdl:portName` in the WSDL 1.1 document. The default value of this property is `WebService.name` suffixed with the string "`port`".

wsdlLocation

This property is a WSDL URL that points to an existing WSDL 1.1 document. This property is used only if an existing WSDL is in use. In a pure code first approach, this property will not be used.

Let us look at a simple example where the `WebService` annotation is used along with some of its attributes. The following code will declare `CalculatorImpl` to be a web service with the name `Calculator` and the `targetNamespace` of that web service will be `http://axis.apache.org`:

```
@WebService(name = "Calculator", targetNamespace = "http://axis.
apache.org")

public class CalculatorImpl {
```

```
        public int add(int a, int b) {
           return a + b;
        }
}
```

If you wish to separate the interface from the implementation class, you can do it as shown here using the `endpointInterface` property. Here, the `endpointInterface` is declared to be `org.apache.axis2.jaxws.sample.ICalculator`.

```
@WebService(endpointInterface = "org.apache.axis2.jaxws.sample.
ICalculator")
public class CalculatorImpl {
     public int add(int a, int b) {
        return a + b;
     }
}
```

The `ICalculator` interface is declared to be a web service with the name `Calculator` and `targetNamespace` `http://axis.apache.org`:

```
@WebService(name="Calculator", targetNamespace = "http://axis.apache.
org")
public interface ICalculator{
     int add(int a, int b);
}
```

javax.jws.WebMethod

This annotation can be used to customize a method which is exposed as a web service operation. This annotation is defined as follows:

```
@Retention(value=RetentionPolicy.RUNTIME)
@Target({METHOD})
public @interface WebMethod {
    String operationName() default "";
    String action() default "";
    boolean exclude() default false;
}
```

Let us look at the properties of the `WebMethod` annotation.

operationName

This property defines the name of the `wsdl:operation` in WSDL 1.1 corresponding to this method. The namespace of this name is taken from the value `WebService.targetNamespace` or its default value. The default value of this property is the name of the annotated Java method.

action

This property defines the action of this method's web service operation. It defaults to "" (an empty string).

exclude

This property is used for excluding certain methods from a web service. Such methods will not show up as web service operations. The default value is `false`.

Let us look at an example where the `WebMethod` annotation is used along with some of its attributes. In this example, the add method will be exposed as a web service operation with the name `addInt` and the action will be set to `urn:addInt`. The `print` method will not be exposed as a web service, as the `exclude` attribute of the `WebMethod` annotation has been set to `false`.

```
@WebService
public class CalculatorImpl {
     @WebMethod(operationName="addInt", action="urn:addInt")
     public int add(int a, int b) {
        print(a, b);
        return a + b;
     }
     @WebMethod(exclude="true")
     public void print(int a, int b) {
         System.out.println("a=" + a + ", b=" + b);
     }
}
```

javax.jws.OneWay

This annotation is used for marking a method as a one-way method. The corresponding web service operation will be an in-only operation. This annotation has no properties and is defined as follows:

```
@Retention(value=RetentionPolicy.RUNTIME)
@Target({METHOD})
   public @interface Oneway {
}
```

Let us look at a simple example that demonstrates the usage of the `OneWay` annotation. Here, the process operation has been declared as an in-only operation using the `OneWay` attribute.

```
@WebService
public class Processor {
     @WebMethod(operationName="process", action="urn:process")
     @OneWay
     public void process(String id) {
        System.out.println("Processing " + id);
     }
}
```

javax.jws.WebParam

You can use this annotation for customizing the mapping of a single parameter to a WSDL message part or an XML element. This annotation is defined as follows:

```
@Retention(value=RetentionPolicy.RUNTIME)
@Target({PARAMETER})
public @interface WebParam {
    public enum Mode {
    IN,
    OUT,
    INOUT
    };
    String name() default "";
    String targetNamespace() default "";
    Mode mode() default Mode.IN;
    boolean header() default false;
    String partName() default "";
}
```

Let us take a look at the properties of the `WebParam` annotation.

name

For RPC bindings, this is the name of the `wsdl:part` representing the parameter. For document/literal wrapped bindings, this is the local name of the XML element, representing this parameter. This property is not used for document/literal bare bindings.

targetNamespace

This is the namespace of this operation parameter. It is only used with document/literal wrapped bindings. The default value of this property is the `targetNamespace` of the web service.

mode

This property represents the parameter flow direction for this method. Possible values are `IN`, `INOUT`, and `OUT`.

header

This property defines whether the annotated parameter should be carried in the SOAP header. The default value is `false`.

partName

This property defines the `wsdl:part` for the annotated parameter with RPC or document/bare operations. The default value is `WebParam.name`.

The following example illustrates how the `WebParam` annotation is used in practice. The `add` operation has been declared to have two web service parameters with the names `intA` and `intB`:

```
@WebService(name = "Calculator", targetNamespace = "http://axis.
apache.org")
public class CalculatorImpl {
     @WebMethod(operationName="addInt", action="urn:addInt")
     public int add(@WebParam(name="intA") int a,
                    @WebParam(name="intB") int b) {
        return a + b;
     }
}
```

javax.jws.WebResult

If you wish to customize the mapping of the return value of a web service method to a `wsdl:part` or an XML element, you should use the `WebResult` annotation. This annotation is defined as follows:

```
@Retention(value=RetentionPolicy.RUNTIME)
@Target({METHOD})
public @interface WebResult {
```

```
    String name() default "return";
    String targetNamespace() default "";
    boolean header() default false;
    String partName() default "";
}
```

Let us look at the properties of the `WebResult` annotation.

name

This property defines the name of the return value in the WSDL. For RPC bindings, this is the part name of the return value in the response message. For document bindings, this is the local name of the XML element representing the return value.

targetNamespace

This is the XML namespace of the return value and defaults to the `targetNamespace` of the web service.

header

This property specifies whether the return value needs to be carried in the SOAP header and the default value is `false`.

partName

This property specifies the `wsdl:part` of the return value. It defaults to `WebResult.name`.

The following examples illustrate the usage of this annotation. The `WebResult` annotation declares that the name of the return type should be `return`:

```
@WebService(name = "Calculator", targetNamespace = "http://axis.
apache.org")
public class CalculatorImpl {
     @WebMethod(operationName="addInt", action="urn:addInt")
     @WebResult(name="return")
     public int add(@WebParam(name="intA") int a,
                    @WebParam(name="intB") int b) {
        return a + b;
     }
}
```

javax.jws.soap.SOAPBinding

This annotation specifies how the web service is mapped to the SOAP message or the wire format. This annotation is defined as follows:

```
@Retention(value=RetentionPolicy.RUNTIME)
@Target({TYPE, METHOD})
public @interface SOAPBinding {
    public enum Style {
       DOCUMENT,
       RPC,
    };
    public enum Use {
        LITERAL,
        ENCODED,
    };
    public enum ParameterStyle {
        BARE,
        WRAPPED,
    };
    Style style() default Style.DOCUMENT;
    Use use() default Use.LITERAL;
    ParameterStyle parameterStyle() default ParameterStyle.WRAPPED;
}
```

Let us look at the usages of the annotation properties.

style

This property defines the style for messages used in a web service. The value can be either `DOCUMENT` or `RPC`, and the default value is `DOCUMENT`.

use

This property defines the encoding used for messages in a web service, and the default value is `LITERAL`. JAX-WS 2.0 and newer specifications only allow `LITERAL`.

parameterStyle

This property determines whether the method's parameters represent the entire message body or whether the parameters are wrapped in a body element named after the web service operation. The possible values are `WRAPPED` or `BARE`. The `BARE` parameter style can only be used with the `DOCUMENT` style bindings. The default value of this property is `WRAPPED`.

Let us look at a simple example:

```
@WebService(name = "Calculator", targetNamespace = "http://axis.
apache.org")
```

```
@SOAPBinding(style=SOAPBinding.Style.RPC, use=SOAPBinding.Use.LITERAL)
public class CalculatorImpl {
     public int add(int a, int b) {
        return a + b;
     }
}
```

JSR 224 (JAX-WS) annotations

In this section, we will be looking at some of the useful annotations introduced in the JSR 224 (JAX-WS) specification. These are additional annotations that supplement the JSR-181 annotations. We will be looking at the following annotations:

- `javax.xml.ws.BindingType`
- `javax.xml.ws.RequestWrapper`
- `javax.xml.ws.ResponseWrapper`
- `javax.xml.ws.ServiceMode`
- `javax.xml.ws.WebEndpoint`
- `javax.xml.ws.WebFault`
- `javax.xml.ws.WebServiceClient`
- `javax.xml.ws.WebServiceProvider`
- `javax.xml.ws.WebServiceRef`

javax.xml.ws.BindingType

The `BindingType` annotation is used to specify the binding to use for a web service endpoint implementation class. It has a single attribute, `value`, which is a binding URI, and the default value is `SOAP 1.1 / HTTP`. Let us look at a simple example:

```
@WebService
@BindingType(value="http://www.w3.org/2003/05/soap/bindings/HTTP/")
public class AddNumbers {
     public int add(int a, int b) {
         …
     }
}
```

The deployed endpoint would use SOAP1.2 over HTTP binding, as indicated by the `value` property of the `BindingType` annotation.

javax.xml.ws.RequestWrapper and javax.xml.ws.ResponseWrapper

The `javax.xml.ws.RequestWrapper` annotation annotates methods in the SEI with the request wrapper bean to be used at runtime. The `javax.xml.ws.ResponseWrapper` annotation annotates the methods in the SEI, with the response wrapper bean to be used at runtime. These annotations have three properties, namely, `localName`, `targetNamespace`, and `className`.

localName

This property defines the `localName` of the XML Schema element representing this request/response wrapper and defaults to the `operationName`, as defined by `javax.jws.WebMethod`.

targetNamespace

This property defines the namespace of the request/response wrapper element and the default value is `targetNamespace` of the SEI.

className

This property defines the name of the class representing the request/response wrapper.

Let us look at the following example:

```
public interface AddNumbersImpl {
    @WebMethod
    @WebResult(targetNamespace = "")
    @RequestWrapper(localName = "addNumbers",
      targetNamespace = "http://axis.apache.org/axis2/jaxws,
      className = "org.apache.axis2.jaxws.sample.AddNumbers")
    @ResponseWrapper(localName = "addNumbersResponse",
       targetNamespace = "http://server.fromjava/",
       className = " org.apache.axis2.jaxws.sample.AddNumbersResponse")
    public int addNumbers(
       @WebParam(name = "arg0", targetNamespace = "")
       int arg0,
```

```
        @WebParam(name = "arg1", targetNamespace = "")
        int arg1);
}
```

javax.xml.ws.ServiceMode

Web service endpoints may choose to work at the XML message level by implementing the `Provider` interface. This is achieved by implementing either `Provider<Source>`, `Provider<SOAPMessage>`, or `Provider<DataSource>`. The endpoint accesses the message or message payload using this low-level, generic API. All the Provider endpoints must have the `@WebServiceProvider` annotation. The `@ServiceMode` annotation is used to convey whether the endpoint wants to access the message (`Service.Mode.MESSAGE`) or payload (`Service.Mode.PAYLOAD`). If there is no `@ServiceMode` annotation on the endpoint, the payload is the default value. This annotation contains a single attribute `value`, which conveys whether the `Provider` endpoint wants to access the entire message (`MESSAGE`) or just the payload (`PAYLOAD`), and the default value is `PAYLOAD`. The following example demonstrates the usage of this annotation:

```
@ServiceMode(value=Service.Mode.PAYLOAD)
public class AddNumbersImpl implements Provider<Source> {
    public Source invoke(Source source) throws RemoteException {
    }
}
```

javax.xml.ws.WebEndpoint

This annotation is used to annotate the `getPortName()` methods of a generated service interface. The information specified in this annotation is sufficient to uniquely identify a `wsdl:port` element inside a `wsdl:service`. It has a single attribute, name, which defines the local name of the XML element representing the corresponding port in the WSDL and defaults to "" (an empty string). The following example demonstrates this annotation:

```
@WebServiceClient(name = "AddNumbersImplService",
targetNamespace = "http://axis.apache.org/axis2/jaxws", wsdlLocation =
"http://localhost:8080/services/addnumbers?wsdl")
public class NumberAdder {
    ...
    @WebEndpoint(name = "AddNumbersImplPort")
    public AddNumbersImpl getAddNumbersImplPort() {
        ...
    }
}
```

javax.xml.ws.WebFault

This annotation is generated by the JAX-WS tools into service-specific exception classes generated from a WSDL to customize the local and namespace name of the fault element and the name of the fault bean and to mark the service-specific exception as one generated from WSDL. It contains three attributes—`name`, `targetNamespace`, and `faultBean`.

name

This property defines the local name of the XML element representing the corresponding fault in the WSDL, and the default value is "" (an empty string).

targetNamespace

This property defines the namespace of the XML element representing the corresponding fault in the WSDL, and the default value is "" (an empty string).

faultBean

This property defines the qualified name of the Java class that represents the details of the fault message, and the default value is an empty string "".

The following example shows the usage of this annotation:

```
@javax.xml.ws.WebFault(name="AddNumbersException",
                       targetNamespace="http://axis.apache.org/axis2/
jaxws")
public class AddNumbersException_Exception extends Exception {
    private org.apache.axis2.jaxws.AddNumbersException fault;
}
```

javax.xml.ws.WebServiceClient

The information specified in this annotation is sufficient to uniquely identify a `wsdl:service` element inside a WSDL document. This `wsdl:service` element represents the web service for which the generated service interface provides a client view. This annotation can contain three attributes—`name`, `targetNamespace`, and `wsdlLocation`.

name

This property defines the local name of the `wsdl:serviceName` in the WSDL, and the default value is "".

targetNamespace

This property defines the namespace for the `wsdl:serviceName` in the WSDL, and the default value is "".

wsdlLocation

This property defines the location of the WSDL that defines this service, and the default value is "".

Let us look at an example:

```
@WebServiceClient(name = "AddNumbersClient",
  targetNamespace = "http://axis.apache.org/axis2/jaxws/",
                    wsdlLocation = "http://localhost/services/
                    AddService?wsdl")
public class AddNumbersClient {
}
```

javax.xml.ws.WebServiceProvider

This annotation is used to annotate a `Provider` implementation class.

targetNamespace

This is the XML namespace of the WSDL and some of the XML elements generated from this web service. Most of the XML elements will be in the namespace according to the JAXB mapping rules.

serviceName

This is the service name of the web service: `wsdl:service`, and the default value is an unqualified name of the Java class or interface + `service`.

portName

This is the `wsdl:portName`.

wsdlLocation

This is the location of the WSDL description of the service.

Let us look at an example:

```
@ServiceMode(value=Service.Mode.PAYLOAD)
@WebServiceProvider(wsdlLocation="WEB-INF/wsdl/AddNumbers.wsdl")
public class AddNumbersImpl implements Provider {
    public Source invoke(Source source) {
        …
```

```
        }
    }
```

javax.xml.ws.WebServiceRef

This annotation is used to define a reference to a web service and, optionally, an injection target for it. Web service references are resources in the Java EE 5 sense. It can have the following properties:

name

This property defines the JNDI name of the resource. For field annotations, the default value is the field name. For method annotations, the default is the JavaBeans property name corresponding to the method. For class annotations, there is no default and this must be specified.

type

This property defines the Java type of the resource. For field annotations, the default value is the type of the field. For method annotations, the default is the type of the JavaBeans property. For class annotations, there is no default and this must be specified.

mappedName

This property defines the product-specific name that this resource should be mapped to.

value

This property defines the service class and is always a type extending `javax.xml.ws.Service`. This element must be specified whenever the type of the reference is a service endpoint interface.

wsdlLocation

This property defines the location of the WSDL description for the service.

JSR 222 (JAXB) annotations

In this section, we will be looking at some of the useful annotations introduced in the JSR 222 (JAXB) specification. JAXB is a specification which defines how JavaBeans can be data bound to XML. We will be looking into the following annotations:

- `javax.xml.bind.annotation.XmlRootElement`
- `javax.xml.bind.annotation.XmlAccessorType`
- `javax.xml.bind.annotation.XmlElement`

javax.xml.bind.annotation.XmlRootElement

This annotation is used for mapping a top-level class to a global element in the WSDL's XML schema. This annotation is defined as follows:

```
@Retention(RUNTIME)
@Target({TYPE})
public @interface XmlRootElement {
    String namespace() default "##default";
    String name() default "##default";
}
```

This annotation contains two properties—`namespace` and `name`.

namespace

This is the namespace of the XML element representing the annotated class and the default is the namespace derived from the package containing the relevant class.

name

This is the local name of the XML element representing the annotated class and the default is the name of the relevant class.

Let us look at a simple example:

```
@WebService(name = "Calculator", targetNamespace = "http://axis.
apache.org/axis2")
public class CalculatorImpl {
     public int add(Add add) {
        return add.a + add.b;
     }
}
@XmlRootElement(name="Add", namespace="http://axis.apache.org/axis2")
public class Add {
     ...
}
```

javax.xml.bind.annotation.XmlAccessorType

This annotation is used to specify whether fields or properties are serialized by default. This annotation contains a single property, value, which specifies whether fields or properties are serialized by default. The value can be `AccessType.FIELD` or `AccessType.PROPERTY`, and the default value is `AccessType.PROPERTY`. The following example demonstrates the usage of this annotation:

```
@XmlRootElement(name="addNumbers", namespace="http://axis.apache.org/
axis2/jaxws")
@XmlAccessorType(AccessType.FIELD)
public class AddNumbers {
    @XmlElement(namespace="", name="number1")
    @ParameterIndex(value=0)
    public int a;
    @XmlElement(namespace="", name="number2")
    @ParameterIndex(value=1)
    public int b;
    public AddNumbers (){}
}
```

javax.xml.bind.annotation.XmlElement

This annotation is used to map a property contained in a class to a local element in the XML Schema complex type to which the containing class is mapped. This can be best understood by looking at the following example:

```
@WebService(name = "Calculator", targetNamespace = "http://axis.
apache.org/axis2")
public class CalculatorImpl {
     public int add(Add add) {
        return add.a + add.b;
     }
}
@XmlRootElement(name="Add", namespace="http://axis.apache.org/axis2")
public class Add {
     @XmlElement(namespace="http://axis.apache.org/axis2",
name="numberA")
     public int a;
     @XmlElement(namespace="http://axis.apache.org/axis2",
name="numberB")
     public int b;
}
```

The two significant properties of this annotation are `namespace` and `name`.

name

This is the local name of the XML element representing the property of the annotated JavaBean and the default value is the name of the annotated Java attribute.

namespace

This is the namespace of the XML element representing the property of the annotated JavaBean, and the default value is the namespace of the class containing the Java attribute.

JSR 250 (Common Annotations)

In this section, we will be looking at some of the useful annotations introduced in the JSR 250 (Common Annotations) specification. We will be looking into the following annotations:

- `javax.annotation.Resource`
- `javax.annotation.PostConstruct`
- `javax.annotation.PreDestroy`

javax.annotation.Resource

This annotation is used to mark a `WebServiceContext` resource that is needed by a web service. It is applied to a field or a method for JAX-WS endpoints. The container will inject an instance of the `WebServiceContext` resource into the endpoint implementation when it is initialized. This annotation is illustrated in the following example:

```
@WebService
public class HelloImpl {
    @Resource
    private WebServiceContext context;

    public String echo(String name) {
      ...
    }
}
```

javax.annotation.PostConstruct

This annotation is used on a method that needs to be executed after a dependency injection is done to perform any initialization. This method must be invoked before the class is put into a service. This annotation is illustrated in the following example:

```
@WebService
public class HelloImpl {
    @PostConstruct
    private void initialize() {
        ...
    }
    public String echo(String name) {
        ...
    }
}
```

javax.annotation.PreDestroy

The `PreDestroy` annotation is used on methods as a callback notification to signal that the instance is in the process of being removed by the container. The method annotated with `PreDestroy` is typically used to release resources that it has been holding. This annotation is illustrated in the following example:

```
@WebService
public class HelloImpl {
    public String echo(String name) {
        ...
    }
    @PreDestroy
    private void preDestroy() {
        ...
    }
}
```

Now that you have a good understanding of the JAX-WS annotations, let's get our hands dirty by writing some code.

Code first service development with JAX-WS

We already have looked at several code segments, involving the code first JAX-WS development. But for the sake of completeness, let us look at a complete sample. In this example, the `StudentMarksService` class contains two operations called `computeAverage` and `computeHighestMarks`. Both these methods take in a `Student` object as a parameter. You can see how we can customize the WSDL service interface using the annotations. The sample code for this sample is shown here. The `@WebService` annotation specifies that the name of this service should be `StudentMarksService` with the namespace `http://apache.org/jaxws/sample`. The WSDL style is declared as document-literal, and the parameters are wrapped.

StudentMarksService.java class

```
package org.apache.jaxws.sample;

import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.soap.SOAPBinding;

@WebService(
        name = "StudentMarks",
        serviceName = "StudentMarksService",
        targetNamespace = "http://apache.org/jaxws/sample"
)
@SOAPBinding(
        style = SOAPBinding.Style.DOCUMENT,
        use = SOAPBinding.Use.LITERAL,
        parameterStyle = SOAPBinding.ParameterStyle.WRAPPED
)
public class StudentMarksService {

    public double getAverage(); // see details below
    public int getHighestMarks();// see details below
}
```

The JAX-WS Web Service operation, `getAverage`, is shown next. This method calculates the average marks of the specified student. The `@WebMethod` annotation declared that the web service operation name should be `computeAverage` and the action of this operation is `urn:getAverage`:

```
    @WebMethod(
            operationName = "computeAverage",
            action = "urn:getAverage"
    )
    @WebResult(
            name = "average",
            targetNamespace = "http://apache.org/jaxws/sample"
    )
    public double getAverage(
            @WebParam(name = "student", targetNamespace =
                                        "http://apache.org/jaxws/
sample")
            Student student) {
        double totalMarks = 0;
        for (int i = 0; i < student.getMarks().length; i++) {
            totalMarks += student.getMarks()[i];
        }
        return totalMarks / student.getMarks().length;
    }
```

JAX-WS Web Service operation, `getHighestMarks`, is shown next. This method returns the highest marks obtained by the specified student. The `@WebMethod` annotation declared that the web service's operation name should be `computeHighestMarks` and the action of this operation is `urn:getHighestMarks`:

```
    @WebMethod(
            operationName = "computeHighestMarks",
            action = "urn:getHighestMarks"
    )
    @WebResult(
            name = "highest",
            targetNamespace = "http://apache.org/jaxws/sample"
    )
    public int getHighestMarks(
            @WebParam(name = "student", targetNamespace =
                              "http://apache.org/jaxws/sample")
            Student student) {
        int highest = 0, temp;
        for (int i = 0; i < student.getMarks().length; i++) {
```

```
                temp = student.getMarks()[i];
                if (temp > highest) {
                    highest = temp;
                }
            }
            return highest;
        }
```

The `Student` class is a simple JavaBean containing the name, age, and marks of the respective students. It is declared as follows:

Student.java class

```
package org.apache.jaxws.sample;
public class Student {
    private String name;
    private int age;
    private int[] marks;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public int[] getMarks() {
        return marks;
    }

    public void setMarks(int[] marks) {
        this.marks = marks;
    }
}
```

Compile the given classes, create a JAR file out of the compiled code, and deploy it in the Axis2 `repository/servicejars` directory. Start Axis2 and check out the WSDL by visiting the URL `http://localhost:8080/axis2/services/StudentMarksService?wsdl`.

Contract first development with JAX-WS

The primary purpose of JAX-WS is to allow the Java developers to develop web services with the convenience provided by the Java language. However, there may be situations where the contract or WSDL has already been designed and you are forced to use that WSDL. In this section, you will see how an existing WSDL document can be used for developing a JAX-WS service or client. You will be using the wsimport tool that is shipped with the JDK.

The Java artifacts created using the wsimport tool are:

- Service Endpoint Interface (SEI)
- Service class
- Exception class that is mapped from the `wsdl:fault` class (if any)
- JAXB generated type values (that are Java classes mapped from XML schema types)

This is what you will see if you run `wsimport -help`:

```
Usage: wsimport [options] <WSDL_URI>
where [options] include:
-b <path>     specify jaxws/jaxb binding files or additional schemas
(Each         <path> must have its own -b)
-B<jaxbOption>   Pass this option to JAXB schema compiler
-catalog <file>   specify catalog file to resolve external entity
references           supports TR9401, XCatalog, and OASIS XML Catalog
format.
-d <directory>   specify where to place generated output files
-extension     allow vendor extensions - functionality not specified
by the         specification.  Use of extensions may result in
applications that         are not portable or may not interoperate
with other             implementations
-help       display help
-httpproxy:<host>:<port> specify a HTTP proxy server (port defaults to
8080)
-keep       keep generated files
-p <pkg>     specifies the target package
-quiet     suppress wsimport output
-s <directory>   specify where to place generated source files
-target <version>   generate code as per the given JAXWS specification
version.         version 2.0 will generate compliant code for JAXWS
2.0 spec.
-verbose     output messages about what the compiler is doing
-version     print version information
-wsdllocation <location> @WebServiceClient.wsdlLocation value
Examples:
  wsimport stock.wsdl -b stock.xml -b stock.xjb
```

```
wsimport -d generated http://example.org/stock?wsdl
```

As shown in the second example, generate `.java` files as well, if you want to create a service out of this. There is a generated interface of your web service among these generate classes. Just implement that interface and that will be your web service class. Then add the `@WebService` annotation at the top as follows.

We will see how to generate a service for the following WSDL 1.1 document (`StudentsMarksService.wsdl`):

```
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:tns="http://apache.org/jaxws/sample1"
             name="StudentMarksService"
             targetNamespace="http://apache.org/jaxws/sample1">

  <types>…</types>
  <message>*
  <portType>…</portType>
  <binding>

      <service name="StudentMarksService">
          <port name="StudentMarksPort" binding="tns:StudentMarksPort
Binding">
            <soap:address location=
            "https://localhost/services/StudentMarksService.
StudentMarksPort/"/>
          </port>
      </service>
</definitions>
```

The XML Schema type definitions for this WSDL are shown next:

```
    <types>
        <xs:schema  xmlns:xs="http://www.w3.org/2001/XMLSchema"
                    xmlns:tns="http://apache.org/jaxws/sample1"
                    attributeFormDefault="unqualified"
                    elementFormDefault="unqualified"
                    targetNamespace="http://apache.org/jaxws/sample1">
            <xs:element name="computeAverage"
type="tns:computeAverage"/>
            <xs:element name="computeAverageResponse"
                                type="tns:computeAverageResponse"/>
            <xs:element name="computeHighestMarks"
```

```
                                type="tns:computeHighestMarks"/>
            <xs:element name="computeHighestMarksResponse"
                                type="tns:computeHighestMarksResponse"/>
            <xs:complexType name="computeAverage">
                <xs:sequence>
                    <xs:element form="qualified" minOccurs="0"
name="student"
                                            type="tns:student"/>
                </xs:sequence>
            </xs:complexType>
            <xs:complexType name="student">
                <xs:sequence>
                    <xs:element name="age" type="xs:int"/>
                    <xs:element maxOccurs="unbounded" minOccurs="0"
name="marks"
                                                nillable="true"
type="xs:int"/>
                    <xs:element minOccurs="0" name="name"
type="xs:string"/>
                </xs:sequence>
            </xs:complexType>
            <xs:complexType name="computeAverageResponse">
                <xs:sequence>
                    <xs:element form="qualified" name="average"
type="xs:double"/>
                </xs:sequence>
            </xs:complexType>
            <xs:complexType name="computeHighestMarks">
                <xs:sequence>
                    <xs:element form="qualified" minOccurs="0"
name="student"
                                            type="tns:student"/>
                </xs:sequence>
            </xs:complexType>
            <xs:complexType name="computeHighestMarksResponse">
                <xs:sequence>
                    <xs:element form="qualified" name="highest"
type="xs:int"/>
                </xs:sequence>
            </xs:complexType>
        </xs:schema>
    </types>
```

The message definitions of this WSDL are shown next:

```
    <message name="computeHighestMarks">
        <part name="parameters" element="tns:computeHighestMarks">
        </part>
    </message>
    <message name="computeAverage">
        <part name="parameters" element="tns:computeAverage">
        </part>
    </message>
    <message name="computeHighestMarksResponse">
        <part name="parameters" element="tns:computeHighestMarksRespo
nse">
        </part>
    </message>
    <message name="computeAverageResponse">
        <part name="parameters" element="tns:computeAverageResponse">
        </part>
    </message>
```

The `portType` definition of this service in the WSDL is shown here:

```
   <portType name="StudentMarks">
        <operation name="computeAverage">
            <input message="tns:computeAverage">
            </input>
            <output message="tns:computeAverageResponse">
            </output>
        </operation>
        <operation name="computeHighestMarks">
            <input message="tns:computeHighestMarks">
            </input>
            <output message="tns:computeHighestMarksResponse">
            </output>
        </operation>
    </portType>
```

The binding definition in this WSDL is as follows:

```
    <binding name="StudentMarksPortBinding" type="tns:StudentMarks">
        <soap:binding style="document"
          transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="computeAverage">
            <soap:operation soapAction="urn:getAverage"/>
            <input>
                <soap:body use="literal"/>
```

```
            </input>
            <output>
                <soap:body use="literal"/>
            </output>
        </operation>
        <operation name="computeHighestMarks">
            <soap:operation soapAction="urn:getHighestMarks"/>
            <input>
                <soap:body use="literal"/>
            </input>
            <output>
                <soap:body use="literal"/>
            </output>
        </operation>
    </binding>
```

Let us generate the code for this WSDL using the wsimport tool, as shown here:

```
wsimport -s wsdl2java StudentsMarksService.wsdl
```

This will generate the `org.apache.jaxws.sample1.StudentMarksPort` and `org.apache.jaxws.sample1.Student` classes. Now we will write a sample web service using those generated classes:

```
package org.apache.jaxws.sample1.service;

import org.apache.jaxws.sample1.StudentMarksPort;
import org.apache.jaxws.sample1.Student;

import javax.jws.WebService;

@WebService(
        serviceName = "StudentMarksService",
        portName = "StudentMarksPort",
        targetNamespace = "http://apache.org/jaxws/sample1",
        endpointInterface = "org.apache.jaxws.sample1.
StudentMarksPort",
        wsdlLocation = "StudentMarksService.wsdl"
)
public class StudentMarksImpl implements StudentMarksPort {

    public double computeAverage(Student student) {
        double totalMarks = 0;
```

```
            for (int i = 0; i < student.getMarks().size(); i++) {
                totalMarks += student.getMarks().get(i);
            }
            return totalMarks / student.getMarks().size();
        }
        public int computeHighestMarks(Student student) {
            int highest = 0, temp;
            for (int i = 0; i < student.getMarks().size(); i++) {
                temp = student.getMarks().get(i);
                if (temp > highest) {
                    highest = temp;
                }
            }
            return highest;
        }
    }
```

Now compile your project and create the JAR archive to be deployed. Note that you have to include your WSDL file at the root level of your JAR archive (find the completed JAR archive under attachments). Finally, deploy the service in Axis2 by copying the JAR file into the `repository/servicejars` directory and invoking it.

Client-side JAX-WS

In this section, we will look at how to write Web service clients using JAXWS annotations. The dynamic client API for JAX-WS is called the dispatch client (`javax.xml.ws.Dispatch`). The dispatch client is an XML messaging-oriented client. The data is sent in either the PAYLOAD or MESSAGE mode. When using the PAYLOAD mode, the dispatch client is only responsible for providing the contents of the `<soap:Body>` and JAX-WS adds the `<soap:Envelope>` and `<soap:Header>` elements. When using the MESSAGE mode, the dispatch client is responsible for providing the entire SOAP envelope including the `<soap:Envelope>`, `<soap:Header>`, and `<soap:Body>` elements, and JAX-WS does not add anything additional to the message. The dispatch client supports asynchronous invocations using a callback or polling mechanism. The static client programming model for JAX-WS is called the proxy client. The proxy client invokes a web service based on a Service Endpoint interface (SEI), which must be provided.

Let us look at the Dispatch client and the Proxy client in detail.

The Dispatch client

You will be using this client when you want to work at the XML message level or when you want to work without any generated artifacts at the JAX-WS level. The Dispatch client API requires application clients to construct messages or payloads as XML, which requires a detailed knowledge of the message or message payload. The Dispatch client supports the following types of objects:

- `javax.xml.transform.Source`: You will use Source objects to enable clients to use XML APIs directly. You can use Source objects with SOAP or HTTP bindings.
- JAXB objects: You will use JAXB objects so that clients can use JAXB objects that are generated from an XML schema to create and manipulate XML with JAX-WS applications. JAXB objects can only be used with SOAP or HTTP bindings.
- `javax.xml.soap.SOAPMessage`: You will use SOAPMessage objects so that clients can work with SOAP messages. You can only use SOAPMessage objects with SOAP bindings.
- `javax.activation.DataSource`: You will use DataSource objects so that clients can work with Multipurpose Internet Mail Extension (MIME) messages. DataSource can be used only with HTTP bindings.

The following example demonstrates how the JAX-WS Dispatch Client APIs can be used. This client talks to a `HelloService`, which contains a single operation, namely, greet. This option takes in a single string parameter. Pay special attention to the highlighted code. As you can see, the client has to construct the SOAP envelope, header, body, and body contents since the Message Service mode is used:

```
package org.apache.jaxwsclient;

import javax.xml.namespace.QName;
import javax.xml.soap.*;
import javax.xml.ws.Dispatch;
import javax.xml.ws.Service;
import javax.xml.ws.soap.SOAPBinding;

public class DispatchClient {

    public static void main(String[] args) {
        try {
            String endpointUrl = "http://localhost:8080/axis2/
services/HelloService";

            QName serviceName = new QName("http://apache.org/
types","HelloService");
            QName portName = new QName("http://apache.org/types",
                        "HelloServiceHttpSoap11Endpoint");
```

```
            /** Create a service and add at least one port to it. **/
            Service service = Service.create(serviceName);
            service.addPort(portName, SOAPBinding.SOAP11HTTP_BINDING,
endpointUrl);
            /** Create a Dispatch instance from a service.**/
            Dispatch<SOAPMessage> dispatch =
            service.createDispatch(portName, SOAPMessage.class,
Service.Mode.MESSAGE);
            /** Create SOAPMessage request. **/
            // compose a request message
            MessageFactory mf =
              MessageFactory.newInstance(SOAPConstants.SOAP_1_1_
PROTOCOL);
            // Create a message.  This example works with the
SOAPPART.
            SOAPMessage request = mf.createMessage();
            SOAPPart part = request.getSOAPPart();
            // Obtain the SOAPEnvelope and header and body elements.
            SOAPEnvelope env = part.getEnvelope();
            SOAPHeader header = env.getHeader();
            SOAPBody body = env.getBody();
            // Construct the message payload.
            SOAPElement operation = body.addChildElement("greet",
"ns2",
                                    "http://apache.org/types");
            SOAPElement age = operation.addChildElement("name");
            age.addTextNode("Packt");
            request.saveChanges();
            /** Invoke the service endpoint. **/
            SOAPMessage response = dispatch.invoke(request);
            System.out.println(response.getSOAPBody().getFirstChild().
toString());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

You will need JDK 1.5 or higher to compile the abovementioned class.

The Dynamic Proxy client

The static client programming model for JAX-WS is the called the dynamic proxy client. The dynamic proxy client invokes a Web service based on a service endpoint interface that is provided. After you create the proxy, the client application can invoke methods on the proxy just like a standard implementation of those interfaces. For JAX-WS Web service clients using the dynamic proxy programming model, use the JAX-WS tool, wsimport, to process a WSDL file and generate portable Java artifacts that are used to create a Web service client.

In the following example, the `org.apache.types.HelloService` and `org.apache.types.HelloServicePortTye` classes have been generated by using the wsimport tool that ships with the JDK. The example shows how the generated classes can be used in the ProxyClient class. As you can see, unlike in the case of the DispatchClient, the number of lines of code is much less, but you have less control over what goes on underneath the hood:

```
package org.apache.jaxwsclient;

import org.apache.types.HelloService;
import org.apache.types.HelloServicePortType;

public class ProxyClient {

    public static void main(String[] args) {
        HelloService service = new HelloService();
        HelloServicePortType portType = service.
getHelloServiceHttpSoap11Endpoint();
        System.out.println(portType.greet("Packt"));
    }

}
```

MTOM with JAX-WS Services

Message Transmission Optimization Mechanism (**MTOM**) is a method of efficiently sending binary data to and from the web services. To use MTOM in JAX-WS services, first of all, you will have to use the `BindingType` annotation to set the binding type to SOAP11 MTOM. You will be using `javax.activation.DataHandler` to represent your binary data in your parameters or return types. Let us look at an example MTOM enabled JAX-WS service. The `uploadFile` web service operation shows how binary data is used as the input parameter to an operation. The following `getBinaryTestData` operation illustrates how binary data can be used as output from an operation:

```
package org.apache.jaxws.service;
import com.sun.xml.internal.ws.util.ByteArrayDataSource;
import javax.activation.DataHandler;
import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.xml.ws.BindingType;
import javax.xml.ws.soap.SOAPBinding;
import java.io.File;
import java.io.IOException;
import java.io.InputStream;

@WebService(serviceName = "MTOMSampleService",
            targetNamespace = "http://mtom.jaxws.apache.org")
@BindingType(value = SOAPBinding.SOAP11HTTP_MTOM_BINDING)
public class MTOMService {

  @WebMethod(action = "urn:uploadFile")
  public String uploadFile(DataHandler data) {
    try {
      InputStream is = data.getInputStream();
      String msg = "File " + data.getName() + " of type " +
data.getContentType() + " successfully received";
      return msg;
    } catch (IOException e) {
      e.printStackTrace();
    }
    return null;
  }

  @WebMethod(action = "urn:getTestData")
  public DataHandler getBinaryTestData(String stmtId) {
    byte[] testData = new byte[10240];
    for (int i = 0; i < testData.length; i++) {
      testData[i] = 0x7f;
    }
    ByteArrayDataSource bds =
      new ByteArrayDataSource(testData, "application/octet-stream");
    return new DataHandler(bds);

  }
}
```

The highlighted fields show the `BindingType` being set to `SOAP11HTTP_MTOM_BINDING` and `DataHandler`. Java bean type classes can also contain `DataHandlers`. You will need JDK 1.5 or higher to compile the mentioned class.

Asynchronous invocation of JAX-WS services

JAX-WS supports asynchronous invocation of services from clients. JAX-WS provides support for both a callback and polling model when calling web services asynchronously. Both the callback model and the polling model are supported by the Dispatch client and the Proxy client.

An asynchronous invocation of a web service sends a request to the service endpoint and then immediately returns control to the client without waiting for the response to return from the service. JAX-WS asynchronous web service clients consume web services using either the callback approach or the polling approach.

Polling model

In the **polling model**, a client can issue a request and receive a response object that is polled to determine if the server has responded. When the server responds, the actual response is retrieved. The response object returns the response content when the `get` method is called. The client receives an object of type `javax.xml.ws.Response` from the `invokeAsync` method. That `Response` object is used to monitor the status of the request to the server, determine when the operation has completed, and to retrieve the response results.

Callback model

In the **callback model**, the client provides a callback handler to accept and process the inbound response object. The `handleResponse` method of the handler is called when the result is available. In order to implement an asynchronous invocation that uses the callback model, the client provides an `AsynchHandler` instance to accept and process the inbound response object. The client callback handler implements the `javax.xml.ws.AsynchHandler` interface, which contains the application code that is executed when an asynchronous response is received from the server. The `javax.xml.ws.AsynchHandler` interface contains the `handleResponse(java.xml.ws.Response)` method that is called after the runtime has received and processed the asynchronous response from the server. The response is delivered to the callback handler in the form of a `javax.xml.ws.Response` object. The response object returns the response content when the `get()` method is called.

Additionally, if an error is received, an exception is returned to the client during that call. The `response` method is then invoked according to the threading model used by the executor method, `java.util.concurrent.Executor`, on the client's `java.xml.ws.Service` instance that was used to create the Dynamic Proxy or Dispatch client instance. The executor is used to invoke any asynchronous callbacks registered by the application. Use the `setExecutor` and `getExecutor` methods to modify and retrieve the executor configured for your service.

Both these models enable the client to focus on continuing with its course of action without having to wait for a response from the service. By default, asynchronous client invocations do not have asynchronous behavior of the message exchange pattern on the wire. The programming model is asynchronous, but the exchange of request and response messages is not asynchronous. To use a truly asynchronous message exchange, the `org.apache.axis2.jaxws.use.async.mep` property must be set on the client request context with the value `true`. When this property is enabled, the messages exchanged between the client and server are different from the messages exchanged synchronously. With an asynchronous exchange, the request and response messages have WS-Addressing headers added that provide additional routing information for the messages.

Let us look at some sample code which demonstrates these concepts.

The following code segment illustrates a web service interface with methods for asynchronous requests from the client:

```
@WebService
public interface CreditRatingService {

    // Asynchronous operation with polling.
        Response<Score> getCreditScoreAsync(Customer customer);

    // Asynchronous operation with callback.
        Future<?> getCreditScoreAsync(Customer customer,
AsyncHandler<Score> handler);
}
```

The `callback` method requires a callback handler that is shown in the following code segment. When using the `callback` procedure, after a request is made, the callback handler is responsible for handling the response. The response value is a response or possibly an exception. The `Future<?>` method represents the result of an asynchronous computation and is checked to see if the computation is complete. When you want the application to find out if the request is completed, invoke the `Future.isDone()` method. Note that the `Future.get()` method does not provide a meaningful response and is not similar to the `Response.get()` method.

```
CreditRatingService svc = ...;
Future<?> invocation = svc.getCreditScoreAsync(customer,
       new AsyncHandler<Score>() {
              public void handleResponse (Response<Score> response) {
                     score = response.get();
                     ...
              }
    }
);
```

The following code segment illustrates an asynchronous polling client:

```
CreditRatingService svc = ...;
Response<Score> response = svc.getCreditScoreAsync(customerTom);
while (!response.isDone()) {
       // Do something while we wait.
}
score = response.get();
```

Summary

In this chapter, we looked at developing web services and web service clients using the JAX-WS standard. We had an in-depth look at the different JAX-WS annotations and their usages. We also looked at several code samples that demonstrated how easy it is to develop web services and clients using JAX-WS.

In the next chapter, we will be looking at clustering Apache Axis2, which will allow you to deploy Axis2 in large scale production deployments.

12
Axis2 Clustering

Clustering for high availability and scalability is one of the main requirements of any enterprise deployment. This is also true for Apache Axis2. High availability refers to the ability to serve client requests by tolerating failures. Scalability is the ability to serve a large number of clients sending a large number of requests without any degradation to the performance. Many large scale enterprises are adapting to web services as the de facto middleware standard. These enterprises have to process millions of transactions per day, or even more. A large number of clients, both human and computer, connect simultaneously to these systems and initiate transactions. Therefore, the servers hosting the web services for these enterprises have to support that level of performance and concurrency. In addition, almost all the transactions happening in such enterprise deployments are critical to the business of the organization. This imposes another requirement for production-ready web services servers, namely, to maintain very low downtime. It is impossible to support that level of scalability and high availability from a single server, despite how powerful the server hardware or how efficient the server software is. Web services clustering is needed to solve this. It allows you to deploy and manage several instances of identical web services across multiple web services servers running on different server machines. Then we can distribute client requests among these machines using a suitable load balancing system to achieve the required level of availability and scalability.

Axis2 has extensive support for clustering. State replication amongst members in the same group as well as cluster management is supported in Axis2. Cluster management refers to managing a single group or several groups of Axis2 nodes. It is also noteworthy that third-party software such as Apache Synapse, which builds on Apache Axis2, also automatically benefits from Axis2 clustering capabilities.

In this chapter, we will be covering the following areas:

- Axis2 cluster configuration language
- Membership management schemes
- Cluster management

By the end of this chapter, you will learn the finer details of configuring an Axis2 cluster in a production deployment.

Setting up a simple Axis2 cluster

Enabling Axis2 clustering is a simple task. Let us look at setting up a simple two node cluster:

1. Extract the Axis2 distribution into two different directories and change the HTTP and HTTPS ports in the respective `axis2.xml` files.
2. Locate the "Clustering" element in the `axis2.xml` files and set the `enable` attribute to `true`. Start the two Axis2 instances using Simple Axis Server. You should see some messages indicating that clustering has been enabled. That is it! Wasn't that extremely simple?
3. In order to verify that state replication is working, we can deploy a stateful web service on both instances. This web service should set a value in the `ConfigurationContext` in one operation and try to retrieve that value in another operation. We can call the set value operation on one node, and next call the retrieve operation on the other node. The value set and the value retrieved should be equal.

Next, we will look at the clustering configuration language in detail.

Writing a highly available clusterable web service

In general, you do not have to do anything extra to make your web service clusterable. Any regular web service is clusterable in general. In the case of stateful web services, you need to store the Java serializable replicable properties in the Axis2 `ConfigurationContext`, `ServiceGroupContext`, or `ServiceContext`. Please note that stateful variables you maintain elsewhere will not be replicated. If you have properly configured the Axis2 clustering for state replication, then the Axis2 infrastructure will replicate these properties for you. In the next section, you will be able to look at the details of configuring a cluster for state replication. Let us look at a simple stateful Axis2 web service deployed in the `soapsession` scope:

```
public class ClusterableService {
    private static final String VALUE = "value";

  public void setValue(String value) {
    MessageContext.getCurrentMessageContext().getServiceContext();
    serviceContext.setProperty(VALUE, value);
   }

    public String getValue() {
    MessageContext.getCurrentMessageContext().getServiceContext();
    return (String) serviceContext.getProperty(VALUE);
    }
}
```

You can deploy this service on two Axis2 nodes in a cluster. You can write a client that will call the `setValue` operation on the first, and then call the `getValue` operation on the second node. You will be able to see that the value you set in the first node can be retrieved from the second node. What happens is, when you call the `setValue` operation on the first node, the value is set in the respective ServiceContext, and replicated to the second node. Therefore, when you call `getValue` on the second node, the replicated value has been properly set in the respective ServiceContext. As you may have already noticed, you do not have to do anything additional to make a web service clusterable. Axis does the state replication transparently. However, if you require control over state replication, Axis2 provides that option as well. Let us rewrite the same web service, while taking control of the state replication:

```
public class ClusterableService {
    private static final String VALUE = "value";

  public void setValue(String value) {
```

```
        MessageContext.getCurrentMessageContext().getServiceContext();
        serviceContext.setProperty(VALUE, value);
        Replicator.replicate(serviceContext);
       }

        public String getValue() {
        MessageContext.getCurrentMessageContext().getServiceContext();
        return (String) serviceContext.getProperty(VALUE);
        }
    }
```

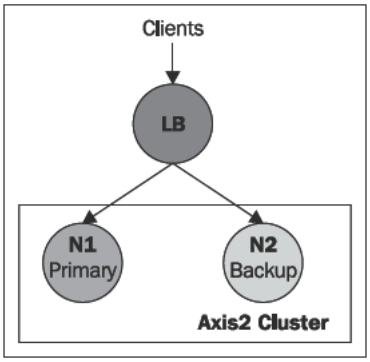
`Replicator.replicate()` will immediately replicate any property changes in the provided Axis2 context. So, how does this setup increase availability? Say, you sent a `setValue` request to node 1 and node 1 failed soon after replicating that value to the cluster. Now, node 2 will have the originally set value, hence the web service clients can continue unhindered.

Stateless Axis2 Web Services

Stateless Axis2 Web Services give the best performance, as no state replication is necessary for such services. These services can still be deployed on a load balancer-fronted Axis2 cluster to achieve horizontal scalability. Again, no code change or special coding is necessary to deploy such web services on a cluster. Stateless web services may be deployed in a cluster either to achieve failover behavior or scalability.

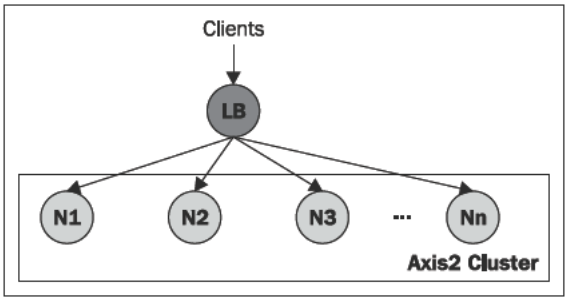
Setting up a failover cluster

A failover cluster is generally fronted by a load balancer and one or more nodes that are designated as primary nodes, while some other nodes are designated as backup nodes. Such a cluster can be set up with or without high availability. If all the states are replicated from the primaries to the backups, then when a failure occurs, the clients can continue without a hitch. This will ensure high availability. However, this state replication has its overhead. If you are deploying only stateless web services, you can run a setup without any state replication. In a pure failover cluster (that is, without any state replication), if the primary fails, the load balancer will route all subsequent requests to the backup node, but some state may be lost, so the clients will have to handle some degree of that failure. The load balancer can be configured in such a way that all requests are generally routed to the primary node, and a failover node is provided in case the primary fails, as shown in the following figure:



Increasing horizontal scalability

As shown in the figure below, to achieve horizontal scalability, an Axis2 cluster will be fronted by a load balancer (depicted by **LB** in the following figure). The load balancer will spread the load across the Axis2 cluster according to some load balancing algorithm. The **round-robin load balancing algorithm** is one such popular and simple algorithm, and works well when all hardware and software on the nodes are identical. Generally, a horizontally scalable cluster will maintain its response time and will not degrade performance under increasing load. Throughput will also increase when the load increases in such a setup. Generally, the number of nodes in the cluster is a function of the expected maximum peak load. In such a cluster, all nodes are active.



Setting up and configuring Axis2 clusters in production

This section will describe in detail how Axis clusters are set up and configured in production systems. The Axis cluster configuration is used for configuring a cluster. Understanding of the Axis2 cluster configuration language is very important when it comes to configuring different clustering setups as well as configuring and fine tuning an Axis2 cluster in production. The clustering configuration section in the `axis2.xml` file has six main parts. They are as follows:

- Clustering agent
- Parameters
- State management
- Node management
- Group management
- Members

Let us take a look at the semantics and usage of each of these configuration sections in detail.

Clustering agent

The clustering agent is responsible for initializing all clustering-related functionality of an Axis2 member or node. Typically, the initialization of a node in a cluster is handled here. It is also responsible for getting a node to join the cluster. In the default Axis2 clustering implementation, which is based on Apache Tribes, we use `org.apache.axis2.clustering.tribes.TribesClusteringAgent`.

Clustering agent parameters

In order to initialize the clustering agent, we require some parameters. Here are the parameters that are used in the current implementation.

AvoidInitiation

This parameter indicates whether the cluster has to be automatically initialized when the `AxisConfiguration` is built. If set to `true`, the initialization will not be done at that stage, and some other party will have to explicitly initialize the cluster. In the case where Axis2 is embedded by third parties, they may need to take control over the cluster initialization process. Hence this parameter has been provided. For example, we can set this parameter to `true` so that the cluster can be initialized after the server has completely started and the relevant transports have been initialized.

membershipScheme

The membership scheme is used in this setup. The only values supported at the moment are multicast and WKA.

- **multicast**: Multicast-based membership management. Membership is automatically discovered using multicasting. In order for this to work, multicasting should be allowed in the network and all members in the cluster should use the same multicast address (as defined by the `mcastAddress` parameter) and the exact multicast port (as defined by the `mcastPort` parameter).
- **Well-known address** (**WKA**): Membership management based on well-known address. Membership is discovered with the help of one or more nodes running at a well-known address. New members joining a cluster will first connect to a well-known node, register with the well-known node, and get the membership list from it. When new members join, one of the well-known nodes will notify the others in the group. When a member leaves the cluster, or is deemed to have left the cluster, it will be detected by the **Group Membership Service** (**GMS**) using a TCP ping mechanism. WKA-based membership is necessary when multicast-based membership discovery is not possible. For example, on Amazon EC2, multicasting is not allowed and there is no control over the IP address assigned to EC2 instances. Hence, in such a scenario, WKA-based membership discovery tends to be used.

domain

The clustering domain/group: Note that the words 'domain' and 'group' are used synonymously. There will not be any interference between nodes in different groups. Messages received from members outside the group will generally be ignored. However, special messages, such as cluster management messages or membership messages from members outside the group will be allowed.

synchronizeAll

When a web service request is received, and processed, should we update the states of all members in the cluster prior to the response being sent to the client? If the value of this parameter is set to `true`, the response to the client will be sent only after all the members have been updated. Obviously, this can be time consuming. In some cases, where this overhead may not be acceptable, the value of this parameter should be set to `false`. The risk in this case will be that all members will not be in the same state when the response is sent to the client. This condition will have to be handled by the user. In summary, this parameter defines whether or not we should synchronize the state of all members in the group before we send the response to the client.

maxRetries

This defines the maximum number of times we need to retry to send a message to a particular member before giving up and considering that node to be faulty. After reaching the `maxRetries` number of retries, we give up.

mcastAddress

This is the multicast address to be used. This parameter will only be taken into consideration when the `membershipScheme` is set to `multicast`.

mcastPort

This is the multicast port to be used. This parameter will only be taken into consideration when the `membershipScheme` is set to `multicast`.

mcastFrequency

This defines the frequency of sending membership multicast messages. The value should be specified in milliseconds. This parameter will only be taken into consideration when the `membershipScheme` is set to `multicast`.

memberDropTime

This defines the time interval within which, if a member does not respond, the member is deemed to have left the group. The value should be specified in milliseconds.

mcastBindAddress

This defines the IP address of the network interface to which the multicasting has to be bound to. Multicasting would be done using this interface. This parameter will only be taken into consideration when the `membershipScheme` is set to `multicast`. Note that this can be different from the `localMemberHost` parameter. So we can listen for point-to-point messages on the `localMemberHost` network interface, while listening for multicast messages on the network interface bound to the `mcastBindAddress`.

localMemberHost

This defines the hostname or IP address of this member. This is the IP address advertised by this member when it joins the group and sends messages. This should be set to a valid value other than localhost or 127.0.0.1. In most cases, it would suffice to set it to the IP address bound to the network interface, which is used for communicating with members in the group.

localMemberPort

This defines the TCP port used by this member. This is the port through which other members will contact this member. The default value for this port is 4000.

preserveMessageOrder

This parameter indicates that message ordering should be preserved. This will be done according to the sender order. The default value for this parameter is `true`.

atmostOnceMessageSemantics

This parameter indicates that atmost once message processing semantics need to be maintained. It guarantees that a given clustering message will be delivered at most once to the target process or node. The default value for this parameter is `true`.

properties

There are properties specific to this member. These properties are simply name-value pairs. When a member joins groups, these properties are bound to this member so that other members in the group can detect these properties. These properties can be used in cases where central cluster management needs to be carried out. For example: Let's think of a situation where an Axis2 cluster can be managed by a separate server. Each Axis2 member will provide two member properties, namely, `backendServerURL` and `mgtConsoleURL`. The `backendServerURL` property is used to connect to the backend management web services of a member. Similarly, if this member wishes to expose its own management console, it can do so by providing the "mgtConsoleURL" property. The backendServerURL property is specified using the `<property name="backendServerURL" value="https://${hostName}:${httpsPort}/services/"/>` entry. Here, `${hostName}` signifies the `hostName` member property and `${httpsPort}` denotes the `httpsPort` member property. `hostName` and `httpsPort` are two implicit member properties. `httpPort` is another implicit member property. Properties that have been previously declared can be used in subsequent property definitions. For example, we may define a new property as follows: `<property name="foo" value="${backendServerURL}/foo"/>`

State management

The `stateManager` element needs to be enabled if you are required to synchronize state across members in a cluster group. An implementation of the `org.apache.axis2.clustering.state.StateManager` needs to be provided as the value of the class attribute. In the default Apache Tribes-based implementation, we provide the `org.apache.axis2.clustering.state.DefaultStateManager` class. The user can choose not to replicate certain properties. This is done by providing a property name pattern. Let us look at the following example:

```
<stateManager class="org.apache.axis2.clustering.state.
                    DefaultStateManager" enable="true">
  <replication>
    <defaults>
     <exclude name="local_*"/>
     <exclude name="LOCAL_*"/>
    </defaults>

    <context class="org.apache.axis2.context.ConfigurationContext">
     <exclude name="UseAsyncOperations"/>
     <exclude name="SequencePropertyBeanMap"/>
    </context>

    <context class="org.apache.axis2.context.ServiceGroupContext">
     <exclude name="my.sandesha.*"/>
    </context>
```

```
      <context class="org.apache.axis2.context.ServiceContext">
       <exclude name="my.sandesha.*"/>
      </context>
    </replication>
  </stateManager>
```

In the preceding example, specifying `<exclude name="local_*"/>` will exclude all properties having the names prefixed with `local_` from replication and `<exclude name="*_local"/>` will exclude all properties having the name suffixed with `_local` from replication. It follows that `<exclude name="*"/>` excludes all properties from replication. Excluding all properties from replication may be useful when one needs to replicate only the properties in a certain contexts. For example, if a user wishes to replicate properties only in the `serviceContext`, under the exclusion entries of `org.apache.axis2.context.ConfigurationContext` and `org.apache.axis2.context.ServiceGroupContext`, the `<exclude name="*"/>` entry should be added.

The exclusion patterns under the defaults element will be applicable to all contexts. Hence `<exclude name="local_*"/>`, under the `defaults` element, means that all the properties prefixed with `local_` in all the properties having names prefixed with `local_` will be excluded from replication. Shown next is a sample `stateManager` entry.

Node management

The `nodeManager` element needs to be enabled in order to have node management functionality. An implementation of the `org.apache.axis2.clustering.NodeManager` interface needs to be provided as the value of the class attribute, as shown next:

```
<nodeManager class="org.apache.axis2.clustering.management.
DefaultNodeManager"
       enable="true"/>
```

Group management

When this member is deployed as a cluster manager, the `groupManagement` element needs to be enabled. A group management agent, which is an instance of the `org.apache.axis2.clustering.management.GroupManagementAgent` interface, needs to be specified for each group (`applicationDomain`) that is being managed. The example configuration shows how cluster management has been enabled for two groups—`group1` and `group2`:

```
<groupManagement enable="true">
  <applicationDomain name="group1" description="This is the first
```

```
group" agent="org.apache.axis2.clustering.management.
  DefaultGroupManagementAgent"/>

  <applicationDomain name="group2" description="This is the second
    group" agent="org.apache.axis2.clustering.management.
    DefaultGroupManagementAgent"/>
</groupManagement>
```

Static members

The `members` element is used for specifying static or well-known members. The `hostName` and the primary port of these members need to be specified. The sample member configuration shows two static members with different IP addresses and ports:

```
<members>
  <member>
    <hostName>10.100.1.202</hostName>
    <port>4000</port>
  </member>
  <member>
    <hostName>10.100.1.154</hostName>
    <port>4001</port>
  </member>
</members>
```

Full configuration

Shown next is a typical clustering configuration in Axis2, which uses the Apache-Tribes-based implementation. In this configuration, we have enabled clustering by setting the value of the `enable` attribute of the clustering element to `true`. We are using multicast-based membership management scheme. The value of the `synchronizeAll` parameter has been set to `true`; hence the responses to web service requests will only be sent after the state changes have been updated on all the nodes. Even though static members have been defined, they will be ignored as we are using a multicast-based scheme. However, if we were using the well-known address-based scheme (WKA), then the static members will be looked up during cluster initialization. Similarly, the parameters pertaining to multicast-based membership such as `mcastAddress`, `mcastPort`, and `mcastFrequency` are applicable only in the case of multicast-based membership management. Group management is only relevant to a node that acts as a cluster manager, hence is disabled in this configuration. We have enabled state replication by enabling the `stateManager` element.

```
<clustering class="org.apache.axis2.clustering.tribes.
TribesClusteringAgent"                        enable="true">
  <parameter name="AvoidInitiation">true</parameter>
  <parameter name="membershipScheme">multicast</parameter>
  <parameter name="domain">apache.axis2.domain</parameter>
  <parameter name="synchronizeAll">true</parameter>
  <parameter name="maxRetries">10</parameter>
  <parameter name="mcastAddress">228.0.0.4</parameter>
  <parameter name="mcastPort">45564</parameter>
  <parameter name="mcastFrequency">500</parameter>
  <parameter name="memberDropTime">3000</parameter>
  <parameter name="mcastBindAddress">10.100.1.20</parameter>
  <parameter name="localMemberHost">10.100.1.20</parameter>
  <parameter name="localMemberPort">4000</parameter>
  <parameter name="preserveMessageOrder">true</parameter>
  <parameter name="atmostOnceMessageSemantics">true</parameter>
  <members>
   <member>
     <hostName>10.100.1.21<hostName>
     <port>4000</port>
   </member>
   <member>
     <hostName>10.100.1.22</hostName>
     <port>4001</port>
   </member>
  </members>
  <groupManagement enable="false">
   <applicationDomain name="apache.axis2.application.domain"
             description="Axis2 group"
agent="org.apache.axis2.clustering.management.
DefaultGroupManagementAgent"/>
  </groupManagement>
  <nodeManager class="org.apache.axis2.clustering.management.
DefaultNodeManager"
       enable="false"/>
  <stateManager class="org.apache.axis2.clustering.state.
DefaultStateManager"
          enable="true">
     <replication>
       <defaults>
        <exclude name="local_*"/>
        <exclude name="LOCAL_*"/>
       </defaults>
       <context class="org.apache.axis2.context.ConfigurationContext">
         <exclude name="UseAsyncOperations"/>
         <exclude name="SequencePropertyBeanMap"/>
       </context>
       <context class="org.apache.axis2.context.ServiceGroupContext">
         <exclude name="my.sandesha.*"/>
       </context>
```

```
          <context class="org.apache.axis2.context.ServiceContext">
            <exclude name="my.sandesha.*"/>
          </context>
        </replication>
      </stateManager>
    </clustering>
```

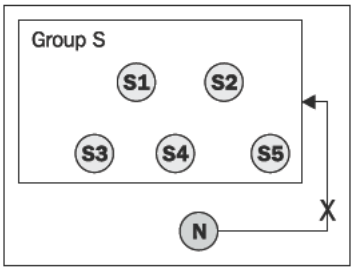
Membership schemes

The term membership scheme refers to the manner in which group membership is managed in a cluster. A cluster consists of several processes, and each of these processes is known as a member. In the context of Axis2, each member is an Axis2 process. The Axis2 clustering implementations support several membership schemes. Next, we will take a look at the three membership schemes supported by Axis2, and how the clusters are configured to support these membership schemes. We will be looking at:

- Static membership
- Dynamic membership
- Hybrid membership

Static membership

In this scheme, only a defined set of members can be in a group. The Group Membership Service (GMS) will detect members joining or leaving the group. External members cannot join the group. Each node may obtain group member details from a central repository or configuration file.



In the figure, **S1** up to **S5** are static members. These static members can join or leave the group at any time. However, these members need to be specified in the `axis2.xml` file. A new member, **N**, which is not specified in the `axis2.xml` file, cannot join the group in general. Later, we will look at hybrid membership, where the new member, **N**, will be allowed to join the group, if it knows the details of at least one static member. The `members` configuration section in the `axis2.xml` of each member will look like this:

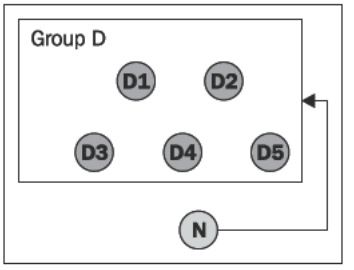
```
<members>
 <member>
  <hostName>S1</hostName>
  <port>4000</port>
 </member>
 <member>
  <hostName>S2</hostName>
  <port>4000</port>
 </member>
 <member>
  <hostName>S3</hostName>
  <port>4000</port>
 </member>
 <member>
  <hostName>S4</hostName>
  <port>4000</port>
 </member>
 <member>
  <hostName>S5</hostName>
  <port>4000</port>
 </member>
</members>
```

All the members should belong to the same group or domain. This will be specified using the `domain` parameter in the `axis2.xml` clustering configuration section. For example:

```
<parameter name="domain">Group S<parameter>
```

Dynamic membership

In this scheme, membership is not predefined. Members can join a group by specifying the appropriate group name, and also leave the group. The Group Management Service (GMS) will detect new members joining or leaving. Group membership information may be obtained from the GMS.



The members in a dynamic group need not be specified in the `axis2.xml` file. Membership will be discovered using multicasting. However, all members should use the same multicast IP address, multicast port, and domain name, in order to belong to the same group. As shown in the previous figure, a new member, **N**, can join the group by multicasting its details to the multicast socket of the group and specifying that it belongs to the same group, which is **Group D**.

The multicast address and multicast port are specified using the following parameters in the `axis2.xml` clustering configuration section:

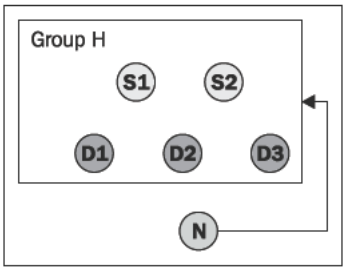
```
<parameter name="mcastAddress">228.0.0.4</parameter>
<parameter name="mcastPort">45564</parameter>
```

The domain is specified as follows:

```
<parameter name="domain">Group D</parameter>
```

Hybrid membership

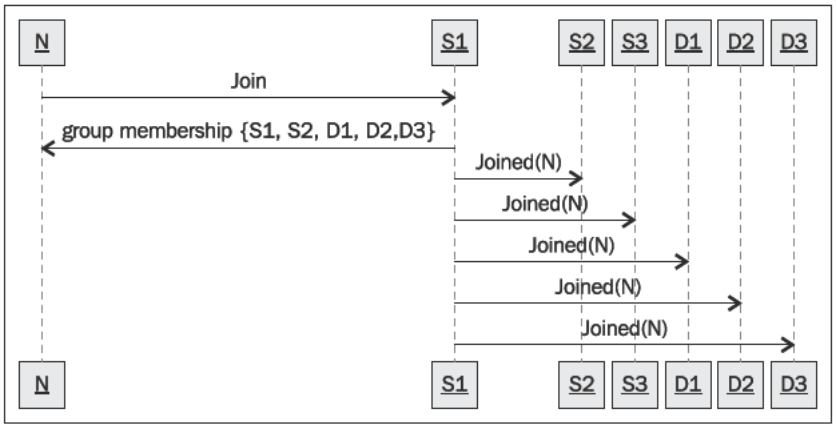
This scheme is also called well-known-addressed- or WKA-based membership. In this scheme, there are a set of well-known members. We can consider these members as belonging to a static group. External members can join this group by notifying one of the well-known members. These external members can then get the current group membership from this well-known member. When new members join the group, well-known members will notify all other members. When members leave the group, the GMS can detect this event.



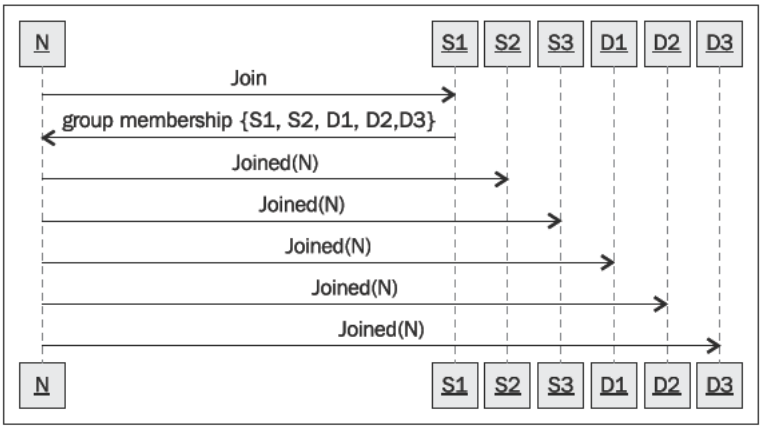
The static members are specified in the `axis2.xml` file of each member as follows:

```
<members>
  <member>
   <hostName>S1</hostName>
   <port>4000</port>
  </member>
  <member>
   <hostName>S2</hostName>
   <port>4000</port>
  </member>
</members>
```

As shown in the preceding figure, a new member, **N**, belonging to the domain **Group H** and having knowledge about at least one static member in the group, can join the group. This new member will first contact one of the static members and learn about the group membership. The static member will inform **N** about the group membership {**S1**, **S2**, **D1**, **D2**, **D3**} and then notify each member in the group about **N** joining the group, as shown in the following sequence diagram:



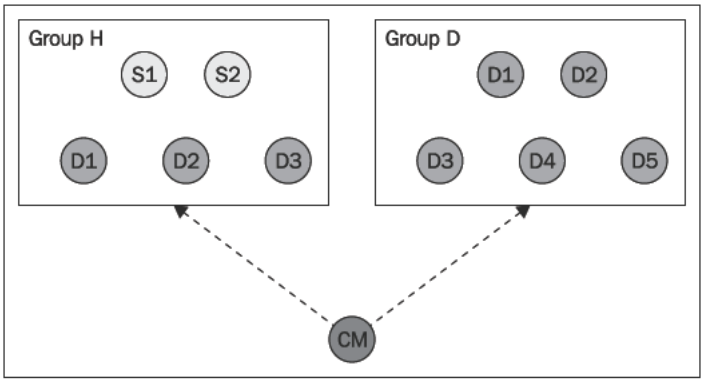
An alternative way is for **N** to inform everyone in the group that it has joined, once it receives the membership details from the static member, as shown in the following sequence diagram:



Typically, there should be at least two static members to avoid the single point of failure scenario. If there is only a single static member and that member fails, no new members will be able to join the group. However, a dynamic remapping of the IP address of the failed static member can be done. Suppose there is only a single static member; on failure of this static member, another member in the group can remap the IP address of the failed static member to itself, thereby becoming the new sole static member.

Cluster management

Axis2 can run as a cluster manager for managing several groups in a cluster, as depicted in the following figure. If multicast-based membership discovery is used, the **Cluster Manager** (CM) needs to be running on the same multicast domain and use the same multicast address and port used by members in all the groups.



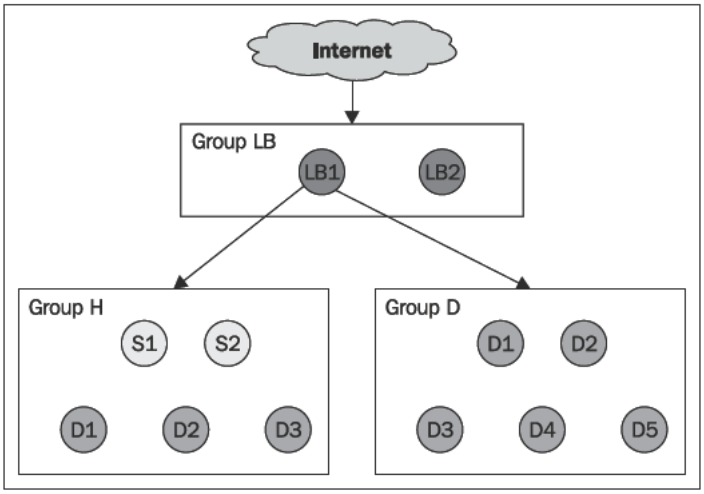
The cluster manager needs to have the following configuration in order to manage the two groups, **Group H** and **Group D**, as shown in the previous figure:

```
<groupManagement enable="true">
  <applicationDomain name="GroupH"
        description="This is the first group" agent="org.apache.
axis2.clustering.management.DefaultGroupManagementAgent"/>

  <applicationDomain name="GroupD"
        description="This is the second group" agent="org.apache.
axis2.clustering.management.DefaultGroupManagementAgent"/>
</groupManagement>
```

Highly available load balancing

In order to avoid single points of failure, the worker Axis2 clusters—**Group H** and **Group D**—can be fronted by a load balancer cluster, **Group LB**, as shown in the following figure. One load balancer in **Group LB** will be the primary load balancer. The Apache Synapse load balancer can be deployed in this manner, and uses the underlying Axis2 clustering mechanism. The web service clients will talk to the endpoints in the primary load balancer and will be unaware of the backend worker node Axis2 clusters.



The Axis2 clustering management API

Axis2 clustering is designed in a way that all functionality is abstracted out by a set of interfaces. By implementing these interfaces and providing the implementation classes in the clustering configuration section, any different clustering implementation can be plugged in. The default clustering implementation in Axis2 is based on Apache Tribes (Apache Tribes n.d.), the popular group management framework used by Apache Tomcat (Apache Tomcat n.d.). One may be required to use a group management framework of one's choice, in which case, he/she can implement these interfaces and then plug them using the clustering configuration section in the `axis2.xml` file. We will be taking a look at the usages of these interfaces later in this chapter.

The Axis2 clustering APIs address two concerns—state replication and management. **State replication** refers to the synchronizing of states related to different members in a cluster group. **Management** can again be divided into two aspects—**group management** and **node management**.

The four interfaces that may be used by an Axis2 clustering implementation are:

- `org.apache.axis2.clustering.ClusteringAgent`
- `org.apache.axis2.clustering.state.StateManager`
- `org.apache.axis2.clustering.management.NodeManager`
- `org.apache.axis2.clustering.management.GroupManagementAgent`

Let us take a look at these four interfaces in detail.

org.apache.axis2.clustering.ClusteringAgent

This is the main interface in the Axis2 clustering implementation. In order to plug in a new clustering implementation, this interface has to be implemented. It is mandatory to provide an implementation of this interface.

The `ClusteringAgent` is responsible for initializing all clustering-related functionalities of an Axis2 member or node. Generally, the initialization of a node in a cluster is handled here. It is also responsible for getting this node to join the cluster. This Axis2 node should not process any web services requests until it successfully joins the cluster. Generally, this node will also need to obtain state information and/or configuration information from a neighboring node. State information needs to be obtained, as prior to joining the group this new member needs to be in sync with the other members with respect to the state. It is also possible that configuration changes have taken place. For example, new service deployments or service undeployments may have taken place after nodes in a group were initialized. Hence, it is essential that all configuration changes are kept in sync across a group. This interface is also responsible for properly initializing the `org.apache.axis2.clustering.state.StateManager`, `org.apache.axis2.clustering.management.NodeManager`, and `org.apache.axis2.clustering.management.GroupManagementAgent` implementations. In the case of a static membership scheme, members are read from the `axis2.xml` file and added to the `ClusteringAgent`. Later in the chapter, we will take a look at different membership schemes supported by Axis2.

In the `axis2.xml` file, the instance of this interface is specified using the `clustering` class attribute.

There can also be several `parameter` elements, which are children of the `clustering` element in the `axis2.xml` file. Generally, these parameters will be specific to the `ClusteringAgent` implementation. In the default Apache Tribes-based implementation, there are several parameters that are required to initialize the Tribes Channel properly.

org.apache.axis2.clustering.state.StateManager

This interface is responsible for handling state replication. This is an optional interface and if state replication is not required, then this interface can be omitted. The word 'state' here means the serializable values stored in the Axis2 context hierarchy which need to be kept in sync across all the members of a cluster group. Property changes (state changes) in the Axis2 context hierarchy in the node that runs this `StateManager` will be propagated to all other nodes in its group. In the default Axis2 clustering implementation, we have only enabled replication of serializable objects stored in the Apache Axis2 `ConfigurationContext`, `ServiceGroupContext`, and `ServiceContext`. Hence, if a user requires state replication, it is this user's responsibility to store these values in the proper Axis2 contexts. Generally, web services authors do not have to handle state replication, as it is handled by the clustering implementation just before request completion. This is done at the Axis2 `MessageReceivers`. However, if the developer writes his own `MessageReceiver`, he will need to call the `org.apache.axis2.clustering.state.Replicator#replicate()` method. Also, note that at any point, `org.apache.axis2.clustering.state.Replicator#replicate()` can be called, if the developer wishes to force state replication.

It is not mandatory to have a `StateManager` in a node. If we are not interested in high availability, we may disable state replication. In such a scenario, in general, the purpose of a clustered deployment is to achieve scalability. In addition, one may also enable clustering without state replication simply to utilize the group communication capabilities of the underlying **Group Communication Framework** (**GCF**). In such a case, the purpose of a clustered deployment may also be management of the cluster using the underlying GCF. The implementation of this interface is set by reading the `stateManager` element in the `axis2.xml` clustering section.

org.apache.axis2.clustering.management.NodeManager

This interface is responsible for handling management of a particular member. It is not mandatory to have a `NodeManager` in a node. Node management is generally used for deploying or undeploying services across a cluster group. The implementation of this interface is set by reading the `nodeManager` element in the `axis2.xml` file.

org.apache.axis2.clustering.management.GroupManagementAgent

This is the interface through which group management events are notified and messages are sent to members in a particular group. This will only be used when a member is running in group management mode. In group management mode, a member is special and belongs to all groups that it is managing. Hence, any membership changes in the groups it manages will be notified by the underlying group management framework to the group management agent. A group management agent should be specified for each group that has to be managed by the member. In Axis2 cluster management, the cluster manager node needs to define group management agents for each group.

Apache Synapse is capable of dynamic load balancing. Here, new members can join and leave the application group, and this will be reflected in the membership-aware dynamic load balancer. The clustering configuration for the dynamic load balancer needs to be configured with group management to allow group membership discovery.

Summary

In this chapter, we looked at how Axis2 clusters can be configured. We also described in detail how the Axis2 cluster configuration language can be used for configuring production clusters. Several membership schemes, which can be used in different scenarios, were also introduced. Towards the end of the chapter, we looked at the Axis2 clustering API, which can be used for writing your own clustering mechanism, which may be based on a different group communications framework.

In the next chapter, we will be looking at some enterprise SOA deployment patterns that make use of the underlying Axis2 clustering infrastructure.

13
Enterprise Integration Patterns

In today's world, integrating multiple business systems in an enterprise is a critical requirement. Applications are no longer monolithic systems from a single vendor. Over the years, enterprise software architects have figured out a number of recurring patterns when it comes to integrating such applications. This led to the development of enterprise integration patterns. A **pattern** is a tried and tested solution for a known problem. These patterns are well documented along with the problem they are trying to solve, so that architects can identify such patterns in a given problem and apply these well known solutions, instead of having to reinvent the wheel. SOA and web services are the de facto standard when it comes to integrating disparate systems. Most of these patterns have been derived from experience gained from deployment and integration scenarios of SOA customers. Hence, a number of well-known SOA integration patterns have been developed over the years. It is noteworthy that 'SOA integration patterns' is a relatively new area.

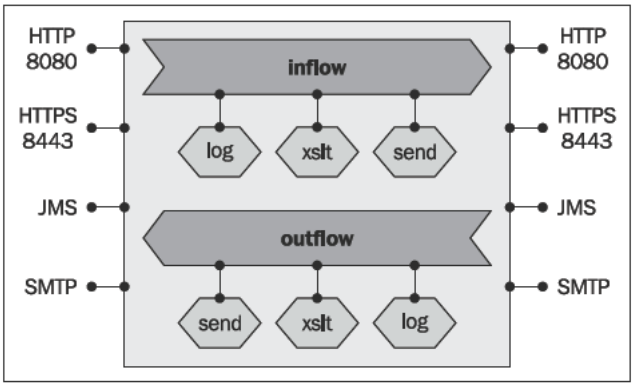
In this chapter, we will look at some SOA enterprise integration fundamentals and patterns. An **Enterprise Service Bus** (**ESB**) is a widely used middleware component that is used in SOA integration. Although there is no industry-wide accepted definition of ESB, the following is a commonly accepted definition: Any to any data connectivity and transformation (including web services) built on an advanced, proven, reliable middleware infrastructure. We will see how Apache Axis2 and the Apache Synapse Enterprise Service Bus (ESB) can be used in realizing these integration patterns. An ESB is used in all of the integration patterns we will be discussing here, as an ESB provides a central location for mediation, control, policy enforcement, auditing, routing, and mainly because with an ESB, such integration patterns can be realized with simple configuration and zero coding.

In this chapter, we will be looking at the following:

- Several open source Enterprise Service Buses (ESBs), including Apache Synapse, WSO2 ESB, and OpenESB
- Several widely used Enterprise Integration Patterns

Apache Synapse

Apache Synapse is an Enterprise Service Bus (ESB), built on top of Axis2. In fact, Apache Synapse can be deployed as an Axis2 module within Axis2. It has been designed to be simple to configure, very fast, and effective at solving many integration and gatewaying problems. Synapse has support for HTTP, SOAP, SMTP, JMS, FTP and file system transports, **Financial Information eXchange** (**FIX**) and Hessian protocols for message exchange, as well as first class support for standards such as WS-Addressing, Web Services Security, Web Services Reliable Messaging, and efficient binary attachments (MTOM/XOP). Synapse can transform messages using key standards such as XSLT, XPath, and XQuery, or simply by using Java. Synapse supports a number of useful functions out-of-the-box without programming, but it also can be extended using popular programming languages such as Java, JavaScript, Ruby, Groovy, and so on.

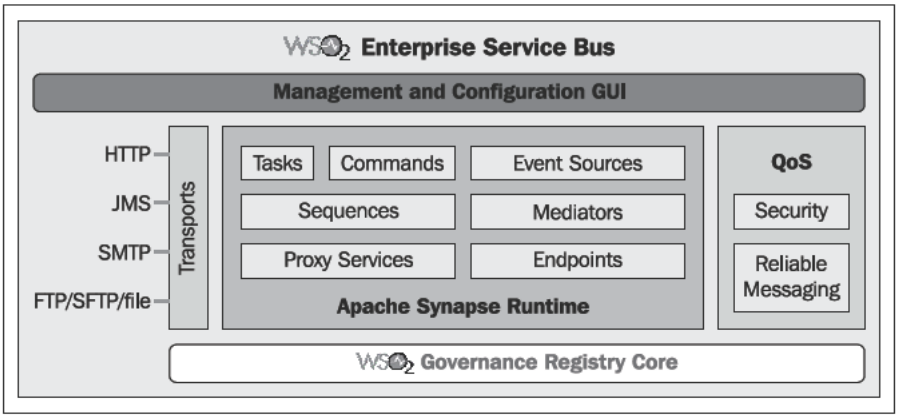


The preceding figure shows the high level architecture of Synapse. Similar to the Axis2 message flows, there is an In-Flow and Out-Flow. Once a message is received on the incoming transport, the message will go through the In-Flow, which in turn will call several mediators. The figure shows the log mediator, XSLT mediator, and send mediator, in that order. So, this message will be logged, transformed, and then sent out to a third party. Once the response to that message comes back, it will go through the Out-Flow, and again will be logged, transformed, and sent out to the original client that sent the message.

WSO2 ESB

WSO2 ESB is an open source Enterprise Service Bus (ESB) released by the open source middleware company, WSO2 Inc. It is shipped under the Apache Software License v2.0. The runtime of WSO2 ESB has been designed to be completely asynchronous, non-blocking, and streaming, based on the Apache Synapse mediation engine. This also means that WSO2 ESB has been built on top of Apache Axis2, as Apache Synapse is based on Apache Axis2. This ESB allows system administrators and SOA architects to simply and easily configure message routing, virtualization, intermediation, transformation, logging, task scheduling, load balancing, failover routing, event brokering, and many enterprise integration patterns.

WSO2 ESB supports many application layer protocols and messaging standards, collectively known as **transports**, including HTTP, HTTPS, e-mail, **Java Message Service** (**JMS**), and **Virtual File System** (**VFS**). It also supports a range of domain-specific protocols such as **Financial Information eXchange** (**FIX**), **Advanced Message Queuing Protocol** (**AMQP**), and **Health Layer 7** (**HL7**). New transports can be easily plugged into the server due to the extensibility offered by the Axis2 transports framework. The following figure illustrates how the different transports can be plugged into the ESB and how the Apache Synapse runtime fits into the WSO2 ESB:

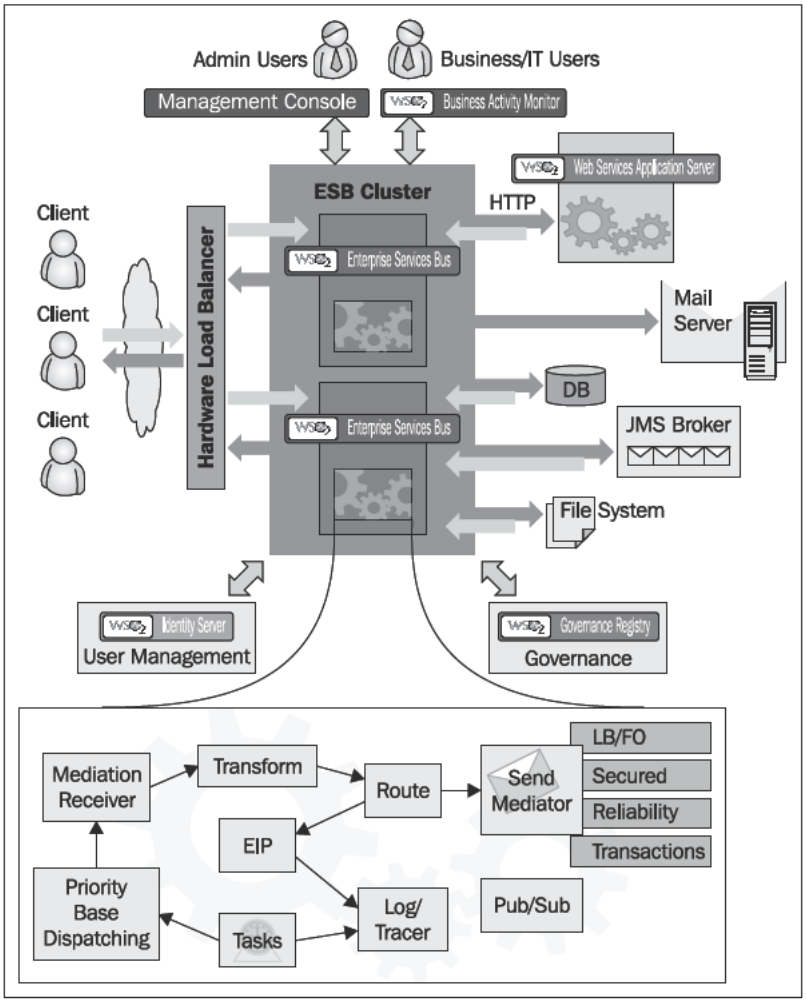


WSO2 ESB provides a comprehensive mediator library that provides various message processing and intermediation capabilities. Using this mediator library, you can implement all the widely used Message Exchange Patterns (MEP) and Enterprise Integration Patterns (EIP). There are simple mediators that provide fundamental message processing capabilities such as logging and content transformation, as well as advanced mediators that can be used to access databases, add security to message flows, and so on. In situations where the built-in mediators are not sufficient to implement a given scenario, you can write your own custom mediators using the APIs provided by WSO2 ESB. Such custom mediators can be implemented by using a variety of technologies including Java, scripting, and Spring, all of which are considered as extension points to the WSO2 ESB.

Tasks in WSO2 ESB provide you with the ability to configure scheduled jobs in your ESB and they allow the execution of internal and external commands for mediation. Qualities of Service components that implement reliable messaging and security for the proxy services and for mediation come with the Apache implementations of those two modules for Axis2, namely, Apache Rampart and Apache Sandesha2.

WSO2 ESB contains enterprise deployment features such as clustering, load balancing, high availability, monitoring, and GUI- and JMX-based management. GUI-based components provide comprehensive management, configuration, and monitoring capabilities to the ESB. The GUI is built on a layered architecture by separating the backend and frontend concerns. This allows the user to connect to multiple backends using a single GUI console. The component-based architecture of the WSO2 ESB has enhanced its loosely coupled nature with the usage of OSGi. All the components are built as OSGi bundles, which even allow advanced users to extend the capabilities of the ESB by developing and deploying their own OSGi bundles.

The WSO2 ESB can be deployed in a clustered manner in collaboration with a number of other products, as shown in the following figure:


Admin Users
Business/IT Users
Management Console
Business Activity Monitor
Web Services Application Server
ESB Cluster
HTTP
Client
Client
Client
Hardware Load Balancer
Enterprise Services Bus
Enterprise Services Bus
Mail Server
DB
JMS Broker
File System
Identity Server
User Management
Governance Registry
Governance
Mediation Receiver
Transform
Route
Send Mediator
LB/FO
Secured
Reliability
Transactions
EIP
Priority Base Dispatching
Tasks
Log/ Tracer
Pub/Sub

OpenESB

OpenESB is an open source **Java Business Integration** (**JBI**) standard-centric Enterprise Service Bus, originally from Sun Microsystems, now Oracle Inc. JBI is built on a web services model and provides a pluggable architecture for a container that hosts service producer and consumer components. Services connect to the container via binding components or can be hosted inside the container as part of a service engine. The services model used is Web Services Description Language (WSDL) 2.0. Open ESB consists of a runtime, a design time, and a management console. The runtime consists of a lightweight JBI core and several components. OpenESB comes with several components for data transformation, orchestration, and connectivity. There is support for HTTP and web services, JMS, databases, MQ Series, SAP, IMS, HL7, and others. Logic can be expressed in BPEL, EJBs, or POJOs.

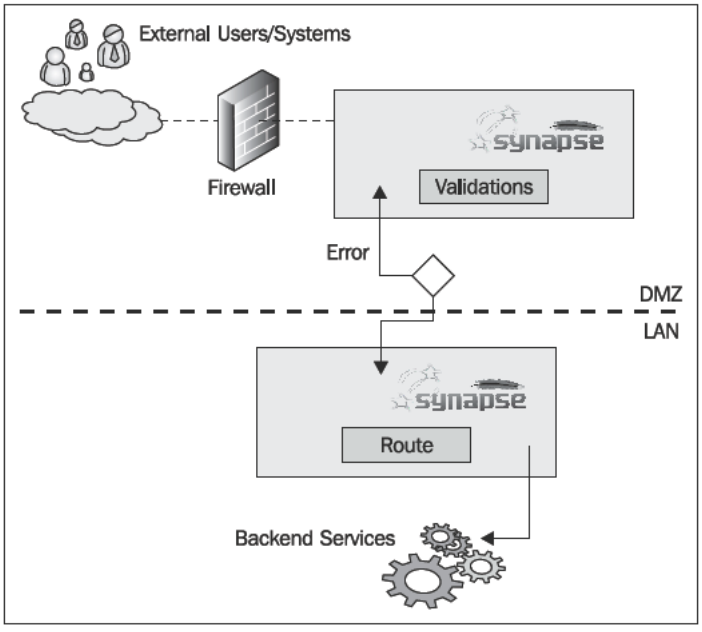
In the rest of this chapter, we will be looking at the following SOA integration patterns. Please note that Apache Synapse has been shown as the ESB in the diagrams, but it can be replaced by any of the ESBs we have discussed in this chapter:

- Protocol bridging
- External authentication and authorization
- Dynamic routing combined with auditing
- **Event Driven Architecture** (**EDA**) with **Master Data Management** (**MDM**)
- Push and pull
- Fault tolerant autoscaling with dynamic load balancing

Protocol bridging

Protocol bridging refers to the request coming on one transport protocol, and being sent in for processing on a different protocol. For example, the corporate policy may dictate that all incoming traffic should be on HTTPS. Some web services may exist that are exposed only via a transport such as JMX. In such a case, there should be a way of bridging from HTTPS to JMX. Furthermore, the backend web services, which may be deployed on Axis2, may be highly secured within a **High Security Zone** (**HSZ**), and the policies may dictate that only certain calls from the **Demilitarized Zone** (**DMZ**) can be made to those web services. We may also implement message validation and request throttling outside the highly secured LAN. This scenario is shown in the following figure. External web service clients talk to the endpoints on the ESB in the DMZ. This ESB will run validations and other Qualities of Service (QoS) functionality on these incoming messages, and if those messages meet the requirements, it will switch to a protocol that is compatible with the backend service deployment, and send the message into the LAN.

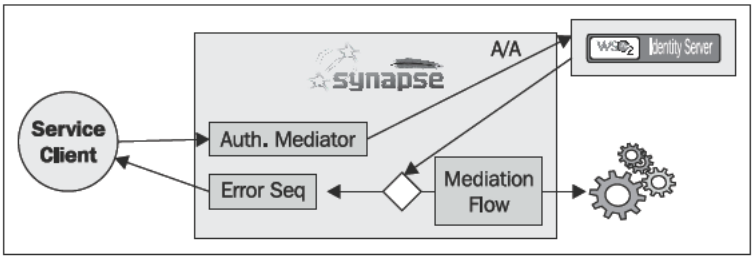
In the following figure, we have shown another optional ESB deployment, which can further deal with things such as transformations. However, that ESB can be omitted, if required, and the request can be directly routed to Axis2. Once Axis2 receives the request, it will dispatch it to the relevant web service and send the response back. The ESB in the DMZ will again perform a protocol transformation, and send the response back to the client that originated the request.



External authentication and authorization

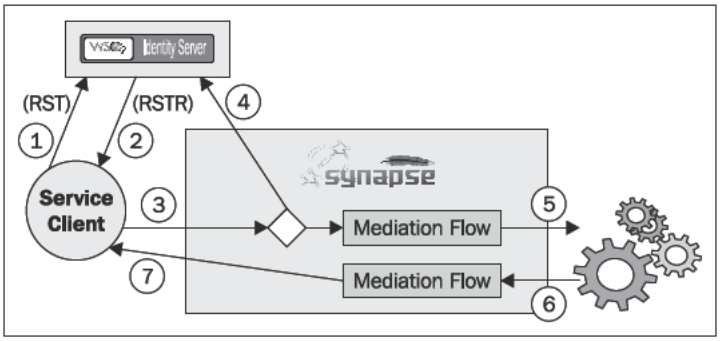
In medium to large scale enterprise deployments, authentication and authorization are handled by an external identity management server such as WSO2 Identity Server. The web service client or the server can participate in authentication and authorization.

As shown in the following figure, the web service client sends a request along with its credentials to Synapse, and the authentication and authorization mediator will talk to the identity management server and verify these credentials. Thereafter, the request will be passed on to Axis2 for processing, and Axis2 will send the response to Synapse, which will forward it to the service client that originated the request.



The second pattern is where the service client itself can talk to the identity management server, as shown in the figure below. Here, WS-Trust or some other security token-based protocol is used. The flow sequence is as follows:

1. The service client sends a **Request Security Token (RST)** message to the identity management server along with its credentials.
2. The Identity Server responds with a token (RSTR) if the credentials are successfully verified.
3. The service client then sends a request to the ESB endpoint along with the token obtained in step 2.
4. The ESB verifies the token by consulting the Identity Server.
5. If the token is accepted, the ESB forwards the request to Axis2.
6. Axis2 processes the request and sends back the response.
7. The ESB sends the response to the service client.



In these two scenarios, authentication and authorization handling can be done within a pure Axis2 instance as well, but the reason for bringing in an ESB into the picture is to handle all authentication and authorization functionality in one central place. Otherwise, if there are multiple Axis2 instances in an enterprise deployment, each of those Axis2 instances would have to separately talk to the identity management server. In addition, additional mediation steps can be introduced in the ESB, when the system evolves over time.

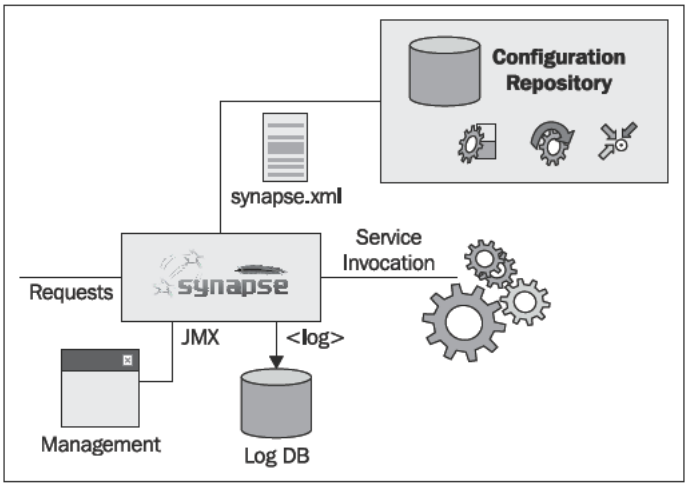
Dynamic routing combined with auditing

In some enterprise deployments, there is a requirement to route the request to different Axis2 service implementations or different Axis2 servers, depending on certain criteria or other parameters such as the time of the day. For example, during the peak hours, requests may be routed to a server having more resources in terms of memory and processing power. In some cases, the requests may need more processing power, depending on the input parameter values. Requests are dynamically routed in order to optimize resource usage and maintain throughput. Note that the service contracts will not change and only the service implementation or deployment location will change. Hence this is transparent to the client.

Dynamic routing may be combined with auditing. In this case, each of the requests, along with other relevant metadata, will be asynchronously logged into a file or database for future auditing purposes.

As shown in the following figure, the Synapse ESB is used as a dynamic router and the central point for auditing. The Synapse configuration is loaded from a configuration repository and this configuration can change over the course of the day so that requests are dynamically routed to different Axis2 service implementations or Axis2 servers. A log mediator is used for logging the incoming requests and outgoing responses to a database. In the case where the Synapse configuration is changed at preconfigured times of the day, the server is put into maintenance mode using the JMX management API.

In maintenance mode, a server will temporarily stop accepting new requests, but will continue to serve the requests that have already been received. Once all the existing requests are served, the new configuration will be loaded, and the Synapse server will be initialized with the new configuration. Reloading the configuration can either be done manually or can be automated using a Cron job or task that runs a script.



Event Driven Architecture (EDA) with Master Data Management (MDM) for Integrating Legacy Systems

This pattern is a combination of **Event Driven Architecture** (EDA) and **Master Data Management** (MDM). You will see how legacy systems can be SOA-enabled and integrated using this pattern.

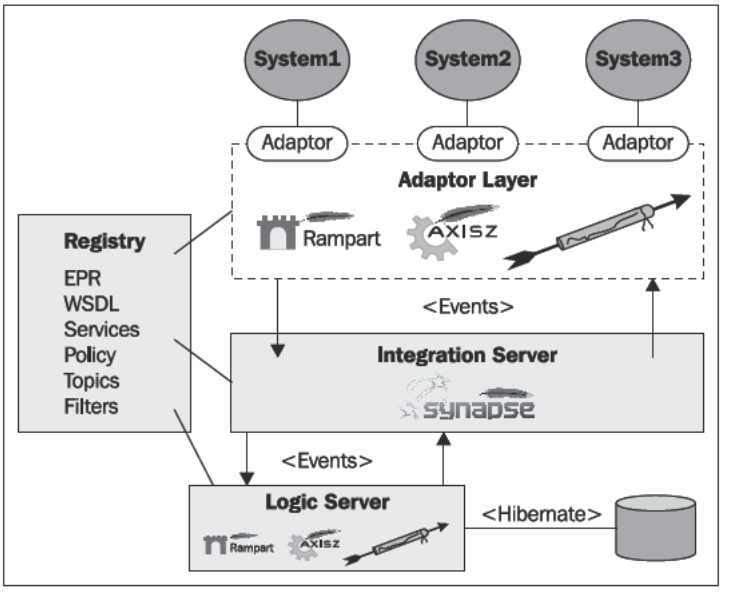
Event Driven Architecture (EDA)

Event Driven Architecture (EDA) is a software architecture pattern where the system runs based on the creation, consumption, and reaction to events. Eventing is one of the popular ways of decoupling SOA systems. In fact, an event-driven architecture is extremely loosely coupled and well distributed. An event can be anything; any message can be considered an event.

Master Data Management (MDM)

Master data refers to the information that is key to the operation of a business. This information may include data about customers, products, employees, suppliers, and so on. Master data is non-transactional in nature, but can support transactional processes and operations. Master data is often used by several functional groups and stored in different data systems across an organization and may or may not be referenced centrally. Therefore, the possibility exists for duplicate and/or inaccurate master data. MDM comprises a set of processes and tools that consistently defines and manages the master data. The objective of MDM is to provide processes for collecting, aggregating, matching, consolidating, quality-assuring, persisting, and distributing master data throughout an organization to ensure consistency and control in the ongoing maintenance and application use of this information.

Most legacy systems in an organization have different representations of the same master data. This is one of the main challenges in integrating these legacy systems. We will look at how EDA with MDM can be used for integrating such legacy systems.



We will demonstrate EDA with MDM using the example scenario illustrated in the preceding figure. Systems 1-3 are legacy systems that have different representations of the master data of the organization. For example, a Person entity is represented in one format in System 1 and in a different format in Systems 2 and 3. These systems have their own databases to store the master data. When new master data is added or existing master data is changed in one system, all other systems have to be notified so that they can correspondingly update their databases. Let us look into each of the subsystems in detail.

Adaptor layer

We use Axis2 Web Service-based adaptors to integrate these systems. The adaptors play a dual role; they natively communicate with the respective backend legacy systems and publish events when a change in the master data of the corresponding legacy system takes place. These adaptors can also be considered as converters that convert **Generic Business Objects** (**GBO**) into **Application Specific Business Objects** (**ASBO**). Each adaptor is interested in changes only to specific master data types. Hence, each of these adaptors will subscribe to specific event types, represented by different topic spaces. The Axis2 Savan module-based WS-Eventing is used here. When events to these specific topics are published by the event server, they will be picked up by the relevant adaptors. Similarly, when master data changes, each adaptor will publish events to the relevant topic. The adaptor web services are deployed on the same logical Axis2 server, which can physically be deployed as a cluster.

Integration server

The **Integration server** is a Synapse server that acts as an event broker. This server is responsible for managing WS-Eventing subscriptions. The adaptors deployed on the adaptor server will be the subscribers. When a particular event is received, this server will consult the logic server, and if it is a new event, it will convert that into a GBO and publish it to the relevant topics.

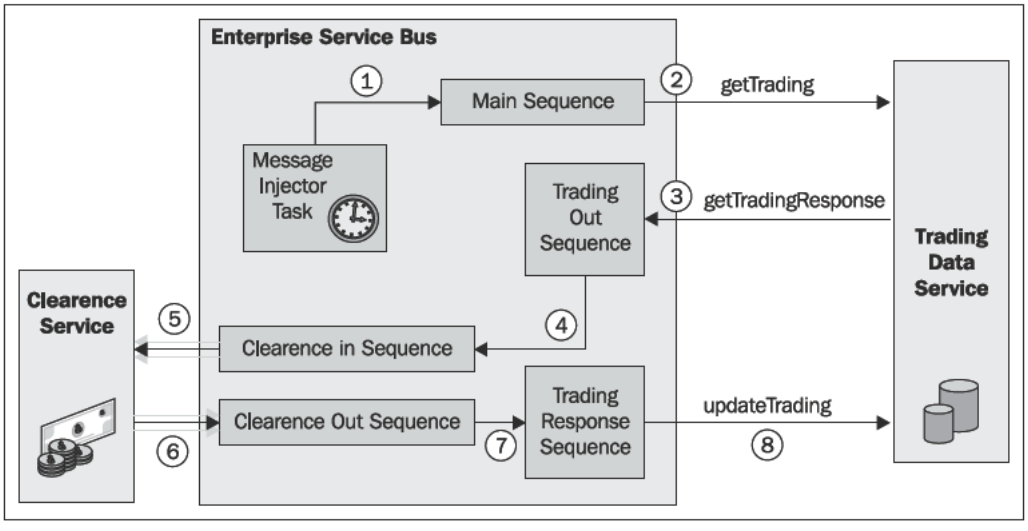
Logic server

This server determines which events are logically equivalent, hence the name. This server is introduced to deal with the 'feedback problem'. Consider this scenario: A Person is added to System 1. System 1 will publish an event. All other systems that are interested in Persons will get this event, and add it to their respective legacy systems. Since the legacy systems are unaware about how the Person got added to their systems, the corresponding adaptors will once again get triggered and will think that a new Person has been added. This will trigger another series of events. Over time, this will lead to a large number of events and eventually the entire system will crash. This is known as the feedback problem. The logic server acts as a central authority that determines whether a particular event has been already seen or not. It maintains a master data repository to keep track of events that have been already seen. The logic server is an Axis2 server.

Registry

A central **registry** is used for storing all metadata that is required by the Adaptor server, Integration server, and Logic server. The registry also helps in loose coupling between the systems. The topics and topic filters are used by the adaptors and integration servers for subscription. The endpoints are used for event dispatching.

Push and pull



Synapse supports tasks that can be run periodically. A task can be written in Java and configured in the `synapse.xml` file.

To demonstrate this integration pattern, we will use the example scenario shown in the previous figure. In this scenario, we try to reconcile the data in two different systems, namely, a trading system and clearance system, periodically. These two systems are web-service-enabled using connector services written using Axis2. In fact, these services are data services, which is a special type of service handled by the Axis2 data services deployer.

1. A Synapse message injector task periodically runs and injects a message into the Synapse main sequence.
2. This sends out a call to a trading Axis2 service, which is a data service, to obtain the trading data from the trading system.
3. The trading service sends back a response.
4. The trading response from the trading service is received by the Synapse trading out sequence, and a corresponding message is initiated and fed into the clearance in sequence.
5. The clearance in sequence sends that message to the clearance service.
6. The clearance service responds and the response is received by the clearance out sequence.
7. Reconciliation happens in the clearance out sequence and a message is sent into the trading response sequence.
8. The trading response sequence sends out a message to the trading service, the trading database is updated, and the two systems are reconciled.

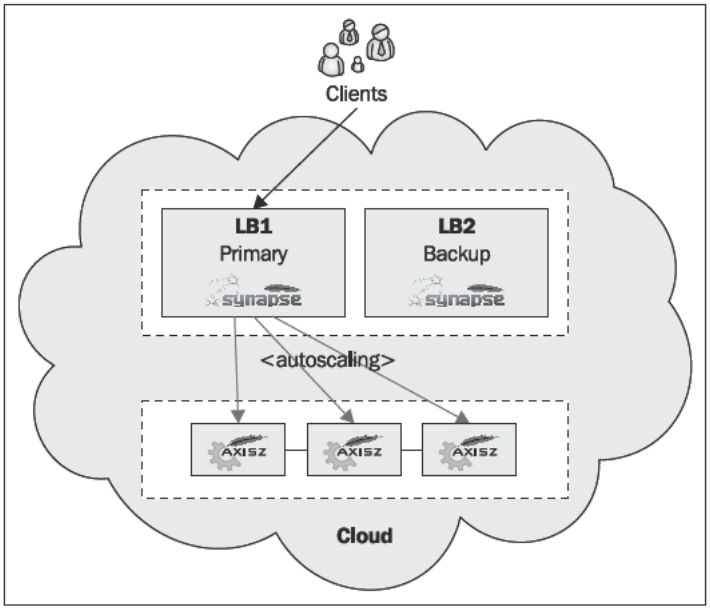
As can be seen in this scenario, a task pushes or initiates the entire flow. In response to this, data gets pulled in from different systems, reconciled, and pushed back to the relevant systems.

Fault tolerant autoscaling with dynamic load balancing

Fault tolerance, high availability, and scalability are essential prerequisites for any enterprise application deployment. One of the major concerns of enterprise application architects is avoiding single points of failure. **Autoscaling** refers to the behavior where the system scales up when the load increases and scales down when the load decreases. Such an approach is essential for cloud deployments such as Amazon EC2 where the charge is based on the actual computing power consumed. Ideally, from the clients' point of view, in an autoscaling system, the response time should be constant and the overall throughput of the system should increase. Dynamic load balancing refers to a mechanism where the load balancer itself uses group communication and group membership mechanisms to discover the domains across which the load is distributed. In a traditional setup, a single load balancer fronts a group of application nodes. In such a scenario, the load balancer can be a single point of failure. Traditionally, techniques such as Linux HA have been used to overcome this. However, such traditional schemes have quite a bit of overhead and also require the backup system to be in close proximity to the primary system.

As shown in the following figure, the clients directly talk to the primary Synapse ESB. Synapse can be configured to run as a load balancer. This load balancer will route the requests to the Axis2 worker nodes according to a preconfigured load balancing algorithm. The load balancer is itself deployed in its own cluster. The primary and backup load balancers belong to the same Axis2 clustering domain. Remember, Synapse is built on Axis2, hence uses Axis2's built in clustering mechanism. If the primary load balancer fails, the backup load balancer will take over, and will become the primary load balancer. It will also start another backup load balancer. On the cloud, well-known address-based membership management is used, as multicasting is not possible. The primary load balancer will be the well-known member. Group management has also been enabled in the primary and backup load balancers, so they can detect membership changes in the Axis2 worker node cluster.

When the load exceeds beyond a certain threshold, the primary load balancer will start new Axis2 instances on the cloud. Similarly, when the load drops below a specified threshold, extra Axis2 nodes will be terminated. Ideally, the number of running Axis2 instances will be proportional to the load.



References

- `Apache Synapse, http://synapse.apache.org`
- `WSO2 ESB, http://wso2.com/products/enterprise-service-bus/`
- `OpenESB, http://open-esb.dev.java.net/`

Summary

In this chapter, we looked at SOA integration patterns, which can be realized using Apache Axis2 and Apache Synapse. Apache Synapse is an enterprise service bus built using Axis2 technology. We looked at six popular integration patterns that have been tried and tested in real world production deployments. The patterns we discussed were protocol bridging, external authentication and authorization, dynamic routing combined with auditing, Event Driven Architecture (EDA) with Master Data Management (MDM), push and pull, and fault tolerant autoscaling with dynamic load balancing.

In the next chapter, we will be discussing some advanced features of Apache Axis2, including REST (Representational State Transfer) support in Axis2, MTOM (Message Transfer Optimization Mechanism), understanding the Axis2 classloader hierarchy, and deploying Axis2 on various other application servers.

14
Axis2 Advanced Features and Usage

In the preceding chapters, we discussed how to install Axis2, write a simple web service, write a simple module, invoke a web service, and many other things. If you followed them thoroughly along with the samples, you should now be in a good position in terms of the Axis2 basics. So, in this chapter, we will be thrown in at the deep end, so to speak, with more advanced features of Axis2. But there's no need to panic! You have being using most of the advanced features of Axis2 already without knowing it in the previous chapters. Now it's time to learn them in a proper manner and use them in your applications. Axis2 offers a number of interesting features that you need to know when you want to develop a complex web service application.

In this chapter, we will discuss several advanced features of Axis2, which you will need when you go beyond the basic web service invocation. Particularly, we will discuss the following:

- Representational State Transfer (REST) and its features
- Use of REST in Axis2
- Message Transmission Optimization Mechanism (MTOM)
- MTOM support on the client side
- MTOM support on the server side
- `AxisConfigurators` for creating more customized Axis2
- Invoking and processing web services in an asynchronous manner

In this chapter, we will discuss several different features and their usage. However, it should be noted that each individual section has its own meaning and application. Moreover, you will need these features when you want to do complex or more customized works.

Representational State Transfer (REST)

We start this chapter by introducing **Representational State Transfer** (**REST**) and its features. REST is a term introduced by Roy Fielding in his Ph.D. dissertation to describe an architectural style of networked systems. The motivation behind REST was to capture the characteristics of the Web that made it a success. Subsequently, these characteristics are being used to guide the evolution of the Web. REST is not a standard but an architectural style, which accounts for the fact that you cannot find any specification in W3C.

Features of REST

The Web consists of resources. Simply put, a resource is any item which holds some interest. For example, the Amazon online store may define a resource for their new Canon camera (DSLR), and the client may access the resource page using the following URL: `http://www.amazon.com/canon/dslr`.

Upon clicking the link, a representation of the resource is returned (for example, `canon_dslr.html`). The representation places the client application in a state. The result of the client traversing a hyperlink in `canon_dslr.html` is it accesses another resource. The new representation places the client application in yet another state. Thus, the client application changes (transfers) state with each resource representation it accesses, and hence, we have Representational State Transfer!

Here is Roy Fielding's explanation of the meaning of Representational State Transfer:

> "*Representational State Transfer is intended to evoke an image of how a well-designed web application behaves: a network of web pages (a virtual state-machine) where the user progresses through an application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use.*"

The Web is a REST system! Many of those web services that you have been using these many years—book-ordering services, search services, online dictionary services, and so on—are REST-based web services, which goes to show that you have been using REST and building REST services without even knowing it.

REST services in Axis2

Web Service Description Language (WSDL), especially WSDL 2.0 HTTP Binding defines a way to support REST APIs in web services. Axis2 implements most of what is defined in the HTTP binding specification. The Axis2 REST implementation assumes the following properties:

- REST web services are synchronous and request some response in nature.
- When REST web services are accessed through GET, the service and the operations are identified based on the URL. The parameters are assumed as parameters of the web service (name values pairs in HTTP GET request `name=Axis2&value=1.5`). In this case, the GET-based REST web services support only simple types as arguments and it should adhere to the IRI style.
- POST-based web services do not need a SOAP envelope or a SOAP body. It only sends the XML payload or URL parameters, as in GET.

In Axis2, by default, any given web service is exposed as both SOAP and REST. The result is that as a service author, you do not need to do additional work. However, it should be noted that for some services that are exposed using REST API will not work. For example, if the application takes a complex object, it is hard to expose that using REST GET API. Hence, there are situations where the user is required to change the auto-generated WSDL file to provide the correct information. Axis2 generates REST binding for both WSDL 1.1 and WSDL 2.0, and more importantly, with WSDL 2.0, users get more features to customize the REST binding, as WSDL 2.0 is more focused on REST than WSDL 1.1.

REST web service with GET and POST

A service deployed in Axis2 can be accessed through an HTTP POST or a GET request. However, in the case of a GET, you have to send the request as URL parameters (remember the IRI style). The IRI style has some limitations when encoding or serializing complex objects, thus, in reality, you can only invoke only a set of service using the REST GET method, but you can invoke almost all the services using the REST POST method. For example, if the web service method takes a complex type as its method parameter, then you cannot use the HTTP GET method. Therefore, a service like the following cannot be accessed using HTTP GET:

```
public class Man {
  String name;
  int age;
  Address address;
}
public String getName(Man man) {
  //processing logic
}
```

As shown in the preceding code segment, the `Man` object contains two simple type attributes (`name` and `age`) and one complex type attribute (`address`). Hence the `Man` class can be considered as a complex object.

However, a service like the following can be easily accessed using the HTTP GET method:

```
public String getName(String id, int age){
}
```

To invoke a service like this, you can send the request as follows:

```
http://localhost:8080/axis2/services/ServiceName/getName?id=ID&age=10
```

With the HTTP POST method, there is no limitation, like in HTTP GET. You can send the request as an encoded URL or as a request body (in a POX message). When sending the message as the request body with HTTP POST, you only send the SOAP body.

You can use any given HTTP client to invoke a web service hosted in Axis2 in REST manner, using either the GET or the POST method. However, as we discussed earlier, you will have to make a judgment call, depending on the method signature of the web service operation, on whether to use GET or POST. You can even use `ServiceClient` to invoke a remote service in REST manner (with either GET or POST). The following code demonstrates invoking a service in REST manner using POST:

```
ServiceClient client = new ServiceClient();
Options opts = new Options();
opts.setTo(new EndpointReference("address of the service "));
opts.setAction("soap action ");
opts.setProperty(Constants.Configuration.ENABLE_REST, Boolean.TRUE);
client.setOptions(opts);
OMElement res = client.sendReceive(createPayLoad());
```

When you invoke this code, if you send the request via a TCP monitor, you will see that it only sends the request payload (only the SOAP body) and you will get only the payload for the response as well.

To send the request as a GET request, you need to set the following flag:

```
opts.setProperty(Constants.Configuration.HTTP_METHOD_GET, Boolean.
TRUE);
```

This will send the request in a URL-encoded format (if it can be URL encoded).

Message Transmission Optimization Mechanism (MTOM)

Regardless of the flexibility, interoperability, and global acceptance of XML, there are times when serializing data into XML just does not make sense. Web services users may want to transmit binary attachments of various sorts such as images, drawings, XML documents, and so on. together with a SOAP message. Such data is often originally available in a particular binary format.

There are two traditional approaches to dealing with the sending of binary data in XML.

By value

Sending binary data by value is achieved by embedding opaque data (after some form of encoding has taken place of course) as an element or an attribute content of the XML component of data. The main advantage of this technique is that it gives applications the ability to process and describe data based only on the XML component of the data. XML supports opaque data as content through the use of either base64 or hexadecimal text encoding. Unfortunately, both of these techniques bloat the size of the data. For underlying text encoding of UTF-8, base64 encoding increases the size of the binary data by a factor of nearly 1.33 of the original size, while hexadecimal encoding expands data by a factor of about 2. The above factors will be doubled if UTF-16 text encoding is used. Also of concern is the overhead in processing costs (both real and perceived) for these formats, especially when decoding back into raw binary data.

By reference

Sending binary data by reference is achieved by attaching pure binary data as external unparsed general entities outside the XML document and then embedding reference URIs to those entities as elements or attribute values in the XML. This prevents the unnecessary bloating of data and the waste of processing power. The primary obstacle for using these unparsed entities is their heavy reliance on DTDs, which impedes modularity as well as the use of XML namespaces. There were several specifications introduced in the web services world to deal with this binary attachment problem using the "by reference" technique; SOAP with Attachments (SWA) is one such example. Since SOAP prohibits document type declarations (DTD) in messages, this leads to the problem of not representing the data as part of the message infoset, thereby creating two data models. This scenario is akin to sending attachments with an e-mail message. Even though these attachments are related to the message content, they are not contained inside the message. This causes the

technologies that process and describe the data based on the XML component of the data to malfunction. One such example is WS-Security.

MTOM is another specification that focuses on solving the 'attachments' problem. MTOM tries to leverage the advantages of the above two techniques by attempting to merge them. MTOM is clearly a 'by reference' method. The wire format of an MTOM optimized message is the same as a SOAP with Attachments message, which makes it backward compatible with SwA endpoints. The most notable feature of MTOM is the use of the XOP: Include element, which is defined in the XML Binary Optimized Packaging (XOP) specification to refer to the binary attachments (external unparsed general entities) of the message. With the use of this exclusive element, the attached binary content logically becomes inline (by value) with the SOAP document, even though it is actually attached separately. This merges the two realms by making it possible to work only with one data model. This allows the applications to process and describe the data by only looking at the XML part, making the reliance on DTDs obsolete. On another note, MTOM has standardized the referencing mechanism of SwA.

AXIOM (see *Chapter 3*, *Axis 2 XML Model (AXIOM)*), used in Axis2, is an Object Model that has the ability to hold binary data. It has this ability since `OMText` can hold raw binary content in the form of a `javax.activation.DataHandler` class. `OMText` has been chosen for this purpose for two reasons. One is that XOP (MTOM) is capable of optimizing only base64-encoded Infoset data that is in the canonical lexical form of XML Schema base64Binary datatype. The other reason is to preserve the infoset in both the sender and receiver (to store the binary content in the same kind of object, regardless of whether it is optimized or not). MTOM allows you to selectively encode portions of the message, which facilitates the sending of base64encoded data as well as externally attached raw binary data referenced by the 'XOP' element (optimized content) to be sent in a SOAP message. You can specify whether an `OMText` node that contains raw binary data or base64encoded binary data is qualified to be optimized at the time of construction of that node or later. For optimum efficiency of MTOM, a user is advised to send smaller binary attachments using base64encoding (non-optimized) and larger attachments as optimized content.

```
OMElement imageElement = fac.createOMElement("image", omNs);

// Creating the Data Handler for the file. Any implementation of
// javax.activation.DataSource interface can fit here.
javax.activation.DataHandler dataHandler = new javax.activation.
DataHandler(new FileDataSource("SomeFile"));

//create an OMText node with the above DataHandler and set optimized
to true
OMText textData = fac.createOMText(dataHandler, true);
imageElement.addChild(textData);
```

```
//User can set optimized to false by using the following
//textData.doOptimize(false);
```

Also, a user can create an optimizable binary content node using a base64 encoded string, which contains encoded binary content, given with the MIME type of the actual binary representation.

```
String base64String = "some_base64_encoded_string";

OMText binaryNode = fac.createOMText(base64String,"image/jpg",true);
```

Axis2 uses `javax.activation.DataHandler` to handle the binary data. All the optimized binary content nodes will be serialized as Base64 Strings if MTOM is not enabled. You can also create binary content nodes, which will not be optimized in any case. They will be serialized and sent as Base64 Strings.

```
//create an OMText node with the above DataHandler and set "optimized"
to false
//This data will be sent as Base64 encoded strings regardless of
whether MTOM is enabled or not

javax.activation.DataHandler dataHandler = new javax.activation.
DataHandler(new FileDataSource("SomeFile"));
OMText textData = fac.createOMText(dataHandler, false);
image.addChild(textData);
```

MTOM on the client side

To enable MTOM on the client side, you need to set the `enableMTOM` property in the options object to `True` when sending messages.

```
ServiceClient serviceClient = new ServiceClient ();
Options options = new Options();
options.setTo(targetEPR);
options.setProperty(Constants.Configuration.ENABLE_MTOM, Boolean.
TRUE);
serviceClient.setOptions(options);
```

When this property is set to `True`, any SOAP envelope, regardless of whether it contains optimizable content or not, will be serialized as an MTOM-optimized MIME message.

Axis2 serializes all binary content nodes as Base64 encoded strings, regardless of whether they are qualified to be optimized or not.

The following code demonstrates getting binary data as the web service response:

```
ServiceClient sender = new ServiceClient();
Options options = new Options();
options.setTo(targetEPR);
// enabling MTOM
options.set(Constants.Configuration.ENABLE_MTOM, Boolean.TRUE);
OMElement result = sender.sendReceive(payload);
OMElement ele = result.getFirstElement();
OMText binaryNode = (OMText) ele.getFirstOMChild();

//Retrieving the DataHandler & then do whatever processing required to
the data
DataHandler actualDH;
actualDH = binaryNode.getDataHandler();
```

`DataHandler` is the standard way of accessing and representing binary data in Java. A user can create a `DataHandler` by giving a file location, and then the API lets us read and write to the specified file.

MTOM on the service side

The Axis2 server automatically identifies incoming MTOM optimized messages based on the content-type and de-serializes them accordingly. The user can enable MTOM on the server side for outgoing messages.

To enable MTOM globally for all services, users can set the `enableMTOM` parameter to True in the `axis2.xml`. When it is set, all outgoing messages will be serialized and sent as MTOM optimized MIME messages. If it is not set, all the binary data in the binary content nodes will be serialized as Base64 encoded strings. This configuration can be overridden in `services.xml` on a per service and per operation basis.

```
<parameter name="enableMTOM">true</parameter>
```

Note that you must restart the server after setting this parameter (if you set the parameter in `axis2.xml`).

You can write a service, like the one that follows, to perform binary data handling:

```
public class MTOMService {
  public void uploadFileUsingMTOM(OMElement element) throws Exception
{
    OMText binaryNode = (OMText) (element.getFirstElement()).
getFirstOMChild();
```

```
    DataHandler actualDH;
    actualDH = (DataHandler) binaryNode.getDataHandler();

    ... Do whatever you need with the DataHandler ...
  }
  public void uploadBinary( DataHandler data) throws Exception {
    // write the code to handle the data handler
  }
}
```

As shown, a service author can use either `OMElement` or `DataHandler` to expose a service, which takes binary data as the input parameters (or output parameters). Once a user has the `DataHandler`, he can process it and read or write to a specified location.

Axis2 configurator

So far, you have learned how to start Axis2 and work with Axis2 both on the client side and on the server side. However, we did not discuss how the underlying logic works. When you start Axis2, it creates an AxisConfiguration object from your local machine, which is considered as the repository. In the case of the Axis2 WAR distribution, the repository is `<TOMCAT_HOME>webapps/axis2/WEB-INF` (if you are using Tomcat), so that when you start Axis2 in an application server, Axis2 automatically picks the `WEB-INF` directory as the repository. This approach is known as **file-system-based Axisconfigurators**, where the Axis2 configuration is created using a filesystem.

In the same way, you can create Axis2 using a remote location as well, or by even using a database. In the Axis2 distribution, there is inbuilt support for URL-based as well as file-based Axis2 configuration creations. The following code illustrates how to create an Axis2 system using the filesystem:

```
ConfigurationContext configCtx = ConfigurationContextFactory.
createConfigurationContextFromFileSystem(
"E:\\urlrepo","");
 SimpleHTTPServer simpleServer = new SimpleHTTPServer(configCtx,
8070);
simpleServer.start();
```

The following code demonstrates creating an Axis2 system using a URL repository:

```
ConfigurationContext configCtx =
  ConfigurationContextFactory.createConfigurationContextFromURIs(null,
            new URL("http://urlrepo/")
            );
```

```
 SimpleHTTPServer simpleServer = new SimpleHTTPServer(configCtx,
8070);
simpleServer.start();
```

One thing you need to remember when using a URL repository is that the `services` directory has to have a file named `services.list`, listing all the services archive files. For example, if you have services named `foo.aar` and `bar.aar`, then the `services.list` will look like the following:

```
foo.aar
bar.aar
```

In the same way, the `modules` directory should contain a file named `modules.list` listing all the modules.

Deploying Axis2 in various application servers

As we discussed in *Chapter 1*, *Apache Web Services and Axis2*, Axis2 is available in the form of several distributions. The application server distribution is one of them. You can download the Axis2 application server distribution or the WAR distribution from the Axis2 website or build them from the binary distribution. Once you have the WAR distribution, deploying it is just a matter of copying the WAR file into the `webapps` directory of the application server. In the case of Apache Tomcat, it is the `webapps` folder, whereas in the case of Sun Glassfish, it is the autodeploy folder, and so on. So, depending on the application server, you have to figure out the correct location to drop the WAR file.

There are some application servers that will help in unpacking the WAR distribution to a permanent or temporary location. For example, Apache Tomcat will unpack the WAR file to a permanent location. You can make changes there, and when the application server restarts, all of those changes will be available. However, there are some application servers that do not do this. Also, there are application servers that will not unpack the WAR file. Even in Apache Tomcat, you can configure whether you want to unpack or not. Depending upon the application server configuration, hot deployment and hot update also vary. For example, if the application server is not going to unpack the WAR file, you do not have the hot deployment available at all.

Irrespective of the application server, you can get Axis2 to work from a custom repository by editing the `web.xml` file of the `Axis2.war` distribution. Once this is done, you do not need to worry about whether the application server will unpack the WAR or not, and if it is, where it is going to unpack and so on. You can even configure the application-server-based Axis2 to start up with a remote repository. First, let's look at how to configure `Axis2.war` to work with a local filesystem. Here, you need to add the following `init` parameters to the `servlet` section of the `web.xml` file:

```
<servlet>
  <servlet-name>AxisServlet</servlet-name>
  <display-name>Apache-Axis Servlet</display-name>
  <servlet-class>org.apache.axis2.transport.http.AxisServlet</servlet-
    class>
    <init-param>
      <param-name>axis2.xml.path</param-name>
      <param-value>path to custom axis2.xml (you need this only if you
         want to override the default axis2.xml)</param-value>
      <param-name>axis2.repository.path</param-name>
      <param-value>full path the custom repository</param-value>
    </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

As mentioned earlier, you can configure the `web.xml` file to start Axis2 from a URL repository. In this case, you need to add the following two parameters:

```
<servlet>
    <servlet-name>AxisServlet</servlet-name>
    <display-name>Apache-Axis Servlet</display-name>
    <servlet-class>org.apache.axis2.transport.http.AxisServlet</
servlet-class>
    <init-param>
     <param-name>axis2.xml.url</param-name>
     <param-value>http://localhot/myrepo/axis2.xml</param-value>
    <param-name>axis2.repository.url</param-name>
    <param-value>http://localhot/myrepo</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
```

As shown here, we have added two parameters called `axis2.xml.url` and `axis2.repository.url`. You can modify the values to suit your environment settings.

Asynchronous web services with Axis2

The concept of an asynchronous web service is a very important feature, where a user can invoke one web service and continue his/her work until he/she gets the reply. In a traditional synchronous model, a client has to wait (or block) until he gets the response. More importantly, in Axis2, you get asynchronous support both on the client side and the server side. In the client side, we can invoke the service in a blocking or non-blocking manner. Selecting which way to invoke is very straightforward. In the server side, we can have services that take a long time to process. In such a situation, a service can be processed in a non-blocking manner, where once the message is received, it will send the acknowledgment, and at a later time, send the response. Axis2 supports asynchronous web service invocations on the client side in two different ways; one of them needs WS addressing support and the other does not. We will discuss each of them separately with code sample.

What is a synchronous web service?

As we discussed earlier, a synchronous web service is a blocking service invocation, where the application gets blocked until it receives the response. With this type of invocation, the user will have to wait until he/she gets the response he/she wants. If this is a GUI application, then the application becomes non-responsive until it receives the reply.

What is an asynchronous web service?

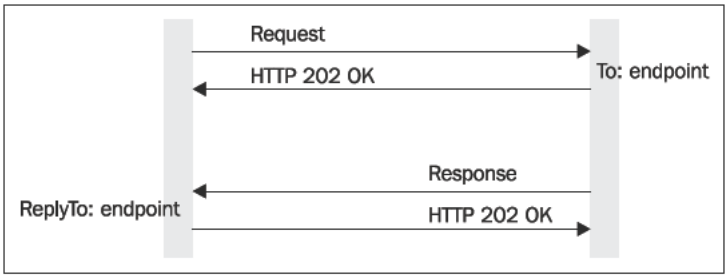
In the previous section, we discussed about synchronous web services. An asynchronous web service is the opposite of that. An asynchronous web service is a non-blocking invocation, where the user can invoke the service without blocking the user application.

Client side asynchronous

One of the main design goals of Axis2 is to have very good support for asynchronous web service support, and asynchronous support is much more useful for the client side. Thus, Axis2 has very good support for asynchronous service invocation on the client side. Axis2 supports two types of asynchronous invocation on the server side:

- Application level asynchronous support: In this case, we only have application level asynchronous support. However, at the transport (transport sender), we have blocking behavior. In other words, when we invoke the service, we provide a callback object and then Axis2 registers that with transport. Once the transport gets the response, it invokes the callback. To have this type of asynchronous support, it does not need to have WS-Addressing support, because we do not have any asynchronous behavior at the transport level.

- Transport level asynchronous support: With this type of asynchronous support, we have non-blocking behavior both at the application level and the transport level. One of the big advantages with this type of invocation is that we can invoke the service using HTTP and ask it to send the response on SMTP. With this type of invocation, we have the callback, we register that with Axis2, and we specify the types of transport we want (sending and receiving). Then Axis2 sends the request with the `wsa:replyTo` address. The server then immediately sends the acknowledgment and sends the reply using the `replyTo` address. The following figure represents this type of invocation:



We have discussed the different types of asynchronous support in Axis2. Now it's time to actually use them. So first, let's write a service and deploy it in Axis2. You can use any of the deployment mechanisms you want.

Sample service:

```
public class EchoService {
     public String echo(String value){
    return value;
  }
}
```

Deploy service (here we deploy the service programmatically):

```
public class EchoServer {
  public static void main(String[] args) {
     new AxisServer().deployService(EchoService.class.getName());
    }
  }
```

Now our service runs on the following address: `http://localhost:6060/`.

If it lists your service, then our service is up and running. The first step to implementing an asynchronous client is to implement a callback interface. A callback interface consists of four methods, and two of them are very important. The first one is what to do when the response is ready and the second one is what happens if something goes wrong.

```
public class MyCallBack implements AxisCallback {
  public void onMessage(MessageContext messageContext) {
    SOAPBody msg = messageContext.getEnvelope().getBody();
    System.out.println(msg);
  }
  public void onFault(MessageContext messageContext) {
    messageContext.getFailureReason().printStackTrace();
  }
  public void onError(Exception e) {
    System.err.println(e.getMessage());
  }
  public void onComplete() {
    System.out.println("Invocation is complete");
    //In real application you do not need to terminate the program
    System.exit(0);
  }
}
```

As we can see here, there are four methods in the interface. Now let's look at each one-by-one:

- `onMessage`: As we discussed previously, WSDL 2.0 defined MEP or message exchange patterns. The idea behind it is that of the message invocation patterns between the client and the server. The client can send two messages and get one response (in-in-out), or the client sends the request and gets the response (in-out), and so on. The idea behind `onMessage` is to invoke user action when a message is received. In our implementation, we have to print the message, but depending on the requirements, we can implement our code inside the method.
- `onFault`: Sometimes we may receive a fault as part of the service invocation, and a user might be interested in that. So this method helps to achieve it.
- `onError`: While invoking the service, if something went wrong at the client side, it will use this method to let the user know about it.
- `onComplete`: When the MEP is complete, the user will be notified using this method. In this example, we terminate the application when Axis2 calls the `onComplete` method. However, in a real application, we can implement our user action there.

Application-level asynchronous support

Now, we have the service up and running as well as an implementation of the callback class. Next, we can use our callback class and invoke the service in an asynchronous manner. In the first example, we are going to invoke the service using application-level asynchronous support. In our example, we use the `ServiceClient` and the following is our client code to invoke the service:

```
public class EchoClient {
  public static void main(String[] args) throws Exception {
    ServiceClient client = new ServiceClient();
    Options opts = new Options();
    opts.setTo(new EndpointReference
              ("http://localhost:6060/axis2/services/EchoService"));
    opts.setAction("urn:echo");
    client.setOptions(opts);
    client.sendReceiveNonBlocking(createPayLoad(), new MyCallBack());
    System.out.println("send the message");
    while (true) {
        Thread.sleep(100);
      }
    }
    public static OMElement createPayLoad() {
      OMFactory fac = OMAbstractFactory.getOMFactory();
      OMNamespace omNs = fac.createOMNamespace(
          "http://sample.org", "ns");
      OMElement method = fac.createOMElement("echo", omNs);
      OMElement value = fac.createOMElement("value", omNs);
      method.addChild(value);
      value.setText("Axis2");
      return method;
    }
  }
```

Note that in the preceding code segment, we have a while [CIT] loop with `Thread.sleep(100)`. The main idea behind that is to prevent the main thread termination.

We first create an instance of `ServiceClient` and then we create an `Option` object. To configure Axis2 in the client side, we need to have an `Option` object. For more information about using `ServiceClient` and `Option` object, please refer to the resources section. Next, we set the end point reference of the service or the service URL and then we set the `SOAPAction`. Next, we invoke the service using an asynchronous manner. As you can see there, we pass two parameters, an object created from the `createPayLoad` method and an instance of the `MyCallBack` class. To invoke a service in an asynchronous manner, we need to use `sendReceiveNonBlocking` in the client API.

When we run the previous code, we will get following output:

```
send the message
<soapenv:Body xmlns:soapenv="http://schemas.xmlsoap.org/
soap/envelope/"><ns:echoResponse xmlns:ns="http://sample.
org"><return>Axis2</return></ns:echoResponse></soapenv:Body>
Invocation is complete
```

Transports-level asynchronous support

Now let's use full asynchronous support in Axis2. For that, we need to have addressing support for both the client and the server side. We can engage the addressing module at the server side using any of the available ways of engaging a module. However, in this example, we will use the following source code to start Axis2 and engage in addressing. The source code for starting and engaging the addressing module is as follows. We do not need to worry about the source code, because in real life, we are not going to use the following code to start Axis2:

```
public class AddressingEnableServer {
  public static void main(String[] args) throws Exception{
    ConfigurationContext configurationContext =
ConfigurationContextFactory.createConfigurationContextFromFileSystem(
null, null);
    SimpleHTTPServer smt = new SimpleHTTPServer(configurationContext,
      8080);
    AxisService service1 =
      AxisService.createService(EchoService.class.getName(),
      configurationContext.getAxisConfiguration());
    configurationContext.getAxisConfiguration().
engageModule("addressing");
      configurationContext.getAxisConfiguration().
addService(service1);
      smt.start();
  }
}
```

```
http://localhost:8080/axis2/services/EchoService
```

As we mentioned before, we also need to engage in addressing at the client side. We can do that as follows:

```
public class EchoClient {
  public static void main(String[] args) throws Exception {
    ServiceClient client = new ServiceClient();
    Options opts = new Options();
    opts.setTo(new EndpointReference
```

```
        ("http://localhost:8080/axis2/services/EchoService"));
      opts.setAction("urn:echo");
      client.engageModule("addressing");
      opts.setUseSeparateListener(true);
      client.setOptions(opts);
      client.sendReceiveNonBlocking(createPayLoad(), new MyCallBack());
      System.out.println("send the message");
      while (true) {
        Thread.sleep(100);
      }
    }
  }
```

The first highlighted line shows how we have engaged the addressing module, and the second highlighted line shows how to enable a separate listener in the client side by setting the `Option` value in the `Option` object.

Summary

In this chapter, we have discussed a number of the advanced features of Axis2, with examples. In particular, REST and MTOM are very useful in the context of web services. You can use REST as a way of optimizing the wire format of the message and you can use MTOM to send binary data in an efficient manner. Then we discussed creating Axis2 using a custom repository, which may be located locally or remotely. Next, we looked into deploying Axis2 in various application servers. Finally, we discussed how to use Axis2 to invoke services in an asynchronous manner.

Now we have covered most of the necessary concepts and features of Axis2. We have discussed how to use Axis2, both as a server and a client.

In the next chapter (the last chapter of the book), we will discuss how to add quality of services to your web service. Particularly, we will discuss security and reliability and how to use them with code examples.

15
Building a Secure Reliable Web Service

As we discussed in *Chapter1*, *Apache Web Services and Axis2*, when Apache introduced its first web service framework—known as Apache SOAP—one of the main ideas was to show the usefulness of the technology. When people started to use it, they wanted to have several new features in it. As a result, Apache introduced Apache Axis1 and finally, Apache Axis2. In the meantime, there were a number of changes in the web service landscape; people demanded better quality of service support, including security and reliability. As a result, standard bodies around web services (for example, W3C and OASIS) introduced a set of web services standards. Most of the existing web services frameworks support three main QoS standards, namely, security, reliability, and addressing. Interestingly, different frameworks have to be implemented in different manner. In Apache Axis2, due to its flexibility, adding a new quality of service support is as simple as adding a module. In contrast, in Apache Axis, the process is to write one or more handlers and specify them in the global configuration file.

In this chapter, we will introduce both security and reliability and discuss how to use them in Axis2. Particularly, we will discuss the following topics:

- Reliable web services and concepts
    - One way invocation
    - Request response invocation
    - Managing sequences
- Secure web services

Reliable web services

Reliability of a web service is a major concern in most enterprise applications. It is widely believed that adding reliability has an impact on the performance. However, some services become useless if they don't have proper support for reliability. As you know, UDP and TCP are two good protocols to understand the importance of the reliable delivery and performance characteristics. UDP is not a realizable protocol; meaning once you send a message using UDP, it is not guaranteed that the recipient is going to receive it (the recipient may not acknowledge it or the UDP message may get lost). In contrast, once you use TCP, the protocol guarantees the message delivery. Providing the reliability for the guarantee of the delivery of message does not come for free. It needs to transmit a number of headers and acknowledgments, which adds additional communication cost.

TCP and UDP are very low level message transmission protocols. However, when it comes to web services, sending SOAP message involves sending more than one TCP packet and more than one SOAP message. Thus, it is required to make sure that the receiver receives all the messages and in the right order. Reliable messaging provides those requirements. In Axis2, Apache Sandesha2 provides reliability by implementing WS-ReliableMessaging specifications.

Apache Sandesha2 is a separate project in Apache and it has its one release cycle. However, it always tries to follow the same release cycle as Axis2 to release the compatible version with the Axis2 release. In this section, we will discuss how to install and use Apache Sandesha2 and build reliable web services.

You can download the latest version of Apache Sandesha2 from the following link:

`http://ws.apache.org/sandesha/sandesha2/download.cgi`

Once you finish downloading, extract it and find `sandesha-version.mar`. Deploy that in to your `AXIS2-HOME/modules` directory (here Axis2 home is the location where you have extracted Axis2 binaries). If you have deployed Axis2 into Apache Tomcat or any other application server, then deploy it to the right location (we discussed more about module deployment in *Chapter 8*, *Writing an Axis2 Module*).

Even though Sandesha2 is a self-contained package, it needs a small modification in the Axis2 global configuration file (`axis2.xml`). Notably, the later version of Axis2 does not require this modification because it provides the required feature on the fly. To provide full functionality of Sandesha2, it needs to add a new phase called **RMPhase**. You can make the changes by editing your `axis2.xml` as follows:

```
<axisconfig name="AxisJava2.0">
<!-- REST OF THE CONFIGURATION-->
  <phaseOrder type="InFlow">
```

```
        <!-- REST OF THE PHASE ORDER-->
      <phase name="RMPhase"/>
    </phaseOrder>
    <phaseOrder type="OutFlow">
      <phase name="RMPhase"/>
      <!-- REST OF THE PHASE ORDER-->
    </phaseOrder>
    <phaseOrder type="InFaultFlow">
      <phase name="PreDispatch"/>
      <!-- REST OF THE PHASE ORDER-->
      <phase name="RMPhase"/>
    </phaseOrder>
    <phaseOrder type="OutFaultFlow">
      <phase name="RMPhase"/>
      <!-- REST OF THE PHASE ORDER-->
    </phaseOrder>
  <!-- REST OF THE CONFIGURATION-->
  </axisconfig>
```

Once you extract Sandesha2 binary distribution, you can find `Sandesha2-policy-<VERSION>.jar`. Copy that in the `AXIS2_HOME/lib` directory. Reliability is a key quality of service and some of its functionality should reflect in the generated WSDL file. Hence, it needs to provide the necessary resources to generate the WSDL and WS-policy. Once you copy the mentioned JAR file, it will provide the necessary requirements.

Now, you have created everything you need to have reliability support to your web services. The last step remaining now is to engage the module globally or to a particular service.

WS-Reliable messaging needs to have support of WS-addressing. Thus, before using Sandesha2, you need to have addressing module in your `AXIS2_HOME/modules`.

Sample service

The first step of our sample is to create a simple service with two methods and deploy the service in Axis2. Our simple service looks like the following snippet of code, which has an `echo` method and a `ping` method:

```
public class RMSampleService {
  public OMElement echoString(String in) {
     System.out.println("Received text:" + in);
     return in;
```

```
    }
     public void ping{String in}{
      System.out.println("Received text:" + in);
    }
}
```

Next, create a `services.xml` file, using the RPC message receiver, as shown in the following code snippet:

```
<service name="RMSampleService">
   <parameter name="ServiceClass">RMSampleService</parameter>
    <description>
        My Sample service for RM.
    </description>
    <module ref="sandesha2" />
    <module ref="addressing" />
    <messageReceivers>
        <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-out"
class="org.apache.axis2.rpc.receivers.RPCMessageReceiver"/>
        <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-only"
class="org.apache.axis2.rpc.receivers.RPCInOlnyMessageReceiver"/>
    </messageReceivers>
</service>
```

Now, create a service archive file and deploy the service. As you can see in the `services.xml` file, we have engaged the Sandesha2 module to this service. Go to the web console and try to access the service WSDL. If you can access it, you have successfully deployed your service and the Sandesha2 module.

You also need to deploy Sandesha2 and the addressing module into your client classpath before we go to the next step. If you have an Axis2 repository for the client, copy the downloaded Sandesha2 module into the `modules` directory. Remember to add the RM phases into the client-side `axis2.xml` as well. In addition, copy the JAR file of Sandesha2 into the client classpath.

One way invocation

The idea behind one way invocation is to send messages reliability to the receiver and the sender does not need an application reply, but needs a reliable delivery of the messages. Here we use the following client to send three messages.

Global configuration:

```
public class PingClient {

  private static final String tns = "http://ws.axis2.apache.org";
  private static String toEPR =
      "http://127.0.0.1:8070/axis2/services/RMSampleService";

  private static String CLIENT_REPO = "location to client repository";

 public static void main(String[] args) throws AxisFault {

 String axis2_xml = CLIENT_REPO + File.separator + "client_axis2.xml";
 ConfigurationContext configContext = ConfigurationContextFactory.crea
teConfigurationContextFromFileSystem(CLIENT_REPO,axis2_xml);
```

Creating and configuring the client:

```
Options clientOptions = new Options ();
clientOptions.setTo(new EndpointReference (toEPR));
ServiceClient serviceClient = new ServiceClient (configContext,null);
clientOptions.setAction("urn:wsrm:Ping");
serviceClient.setOptions(clientOptions);
serviceClient.engageModule(new QName ("sandesha2"));
serviceClient.engageModule(new QName ("addressing"));
serviceClient.fireAndForget(createElement("ping1"));
serviceClient.fireAndForget(createElement("ping2"));
clientOptions.setProperty(SandeshaClientConstants.LAST_MESSAGE,
  "true");
serviceClient.fireAndForget(createElement("ping3"));
serviceClient.cleanup();
}
```

Creating the payload:

```
        private static OMElement createElement(String text) {
                OMFactory fac = OMAbstractFactory.getOMFactory();
                OMNamespace namespace = fac.
createOMNamespace(tns,"ns1");
                OMElement pingElem = fac.createOMElement("ping",
namespace);
                OMElement textElem = fac.createOMElement("in",
namespace);

                textElem.setText(text);
                pingElem.addChild(textElem);
                return pingElem;
        }
}
```

When you use Sandesha2, it first creates a sequence (reliable messaging-specific session) and sends the messages, and when the client marks the last message, it terminates the sequence. In the preceding code sample, the `ping1` and `ping2` messages are in the Sandesha2 session, while `ping3` is not in the session. Sandesha2 provides the reliable delivery to `ping1` and `ping2`, but not to `ping3`.

Request-reply invocation

Here, we provide how to use request-reply message invocation with Sandesha2. Just like the preceding example, `echo1` and `echo2` are in the Sandesha2 session and they get the reliable support. `echo3` does not get the reliable support because prior to sending `echo3`, it sends the last message.

```
public class EchoClient {
  private static final String tns = "http://ws.axis2.apache.org";
  private final static String echoString = "echoString";
  private static String toEPR =
    "http://127.0.0.1:8070/axis2/services/RMSampleService";

  private static String CLIENT_REPO = "Location of Client repository";
  public static void main(String[] args) throws Exception {
    String axis2_xml = CLIENT_REPO + File.separator
                       +"client_axis2.xml";
    ConfigurationContext configContext =
  ConfigurationContextFactory.createConfigurationContextFromFileSystem
  (CLIENT_REPO,axis2_xml);
```

Creating the client:

```
ServiceClient serviceClient = new ServiceClient (configContext,null);
Options clientOptions = new Options ();
clientOptions.setTo(new EndpointReference (toEPR));
clientOptions.setUseSeparateListener(true);
serviceClient.setOptions(clientOptions);

serviceClient.engageModule(new QName ("sandesha2"));
serviceClient.engageModule(new QName ("addressing"));

Callback callback1 = new TestCallback ("Callback1");
```

Invoking the service (first time):

```
serviceClient.sendReceiveNonBlocking (createElement("echo1"),callba
ck1);
Callback callback2 = new TestCallback ("Callback2");
```

Invoking the service (second time):

```
                serviceClient.sendReceiveNonBlocking(createElement("ec
ho2"),callback2);
```

Sending the last message:

```
clientOptions.setProperty(SandeshaClientConstants.LAST_MESSAGE,
"true");
Callback callback3 = new TestCallback ("Callback3");
```

Invoking the service (third time):

```
serviceClient.sendReceiveNonBlocking(createElement("echo3"),callba
ck3);
while (!callback3.isComplete()) {
  Thread.sleep(1000);
}
Thread.sleep(2000);
}
```

Creating the payload :

```
private static OMElement createElement(String text) {
OMFactory fac = OMAbstractFactory.getOMFactory();
OMNamespace applicationNamespace = fac.createOMNamespace(tns,"ns1");
OMElement echoStringElement = fac.createOMElement(echoString,
applicationNamespace);
OMElement textElem = fac.createOMElement("in",applicationNamespace);

textElem.setText(text);
echoStringElement.addChild(textElem);

return echoStringElement;
}
```

Implementing the `Callback` class:

```
static class TestCallback extends Callback {
String name = null;
public TestCallback (String name) {
this.name = name;
}

public void onComplete(AsyncResult result) {
SOAPBody body = result.getResponseEnvelope().getBody();
```

```
                        System.out.println("Callback '" + name +  "'
got result:" + body);
}
public void onError (Exception e) {
System.out.println("Error reported for test call back");
   e.printStackTrace();
}
 }
}
```

If you run the given code and observe it through The TCP monitor, you will see that only the first two messages are in the RM session and the last one is not invoked inside the RM session.

Managing sequences

In the previous section, we discussed Sandesha2 sequences, and we mentioned that it internally creates a sequence and sends the messages until the user specifies the last message. Here, we will discuss how to manage sequences using Sandesha2 API. Sandesha2 provides a utility class called `SandeshaClient` that internally uses instances of `ServiceClient`. Hence, creating an instance of `SandeshaClient` requires an instance of `ServiceClient`.

By default, Sandesha2 assumes that messages going to the same endpoint should go in the same RM sequence. Messages will be sent in different RM sequences only if their WS-Addressing To address is different. However, if required, you can instruct Sandesha2 to send messages that have the same WS-Addressing To address in two or more sequences. To do this, you will have to set a property called `Sequence Key`.

The following line represents how to use the sequence ID to differentiate between sequences:

```
clientOptions.setProperty(SandeshaClientConstants.SEQUENCE_
KEY,<sequnce id>);
```

Creating a sequence without sending a message

In some of the applications, you need to start a sequence without sending any application messages to the server. In that case, you can use the following API. When you ask for this, Sandesha2 will do a Create Sequence message exchange and obtain a new sequence ID from the server. The sequenceKey value of the newly created sequence will be returned from this method, which could be used do message invocations with it. This method also has a boolean parameter, which tells whether to offer a sequence for the response side.

```
String sequenceKey = SandeshaClient.createSequence (serviceClient,
booleanoffer);
```

Terminate a sequence

You can terminate the sequence by using the following code; there Sandesha2 will send the required control messages and remove the sessions from both the client and the server:

```
SandeshaClient.terminateSequence (serviceClient);
```

Sandesha2 provides many other features that you can use to configure and achieve better customization; however, we are not going to discuss them here. I encourage you to refer to online documentation to understand them better.

Secure web services

In the previous section, we discussed how to add the reliability to your service. In this section, we will discuss how to provide the security to your service. In Axis2, we use Apache Rampart to provide the security support.

Download the latest version of Apache Rampart and extract it. Copy the `rampart-version.mar` file into the `module` directory of your `AXIS2_HOME` directory. Next, copy the other's JAR files from the extracted location to the `lib` directory (for example, `TOMCAT_HOME/axis2/WEB-INF/lib`). Now, restart Axis2 and log in to the administrative console and engage the Rampart module. After a few seconds, if you see the Rampart module under the engaged module, then you have successfully added the security support to your web service framework.

Sample service

First, create the service class that we are going to use for our security sample. Here, we are going to use a similar service as in the reliable messaging case:

```
public class SecureService {

   public String echo(String in) {
          return in;
   }
}
```

Writing the password callback

We need to have a password callback object to authenticate the user. We can store the information about username and password in any location we want (for example, database, LDAP, and so on). At runtime, use the password callback class to authenticate the user, as shown:

```
import org.apache.ws.security.WSPasswordCallback;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import java.io.IOException;
public class PWCBHandler implements CallbackHandler {
  public void handle(Callback[] callbacks) throws IOException,
    UnsupportedCallbackException {
      for (int i = 0; i < callbacks.length; i++) {
        WSPasswordCallback pwcb = (WSPasswordCallback)callbacks[i];
          if(pwcb.getIdentifer().equals("axis2uers") &&
            pwcb.getPassword().equals("password")) {
              //If authentication successful, simply return
              return;
          } else {
                throw new UnsupportedCallbackException(callbacks[i],
                "check failed");
      }
    }
  }
}
```

The preceding code snippet has the password callback class we will be using for this sample. As you can see, we have hardcoded the password; but when you write complex callback classes, you can retrieve the logic and set the correct password.

Creating the policy element

Rampart uses policy-based configuration for its server and client side. So to configure the service, we need to provide the required WS-policy configuration. Here we are not going to discuss the WS-policy in detail. We simply provide the required policy configuration for the service, as shown in the following code snippet:

```
<wsp:Policy wsu:Id="UsernameTokenOverHTTPS"
  xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
  wssecurity-utility-1.0.xsd"
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
    <wsp:ExactlyOne>
      <wsp:All>
      <sp:TransportBinding
      xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
        <wsp:Policy>
        <sp:TransportToken>
          <wsp:Policy>
          <sp:HttpsToken
        </wsp:Policy>
        </sp:TransportToken>
        <sp:AlgorithmSuite>
          <wsp:Policy>
          <sp:Basic256/>
          </wsp:Policy>
        </sp:AlgorithmSuite>
        <sp:Layout>
          <wsp:Policy>
          <sp:Lax/>
          </wsp:Policy>
        </sp:Layout>
        <sp:IncludeTimestamp/>
        </wsp:Policy>
      </sp:TransportBinding>
      <sp:SignedSupportingTokens xmlns:sp="http://schemas.xmlsoap.org/
ws/2005/07/securitypolicy">
        <wsp:Policy>
          <sp:UsernameToken sp:IncludeToken="http://schemas.xmlsoap.
org/ws/2005/07/securitypolicy/IncludeToken/AlwaysToRecipient" />
        </wsp:Policy>
      </sp:SignedSupportingTokens>
```

```
      <ramp:RampartConfig xmlns:ramp="http://ws.apache.org/rampart/
policy">
        <ramp:passwordCallbackClass>PWCBHandler</
ramp:passwordCallbackClass>
      </ramp:RampartConfig>
      </wsp:All>
    </wsp:ExactlyOne>
  </wsp:Policy>
```

If you are familiar with WS-security and WS-policy, you can see that it contains two main security assertions—transport binding assertion and signed supporting token assertion. **Transport binding assertion** defines the requirement of using an SSL transport using the HTTPS transport token assertion. **Signed supporting token** defines the requirement of a username token that should be integrity-protected at the transport level.

Now we have everything. Next, we need to create the `services.xml` file for our sample service with the following code:

```
<service>
  <module ref="rampart"/>
  <parameter name="ServiceClass"> SecureService</parameter>
  <messageReceivers>
    <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-out"
    class="org.apache.axis2.rpc.receivers.RPCMessageReceiver"/>
    <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-only"
    class="org.apache.axis2.rpc.receivers.RPCInOlnyMessageReceiver"/>
  </messageReceivers>
  <wsp:Policy wsu:Id="UsernameTokenOverHTTPS"
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
    wss-wssecurity-utility-1.0.xsd"
    xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
  </wsp:Policy>
</service>
```

Now compile the `service` class and the callback object and create the service archive file by using the `services.xml` file mentioned. You can validate the service deployment by logging in to the web administrative console. If you follow the preceding procedure correctly, you should be in good shape.

Generating client stubs

In this case, we are going to use our WSDL2Java code generator and generate client stubs to invoke the service. As our service has WS-Security, writing a service using `ServiceClient` is not a trivial task; WSDL2Code tool has better support for WS-Policy and generates the right stub to invoke the service. We use the following command to generate a client stub (make sure the service is up and running):

```
[Linux]
$ sh WSDL2Java.sh -uri http://localhost:8080/axis2/services/
SecureService?wsdl -p client -uw  -o /out-path/

[Windows]
WSDL2Java.bat -uri http://localhost:8080/axis2/services/
SecureService?wsdl -p client -uw -o /out-path/
```

Invoking the service without security

First, you need to copy the generated source code into your project (if you have not done so already). Next, use the following code to invoke the service:

```
public class SecureServiceCGClient {
  public static void main(String[] args) throws Exception {
    SecureServiceStub stub = new SecureServiceStub(null,"https://
localhost:8443/axis2/services/SecureService");
    String result = stub.echo("MyMesage");
    System.out.println(result);
  }
}
```

Here we invoked the service without sending any security information. You can understand it by sending the message through the TCP monitor.

Next, we are going to invoke the service with security support. Here you need to make sure that you have deployed and engaged the Rampart modules into the client repository as well (need to put into the repository).

Invoking the service with security

We can use the following code to invoke the service with security support. We provide the username and password.

```
ConfigurationContext ctx = ConfigurationContextFactory.createConfigura
tionContextFromFileSystem("path/to/client/repo", null);


   SecureServiceStub stub = new SecureServiceStub(ctx,"https://
localhost:8443/axis2/services/SecureService");
```

```
ServiceClient sc = stub._getServiceClient();
sc.engageModule("rampart");
Options options = sc.getOptions();
options.setUserName("axis2user");
options.setPassword("password");
     String result = stub.echo("MyMesage")
     System.out.println(result);
```

As we are going to use Rampart, we need to create the service client with the repository which contains the Rampart module; we provide the username and password and invoke the service. You can see the difference in the messages by sending them through the TCP monitor. In addition, as an exercise, send the message with a wrong username or password and observe the difference.

WS-security, in particular, Apache Rampart, provides a number of features and we cannot discuss everything here. We will just provide a simple introduction to Apache Rampart. If you need to learn more, go to the official website (`http://axis.apache.org/axis2/java/rampart/`) and search for online articles and tutorials (`http://axis.apache.org/axis2/java/rampart/articles.html`).

Summary

In this chapter, we discussed the two most important and useful 'quality of services' for a web service, namely, reliable messaging and security. We introduced the two specifications briefly and then provided a complete sample as to how to use them on the server side and in the client side. We did not go into more detail, as it is out of the scope of this chapter. If you want to learn more about them, you can go to the official websites and search for online articles and tutorials.

In this book, we have tried to get a basic understanding of the Axis2 web service framework. There we used sample code to explain the concept well. We discussed a number of new features of Axis2 and how to use them. Additionally, we discussed various advanced features of Axis2, and finally, two important QoS specifications for web services.

We hope you enjoyed the book.

Good Luck.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

T

U

V

W

X