From Technologies to Solutions

Quickstart

Apache Axis2

A practical guide to creating quality web services

Deepal Jayasinghe

PACKT PUBLISHING

Quickstart Apache Axis2

A practical guide to creating quality web services

Deepal Jayasinghe

PACKT PUBLISHING

BIRMINGHAM - MUMBAI

Quickstart Apache Axis2

First published: May 2008

Production Reference: 1160508

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

ISBN 978-1-847192-86-8

www.packtpub.com

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

Credits

Author

Deepal Jayasinghe

Reviewers

Ramanujam A. Rao

Senior Acquisition Editor

Rashmi Phadnis

Technical Editor

Shilpa Dube

Editorial Team Leader

Mithil Kulkarni

Project Manager

Abhijeet Deobhakta

Project Coordinator

Patricia Weir

Indexer

Monica Ajmera

Proofreader

Chris Smith

Production Coordinator

Aparna Bhagat

Cover Work

Aparna Bhagat

About the Author

Deepal Jayasinghe is a Technical Lead at WSO2 Inc., an open-source software development company that creates middleware platforms for Web Services. He joined WSO2, Inc. in August, 2005. He has more than 3 years of experience with SOA and Web Services in addition to being a contributing member of the Apache Axis2 project since its inception. He is a key architect and a developer of the Apache Axis2 Web Service project and has led a number of releases. In addition to Axis2, he has made major contributions to Apache Synapse, Apache Axiom, and Apache XMLSchema projects. Deepal has written more than 30 technical magazine articles, research papers, and has delivered speeches in various SOA and Web Services conferences. He is an Apache Web Services PMC member, an Apache committer, and an Apache Member. His expertise lies mainly in distributed computing, fault-tolerant systems, and Web Services-related technologies. He has a B.Sc. in Engineering from the University of Moratuwa, Sri Lanka and and will be starting graduate studies at Georgia Institute of Technology in fall, 2008.

Contact: `deepalk@gmail.com`.

First of all, I want to thank the Apache Axis2 developers and Axis2 community who have contributed towards making this Web Services framework a world-renowned success in a relatively short period of time. Thank you!

I would like to thank Dr Sanjiva Weerawarana, Founder, Chairman and CEO of WSO2, Inc., without whose support, vision, guidance, and belief in my work this effort would never have been realized. I owe countless thanks to my parents and dear wife for always being there and supporting me in so many ways. This book would not have been possible without everything that they have done for me. Special thanks to my colleagues Suran Jayathilaka, Devaka Randeniya, and Charitha Kankanamage for reviewing my writings, validating the samples, providing insight, and contributing to this effort in many other ways. Special thanks to Srinath Hemapani, Ajith Ranabahu, Eran Chinthaka, Davanam Sirinivas, Glen Daniels, Paul Fremantle, Chathura Herath, Jaliya Ekanayake, and all other key members of the Axis2 team without whose tremendous contributions and wisdom Axis2 would not have been possible. For the creation of this work, I am blessed with a strong team of technical reviewers, superior editorial and production professionals from Packt Publishing. My sincere thanks to all of you for your tireless efforts.

About the Reviewer

Ramanujam A. Rao is a software architect and an engineer with over 12 years of experience in designing and developing large-scale enterprise applications. He is currently involved in consulting in the area of enterprise-application architecture and helps to build scalable and distributed applications. He also has expertise in building enterprise-technology standards and cultivating architecture capabilities.

He lives in Columbus, OH, USA with his wife and two-year old daughter.

Contact: `arrao@acm.org`.

Table of Contents

Preface

A new architecture for Axis2 was introduced during the first Axis2 summit, which was held in 2004 in Colombo, Sri Lanka. This new architecture is more flexible, efficient, and configurable in comparison to Axis1.x architecture. Some well established concepts from Axis 1.x, like handlers, have been preserved in the new architecture.

Since its introduction less than four years ago, Apache Axis2 has become widely accepted as a framework for practical Web Service development, which is suitable not only for commercial application development, but also as a teaching tool and as a research platform. Apache Axis2 itself has evolved during the past three years, taking into consideration community requirements, and has acquired a number of new features. All of these have been contributed in an effort to create a simple and easy-to-use Web Service framework.

The main goal of this book is to provide an introduction to Axis2. It teaches how to develop web applications using Axis2 and how to make proper use of available features. It does not attempt to cover either Web Services or Axis2 in minute detail, opting rather to provide a good understanding for using both. The in-depth technical details of Axis2, I believe, are best covered in a book in their own right.

When designing and writing this book, my objective was to achieve a number of goals. Firstly, I wanted to present a very clear introductory text, free of verbosity and nonsense, so that a beginner can understand the concepts easily and develop confidence for using the technology within a short period of time. Secondly, I have, as far as possible, tried to cover the concepts in the form of a discussion combined with the instruction style of a tutorial, so that the reader can try out the concepts for himself/herself and grasp them easily. Because of this most of the chapters contain a plethora of comprehensive samples. Thirdly, I have intentionally avoided presenting full descriptions of Axis2 features, while making sure that no important points have been omitted. Descriptions of some of the minor and rarely used features have been left out for the sake of simplicity. And finally, I want this book to help you, the reader, explore, understand, and realize the potential of Web Services and Axis2.

What This Book Covers

This book is organized in a such a way that it will lead you to gain a very good understanding of Web Services and Axis2. At the end of the book, you will have become familiar with most of the commonly used Axis2 features and concepts. You will be able to write a Web Service, invoke a remote Service, and extend the core functionality of Axis2.

Chapter 1 defines Web Services, their architecture, and components. It also discusses the Apache Web Services stack and the motivation for Axis2. Finally, it tells you how to go about downloading and deploying Axis2.

Chapter 2 gives an overview of the Axis2 architecture, its dominant features and extensible nature. Furthermore, it opens the way to learning key terminologies used in Axis2 and getting familiar with them.

Chapter 3 introduces AXIOM, the Axis2 object model and discusses the key features of AXIOM with code samples.

Chapter 4 discusses the smallest execution unit in Axis2—a handler, and then discusses phase and phase rules. Finally, it describes the execution chain and how to change it using phase rules.

Chapter 5 describes how deployment works and available deployment mechanisms in Axis2.

Chapter 6 discusses the dynamic and static data hierarchies in Axis2; how they are stored, how they get created and related, and so on.

Chapter 7 discusses everything you need to know about how to write and deploy a Web Service in Axis2. This includes both POJO and archive-based service development.

Chapter 8 discusses everything about how to write and deploy a service extension or a module in Axis2.

Chapter 9 discusses the Axis2 client API, synchronous and asynchronous Web Service invocations, and different configuration options available for the client side.

Chapter 10—If you are looking to implement session-aware services then this chapter will help you out as it describes the types of available sessions in Axis2 and their proper usage.

Chapter 11 describes how Axis2 handles POJOs, Axis2 data-binding, and code generation.

Chapter 12 discusses other features of Axis2 such as REST support and MTOM as well as the advanced configuration mechanism of Axis2. Finally, it discusses deploying Axis2 in various application severs.

What You Need for This Book

This book is completely based on version 1.3 of Axis2, which is one of the most stable and widely used versions of Axis2. Axis2 1.3 is compatible with JDK 1.4 and above.

Who This Book is For

This book explains and demonstrates the core features of Axis2 and its architecture. Even though Axis2 makes the development of Web Service applications much easier and simpler, it is still a fairly complex piece of middleware that requires a considerable amount of time and effort to master. This book provides straightforward explanations and samples of the underlying technologies and features of Axis2.

As the name of the book implies, "Quickstart Apache Axis2" can be considered an introductory-level book for getting started with Axis2, learning and applying Web Services concepts in practice easily. In order to gain the most from this book, you should have good understanding of the Java programming language. Experience with SOA (Service-Oriented Architecture) and Web Services is not a must to understand the concepts and examples discussed here. I would recommend this book for users who want to start using Axis2 as well those looking to master it.

Conventions

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

There are three styles for code. Code words in text are shown as follows: "A module will not be functional unless it has a `module.xml` file."

A block of code will be set as follows:

```
public OMNamespace declareNamespace(String uri, String prefix);

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

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

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

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

Important notes appear in a box like this.

Tips and tricks appear like this.

Reader Feedback

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

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

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

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

Customer Support

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

Downloading the Example Code for the Book

Visit `http://www.packtpub.com/files/code/2868_Code.zip` to directly download the example code.

The downloadable files contain instructions on how to use them.

Errata

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

Questions

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

1
Introduction

Axis2, the next generation of the Apache Web Service stack, has taken one more step closer to the first production version by releasing another developer version. In this chapter, we will learn more about Web Services, their history, and the 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.

Web Service History

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 they have never done before. The world involves hardware that can support the systems that connect thousands of computers, and those systems have the capacity to wield power that was once only dreamed of.

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

These proposals fail to consider the critical nature of the problem. The systems are a composition of numerous heterogeneous sub-systems. The above proposals require all the participants to share a particular programming language, or a few of those languages. Web Services provide an answer by defining a common XML-based wire representation for the interactions, thus enabling the different systems to interact.

Web Services define SOAP, the message format. They also define WSDL, which is a common way to describe Web Services. 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 Services Overview

The Internet is revolutionizing business by providing an affordable and efficient way to link companies with their partners as well as to their customers. However, there are certain issues that reduce the effectiveness of the Internet. Among those issues, incompatible applications and frameworks that cannot interoperate or that cannot exchange business data have become a major concern. The Web Service is a new e-business model that is expected to change in a way in which the business applications are developed, integrated, and interoperated. Web Services are self-describing and self-contained. A Web Service is a modular application that is accessible over the web. It is exposed as an XML interface, and it communicates with other services by using XML messages over standard Web protocols.

W3C, one of the standard bodies of Web Services 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 machine-processable and human readable format called WSDL (Web Services Description Language). Other applications communicate with the Web Service in a manner prescribed by its description using SOAP (Simple Object Access Protocol) messages, typically conveyed using HTTP with an XML serialization in conjunction with other web-related standards.

The Web Service, a very well known open technology standard provides a number of benefits as listed below:

1. Increase competition among vendors, resulting in lower product costs.
2. Ease transition from one product to another, resulting in lower training costs.
3. Increase the ability for parties to interoperate, resulting in lower maintenance costs.
4. Ensure a greater degree of adoption and longevity for a standard. A large degree of usage from the vendors and the users leads to a higher degree of acceptance.

One can argue that the Web Service concept is a logical evolution from an object-oriented system to a system of services. As in an object-oriented system, 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. The information as to what the service does, where is it located, how it is invoked, the quality of service, and the security policy related to the service can be published in the service interface (WSDL).

How Do Organizations Move into Web Services?

There are three main ways in which an organization can move into Web Services. These are as follows:

1. Creating a new Web Service from scratch. The developer creates the functionalities of the services as well as preparing a document to describe those services.
2. Exposing an existing functionality through a Web Service. Here, the functionalities of the service already exist. Only the service description needs to be implemented.
3. Integrating Web Services from other vendors or business partners. There are instances where using a service implemented by another is more feasible than building from scratch. On these occasions, the organization will be required to integrate others' or even business partners' Web Services.

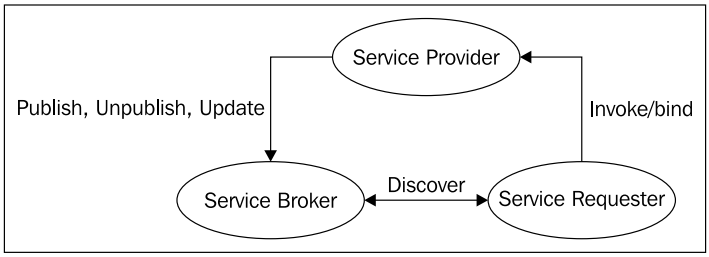
The real utility of the Web Service concept is present in the second and the third methods, which leads to other Web Services and applications that can be used in existing applications.

Web Services describe a new model for using the Web. This model allows publication of business functions to the Web and provides 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. In addition to that, Web Services serve the users' needs by enabling them to choose, configure, and assemble their own Web Services through an intuitive, browser-based interface.

The fundamental concept behind Web Services is the SOA (Service-Oriented Architecture), where an application is no longer a large monolithic program, but is divided into smaller, loosely coupled programs, and provided services are loosely coupled together with standardized and well-defined interfaces. These loosely coupled programs make the architecture very extensible, as it acquires the ability to add or remove services with limited costs. Therefore, new services can be created by combining existing services.

Web Services Model

The Web Services Model consists of basic functionalities such as describe, publish, discover, bind, invoke, update, and unpublish. In the meantime, the model consists of three actors: service provider, service broker, and service requester. The functionalities as well as actors are shown in the figure below:



Service Provider

The Service Provider is an individual (organization) that provides services. The Service Provider's job is to create, publish, maintain, and unpublish their services. From a business point of view, the Service Provider is the owner of the service, whereas from an architectural view, it is a platform that holds the implementation of the service.

Service Broker

The 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 needed services, and obtain the binding information for these services. A service broker can either be public, where the services are universally accessible, or private, where only specified sets of Service Requesters are able to access the service.

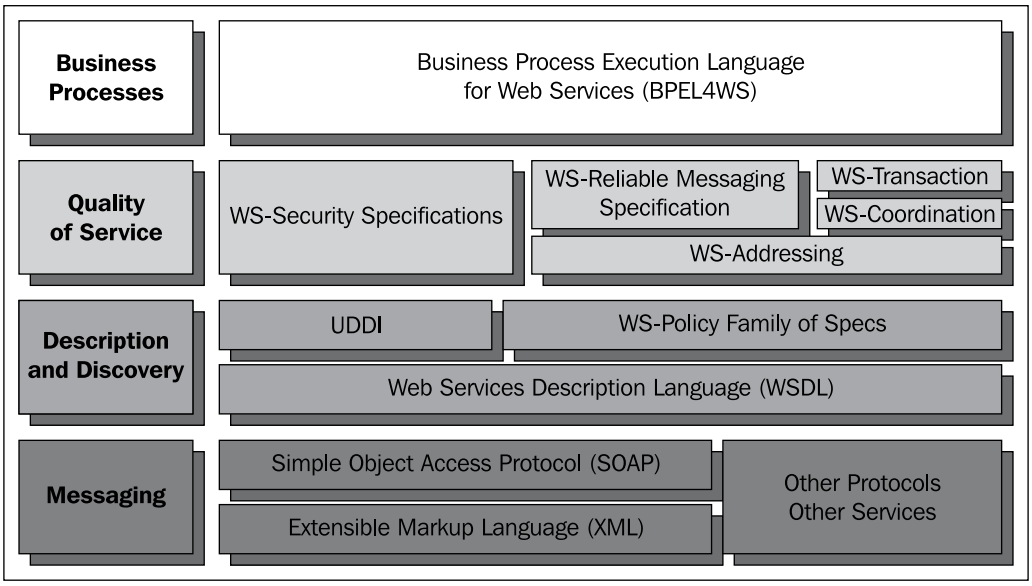
Service Requester

The Service Requester is a party that looks for a service to fulfill its requirements. A requester can either be a human accessing the service, or an application program (The program could also be another service). From a business view, it is a business that wants to consume a particular service. From an architectural view, it is an application that looks for and invokes a service.

Web Services Standards

Web Services are one of the key technologies in today's software industry. As a result, we are in a rapid development process and the stack of interrelated standards that characterize the Web Services infrastructure is maturing. The growing collection

of WS-* standards, supervised by the Web Services governing bodies defines the Web Service protocol stack, as shown in the figure below. Here, we will be looking at the standards that are specified in the most basic layers: messaging and description and discovery.



The messaging standards are intended to give a framework in order to exchange information in a distributed environment. These standards have to be reliable so that the messages are sent only once, and only the intended receiver receives them. This is one of the primary areas wherein a lot of research work is being done, because everything depends on the messaging ability.

XML-RPC

The XML-RPC standard was created by Dave Winer in 1998 with Microsoft. The available RPC systems seemed very bulky. Therefore, in order to create a light-weight system, the developer simplified them by specifying only the essentials, and defining 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, where the body of the request is in XML. A procedure is executed on the server and the returned value is formatted into XML. The parameters can be scalars, numbers, strings, or dates, as well as complex record and list structures.

As new functionalities were introduced, XML-RPC evolved into SOAP. Still, some people prefer using XML-RPC because of its simplicity, minimalism, and ease of use.

SOAP

The SOAP standard was originally designed by four developers with the backing of Microsoft as an object-access protocol. The protocol specifies the exchange of XML-based messages over computer networks in a 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 transition to a SOAP-based implementation.

The concept of SOAP is a stateless, one-way message exchange paradigm. However applications can create more complex interaction patterns, such as request-response, request-multiple responses, and so on. This is done by combining such one-way exchanges with features provided by an underlying protocol and application-specific information. In addition, 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 had been 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 so they were able to simplify Web Services communication and development, and thus avoid the need to develop costly, ad hoc solutions that are often difficult to interoperate across platforms. WS-Addressing addresses this and enables organizations to build reliable and interoperable Web Service applications by defining a standard mechanism for identifying and exchanging Web Service messages between multiple end points.

Web Services Addressing provides transport-independent mechanisms to address messages and identify Web Services, corresponding to the concepts of *address* and *message correlation* described in the Web Services architecture. Web Services Addressing defines XML elements to identify Web Services endpoints, and to secure end-to-end identification of endpoints 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

It is important to note that the description of a Web Service is essential for classifying, discovering, and using the service. The description should be understandable for humans as well as applications. A Web Service description is required at the semantic level as well as at the syntactic level. The semantic information has to contain details about the service provider as to what the service does and its characteristics such as reliability, security, and sequencing of messages. The semantic information enables the 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 be concerned about the non-functional requirements such as security, as well as authentication.

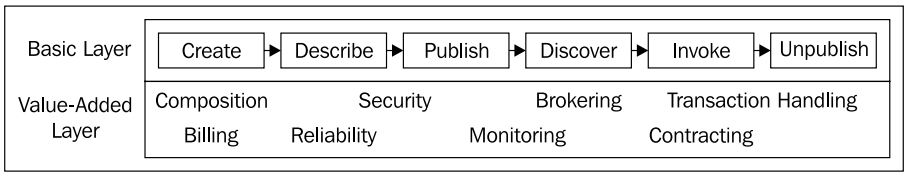
Web Services Description Language (WSDL)

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 in order to provide Web Services over networks. A service requester that connects to a Web Service can read the WSDL to determine the functions that are available in the Web Service. Special data types are embedded in the WSDL file in the form of an XML Schema. The client can then use SOAP to call functions listed in the WSDL.

WSDL enables one to separate the description of the abstract functionality offered by a service from concrete details of a service description such as how and where that functionality is offered. WSDL 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 the interface. A port is defined by associating a network address with a reusable binding. A collection of ports defines a service. Messages are abstract descriptions of the data being exchanged and port types are abstract collections of supported operations. The concrete protocol and the data format specifications for a particular port type together constitute a reusable binding, where the messages and operations are then bound to a concrete network protocol and message format to define an endpoint.

Web Services Life Cycle

As shown in the figure below, Web Services consist of a number of activities. These activities can be divided into two layers: a basic layer and a value-added layer. The basic layer consists of the main activities that have to be supported by any Web Service. The value-added layer adds value, and thus enhances the performance of the Web Service.



The first activity is the creation of the Web Service. This can be achieved either by building from scratch, or by integrating existing Web Services. After creating the Web Service, it has to be described so that others can access it. Then, it has to be published on the Web. The discovery of a Web Service can be facilitated by a service broker who will support requirement analysis and description of a requester's need, and then matching the users' needs to available Web Services, negotiation, and binding. After the Web Service has been discovered, and it has been decided to use the Web Service, a number of activities related to contracting takes place. During the life time of a Web Service, it will be updated and maintained throughout by the service provider. If the Web Service description is changed, then it will be updated at the service broker's end. Finally, the service can remain unpublished, if it is no longer available or needed.

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

Apache Web Service Stack

The history of Web Services has gone through several iterations during its evolution. The first generation of Web Services were highly controlled interactions, and can be considered mere tests of feasibility. Apache SOAP was one of the notable SOAP engines in the first generation. It was meant mainly to be a "proof of concept", and not at all concerned about performance. The whole idea of the first generation SOAP engines was to convince the world that Web Services are a feasible option.

Soon, the interest in these first generation SOAP engines paid off. More companies started showing interest and the SOA started building up. This stage can be called the second generation of Web Services, and it required better and faster SOAP engines. Aspects such as discovery and definition are already standardized, and SOAP engines are also required to support these standards. Apache Axis was born as one of the second generation SOAP engines.

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 into the Web Service arena. Aspects governing different facets of Web Service interactions have been standardized. The third generation of Web Services requires faster and more robust SOAP engines, as the existing Axis is not good enough. Axis2 was made to fill this gap.

Why Axis2?

As we have discussed earlier, Web Services are growing rapidly and a large number of organizations have moved to the Web Services field. As a result , new requirements have been encountered, and new standards are being defined. At the same time, the organizations are not just looking for Web Services. They also pay attention to the reliability, security, and performance of Web Services. In addition to these requirements, new WS* specifications have been defined, and Web Service engine need to support them.

While considering the Apache Web Service stack, Axis1 is one of the stable Web Service engines. A number of organizations as well as a number of applications use Axis1. So, changing Axis1 architecture to support the new requirements as well as new Web Service standards was not a good idea. In the meantime, software has its own life cycle. It can evolve up to a certain point and after that, a revolution is needed. The same theory was applicable to Axis1 as well. Rather than change the Axis1 architecture, the Apache Web Service development team came up with a new architecture.

In addition to the new requirements and the WS* specifications, performance was another area of major concern. Changing the Axis1 architecture in order to improve its performance was not that easy. Axis1 uses DOM as its XML representation mechanism. As a result, complete messages need to be loaded into the memory before the processing starts. The system slows down and the memory usage also increases. Therefore, one of the key requirements behind the introduction of Axis2 was to improve system performance.

The Apache Web Service community discussed and agreed to introduce a new Web Service engine called "Axis2" with a number of new requirements. It requires a very flexible and an easily extensible architecture that supports WS* standard as well as future standards. As a result, Axis2 or The Apache third-generation Web Service engine came into picture.

Download and Install Axis2

Apache Axis2 had a number of releases. Among them, release 1.3 can be considered one of the most robust releases. This book is also based on that particular release. As a result of Axis2 using an incremental development process, there might be compatible issues from one release to another, since there are instances where Axis2 changes its API in order to increase its flexibility as well as usability.

One of the points in an open-source project is that its release consists of source code, which is used to create the binary files by compilation and linking. Each Axis2 release also consists of a source code distribution in addition to its binary distribution.

We can download the latest Axis2 release from `http://ws.apache.org/axis2/download.cgi`. Each Axis2 release consists of four main release artifacts or distributions:

- Binary distribution
- WAR distribution
- Source distribution
- JAR distribution

Binary Distribution

An Axis2 binary distribution consists of all the relevant third-party libraries, a set of samples, and the Axis2 runtime. Installing a binary distribution involves extracting ZIP archive files into a desired location. Once we download and extract the binary distribution, then we will be able to see a set of subdirectories inside it (`bin`, `lib`, `samples`, `repository`, `webapp`).

We can use the Axis2 binary distribution to develop and deploy our enterprise-level applications. The distribution consists of all the resources that are required to develop Web Service application with Axis2. Once we develop our Web Service application, we can deploy that in the same distribution. We can use the binary distribution to start a standalone server, or to create a WAR file to deploy in the application server.

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

WAR Distribution

The Axis2 WAR distribution is useful for deploying Axis2 in application servers such as Tomcat, Jboss, Weblogic, and so on. We can deploy the Axis2 WAR file into an application server, and check whether it works by typing the server address in a browser. As an example, if you deploy the Axis2 WAR file in Apache Tomcat, by typing `http://localhost:8080/axis2`, we can figure out whether Axis2 is up and running. However, the Axis2 WAR distribution does not have any Web Services other than the version service. So, by the deploying default WAR file in an enterprise-level application, we will not gain anything.

To add a new Web Service in Axis2, we need to add the corresponding resources into a WAR file. But most of the application servers do not unpack the WAR file. Therefore, when we use a WAR file in a real application, we have to unpack the WAR file, put our resource into it and then deploy it. However, if the application server unpacks the WAR file, then we can drop our new Web Service into the unpack location. We will be learning about Axis2 Web Services in detail, later in this book.

The following are the steps for installing the WAR distribution:

Step 1: Install the application server. If we do not have any application server in our machine, then we need to download and install an application server. Among the available application servers, Apache Tomcat can be considered one of the best application servers. We can download Tomcat (4.x or above) and install it.

Step 2: Depending on an application server, we can find the location where we need to deploy WAR files. If we take Tomcat as an example, then we need to put the WAR file into the `webapps` directory. So let's drop the Axis2 WAR distribution into the `webapps` directory of the application server.

Step 3: As a final step, open the browser and type `localhost:8080/axis2`. Thus, we will be able to view the Axis2 web application homepage (here, the URL might be different depending on the application server).

Source Distribution

As the name implies, the source distribution consists of the source code that is used to build the binary distribution. Since Apache Axis2 is released under the Apache license, we are free to use the source code. We can also use the Axis2 source distribution in order to hack Axis2. In addition to that, we can help fix issues in the project and can contribute to the open-source community.

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

JAR Distribution

When we want to develop a Web Service application, then we need to have an Axis2 library. The Axis2 JAR Distribution is the Axis2 library. To develop a Web Service application on Axis2, we need to have Axis2 libraries, meaning Axis2 JAR files. In the meantime, there are a number of projects that depends on Axis2, and they need to have the Axis2 library distribution.

Summary

Is the Web Services concept a new revolutionary technology? The idea of splitting large programs into smaller modules is an established principle of higher-level programming languages. Even in assembly programming, procedures were separated from other program parts for better re-usability. Languages such as CORBA can be used to connect programs over a network with a language-independent remote procedure call protocol that already exists with CORBA. CORBA also has an Interface Definition Language (IDL) similar to WSDL. As a fact, the popularity of Web Services is achieved by the standardization of techniques, and by standards that are adaptable to different situations. There a number of solutions exist to realize these concepts of Web Services.

2
Looking into Axis2

Flexibility and extensibility are two main design criteria that software designers would like to have in their applications. When it comes to Axis2, its architecture is extremely flexible and extensible. Axis2 has a modular architecture. In this chapter, we will learn more about Axis2 architecture, its core components, and its main features.

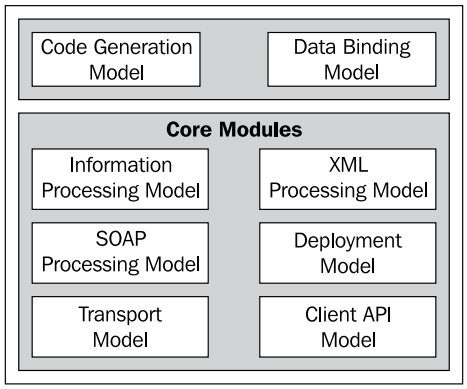
Axis2 Architecture

Axis2 is built upon a modular architecture that consists of core modules and non-core modules. The core engine is said to be a pure SOAP processing engine (there is not any 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 or JMX). But at the transport level, it will be converted into a SOAP message.

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

- Separation of logic and state to provide a stateless processing mechanism. (This is because Web Services are stateless.)
- A 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 figure below shows all the key components in Axis2 architecture (including core components as well as non-core components).



Core Modules

- **XML Processing Model**: Managing or processing the SOAP message is the most difficult part of the execution of a message. The efficiency of message processing is the single most important factor that decides the performance of the entire system. Axis1 uses DOM as its message representation mechanism. However, Axis2 introduced a fresh XML InfoSet-based representation for SOAP messages. It is known as AXIOM (AXIs Object Model). AXIOM encapsulates the complexities of efficient XML processing within the implementation.
- **SOAP Processing Model**: This model 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 in specific places.
- **Information Model**: This keeps both static and dynamic states and has the logic to process them. The information model consists of two hierarchies to keep static and run-time information separate. Service life cycle and service session management are two objectives in 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 Web Services using Axis2. The API consists of two sub-APIs, for average and advanced users. Axis2 default implementation supports all the eight MEPs (Message Exchange Patterns) 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, JMX, TCP and so on). However, the user can write or plug-in custom transports, if needed.

XML Processing Model

As mentioned in Chapter 1, 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 object hierarchy (corresponding to incoming message) to be kept in memory. This will not be a problem for a message of small size. But when it comes to a message of large size, it becomes an issue. To overcome this problem, Axis2 has introduced a new XML representation.

AXIOM (AXIs Object Model) forms the basis of the XML representation for every SOAP-based message in Axis2. The advantage of AXIOM over other XML InfoSet representations is that it is based on the PULL parser technique, whereas most others are based on the PUSH parser technique. The main advantage of PULL over PUSH is that in the PULL technique, the invoker has full control over the parser and it can request the next event and act upon that, whereas in 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 its processing of the document.

Since AXIOM is based on the PULL parser technique, it has 'on-demand-building' capability whereby it will build an object model only if it is asked to do so. If required, one can directly access the underlying PULL parser from AXIOM, and use that rather than build an OM (Object Model).

SOAP Processing Model

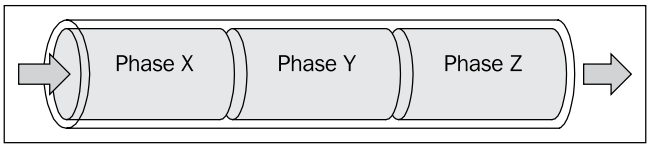
Sending and receiving SOAP messages can be considered two of the key jobs of the SOAP-processing engine. The architecture in Axis2 provides two Pipes ('Flows'), in order to perform two basic actions. The AxisEngine or driver of Axis2 defines two methods, `send()` and `receive()` to implement these two Pipes. The two pipes are named **In***Flow* and **Out***Flow*. 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, and those two help in handling incoming Fault messages and sending a Fault message.

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.

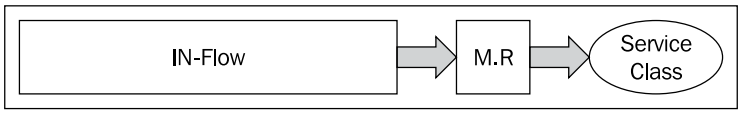
The handlers act as interceptors, and they process parts of the SOAP message and provide the quality of service features (a good example of quality of service is security or reliability). Usually handlers work on the SOAP headers; but they may access or change the SOAP body as well.

The concept of a flow is very simple and it constitutes a series of phases wherein a phase refers to a collection of handlers. Depending on the MEP for a given method invocation, the number of flows associated with it may vary. In the case of an in-only MEP, the corresponding method invocation has only one pipe, that is, the message will only go through the in pipe (inflow). On the other hand, in the case of in-out MEP, the message will go through two pipes, that is the in pipe (inflow) and the out pipe (outflow). When a SOAP message is being sent, an OutFlow begins. 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 need 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 figure below illustrates a flow and its phases.



If the message has gone through the execution chain without having any problem, then 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 figure below 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 the stages of the pipes are named 'phases' in Axis2.

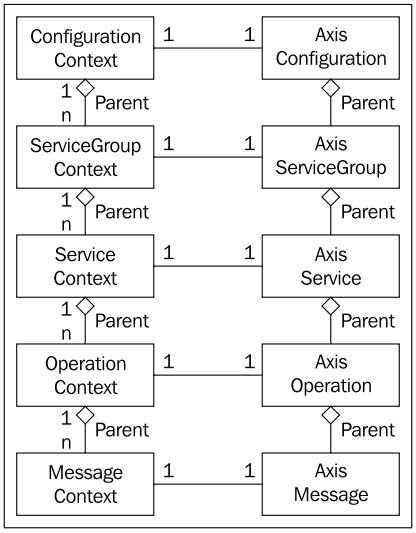
A handler always runs inside a phase, and the phase provides a mechanism to specify the ordering of handlers. Both pipes have built-in phases, and both define the areas for 'User Phases', which can be defined by the user, as well.

Information Model

As shown in the figure below, 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 hot deployment is turned off, then the description hierarchy is not likely to change. If hot deployment is turned on, then we can deploy the service while the system is up and running. In this case, the description hierarchy is updated with the corresponding data of the service. The context hierarchy keeps run-time data. Unlike 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 key value pairs. When the values are to be searched for at a given level, they are searched 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 for 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 then it would search for in the Description hierarchy as well.

This allows the user to declare and override values, with the result being 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.



Deployment Model

The previous versions of Axis failed to address the usability factor involved in the deployment of a Web Service. This was due to the fact that Axis 1.x was released mainly to prove the Web Service concepts. Therefore in Axis 1.x, the user has to manually invoke the admin client and update the server classpath. Then, you need to restart the server in order to apply the changes. This burdensome deployment model was a definite barrier for beginners. Axis2 is engineered to overcome this drawback, and provide a flexible, user-friendly, easily configurable deployment model.

Axis2 deployment introduced a 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.

The concept of hot deployment and hot update is not a new technical paradigm, particularly for the Web Service platform. But in the case of Apache Axis, it is a new feature. Therefore, when Axis2 was developed, "hot" deployment features were added to the feature list.

- **Hot deployment**: This refers to the capability to deploy services 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, then the loss might be substantial and it may affect the viability of the business. In the meanwhile, it is required to add new service to the system. If this can be done without needing to shut down the servers, it will be a great achievement. Axis2 addresses this issue and provides a Web Service hot deployment ability, wherein we need not 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, which is best suited to use 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 in Web Services. There are currently two approaches to invoking a Web Service in a non-blocking manner. The first is the client programming model, where a client invokes the service in a non-blocking manner. The second way is the transport level non-blocking invocation where invocation occurs in two transports (it could either be two single-channel transports like SMTP, or two double-channel transports like HTTP). Axis2 client API supports both the non-blocking invocation scenarios.

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

ServiceClient has the following API for invoking a service:

- **sendRobust**: The whole idea of this is to just send an XML request to the Web Service and not care about its response. However, if something goes wrong, we require 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 either the response, or any 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, which is called once the invocation is complete.

As mentioned earlier, the OperationClient class is for advanced users, and working with OperationClient requires us to know Axis2 in depth. In ServiceClient, we do not have to know anything about SOAP envelope, message context, and so on. 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:

- Create a ServiceClient.
- Then create OperationClient with the use of the ServiceClient that we have created.
- Create SOAP envelop.
- Create Message context.
- Add the SOAP envelope to message context.
- Add the messagecontext to OperationClient.
- Then invoke the OperationClient.
- If there is a response, then 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". We can define transports along with senders and receivers in Axis2 global configuration. The Transport Receiver is the one via 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 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 email account. The transport receiver is a thread that checks for emails at fixed time intervals.
- **JMS**: This provides a way to invoke a Web Service using the JMS way.
- **XMPP**: This provides a standard way to communicate with Jabber server, and to invoke Web Services using XMPP protocol.

Other Modules

- **Code Generation**: Axis2 provides a code generation tool that generates server-side (skeleton) and client-side (stub or proxy) code along with descriptors and a test case. The generated code simplifies the service deployment and the service invocation. This increases 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

Although the basic objective of the code generation tools has not changed, the code generation module of Axis2 has adopted a different approach. Primarily, the change is in the use of templates, namely XSL templates, which gives flexibility to the code generator so as to generate code in multiple languages.

Data Binding

Databinding for Axis2 is implemented in an interesting manner. Databinding 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 that is passed to the code generator wherein the emitter operates.

Axis2 supports the following data binding frameworks including its own data binding framework called ADB:

- **ADB:** ADB (Axis2 Data Binding) is a simple and lightweight framework that works off StAX and is fairly performant.
- **XMLBeans:** XMLBeans is preferred if we want to use full schema support 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 extensions.

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 below:

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

Service Extension of 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 the above problem as well as to add new features, Axis2 introduced the concept of Web Service extensions or a modules, wherein 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 Axis2 modules over Axis 1.x handler chains is that we 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 modules and drop them into the modules directory of the Axis2 repository. We can engage the module at deployment time by adding `<module ref="module name"/>` to `axis2.xml` (global configuration file). In addition to that, if we want to engage a module at run time, 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. We 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 to support 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, then he or she can achieve that goal with the use of custom deployers. We will discuss more about custom deployers in Chapter 5.

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, in itself, is a handler. However, it is different from other handlers because Axis2 treats the Message Receiever in a different manner. 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, then 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 inside 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 handled over to the message receiver.

Summary

Axis2 is enterprise-ready. Its Web Service engine provides a better SOAP processing model, with considerable increase in performance for both speed and memory usage with respect to Axis 1.x and other existing Web Service engines. In addition, it provides the user with a convenient API for service deployment, 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. Thus, we have learned that Axis2 architecture helps in attaining a more flexible and extensible Axis2.

3
AXIOM

AXIOM stands for AXIs Object Model and refers to the XML Infoset model that was initially developed as a part of Apache Axis2. However, later it became a WS Commons Project so that the projects other than Axis2 could also benefit from it. XML Infoset refers to information included in XML. For programmatic manipulation, it is convenient to have a representation of this XML Infoset in a language-specific manner. For an object-oriented language, a model made up of objects would be a the best choice. DOM and JDOM are two such XML models. AXIOM is conceptually similar to such an XML model by its external behavior. At the end of this chapter, you will understand the basics in AXIOM and the best practices to be followed while using AXIOM.

Overview and Features

AXIOM is a lightweight, deferred built XML Infoset representation based on StAX (JSR 173), which is the standard streaming pull parser API. The object model can be manipulated flexibly just like any other object model (such as JDOM). But underneath, the objects will be created only when they are absolutely required. This leads to much less memory intensive programming.

Looking at the features of AXIOM, deferred building can be considered the best. And that was one of the design goals of AXIOM too. One of the drawbacks of Axis1 is its XML representation. AXIOM was introduced to solve those issues, and in addition to that, has the following key features as well:

- **Lightweight**: AXIOM is specifically targeted to be lightweight. This is achieved by reducing the depth of the hierarchy, the number of methods, and the attributes enclosed in the objects. This makes the objects less memory intensive.

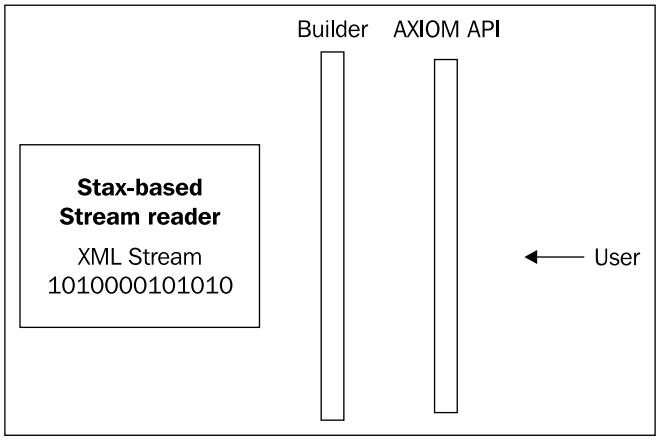
- **Deferred building**: This is one of the most important features of AXIOM. The objects are not made, unless a need arises for them. This passes the control of building over to the object model itself, rather than to an external builder.
- **Pull-based**: For a deferred building mechanism, a pull-based parser is required. AXIOM is based on StAX, the standard pull parser API.

What is Pull Parsing?

Let us understand the meaning and the concept behind pull parsing. An XML document can be parsed by using either a "pull-based" process, or a "push-based" process. Pull parsing is a recent trend in XML processing. The XML processing frameworks such as SAX and DOM are "push-based", which means that the control of the parsing is in the hands of the parser itself. This approach is fine and easy to use, but was not efficient in handling large XML documents, since a complete document object model was generated in the memory. Pull parsing inverts the control, and hence, the parser only proceeds at the users command. The user can decide to store or discard events generated by the parser. OM is based on pull parsing.

AXIOM—Architecture

The AXIOM architecture is quite simple. Before we delve into the architecture of AXIOM, we need to learn the basic concepts and usage of AXIOM. The AXIOM user API, the actual XML stream, and the AXIOM components are shown in the following figure:



AXIOM is also known as the OM (Object Model), and the OM Builder wraps the raw XML character stream through the StAX reader API. Hence, the user does not see the complexities of the pull event stream.

In addition to the deferred building capability, we have another useful feature. This feature includes caching and non-caching building of the XML tree. Since AXIOM is a deferred built object model, it incorporates the concept of caching. Caching refers to the creation of the objects while parsing the pull stream. The reason why this is so important is because caching can be turned off in certain situations. If so, the parser proceeds without building the object structure. The user can extract the raw pull stream from the AXIOM element and use that instead of navigating the AXIOM element. In this case, it is sometimes beneficial to switch off caching.

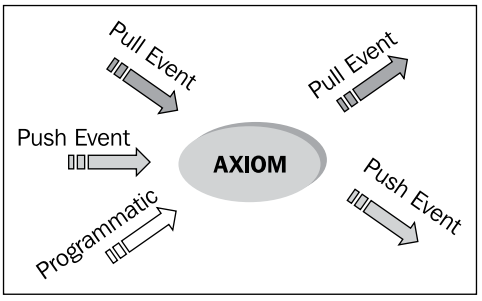
Working with AXIOM

We can either download the AXIOM binary, or we can build the binary using the source distribution (or from the source repository). AXIOM was started as a part of Axis2, but now, it has its own release cycle. Therefore, we can either download the an AXIOM binary in AXIOM release, or we can find the AXIOM binary in the Axis2 release.

Once we have the AXIOM binary, the next step is to add that binary into our classpath (and the dependent binary files as well). Then, we can start working with AXIOM. If your application has a build system similar to Maven's, then you can add dependency to that, and let it download AXIOM JARs automatically.

Creating an AXIOM

We can create an AXIOM in three ways as shown in the following figure.We can create an AXIOM using either the Pull Event stream or the Push Event stream, or, we can create an AXIOM programatically. In this chapter, we will learn how to create an AXIOM using a Pull Event stream, as well as how to create an AXIOM programmatically, as those are the two most common methods that are used to create an AXIOM.



First, let us have a look at how we can create an AXIOM by using a pull event stream. AXIOM provides the concepts of a factory and a builder, in order to create objects. The factory helps to keep the code at the interface level and the implementations separate. Since AXIOM is tightly bound to StAX, a StAX-compliant reader should be created first with the desired input stream. Then, one can select a builder from those that are available. In AXIOM, we can find different types of builders provided for the convenience of different users. AXIOM has OM builders as well as SOAP builders. So, we can use the appropriate builder for our requirements. 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 an AXIOM from an Input Stream

The code below demonstrates the correct method of creating an AXIOM document from a file input stream.

```
//create the parser
XMLStreamReader parser = XMLInputFactory.newInstance().createXMLStream
Reader(new FileInputStream(file));
//create the builder
StAXOMBuilder builder = new StAXOMBuilder(parser);
//get the root element (in this case the envelope)
OMElement documentElement = builder.getDocumentElement();
```

The following steps should be adopted to create an AXIOM from an input stream:

Step 1: First, we need to create a parser or a reader. In this case, we will be creating a parser.

Step 2: Next, we create the builder by using the parser (or reader). In this case, we create StAXOMBuilder.

Step 3: Get the AXIOM document element from the builder.

Note that when we ask for the document element from the builder, it will give us a pointer to the AXIOM wrapper. But the XML stream is still present in the stream, and the object tree is not created at that time. The object tree is created only when we navigate or build the AXIOM.

Creating an AXIOM Using a String

Now let us try to create an AXIOM document from a string, which is very simple and easy.

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

Thus, we observe that when we create an AXIOM from a string, we first get an input stream, and then we can follow the same procedure as we have followed earlier. From the above example, it is clear that creating an AXIOM from an input stream or from a string is quite simple. However, elements and nodes can also be created programmatically to modify the structure of the AXIOM element, which we have created above. The recommended way to create AXIOM objects programmatically is to use a factory.

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

Creating an AXIOM Programmatically

Creating an AXIOM programmatically is a bit more difficult when compared to the previous two cases. It also involves some additional 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);
```

We can see a set of `factory.create` methods. These can be used to cater for different implementations while keeping the programmers code intact. In AXIOM, it is better to use the factory for creating AXIOM objects, as 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 has 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 being 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 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 the OMElement). Hence, one can access a SOAP document either by the abstraction of SOAP, or drill down to the underlying XML Object model with a simple casting.

Adding a Child Node and Attributes

So far, we have learned to create an AXIOM programmatically, and by using StAX API, But this is not enough to work with an AXIOM. We need to learn how to create and add a child node to an 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 that we have created by adding "name" and "isbn" child elements to the root element.

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

- `addChild()` 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 a particular node internally.

- Namespaces are a tricky part of any XML object model, and it is the same in AXIOM. However, the interface to the namespace has been made very simple. **OMNamespace** is a 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 have discussed, namespace handling forms a key part of XML processing. Hence, AXIOM provides a set of API methods to handle namespaces.

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

From the above, it is clear that there are two `declareNamespace` methods that are simple to understand. Furthermore, they are used to add a namespace to the namespace declarations section.

A namespace declaration that has already been added once can not be added again.

The `findNamespace` is a very handy method to locate a namespace object in the object tree. It looks for a matching namespace in its own declaration section and jumps to the parent node, if it is not found. The search process progresses up the tree, until a matching namespace is found, or the root has been reached.

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

We will learn to serialize the AXIOM element later in this chapter. However, if we serialize the element that we have created, then we get the following output:

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

Working with Attributes

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 we serialize the element again, then we will get the 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, we learned how to create an AXIOM element. We have learned to create and add child nodes, and also to create namespaces and attributes. Now, let us try to traverse the AXIOM tree.

Traversing the object structure can be done in the usual way, by using the list of child nodes. However, in AXIOM, the child nodes are returned as an iterator. The iterator supports the 'OM way' of accessing elements, and is more convenient than a list for sequential access. The code sample below demonstrates how to access the child nodes in a given node. The children are of the type OMNode, and can be either of type OMText, or OMElement.

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

In addition to this, every `OMNode` has links to its siblings. If more thorough navigation is required, the `nextSibling()` method or the `PreviousSibling()` method can be used. A more selective set can be chosen by using the `getChildrenWithName (QName)` methods. The `getChildWithName (Qname)` method returns an iterator having all the child elements with the given Qname. The advantage of these iterators is that they will not build the whole object structure at once, but only when it is required.

Serialization

Now, since we have a good understanding of creating and traversing the AXIOM tree, we will learn how to serialize or write the AXIOM tree into an output stream.

The AXIOM can be serialized either as a pure object model, or a pull event stream. The serialization uses a XMLStreamWriter object to write the output, and hence, the same serialization mechanism can be used to write different types of outputs (such as text, binary, and so on).

A caching flag is provided by AXIOM to control the building of the in-memory AXIOM. The OMNode has two methods, `serializeAndConsume` as well as `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 such a case, it will serialize the XML stream directory to the output stream, without creating the object model. If we call the `serializeAndConsume` method, then we can `serialize` the AXIOM tree only once, since it does not build the AXIOM tree into memory. However, we can call the `serialize` method any number of times. We will learn the difference between the above two methods later in this section.

The serializer serializes namespaces in the following ways:

- When a namespace is encountered that is in scope and not yet declared, then it will be declared.
- When a namespace is encountered that is in scope and is already declared, then the existing declarations prefix is used.
- When the namespaces are declared explicitly using the elements' **declareNamespace()**, they will be serialized even if those namespaces are not used in that scope.

Because of these behaviors, if a fragment of the XML is serialized, it will also be namespace-qualified with the necessary namespace declarations. Here is an example that shows how to write the output to the console:

```
XMLStreamWriter writer = XMLOutputFactory.newInstance().createXMLStrea
mWriter(System.out);
//dump the output to console with caching
root.serialize(writer);
writer.flush();
```

Now let us try to understand the difference between the `serializeAndConsume()` and `serialize()` methods. First, let us try to call the serialize method twice in an 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 we want the XML as a plain XML,
                                                    we 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 we run the preceding sample code, then we 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 we call `serializeAndConsume()` first, and then call `serialize()`, we get an exception. This is because once the `serializeAndConsume()` method is called, the AXIOM tree will not be built, and the cache flag is reset. So the next time we try to call the serialize method, we have nothing left to serialize and hence 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(x
mlStream.getBytes());
//create a builder. Since we want the XML as a plain XML, we 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

We know how to create and serialize AXIOM. But this is not sufficient for doing advanced operations with Axis2 or AXIOM. For that, we need to learn the following:

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

Using OMNavigator for Traversing

AXIOM provides a utility class in order to navigate the AXIOM structure. The navigator provides an in-order traversal of the AXIOM tree up to the last-built node. The Navigator has two states called the "navigable state" and the "completion state". Since the navigator provides the navigation starting from an OMElement, it

is deemed to have completed the navigation when the starting node is reached again. This state is known as the completion state. Once the navigator has reached the complete status, its navigation is done and it cannot proceed.

It is possible that the AXIOM tree does not get built completely when it is navigated. The navigable status shows whether the tree structure is navigable. Once the navigation is complete, we cannot navigate it further. However, it is possible for a navigator to become non-navigable without being complete. The following code sample shows how the navigator should be used and handled by using its states:

```
//Create a navigator
OMNavigator navigator = new OMNavigator(root);
OMNode node = null;
while (navigator.isNavigable()) {
node = navigator.next();
}
```

Xpath Navigation

AXIOM has Xpath navigation support through Jaxen. So, we can write an Xpath query and invoke it in AXIOM. Let us write an Xpath query to get the 'isbn' number from the book element that we have created. If we run the following sample, then we 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

AXIOM is tightly integrated with StAX and the `getXMLStreamReader()` method or `getXMLStreamReaderWithoutCaching()` method in the OMElement provides a XMLStreamReader object. 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 not visible to the user.

AXIOM has the concept of caching, and is the actual cache of the events that are being fired. However, the requester can choose to get the pull events from the underlying stream, rather than from the AXIOM tree. This can be achieved by getting the pull-parser with the cache off. If the pull parser was obtained without switching off caching, the new events fired will be cached, and the AXIOM tree will be updated. Thus, the returned pull parser will switch between the object structure and the stream underneath. The users need not worry about the differences caused by the switching. The exact pull stream that we have seen from the original document would be produced even if the AXIOM tree was fully or partially built. The `getXMLStreamReaderWithoutCaching()` method is very useful when the events need to be handled in a pull-based manner without any intermediate models. This in turn makes the operations faster and more efficient.

AXIOM and SOAP

We have already discussed that AXIOM was developed as an XML representation mechanism in Axis2. As a SOAP processing framework, Axis2 needs to work with SOAP. We know that SOAP is also XML, but has its own structure. So, it is easy if we can get a SOAP level API from AXIOM. Therefore, AXIOM has in-built support for SOAP representation and navigation. We can easily create SOAP 1.1 and 1.2 documents with AXIOM and navigate them. When we navigate SOAP, AXIOM has inbuilt support for getting the SOAP header and SOAP body. Therefore, we do not need to get an iterator and navigate. We can call the getHeader method and the getBody method rather than the getChild method. The following code samples show how we can create the SOAP 1.1 document and SOAP 1.2 document with AXIOM.

Creating a SOAP 1.1 Document

```
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);
```

From the preceding code, it is evident that we first create a book element as the body of the SOAP message, and then we create the default SOAP 1.1 envelope and add the created book element as the body.

Creating a SOAP 1.2 Document

Creating the SOAP 1.2 document is almost the same as before, except for the factory. In the following sample, we need to use a 1.2 factory, instead of a 1.1 factory.

```
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 have learned how to create, serialize, and navigate an AXIOM in a number of ways. We have also learned SOAP handling in AXIOM, which is an important part when we consider AXIOM in Axis2.

4
Execution Chain

The fundamental goal of any given SOAP processing framework is to deliver an incoming SOAP message to the target application. However, if we consider today's Web Services or any other application, just delivering the message to the application is not sufficient. We need to provide quality of service, such as reliability and security. To provide these features, most SOAP processing frameworks have the concept of pipes, where, any incoming or outgoing message goes through the pipe, and the pipe gets divided into smaller pieces. Each piece is known as an interceptor.

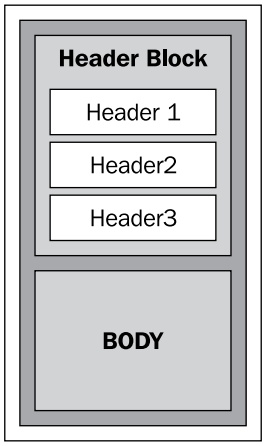
Handler

If you have used any version of Apache Axis, then you will be familiar with the term "Handler"—the Apache terminology for "message interceptor". In any messaging system, the interceptor has its factual meaning in the context of messaging, where it intercepts the flow of messaging and does whatever task it is assigned to do. In fact, an interceptor is the smallest execution unit in a messaging system, and an Axis2 handler is also an interceptor.

Handlers in Axis are stateless, that is, they do not keep their pass execution states in the memory. A handler can be considered as a logic invoker with the input for the logic evaluation taken from the MessageContext (We will learn more about MessageContext in Chapter 6.) A Handler has both read and write access permissions to MessageContext (MC) or to an incoming SOAP message.

For continuity purposes, we can consider MessageContext as a property bag that keeps incoming or outgoing messages (maybe both) and other required parameters. It may also include properties to carry the message through the execution chain. On the other hand, we can access the whole system including the system runtime, global parameters, and property service operations via the MC.

In most cases, a handler only touches the header block part of the SOAP message, which will either read a header (or headers), add a header(s), or remove a header(s). (This does not mean that the handler cannot touch the SOAP body, nor does it mean that it is not going to touch the SOAP body.) During reading, if a header is targeted to a handler and is not executing properly (the message might be faulty), then it should throw an exception, and the next driver in the chain (in Axis2, it is the Axis engine) would take the necessary action. A typical SOAP message with few headers is shown in the figure given below:



Any handler in Axis2 has the capability to pause the message execution, which means that the handler can terminate the message flow if it cannot continue. Reliable messaging (RM) is a good example or use case for that scenario, when it needs to pause the flow depending on some of the preconditions and the postconditions as well and it works on a message sequence. If a service invocation consists of more than one message, and if the second message comes before the first one, then the RM handler will stop (or rather pause) the execution of the message invocation corresponding to the second message until it gets the first one. And when it gets, the first message is invoked, and thereafter it invokes or resumes the second message.

Writing a Simple Handler

Just learning the concepts will not help us in remembering what we have discussed. For that, we need to write a handler and see how it works. Writing a handler in Axis2 is very simple. If you want to write a handler, you either have to extend the `AbstractHandler` class or implement the Handler interface.

A simple handler that extends the `AbstractHandler` class will appear 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;
    }
}
```

Note the return value of the invoke method. We can have the following three values as the return value of the invoke method:

- **Continue**: The handler thinks that the message is ready to go forward.
- **Suspend**: The handler thinks that the message cannot be sent forward since some conditions are not satisfied; so the execution is suspended.
- **Abort**: The handler thinks that there is something wrong with the message, and cannot therefore allow the message to go forward.

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

When a message is received by the Axis engine, it calls the invoke method of each of the handlers by passing the argument to the corresponding MessageContext. As a result of this, we can implement all the processing logic inside that method. A handler author has full access to the SOAP message, and also has the required properties to process the message via the MessageContext. In addition, if the handler is not satisfied with the invocation of some precondition, the invocation can be paused as we have discussed earlier (Suspend).

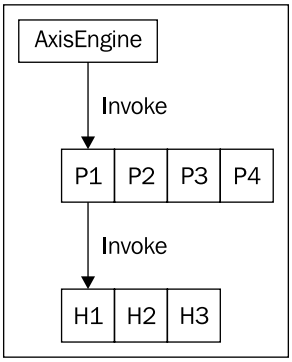
If some handler suspends the execution, then it is its responsibility to store the message context, and to forward the message when the conditions are satisfied. For example, the RM handler performs in a similar manner.

Phase

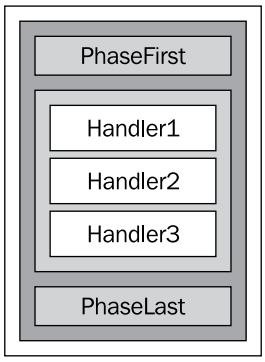
The concept of phase is introduced by Axis2, mainly to support the dynamic ordering of handlers. A phase can be defined in a number of ways:

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

A flow or an execution chain can be considered as a collection of phases. Even though it was mentioned earlier that an Axis engine calls the invoke method of a handler, that is not totally correct. In fact, 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 (refer to the following figure). As we know, we can extend AbstractHandler and create a new handler; in the same way one can extend the Phase class and then create a new phase. But remember that we need not always extend the Phase class to create a new phase. We can do it by just adding an entry into `axis2.xml` (adding a phase to `axis2.xml` is explained in Chapter 6). A phase has two important methods—precondition checking and postcondition checking. Therefore, if we are writing a custom phase, we need to consider the methods that have been mentioned. However, writing a phase is not a common case; you need to know how to write a handler.



A phase is designed to support different phase rules. These rules include phaseFirst as well as phaseLast. In a phase, there are reserved slots to hold both phaseFirst handlers and phaseLast handlers. The rest of the handlers are in a different list. A phase can, therefore, be graphically represented as follows:



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

- First, check whether the precondition is satisfied.
- Then check whether the phaseFirst handler is there. If it is present, then it will be invoked.
- Next, the rest of the handlers will be invoked, except for the phaseLast handler.
- If the phaseLast handler is there, then it will be invoked.
- Finally, it will check whether the postcondition is satisfied so as to forward the message.

Types of Phases

There are two types of phases defined in `axis2.xml`. There are no differences between the ways the logic is implemented. However, the location of the phase in the execution chain differs. The two types of phases are:

1. Global phase
2. Operation phase

Global Phase

A global phase is a phase that is invoked irrespective of the service. In simple terms, whenever a message comes into the system, it will go through the global phase. The whole idea of defining a global phase and an operation phase in `axis2.xml` is to provide an easy path for module authors. In Chapter 8, we will see how module authors create their modules along with the module descriptor file, and how the module descriptor makes assumptions about the different phases that are defined in the `axis2.xml` file.

If we consider the default `axis2.xml` file which is present in our download directory, we will observe that it has a set of global as well as operation phases. The splitting point of the global phase and operation phase is the Dispatch phase. All the phases up to and including Dispatch phase are considered global phases, whereas the rest are considered operation phases.

The phase section of the default `axis2.xml` file is given below. It is a bit complicated. But all you need to do is focus on the keyword 'phase'.

```
<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 or she wants      -->
        <phase name="OperationInPhase"/>
        <phase name="soapmonitorPhase"/>
    </phaseOrder>
```

In the previous code, '`Transport`', '`Security`', '`PreDispatch`', and '`Dispatch`' are the global phases while '`OperationInPhase`' and '`soapmonitorPhase`' are the operation-specific phases.

All the global phases have semantic meanings from their names as well. Transport phase consists of a handler, which performs the tasks that are dependent on the type of transport. In the Security phase, WS-Security implementation will include their handlers. As the name implies, PreDispatch phase is the phase consisting of a set of handlers that performs the work needed for dispatching. Handlers such as WS-Addressing, are always included in this phase. Dispatch phase is the phase that does the dispatching by simply finding the corresponding service as well as the operation for the incoming message. As a result, it is evident that Dispatch phase consists of dispatching handlers.

Operation Phase

For instance, suppose that we have a handler, that we do not have to run for every message coming to the system, rather we need to run that for a selected set of operations. This is where the operation phase comes into the picture.

Phase Rules

The main idea of the phase rule is to correctly locate a handler, relative to the one that is inside a phase, either at deployment time or at run time. In Axis1, we did not have the concept of phases or phase rules. What it had was a global configuration file wherein we defined our handlers. But it had a number of limitations such as loosing the dynamic nature of the handler chain. One aspect of phase rule is addressing the issues of dynamic execution chain-building capability.

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

- **Phase name**: Name of the phase where the handler must be placed.
- **Phase first** (phaseFirst): First handler of the phase.
- **Phase Last** (phaseLast): Last handler of the phase.
- **Before** (before): should be positioned before a given handler.
- **After** (after): should be positioned after a given handler.
- **Before and after**: should be placed between two given handlers.

Phase Name

"Phase" is a compulsory attribute for any phase rule, which specifies the phase in which the handler must fit. For validity, the name of the phase should be either in the global phase, or in the operational phase, and should be defined in `axis2.xml`.

phaseFirst

If we want a handler to be invoked as the first handler in a given phase, irrespective of the other handlers in the phase, then we need to set the phaseFirst attribute to "true". A handler, which has phaseFirst as the only phase rule, is shown below:

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

phaseLast

If we want a handler to run last in a given phase irrespective of the other handlers, then the phaseLast attribute should be set to "true". Refer to the following code:

```
<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 attributes set to true, then that phase cannot have any other handlers. As a result, the phase has only one handler.

before

There may be situations where a handler should always run before some other handler, irrespective of its exact location. A real-time use case for this can be a security handler, which needs to run before the RM handler. Refer to the following code
to understand the logic. Here, the value of the 'before' attribute is the name of the handler before which this handler must run.

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

Axis2 phase rule processing logic is implemented in such a way that if the handler referred to by the attribute 'before' is not available in the phase but only at the time the rule is being processed, then it just ignores the rule and the handler is placed immediately after the phaseFirst handler. (If it is available, it places the handler somewhere in the phase.)

after

As before, if a handler always runs after some other handler, then the phase rule can be written using the attribute 'after', and the value of 'after' attribute is the name of the handler after which this handler must run.

```
<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 between two different handlers, then the phase rule can be written using both `before` and `after` attributes. The values of both `before` and `after` attributes are the names of the relevant handlers. Here is the appropriate way of writing a phase rule.

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

Invalid Phase Rules

The 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. But when writing a phase rule, it is required to check whether the rule is a valid rule. There are many ways in which a phase rule can become an invalid rule. Some of them are:

1. If there is a phase rule for a handler with either the phaseFirst or the phaseLast attribute set to true, then it can have neither 'before' nor 'after' appearing in the phase rule. If they do, then the rule is invalid.
2. If there is a phase rule for 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.

3. There cannot be two handlers in one phase with their phaseFirst attribute set to true.
4. There cannot be two handlers in one phase with their phaseLast attribute set to true.
5. If the rule is such that the attribute 'before' refers to a phaseFirst handler, then the rule is invalid.
6. If the rule is such that the attribute 'after' refers to a 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 on a per-handler basis, and any handler in the system must fit into a phase in the system.

Flow

Flow is simply a collection of phases. The order of phases inside a flow is defined in `axis2.xml`. Since phase is a logical collection (which is in fact a virtual concept), a flow can be assumed to be the execution chain (collection of handlers).

The following are the four types of flows in Axis2:

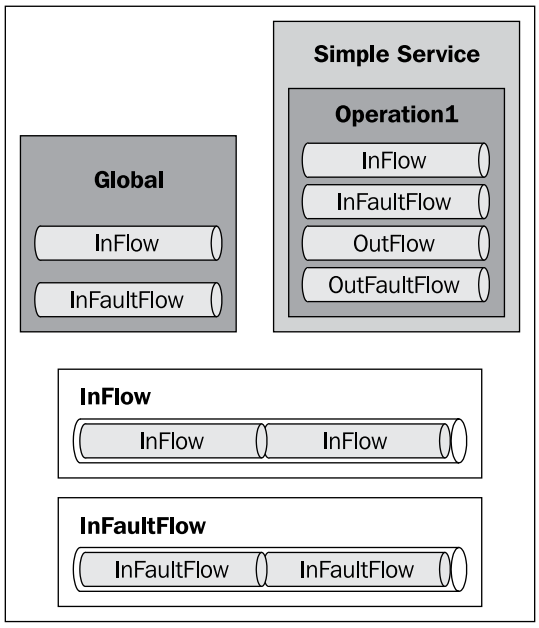
1. InFlow
2. OutFlow
3. InFaultFlow
4. OutFaultFlow

InFlow: When a message comes in (request message), it has to go via the InFlow. This in turn invokes all the handlers in the InFlow. InFlow is somewhat different from OutFlow. The Inflow consist of two parts. The first part starts from the Transport Receiver, and ends at Dispatcher (up to and including Dispatch phase). The second part will be there only if the corresponding service and operations are found at the end of Dispatch phase. Therefore, the second part of the flow is the InFlow of the corresponding operation for the incoming message. So the InFlow consists of global as well as operation parts.

InFaultFlow: This flow is 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), the OutFlow 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 the OutFaultFlow is invoked.



Module Engagement and Dynamic Execution Chain

In Chapter 8, 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, then that will affect all the services in the system, and there is a probability of changing both the global and operations flows (each of the operations in the service). And if a module is engaged to a service, then the flows belonging to all the operations in that particular service will be changed. If a module is engaged to an operation, then each 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 result in a change in the handler chain dynamically. A module can be engaged dynamically (at run time) or statically (at deployment time). If we want to engage a module statically, then we need to specify it in the description file. (If it is to a service or to an operation, then it must be in the `services.xml` file, and if it is globally, then it must be specified in the `axis2.xml` file).

Special Handlers in the Chain

When we consider the execution chain we can find the following four types of special handlers in the execution chain:

1. TransportReceiver
2. Dispatcher(s)
3. MessageReceiver
4. TransportSender

TransportReceiver

Whenever a message comes into the system, it will first reach a transport receiver, which can therefore be considered as an agent that is waiting to accept incoming message. (In the case of the Application server, the transport receiver could be a servlet). Therefore, the InFlow of the execution chain always starts with the transport receiver.

Dispatchers

As we have discussed earlier, one of the fundamental goals of SOAP processing frameworks is to deliver an incoming message in the targeted application. The process of finding the correct targeted application is called dispatching. In Axis2, dispatching takes place in the middle of the incoming execution chain, where dispatchers are the handlers in the chain. When we consider dispatching, the following ways can be adopted:

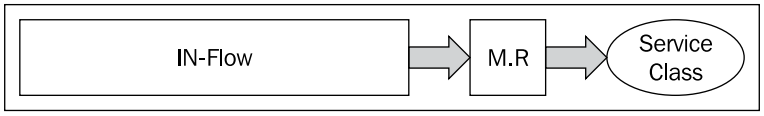
- Using Transport headers and transport-level data
- Using WS-Addressing information
- Using the incoming SOAP message

To cater to the above mentioned types of dispatching, Axis2 has a set of default dispatchers and we can change the order of their execution using `axis2.xml`. If we look at the InFlow element that we discussed in the global phase section, we will find the set of available dispatchers in the Dispatch phase:

- **RequestURIBasedDispatcher**: Tries to find the service and operation using the transport URI.
- **SOAPActionBasedDispatcher**: Tries 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 a SOAP message, especially the body, to find the operation.
- **HTTPLocationBasedDispatcher**: Is used to dispatch WSDL 2.0-related SOAP messages.

MessageReceiver

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



On the other hand, it is the MessageReceiver that directly interacts with both the actual service implementation class and the Axis Engine. (There may be instances where the service itself is a MessageReceiver.) In Axis 1.x, we have the concept of the "pivot point" where a request path and response path meet together, and where actual service invocation takes place. As mentioned earlier, the MessageReceiver is the end of the inflow that interacts with the service implementation class. Therefore, axis2 has got nothing to do with it once it hands over the message to the MessageReceiver.

The Axis2 distribution includes a set of message receivers to support commonly used MEPs (In-Out and In-only) as well as to support the JavaBean case.

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

TransportSender

As we discussed earlier, the transport receiver is the starting handler of the inflow. However, it is the transport sender that runs in the outFlow as the last handler of the outflow. We can have different types of transport senders for different transports. For example, Axis2 has transport senders for HTTP, SMTP, TCP, and so on. When we send the message using HTTP transport, the HTTP sender is invoked. On the other hand, if we are sending the message via SMTP, then the SMTP transport sender is invoked.

Summary

Axis2 is good enough to provide Web Service interaction with a dynamic and flexible execution framework. Flexibility is achieved using the concept of phases and phase rules, whereas the dynamic nature of the execution chain has been achieved by run-time module engagement.

5
Hacking Deployment

It is not just important to have features, but we need to have features that are user friendly. In the meantime, hardware resources are no longer a problem for users. The only concern is that it must not be time-consuming. In the previous versions of Apache Axis, user-friendliness was not given high priority, as they were mainly used to prove the Web Service concepts. Therefore, in Axis 1.x, the user has to invoke the admin client manually and update the server classpath. Then, the user needs to restart the server in order 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 easily configurable deployment model.

What's New in Axis2 Deployment?

The Axis2 deployment model has introduced a number of new features the into Apache Web Service stack, some of which are already familiar from the Web Service paradigm. However, they are new compared to Apache Axis 1.x. The following are the key features of its deployment model:

- J2EE-like deployment mechanism (archive based)
- Hot deployment and Hot update
- Repository
- Change in the way of deploying handlers (Modules)
- New deployment descriptors
- Available deployment options

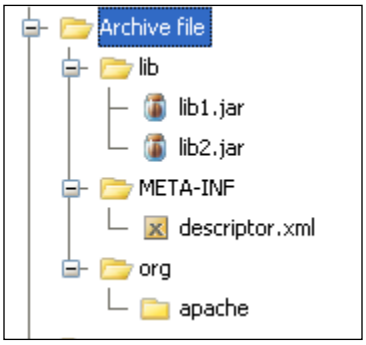
J2EE-like Deployment Mechanism

In any J2EE application server, we can deploy an application as a self-contained package, where we can bundle all our resources, configuration files, and binary files together into one file and deploy it. As this is clearly useful, Axis2 has introduced the same mechanism to deploy services (and modules) in a very convenient manner.

Let us think of a scenario wherein we have a service with several third-party dependencies, and a number of property files, but do not have a J2EE-like deployment mechanism. Then, what we have to do is to put all those dependent JAR files and property files into the application classpath. That would be double, if we have two servers. But what will happen if we are in a clustering environment with hundreds of replicas? Practically, it will not be possible to go and add dependent JAR files and other resources into the classpath of each and every replica. But, when we have a J2EE-like deployment mechanism, we do not need to worry about such issues. We can just go and drop the self-contained package, in this case, the service archive file, into the replicas. This definitely reduces our work, and prevents common human errors as well.

The internal structure of an Axis2 self-contained package (or archive file) is shown in the following figure. Both service archives and module archives have an almost identical structure with a few minor disparities.

1. In the case of an Axis service archive, the `descriptor.xml` file gets converted to `services.xml`, and in the other case, it gets converted to `module.xml` file.
2. The file extension for an Axis2 service archive is "`.aar`" and the file extension for a module archive is "`.mar`" (a service archive or module archive is just a ZIP file with a change in the file extension to either `.aar` or `.mar`).



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

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. Even a short amount of downtime can be highly detrimental; so restarting a server is not a good option. We need to update and change the system without shutting it down. This is how hot deployment and hot update come into the picture. It is clear that when our application has these features, then we do not need to shut down the system for updating.

The concept of hot deployment and hot update is not a new terminology in technical paradigms, although it is a new feature in the Apache Axis Web Service stack.

- **Hot deployment:** This is the ability to deploy new services while the system is up and running. As an example, let's say that we have two services, 'service1' and 'service2' that are up and running, and we deploy a new service called 'service3' without shutting the system down. Then, the system makes 'service3' a running service. This scenario is called hot deployment.

  As a system administrator, if we do not like the hot deployment of a service, then we can turn that off easily by editing the Axis2 global configuration file called `axis2.xml`, and changing the global configuration parameter as follows:

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

- **Hot update:** This is the ability to make changes to an existing Web Service, without shutting down the system. It is an important feature and is required in a testing environment. However, it is not advisable to use hot update in a real-time system, because a hot update could result in a system that is in an unknown state.

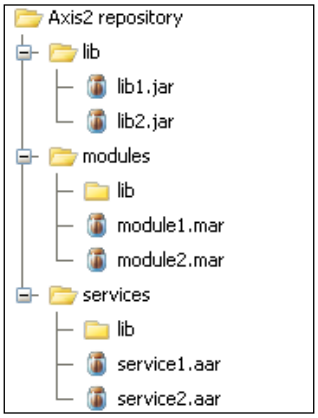
  Additionally, there is a possibility of loosing the existing service data of that service. To prevent this, Axis2 comes with the hot update parameter set to FALSE, by default, and if we want to use this feature, we need to change the configuration parameter as follows:

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

Repository

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

The repository directory consists of two main subdirectories called **services** and **modules**. We have an optional subdirectory called **lib** as well. If we want to deploy a service, then we need to drop the service archive file into the **services** directory. Similarly, if we want to deploy a module, then we need to drop a module archive file into the **modules** directory. The idea behind the **lib** directory is to place the third-party libraries that are going to be shared across both services and modules in it.



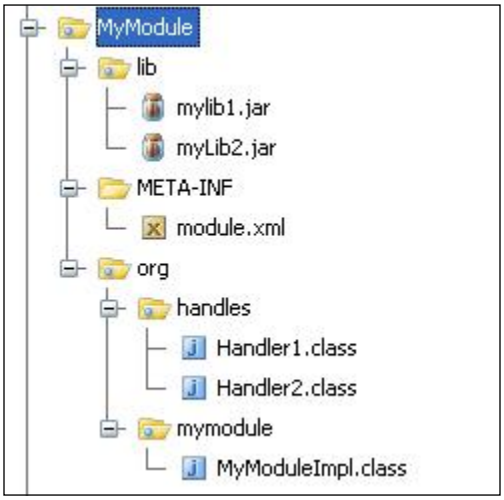
If one or more modules in the **modules** directory have to share some resources, then they have to add those resources into the **lib** directory, inside the **modules** directory. Similarly, if a number of services from the **services** directory want to share some resources, then they need to place them in the **lib** directory, inside the **services** directory.

Change in the Way of Deploying Handlers (Modules)

The concept of service extension is a new feature of the Apache Axis paradigm, but in Axis 1.x, the developers had to work hard to achieve the same goal. So the idea is to extend the core functionality of the system and to provide quality of service. In the case of Axis 1.x, if we have to extend the core functionality, then we have to write a handler (the smallest unit in the execution chain), and also change the global configuration files to add the handler, and finally restart the system.

A module does the same amount of work, but reduces the amount of work we need to do. In the meantime, a module can have one or more handlers, and these handlers should have only a module descriptor, "`module.xml`". Most of the time, a module is an implementation of a specific WS specification. As an example, the Axis2 addressing module is an implementation of WS-Addressing, and Sandesha is an implementation of WS-Reliable Messaging.

As mentioned earlier, we can deploy a module as an archive file. The structure of the module archive file is shown in the figure below:



New Deployment Descriptors

The flexibility and extensibility of Axis2 is focused on its deployment descriptors as well, rather than just a configuration file. Axis2 has different configuration files for different levels of configuration. Let's say we want to have different types of configuration for different levels. Having multiple configuration files for different levels will solve the problem for us. There are three types of descriptors or configuration files in Axis2. They are as follows:

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

Global Descriptor (axis2.xml)

As we have already mentioned, all the configurations in Axis2 can be specified using XML descriptors. This gives us more flexibility in extending and changing Axis2. We are not required to go and change the code to have different configurations. The case is similar with core functionality. If you consider the global configuration file of Axis2, it has minimal configuration that is required to run Axis2. It 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 above terms may not be familiar to you, but you need not worry. We will be learning about these most commonly used terms in the forthcoming chapters. Axis2 has `axis2.xml` by default, and it has the minimum configuration that is required to start Axis2. Note that if we make any changes to `axis2.xml`, then we have to restart the system in order to view those changes.

Service Descriptor (services.xml)

As we discussed in the previous section, `axis2.xml` is used to specify the configuration that affects the whole system. However, `services.xml` is used to configure a particular service or a service group. In the next section, we will have a look at some of the ways that are available for deploying services in Axis2. The archive-based deployment mechanism and the directory-based deployment mechanism are the two most commonly used techniques. In either of the two cases, `services.xml` is required for the service to be a valid service. The service descriptor is used to specify a number of configurations. Some of them are optional.

- Name of the service
- Target namespaces of the service
- Session Scope
- Exposed Transports
- Service-level and operation-level parameters

- Message receivers
- Service-level modules
- Operations, exposed operations as well as non-exposed operations
- Bean Mapping
- Object Suppliers
- Service-level policy and Operation-level policy

We will learn more about each of these configurations in Chapter 7. We will also learn to write and create services.

Module Descriptor (module.xml)

As the name implies, `module.xml` is used to configure Axis2 modules. So, it has different types of configurations. These include:

- Handlers and their phase rules
- Module Parameters
- Endpoints
- WS-Policy

We will learn more about Axis2 modules in Chapter 8. We will also learn to write the modules.

Available Deployment Options

In the initial stages of Axis2, we only had archive-based deployment, but later, a number of deployment options made service authors' jobs easier. Axis2 has the following deployment options:

- Archive-based deployment
- Directory-based deployment
- Deploying a service programmatically using archive files
- Programmatically making a Java class into a Web Service
- POJO (Plain Old Java Object) deployment support along with annotation
- Deploying and starting Axis2 in one line

Archive-Based Deployment

The most common and recommended approach for deploying a service in Axis2 is archive-based deployment. In archive-based deployment, we get many configuration options and more flexibility as compared to the other types. In Chapter 7, we will discuss more about archive-based deployment, along with examples.

Directory-Based Deployment

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

Deploying a Service Programmatically

Deploying a service programmatically using an archive file is not really a user requirement; it is rather the module author's requirement, where some module needs to deploy a Web Service at run time in order to provide the full functionality of that particular module.

To create a Service (ServiceGroup) programmatically, we need to have a file object representing the service archive file, and a pointer to an Axis2 runtime or `ConfigurationContext`. Once we have these two, we can create a Web Service as shown in the code below. The advantage of this approach is that we do not need to copy our service archive file into the repository. It is only at the runtime that the service is visible.

```
//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);
```

Here `getConfigurationContext()` is to get the `ConfigurationContext` at any available location. We do not have such a method in Axis2, but we can write one to get the `ConfigurationContext`.

Once we have created a service, the next step is to add the service to the system, and we can do that in the following manner:

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

POJO Deployment

To continue the discussion on other deployment options, we first need to create a Java class, and then we need to expose a service. Assume that we have a Web Service with two methods, "sayHello" and "add". The service Java class will appear as follows:

```
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 while debugging in developing Web Services. In this case, we just need to know about the archive file concept, `services.xml`; all we need is to have a pointer to `AxisConfiguration`. Then, we can make a Web Service by using the above 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 above deployment mechanism can also be considered as POJO deployment, where we make the POJO into a Web Service. However, we can deploy the service as shown above, if and only if we have a pointer to an `AxisConfiguration`.

Therefore, in order to use the above mechanism, we need to have `AxisConfiguration` around, otherwise we cannot use the above mechanism. In the instances where we do not have a way of accessing `AxisConfiguration`, we have to find another way of achieving our goal. Here, another type of POJO deployment mechanism can help us. We can deploy `.class` files into a directory called "pojo", in the Axis2 repository. Then Axis2 will process the `.class` file and make it into a Web Service for us.

Now, let us compile our Java class in order to get the `MyService.class` file. We first need to create a directory inside the repository (the repository directory inside the place where we unpack the Axis2 binary distribution) called "pojo", and it should be on the same level as the services and modules directories. Now, we have the repository structure as follows:

Axis2-1.2

-repository

-services

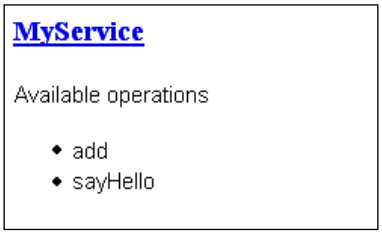
-modules

-pojo

Let's drop the `MyService.class` file into the "pojo" directory. If the server is not running, we need to start Axis2 and type the following URL in the browser and see the result.

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

We will see the following:



This gives us a hint that our service is up and running. Now, let us try to invoke the service and see whether it is working. Since we have not learned about the Axis2 client programming model yet, let us try to invoke the service by using the "REST" method. Type the following URL in the browser and observe the output:

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

We will see the following:

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

This implies that the service has been invoked. Now, let us try to invoke the add method.

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

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

This is exactly the sum of the above two numbers. Now, we are sure that we have exposed and invoked our Java class as a Web Service.

If we want to test this with Axis2 web distribution, then we can do it by copying the `MyService.class` file into:

```
TOMCAT_HOME/webapps/axis2/WEB-INF/pojo
```

Then, we need to follow the above steps and observe the output.

Deploying and Running a Service in One Line

Among all the previous deployment options, this option can be considered the most convenient way of deploying a service and starting the server. We require neither a repository, nor `services.xml`. The only thing we need to have is an Axis2 library file (`axis2-1.3.jar`) and its dependent libraries. Then, we can deploy and start the Axis2 server as shown below. This way of deploying and running the server is very useful when we debug and develop a service.

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

When we start the AxisServer, it will start up the SimpleHttpServer on the port specified in the Axis2 default configuration file, which is port 6060.

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

Now, if we run the following URLs in the browser, then we will get the same result as above.

```
http://localhost:6060/axis2/services/MyService/sayHello?name=Axis2
http://localhost:6060/axis2/services/MyService/add?a=10&b=15
```

Summary

In this chapter, we had a look at the working of Axis2 deployment. We also had a look at the available types of deployment descriptors, and their structures. At the end of the chapter, we learned about the available deployment options in Axis2. The next step would involve creating few more services and seeing the result.

6
Information Model

Service-Oriented Architecture (SOA) has gained a fair amount of recognition in today's information technology industry. However, it is not only the world of computing that is moving its applications onto the SOA platform. By moving into the Web Services field, or by providing functionality in a Web Service-oriented manner, firms gain a number of benefits such as interoperability, flexibility, and extensibility.

When we consider any type of application, there is a set of data associated with that application, which could be either static or dynamic. And the application would require the ability to support processing both these types of data while using a Web Service framework. Currently, as there are a number of Web Service frameworks that can handle data processing in different manners, this requirement can be met. Especially, in Axis2, there exist two types of object hierarchies to support static as well as dynamic data.

Introduction

In Chapter 2, we discussed how Axis2 architecture maintains logic and data separately. Having such a segregation makes the system more flexible and extensible. Moreover, Axis2 also has two different object hierarchies to keep the static data and run-time data separately. In this chapter, we will discuss those two types of object hierarchies with an example showing how we can create and populate them.

Axis2 Static Data

Consider an application that has configuration options. It is obvious that there should be a mechanism to store and persist either the data or the configuration values that will be used to configure the system. There can be some other ways of representing these data at run time. Consider a situation where we do not have a mechanism to store configuration data at run time. Then, we will have to read

the configuration files to retrieve the data, and such a mechanism often results in performance issues. Performance is of major concern when it comes to Web Services. Therefore, loading configuration data at run time from the secondary storage whenever required is not a good option. It would be better to keep such data in the memory, and ready to use whenever required. On the other hand, if we have the entire configuration data in one large object, then that can add to the performance overhead. To address the problem in a more efficient manner, Axis2 has an object hierarchy to store the configuration data. Some of the objects in the hierarchy will be created at deployment time and some at run time, 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.

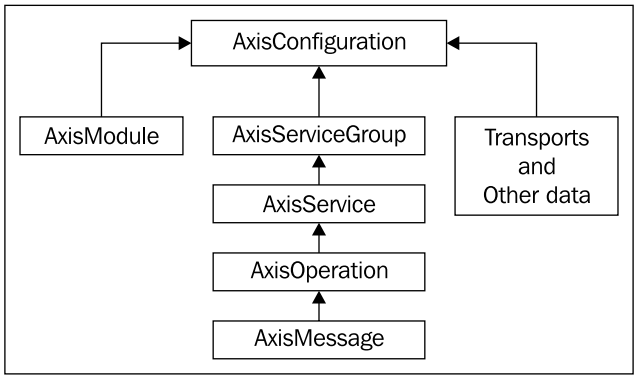
Axis2 has three types of configuration files or deployment descriptors to configure the object hierarchy. The three types of configuration files are as follows:

- 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 known as `axis2.xml` and it contains all the bare minimum configuration data that are needed to start an Axis2 server. Meanwhile, we can edit the `axis2.xml` file to suit our requirements and start Axis2 by using the edited file. We can conclude that a framework will be required for the different types of configuration data, such as the default SOAP version to be used, default HTTP version, and so on. Being a very configurable Web Service framework, the above statement holds true for Axis2. A typical `axis2.xml` file which can be used 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 discussed some of the above terms, and will discuss more about them in detail, in this chapter.



The above figure shows the relationship between various types of descriptions or metadata in Axis2. As we can seen in the above figure, the top-most component in the hierarchy is **AxisConfiguration**, which keeps track of all the configuration data either directly or indirectly. It is observed that there are three major types of objects. Firstly, **AxisModule**, which 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 transports and other data, which are read directly from "`axis2.xml`".

AxisConfiguration

AxisConfiguration is the top-most component of the static data hierarchy. Although we call them static data, there are instances where we can change those data at run time as well. However, it is not something which is done very often. 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 either by using a local file system, a remote repository, or by using a database. We will discuss these options in Chapter 12. In this chapter, we will focus on how to create an AxisConfiguration with the use of a default `axis2.xml` file from the local file system. A typical `axis2.xml` file which has the minimum configuration data to start an Axis2 server is shown below:

```
<axisconfig name="AxisJava2.0">
    <parameter name="name">value</parameter>
    <messageReceivers>
        <messageReceiver mep="MPE"
class="org.apache.axis2.receivers.RawXMLINOnlyMessageReceiver"/>
    </messageReceivers>
```

```
        <messageFormatters>
            <messageFormatter contentType="application/x-www-
                                                    form-urlencoded"
    class="org.apache.axis2.transport.http.XFormURLEncodedFormatter"/>
        </messageFormatters>
        <messageBuilders>
            <messageBuilder contentType="application/xml"0
    class="org.apache.axis2.builder.ApplicationXMLBuilder"/>
        </messageBuilders>
        <transportReceiver name="http"
    class="org.apache.axis2.transport.http.SimpleHTTPServer">
            <parameter name="port">6060</parameter>
        </transportReceiver>
        <transportSender name="http"
    class="org.apache.axis2.transport.http.CommonsHTTPTransportSender">
          <parameter name="PROTOCOL">HTTP/1.1</parameter>
          <parameter name="Transfer-Encoding">chunked</parameter>
        </transportSender>
        <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>
    </axisconfig>
```

Parameters

As you can see opposite, the `axis2.xml` file has parameters, and they can be defined at different levels as well. Here, we have parameters at the top-level as well as inside transports. The main use of a parameter is to configure the system and to provide the configuration data required at run time. For example, if we need to log some request to a particular location, then that location can be provided with the use of a parameter. Parameters are designed to store primitive data types (string, int, double, OMelement, and so on), but not any other type of objects. Even though storing object types inside a parameter is not invalid, it is not a correct method to follow when it comes to clustered applications (due to serialization issues).

Each parameter has an optional attribute called "locked" as shown below:

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

The idea of a "locked" attribute is to provide a control mechanism in order to make sure that none of the child nodes overrides that parameter. Let us say, we have a parameter as the immediate child of `axis2.xml`, as shown below:

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

This will ensure that we have a unique parameter named "port", as in the code below. The case would be similar for any child node. Apart from that, we cannot have the parameter with the name "port" in either of the two descriptors (`services.xml` and `module.xml`).

```
<axisconfig name="AxisJava2.0">
    <parameter name="port" locked="true">6060</parameter>
    <transportReceiver name="http"
class="org.apache.axis2.transport.http.SimpleHTTPServer">
    <parameter name="port">6060</parameter>
  </transportReceiver>
</axisconfig>
```

There is a scope associated with the parameter as well. A parameter defined in `axis2.xml` as an immediate child can be accessed by any of the descriptors in the system. Whereas, if we define a parameter inside a child node, for example transportSender, then that parameter can be accessed only inside that particular child, in this case, that specific transportSender.

While accessing a parameter, Axis2 first checks whether the parameter is defined in the current description. If not, it checks its immediate parent for the parameter. If found, the parameter is returned, otherwise, the parent's parent is checked, and so on. In this manner, it searches the hierarchy when a request for a parameter is made.

MessageFormatters and MessageBuilders

We know that the content-type header is used to specify the type of data in the message body, and depending on the content type, the wire format varies. Therefore, we need to have a mechanism to format the message depending on content type. We 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 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, but for it to go though Axis2, an AXIOM element needs to be created. As a result, MessageBuilders are employed to construct the message depending on the content type.

These descriptions can be considered to be complex and since it is not likely we will want to change them, we can have `axis2.xml`, since it has configured all commonly used content types along with their corresponding Builders and Formatters.

A simple example of this is JSON. We can send a JSON message by setting the necessary content type. Axis2 will then identify that, and use the JSON Builder to build the message. As a result, we will get the AXIOM element from the JSON message. In the same way, when we want to send out Axis2, we can serialize the AXIOM element as the JSON message.

TransportReceiver and TransportSender

Axis2 is said be transport-independent, and hence theoretical. We can communicate with Axis2 by using any of the given transports.

For example, Axis2 has inbuilt support for HTTP, TCP, SMTP, and JMS. These can be easily configured.

The Transport Sender helps to serialize (AXIOM -->XML) and handle the message exchanges depending on the underlying protocol. Whereas, the Transport Receiver deserializes (XML-->AXIOM) an input stream into AXIOM and responds to the client according to the protocol. We do not need to bother about configurations, as the default `axis2.xml` file comes with a set of transports along with their default configurations.

Flows and PhaseOrder

In Chapter 4, we discussed the use of Flows and Phase orders when we were discussing the execution chains of Axis2. Even for these, Axis2 comes with a default configuration, and most of the time, we do not have to make changes. However, if it is necessary to change the configurations, then we can refer to Chapter 4.

So far, we have discussed the different types of configuration data that come from `axis2.xml`. Now, we will have a look at the other types of descriptions.

AxisModule

In simple terms, an AxisModule is a run-time representation of `module.xml`. So, the configuration data that is found in `module.xml` is also present in AxisModule. A module configuration file or `module.xml` contains the following types of 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 with the help of the data from `module.xml`. And at run time, any of this data can be retrieved via the same AxisModule. The parent description of the AxisModule is AxisConfiguration.

Service Description Hierarchy

We know that the AxisService hierarchy is created either by using a `services.xml` file or the service descriptor. This hierarchy contains four types of descriptions. When we deploy a service into Axis2, an object hierarchy will be created, and then it will be added to AxisConfiguration. Therefore, unless the services are deployed in Axis2, we will not have the service objects hierarchy in the AxisConfiguration. Unlike AxisModules and other descriptions (Transports, Message Formatters, and so on), the service description hierarchy is likely to get changed at run time depending on the deployment options. A typical `services.xml` file is shown below, so that we can discuss this object hierarchy in a more specific manner.

```
<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 it is the child of AxisConfiguration. AxisServiceGroup can be considered the parent of a set of AxisServices which are defined in `services.xml`. Once we define a parameter in AxisServiceGroup, it can be accessed either from AxisService, AxisOperation, or AxisMessage. In addition to the parameters, an AxisServiceGroup may contain a collection of modules engaged to this particular AxisServiceGroup.

AxisService

An AxisServiceGroup should contain one or more AxisServices as children. Therefore, any of the configurations (such as parameters) defined in AxisServiceGroup or AxisConfiguration are easily accessible within an AxisService.

The following data is found within AxisService:

- AxisOperations
- Parameters
- Engaged modules
- Namespaces
- Exposed transports
- Description about the service
- Message Receivers
- WS-Policy

We will be discussing each of the above topics in detail in the next chapter. In the next chapter, we will also look at `services.xml`.

AxisOperation

AxisOperation is the run-time description representation of an exposed Web Service operation. Let us 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, or AxisService can be accessed and used inside AxisOperation. In addition to these parameters, AxisOperation contains the following:

- AxisMessages
- Engaged modules

- Operation name
- SAOP actions
- WS-Policy

AxisMessage

AxisMessage is the bottom element of the service hierarchy, and its parent is an AxisOperation. Unlike AxisService, an AxisOperation does not have a set of AxisMessages. The number of AxisMessages in an AxisOperation is based on the MEP of that particular AxisOperation. For example, if the operation is "in-out", then it will have only two AxisMessages, one to represent the in-message configuration, and the other 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)

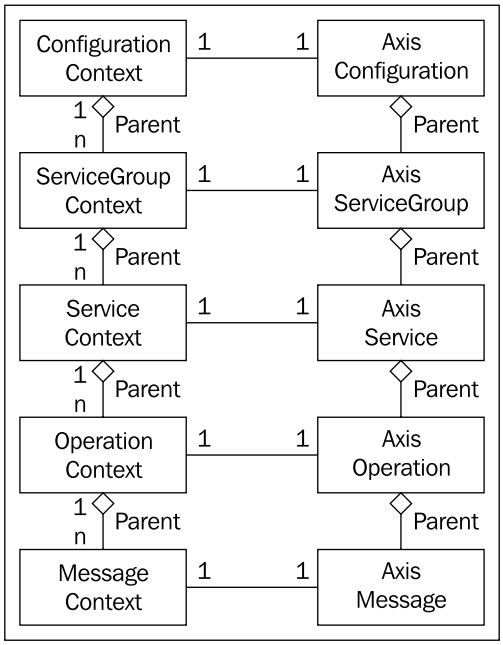
Having message elements inside the operation element of `services.xml` is optional. You need to add them only if you have to override the default.

Now we have a good understanding of the most commonly and publicly available descriptions, that is, static data, in Axis2. Now, it's time to have a look at the other type of data in Axis2, namely run-time data. We will also have a look at the relationship between static data and run-time data. The following figure (overleaf) shows the relationship between the description hierarchy, its contexts and the run-time data hierarchy.

Axis2 Contexts

The Axis2 Context hierarchy is the run-time data representation of Axis2. The run-time data comes into the picture only when Axis2 receives a message. The run-time data is used to share data across multiple invocations, or among handlers in one of the invocations. When we were discussing the description hierarchy, we looked at how parameters act as the main configuration mechanism. In contexts, the main configuration or data-sharing mechanism is the use of properties. Unlike parameters, we do not need to define properties anywhere, since we can easily create and use them. The properties are stored as name-value pairs in the context hierarchy, which means that if we add a property to one of the contexts, then that property can be accessed as well as overridden by any of its child contexts.

The figure below shows the relationship between static data and run-time data in Axis2.



From the above figure, we can observe that the top-most component of the hierarchy is **ConfigurationContext**. The only difference in ConfigurationContext when compared to the other contexts is that it is the only context that exists in the system before a message is received. ConfigurationContext has a reference to an AxisConfiguration, and to create ConfigurationContext, it is required to have an AxisConfiguration available.

ConfigurationContext

A ConfigurationContext is a run-time representation of the whole system. To start Axis2, we need to have a ConfigurationContext. The lifetime of the ConfigurationContext will be the lifetime of the system. So, if we store a state (property), it will last forever, that is until the system shuts down. ConfigurationContext can be considered not only the parent of all the other contexts, but also the original element of the entire Axis2 system.

ServiceGroupContext

When a message is received by Axis2, the ServiceGroupContext is created to store and share data across services. To create a ServiceGroupContext, it is required to have an AxisServiceGroup object available. An AxisServiceGroup may have one or many ServiceGroupContexts. However, there is only one AxisServiceGroup associated with the context. The lifetime of the ServiceGroupContext depends on the service scope. If the service scope is "application", then the lifetime will be similar to the system's 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 any of the ServiceGroups at run time, a ServiceContext helps to provide the same.

ServiceContext

A ServiceContext represents the run-time 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 service scope. If we want to share data across multiple invocations of the same service, then ServiceContext can be used to store that data.

Let us say, we have a service with three operations: "login" ,"doSomething", and "logout". Now, assume that we need to share data across these three operations. In such a case, the ServiceContext can be used. The number of ServiceContexts in a ServiceGroupContext depends 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 OperationContext is shorter than that of the ServiceContext. 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 an OperationContext can be used. The number of OperationContexts in a ServiceContext is not related to the number of AxisOperations in an AxisService. It is dependent on the number of method invocations in a given service.

MessageContext

When a message is received by any transport, the transport creates a MessageContext to represent the incoming message. To create a MessageContext, we need to have the ConfigurationContext available. The lifetime of the MessageContext is similar to 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 leaves the message receiver.

Although we can see a hierarchy, 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 have discussed the run-time data hierarchy and the static data hierarchy. We discussed how and when these are created. Most of the things we discuss here are important if you plan to create a rather complex service such as a session-aware service, or you plan to write handlers. If you are going to use Axis2 to simply deploy and invoke a service, then you do not need to worry too much about the facts discussed here.

7
Writing an Axis2 Service

Axis2 involves a number of ways in which a Web Service can be deployed. We have already discussed some of them briefly in Chapter 5. The main objective of this chapter is to learn how to write and deploy a Web Service as a service archive file. Here, we cover the two fundamental approaches that are commonly used in the industry to create a Web Service. At the end of this chapter, we will have a very good understanding of creating a Web Service and deploying it in Axis2.

Introduction

When it comes to creating a Web Service, we have two commonly used techniques. Each of the techniques has its own advantages, as well as disadvantages. These commonly used techniques can be classified as follows:

- Code-first approach
- Contract-first approach

In the code-first approach, we start Web Service development from code, that is, we first discuss and decide what to do, and later implement that. In simple words, we first write the service implementation class, and then create the Web Service based on that.

However, in the second approach, we first write the Web Service Description Language document (WSDL), and then we create (or generate) the service class, by using WSDL. This approach is called the contract-first approach, since WSDL is assumed to be a contract between two or more parties.

When we consider both these approaches together, we can consider their pros and cons. In the first approach, we write the service implementation class and other relevant classes first. When we do so, we do not need to understand the Web Service concepts. What we do is write our service implementation class with a set of public methods that we are planning to expose as Web Service operations. This way, one can deploy a service without having any knowledge of WSDL and SOAP.

This approach is also called the POJO (Plain Old Java Object) approach. Another advantage here is that you can change your service class as and when you wish to, since you do not need to worry too much about method parameters, methods names, and so on. So the key objective is to meet the customers' service requirement in a very convenient manner.

However, in the contract-first approach, normally there is an agreement between two or more parties, and they come up with one or more contracts among them. And those contracts might be based on the underline protocols, security considerations, and some other factors such as policies. Here, the contract is written using WSDL. By using the contract WSDL, one can create a client while the other can create a service. We can find tools in any Web Service framework to create both the client-side as well as server-side code by using the given WSDL. The approach of creating service and client code using WSDL is called code generation. Further, the generated code are called data-bound classes. The advantage here is that we can use the tools available in the distribution to create a service skeleton, or the service proxy (stub), and we can fill the skeleton. The disadvantage is that, if we need to change the WSDL, then we have to generate the code again. But in the code-first approach, changing the service class is very simple. However, once two parties have agreed upon a set of policies, and have come up with the WSDL, then it is not likely to be changed.

If we are new to the Web Services area, then the best thing is to start with the POJO approach, since that is just writing a set of lines of Java code. We can expose the Java class as a Web Service and consume it universally. So, let us discuss the code first, and see how we can write an Axis2-compatible Web Service using a Java class.

Code-First Approach

As we have discussed, in the code first approach, we start by writing the service implementation class or the class that provides the service. So, first, we will write a simple Web Service, which says hello, when the service is invoked. And we will further assume that we need to give our name as the input parameter to the Service operation.

Single-Class POJO Approach

So in this case, let us assume that our service class does not have any package name and will appear as follows:

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

As we discussed in the Axis2 deployment model, we have a number of options to deploy the service as well. Let us start with the simplest approach to deploy and invoke the service:

Step 1: Compile the Java class, then find the **HelloWorld.class** file.

Step 2: Go to **<TOMCAT_HOME> | webapps | axis2 | WEB-INF**

Step 3: Create a directory called '**pojo**' inside the WEB-INF directory.

Step 4: Then, copy the **HelloWorld.class** file into the pojo directory.

Step 5: If Tomcat is not running, then start it.

Step 6: Go to `http://localhost:8080/axis2/services/listServices`, where we can find a service called HelloWorld.

Step 7: Now type `http://localhost:8080/axis2/services/HelloWorld/sayHello?name="Axis2"` there. We will be able to see '**Hello Axis2**' in the browser. Let us do the same thing again changing the name parameter, which will give us the expected output.

What this simply means is that we have successfully deployed our service and we have invoked the service in the REST manner.

This is the easiest way to write and deploy services in Axis2. Though the sample we have chosen is a very simple one, we can write any kind of complex services in this manner. However, the only limitation is that we cannot have a package name in the service class. If we want to have the package name, then we have to follow some other alternative, which we will be discussing later in this chapter.

In the case of Axis2 POJO support, we can deploy with an annotated Java class on a non-annotated (plain) Java class. It should be mentioned here that Axis2 has support for JSR 181 annotation. For those who are not familiar with annotation, it is a mechanism to provide metadata while using POJO. A simple Java class with a basic JSR 181 annotation can be written as follows. It should be noted that we can provide much data and can do very complex applications with annotation.

```
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;
    }
}
```

This type of annotation is only supported in JDK 1.5 or above. Even if you do not know much about annotation, there is no need to worry. First, let's try to learn the Axis2 concept. Then applying annotation will be easier.

POJO with Class Having Package Name

As we have discussed earlier, when we deploy POJO as '`.class`' files, we cannot have a package name in the Java class. Another issue is to have a POJO with more than one class, in which case we cannot use the single class POJO approach. Now, let us say our class looks like this:

```
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;
    }
}
```

Please note here that we have annotated the Service class, and the Address class will look like this, which is simply a Java bean.

```
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 what we have to do is to compile the source code and create a Java Archive file (`.jar`) file. Next, we need to edit our `axis2.xml` file to add a deployer to handle the `.jar` file. We can do that by adding following entry to the `axis2.xml` file in **<TOMCAT_HOME> | webapps | axis2 | WEB-INF | conf | axis2.xml**

```
<deployer extension=".jar" directory="pojo" class="org.apache.
axis2.deployment.POJODeployer"/>
```

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

We need to keep in mind that when we make any configuration changes in `axis2.xml`, we have to restart the system to apply the changes.

As we know, a `.jar` file may contain one or more `.class` files. 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`. Once we have that, Axis2 will identify it and expose it as a Web Service.

To see what has happened, if Tomcat is running then let's restart and go to `http://localhost:8080/axis2/services/listServices`, where we can find a service called `AddressService`.

If we have turned on hot deployment, then we can change either the `.class` files or the `.jar` file and redeploy them. Axis2 will implement the changes we made.

Deploying a Service Using a Service Archive File

The POJO approach cannot be considered the most suitable approach, or the most flexible approach, when compared with the different options available in the code-first approach. Even Axis2 does not recommend the POJO deployment approach. The recommended one is the service archive-based approach. We can use the POJO approach during the initial stages of service development, as it is very convenient. Now, let us see how we can create a service archive file from the `HelloWorld` Java class.

Writing the services.xml File

In order for it to be a valid service in Axis2, you should have the `services.xml` file inside the service archive file, while deploying a service as an archive file. The `services.xml` file is nothing but the deployment descriptor for the service that you are trying to deploy. The service deployment descriptor tells the deployment module how to configure and deploy the service. Writing `services.xml` for the service mentioned earlier is very simple and straightforward. There are few things that we have to keep in mind while writing a `services.xml` file:

- The fully qualified class name of the service implementation class
- The MessageReceiver or receivers that we are going to use

Axis2 has a set of built-in MessageReceivers. Some of them can only handle the XML-In and XML-Out scenario, and these are called RawXML MessageReceivers. There are also message receivers that can handle any kind of JavaBeans + simple Java types + XML, and they are called RPC MessageReceivers. From the earlier sample code, it is obvious that we can use none of the RawXML MessageReceivers for this particular service. So the simple answer is to use the RPC MessageReceivers.

There are different ways of writing `services.xml`, and they vary depending on the way we specify the message receivers, operation overriding, and so on. Let's start with a very basic `services.xml` file to understand the concept easily. The `services.xml` file corresponding to our 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, we have highlighted a few lines in the above `services.xml` file. They are the important XML tags that you have to remember while writing `services.xml`.

Service Implementation Class

To specify the corresponding service implementation class, we 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. As you can see, in this case it is `HelloWorld`. If we have a package name in the service class, then the parameter would be a fully qualified name.

Specifying the Message Receiver

There are few ways to specify the MessageReceiver for a given service. One way is to add the operation tag with the actual Java method name, and include the message receiver inside the operation tag. The `services.xml` file shown opposite 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, and inside the operation element, it has added the message receiver element.

Creating a Service Archive File

The next step is to create a service archive file containing the compiled code of the service implementation class, and the `services.xml` file. The internal folder structure of the service archive file looks like this:

```
HelloWorld.aar
    META-INF
        Services.xml
    HelloWorld.class
```

Deploying the service is just a matter of dropping the service archive file into the services directory in our Axis2 server repository. If your server is Tomcat, then we can easily upload the service using the Axis2 web administration console as well. In this case, the name of the service will be "**HelloService**".

Different Ways of Specifying Message Receivers

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

- Specify message receivers at the operation level for each operation.
- Specify message receivers at the service level for the whole service.
- Specify service-level message receivers and override them with operations, as and when required.

Specify Message Receivers at the Operation Level

The example we have discussed used the first approach where it specified the message receiver at the operation level.

Specify Message Receivers at the Service Level for the Whole Service

Think about the scenario where we have a number of operations to be published in `services.xml`. In such a case, adding the message receiver for each and every operation seems troublesome. If we can specify the message receiver for the whole service, then it will make the job of the service author easy, and will simplify the `services.xml` as well.

Axis2 has built-in support for all the eight MEPs (Message Exchange Patterns) defined in WSDL 2.0. In `services.xml`, we can specify the MEP and the corresponding message receiver. Depending on the MEP that the operation belongs to, Axis2 picks up the message receiver automatically, and sets the selected message receiver for the operation.

Inside the operation tag in `services.xml`, we 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" />
```

Defining the service-level message receivers for a given service is shown below:

```
<service>
  <description>This is my first service , which say hello</description>

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

According to `services.xml`, the `RPCMessageReceiver` is the message receiver for all the in-out operations (any operation that belongs to the in-out MEP will assign this message receiver as its own message receiver) in the service. The service-level message receiver for the in-only MEP is `RPCInOnlyMessageReceiver`. If we re-deploy the `HelloWorld` service with the new `services.xml`, we will definitely get the same result, if we invoke the service again.

Specify Service-Level Message Receivers and Override Them with Operations

There may be instances when the service author wants to use a different message receiver for one or two operations, when he or she defines the service-level message receivers. Overriding the service-level message receiver by operation can be easily achieved by just adding the message receiver element to the operation for which you want to override. A sample `services.xml`, which follows this technique, is shown below:

```
<service>
  <description>This is my first service , which say hello</description>
  <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">HelloWorld</parameter>

  <operation name="sayHello">
    <messageReceiver class="org.apache.axis2.receivers.RawXMLINOutMessageReceiver" />
  </operation>

</service>
```

Operation `sayHello` uses a different message receiver as compared to its service-level message receivers.

All the public methods in the service implementation class are exposed whether we have specified that in `services.xml` or not. Axis2 calculates the MEP of an operation by checking its corresponding Java method. If the method is void, then the MEP will be in-only, else it will be in-out, depending on which, the right MEP message receiver will be set.

Service Group and Single Service

There may be many instances where we want to deploy multiple services (which may be logically related or not) together in a single service archive file. To do that, Axis2 has the concept of a `ServiceGroup`. Here, we can have multiple service implementation classes and only one `services.xml` file to describe all the services. The only difference here is that the root element of the `services.xml` is changed to serviceGroup instead of service. As an example, say we want to deploy two services together in a single service archive file and further assume that their names are `MyService1` and `MyService2` respectively. The `services.xml` file can then be written as follows:

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

The only difference in the service element, as compared to HelloWorld's `services.xml`, is that the service element has an additional attribute called name. If we want to have multiple service elements in the `services.xml` file, then it is must have the name attribute in each and every service element.

Adding Third-Party Resources

There may be many instances where we want to use a third-party library in our Web Service. We already know that each service and module in Axis2 is isolated (meaning each service and operation gets its own class loader). So, when we want a third-party library in our Web Service, Axis2 has a mechanism to do that, which is to simply create a `lib` folder inside the service archive file, and drop the library or resource inside that. Assume that we have a service, and we need to use `foo.jar` and `bar.jar`, and `xyz.properties` in our service, then our service archive file will look like this:

```
MyService.aar
  META-INF
      services.xml
   lib
     foo.jar
     bar.jar
      xyz.properties
```

Service WSDL and Schemas

When we write complex applications, we will find the need to have WSDL files and Schema files inside the service archive file. When we consider an enterprise-level application, it cannot let the client rely on auto-generated WSDL. Then, we need to have a solid and well-defined WSDL file. Therefore, we need to deploy our service along with the WSDL file and the corresponding XML Schema files as well. Therefore, when we want to add WSDL files, we can add them to the META-INF directory. The only thing we have to remember when we do this is that the service name in the `services.xml` file and the WSDL service name should be the same. The service archive file with the WSDL file will look like this:

MyService.aar

META-INF

services.xml

myservice.wsdl

schema.xsd

So when we have the WSDL file and `services.xml`, then the relationship will look like this:

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 WSDL

The easiest and the most suitable way to create a service is to start from WSDL, and that is what happens in most enterprise-level applications. When it comes to enterprise-level applications, they have business scenarios and a corresponding business contract or the WSDL file so why not start from there? The interesting thing here is that both the client and the service provider are given the same WSDL and they have to act according to that.

Axis2 has built-in support for generating service code. Once we have the WSDL, so in this case as a service author we have only to do the following few steps:

- Generate the service code (service skeleton).
- Fill the service skeleton according to the business agreement.
- Run the generated ant build file.
- Deploy the ant-created service archive file into our application server where Axis2 is running.

Generating Code

Axis2 comes with a set of tools and IDE plug-ins for code generation (WSDL2Code) in order to make the work easier. So, we can choose any kind of code generation tool to generate the service skeleton. In the meantime, there are a set of data-binding frameworks supported, so you can select one of them as your data-binding. As an example, you can select xmlbeans, adb, or any other available data-binding framework (jibx, jaxme, and so on). Once we generate the server-side code, it generates:

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

Filling the Service Skeleton

Axis2 generates the service skeleton class to throw UnSupportOperation from each method. So, what we have to do now is to implement the service skeleton class, as we wish.

Running the Ant Build File

After completing the service skeleton, the next step is to create the service archive file using generated code. To make the job simpler, Axis2 generates the ant build file to create the service archive file for us. So what we have to do is to open the command-line console, and go to the folder where we generate the code, and then type Ant `build` in the console to run the ant build file, which then creates the service archive file for us.

In this chapter, we have only discussed the contract-first approach. We will learn more about code generation and data-binding in detail, later in this book. There, we will cover a number of examples as well as available tools.

Deploying the Ant-Created Service Archive File

Once we create the archive file, deploying it is just a matter of copying the file into `repository/service directory`. If we use WAR-based deployment mechanisms, then we can upload the service.

Summary

Making a Java class into a Web Service is very straightforward in Axis2. Once we know how to write `services.xml` correctly, then we can do more complex applications than just POJOs. And deploying a service is just a matter of creating the service archive file and dropping it into the services directory in the repository. The WSDL-first approach is the easiest way of creating a service, since Axis2 has built-in support for code-generation and also a set of tools to make the job easier.

8
Writing an Axis2 Module

Web Services are becoming popular and so everyone is moving into the Web Services area, as a result of its various advantages. Therefore, when applications are converted into Web Services, a number of requirements come into the picture. In such instances, just having a Web Service framework is not enough; the quality of service also forms an essential part. Therefore, in order to be a useful Web Service processing framework, it is essential to have qualty-of-service support as well. Quality of services is nothing but Web Service enhancement, such as security, reliability, transaction support, and so on. The Axis2 architecture is such that we can extend the system to have quality-of-services support with little effect; that is achieved by using the concept of modules.

Introduction

Looking back at the history of Apache Web Services, the stack Handler concept can be considered one of the very useful ideas. Therefore, Axis2 has incorporated the handler concept into its architecture. However, the way in which we deploy handlers in Axis2 and Axis1 is quite different. In Axis1, we need to make some global configuration changes in order to add a handler. But when it comes to Axis2, we can add a handler very easily without making much change to the global configuration files.

At the design stage of Axis2, one of the key considerations was to have a mechanism to extend the core functionality with less effect. During that time, WS-Reliable implementation for Axis1 was doing a large amount of work to implement Reliable Messaging. This was the major reason for this consideration. Therefore, as a result of learning a lesson from Axis1, Axis2 introduced a very convenient and a very flexible way of extending the core functionality, apart from providing quality of services. This particular mechanism is known as the module concept.

Module Concept

In Chapter 4, we discussed what a handler is all about, and its various uses. The main purpose of handlers is to intercept the message flow along with some processing. A module is nothing but a collection of one or more handlers along with a number of configuration files and other resources. We can look at a module as an implementation of a Web Service specification. As an illustration, Apache Sandesha is an implementation of WS-RM specification, whereas Apache Rampart is an implementation of WS-security. Similarly, a general module is an implementation of a Web Service specification. On the other hand, one can also write a module to do custom processing . For example, one can write a module to log all the incoming messages or to count the number of requests.

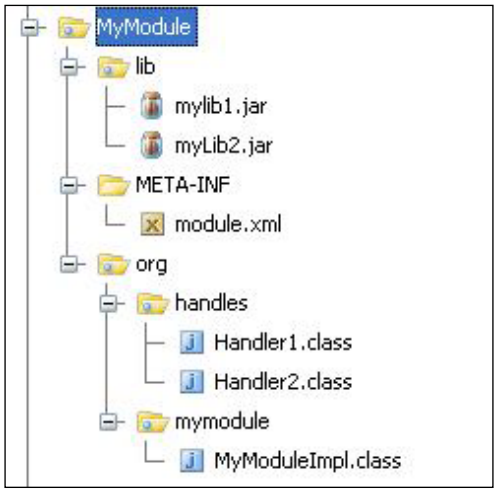
Module Structure

One of the issues we have in Axis1, when we want to implement a new specification, is the amount of work. Let's say we want to implement a new specification with a set of handlers and configurations. The amount of work involved to get the job done would be very high. The problem becomes more difficult when we want to have third-party libraries for our new specification support. To overcome this issue, the Axis2 module concept and its structure can be considered a good option. As we discussed in the section on deployment, both Axis2 services and modules can be deployed as archive files. Inside any archive file, we can have configuration files, resources, and the module required by the author.

One thing to note here is that, though we can deploy services when the system is up and running, we cannot deploy modules at run time. (We can drop them into the directory but Axis2 will not recognize them, so there would be no hot deployment or hot update). The main reason behind this is that, unlike services, modules tend to change the system configurations and making system changes at run time to an enterprise-level application cannot be considered advisable.

As we have discussed earlier, adding a handler to Axis1 involves global configuration changes and obviously a system restart. However, when it comes to Axis2, we can add handlers using modules, without making any global-level changes (there are instances where we do have to make global configuration changes, but they are not all that frequent). Also, you can change the handler chain at run time without shutting down the system (I mentioned earlier that changing the handler chain or any global configuration at run time cannot be considered advisable; but at the testing stage, we can do that).

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 services archive file to be a valid one, it must have a `services.xml` file; in the same way a module archive has to have a `module.xml` file inside the META-INF directory of the archive in order to be a valid one. A typical module archive file takes the structure as shown in the figure below. We will discuss each of the items in detail and create our own module.



Module Configuration File (module.xml)

We know that a module archive file is a self-contained package. So, it has to have all the configurations required to be a valid and useful module, and this is simply the beauty of a self-contained package. The module configuration file or `module.xml` file is a configuration file that Axis2 can understand well and perform the necessary work.

A simple `module.xml` file has only one handler but when it comes to complex modules, we can have some other configurations in `module.xml`. First, let's look at the types of configurations available in `module.xml` with an example module, which counts all the incoming and outgoing messages, under these headings:

- Handlers and Phase Rules
- Parameters
- Module implementation class
- WS-Policy
- End points

Handlers and Phase Rules

As we have discussed earlier, a module is a collection of handlers. Irrespective of the number of handlers in a module, the `module.xml` file provides a convenient way to specify the handlers. The most important fact is that `module.xml` can be used to provide enough configuration options to add a handler into the system at the exact location where the module author wants to see the handler running. In Chapter 4, we learned about phase rules being a mechanism to direct Axis2 to put handlers into a particular location in the execution chain. Let's have a look at them with the help of some examples.

Before learning how to write phase rules and specifying handlers in `module.xml`, let's look as to how a handler is written. The following are the two ways of writing a handler in Axis2:

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

Since we are going to make a very simple application, we do not have to worry much about the Handler API. We can make our job easier by extending the `AbstractHandler` class. When we extend the abstract class, we only need to implement one method called "invoke". The code below illustrates 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;
    }
}
```

From the preceding code, it is clear that `messageContext` is taken as a method parameter, and `InvocationResponse` is taken as response. If we look at the implementation of the method it does the following:

- Gets the configurationContext from the messageContext
- Gets the property value specified by the property name
- Then increments the value by one
- Next, sets it back to configurationContext

As a module author we have to do all the logic processing inside the invoke method, and on the basis of its result, we can decide whether AxisEngine should be continued, suspended, or aborted. This can be done by returning the corresponding return value. The available return types can be classified as below:

- `InvocationResponse.CONTINUE`: Gives the signal to continue the message.
- `InvocationResponse.SUSPEND`: Message cannot continue since some of the conditions are not yet satisfied. So we have to pause the execution, and wait.
- `InvocationResponse.ABORT`: Something has gone wrong. We have to drop the message, and let the initiator know about it.

The corresponding CounterConstants class is a collection of constants, which appears 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 know, the sample module that we are going to create is to count the number of requests coming into the system and 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. Refer to the code below.

```
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 for the incoming handler processing, except for the property name, which is used in two places.

Parameters

Adding a parameter here is the same as adding a parameter in the `services.xml` or `axis2.xml` files. We just have to add the following tag to `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` file, and when we want to access the parameter, we can do that by following these steps:

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

Module Implementation Class

When we consider an enterprise-level application, it is obvious that we have to initialize a number of things such as database connections, reading property files, starting up threads, and so on. Therefore, a place was required wherein we could put the logic in our module. We know that handlers run only when a request comes into the system, and not at the system initialization time. The module implementation class provides a way to achieve system initialization logic as well as system shut-down time processing. As we have mentioned above, the module implementation class is optional. If we take the Axis2 default addressing module, it does not have a module implementation class. For understanding purposes, we will write a module implementation class as shown:

```
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());
        }
    }
}
```

From this, we can see that there are several methods in the implementation class, but not all of them are in the module interface. We have the following methods for supporting our counter module-related stuff:

- `Init`
- `engageNotify`
- `applyPolicy`
- `shutdown`

At the system start-up time, the `init` method is called, and at that time, the module can do various initialization stuff. In our sample module, we have initialized both the in-counter and the out-counter.

When we engage this particular module to the whole system, a service, or to an operation, the `engageNotify` method is called. And at that time, the module can decide whether it can allow this engagement or not. For example, we try to engage security module to a service and at that time, the module finds that there is a conflict in encryption algorithm. In such a case, the module will not be able to engage and will throw an exception. As a result, Axis2 will not engage the module. In this sample, we will not be doing anything inside the `engageNotify` method.

Now, we know that WS-Policy plays a major role in Web Service technology, and everyone uses WS-Policy as a mechanism to configure the quality of services. In addition, the WSDL file should expose all the policies to the end users as well. As a result, the end user knows what he or she needs to do. When we engage a particular module to a service and view the WSDL of that particular service, then the module policy should become visible. So, when we call the `applyPolicy` method, the corresponding module will apply its policy to the service or the operation. When we engage a module, Axis2 automatically calls the `applyPolicy` method. At that time, the module will apply its policy to the corresponding service or to the module. In this particular sample, we do not have any policy associated with the module, so we do not have to worry about this method as well.

The `shutdown` method is called when the system shuts down. So if we want to do any kind of processing at that time, we can add that logic into that particular method. In our sample for demonstratation purposes, we have added code to store the counter values in a file.

WS-Policy

Specifying a WS-Policy element in a `module.xml` file is one way of configuring a module to add WS-Policy. 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>
.....................
                  </sandesha2:RMAssertion>
</wsp:Policy>
```

Endpoints

In Axis2, an endpoint is an operation of a Web Service. So adding an endpoint is similar to adding an operation. Why, then, do we need to add an endpoint from a module? Let's say we have a module , and that module has a set of control operations. The most suitable example is reliable messaging, since it has a number of control messages. Say, we need to invoke a service in a reliable manner; then, we first have to set up a sequence with the service. To do that it will send the control message called "createSequence" to the service we need to access. But we know that our service does not have the createSequence 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; that is, when we engage the module, it adds a method called "createSequence" to the service at the time we engage the module (or to all the services, if we engage it to the whole system). Then when a request comes, Axis2 will dispatch without having any problem, and those operations have their, or that endpoint has its, own message receiver for the purpose of method invocation.

So it is obvious that when a module needs the exchange of control operations, then endpoints need to be added to represent those operations. Adding an endpoint is very simple. All that we need to do is to add an operation element or elements with message receiver and a set of action mappings. To get an idea about that, let's take the Sandesha module as a reference. Its `module.xml` file has the following operation tag to add the control operations. Remember when we have the operation tag in the `module.xml` file, Axis2 will do all the processing, including creating AxisOperation and adding it. As a module author, what we need to do is just to 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 from it and find out more about writing a `module.xml` file.

Writing the module.xml File

So far, we have written two handlers. Now, the remaining thing to be done is to write the module descriptor file. While writing the `module.xml` file we have to use phase rules to specify the location of handlers (we have discussed phase rules before and it is now time to refresh our minds about the phase rule). The simplest `module.xml` file for our module is given below:

```
<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>
```

According to the `module.xml` file, we can identify that the description of the module, is "Counts the incoming and outgoing messages". Moreover, it has specified two handlers 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 will find those two handlers in the inFlow. Meanwhile, the outgoing message counter is added to the MessageOut phase, in spite of the fact that it does not specify where exactly the handler needs to be placed in the phase.

If you look carefully, you will observe that in the root element, there is an attribute called "class", which specifies the module interface class. And we have to remember that this attribute is an optional one. Modules may or may not have this attribute.

Deploying and Engaging the Module

Now, we have written everything that is required for a valid module. The only thing remaining to be done is to create the module archive file and deploy it to the repository. First we compile our source code, which we know creates `.class` files.

Assuming `org.apache.axis2.sample.module.request` to be 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 will appear as follows:

```
classes
   META-INF
      module.xml
   org
     apache
         axis2
             sample
                  module
                      request
                           CounterConstants.class
                           CounterModule.class
                           IncomingCounterHandler.class
                           OutgoingCounterHandler.class
```

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

Deploying the module is just a matter of copying the file into `TOMCAT_HOME/webapps/axis2/WEB-INF/modules` or `repository/modules` directory. In this case, let us focus on deploying the module in Tomcat or on our favorite web application server. As we know, Axis2 does not support module hot deployment, so just dropping it won't make it an Axis2 module. What we need to do is to restart the Tomcat (or other Axis2 server), and then it will deploy the module.

It should be noted that just deploying a module does not add its handlers into the handler chain. To add handlers into the system, we need to engage the module. So, now it's time to see how we can engage the module. For that, let us use the Axis2 web administration console. In this case, let us try to engage the module to all the services in the system using the web administration console (we can also engage a single service using the administration console). In order to engage the module to the system, follow the steps given below:

1. Go to `http://localhost:8080/axis2/`.
2. Click on the **Administration** tab. Then it will open up a new page that asks for username and password.
3. Type "admin" as username and "axis2" as password, then it will open up the administration console.
4. Then click on the **Available Modules** in the left-hand side navigation menu, where we can find our Counter module as:

   counterModule: This counts the incoming and outgoing messages.

5. Now go to **Engage Module | For all Services**, which will open up a new page with a drop-down menu.
6. Select **counterModule** from the drop-down menu, and click **engage**.
7. Now, you will see the module "`counterModule`" engaged successfully.
8. Then go to **Global Chains**. You will be able to see the **IncomingMessageCountHandler** handler in the transport phase between **RequestURIBasedDispatcher** and **SOAPActionBasedDispatcher**.

This simply tells us that we have engaged the module successfully and have added the handler into the correct phase. Now, in order to invoke the version service, type the following in the browser:

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

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 through the incoming counter handler as well as through 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` file and see what happens. Also, we can change the phase rules and see whether it places the handlers to the correct location. In the meantime, we can restart Tomcat, and try to invoke the service, without engaging the Module. Then we will not see any output in the console. This will help us understand that handlers are invoked only when we engage the module.

Advanced module.xml

Here we had a look at a very simple application. But when it comes to a very complex application, we need to have more configurations in the `module.xml` file. In such cases, we might need to have parameters, WS-Policy, and endpoints.

Summary

To conclude, the quality of services is an essential part in today's Web Service world; just having services will not solve the needs of industry. In this chapter, we learned about the need for quality of services and the various techniques available in Axis2 to extend the system functionality. Here, we discussed the module concept, and we wrote a sample module to understand the concept in a very clear manner. One thing we have to keep in mind is that the sample that we have discussed in this chapter is just to get a feel about modules. To learn more, we have to do a few more samples, and get the concepts clear.

9
Client API

A Web Service framework can be used to deploy services as well as to access Web Services. So far, we have discussed deployment. In this chapter, we will focus on the client side. Note that the Axis2 runtime does not have two different concepts called server and client. It uses the same execution chain at the server side as on the client side. We are aware that there are services on the server side. So, in order to have synchronization on the client side, Axis2 creates a dummy service when we use the client API. In this chapter, we will have a look at the various aspects of the client API with the help of examples.

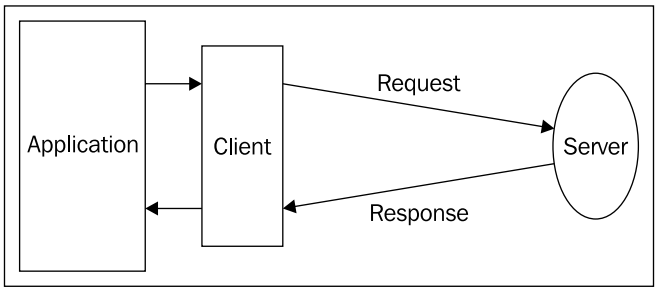
Introduction

When it comes to a client, there are a number of factors that need to be taken into consideration. Among them, the way in which we invoke a service is most important—whether we invoke the service in synchronous manner or asynchronous manner. In simple terms, invocation takes place either in a blocking manner or in a non-blocking manner. Previously, most Web Service frameworks were focused on the blocking invocation pattern, but now the trend is totally different. What users are looking for is an asynchronous or non-blocking way of invoking the service, not only in Web Services, but also in Web-based applications. They use techniques such as AJAX to have asynchronous invocation support. In Axis2, we have two types of asynchronous invocation, while it has support for WSDL 2.0 basic MEPs.

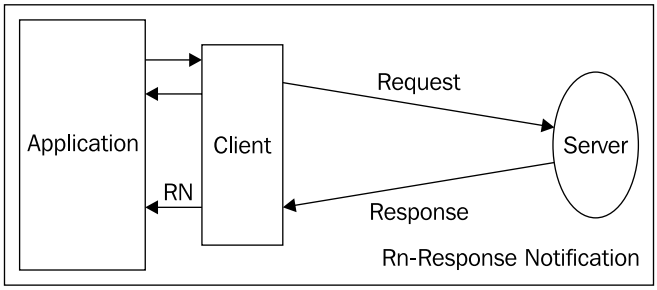
Blocking and Non-Blocking Invocation

As we have just discussed, there are two main ways of utilizing a Web Service, synchronous and asynchronous (blocking and non-blocking). In the case of synchronous invocation, you invoke the service and wait until you get the response. Therefore, in the case of synchronous invocation, the application blocks until you get the response.

The following figure shows how synchronous invocation Web Service utilization takes place:



In case of asynchronous invocation, the user application (e.g. GUI) does not block, so the user can continue working. The Axis2 way of utilizing a service in asynchronous invocation is shown in the following figure:



Inside Axis2 Client API

It should be noted that one of the key designs of the Axis2 Client API is to provide asynchronous Web Service invocation support. Meanwhile, user friendliness is the main consideration of Axis2. So, by combining both, the new client API becomes very convenient to work with. To make the Web consumers' or the end users' job easier, the Axis2 client API consists of two sub-APIs called ServiceClient and OperationClient, the first for average users and the second for advanced users.

ServiceClient API

As we mentioned earlier, ServiceClient is mainly designed for average users, or for Web Services beginners. But nevertheless, it has the notion of interacting with a service. If we take a calculator as an example of a Web Service, then that service would have operations such as "add", "subtract", "multiple" and "divide". So the idea of the service client is to provide an API to invoke any of those operations in a very convenient manner.

Available Ways of Creating a ServiceClient

There are multiple ways to create a `ServiceClient` instance. We have a variety of constructors to use. No matter how we create a `ServiceClient`, we are required to have an Axis2 runtime (`ConfigurationContext`) in order to invoke the service.

We have already discussed `ConfigurationContext`, so we know how to create it. As a result, we can use those techniques to create `ConfigurationContext` while creating a `ServiceClient`, or we can create a `ServiceClient` with `ConfigurationContext` as null, and allow Axis2 to create `ConfigurationContext` for the users.

Type 1: Creating a ServiceClient using Its Default Constructor

The easiest way to create a `ServiceClient` is to use its default constructor. In this case, it creates a `ConfigurationContext` by using the Axis2 default configuration file, which is available in the Axis2 JAR file. At the same time, it creates an anonymous (dummy service) service with three operations (operations to support WSDL 2.0 MEPs). Even though we create a `ServiceClient` in the above manner, we can use the client that has already been created in order to access any Web Service.

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

When we try to create a `ServiceClient` inside an Axis2 system (as an example, a handler tries to create a `ServiceClient` to invoke a service, or one service tries to invoke some other service), then it has to be created by using the `ConfigurationContext` of the server. In this case, all the properties, transports, and modules in the server are accessible to the `ServiceClient`. This particular scenario is referred to 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 a constructor as shown below. As we already know, there are a number of ways to create `ConfigurationContext` as well. At the same time, there are many instances in which we want to create the `ServiceClient` with our own `axisService` that might have been configured with custom QoS (Quality of Service), different parameters, and WS-Policy.

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

In the above case, either of the arguments or both the arguments can be null. If both are null, then it is similar to the case of ServiceClient's default constructor. If `ConfigurationContext` is null, then either a new one will be created by using the Axis2 default configuration, or it will use the `ConfigurationContext` of the server. This will depend on the location in which you are trying to create the `ServiceClient`. Similarly, if `axisService` is null, an anonymous service is created.

Type 3: Creating a Dynamic Client (Client on the Fly)

The idea of a dynamic client is to create a `ServiceClient` on the fly, or to simply create a client for the given WSDL at run time, and use the created `ServiceClient` to invoke the corresponding service. When we create the `ServiceClient` in this manner, a corresponding `axisService` object is configured as per the WSDL document. We will have a look at the advantages of the dynamic client later in this chapter. The constructor for creating a dynamic client is as follows:

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

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

In order to understand these service elements, a sample WSDL is given below:

```
<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 a Working Sample

We are aware of the types that are available to create a ServiceClient, but we have seen no code to explain it. 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 sample service in the server, then, we can write a useful client to invoke the service. We can create and deploy a service by using the following steps:

Step 1: Open an IDE, and write the following Java class or 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);
}
}
```

This sample service implementation class has two methods, one which possess a return value, and the other which does not possess a return value.

Step 2: Write the `services.xml` file for the service. The `services.xml` file will appear as follows:

```
<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>
```

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

After deploying a service in the server, we need to invoke that service. Here, we will look at a number of scenarios to understand the concept clearly.

Scenario 1: Invoking a service in 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 such a way that they have input (s) and output, so we need use the request-response invocation pattern. In Axis2, it can be done in two ways: in a blocking manner or in a non-blocking manner. The first sample demonstrates how to invoke a service in a blocking manner.

Step 1: Create a `ServiceClient` by using any of the constructors that are mentioned earlier.

Step 2: Create an `OMElement` for the payload (as the first child of the SOAP body). In Axis2, XML representation is built on AXIOM, so we need to create `OMElement` (see Chapter 3 to learn about AXIOM).

We can use the following code snippet to create the payload that is required to invoke the service. If you look at the service WSDL carefully, you will 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;
}
```

Step 3: Before invoking the service, we need to create a metadata (property bag) object called `Options` and set that to `ServiceClient`. The object `Options` 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 we can create the `Options` 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 the `ServiceClient` as a dynamic client, then we do not need to worry about creating the `Options` object; it will be created automatically.

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

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

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>
```

`sendReceive` is the API for invoking a service in a blocking manner. Remember that `sc.sendReceive (createPayLoad())` works only if we create `ServiceClient` by either using its default constructor, or passing a null value as `axisService` parameter for any other constructors. When we create the `ServiceClient` by either using our own `axisService`, or 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` by using the WSDL service, then we have to use the operation name as shown in the code below:

```
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"),createP
ayLoad());
```

From the above, we can conclude that it is easy to configure and invoke a service.

Scenario 2: Utilizing a Service in a Non-Blocking Manner (sendReceiveNonBlocking())

In order to invoke an in-out MEP in an asynchronous manner, we can use this API. There are two mechanisms for implementing asynchronous type invocation: callback and pooling. Axis2 uses the callback mechanism to provide asynchronous support. Therefore, in order to use a 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 to Step 3 in Scenario 1, without any modification, but Step 4 needs to be changed 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 callback 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");
}
}
```

The main differences between Scenario 1 and Scenario 2 are as follows:

- The need to create an `AxisCallback` object
- `sendReceiveNonBlocking` is a void operation

Once we run the code sample, we will get the following output:

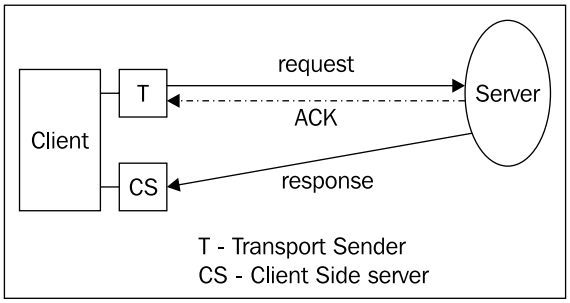
```
-------Invoke the service---------
<ns:echoResponse
xmlns:ns="http://ws.apache.org/axis2/xsd">
    <return>
Hello , my first service utilization
    </return>
</ns:echoResponse>
```

Similar to Scenario 1, 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

With minor modifications, we can use Scenario 1 and Scenario 2 to send the request using one transport and receive via another transport (for example send via HTTP and receive via TCP). It can be graphically represented as shown in the following figure:



In order to invoke a service in the above manner, we need to have WS-Addressing support. We also need to engage the addressing module on both the client side and the server side. In Scenario 2, we only had an application-level asynchronous support, but here we have transport-level asynchronous support as well.

By changing Step 3 in Scenario 2 in the manner given below, we can invoke the service via two transports. Let us send the 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);
```

Here, it should be noted that we can send and receive via HTTP as well. In this case, Axis2 will start up a new HTTP listener in order to receive the incoming message.

Scenario 4: Utilizing an In-Only MEP (FireAndForget())

If we want to send some 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 us try to invoke the "update" operation in MyService.

Step 1: Create a ServiceClient.

Step 2: Create the payload `OMElement`. Then, you will have 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;
}
```

Step 3: When we run the following code, we will see "**value is:10**" in the server's console. Even if something goes wrong with the server, we do not get any response or exception.

```
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 endpoint, and see whether you get any exception; obviously you will get nothing.

Scenario 5: Utilizing an In-Only MEP (sendRobust())

This API also invokes a one-way operation, the only difference being that unlike in Scenario 4, if something goes wrong with the server, the client is informed. We can use the Scenario 4 code with minor changes to invoke the service in a robust manner. Step 1 and Step 2 should remain unchanged, and Step 3 needs to be changed 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 endpoint and see whether you are getting any exception.

Working with OperationClient

We already know that, with a `ServiceClient`, we only have access to the payload on both sending and receiving sides. This will not be enough if we are trying to implement enterprise-level web applications; in such a situation, we need to have more control. Also, we may have to add custom headers into the outgoing SOAP messages as well as access the incoming SOAP process directly. We would be required to access incoming as well as outgoing message contexts. With `ServiceClient`, we can do none of these things (however, we can get the current `OperationContext` once we invoke the service, and from that we can access both request `MessageContext` and response `MessageContext`). The solution is to use OperationClient for both these scenarios. Let us invoke the "echo" operation by using OperationClient for better understanding of the API.

Step 1: Create a `ServiceClient` instance:

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

Step 2: Create an `OperationClient` (we need to pass the fully-qualified operation name similar to the one in the dynamic client case, in order to create an OperationClient):

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

When we create `ServiceClient` by using its default constructor, it creates an anonymous service with three operations, the constant `ServiceClient`. ANON_OUT_IN_OP is one of them.

Step 3: Create a `MessageContext`, and set properties on its option object. Refer to the code given below:

```
//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");
```

Step 4: Create a `SOAPEnvelope`, and add that to `MessageContext`. Here, you need to create a full SOAP envelope:

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

The `creatSOAPEnvelope` method appears as follows:

```
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 the above sample, we have a default `SOAPEnvelope` with a sample payload. But, we can also create a complex `SOAPEnvelope` on the basis of our requirements.

Step 5: Add `MessageContext` to `OperationClient` as shown below:

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

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

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

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

Step 7: Access the response message context, and response SOAPEnvelope:

```
//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 `MessageContext`.

Once we have the `MessageContext`, we can use that to access `SOAPEnvelope`, properties, transport headers, and so on.

When we run this code sample, we should get the following as 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 a means of accessing the incoming and outgoing `MessageContext`. One of the most useful cases is accessing the outgoing `MessageContext` or the `SOAPEnvelope` to add headers. However, the `ServiceClient` API can also be used to add SOAP headers as shown below:

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

Another way of adding headers is also shown below:

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

If we intercept the message using TCP monitor or similar mechanisms, we can see the SOAP headers 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` by using `ServiceClient`. Refer to the code given below.

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

From `OperationContext`, we can either get "In Message Context", or "Out Message Context".

Summary

The Axis2 Client API is very convenient and it has a number of features such as asynchronous Web Service invocations, multiple transport selection, and so on. After the execution of the programs, we understand the basics of the Axis2 Client API; understating the rest of the API requires us to write complex samples.

10
Session Management

Web Services are known to be stateless. However, when it comes to enterprise-level applications, we have to address the issue of sessions. As a result, we cannot fulfill the requirements of enterprise-level applications with a stateless service. While implementing complex applications with Web Services, keeping the history or managing the session becomes a key part of Web Services. In other words, session management becomes an essential part while implementing enterprise-level Web Services applications.

Introduction

As stated above, Web Services are said to be stateless. However, it is difficult to implement complex applications without the support of session management. To understand the need for session support, let us consider a typical bank application. If we consider the following sequence of events associated with a typical banking application, we can get an idea about the need for sessions:

- First, the user starts the transaction by invoking the login method.
- He or she withdraws money (invoking some operation on his or her account).
- He or she completes the transaction by using the log out method.

We can easily understand that the three operations stated above are interrelated, and that the same user makes the above invocations. So it means that someone needs to keep track of the user and user data throughout the invocation of methods. This simply implies the requirement of session management to implement banking applications on Web Services. Of course, there are some alternative ways for implementing this application. But there is still a need to have some other way of identifying and authenticating users.

Stateless Nature of Axis2

As we know, Axis2 architecture is such that it keeps logic and data separately. We have two types of data models: static data and runtime data. We have already discussed them; in this chapter, we will get detailed information about them. The samples in this chapter will help you to understand more about runtime as well as static data.

As far as the Axis2 SOA processing framework is concerned, everything except run-time data can be considered stateless. A stateless nature provides better support for concurrency handling. In Axis2 handlers, MessageReceivers, TransportSenders, TransportReceivers, and the AxisEngine are said to be stateless, so they do not keep any state in their classes. As a result, it does not matter whether we have one instance or a number of instances for the same handlers. Since Axis2 has the notion of the stateless nature of handlers, while writing handlers, we need to write them in such a way that they do not keep any state in them. As an example, we cannot consider the following handler implementation a good approach. We can see that it has a class variable to store the `MessageContext`. So, when we deploy Axis2 in a concurrent environment, we will definitely have issues. As a best practice, we need not to use any class variables in our code.

```
public class InvalidHandler extends AbstractHandler {
    //Class varible 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;
    }
}
```

Well, this does not mean that we cannot maintain state in Axis2. It simply tells us that keeping states in implementation classes is not the right approach. Axis2 supports a better approach to storing and handling sessions using context hierarchy.

Types of Sessions in Axis2

As mentioned earlier, it is very difficult to implement enterprise-level applications using Web Services without having proper session management. On the other hand, it is not mandatory to have session management support in a Web Service framework. We might have services that do not require session management at all. However, Axis2 development team decided to have session management support in Axis2; then it would be helpful for both types of users (developers who need session support and developers who do not need session support).

It is obvious that when we have more functionality and features, it slows down the system, and that is inevitable. So, while considering session management, we come across some issues, especially the need to keep session-related data in memory, which in turn increases the memory footprint. But we know that memory is not a big issue in today's computer industry. Though we know that session management has memory-related issues as well as performance-related issues, we need to have a compromise on whether we support stateful service or not. Web Service frameworks such as Axis2 are for enterprises, so they need to cater to enterprise-level requirements. As a result, Axis2 has the capability to manage sessions.

There are different types of sessions, and the lifetime of the session may vary from one to another. Some sessions last for a few seconds while others last for the lifetime of the whole system. Axis2 architecture has been designed to support four types of sessions, and we observe that there are minor differences between one type and another. By considering the different types of use cases, Axis2 has the following four types of session scopes:

- Request
- SOAPSession
- Application
- Transport

While discussing Axis2 run-time data, we mentioned that we need run-time data or context hierarchy for session management. As a result, we need to have a better understanding of session management.

There are five types of contexts in the hierarchy, which have been listed below with a brief explanation:

- **ConfigurationContext**: This is the run-time representation of whole system. To start an Axis2 system, we need to have configuration context. The lifetime of 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. Then, the run-time representation of that is called ServiceGroupContext.
- **ServiceContext**: This represents the run time of one service. 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 represents the lifetime of an MEP (Message Exchange Pattern). The lifetime of an operation context is, usually 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 to store them is in message context. One OperationContext may have one or more MessageContexts.

Session Creation and Session Destruction

Talking about session management, we are aware that there is a lifetime management associated with it. There is a specific time when a session gets started and a specific time when the session finishes. So, whoever writes a session-aware service would need to know when a session starts, and when it ends. To provide that information, Axis2 uses Java reflection and an optional interface to inform the service implementation class. Actually there are two ways for a service author to get notification about session start and end time, irrespective of the session scope.

Java Reflection

In this case, the service author has to implement the following two methods in the service implementation class, if he or she want to be notified when the session starts and when it finishes. At run time, when a session starts, Axis2 checks whether the following methods are in the service implementation class. If so, it calls the right method.

```
This method will be called when a session starts.
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 very little probability of making mistakes on the methods name and method parameters. So it is good for a user interface. Then, we are not likely to make any mistakes. Axis2 has an interface called `org.apache.axis2.service.Lifecycle`, which has the same two methods that we discussed above. If we want to get a notification when a session starts, we write our 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 this chapter, we discussed that keeping class variables just about anywhere in Axis2 is not a good practice. So it is not a good idea to keep variables inside service implementation classes as well. While managing session, we need to have a way of accessing contexts to get and set session-related data. Though Axis2 passes ServiceContext when the session starts, it is not a good idea to keep that inside the service implementation class. Therefore, we need to have a mechanism for accessing it. We know that once we have `MessageContext`, we can access almost everything using that. So, if we can get access to the current `MessageContext`, then we can consider that we have everything. The next question is how to get the current `MessageContext` inside the service class. Axis2 sets `MessageContext` into ThreadLocal, from where we can access the `MessageContext`. Let's say, we want to access the `MessageContext` inside the method called `foo`, then 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.

Request Session Scope

Request session scope is the default session scope in Axis2. When we deploy a service without knowing anything about session management, then our service will be deployed in request session scope. The lifetime of this session is limited to the method invocation lifetime, or request processing time. When we deploy a service in request scope, it simply means that we are not going to worry about the session management at all. So it is not like session management at all.

When we deploy a service in request session scope, for each and every invocation a new service implementation class is created. Say we have deployed a service called `Foo` in request scope, and a client invokes the service ten times, then there will be ten instances of the service implementation class.

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, to deploy a service in request scope, we need to do the following:

```
<service name="Foo"  scope=" request">
</service>
```

To get an understanding of 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 need to type the following in the browser:

```
http://localhost:8080/axis2/services/MyService/foo?foo=foo
```

Then we will see the following in the server console:

I'm inside init method.

If we continue to do the same, 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 a stateful service . One way is to store the state in the Axis2 global run time (ConfigurationContext), and retrieve it whenever necessary.

SOAP Session Scope

The idea of a SOAP session is to have a transport-independent way of managing session between two SOAP nodes, obviously between the client and the server. Here, Axis2 uses SOAP headers in order to manage the session. SOAP session scope has slightly longer lifetime as compared to a request session scope, and deploying a service in a SOAP session requires changing `services.xml` as well. Managing 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 it wants to access the same session, and the service has to validate the user using session-related data.

In order to manage SOAP session, a client has to send an additional reference parameter in the SOAP header, which is named `serviceGroupId`. The client will receive `serviceGroupId` when a service is invoked and is deployed in a SOAP session for the first time). SOAP session provides a way to manage sessions across not only a single service invocation, but also 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 the `ServiceGroupContext` to store the session-related data.

When we deploy a service in a SOAP session, and a client tries to access the service for the first time, Axis2 will generate a `serviceGroupId` and send that to the client as a reference parameter in `wsa:ReplyTo`, as shown below. However, it should be mentioned that, in order to have SOAP session support, the client as well as the server should 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/
axis2">urn:uuid:65E9C56F702A398A8B11513011677354</axis2:
ServiceGroupId>
            </wsa:ReferenceParameters>
 </wsa:ReplyTo>
```

So if a client wants to live in the same session, then it has to copy that reference parameter and send it back to the server when it invokes the service for the second time. As long as a client sends the valid `serviceGroupId`, it can use the same session, and the service can maintain the session-related data. Unlike a request session, a SOAP session has default time-out period. So if the client does not touch the service for a period of 30 seconds, then the session will expire. If the client then sends the old `serviceGroupId`, it will get an AxisFault. We can change the default time-out period by changing the server's `axis2.xml` as follows:

```
<parameter name="ConfigContextTimeoutInterval">30000</parameter>
```

By changing the parameter value, we can have the time-out interval that we want.

As mentioned earlier, deploying a service in SOAP session requires changing `services.xml` as follows:

```
<service name="MyService" scope=" soapsession">
</service>
```

Though we discussed earlier that we have to copy the reference parameter, it should be noted here that once we use Axis2 client, we can configure it so as to copy that parameter automatically.

Just to get an idea about SOAP session management, let's write the following service class and deploy it. If you look carefully at what the service class does, it adds the current invocation's value to the previous value, and sends the result. If we kept on doing this, we would get incremental values. First, we will write the service and deploy that in the SOAP session, making 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 an addressing module, but 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 below:

```
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 that though we have invoked the service twice, we have got the same output, which simply means that no session has taken place. So now, let's change our client code a bit and see what we get. 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
```

This simply tells us that we have invoked the service in a session-aware manner.

Transport Session Scope

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 session. Then the lifetime of the session is controlled by the transport session and not by Axis2. What Axis2 does is store service context and `ServiceGroupContext` in the transport session object so that the service can access those contexts, as long as the session lives.

One of the key advantages that the Transport Session has 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 transport session requires changing `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. Then, let's try to invoke the service in following ways:

Option 1: Using the browser

```
http://localhost:8000/axis2/services/MyService/add?value=10
```

If we keep on typing this, that then we will get the output as **10**, **20**, **30** etc.

Option 2: Using ServiceClient

When we use `ServiceClient` and set the session management flag to true, it will send the transport cookie back, as well.

```
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());
```

If we run the above code, we will see the following outcome:

```
10
20
```

This simply tells us that we have successfully invoked the service, which has been deployed in the Transport Session in a session-aware manner. In this case, we do not need to have an addressing module.

Application Scope

Application scope has the longest lifetime as 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 application scope, we will find that there is only one instance of the service implementation class. In addition to that, there will be only one ServiceContext for the deployed service. Considering the memory footprint, in the Axis2 world, it is better to deploy the service in application scope, if we don't want to manage the session.

When we deploy a service in application scope, a client does not need to send any additional data to use the same session.

To deploy a service in application scope, we need to change `axis2.xml` as shown below:

```
<service name="foo" scope=" application">
</service>
```

Managing Session Using ServiceClient

As we know by now, managing session in the client side is bit of a work. As mentioned earlier, both in SOAP session and Transport Session, a client has to send the session-related data if it wants to live in the same session. The client can also do that for a SOAP session by copying the required reference parameters. But in Transport Session, how can a user get access to copy and send cookies?

To make life easier, Axis2 has the inbuilt ability to manage sessions in the client session, by just setting a flag (which we have used already). Then, depending on the service-side session, it sends the corresponding data as long as we continue to use the same `ServiceClient`. 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, then we can create `ServiceClient` as shown below and re-use the service client object created to invoke the service.

```
Options options = new Options();
options.setManageSession(true);
ServiceClient sender = new ServiceClient();
sender.setOptions(options);
```

Once we create `ServiceClient` as shown above, and the service is deployed in SOAP session, the `ServiceClient` will copy the `serviceGroupId`, and send that from the second invocation. If the server sends the session ID, like web cookies, it will copy that to `ServiceContext` (on 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, but it is also a limitation for advanced Web Service developers. Developing an enterprise-level application using Web Services is not easy unless we have a session management layer. Axis2 has four types of sessions to address enterprise-level Web Service development issues.

11
Contract First or Code First

When it comes to Web Service development, especially at the server side, there are two main approaches for creating a service. The first approach starts from a contract. In this context, the contract is nothing but WSDL. The second approach starts from code, and makes a service out of that; the POJO approach is a commonly used mechanism in the code first approach. Each approach has its own advantages and disadvantages. When considering the two approaches, some may argue that the code-first approach is the best, while some others may argue that the contract-first approach is the best. When it comes to complex systems, contract approach is the best, while the code-first approach would be the best in the development stage, and for small applications. In the meantime, if we have a legacy system that we are going to convert to a Web Service, then we have to use the code-first approach.

Introduction

In this chapter, we will discuss the above approaches and their use in an appropriate manner. First, we will start with the code-first approach and discuss how to make a Web Service out of our code. We have already discussed how to make a Java class into a Web Service (POJO-based). After the code-first approach, we will be focusing on the contract-first approach and how Axis2 supports that.

Code-First Approach

In this approach, we start the Web Service development from the code. We first design what we are going to do and then just convert our design into code. So when we use this approach, we do not need to know much about Web Service technologies. Most importantly, you need to know neither about WSDL nor about SOAP. So the code-first approach is a powerful tool for developers who want to write a Web Service without having a very good understanding about it. And that helps in bringing new users to the landscape of Web Services.

One of the advantages of using the code-first approach is having fewer restrictions. In simple words, if we have a contract, then we have restrictions over the method signatures, return types, and so on. In this case, we can write code with a high level of freedom. Even changing and adding new functionality is not a big deal. Since we have the code, we can change it appropriately so that it suits the requirements.

In Axis2, we have very good support or equal support for the code-first approach. It includes almost all the components that the code-first approach requires. We know that Web Service communications takes place through SOAP. However, when we write the code, we need not worry about the wire format. The code-first approach class will appear as follows:

```
public Foo getFoo(String foo){
return new Foo();
}
```

In this particular case, there should be a mechanism for converting the message payload (SOAP body) to a String object. We also need to convert the `Foo` object into a response payload. To have better support for the code-first approach, Axis2 has built-in message receivers. These message receivers are based on Java reflection, which knows how to serialize XML to Object and deserialize Object to XML. RPCMessageReceiver is one of the message receivers that does the job for in-out MEP, while the RPCInOnlyMessageReceiver handles in-only MEP. In addition to that, both the message receivers are capable of serializing objects to either a literal wrapped-style document, or to a literal bare-style document. Similarly, it can deserialize a payload to a Java object in either (doc-lit wrapped or doc-lit bare) format. If you do not like the way in which Axis2 does the type mapping then you can change that just by writing a message receiver.

In the case of the code-first approach, the key steps for making a code into Web Service are as follows:

- Firstly, we need to write the code for the Web Service that will be exposed.
- There should be a single Service implementation class. But we can have any number of utility classes and bean classes.
- Then we need to write a custom message receiver if we are not using Axis2 built-in message receivers.
- The next step is to write the `services.xml` file indicating our configurations. And these configurations include a method that needs to be exposed and unexposed with corresponding message receivers and other parameters.
- Compile the project.

- Finally, we need to create a service archive file. Creating a service archive file can be easily done with the tools provided by Axis2 that includes Eclipse as well as IntelJ Idea plugins.
- As the last step, we need to deploy the service.

We know how to create and deploy a service as we have already discussed these things in previous chapters. You can go through Axis2 deployment mechanisms and Axis2 service creation for reference.

The most important thing with the code-first approach is that we start without having a WSDL, or service description document in our hand. But when we deploy the service in Axis2 or when we do `? wsdl` on our service, we get a WSDL document. So if a client tries to access the service, he or she will not be able to guess whether this service was developed by using the code-first approach or the contact-first approach.

Changing or providing new functionality is just a matter of changing the service implementation class and re-deploying the service. When we discussed Axis2 service, we wrote a very basic service. But in order to get a proper understanding of the concept, we need to write a complex service so as to get a feel, and know the power the code-first approach.

The code-first approach can be employed in a situation where we need to write a prototype or when we want to generate a Web Service contract from our code. Writing code and getting WSDL generated by Axis2 is faster than writing a WSDL file. Axis2 has an built-in tool called Java2WSDL to create a WSDL document from a Java class, or you can use IDE plugins as well.

Why Not the Code-First Approach?

Usually while developing Web Services, developers like to code the business logic first and then expose that logic as a Web Service. Developers find this a more convenient approach, since they are already competent in the programming languages they use, without having to learn the intricacies of Web Services. It happens to be more convenient in exposing existing programs as Web Services. However, the code-first approach has the following drawbacks:

- The control that a developer has over exposing the code as a Web Service is less. A change to the code may mean a regeneration of the publicly visible Web Services interface, and often it is difficult to agree on such a generated interface, from a business perspective. The client programs are often generated using the service's WSDL file. If the service WSDL file is likely to change, the point of having generated clients becomes less obvious.

- The code for the service process is likely to change between service frameworks and even framework versions, and it becomes difficult to maintain a single interface across versions.

When it comes to annotation, it is true that by using annotations (JSR 181), the impact of some of these issues can be reduced. Annotations help the developers to take control of the process of exposing code. However, there is no such thing as a generic annotation scheme to make it universally applicable over multiple languages and multiple frameworks.

Contract-First Approach: Why is it So Special?

As opposed to code first, the contract-first approach takes the contract as the primary artifact. The "contract" in a Web Service interaction is the WSDL document. Therefore, in the contract-first approach, the focus is on creating the WSDL file and the associated XML schema. The WSDL file and the schema clearly define the message formats, the operation, the interface names, and other relevant information for a complete Web Service interaction, and can be agreed on by multiple parties. Almost all major Web Service frameworks allow service generation from WSDL, and it becomes easier for the service implementer as well. The reason is that if a major portion of the code is generated, only the necessary business logic will need to be filled in.

We have discussed the problems associated with the code-first approach. But there are some problems with the contract-first approach as well, the most notable one being the need for WSDL and schema expertise. One can argue whether the Web Service implementers would need to do anything with WSDL since the primary requirement of the WSDL is to provide a description of the service rather than to provide the service itself. This would have been a major problem in earlier times, but now, some very good visual tools are available, both free and commercial, allowing easy construction of WSDL. Axis2 also has a set of tools to generate code from a WSDL document.

Code-Generation Support in Axis2

The Apache Axis2 project comes bundled with a convenient code-generator tool. This code generator allows multiple data-binding frameworks to be incorporated, and is easily extensible.

In its simplest form, the code generator is a command-line tool. It also comes in other flavors such as the Eclipse or IDEA plug-in or the custom Ant task. However, these use the same tool to generate code and the options available are the same. All the examples in this chapter use the command-line tool, but the graphical equivalent is easy to figure out and can be used appropriately.

The batch or shell scripts for the code-generator tool are available in the `bin` directory of the Axis2 binary distribution [`axis2-1.3-bin.zip`, 17 MB]. Once the standard Axis2 distribution is downloaded and unzipped, the scripts are ready for action. If you are having problems running the tools, try to follow the installation procedure properly, or you can refer to the Axis2 site. It has a great deal of up-to-date documentation about installing Axis2, seting up the classpath, and more.

Sample 1: Use Default Code-Generation Options to Generate Server-Side Code

To generate server-side code from WSDL, you have to provide the `-ss (--server-side)` and `-sd (--service-descriptor)` flags to the code generator. The '`-ss`' flag indicates whether it is server-side code. But, it is recommended to use the '`-sd`' flag along with the '`-ss`' flag, as the service cannot be deployed without a service descriptor in Axis2. In addition, you can also specify the output location using the '`-o`' option.

Go to the `bin` directory of the Axis2 binary distribution, and run either the batch or the shell file. The WSDL files are available in the `samples/wsdl` directory. The following command generates server-side code with default options:

```
>wsdl2java.bat -uri ..\samples\wsdl\Axis2SampleDocLit.wsdl -o skel
                                                          -ss -sd
```

The above sample is based on the Windows environment. So, you can see the batch file as well as the Windows-specific path.

Then, the code will generate under the `bin/skel` directory, as we have given the output location as `skel`. Three artifacts will be visible in the output location (`bin/skel` in this case) immediately after the code generation:

1. `build.xml`
2. `src` directory
3. `resources` directory

Inside the `src` directory, the source files (service skeleton) are available inside the `org\apache\axis2\userguide\axis2sampledoclit` directory, as it takes the default package when nothing is specified. The most interesting item here is the generated skeleton. This skeleton is meant for the service implementer to fill in and is the only piece of code that needs to be modified in order to have a successful service.

The `resources` directory contains two files: the WSDL file for the service, and the `services.xml` file (depending on the WSDL, there would also be mutiple files such as schema files). These files need to go into the `META-INF` directory of the `aar` file. The skeleton will be named `Axis2SampleDocLitServiceSkeleton` to match the service name. For each operation in WSDL, a corresponding method will be generated in the skeleton class.

The following code snippet shows an implementation of the skeleton for only one method the (generated skeleton has three methods):

```
public class Axis2SampleDocLitServiceSkeleton {
    public org.apache.axis2.userguide.xsd.EchoStringReturn echoString(
                    org.apache.axis2.userguide.xsd.EchoStringParam
echoStringParam) {
        EchoStringReturn response = new EchoStringReturn();
        response.setEchoStringReturn(echoStringParam.
getEchoStringParam());
        return response;
    }

    public org.apache.axis2.userguide.xsd.EchoStringArrayReturn
echoStringArray(
            org.apache.axis2.userguide.xsd.EchoStringArrayParam
echoStringArrayParam) {
        throw new java.lang.UnsupportedOperationException("Please
implement " + this.getClass().getName() + "#echoStringArray");
    }
    public org.apache.axis2.userguide.xsd.EchoStructReturn echoStruct(
                    org.apache.axis2.userguide.xsd.EchoStructParam
echoStructParam) {
        throw new java.lang.UnsupportedOperationException("Please
implement " + this.getClass().getName() + "#echoStruct");
    }
}
```

Once the skeleton is implemented, the recommended way to generate the service archive is to use the Ant build file. While it is certainly possible to do the archive creation manually, the Ant build is very convenient and it is very helpful to developers.

Run the Ant build file by typing 'ant'. Note that, for the Ant build to succeed, you should have the `AXIS2_HOME` environment variable set to point to the Axis2 installation location.

Once we generate the service `aar` file, we can test the service by deploying that to a running Axis2 instance.

Sample 2: Use a Different Databinding

Axis2 allows the use of different data-binding frameworks for code generation. Axis2 has support for a number of data-binding options as well, and for this, we'll try to use XMLBeans. When we do not specify the databinding framework, code will be generated using Axis2 default, which is ADB.

The following command generates server-side code by using XMLbeans for the databinding. Note that the earlier generated artifacts (if they are still there) need to be manually removed, since the code generator does not overwrite existing files:

```
>wsdl2java.bat -uri ..\samples\wsdl\Axis2SampleDocLit.wsdl -o skel -ss
-sd -d xmlbeans
```

The immediately noticeable change among the artifacts is the presence of some more files in the `resources` and `src` directories. In this case, the XMLBeans-specific binary file set (with the extension xsb) is available in the `resources` directory.

The skeleton is very similar to the one in sample 1, except that it has XMLBeans-specific classes in the skeleton and not ADB classes. Filling the skeleton is almost the same as in sample 1.

We can see that the skeletons have a strong resemblance to each other even though the underlying data-binding mechanism has changed completely. All the issues related to databinding are handled under the hood, and the service implementation would not need to worry about anything other than filling in the business logic.

The importance of using the Ant build file for making the `aar` file becomes clear while using XMLBeans. The XMLBeans databound classes require the `.xsb` files to be in the classpath, and the generated Ant build comes with targets that copy these non-class files to the appropriate locations.

Sample 3: Generate an Interface Instead of a Concrete Class

There are instances where the service developers have an interface for the skeleton, and then they name a particular service implementation for the configuration. This can easily be done with Axis2 by using the '`-ssi`' (server-side interface) flag:

```
>wsdl2java.bat -uri ..\samples\wsdl\Axis2SampleDocLit.wsdl -o skel
                                                        -ss -sd -ssi
```

This code causes the emitter to emit an interface in place of a concrete class. Of course, the concrete class is also generated, but it is not referenced inside the message receiver. Instead, the interface is used, and the user is free to place any class that implements that interface as the service class.

If you have a different implementation, then you need to update the following parameter in `services.xml`:

```
<parameter name="ServiceClass">org.apache.axis2.userguide.
axis2sampledoclit.Axis2SampleDocLitServiceSkeleton</parameter>
```

Sample 4: Generating Client-Side Code

When it comes to the code-generation tool compared to server-side code-generation, client-side code-generation has more usage. The main reason is that we have a remote WSDL document, and we need to write client code to invoke the service. In that case, we need to have a tool for generating client-side code or a stub (service proxy). Now, let us try to generate the client-side code for the same WSDL. In this case, we can use the following option (no need to have any of the '`-ss`', '`-sd`' flags in this case):

```
>wsdl2java.bat -uri ..\samples\wsdl\Axis2SampleDocLit.wsdl -o stub
```

When we run the above code, we will find the stub Java class under `src\org\apache\axis2\userguide\axis2sampledoclit` called `Axis2SampleDocLitServiceStub.java`. If you open the stub class, you will find it very difficult to read. You don't have to try to read it; unlike in the case of the skeleton, we are not going to edit the stub. Rather we will use the stub to invoke a remote service. The stub has all the service invocation logic, such as address, SOAP action and so on. Now, let us see how to write client-side code to invoke a service by using the generated stub.

```
Axis2SampleDocLitServiceStub stub = new
Axis2SampleDocLitServiceStub();
        Axis2SampleDocLitServiceStub.EchoStringParam request =new
                    Axis2SampleDocLitServiceStub.EchoStringParam();
        request.setEchoStringParam("Hello");
        Axis2SampleDocLitServiceStub.EchoStringReturn response =
                                            stub.echoString(request);
```

As we have discussed, the '`-d`' option is applicable here as well, to generate the client based on different databinding framework we can use the '`-d`' option with the name of the data-binding framework that we want.

One other interesting tweak to code generation is generating both server-side and client-side code in a single shot. This can be achieved using the '`-g`' flag. The generated code will contain the skeleton as well as the stub and will be useful during code generation.

The Axis2 code-generation tool has a number of configuration options, which are listed below. We can make use of them when we generate code using Axis2.

-o <path>: Specify a directory path for the generated code.

-a: Generate async-style code only (Default: off).

-s: Generate sync-style code only (Default: off). Takes precedence over -a.

-p <pkg1>: Specify a custom package name for the generated code.

-l <language>: Valid languages are java and c (Default: java).

-t: Generate a test case for the generated code.

-ss: Generate server-side code (such as skeletons) (Default: off).

-sd: Generate service descriptor (such as `services.xml`). (Default: off). Valid with -ss.

-d <databinding>: Valid databinding(s) are adb, xmlbeans, jibx, jaxme and jaxbri (Default: adb).

-g: Generates all the classes. Valid only with -ss.

-pn <port_name>: Choose a specific port when there are multiple ports in the WSDL.

-sn <service_name>: Choose a specific service when there are multiple services in the WSDL.

-u: Unpacks the data-binding classes.

-r <path>: Specify a repository against which code is generated.

-ns2p ns1=pkg1,ns2=pkg2: Specify a custom package name for each namespace specified in the WSDL's schema.

-ssi: Generate an interface for the service implementation (Default: off).

-wv <version>: WSDL Version. Valid Options: 2, 2.0, 1.1

-S: Specify a directory path for generated source.

-R: Specify a directory path for generated resources.

-em: Specify an external mapping file.

-f: Flattens the generated files.

-uw: Switch on un-wrapping.

-xsdconfig <file path>: Use XMLBeans `.xsdconfig` file. Valid only with -d xmlbeans.

-ap: Generate code for all ports.

-or: Overwrite the existing classes.

-b: Generate Axis 1.x backward compatible code.

-sp: Suppress namespace prefixes (Optimization that reduces size of SOAP request or response)

-E<key> <value>: Extra configuration options specific to certain databindings.

-Ebindingfile <path> (for jibx): Specify the file path for the binding file.

-Etypesystemname <my_type_system_name> (for xmlbeans): Override the randomly generated type system name.

-Emp <package name> (for ADB): Extension mapper package name.

-Eosv (for ADB): Turn off strict validation.

--noBuildXML: Do not generate the `build.xml` file in the output directory.

--noWSDL: Do not generate WSDL in the resources directory.

--noMessageReceiver: Do not generate a MessageReceiver in the generated sources.

Summary

The code-first approach is very handy for most of the Web Service applications, since it helps to bring code to a Web Service easily. The contract-first approach is a better way when it comes to implementing Web Services. Fortunately, Axis2 has a flexible code generator that supports contract-first development in a very convenient manner.

12
Advanced Topics

In the previous chapters, we discussed how to install Axis2, write a simple Web Service, write a simple module, invoke a Web Service, and so on. In this chapter, we will discuss some more advanced features of Axis2.

REST—Representational State Transfer

REST is a term introduced by Roy Fielding in his Ph.D. thesis to describe an architectural style of networked systems. The motivation behind REST was to capture the characteristics of the Web that in turn made it successful. Subsequently, these characteristics are being used to guide the evolution of the Web. REST is not a standard, but it is an architectural style, which accounts for the fact that you cannot find any specification in W3C. We can understand REST and design our Web Services using its style.

Features of REST

The Web comprises many resources. A resource can be any item that holds some interest. For example, the Boeing Aircraft Corporation defines a resource for its 747 line of aircraft. Clients can access that particular resource with the following URL:

```
http://www.boeing.com/commercial/747family/
```

After clicking the link, a representation of the resource is returned (for example `Boeing747.html`). The representation sets the client application in a state. The result of the client traversing a hyperlink in `Boeing747.html` is its access to another resource. The new representation sets the client application in yet another state. Thus, the client application changes (transitions) state with each resource representation that is accessed. As a result, we have Representational State Transfer.

Here is Roy Fielding's explanation for 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. Web Services such as book-ordering services, search services, online dictionary services, and so on, are REST-based Web Services.

REST Services in Axis2

When it comes to Axis2 REST support, the service author does not need to take any additional steps in order to add REST support to the service. Any service deployed in Axis2 gets REST support by default. Even in an auto-generated WSDL file, you can see the HTTP (REST) binding for both WSDL 1.1 and WSDL 2.0. Here, it should be noted that with WSDL 2.0, you get a number of additional configuration support options as compared to WSDL 1.1.

A service deployed in Axis2 can be accessed through an **HTTP POST** or a **GET** request. However, in the case of **GET,** you have to send the request as URL parameters. As a result of the limitations, you may not be able to invoke a given Web Service using the **HTTP GET** method. As an example, if the Web Service method takes a complex value as its method parameter, then the **HTTP GET** method cannot be used. Therefore, a service similar to the one given here cannot be accessed by using the **HTTP GET** method.

```
public String getName(Man man) {
}
However, a service like the following can be easily accessed using the
HTTP GET method.
public  String getName(String id , int age){
}
```

To invoke the above service, you can send the request in the following manner:

```
http://localhost:8080/axis2/services/ServiceName/getName?id=ID&age=10
```

Unlike in the **HTTP GET** method, there are no limitations in the **HTTP POST** method. You can send the request as an encoded URL, or as a request body (in a POX message). While sending the message as the request body with the **HTTP POST** method, you only send the SOAP body.

You can use any of the given HTTP clients in order to invoke a Web Service that is hosted in Axis2 using the REST manner, that is, by using either the GET method or the POST method. But as we discussed earlier, it depends on the method signature of the Web Service operation, and on which method we use, the GET method or the POST method. You can even use **ServiceClient** to invoke a remote service in the REST manner (with either GET or POST). The following code demonstrates invoking a service in the REST manner using the POST method:

```
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());
```

While invoking the above code, if you send the request via a TCP monitor, you will see that it only sends the request payload (only the SOAP body). As a result, you will get 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, in turn will send the request in URL-encoded format (if it can be URL encoded).

MTOM—Message Transmission Optimization Mechanism

Regardless of the flexibility, interoperability, and global acceptance of XML, there are times when serializing data into XML does not make much sense. Web Service users may require transmission of binary attachments of various types such as images or drawings as XML docs together with a SOAP message. Such data is often available in a particular binary format.

The following two traditional approaches deal with sending the binary data in XML:

1. **By Value**

   Sending binary data by value is achieved by embedding opaque data (after some form of encoding has taken place) 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 that is based only on the XML component of the data. XML supports opaque data as content by

the use of either base64 or hexadecimal text encoding. Unfortunately, both these techniques expand 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 times its 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. Another concern is the processing costs (both real and perceived) of these formats, especially when decoding back into raw binary data.

2. **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 unnecessary expansion of data and waste of processing power. The primary obstacle in using these unparsed entities is their heavy reliance on DTDs, which impedes modularity and 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 similar to sending attachments with an email message. Even though these attachments are related to the message content, they are not contained within 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 a MTOM optimized message is the same as a SOAP with Attachments message that makes it backward compatible with SwA endpoints. The most notable feature of MTOM is the use of the XOP:Include element that is defined in the XML Binary Optimized Packaging (XOP) specification so as to refer to the binary attachments (external unparsed general entities) of the message. By the use of this element, the attached binary content logically becomes inline (by value) with the SOAP document, even if it is attached separately. This merges the two areas by making it possible to work with only one data model. This allows the applications to process and describe the data by just looking at the XML part, making the reliance on DTDs obsolete.
On another note, MTOM has standardized the referencing mechanism of SwA.
Axis2 supports Base64 encoding, SOAP with Attachments, and MTOM.

AXIOM 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 the XML Schema base64Binary datatype. Another is to preserve the infoset in both the sender and the 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 base64-encoded data, and externally attaches raw binary data that is 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 base64-encoded 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 expected to send smaller binary attachments by using base64-encoding (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 by using a base64-encoded string that 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. If MTOM is not enabled, all the optimized binary content nodes will be serialized as Base64 strings. You can also create binary content nodes, which will not be optimized in any case. These will be serialized and sent as Base64 strings. Refer to the following code:

```
//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 to `True` in the `options` object while 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();
```

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 `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 overriden in `services.xml` on the basis of each service and operation.

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

Note that you must restart the server after setting this parameter.

You can write a service similar to the following one in order 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
    }
  }
```

Axis2 ClassLoader Hierarchy

We have already discussed that any service or module deployed in Axis2 gets its own class loader, meaning that any service or module in Axis2 is isolated. Axis2 achieves this by using its own class loader mechanisms. What is the advantage in having different class loaders for different services? Say you have two services that need to use two different versions of some third-party library. In this case, if you include both versions of the third-party library, and try to use the different version classes from those two services using a single class loader, classes will be loaded only from one library. This is how the Java class loader works. To solve this problem, Axis2 has introduced the aforementioned class loader mechanism.

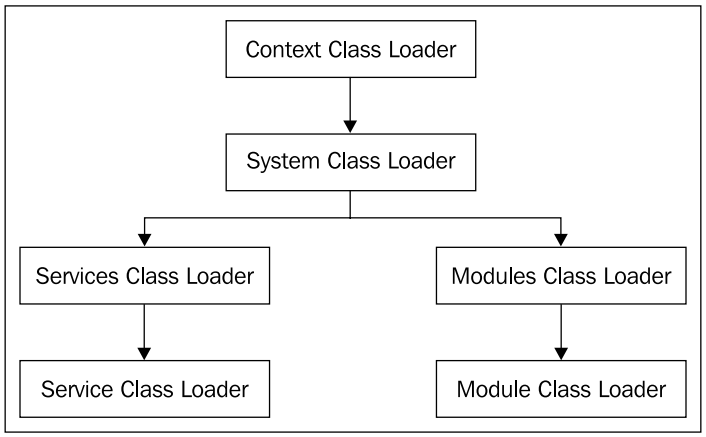
In Axis2, a service class loader or a module class loader is created using its archive file, that is a URL class loader will be created from the archive file. From the created class loader, the service or module will be created, and the created class loader will be stored in the corresponding description (in the case of a service, inside AxisService, and in the case of a module, inside AxisModule).

Sharing Libraries Using the Class Loader Hierarchy

With the class loader hierarchy, you can share classes across services and modules. This is done as follows:

- If you want to share some third-party libraries, then you can put those libraries into the classpath or the `lib` directory of the Axis2 distribution.
- Sharing some third-party libraries across a set of modules, or all of the modules, can be done by putting those libraries into the `repository/modules/lib` directory.
- Sharing some third-party libraries across a set of services, or all of the services, can be done by putting those libraries into the `repository/services/lib` directory.

The following figure demonstrates the Axis2 classloader hierarchy:



According to the above figure, the **Context Class Loader** is the one that Axis2 gets when it is started. Then the **System Class Loader** will be created by adding repository-level libraries. Subsequently, the **Services Class Loader** will be created by adding service-level libraries to the **System Class Loader**. In the same way, a **Service Class Loader** will be created by adding a service archive file to the **Services Class Loader**. Similarly, a **Modules Class Loader** will be created by adding module-level libraries to the **System Class Loader**, and then a **Module Class Loader** will be created by adding the module archive file to the **Modules class loader**.

With this approach, a child class can access any class or property in the parent, whereas a **Module Class Loader** cannot access any class or resource in a **Service Class Loader**.

Axis2 Configurator

So far, you have learned how to start Axis2 and work with Axis2, both on the client side and the server side. But we did not discuss how the underlying logic works. When you start Axis2, it creates an AxisConfiguration object from the local machine that is considered as the repository. In the case of Axis2 WAR distribution, the repository is `<TOMCAT_HOME>webapps/axis2/WEB-INF` (if you are using Tomcat). So, when you start Axis2 in an application server, Axis2 automatically selects the `WEB-INF` directory as the repository. This approach is known as "file system-based AxisConfigurators", in which the Axis2 configuration is created using a file system.

In the same way, you can create an Axis2 system by using a remote location as well, or even by using a database. In the Axis2 distribution, there is built-in support for URL-based and file-based Axis2 configuration creation. The following code demonstrates how to create an Axis2 system by using the file system:

```
ConfigurationContext configCtx = ConfigurationContextFactory.
createConfigurationContextFromFileSystem(
"E:\\urlrepo","");
 SimpleHTTPServer simpleServer = new SimpleHTTPServer
                                                    (configCtx, 8070);
simpleServer.start();
```

The following code demonstrates how to create an Axis2 system by 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 while using a URL repository is that the `services` directory must have a file named `services.list`, which lists all the services' archive files. For example, if you have services named `foo.aar` and `bar.aar`, then the services.list will appear as follows:

```
foo.aar
bar.aar
```

In the same way, the `modules` directory should contain a file named `modules.list` that lists all the modules.

Deploying Axis2 in Various Application Servers

As we had discussed in Chapter 1, 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 case of Apache Tomcat, it is the `webapps` folder, whereas, in 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 place the WAR file.

There are some application servers that will unpack the WAR distribution to a permanent location, or to a temporary location. For example, Apache Tomcat will unpack the WAR file to a permanent location, where changes can be made. When the application server restarts, all those changes will be available. However, there are some application servers that do not do that. There are some application servers that do not unpack the WAR file. Even in Apache Tomcat, you can configure whether you want to unpack or not. Hot deployment and hot update vary depending upon the application server configuration. For example, if the application server is not going to unpack the WAR file, then you do not have hot deployment available.

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 whether the application server will unpack the WAR, and if it does, where it is going to unpack it, and so on. You can even configure the application server-based Axis2 to start with a remote repository as well. Firstly, we will have a look at how `Axis2.war` is to be configured to make it work with a local file system. 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 as well. In such a 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>
```

Summary

In this chapter, we have discussed a number of advanced features of Axis2 along with samples. 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 how to create an Axis2 system using a custom repository, which may be located locally or remotely. Finally, we discussed how to deploy Axis2 in various application servers.

We have covered most of the concepts and features of Axis2. We have discussed how to use Axis2, both as a server and as a client. You need to keep in mind that both Web Services and Axis2 are liable to change rapidly. So, do visit the Axis2 website often to see the changes in the various versions, and changes in Web Services technology.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X