

Programmer to Programmer™

The Apache eXtensible Interaction System
(AXIS) is the most up-to-date, powerful,
and flexible SOAP implementation
for the Java Platform.

Taking over where Apache
SOAP left off, AXIS promises
to become the most
important web services
tool for Java developers.

AXIS

Next Generation Java SOAP

Romin Irani, S. Jeelani Basha

Wrox Technical Support at: support@wrox.com

Updates and Source code at: www.wrox.com

Peer Support and Discussion at: p2p.wrox.com



AXIS: The next Generation Java SOAP by R. Irani and S. J. Basha

In March 2002, Apache released the beta of its latest version of Apache SOAP version 3, codenamed AXIS (**A**pache **eX**tensible **I**nteraction **S**ystem). The significance of Apache SOAP 3 is that it represents a complete ground up re-architecture of previous implementations. The earlier version, 2.2, has become the de facto standard SOAP implementation for Web Services development with Java, presented some important shortcomings that could only be addressed through complete re-engineering.

By applying a modular architecture (through the use of Handlers), AXIS presents a much more flexible framework than earlier incarnations. The added flexibility expresses itself through increased performance on the one hand and the ability to better incorporate any possible changes to the W3C SOAP protocol.

Despite AXIS being in beta at time of publication, its superiority when compared with earlier versions together with the relative stability of the specification has meant that AXIS has already started to be deployed on production systems.

The aim of the present title is to give you a basic understanding of the internals of AXIS and enable you through a series of examples to deploy, in a fast and efficient manner, Java Web Services.

Who is this book for?

Java developers who wish to develop and deploy web services: both in terms of server side services and clients.

What you need to read this book?

- AXIS beta 1
- Server Container such as Tomcat 4.0
- Knowledge of Java

What does this book cover?

Introduction	1
Chapter 1: Introduction to AXIS	5
Chapter 2: Getting Started with AXIS	33
Chapter 3: Architecture	67
Chapter 4: Custom Handlers in AXIS	107
Chapter 5: Advanced AXIS Features	141
Chapter 6: Interoperability	187
Chapter 7: Future Directions for AXIS	203
Chapter 8: Case Study	215
Appendix A: AXIS TCP Monitor	247
Appendix B: J2EE Integration	253
Appendix C: JAX-RPC	267
Appendix D: Support and Code Download	273

AXIS: The next Generation Java SOAP by R. Irani and S. J. Basha

In March 2002, Apache released the beta of its latest version of Apache SOAP version 3, codenamed AXIS (**A**pache **eX**tensible **I**nteraction **S**ystem). The significance of Apache SOAP 3 is that it represents a complete ground up re-architecture of previous implementations. The earlier version, 2.2, has become the de facto standard SOAP implementation for Web Services development with Java, presented some important shortcomings that could only be addressed through complete re-engineering.

By applying a modular architecture (through the use of Handlers), AXIS presents a much more flexible framework than earlier incarnations. The added flexibility expresses itself through increased performance on the one hand and the ability to better incorporate any possible changes to the W3C SOAP protocol.

Despite AXIS being in beta at time of publication, its superiority when compared with earlier versions together with the relative stability of the specification has meant that AXIS has already started to be deployed on production systems.

The aim of the present title is to give you a basic understanding of the internals of AXIS and enable you through a series of examples to deploy, in a fast and efficient manner, Java Web Services.

Who is this book for?

Java developers who wish to develop and deploy web services: both in terms of server side services and clients.

What you need to read this book?

- AXIS beta 1
- Server Container such as Tomcat 4.0
- Knowledge of Java

What does this book cover?

Introduction	1
Chapter 1: Introduction to AXIS	5
Chapter 2: Getting Started with AXIS	33
Chapter 3: Architecture	67
Chapter 4: Custom Handlers in AXIS	107
Chapter 5: Advanced AXIS Features	141
Chapter 6: Interoperability	187
Chapter 7: Future Directions for AXIS	203
Chapter 8: Case Study	215
Appendix A: AXIS TCP Monitor	247
Appendix B: J2EE Integration	253
Appendix C: JAX-RPC	267
Appendix D: Support and Code Download	273

Custom Handlers in AXIS

In the last chapter, we covered the AXIS Architecture and are now in a better position to understand how AXIS works. The focus of this chapter will be to plug-in your custom functionality inside AXIS, in other words extending the default behavior of AXIS. Extensibility is one of the strong features of AXIS over other SOAP Engines. AXIS provides an elegant mechanism for extending its behavior via **Handlers**: objects that are capable of processing messages.

In this chapter, we will:

- ❑ Conduct a quick tour of Handlers: what they are, the various types of Handlers, and how they fit into the AXIS Architecture. We will also cover the classes that already implement some of the Handler functionality in AXIS and how to extend them to write our own custom Handlers.
- ❑ Discuss how to configure our own custom Handlers using the WSDD format.
- ❑ Write our first Request Handler in AXIS that will intercept SOAP Request messages in order to log a number of parameters that we are interested in.
- ❑ Write a couple of additional Handlers: a SOAP Request Handler and a SOAP Response Handler and use them in conjunction to determine the time it takes to make a call to our back-end service.
- ❑ Discuss different mechanisms for handling errors (faults) in our Handlers.

AXIS Handlers

In this section, we discuss in detail what the AXIS Handler and AXIS Chain are and how they fit into the AXIS Architecture. To recap from Chapter 3, a Handler provides a mechanism for the processing of Message(s) contained in the `MessageContext`. Handlers extend the functionality of the AXIS runtime system (`AxisEngine`) by providing additional processing behavior. A **Chain** in the AXIS System is itself a handler that contains a sequence of Handlers to be invoked on specified messages.

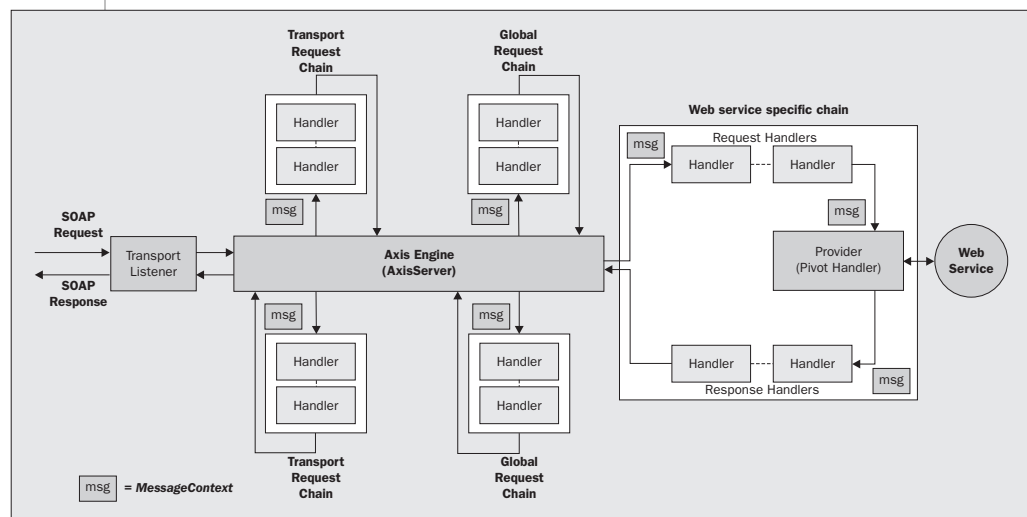
Why do we need to extend the behavior of AXIS via Handlers and Chains? The fact of the matter is that no system can be developed that can meet the needs of one and all. The same applies to AXIS. Take the examples that we have covered in the book so far, where we used the `RPCProvider` to invoke the back-end functionality present in a Java class. The AXIS Engine processes the SOAP Request and passes control to the `RPCProvider`, which invokes our Java class, and returns the SOAP Response.

This Simple Request/Response message processing is provided out of the box by AXIS. However, requirements vary from application to application. For example, what if we wanted to track every request to our Web Service and log that into a file. Suppose that we wish to track a particular SOAP Body Element, for example, the Price, in order to alert Sales Personnel by e-mail should the value exceed 10,000. For both these cases, we would need to intercept the Request Message. Similarly, we may wish to track the SOAP Response and note down the time it took to execute the service method.

The above scenarios stress the fact that it is not possible for the AXIS development team to provide solutions for every eventuality provided by production systems. However, by providing an architecture that allows developers to plug their custom functionality into the SOAP Request/Response, AXIS allows developers to extend its functionality. Your custom functionality can augment the mechanisms that AXIS provides through Handlers and Chains. In addition, many of these services should be provided in a way that is transparent to the service itself. For example, it should be possible to switch on and off both logging and performance metrics, and they should be implemented in a way that is transparent to the service invocation itself, and it should also be possible to otherwise customize the services according to the current business needs.

To best understand where Handlers fit into the AXIS Architecture, we have reproduced the diagram from the previous chapter that shows the flow of a message through the AXIS System from the serverside.

Figure 1



Let's see how the AXIS Engine deals with Handlers at run time. From the above diagram, the Transport Listener packages the SOAP Request into a Message object that is placed inside a MessageContext object. Once this is done, control is handed over to the Axis Engine along with the MessageContext object. The AXIS Engine first determines if there is a **Transport Request Chain**. If it exists, it will be invoked and the MessageContext object will be passed to it. The Transport Request Chain in turn will invoke all the Handlers present in the Transport Request Chain.

After the Transport Request Chain, the Axis Engine will determine if there is a **Global Request Chain** configured. If there is, it will invoke that Chain, which internally will result in the invocation of all the Handlers within it. Finally, the Axis Engine will invoke the **Web Service Request Chain**. The Web Service-specific Chain will contain a special type of Handler called the **Pivot Handler**, which is actually responsible for making the call to the Web Service. A Pivot can be thought of as the point where you actually make the request and get back a response. The `RPCProvider` and `MsgProvider` are Pivot Handlers provided by AXIS for handling of RPC-based Web Services and Message-based (Document) Web Services respectively.

Once the Pivot Handler has invoked the Web Service and obtained the result, the Axis Engine invokes the Chains in the reverse order as shown. That is, first the Web Service-specific Chain followed by the Global Response Chain and Transport Response Chain.

Let us now investigate Chains and Handlers in more detail. As we have seen so far, a Handler is an object that processes a message. A Chain is also a type of a Handler: it is basically a collection of Handlers where each handler is invoked in sequence. To understand how a message gets processed within a Chain, take a look at Figure 2 below.

Figure 2

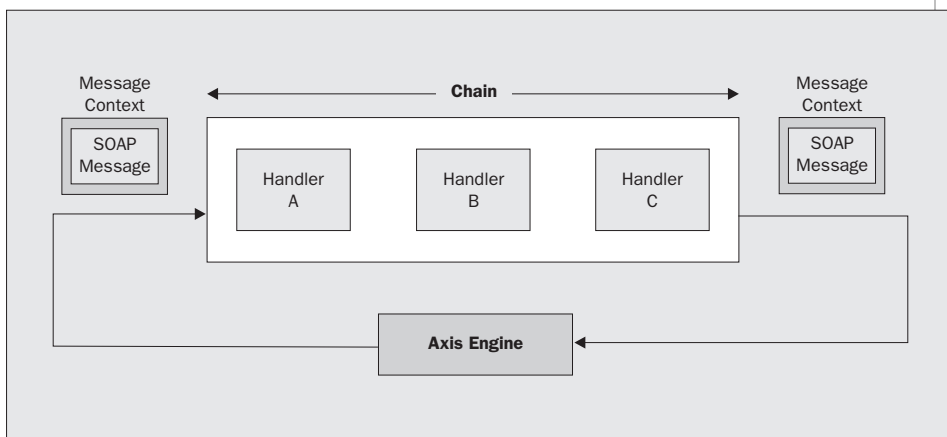
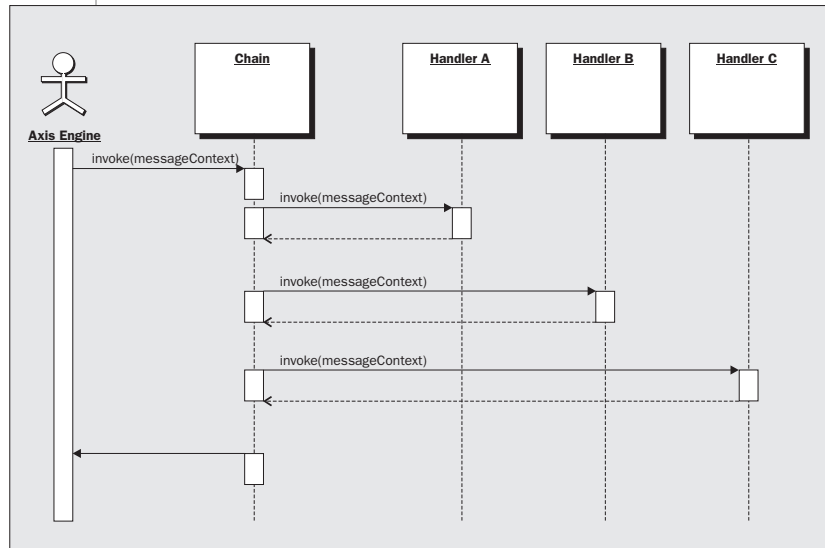


Figure 2 shows the AXIS Engine passing the `MessageContext` to the Chain. The `MessageContext` contains the `Message`, which contains the `SOAP Message`. The Chain in turn will call the `invoke()` method on each Handler, passing the `MessageContext` as a parameter to the `invoke()` method. Each Handler processes the message and returns control back to the Chain. So in the figure above, the Chain first invokes Handler A, passing it the `MessageContext`, Handler A processes the message, for example to log certain attributes of the message, send an e-mail, or perform XSL Transformation. It then returns control to the Chain. The Chain then invokes Handler B and finally Handler C.

The sequence diagram below describes this process visually:

Figure 3



A Handler is thus an intermediary object: it is used to pre-process (in the Request Chain) or post-process (in the Response Chain) the message. In most cases, it will not have anything to do with the final destination – the Web Service. Key design goals for any Handler are that it should seamlessly fit into the Request/Response processing, do its task and then pass on the request to the next element in the Chain – it should not introduce any dependencies into the Web Service itself.

Writing an AXIS Handler

So, how do we go about writing our own handler in AXIS? AXIS defines an interface (`org.apache.axis.Handler`) that defines the methods for a Handler. The Lifecycle methods for a Handler are:

```
❑ public void init()
```

When the AXIS Engine creates an instance of the Handler, the `init()` method is called to allow the Handler to initialize itself.

```
❑ public void invoke(MessageContext messageContext)
    throws org.apache.axis.AxisFault
```

This is the main method where the Handler does its work. From the `messageContext` parameter, the Handler can retrieve the Request or Response Message and process it accordingly. This is really the heart of the Handler.

```
❑ public void onFault(MessageContext messageContext)
    throws org.apache.axis.AxisFault
```

When an error (fault) occurs in the Chain, the AXIS engine will invoke the `onFault()` method of the Handler. In this method, the Handler should accordingly undo the changes as necessary, free resources, etc.

```
❑ public void cleanup()
```

When the instance of the Handler is no longer needed by AXIS, the `cleanup()` method is invoked.

In addition to the lifecycle methods, a Handler also contains a `Hashtable` of options (**name-value** pairs) that you can set and retrieve. The options could be configured during deployment via the WSDO definition of a Handler, which we shall cover in a moment.

In order to define a Handler, we simply implement the `org.apache.axis.Handler` interface. Alternatively, AXIS provides an abstract class, `BasicHandler` (`org.apache.axis.handlers.BasicHandler`), that provides a bare-bones implementation of all of the methods with the exception of the `invoke()` method that implements the specific Handler functionality.

So, writing a custom Handler in AXIS is as simple as extending the `org.apache.axis.handlers.BasicHandler` class and providing an implementation for the `invoke()` method. A skeleton class for a sample `MyHandler` is shown below:

```
import org.apache.axis.AxisFault;
import org.apache.axis.Handler;
import org.apache.axis.MessageContext;

import org.apache.axis.handlers.BasicHandler;

public class MyHandler extends BasicHandler {

    public void invoke(MessageContext messageContext)
        throws AxisFault {
        try {
            //Handler Code here
        }
        catch (Exception e) {
            throw AxisFault.makeFault(e);
        }
    }
}
```

Since the `BasicHandler` provides default implementations for the other Lifecycle methods (`init()`, `onFault()`, and `cleanup()`), we have only to provide the implementation for the `invoke()` method as shown above.

You should now be in a good position to understand how a Handler fits into the AXIS Architecture and what effort it would take to write an AXIS Handler. The kind of functionality that you would want to build into your Handler is obviously application-dependent. But fundamentally, it will involve processing the Request Message or Response Message and invoking some logic on it. Some examples of Handlers are:

- ❑ A **Logging Handler** that logs incoming requests to the Web Service.
- ❑ A **Billing Handler** that tracks the usage of a Web Service and logs the information needed for billing purposes.
- ❑ An **XSLT Transformer** in the Response Chain that will translate the data as per the target device requirements
- ❑ An **Authentication Handler** that will not allow the Request Message to go further if the Authentication fails for the request

But how do we specify to the AXIS Engine what handlers are present, their configuration details, etc.? That is the focus of the next section.

AXIS Handlers – WSDD Details

In this section, we will take a look at how we can configure our Handlers using the WSDD. In the process we will also cover a variety of configurations such as configuring Web Service Request Handlers, Response Handlers, Global Request Handlers, and so on.

WSDD – Looking into the Details

In Chapter 2, we took a first look at writing the WSDD (Web Service Deployment Descriptor). The WSDD as you saw, contained the details of the Web Service to deploy, its classes, the type of web service. (RPC/Message-based), and so on.

In Chapter 3, where we covered the AXIS Architecture, we went into further details about the WSDD. In this section, we will cover only the aspects of the WSDD that are useful for understand the Handler configuration including defining Handlers, Chains, RequestFlow, and ResponseFlow.

Let's first look at how to define a Handler element in WSDD. The syntax of a <handler> element in the WSDD is shown below:

```
<handler name="handlername" type="handlertype">
  <parameter name="param1name" value="param1value"/>
  <parameter name="param2name" value="param2value"/>
  ...
</handler>
```

The <handler> element has two attributes: a **name** for the handler, and the **type** of the handler. A Handler **type** can refer to the Java class that implements the handler, for example, `java:wroaxis.handlers.myhandler`.

The `<handler>` element contains a collection of child `<parameter>` elements, which contain pairs of **name-value** attributes. A `<parameter>` can be used to provide configuration details for the handler. For example, if we have a handler that logs details into a file, we can specify the name of the log file attribute (`"logfile"`) and its value (`"c:\\logfile.txt"`). The `<parameter>` elements are then made available through the `getOption()` method of the `org.apache.axis.handlers.BasicHandler` class.

An example of a Handler configuration is shown below. It defines a Handler called **MyLogHandler**, whose implementation is the Java class `wroxaxis.handlers.MyLogHandler`. The Handler takes a parameter named `logfile`, whose value is `"c:\\logfile.txt"`:

```
<handler name="MyLogHandler"
  type="java:wroxaxis.handlers.MyLogHandler">
  <parameter name="logfile" value="c:\\logfile.txt" />
</handler>
```

Similarly, a **Chain**, which is a sequential collection of one or more Handlers, can be represented as follows:

```
<chain name="MyChain">
  <handler name="handlername" type="handlertype">
    <parameter name="param1name" value="param1value" />
    ...
  </handler>
  ...
  <handler name="handlername" type="handlertype">
    <parameter name="param1name" value="param1value" />
    ...
  </handler>
  ...
</chain>
```

Shown below is an example of a Chain, which consists of two handlers: `MyLogHandler` that we saw above and an `EmailHandler`, which sends an Email to the address specified in the `emailid` parameter:

```
<chain name="MyChain">
  <handler name="MyLogHandler"
    type="java:wroxaxis.handlers.MyLogHandler">
    <parameter name="logfile" value="c:\\logfile.txt" />
  </handler>
  <handler name="MyEmailHandler"
    type="java:wroxaxis.handlers.MyEmailHandler">
    <parameter name="emailid" value="axis@wrox.com" />
  </handler>
</chain>
```

Handlers can be defined separately from a chain, which allows definitions to be reused between chains. The logger handler, for example, is certain to be reused across Web Services, and so should be defined independently of a specific chain. Another way of defining the above sample configuration is as shown below:

```
<handler name="MyLogHandler"
  type="java:wroxaxis.handlers.MyLogHandler">
  <parameter name="logfile" value="c:\\logfile.txt"/>
</handler>
<handler name="MyEmailHandler"
  type="java:wroxaxis.handlers.MyEmailHandler">
  <parameter name="emailid" value="axis@wrox.com"/>
</handler>

<chain name="MyChain">
  <handler type="MyLogHandler"/>
  <handler type="MyEmailHandler"/>
</chain>
```

OK, so now we know how to define a Chain, we can go on to find how we can bind a particular Chain to a Request or Response, and how to define a Fault Processing Chain.

AXIS provides a configurable item in WSDD called a Flow to achieve the above tasks. A Flow defines the sequential invocation of handlers and handler chains for a particular service chain, global chain, or transport chain. AXIS supports three distinct types of Flows: Request flow, Response Flow, and Fault Flow. A Request flow describes inbound Messages. A Response Flow describes the outward flow of a Message, and a Fault Flow represents the fault processing flow to be executed in case of an AxisFault.

Request Flow, Response Flow, and Fault Flow are represented in the WSDD as `<requestFlow>`, `<responseFlow>`, and `<faultFlow>` respectively. Bearing in mind the two ways of defining chains above, the possible formats for a `<requestFlow>` element are shown below. Firstly, a Request Flow can be defined with the handlers defined inline:

```
<requestFlow>
  <handler name="handler1" type="java:...">
    <parameter name="..." value="..."/>
  </handler>
  ...
  <handler name="handler2" type="java:...">
    <parameter name="..." value="..."/>
  </handler>
</requestFlow>
```

Alternatively, the handlers can be defined separately and referenced in the Request Flow definition:

```
<handler name="handler1" type="java:...">
  <parameter name="..." value="..."/>
</handler>
...
<handler name="handler2" type="java:...">
  <parameter name="..." value="..."/>
```

```
</handler>
<requestFlow>
  <handler type="handler1"/>
  <handler type="handler2"/>
</requestFlow>
```

Finally, the Request Flow can refer to an existing defined chain as follows:

```
<requestFlow>
  <chain type="MyChain"/>
</requestFlow>
```

The <responseFlow> and <faultFlow> elements work in the same way. So basically a Flow is an enveloping element around the Chain, which indicates whether the Chain belongs to the **Request Flow**, a **Response Flow**, or a **Fault Flow**.

Any of these flows can then be present in a Transport Chain, Global Chain, and Web Service Chain, each represented by the following elements respectively:

- ❑ <transport>
- ❑ <globalConfiguration>
- ❑ <service>

AXIS Handlers – Sample Configuration

In order to reinforce the material we have covered so far, we will now look at how we can use the <handler>, <requestFlow>, <responseFlow> elements for a particular <service> in the WSDD file. In the example configurations below, we will use the <service> definition for the SparePartPrice service that we covered in Chapter 2. The <service> definition is shown below. The single method for this service, getSparePartPrice is specified, together with the implementing Java class, wroxaxis.chapter2.SparePartPrice:

```
<service name="SparePartPrice" provider="java:RPC">
  <parameter name="allowedMethods" value="getSparePartPrice"/>
  <parameter name="className"
    value="wroxaxis.chapter2.SparePartPrice"/>
</service>
```

A Service Request Handler

We want to track all the requests being made to the SparePartPrice service in a log file so we define a Handler to the SparePartPrice Request processing that logs some of the SOAP Request parameters. For the moment we are not concerned with the logic provided by the Handler implementation class itself, although it is provided in the source code as wroxaxis.handlers.RequestHandler. The log file name is specified by the <parameter> element.

We define the `<handler>` element providing the name of the handler, the Java class that implements the handler, and the log filename.

```
<deployment ...>
  <!--Service Request Handler -->
  <handler name="RequestHandler"
    type="java:wroxaxis.handlers.RequestHandler">
    <parameter name="logfilename" value="c:\\log\\request.log" />
  </handler>
```

We can now define a `<requestFlow>` element in the `SparePartPrice` `<service>` element. The `<requestFlow>` element consists of a sequence of child `<handler>` elements, which in this case points to the `RequestHandler` element defined above.

```
<service name="SparePartPrice" provider="java:RPC">
  <requestFlow>
    <handler type="RequestHandler"/>
  </requestFlow>
  <parameter name="allowedMethods" value="getSparePartPrice"/>
  <parameter name="className"
    value="wroxaxis.chapter2.SparePartPrice"/>
</service>
</deployment>
```

A Service Request and Service Response Handler

It is equally simple to add a Response Handler to the `SparePartPrice` service. All we need to do is define our new `<handler>` element, **ResponseHandler**, as shown below. In this example, an e-mail is sent to an address obtained from the `<parameter>` child element.

```
<deployment ..>
  <!--Service Request Handler -->
  <handler name="RequestHandler"
    type="java:wroxaxis.handlers.RequestHandler">
    <parameter name="logfilename" value="c:\\log\\request.log" />
  </handler>

  <!--Service Response Handler -->
  <handler name="ResponseHandler"
    type="java:wroxaxis.handlers.EmailHandler">
    <parameter name="emailid" value="axis@wrox.com" />
  </handler>
```

We now must add a `<responseFlow>` element to the `<service>` element. The `<responseFlow>` element is similar to the `<requestFlow>` element. It contains a collection of child `<handler>` elements that reference the `<handler>` elements that we defined above:

```
<service name="SparePartPrice" provider="java:RPC">
  <requestFlow>
    <handler type="RequestHandler"/>
  </requestFlow>
  <responseFlow>
```

```

    <handler type="ResponseHandler"/>
  </responseFlow>
  <parameter name="allowedMethods" value="getSparePartPrice"/>
  <parameter name="className"
    value="wroxaxis.chapter2.SparePartPrice"/>
</service>
</deployment>

```

The configuration process is the same for Transport Chains and Global Service Chain. Let us move now to writing and deploying our own Handlers.

SparePartInfo Web Service

The Handlers that we will cover in the next few sections will be written for a sample Web Service that we will develop. To keep our discussion focused on the Handlers, we will keep this Web Service simple. The Web Service is a SparePartInfo Web Service, which works similarly to the SparePartPrice service that we covered in Chapter 2. It has a single method called `getPartInfo()`, which takes a `String` parameter identifying the SKU (Stock Keeping Unit) number of the part. It returns a `String` value containing the part information. To keep our implementation simple, we will be returning a hard-coded string for the part information.

The next few files should be familiar to you now. First, we have the SparePartInfo Web Service shown below:

```

// SparePartInfo.java
package wroxaxis.chapter4;

public class SparePartInfo {

    public SparePartInfo() {
    }

    public String getPartInfo(String PartSKU) throws Exception {
        return PartSKU + " - Part Info";
    }
}

```

Then we have the WSDD file for deploying this Web Service. It is straightforward; consisting of a `<service>` element identifying the SparePartInfo service. It contains `<parameter>` elements that state the classname and the `allowedMethods` of this Web Service.

```

<!-- SPI-deploy.wsdd -->
<deployment name="SparePartInfo"
  xmlns="http://xml.apache.org/axis/wsdd/"
  xmlns:java="http://xml.apache.org/axis/wsdd/providers/java"
  xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance">
  <service name="SparePartInfo" provider="java:RPC">
    <parameter name="className"
      value="wroxaxis.chapter4.SparePartInfo"/>
    <parameter name="allowedMethods" value="getPartInfo"/>
  </service>
</deployment>

```

The Java client program to invoke the SparePartInfo Service follows the same pattern that we have seen for the clients before. You should be able to follow the code easily now.

```
// SparePartInfoServiceClient.java
package wroxaxis.chapter4;

import java.net.URL;

import org.apache.axis.client.Service;
import org.apache.axis.client.Call;
import org.apache.axis.encoding.XMLType;

import javax.xml.rpc.ParameterMode;
import javax.xml.rpc.namespace.QName;

public class SparePartInfoServiceClient {

    public SparePartInfoServiceClient() {}

    public static void main (String args[]) {
        try {

            // EndPoint URL for the SparePartInfo web service
            String endpntURL =
                "http://localhost:8080/axis/services/SparePartInfo";
            // Method Name to invoke for the SparePartInfo web service
            String methodName = "getPartInfo";
            Service service = new Service();
            Call call = (Call) service.createCall();
            call.setTargetEndpointAddress(new java.net.URL(endpntURL));
            call.setOperationName(new
                QName("SparePartInfo",methodName));
            call.addParameter("sku",XMLType.XSD_STRING,
                ParameterMode.PARAM_MODE_IN);
            call.setReturnType(XMLType.XSD_FLOAT);

            //Pass the Part SKU as input parameter to the web service
            Object[] params = new Object[] {"SKU-123"};

            //Invoke the SparePartInfo web service
            String info = (String) call.invoke(params);

            System.out.println("Spare Part Information : " + info);
        }
        catch (Exception e) {
            System.err.println(e.toString());
        }
    }
}
```

Deploying and invoking the SparePartInfo Web Service

To deploy and invoke the SparePartInfo Web Service, follow the steps given below:

1. Create a directory named %AXIS_DEVHOME%\wroxaxis\chapter4. Copy the SparePartInfo.java and SparePartInfoServiceClient.java files to this directory, compile them and then copy the SparePartInfo.class file to the %AXIS_DEPLOYHOME%\WEB-INF\classes\axis\chapter4 directory.
2. Copy SPI-deploy.wsdd to the %AXIS_DEVHOME%\wroxaxis\chapter4 directory. Go to %AXIS_DEVHOME% and deploy the SparePartInfo Web Service by using the AXIS **AdminClient** and SPI-deploy.wsdd file as shown below, making sure that the various JAR files (most of which are in the AXIS lib directory) are available on the classpath:

```
wroxaxis\chapter4> java org.apache.axis.client.AdminClient
-l http://localhost:8080/axis/services/AdminService
wroxaxis\chapter4\SPI-deploy.wsdd
```

3. Go to %AXIS_DEVHOME% directory and run the SparePartInfoServiceClient Java program as shown below. Again, the classpath should include the root of the client's package (in this case the AXIS home folder) and all of the relevant JARS. Verify the output with the output shown below.

```
wrox-axis\chapter4>java
wroxaxis.chapter4.SparePartInfoServiceClient
Spare Part Information : SKU-123 - Part Info
```

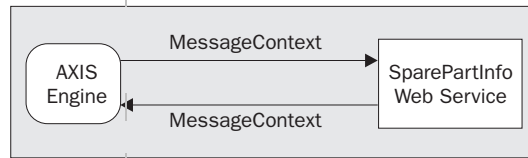
Now that we have our SparePartInfo Web Service working, we will write our custom Handlers that will intercept the **Request/Response flows** to provide additional functionality. Let us move to our first custom Handler, the SOAP Request Logger, whose function it is to log some of the SOAP Request Message information into a log file.

SOAP Request Logger

In this section, we will write an AXIS Handler that will intercept the Request Message for the SparePartInfo Web Service. We will name this AXIS Handler: AxisRequestLogger. This handler will then retrieve some information from the SOAP Request and log that information into a file named Request.log. Readers should be able to expand on this section to provide a much more comprehensive logging mechanism.

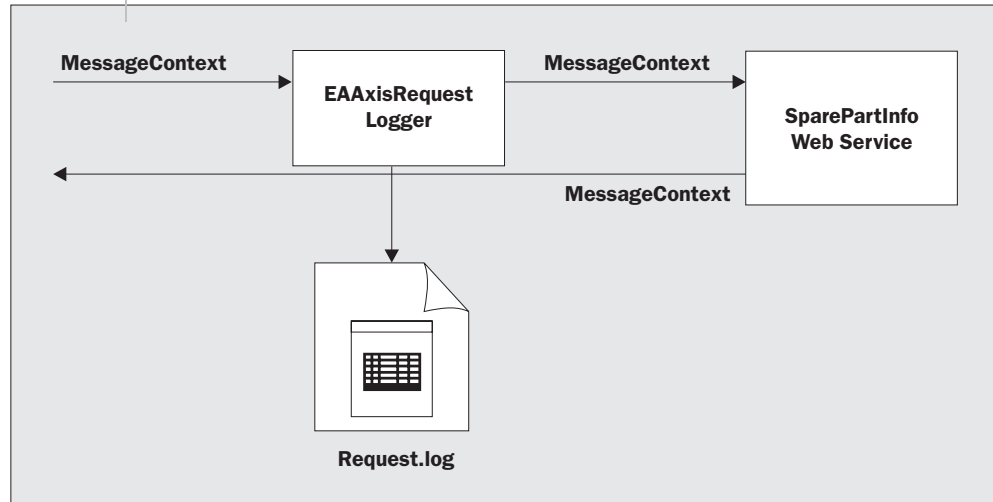
Let us look at the SparePartInfo Web Service conceptually without any handlers:

Figure 4



When we place the `AxisRequestLogger` in the Request stream, it will appear as shown below:

Figure 5



So, what our `AxisRequestLogger` does is intercepts the SOAP Request message, extracts the information that it is interested in logging, and log that information into the `Request.log` file as shown. Let us move on now to writing the code for this.

Writing the Handler – `AxisRequestLogger`

In the introduction to Handlers in this chapter, we discussed the support that AXIS provides for writing your own handlers via pre-built classes like `org.apache.axis.handlers.BasicHandler`. To recap that section, all we need to do to write our Handler is to extend the `org.apache.axis.handlers.BasicHandler` class and provide the implementation for the `invoke()` method that gets invoked by the Axis Engine.

The source code for the `AxisRequestLogger.java` file is shown below:

```
// AxisRequestLogger.java
package wroxaxis.chapter4.handlers;

import org.apache.axis.AxisFault;
import org.apache.axis.Handler;
import org.apache.axis.MessageContext;
import org.apache.axis.handlers.BasicHandler;
```

```
import org.apache.axis.SOAPPart;

import java.io.FileOutputStream;
import java.io.PrintWriter;
import java.util.Date;

public class AxisRequestLogger extends BasicHandler {

    public AxisRequestLogger() {
    }
}
```

The `invoke()` method is the method that we are interested in. The AXIS Engine passes the `MessageContext` as a parameter to the `invoke()` method. The `MessageContext` contains useful information like the `Message` object, the `TargetService`, etc. If any exceptions are caught, you are expected to throw an instance of `org.apache.axis.AxisFault`, which will wrap the SOAP <Fault> element.

```
public void invoke(MessageContext messageContext)
    throws AxisFault {
    try {
```

Then we collect the information from the `MessageContext` element that we are interested in logging. For illustration we will collect three data points of interest, the **TargetService** (`SparePartInfo`), the **TransportName** (HTTP) and the **PastPivot** property (The latter is a boolean value indicating if the Pivot was invoked or not. Since we are in the Request stream, the Pivot is not yet invoked and so it will be always `false`. A Pivot Handler, if you recollect, is a Handler that actually sends a request and gets back a response. Normally it is the point at which your actual back-end service functionality will be invoked).

```
//Collect data from the Request
String targetServiceName = messageContext.getTargetService();
String transportName = messageContext.getTransportName();
boolean getPastPivot = messageContext.getPastPivot();
```

The `getOption()` method shown below is used to retrieve information from the <parameter> elements that have configured for the <handler> element in the WSDO file. In the WSDO file that we will see later, we will see how to configure this <parameter> element having the **name: logfilename** and a **value: C:\\wrox-axis\\Request.log**, which will indicate the file that we want the above information to get logged into. The `getOption()` method takes a `String` identifying the parameter name and returns the parameter value for it.

```
String logfilename = (String) getOption("logfilename");
```

Finally, we use the `java.io.FileOutputStream` object to log the information along with a `Date` stamp.

```
if ((logfilename == null) || (logfilename.equals("")) {
    throw new AxisFault("Server.NoLogFileConfigured",
        "The logfilename parameter option for" +
        " AxisRequestLogger was not set", null, null);
}

FileOutputStream os = new FileOutputStream(logfilename,
    true);
```

```

PrintWriter writer = new PrintWriter(os);
StringBuffer logStr = new StringBuffer();
logStr.append(
    "#####" +
    "\r\n");
logStr.append(
    "##### Request Details #####" +
    "\r\n");
logStr.append(
    "#####" +
    "\r\n");
logStr.append("Request Intercepted at : " +
    new Date().toString() + "\r\n");
logStr.append("Target Service Name : " +
    targetServiceName + "\r\n");
logStr.append("Transport Name : " + transportName + "\r\n");
logStr.append("getPivotPoint property : " + getPastPivot +
    "\r\n");

writer.println(logStr);
writer.close();
}
catch (Exception e) {

```

As mentioned before, any exception that is caught needs to be wrapped into an `org.apache.axis.AxisFault` object. The `AxisFault` class has a utility static method called `makeFault` that takes as an argument a `java.lang.Exception` and returns an `org.apache.axis.AxisFault` instance that can be thrown.

```

    throw AxisFault.makeFault(e);
}
}
}

```

AxisRequestLogger – WSDD details

Since the `AxisRequestLogger` must intercept the Request Flow of the `SparePartInfo` Web Service, we need to add a `<requestFlow>` element to the `<service>` element of the `SparePartInfo` Web Service definition.

First we define our `<handler>` with `name="AxisRequestLogger"` and the `type` attribute indicating the Java class file that has its implementation. Note the `<parameter>` element with `name="logfile"` and the value-`"c:\wrox-axis\Request.log"`. As mentioned before, the `getOption()` method in the `invoke()` method of the source file for the `Logger` will retrieve the parameter and it will then know which file the information needs to be logged into.

```

<!-- SPI-deployWithHandler.wsdd -->
<deployment name="SparePartInfo"
  xmlns="http://xml.apache.org/axis/wsdd/"
  xmlns:java=
    "http://xml.apache.org/axis/wsdd/providers/java">
  xmlns:xsi=
    "http://www.w3.org/2000/10/XMLSchema-instance">
  <!-- Define the Handler i.e. AxisRequestLogger -->
  <handler name="AxisRequestLogger"
    type="java:wroxaxis.chapter4.handlers.AxisRequestLogger">
    <parameter name="logfile"

```

```
value="c:\\wrox-axis\\Request.log" />
</handler>
```

The next part should be familiar to you now. We define a `<requestFlow>` element that references the `<handler>` that we defined above.

```
<!-- Define the Service (SparePartInfo) -->
<service name="SparePartInfo" provider="java:RPC">
  <requestFlow>
    <handler type="AxisRequestLogger" />
  </requestFlow>
  <parameter name="className"
    value="wroxaxis.chapter4.SparePartInfo" />
  <parameter name="allowedMethods" value="getPartInfo" />
</service>
</deployment>
```

Deploying and invoking the SparePartInfo Web Service

Let us run the same `SparePartInfoServiceClient` program and verify that our Handler is able to intercept the request and log the information into the file specified. Follow the steps given below:

1. Copy the `AxisRequestLogger.java` file into the `%AXIS_DEVHOME%\wroxaxis\chapter4\handlers` directory and compile it.
2. Copy the `AxisRequestLogger.class` file to the `%AXIS_DEPLOYHOME%\WEB-INF\classes\wroxaxis\chapter4\handlers` directory.
3. Copy the `SPI-deployWithHandler.wsdd` file into the `%AXIS_DEVHOME%\wroxaxis\chapter4` directory. Go to the `%AXIS_DEVHOME%` directory. Deploy the `SparePartInfo` Web Service by using the `AXIS AdminClient` and the `SPI-deployWithHandler.wsdd` file as shown below:

```
wrox-axis\chapter4> java org.apache.axis.client.AdminClient
-l http://localhost:8080/axis/services/AdminService SPI-
deployWithHandler.wsdd
```

Go to `%AXIS_DEVHOME%` and run the `SparePartInfoServiceClient` Java program; the output should be as before.

4. We are more interested in determining if the information did get logged into `c:\wrox-axis\Request.log`. Navigate to that directory and check if `Request.log` is present. If so, view the file. You should see the SOAP Request information from a couple of sample runs as shown below.

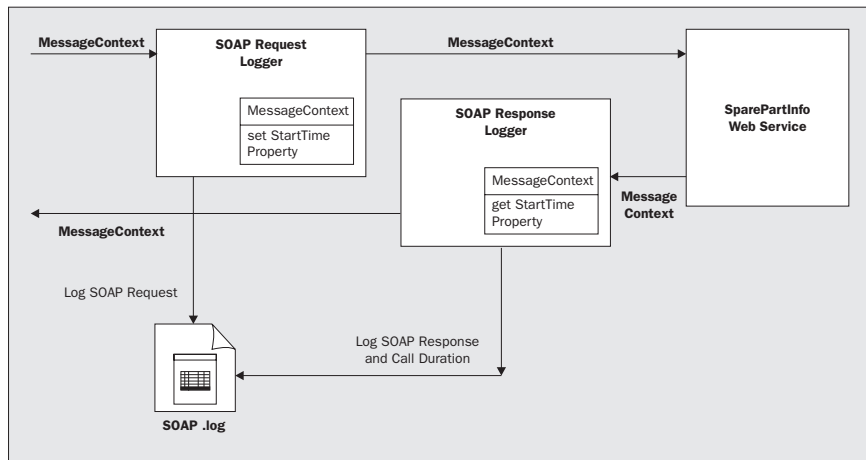
```
#####
##### Request Details #####
#####
Request Intercepted at : Wed Apr 10 15:12:10 BST 2002
Target Service Name : SparePartInfo
Transport Name : http
getPivotPoint property : false
#####
##### Request Details #####
#####
Request Intercepted at : Wed Apr 10 15:14:32 BST 2002
Target Service Name : SparePartInfo
Transport Name : http
getPivotPoint property : false
```

Performance Monitoring Logger

In the previous section, we added a Request Handler. In this section, we will have not only the Request Handler but also a Response Handler for the SparePartInfo Web Service. The Request Handler (`SOAPRequestLogger`) will log the entire SOAP Request into a log file: `SOAP.log`. Then we will use both the handlers in conjunction with each other to determine the time that it took to execute the call to the Web Service. The Response Handler (`SOAPResponseLogger`) will then log that duration along with the SOAP Response into the `SOAP.log` file. So, in this way, we have written our own simple little performance monitoring system.

To recap, look at the diagram shown below:

Figure 6



Let us trace how our Handlers will work in combination to achieve our goal of determining how much time it takes to make the call to SparePartInfo Web Service. As we cover the flow, we will also show the relevant code.

The `invoke()` method of the `SOAPRequestLogger` is called. We use the `getRequestMessage()` on the `MessageContext` object to retrieve the Request Message, which is of type `org.apache.axis.Message`. The Message object has utility methods that allow you to get the `org.apache.axis.message.SOAPEnvelope` object that encapsulates the SOAP Envelope element.

We then call the `getAsDOM()` method on the `org.apache.axis.message.SOAPEnvelope` instance to retrieve the SOAP Envelope as an `org.w3c.dom.Element` object. We use the utility class in AXIS, `org.apache.axis.utils.XMLUtils`, which gives us a convenient method to get a String representation of the SOAP Envelope, which we want to log into a file.

Finally, we use the `java.io.FileOutputStream` object to log the SOAP Envelope string into the `SOAP.log` file that was mentioned as a parameter to the `<handler>` configured in the WSDD. We shall see the `<handler>` definition in the WSDD, a little later in the text, but it follows the same pattern as our `AxisRequestLogger` that we built in the previous section.

```
// SOAPRequestLogger.java
package wroxaxis.chapter4.handlers;

import org.apache.axis.AxisFault;
import org.apache.axis.MessageContext;
import org.apache.axis.Handler;
import org.apache.axis.Message;
import org.apache.axis.SOAPPart;
import org.apache.axis.message.SOAPEnvelope;
import org.apache.axis.handlers.BasicHandler;

import org.w3c.dom.Element;
import org.apache.axis.utils.XMLUtils;

import java.io.FileOutputStream;
import java.io.PrintWriter;
import java.util.Date;

public class SOAPRequestLogger extends BasicHandler {

    public SOAPRequestLogger() {}

    public void invoke(MessageContext messageContext)
        throws AxisFault {
        try {
            Message reqMessage = messageContext.getRequestMessage();
            SOAPEnvelope env = reqMessage.getSOAPEnvelope();
            Element envElement = env.getAsDOM();
            String strSOAPBody = XMLUtils.ElementToString(envElement);
            String logfilename = (String) getOption("logfilename");
            if ((logfilename == null) || (logfilename.equals(""))){
                throw new AxisFault("Server.NoLogFileConfigured",
                    "The logfilename parameter option for" +
                    "SOAPRequestLogger was not set", null,null);
            }
        }
    }
}
```

```

FileOutputStream os = new FileOutputStream(
    logfilename, true);
PrintWriter writer = new PrintWriter(os);
StringBuffer logStr = new StringBuffer();
logStr.append("====SOAP Request : " +
    new Date().toString() + " : =====\r\n");
logStr.append(strSOAPBody);
writer.println(logStr);
writer.close();

```

Now, we have to track the time that we started the call. For this we use the `System.currentTimeMillis()` that returns us the current time in milliseconds. Once we obtain the current time, we use the `setProperty()` method of the `MessageContext` object to set the a custom property called "**StartTime**" and its value (current time). The custom properties that we set in this manner are then present throughout the lifecycle of the entire chain through the `MessageContext` object. Thus as we move along the Request Chain and then into the Response Chain, we can use the `MessageContext` and retrieve the property value using the `getProperty()` method. This is exactly how we will retrieve the value in our Response Handler when we have to calculate the total duration of the method call.

```

//Track the time the call was made
Long startTime = new Long(System.currentTimeMillis());
//Store the startTime in the messageContext
messageContext.setProperty("StartTime", (Long) startTime);
}
catch (Exception e) {
    throw AxisFault.makeFault(e);
}
}
}

```

In the Response Chain, we will first determine the current time and then using the `getProperty()` method of the `MessageContext` object, we will retrieve the **StartTime** of the call. Then we will perform a simple subtraction and represent the duration of the call in seconds.

```

// SOAPResponseLogger.java

package wroxaxis.chapter4.handlers;

import org.apache.axis.AxisFault;
import org.apache.axis.MessageContext;
import org.apache.axis.Handler;
import org.apache.axis.Message;
import org.apache.axis.SOAPPart;
import org.apache.axis.message.SOAPEnvelope;
import org.apache.axis.handlers.BasicHandler;
import org.apache.axis.utils.XMLUtils;

import org.w3c.dom.Element;

import java.io.FileOutputStream;
import java.io.PrintWriter;
import java.util.Date;

public class SOAPResponseLogger extends BasicHandler {
    public void invoke(MessageContext messageContext)

```

```

throws AxisFault {
try {

    //Get the current Time
    long currentTimeMills = System.currentTimeMillis();

    //Retrieve the startTime from the messageContext "StartTime"
    //property that was set by the SOAPRequestLogger
    Long startTime =
        (Long) messageContext.getProperty("StartTime");
    long startTimeMills = startTime.longValue();

    //Determine the duration of the call in seconds
    float duration =
        ((float)(currentTimeMills - startTimeMills))/1000;
    String strDuration = "Time to make call : " +
        String.valueOf(duration) + "seconds";

```

Finally we will log this value along with the SOAP Response into the same log file, SOAP.log. Note that both SOAPRequestHandler and SOAPResponseHandler will be configured with identical child <parameter> elements that will point the parameter name i.e. **logfilename** to the same parameter value "**c:\\wrox-axis\\SOAP.log**".

```

//Reterive the SOAPResponse from the messageContext / Message
Message resMessage = messageContext.getResponseMessage();
SOAPEnvelope env = resMessage.getSOAPEnvelope();
Element envElement = env.getAsDOM();
String strSOAPResponse = XMLUtils.ElementToString(
    envElement);

//Log the SOAPResponse in the same SOAP.log file
String logfilename = (String) getOption("logfilename");
if ((logfilename == null) || (logfilename.equals(""))){
    throw new AxisFault("Server.NoLogFileConfigured",
        "The logfilename parameter option for AccessLogHandler " +
        "was not set", null,null);
}
FileOutputStream os = new FileOutputStream(
    logfilename, true);
PrintWriter writer = new PrintWriter(os);

StringBuffer logStr = new StringBuffer();
logStr.append("==== SOAP Response : " +
    new Date().toString() + " :====\r\n");
logStr.append(strSOAPResponse);
logStr.append("-----");
logStr.append(strDuration);

writer.println(logStr);
writer.close();
}
catch (Exception e) {throw AxisFault.makeFault(e);
}
}
}

```

WSDD File Details for the Loggers

The WSDD for the above scenario should be simple to follow now. We will first define two handlers, SOAPRequestHandler and SOAPResponseHandler. Note that the <parameter> elements for both the handlers point to the same log file, c:\wrox-axis\SOAP.log file. The important part to note is that we are improvising on the same WSDD file that we saw before. In fact, we are not even removing our first Request Handler: AxisRequestLogger. It is perfectly fine to have one or more handlers in the same Request Flow and it also underscores the point that you can add and remove your handlers without affecting the function of the service.

```
<!-- SPI-deployWithSOAPLogger.wsdd -->
<deployment name="SparePartInfo"
  xmlns="http://xml.apache.org/axis/wsdd/"
  xmlns:java="http://xml.apache.org/axis/wsdd/providers/java"
  xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance">

  <!-- Define the Request Handler i.e. AxisRequestLogger -->
  <handler name="AxisRequestLogger"
    type="java:wroxaxis.chapter4.handlers.AxisRequestLogger">
    <parameter name="logfilename"
      value="c:\wrox-axis\Request.log" />
  </handler>

  <!-- Define the Handler i.e. SOAPRequestLogger -->
  <handler name="SOAPRequestLogger"
    type="java:wroxaxis.chapter4.handlers.SOAPRequestLogger">
    <parameter name="logfilename"
      value="c:\wrox-axis\SOAP.log" />
  </handler>

  <!-- Define the Handler i.e. SOAPResponseLogger -->
  <handler name="SOAPResponseLogger"
    type="java:wroxaxis.chapter4.handlers.SOAPResponseLogger">
    <parameter name="logfilename"
      value="c:\wrox-axis\SOAP.log" />
  </handler>

  <!-- Define the Service (SparePartInfo) -->
  <service name="SparePartInfo" provider="java:RPC">
    <requestFlow>
      <handler type="AxisRequestLogger" />
      <handler type="SOAPRequestLogger" />
    </requestFlow>
    <responseFlow>
      <handler type="SOAPResponseLogger" />
    </responseFlow>
    <parameter name="className"
      value="wroxaxis.chapter4.SparePartInfo" />
    <parameter name="allowedMethods" value="getPartInfo" />
  </service>
</deployment>
```

Deploying and Invoking the SparePartInfo Web Service

To invoke the SparePartInfo Web Service with our Performance Monitoring handlers, follow the steps given below:

1. Copy the `SOAPRequestLogger.java` and `SOAPResponseLogger.java` files into the `%AXIS_DEVHOME%\wroxaxis\chapter4\handlers` directory. Compile `SOAPRequestLogger.java` and `SOAPResponseLogger.java`.
2. Copy the `SOAPRequestLogger.class` and `SOAPResponseLogger.class` files to the `%AXIS_DEPLOYHOME%\WEB-INF\classes\wroxaxis\chapter4\handlers` directory.
3. Copy the `SPI-deployWithSOAPLogger.wsdd` file into the `%AXIS_DEVHOME%\wroxaxis\chapter4` directory. Go to `%AXIS_DEVHOME%` and deploy the SparePartInfo Web Service by using the **AXIS AdminClient** and the `SPI-deployWithHandler.wsdd` file as shown below:

```
java org.apache.axis.utils.AdminClient -l
http://localhost:8080/wrox-axis/services/AdminService
wroxaxis/chapter4/SPI-deployWithSOAPLogger.wsdd
```

4. Go to `%AXIS_DEVHOME%` and run the `SparePartInfoServiceClient` Java program. The output will be same as before since there is nothing that we have changed in the SparePartInfo Web Service.
5. Navigate to the `c:\wrox-axis` directory and check if `SOAP.log` file is present in this directory. View the `SOAP.log` file. A sample run is shown below. Note that our `SOAPRequestLogger` logs the entire SOAP Request, whereas the `SOAPResponseLogger` logs the entire SOAP Request and the call duration.

```
====SOAP Request : Wed Apr 10 15:28:18 BST 2002 : =====
<SOAP-ENV:Envelope SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Body>
    <ns1:getPartInfo xmlns:ns1="SparePartInfo">
      <sku xsi:type="xsd:string">SKU-123</sku>
    </ns1:getPartInfo>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
==== SOAP Response : Wed Apr 10 15:28:18 BST 2002 : =====
```

```

<SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Body>
    <ns1:getPartInfoResponse SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:ns1="SparePartInfo">
      <getPartInfoResult xsi:type="xsd:string">SKU-123 - Part Info</getPartInfoResult>
    </ns1:getPartInfoResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

Time to make call : 0.08seconds

Now that we have seen how to write our own custom Handlers in AXIS, it is also appropriate to look at how to handle faults in custom Handlers. It is essential to handle faults correctly in order to make your own Handlers behave like good citizens in the AXIS Architecture.

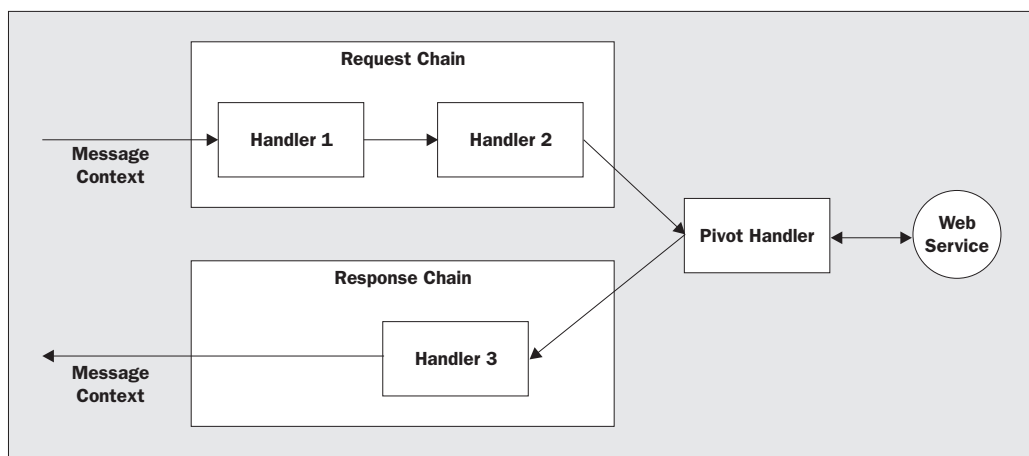
Fault Handling

In this section, we will look at how to handle faults in your AXIS Handlers. We will cover how AXIS handles faults in its processing chain and what steps you as a programmer have to take to ensure that your Handlers intercept the Fault correctly.

Understanding Faults in AXIS

As an example, let us consider the Web Service-specific Chain shown below:

Figure 7



We are keeping the details to a minimum in the diagram. For our Web Service, we have the **Pivot Handler**, which is the `java:RPC Provider` that AXIS provides. Similarly, we have a Request Chain (Handler 1 and Handler 2) and a Response Chain (Handler 3) for the Web service as seen in the diagram.

So far so good, but what would happen if a particular Handler got an exception in its `invoke()` method? We have seen that the `invoke()` method throws an instance of the `org.apache.axis.AxisFault` class that wraps the SOAP Fault element. There could also be a possibility that the Web Service code itself throws an exception. For example, in the `SparePartInfo` Web Service that we covered previously, an invalid part SKU could throw an exception.

In this case, the AXIS Engine will invoke the `onFault()` method of each Handler in the reverse order to the way it called the `invoke()` methods of the Handlers. The `onFault()` method is similar to the `invoke()` method: it has the same parameter (an instance of the `MessageContext`) passed to it. The `onFault()` method provides the Handler a chance to undo any changes that it made in the `invoke()` method that should be reversed in the case of a fault. The Handler may also choose to release allocated resources, in the `onFault()` method.

The `onFault()` method has a default do-nothing implementation in the `org.apache.axis.handlers.BasicHandler` class that we extend. So, in order to handle faults in addition to the `invoke()` method, we must implement the `onFault()` method too:

```
import org.apache.axis.Handler;
import org.apache.axis.handlers.BasicHandler;
import org.apache.axis.MessageContext;
import org.apache.axis.AxisFault;

public class MyHandler extends BasicHandler {

    public void invoke(
        org.apache.axis.MessageContext messageContext)
        throws org.apache.axis.AxisFault
    {
        try {
            //Handler Code here
        }
        catch (Exception e) {
            throw AxisFault.makeFault(e);
        }
    }

    public void onFault(
        org.apache.axis.MessageContext messageContext)
    {
        //Fault Handling Code over here.
    }
}
```

Now that we are clear on how we can handle the faults in our own Handlers, it is still important to understand how the AXIS Engine propagates a fault condition throughout the Chain of Handlers. We mentioned before that if a particular Handler throws a fault, AXIS will call the `onFault()` methods of all the Handlers preceding it, in the reverse order of invocation.

Note that it will not call the `onFault()` method of the Handler that threw the `AxisFault`. It is the responsibility of that Handler to handle the exception.

To help understand this better, let us take a look at some examples. The examples are explained with reference to the diagram shown earlier in this section. It also assumes that all the handlers – **Handler 1**, **Handler 2**, and **Handler 3** – have `onFault()` methods implemented.

1. Exception in Handler 3

In this scenario, let us assume that the `invoke()` method of Handler 3 had some exception and it threw a `AxisFault`. The sequence of `onFault()` methods called will be as follows:
`onFault()` **of Handler 2**
`onFault()` **of Handler 1**

2. Exception in web service

In this scenario, the Pivot Handler will throw an `AxisFault`, which will result in the following `onFault()` methods being called:

`onFault()` **of Handler 2**
`onFault()` **of Handler 1**

3. Exception in Handler 2

In this scenario, the `AxisFault` is thrown from the `invoke()` method of Handler 2. So, the following `onFault()` method will be called:

`onFault()` **of Handler 1**

Now, that we are clear on how fault handling can be done in your own custom handlers and also how AXIS takes care of it, it is time to see that in action.

Faults in Action

In this section, we will look at the first scenario given in the previous section, where Handler 3 throws an exception. We do not want to write a new Web Service, so we will use the SparePartInfo Web Service. However, we will write a new deployment descriptor (WSDD) file for it, so that although we are referencing the same Web Service, we can define a new `<service>` element with its own custom flows. This is another nice AXIS feature to note.

The WSDD file for this is shown below. It defines three handlers: `Handler1`, `Handler2` and `Handler3`. All the handlers have the same `<parameter>` element, a `logfile`, that has a value of `"c:\\wrox-axis\\fault.log"`.

The `<service>` element has a new name – `"FaultService"` – but it references the same class, `SparePartInfo`. It has a `<requestFlow>` that has Handler 1 and Handler 2 in that sequence. In addition, it has a `<responseFlow>` that has Handler 3.

```

<!-- faultdeploy.wsdd -->
<deployment name="FaultService"
  xmlns="http://xml.apache.org/axis/wsdd/"
  xmlns:java="http://xml.apache.org/axis/wsdd/providers/java"
  xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance">

  <handler name="Handler1"
    type="java:wroxaxis.chapter4.handlers.Handler1">
    <parameter name="logfilename"
      value="c:\\wrox-axis\\fault.log"/>
  </handler>
  <handler name="Handler2"
    type="java:wroxaxis.chapter4.handlers.Handler2">
    <parameter name="logfilename"
      value="c:\\wrox-axis\\fault.log"/>
  </handler>
  <handler name="Handler3"
    type="java:wroxaxis.chapter4.handlers.Handler3">
    <parameter name="logfilename"
      value="c:\\wrox-axis\\fault.log"/>
  </handler>

  <!-- Define the Service (FaultService) -->
  <service name="FaultService" provider="java:RPC">
    <requestFlow>
      <handler type="Handler1"/>
      <handler type="Handler2"/>
    </requestFlow>
    <responseFlow>
      <handler type="Handler3"/>
    </responseFlow>
    <parameter name="className"
      value="wroxaxis.chapter4.SparePartInfo"/>
    <parameter name="allowedMethods" value="getPartInfo"/>
  </service>
</deployment>

```

Next are the request handlers: `Handler1.java` and `Handler2.java`. Each of the Handlers implement the `invoke()` method and the `onFault()` method. The methods currently make an entry in the log file if `invoke()` or `onFault()` are called. We could easily include functionality in the `onFault()` method that could for example, send an e-mail to a particular contact person reporting that some error occurred, etc. The handler code should be straightforward at this stage to follow. The important point to note is that we are simulating an exception on purpose in the Request Handler. If you go through the code for Handler 3, you will find that we throw an exception in the code. This section of code has been highlighted.

Request Handler – Handler1

```

// Handler1.java
package wroxaxis.chapter4.handlers;

import org.apache.axis.AxisFault;
import org.apache.axis.Handler;
import org.apache.axis.MessageContext;
import org.apache.axis.handlers.BasicHandler;

import org.apache.axis.SOAPPart;

import java.io.FileOutputStream;
import java.io.PrintWriter;
import java.util.Date;

public class Handler1 extends BasicHandler {

```

```

public Handler1() {
}

public void invoke(
    org.apache.axis.MessageContext messageContext)
    throws org.apache.axis.AxisFault {
    try {
        String logfilename = (String) getOption("logfilename");
        if ((logfilename == null) || (logfilename.equals(""))) {
            throw new AxisFault("Server.NoLogFileConfigured",
                "The logfilename parameter option for Handler1 was" +
                " not set",null,null);
        }
        StringBuffer logStr = new StringBuffer();
        logStr.append("----Handler 1 called----" + "\r\n");
        logMessage(logfilename,logStr.toString());
    }
    catch (Exception e) {
        throw AxisFault.makeFault(e);
    }
}

public void onFault(MessageContext messageContext) {
    try {
        String logfilename = (String) getOption("logfilename");
        if ((logfilename == null) || (logfilename.equals(""))) {
            throw new AxisFault("Server.NoLogFileConfigured",
                "The logfilename parameter option for Handler1 was" +
                "not set",null,null);
        }

        StringBuffer logStr = new StringBuffer();
        logStr.append("Fault in Handler 1 called" + "\r\n");
        logMessage(logfilename,logStr.toString());
    }
    catch (Exception e) {}
}

private void logMessage(String logfileName,String logStr)
    throws Exception {
    try {
        FileOutputStream os =
            new FileOutputStream(logfileName, true);
        PrintWriter writer = new PrintWriter(os);
        writer.println(logStr);
        writer.close();
    }
    catch (Exception e) {
        throw new Exception(e.getMessage());
    }
}
}

```

Request Handler – Handler2

To avoid repeating the same code unnecessarily for Handler2 and Handler3 we'll only show those parts which are different from Handler1, basically the `invoke()` and `onFault()` methods.

```

// Handler2.java
//imports as for Handler1

public Handler2() {
}

public void invoke(
    org.apache.axis.MessageContext messageContext)
    throws org.apache.axis.AxisFault
{
    try {
        String logfilename = (String) getOption("logfilename");
        if ((logfilename == null) || (logfilename.equals("")) {
            throw new AxisFault("Server.NoLogFileConfigured",
                "The logfilename parameter option for Handler2 was not" +
                "set", null,null);
        }
        StringBuffer logStr = new StringBuffer();
        logStr.append("----Handler 2 called----" + "\r\n");
        logMessage(logfilename,logStr.toString());
    }
    catch (Exception e) {
        throw AxisFault.makeFault(e);
    }
}

public void onFault(MessageContext msgContext) {
    try {
        String logfilename = (String) getOption("logfilename");
        if ((logfilename == null) || (logfilename.equals("")) {
            throw new AxisFault("Server.NoLogFileConfigured",
                "The logfilename parameter option for Handler2 was not" +
                " set", null,null);
        }

        StringBuffer logStr = new StringBuffer();
        logStr.append("Fault in Handler 2 called" + "\r\n");
        logMessage(logfilename,logStr.toString());
    }
    catch (Exception e) {}
}
...

```

Response Handler – Handler3

```

// Handler3.java
...

public class Handler3 extends BasicHandler {

    public Handler3() {
    }

    public void invoke(
        org.apache.axis.MessageContext messageContext)
        throws org.apache.axis.AxisFault {
        try {
            String logfilename = (String) getOption("logfilename");
            if ((logfilename == null) || (logfilename.equals("")) {
                throw new AxisFault("Server.NoLogFileConfigured",

```

```

        "The logfilename parameter option for Handler3 was not" +
        " set", null,null);
    }
    StringBuffer logStr = new StringBuffer();
    logStr.append("----Handler 3 called----" + "\r\n");
    logMessage(logfilename,logStr.toString());
    throw AxisFault.makeFault(new Exception(
                "Exception here"));
    }
    catch (Exception e) {
        throw AxisFault.makeFault(e);
    }
}

public void onFault(MessageContext msgContext) {

    try {
        String logfilename = (String) getOption("logfilename");
        if ((logfilename == null) || (logfilename.equals(""))) {
            throw new AxisFault("Server.NoLogFileConfigured",
                "The logfilename parameter option for Handler3 was not" +
                " set",null,null);
        }
        StringBuffer logStr = new StringBuffer();
        logStr.append("Fault in Handler 3 called" + "\r\n");
        logMessage(logfilename, logStr.toString());
    }
    catch (Exception e) {}
}
...

```

We also write a Java client to invoke the FaultService. It follows the same pattern that we have been explaining so far.

FaultService Client

FaultServiceClient.java is shown below. It is in many ways the same as SparePartInfoServiceClient. The only difference is the service name, FaultService. The statements that are different from SparePartInfoServiceClient are shown below:

```

package wroxaxis.chapter4;

import java.net.URL;
import org.apache.axis.client.Service;
import org.apache.axis.client.Call;
import org.apache.axis.encoding.XMLType;

import javax.xml.rpc.ParameterMode;
import javax.xml.rpc.namespace.QName;

public class FaultServiceClient {

    public FaultServiceClient() {}

    public static void main (String args[]) {
        try {

            // EndPoint URL for the FaultService Web Service
            String endpointURL =
                "http://localhost:8080/axis/services/FaultService";

```

```

// Method Name to invoke for the SparePartInfo Web Service
String methodName = "getPartInfo";

// Create the Service call
Service service = new Service();
Call call = (Call) service.createCall();
call.setTargetEndpointAddress(new
                               java.net.URL(endpointURL));
call.setOperationName(new QName("FaultService",methodName));
call.addParameter("sku",XMLType.XSD_STRING,
                  ParameterMode.PARAM_MODE_IN);
call.setReturnType(XMLType.XSD_FLOAT);

//Setup the Part SKU to be passed as input parameter to the
//SparePartInfo Web Service
Object[] params = new Object[] {"SKU-123"};

//Invoke the SparePartInfo Web Service
String info = (String) call.invoke(params);

//Print out the result
System.out.println("Spare Part Information : " + info);
}
catch (Exception e) {
    System.err.println(e.toString());
}
}
}
}

```

Deploying and Invoking the FaultService

1. Copy the Handler1.java, Handler2.java, and Handler3.java files into the %AXIS_DEVHOME%\wroxaxis\chapter4\handlers directory. Compile Handler1.java, Handler2.java and Handler3.java
2. Copy the Handler1.class, Handler2.class the and Handler3.class files to the %AXIS_DEPLOYHOME%\WEB-INF\classes\wroxaxis\chapter4\handlers directory.
3. Copy the faultdeploy.wsdd file into the %AXIS_DEVHOME%\wroxaxis\chapter4 directory. Go to %AXIS_DEVHOME% and deploy the FaultService Web Service by using the AXIS **AdminClient** and faultdeploy.wsdd file as shown below:

```

java org.apache.axis.utils.AdminClient -l
http://localhost:8080/axis/services/AdminService
wroxaxis/chapter4/faultdeploy.wsdd

```

4. Copy FaultServiceClient.java to the %AXIS_DEVHOME%\wroxaxis\chapter4 directory. Go to the %AXIS_DEVHOME% directory and compile the file as shown below:

```

javac wroxaxis/chapter4/FaultServiceClient.java

```
5. Finally, run the FaultServiceClient Java program as shown below. Verify the output with the output shown below. This is the exception that was thrown by Handler 3 and it has been propagated to the client correctly.

```

java wroxaxis.chapter4.FaultServiceClient
java.lang.Exception: Exception here

```

The next point we should check is to view the `fault.log` file in the `c:\wroxaxis` directory. The file contents for the sample run above are shown below. You will note that the `invoke()` method were of the Handlers called first. So the process was **Handler 1 -> Handler 2 -> Handler 3**. Then there was an exception thrown in Handler 3. This did not result in a call to `onFault()` of Handler 3 since we had mentioned that the `AxisEngine` does not invoke the `onFault()` method of the handler that threw the fault. The `onFault()` methods of the preceding handlers are called in the reverse fashion – **Handler 2 -> Handler 1**.

--Handler 1 called--

--Handler 2 called--

--Handler 3 called--

Fault in Handler 2 called

Fault in Handler 1 called

Conclusion

In this chapter we covered how to extend the functionality of AXIS by providing our own custom Handlers. We covered the basics of writing Custom Handlers and looked at several different configurations of Handlers within the context of the AXIS Architecture.

Handlers can be written to handle the Request Flow, Response Flow, and Fault Flow for the Transport Chain, Global Chain, or Web Service Chain. Handlers are a powerful mechanism available within AXIS to help introduce incremental functionality into it in a non-intrusive manner.



Custom Handlers in AXIS

4

Custom Handlers in AXIS